ULTRIX-32™

digital

Supplementary Documents
Volume 3 System Manager

Order Number: AA-MF08A-TE

ULTRIX-32 Supplementary Documents
System Manager

Order No. AA-MF08A-TE

ULTRIX-32 Operating System, Version 3.0

Digital Equipment Corporation

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. Digital Equipment Corporation acknowledges the following individuals and institutions for their role in its development:

"The UNIX Time-Sharing System": Copyright © 1974, Association for Computing Machinery, Inc. reprinted by permission. This is a revised version of an article that appeared in Communications of the ACM, 17, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presnted at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973. Acknowledgements: for their help and support, R.H. Canaday, R. Morris, M.D. McIlroy, and J.F. Ossanna.

"Advanced Editing on UNIX" acknowledgement: Ted Dolotta for his ideas and assistance.

"An Introduction to the UNIX Shell" acknowledgements: Dennis Ritchie, John Mashey and Joe Maranzano for their help and support.

"LEARN - Computer-Aided Instruction on UNIX" acknowledgements: for their help and support, M.E. Bittrich, J.L. Blue, S.I. Feldman, P.A. Fox, M.J. McAlpin, E.Z. Rothkopf, Don Jackowski, and Tom Plum.

"A System for Typesetting Mathematics" acknowledgements: J.F. Ossanna, A.V. Aho, and S.C. Johnson, for their ideas and assistance.

"A TROFF Tutorial" acknowledgements: J. F. Ossanna, Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman, for their help and support.

The document "The C Programming Language - Reference Manual" is reprinted, with minor changes, from "The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978.

"Make - A Program for Maintaining Computer Programs" ackowledgements: S.C. Johnson, and H. Gajewska, for their ideas and assistance.

"YACC: Yet Another Compiler-Compiler" acknowledgements: B.W. Kernighan, P.J. Plauger, S.I. Feldman, C. Imagna, M.E. Lesk, A. Snyder, C.B. Haley, D.M. Ritchie, M.O. Harris and Al Aho, for their ideas and assistance.

"Lex - A Lexical Analyzer Generator" acknowledgements: S.C. Johnson, A.V. Aho, and Eric Schmidt, for their help as originators of much of Lex, as well as debuggers of it.

The document "RATFOR - A Preprocessor for a Rational Fortran" is a revised and expanded version of the one published in Software - Practice and Experience, October 1975. The Ratfor described here is the one in use on UNIX and GCOS at A T & T Bell Laboratories. Acknowledgements: Dennis Ritchie, and Stuart Feldman, for their ideas and assistance.

"The M4 Macro Processor" acknowledgements: Rick Becker, John Chambers, Doug McIlroy, and Jim Weythman, for the help and support.

"BC - An Arbitrary Precision Desk-Calculator Language" acknowledgement: The compiler is written in YACC; its original version was written by S.C. Johnson.

"A Dial-Up Network of UNIX TM Systems" acknowledgements: G.L. Chesson, A.S. Cohen, J. Lions, and P.F. Long, for their suggestions and assistance.

 This manual reflects system enhancements made at Berkeley and sponsored in part by NSF Grants MCS-7807291, MCS-8005144, and MCS-74-07644-A04; DOE Contract DE-AT03-76SF00034 and Project Agreement DE-AS03-79ER10358; and by Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4031, monitored by Naval Electronics Systems Command under Contract No. N00039-80-K-0649.

"Ex Reference Manual" acknowledgements: Chuck Haley contributed greatly to the early development of ex. Bruce Englar encouraged the redesign which led to ex version 1. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and UNIX systems.

"A Guide to the Dungeons of Doom" acknowledgements: Rogue was originally conceived by Glenn Wichman and Michael Toy. Ken Arnold and Michael Toy then smoothed out the user interface, and added many new features. We would like to thank Bob Arnold, Michelle Busch, Andy Hatcher, Kipp Hickman, Mark Horton, Daniel Jensen, Bill Joy, Joe Kalash, Steve Maurer, Marty McNary, Jan Miller, and Scott Nelson for their ideas and assistance.

 Work reported herein was supported in part by the U.S. Department of Energy, Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358, and the National Science Foundation under Grant No. MCS 7807291.

"The FRANZ LISP Manual" acknowledgements: Richard Fateman, Mike Curry, John Breedlove, Jeff Levinsky, Bill Rowan, Tom London, Keith Sklower, Kipp Hickman, Charles Koester, Mitch Marcus, Don Cohen, John Foderaro, and Kevin Layer.

 The financial support of the first and second authors' work by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS80-05144, and the first author's work by an IBM Graduate Fellowship are gratefully acknowledged.

"Introduction to the f77 I/O Library" acknowledgement: Peter J. Weinberger originally wrote the I/O/ Library at A T & T Bell Laboratories.

"Writing Papers with NROFF Using -ME", and "-ME Reference Manual" acknowledgements: Bob Epstein, Bill Joy, Larry Rowe, Ricki Blau, Pamela Humphrey, and Jim Joyce, for their ideas and assistance.

"Refer - A Bibliography System" acknowledgements: Mike Lesk of A T & T Bell Laboratories wrote the original refer software, including the indexing programs. Al Stanberger of the Forestry Department wrote the first version of addbib, then called bibin. Greg Shenaut of the Linguistics Department wrote the original versions of sortbib and roffbib.

"Screen Updating and Cursor Movement Optimization: A Library Package" acknowledgements: For their help and support, Bill Joy, Doug Merritt, Kurt Shoens, Ken Abrams, Alan Char, Mark Horton, and Joe Kalash.

"Disc Quotas in a UNIX Environment" acknowledgements: Sam Leffler and Kirk McKusick, for their

work on the quota code. The current disc quota system is loosely based on a very early scheme implemented at the University of New South Wales and Syndey University.

The document, "Fsck - The UNIX File System Check Program", is a revision by Marshall Kirk McKusick; T.J. Kowalski wrote the original paper. For their help and support, we thank Bill Joy, Sam Leffler, Robert Elz, Dennis Ritchie, Robert Henry, Larry A. Wehr, and Rick B. Brandt. Our sponsors were the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

"A Fast File System for UNIX" acknowledgements: William N. Joy, Samuel J. Leffler, Robert S. Fabry, Marshall Kirk McKusick, Robert Elz, Michael Powell, Peter Kessler, Rober Henry, and Dennis Ritchie. This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

"4.2BSD Networking Implementation Notes" acknowledgements: The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79]. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

"SENDMAIL - An Internetwork Mail Router" acknowledgements: For their ideas and assistance, Kurt Shoens, Bill Joy, Mark Horton, Erick Schmidt, Kirk McKusick, Marvin Solomon, Mike Stonebraker, and Bob Epstein. A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

BEFORE YOU START

This is the third volume of *ULTRIX Supplementary Documents*, a three volume set that contains articles describing the ULTRIX-32 system. The authors are computer scientists and program developers at Bell Laboratories and the University of California at Berkeley. The articles explain the software tools and utilities available on your ULTRIX-32 system. They constitute most of the lore that enriches this operating system; topics range from getting started to the details of screen updating and cursor movement facilities.

Each volume in this set contains several parts, and each part begins with an introduction. The introduction to each part serves as a map that will help you find your way around in the documentation, allowing you to select articles that relate to your interest. Each introduction gives an overview of the material covered in the part and a description of the articles included. Most readers will not need to read all articles, since many articles cover parallel topics.

These articles provide authoritative and accurate information that is unavailable elsewhere. However, you should be aware that some of the information in some articles is dated. We include those articles because many of the concepts they develop are still current and important.

At the end of each volume in this set, you will find a master index identifying topics in all three volumes.

Topics in Volume III

The articles in this third volume are written for people responsible for the installation, administration, and daily maintenance of the ULTRIX-32 system. "Bug Fixes and Changes in 4.2BSD," in Part 1, lists changes in directories, libraries, and utilities between the 4.1BSD software and ULTRIX-32 (based on 4.2BSD).

"A Fast File System for UNIX," by McKusick, Joy, Leffler, and Fabry, compares the new file system used in ULTRIX-32 with the original UNIX file system. The new system is faster and more reliable, and the block size is adjustable. The article also explains considerations and procedures that will help you take full advantage of these improvements.

The articles in Part 2, Maintenance and Administration, deal with disk quotas, fixing corrupted file systems, and management of the *sendmail* utility. The *quota* utility enables the system manager to limit the number of blocks and the number of files available to each user. *Fsck*, the file system check program, lets you examine the integrity of the file system and repair any inconsistencies. The *sendmail* utility lets users send messages between computer systems that are connected to different networks.

Articles in Part 3, Communication, explain the interprocess communication software. Articles in Part 4, Security Considerations, offer a variety of tips on how you can protect your system against crashes and unauthorized access. And Part 5, Supporting Documents, provides information on software changes new to this release.

FSCK - THE UNIX FILE SYSTEM CHECK PROGRAM

SENDMAIL INSTALLATION AND OPERATION GUIDE

SENDMAIL INSTALLATION AND OPERATION GUIDE *(continued)*

PART 3: COMMUNICATIONS

A 4.2BSD INTERPROCESS COMMUNICATION PRIMER

4.2BSD NETWORKING IMPLEMENTATION NOTES

4.2BSD NETWORKING IMPLEMENTATION NOTES *(continued)*

SENDMAIL: AN INTERNETWORK MAIL ROUTER

PART 4: SECURITY CONSIDERATIONS

ON THE SECURITY OF UNIX

PASSWORD SECURITY: A CASE HISTORY

PART 5: SUPPORTING DOCUMENTS

CHANGES TO THE KERNEL IN 4.2BSD

INSTALLING AND OPERATING 4.2BSD ON THE VAX

BUILDING 4.2BSD SYSTEMS WITH CONFIG

BUILDING 4.2BSD SYSTEMS WITH CONFIG *(continued)*

SETTING UP VERSION 1.0 OF UNIX/32V OPERATING SYSTEM

REGENERATING SYSTEM SOFTWARE FOR UNIX/32V

A DIAL-UP NETWORK OF UNIX SYSTEMS

UUCP IMPLEMENTATION DESCRIPTION

PART 1: OPERATING SYSTEM CHANGES

The two articles in Part 1 deal with the differences between the 4.2BSD UNIX system (the ULTRIX-32 system is based on 4.2BSD) and earlier versions of UNIX. "Bug Fixes and Changes in 4.2BSD" gives a comprehensive list of improvements to the system including:

- A new set of interprocess communication facilities
- A new signal package
- Support for advisory locking on files
- Per-user and per-file system disk quotas
- A new symbolic debugger, *dbx*, for C and Fortran programs
- A new internetwork mail router, *sendmail*

Changes to specific software tools and commands are listed alphabetically.

"A Fast File System for UNIX," by McKusick, Joy, Leffler, and Fabry, explains the new file system in detail. This file system is specifically designed for the VAX hardware; it is available only on 4.2BSD and the ULTRIX-32 system. The article is essential to people responsible for management and administration of ULTRIX-32 systems.

The new file system, unlike the original UNIX file system, allows you to select a block size. The block size can be 4096 bytes or any power of 2 greater than 4096; you must choose the size when you create the file system. You can optimize the disk usage and file transfer rates on your ULTRIX-32 system by choosing a block size that:

- Matches the physical characteristics of your disk drives
- Is appropriate for your applications

The article also explains use of:

- A file-locking facility that allows cooperating programs to apply advisory locks on files
- Symbolic links that allow references across separate physical file systems
- A rename facility that replaces three system calls with one
- A *quota* utility that allows the system administrator to set limits on the number of blocks and the number of files available to each user

You can find more detailed information on the *quota* utility in "Disk Quotas in a UNIX Environment" in Part 2 of this volume.

Bug fixes and changes in 4.2BSD

Samuel J. Leffler

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

Notable improvements

- The file system organization has been redesigned to provide at least an order of magnitude improvement in disk bandwidth.
- The system now provides full support for the DOD Standard TCP/IP network communication protocols. This support has been integrated into the system in a manner which allows the development and concurrent use of other communication protocols. Hardware support and routing have been isolated from the protocols to allow sharing between varying network architectures. Software support is provided for 10 different hardware devices including 3 different 10 Mb/s Ethernet modules.
- A new set of interprocess communication facilities has replaced the old multiplexed file mechanism. These new facilities allow unrelated processes to exchange messages in either a connection-oriented or connection-less manner. The interprocess communication facilities have been integrated with the networking facilities (described above) to provide a single user interface which may be used in constructing applications which operate on one or more machines.
- A new signal package which closely models the hardware interrupt facilities found on the VAX replaces the old signals and jobs library of 4.1BSD. The new signal package provides for automatic masking of signals, sophisticated signal stack management, and reliable protection of critical regions.
- File names are now almost arbitrary length (up to 255 characters) and a new file type, symbolic link, has been added. Symbolic links provide a "symbolic referencing" mechanism similar to that found in Multics. They are interpolated during pathname expansion and allow users to create links to files and directories which span file systems.
- The system supports advisory locking on files. Files can have "shared" or "exclusive" locks applied by processes. Multiple processes may apply shared locks, but only one process at any time may have an exclusive lock on a file. Further, when an exclusive lock is present on a file, shared locks are disallowed. Locking requests normally block a process until they can be completed, or they may be indicated as "non-blocking" in which case an error is returned if the lock can not be immediately obtained.
- The group identifier notion has been extended to a "group set". When users log in to the system they are placed in all their groups. Access control is now done based on the group set rather than just a single group id. This has obviated the need for the newgrp command.
- Per-user, per-filesystem disk quotas are now part of the system. Soft and hard limits may be specified on a per user and per filesystem basis to control the number of files and

amount of disk space allocated to a user. Users who exceed a soft limit are warned and if, after three login sessions, their disk usage has not dropped below the soft limit, their soft limit is treated as a hard limit. Utilities exist for the creation, maintenance, and reporting of disk quotas.

- System time is now available in microsecond precision and millisecond accuracy. Users are provided with 3 high-resolution timers which may be set up to automatically reload on expiration. The timers operate in real time, user time, and process virtual time (for profiling). All statistics and times returned to users are now given in a standard format with seconds and microseconds separated. This eliminates program dependence on the line clock frequency.
- A new system call to rename files in the same file system has been added. This call eliminates many of the anomalies which could occur in older versions of the system due to lack of atomicity in removing and renaming files.
- A new system call to truncate files to a specific length has been added. This call improves the performance of the Fortran I/O library.
- Swap space configuration has been improved by allowing multiple swap partition of varying sizes to be interleaved. These partitions are sized at boot time to minimize configuration dependencies.
- The Fortran 77 compiler and associated I/O library have undergone extensive changes to improve reliability and performance. Compilation may, optionally, include optimization phases to improve code density and decrease execution time.
- A new symbolic debugger, dbx, replaces the old symbolic debugger sdb. Dbx works on both C and Fortran 77 programs and allows users to set break points and trace execution by source code line numbers, references to memory locations, procedure entry, etc. Dbx allows users to reference structured and local variables using the program's programming language syntax.
- The delivermail program has been replaced by sendmail. Sendmail provides full internetwork routing, domain style naming as defined in the DARPA Request For Comments document #833, and eliminates the compiled in configuration database previously used by delivermail. Further, sendmail uses the DARPA standard Simple Mail Transfer Protocol (SMTP) for mail delivery.
- The system contains a new line printer system. Multiple line printers and spooling queues are supported through a printer database file. Printers on serial lines, raster printing devices, and laser printers are supported through a series of filter programs which interface to the standard line printer "core programs". A line printer control program, lpc, allows printers and printer queues to be manipulated. Spooling to remote printers is supported in a transparent fashion.
- Cu has been replaced by a new program tip. Tip supports a number of auto-call units and allows destination sites to be specified by name rather than phone number. Tip also supports file transfer to non-UNIX machines and can be used with sites which require half-duplex and/or odd-even parity.
- Uucp now supports many auto-call units other than the DN11. Spooling has been reorganized into multiple directories to cut down on system overhead. Several new utilities and shell scripts exist for use in adminstrating uucp traffic. Operation has been greatly improved by numerous bug fixes.

Bug fixes and changes

Section 1

adb Support has been added for interpreting kernel data structures on a running system and in post mortem crash dumps created by savecore. A **−k** option causes adb to map addresses according to the system and current process page tables. A new command, $p, can be used to switch between process contexts. Many scripts are available for symbolically displaying kernel data structures, searching for a process' context by process ID, etc. A new document, "Using ADB to Debug the UNIX Kernel", supplies hints in the use of adb with system crash dumps.

addbib Is a new utility for creating and extending bibliographic data bases for use with refer.

apply Is a new program which may be used to apply a command to a set of arguments.

ar Has a new key, 'o', for preserving a file's modification time when it is extracted from an archive.

cc Supports the additional symbol information used by dbx. The old symbol information, used by the defunct sdb debugger, is available by specifying the **−go** flag. A new flag, **−pg**, creates executable programs which collect profiling information to be interpreted by the new gprof program. A bug in the C preprocessor, which caused line numbers to be incorrect for macros with formal parameters with embedded newlines has been fixed. The C preprocessor now properly handles hexadecimal constants in "#if" constructs and checks for missing "#endif" statements.

chfn Now works interactively in changing a user's information field in the password file.

chgrp Is now in section 1 and may be executed by anyone. Users other than the superuser may change group ownership of a file they own to any group in their group access list.

cp Now has a **−r** flag to copy recursively down a file system tree.

csh A bug which caused backquoted commands to wedge the terminal when interrupted has been fixed. Job identifiers are now globbed. A bug which caused the "wait" command to uninteruptible in certain cases has been fixed. History may now be saved and restored across terminal sessions with the *savehist* variable. The newgrp command has been deleted due to the new group facilities.

ctags Now handles C **typedefs**.

cu Exists only in the form of a "compatible front-end" to the new tip program.

dbx Is a new symbolic debugger replacing sdb. Dbx handles C and Fortran programs.

delivermail Has been replaced by the new sendmail program.

df Understands the new file system organization and reports all disk space totals in kilobytes.

du Now reports disk usage in kilobytes and uses the new field in the inode structure which contains the actual number of blocks allocated to a file to increase accuracy of calculations.

dump Has been moved to section 8.

error Has been taught about the error message formats of troff.

eyacc A bug which caused the generated parser to not recognize valid null statements has been fixed.

f77 Has undergone major changes.

The i/o library has been extensively tested and debugged. Sequential files are now opened at the BEGINNING by default; previously they were opened at the end.

Compilation of data statements has been substantially sped up. Significant new optimization is optionally available (this is still a bit buggy and should be used with caution). Even without optimization, however, single precision computations execute much faster.

The new debugger, dbx, has replaced sdb for debugging Fortran programs; sdb is no longer supported.

Files with ".F" suffixes are preprocessed by the C preprocessor. This allows C-style "#include" and "#define" constructs to be used. The compiler has been modified to print error messages with sensible line numbers. Make also understands the ".F" suffix. Note that when using the C preprocessor, the 72 column convention is not followed.

The **−I** option for specifying short integers has been changed to **−i**. The **−I** option is now used to specify directory search paths for "#include" statements. A **−pg** option for creating executable images which collect profiling information for gprof has been added.

fed Is a font editor of dubious value.

file Now understands symbolic links.

find Has a new **−type** value, 'l', for finding symbolic links.

fp Is a new compiler/interpreter for the Functional Programming language. A supporting document is present in Volume 2C of the UNIX Programmer's Manual.

fpr Is a new program for printing Fortran files with embedded Fortran carriage controls.

fsplit Is a new program for splitting a multi-function Fortran file into individual files.

ftp Is a new program which supports the ARPA standard File Transfer Protocol.

gcore Is a new program which creates a core dump of a running process.

gprof Is a new profiling tool which displays execution time for the dynamic call graph of a program. Gprof works on C, Fortran, and Pascal programs compiled with the **−pg** option. Gprof may also be used in creating a call graph profile for the operating system. A supporting document, "gprof: A Call Graph Execution Profiler" is included in Volume 2C of the UNIX Programmer's Manual.

groups Is a new program which displays a user's group access list.

hostid Is a new program which displays the system's unique identifier as returned by the new gethostid system call. The super-user uses this program to set the host identifier at boot time.

hostname Is a new program which displays the system's name as returned by the new gethostname system call. The super-user uses this program to set the host name at boot time.

indent Is a new program for formatting C program source.

install Is a shell script used in installing software.

iostat Now reports kilobytes per second transferred for each disk. This is useful as the unit of information transferred is no longer a constant one kilobytes.

last Now displays the remote host from which a user logged in (when accessing a machine across a network). The pseudo user "ftp" may be specified to find out information about FTP file transfer sessions.

lastcomm Now displays flags for each command indicating if the program dumped core, used PDP-11 mode, executed with a set-user-ID, or was created as the result of a fork (with no following exec).

learn Now has lessons for vi (this is user contributed software which is not part of the standard system).

lint Has a new **−C** flag for creating lint libraries from C source code. Has improved type checking on static variables.

lisp Has been ported to several 68000 UNIX systems, the relevant code is included in the distribution. A new vector data type and a form of "closure" have been added.

ln Has a new flag, **−s**, for creating symbolic links.

login Has been extensively modified for use with the rlogind and telnetd network servers.

lpq Is totally new, see lpr.

lpr And its related programs are totally new. The line printer system supports multiple printers of many different characteristics. A master data base, /etc/printcap, describes both local printers and printers accessable across a network. A document describing the line printer system is now part of Volume 2C of the UNIX Programmer's Manual.

lprm Is totally new, see lpr.

ls Has been rewritten for the new directory format. It understands symbolic links and uses the new inode field which contains the actual number of blocks allocated to a file when the **−s** flag is supplied. Many rarely used options have been deleted.

m4 A bug which caused m4 to dump core when keywords were undefined then redefined has been fixed.

Mail Now supports mail folders in the style of the Rand MH system. Has been reworked to cooperate with sendmail in understanding the new mail address formats. Allows users to defined message header fields which are not be displayed when a messages is viewed. Many other changes are described in a revised version of the user manual.

make Understands not to unlink directories when interrupted. Understands the new ".F" suffix for Fortran source files which are to be run through the C preprocessor. Has a new predefined macro MFLAGS which contains the flags supplied to make on the command line (useful in creating hierarchies of makefiles).

mkdir Now uses the mkdir system call to run faster.

mt Has a new command, status, which shows the current state of a tape drive.

mv Has been rewritten to use the new rename system call. As a result, multiple directories may now be moved in a single command, the restrictions on having ".." in a pathname are no longer present, and everything runs faster.

net And all related Berknet programs are no longer part of the standard distribution. These programs live on in /usr/src/old for those who can not do without them.

netstat Is a new program which displays network statistics and active connections.

oldcsh No longer exists.

od Has gobs of new formats options.

pagesize Is a new program which prints the system page size for use in constructing portable shell scripts.

passwd Now reliably interlocks with chsh, chfn, and vipw, in guarding against concurrent updates to the password file.

pc/pi **For** loops are now done according to the standard. Files may now be dynamically allocated and disposed. Records and variant records are now aligned to correspond to C structures and unions (this was falsely claimed before). Several obscure bugs involving formal routines have been fixed. Three new library routines support random access file i/o, see /usr/include/pascal for details.

pc **For** loop variables and **with** pointers are now allocated to registers. Separate compilation type checking can now be done without reference to the source file; this permits movement (including distribution) of .o files and creation of libraries. Display entries are saved only when needed (a speed optimization).

pdx Is a new debugger for use with pi. Pdx is invoked automatically by the interpreter if a run-time error is encountered. Future work is planned to extend the new dbx debugger to understand code generated by the Pascal compiler pc.

ps Has been changed to work with the new kernel and is no longer dependent on system page size. All process segment sizes are now shown in kilobytes. Understands that the old "using new signal facilities" bit in the process structure now means "using old 4.1BSD signal facilities".

pwd Now simply calls the *getwd*(3) routine.

rcp Is a new program for copying files across a network. The complete syntax of cp is supported, including recursive directory copying.

refer Has had many bugs fixed in it and the associated −ms macro package support made to work.

reset Now resets all the special characters to the system defaults specified in the include file <sys/ttychars.h>.

rlogin Is a new program for logging in to a machine across a network. Rlogin uses the files /etc/hosts.equiv and .rhosts in the users login directory to allow logins to be performed without a password. Rlogin supports proper handling of ^S/^Q and flushing of output when an interrupt is typed at the terminal. Its '~' escape sequences are reminiscent of the old cu program (as it is based on the same source code).

rmdir Now uses the rmdir system call to run more efficiently and not require root privileges. Unfortunately, this means arguments which end in one or more "/" characters are no longer legal.

roffbib Is a new program for running off bibiliographic databases.

rsh Is a new program which supports remote command execution across a network.

ruptime Is a new program which displays system status information for clusters of machines attached to a local area network.

rwho Is a new program which displays users logged in on clusters of machines attached to a local area network.

script Has been rewritten to use pseudo-terminals. This allows the C shell job control facilities (among other things) to be used while scripting. A side effect of this change is that scripts now contain everything typed at a terminal.

sdb Has been replaced by dbx; it still lives on in /usr/src/old for those with a personal attachment.

sendbug Is a new command for submitting bug reports on 4.2BSD in a standard format suitable for automatic filing by the bugfiler program.

sh No longer has a newgrp command due to the new groups facilities.

sortbib Is a new command for sorting bibliographic databases.

strip Has been made blindingly fast by using the new truncate system call (thereby eliminating the old method of copying the file).

stty The default system erase, kill, and interrupt characters have been made the DEC standard values of DEL ('^?'), '^U', and '^C'. This is not expected to gain much popularity, but was done in the interest of compatibility with many other standard operating systems.

su Has been changed to do a "full login" when starting up the subshell. A new flag, **−f**, does a "fast" su for when a system is heavily loaded. Extra arguments supplied to su are now treated as a command line and executed directly instead of creating an interactive shell.

sysline Is a new program for maintaining system status information on terminals which support a "status line"; a poor man's alternative to a window manager (or emacs).

tail Has a larger buffer so that "tail −r" and similar show more.

talk Is a new program which provides a screen-oriented write facility. Users may be "talked to"across a network, though satellite response times have indicated overseas conversations are still best done by phone. Can be very obnoxious when engaged in important work.

tar Now allocates its internal buffers dynamically so that the block size can be specified to be very large for streaming tape drives. Also, now avoids many core-core copy operations. Has a new **−C** option for forcing chdir operations in the middle of operation (thereby allowing multiple disjoint subtrees to be easily placed in a single file, each with short relative pathnames). Has a new flag, 'B', for forcing 20 block records to be read and written; useful in joining two tar commands with a remote shell to transfer large amounts of data across a network.

telnet Is a new program which supports the ARPA standard Telnet protocol.

tip Replaces cu as the standard mechanism for connecting to machines across a phone line or through a hardwired connection. Tip uses a database of system descriptions, supports many different auto-call units, and understands many nuances required to talk to non-UNIX systems. Files may be transferred to and from non-UNIX systems in a simple fashion.

ul A bug which sometimes caused an extra blank line to be printed after reaching end of file has been fixed.

uucp And related programs have been extensively enhanced to support many different auto-call units and multiple spooling directories (among other things). A large number of bugs and performance enhancements have been made.

uusnap Is a new program which gives a snap-shot of the uucp spooling area.

vfontinfo Is a program used to inspect and print information about fonts.

vgrind Now uses a regular expression language to describe formatting. A **−f** flag forces vgrind to act as a filter, generating output suitable for inclusion in troff and/or nroff documents. Language descriptions exist for C, Pascal, Model, C shell, Bourne shell, Ratfor, and Icon programs.

vi A bug which caused the ^B command to place the cursor on the wrong line has been fixed. A bug which caused vi to believe a file had been modified when an i/o error occurred has been fixed. A bug which allowed "hardtabs" to be set to 0 causing a divide by zero fault has been fixed.

vlp Is a new program for pretty printing Lisp programs.

vmstat Has had one new piece of information added to **−s** summary, the number of fast page reclaims performed. The fields related to paging activity are now all given in kilobytes.

vpr And associated programs for spooling and printing files on Varian and Versatec printers are now shell scripts which use the new line printer support.

vwidth Is a new program for making troff width tables for a font.

wc Is once again identical to the version 7 program. That is, the **−v**, **−t**, **−b**, **−s**, and **−u** flags have been deleted.

whereis Understands the new directory organization for the source code.

which Now understands how to handle aliases.

who Now displays the remote machine from which a user is logged in.

Section 2.

The most important change in section 2 is that the documentation has been significantly improved. Manual page entries now indicate the possible error codes which may be returned and how to interpret them. The introduction to section 2 now includes a glossary of terms used throughout the section. The terminology and formatting have been made consistent. Many manual pages now have "NOTES" or "CAVEATS" providing useful information heretofore left out for the sake of brevity. As always the manual pages are still for the programmer; they are terse and extremely concise. The "4.2BSD System Manual" is likewise concise, but a bit more verbose in providing an overall picture of the system facilities.

With regard to changes in the facilities, these fall into three major categories: interprocess communication, signals, and file system related calls. The interprocess communication facilities center around the *socket* mechanism described in the "A 4.2BSD Interprocess Communication Primer". The new signals do not have an accompanying document, so the manual pages should be studied carefully. The new file system calls pretty much stand on their own, with a late section of the document "A Fast File System for UNIX" supplying a quick overview of the most important new file system facilities. Finally, it should be noted that the job control facilities introduced in 4.1BSD have been adopted as a standard part of 4.2BSD. No special distinction is given to these calls (in 4.1BSD they were earmarked "2J").

Many of the new system calls have both a "set" and a "get" form. Only the "get" forms are indicated below. Consult the manual for details on the "set" form.

intro Has been updated to reflect the new list of possible error codes. Now includes a glossary of terminology used in section 2.

access Now has symbolic definitions for the *mode* parameter defined in *<sys/file.h>*.

bind Is a new interprocess communication system call for binding names to sockets.

connect Is a new interprocess communication system call for establishing a connection between two sockets.

creat Has been obsoleted by the new *open* interface.

fchmod Is a new system call which does a *chmod* operation given a file descriptor; useful in conjunction with the new advisory locking facilities.

fchown Is a new system call which does a *chown* operation given a file descriptor; useful in conjunction with the new advisory locking facilities.

fcntl Is a new system call which is useful in controlling how i/o is performed on a file descriptor (non-blocking i/o, signal drive i/o). This interface is compatible with the System III fcntl interface.

flock Is a new system call for manipulating advisory locks on files. Locks may be shared or exclusive and locking operations may be indicated as being non-blocking, in which case a process is not blocked if the requested lock is currently in use.

fstat Now returns a larger stat buffer; see below under stat.

fsync Is a new system call for synchronizing a file's in-core state with that on disk. Its intended use is in building transaction oriented facilities.

ftruncate Is a new system call which does a *truncate* operation given a file descriptor; useful in conjunction with the new advisory locking facilities.

getdtablesize Is a new system call which returns the size of the descriptor table.

getgroups Is a new system call which returns the group access list for the caller.

gethostid Is a new system call which returns the unique (hopefully) identifier for the current host.

gethostname Is a new system call which returns the name of the current host.

getitimer Is a new system call which gets the current value for an interval timer.

getpagesize Is a new system call which returns the system page size.

getpriority Is a new system call which returns the current scheduling priority for a specific process, a group of processes, or all processes owned by a user. In the latter two cases, the priority returned is the highest (lowest numerical value) enjoyed by any of the specified processes.

getrlimit Is a new system call which returns information about a resource limit. The getrlimit and setrlimit calls replace the old vlimit call from 4.1BSD.

getrusage Is a new system call which returns information about resource utilitization of a child process or the caller. This call replaces the vtimes call of 4.1BSD.

getsockopt Is a new interprocess communication system call which returns the current options present on a socket.

gettimeofday Is a new system call which returns the current Greenwich date and time, and the current timezone in which the machine is operating. Time is returned in seconds and microseconds since January 1, 1970.

ioctl Has been changed to encode the size of parameters and whether they are to be copied in, out, or in and out of the user address space in the *request*. The symbolic names for the various ioctl requests remain the same, only the numeric values have changed. A number of new ioctls exist for use with sockets and the network facilities. The old LINTRUP request has been replaced by a call to fcntl and the SIGIO signal.

killpg Has now been made a system call; in 4.1BSD it was a library routine.

listen Is a new interprocess communication system call used to indicate a socket will be used to listen for incoming connection requests.

lseek Now has symbolic definitions for its *whence* parameter defined in *<sys/file.h>*.

mkdir Is a new system call which creates a directory.

mpx The multiplexed file facilities are no longer part of the system. They have been replaced by the socket, and related, system calls.

open Is different, now taking an optional third parameter and supporting file creation, automatic truncation, automatic append on write, and "exclusive" opens. The open interface has been made compatible with System III with the exception that non-blocking opens on terminal lines requiring carrier are not supported.

profil Now returns statistical information based on a 100 hz clock rate.

quota Is a new system call which is part of the disk quota facilities. Quota is used to manipulate disk quotas for a specific user, as well as perform certain random chores such as syncing quotas to disk.

read Now automatically restarts when a read on a terminal is interrupted by a signal before any data is read.

readv Is a new system call which supports scattering of read data into (possibly) disjoint areas of memory.

readlink Is a new system call for reading the value of a symbolic link.

recv Is a new interprocess communication system call used to receive a message on a connected socket.

recvfrom Is a new interprocess communication system call used to receive a message on a (possibly) unconnected socket.

recvmsg Is a new interprocess communication system call used to receive a message on a (possibly) unconnected socket which may have access rights included. When using on-machine communication, recvmsg and sendmsg may be used to pass file descriptors between processes.

rename Is a new system call which changes the name of an entry in the file system (plain file, directory, character special file, etc.). Rename has an important property in that it guarantees the target will always exist, even if the system crashes in the middle of the operation. Rename only works with source and destination in the same file system.

rmdir Is a new system call for removing a directory.

select Is a new system call (mainly for interprocess communication) which provides facility for synchronous i/o multiplexing. Sets of file descriptors may be queried for readability, writability, and if any exceptional conditions are present (such as out of band data on a socket). An optional timeout may also be supplied in which case the select operation will return after a specified period of time should no descriptor satisfy the requests.

send Is a new interprocess communication system call for sending a message on a connected socket.

sendto Is a new interprocess communication system call for sending a message on a (possibly) unconnected socket.

sendmsg Is a new interprocess communication system call for sending a message on a (possibly) unconnected socket which may included access rights.

setquota Is a new system call for enabling or disabling disk quotas on a file system.

setregid Is a new system call which replaces the 4.1BSD setgid system call. Setregid allows the real and effective group ID's of a process to be set separately.

setreuid Is a new system call which replaces the 4.1BSD setuid system call. Setreuid allows the real and effective user ID's of a process to be set separately.

shutdown Is a new interprocess communication system call for shutting down part or all of full-duplex connection.

sigblock Is a new system call for blocking signals during a critical section of code.

sigpause Is a new system call for blocking a set of signals and then pausing indefinitely for a signal to arrive.

sigsetmask Is a new system call for setting the set of signals which are currently blocked from delivery to a process.

sigstack Is a new system call for defining an alternate stack on which signals are to be processed.

sigsys Is no longer supported. The new signal facilities are a superset of those which sigsys provided.

sigvec Is the new system call interface for defining signal actions. For each signal (except SIGSTOP and SIGKILL), sigvec allows a "signal vector" to be defined. The signal vector is comprised of a handler, a mask of signals to be blocked while the handler executes, and an indication of whether or not the handler should execute on a signal stack defined by a sigstack call. The old signal interface is provided as a library routine with several important caveats. First, signal actions are no longer reset to their default value after a signal is delivered to a process. Second, while a signal handler is executing the signal which is being processed is blocked until the handler returns. To simulate the old signal interface, the user must explicitly reset the signal action to be the default value and unblock the signal being processed.

Four new signals have been added for the interprocess communication and interval timer facilities. SIGIO is delivered to a process when an fcntl call enables signal driven i/o and input is present on a terminal (and a signal handler is defined). SIGURG is delivered when an urgent condition arises on a socket (and a handler is defined). SIGPROF and SIGVTALRM are associated with the ITIMER PROF and ITIMER VIRTUAL interval timers, and delivered to a process when such a timer expires (the SIGALRM signal is used for the ITIMER REAL interval timer). The old SIGTINT signal is replaced by SIGIO.

socket Is a new interprocess communication system call for creating a socket.

socketpair Is a new interprocess communication system call for creating a pair of connected and unnamed sockets.

stat Now returns a larger structure. New fields are present indicating the optimal blocking factor in which i/o should be performed (for disk files the block size of the underlying file system) and the actual number of disk blocks allocated to the file. Inode numbers are now 32-bit quantities. Several spare fields have been allocated for future expansion. These include space for 64-bit file sizes and 64-bit time stamps. Two new file types may be returned, S IFLNK for symbolic links, and S IFSOCK for sockets residing in the file system.

swapon Has been renamed from the vswapon call of 4.1BSD.

symlink Is a new system call for creating a symbolic link.

truncate Is a new system call for truncating a file to a specific size.

unlink Should no longer be used for removing directories. Directories should only be created with mkdir and removed with rmdir. Creating hard links to directories can cause disastrous results.

utime Is defunct, replaced by utimes.

utimes Is a new system call which uses the new time format in setting the accessed and updated times on a file.

vfork Is still present, but definitely on its way out. Future plans include implementing fork with a scheme in which pages are initially shared read-only. On the first attempt by a process to write on a page the parent and child would receive separate writable copies of the page.

vlimit Is no longer supported. Vlimit is replaced by the getrlimit and setrlimit calls.

vread Is no longer supported in the system.

vswapon Has been renamed swapon.

vtimes Is no longer supported. Vtimes is replaced by the getrusage call.

vwrite Is no longer supported in the system.

wait Now is automatically restarted when interrupted by a signal before status could be returned.

wait3 Returns resource usage in a different format than that which was returned in 4.1BSD. This structure is compatible with the new getrusage system call. Wait3 is now automatically restarted when interrupted by a signal before status could be returned.

write Now is automatically restarted when writing on a terminal and interrupted by a signal before any i/o was completed.

writev Is a new version of the write system call which supports gathering of data in (possibly) discontiguous areas of memory

Section 3

The section 3 documentation has been reorganized to group manual entries by library. Introductory sections for each logical and physical library contain lists of the entry points in the library.

A number of routines which had been system calls under 4.1BSD are now user-level library routines in 4.2BSD. These routines have been grouped under section "3C" headings, "C" for compatibility. Further, certain routines present in the standard C run-time library which do not easily categorize as part of one of the standard libraries, have been group under "3X" headings.

curses A number of bug fixes have been incorporated, and the documentation has been revised.

stdio The standard i/o library has been modified to block i/o operations to disk files according to the block size of the underlying file system. This is accomplished using the new *st blksize* value returned by *fstat*. The resultant performance improvement is significant as the old 1 kilobyte buffer size often resulted in 7 memory-to-memory copy operations by the system on 8 kilobyte block file systems.

End-of-file marks now "stick". That is, all input requests on a stdio channel after encountering end-of-file will return end-of-file until a *clearerr* call is made. This has implications for programs which use stdio to read from a terminal and do not process end-of-file as a terminating keystroke.

Two new functions may be used to control i/o buffering. The *setlinebuf* routine is used to change **stdout** or **stderr** from block buffered or unbuffered to line buffered. The *setbuffer* routine is an alternate form of *setbuf* which can be used after a stream has been opened, but before it is read or written.

bstring Three new routines, *bcmp*, *bcopy*, and *bzero* have been added to the library. These routines use the VAX string instructions to manipulate binary byte strings of a known size.

ctime Now uses the *gettimeofday* system call and supports time conversion in six different time zones. Daylight savings calculations are also performed in each time zone when appropriate.

isprint Now considers space a printing character; as the manual page has always indicated.

directory Is a new directory interface package which provides a portable interface to reading directories. A version of this library which operates under 4.1BSD is also available.

getpass Now properly handles being unable to open /dev/tty.

getwd Has been moved from the old jobs library to the standard C run-time library. It now returns an error string rather than printing on the standard error when unable to decipher the current working directory.

perror Now uses the writev system call to pass all its arguments to the system in a single system call. This has profound effects on programs which transmit error messages across a network.

psignal And sys siglist are routines for printing signal names in an equivalent manner to perror.

qsort Has been greatly sped up by choosing a random element with which to apply its divide and conquer algorithm.

random Is a successor to rand which generates much better random numbers. The old rand routine is still available and most programs have not been switched over to random as doing so would make certain facilities such encrypted mail unable to operate on existing data files.

setjmp And longjmp now save and restore the signal mask so that non-local exit from a signal handler is transparent. The old semantics are available with setjmp and longjmp.

net Is a new set of routines for accessing database files for the DARPA Internet. Four databases exist: one for host names, one for network names, one for protocol numbers, and one for network services. The latter returns an Internet port and protocol to be used in accessing a given network service.

An additional collection of routines, all prefaced with "inet " may be used to manipulate Internet addresses, and interpret and convert between Internet addresses and ASCII representations in the Internet standard "dot" notation.

Finally, routines are available for converting 16 and 32 bit quantities between host and network order (on high-ender machines these routines are defined to be noops).

fstab The routines for manipulating /etc/fstab have been rewritten to return arbitrary length null-terminated strings.

Section 4

The system now supports the 11/730, the new 64Kbit RAM memory controllers for the 11/750 and 11/780, and the second UNIBUS adapter for the 11/750. Several new character and/or block device drivers have been added, as well as support for many hardware devices which are accessible only through the network facilities. Each new piece of hardware supported is listed below.

New manual entries in section 4 have been created to describe communications protocols, and network architectures supported. At present the only network architecture fully supported is the DARPA Internet with the TCP, IP, UDP, and ICMP protocols.

acc A network driver for the ACC LH/DH IMP interface.

ad A driver for the Data Translation A/D converter.

arp The Address Resolution Protocol for dynamically mapping betwee 32-bit DARPA Internet addresses and 48-bit Xerox 10Mb/s Ethernet addresses.

css A network driver for the DEC IMP-11A LH/DH IMP interface.

dmc A network interface driver for the DEC DMC-11/DMR-11 point-to-point communications device.

ec A network interface driver for the 3Com 10Mb/s Ethernet controller.

en A network interface driver for the Xerox 3Mb/s experimental Ethernet controller.

hy A network interface driver for the Network Systems Hyperchannel Adapter.

ik A driver for an Ikonas frame buffer graphics device interface.

il A network interface driver for the Interlan 10Mb/s Ethernet interface.

imp A network interface driver for the standard 1822 interface to an IMP; used in conjunction with either acc or css hardware.

kg A driver for a KL-11/DL-11W used as an alternate real time clock source for gathering kernel statistics and profiling information.

lo A software loopback network interface for protocol testing and performance analysis.

pcl A network interface driver for the DEC PCL-11B communications controller.

ps A driver for an Evans and Sutherland Picture System 2 graphics device connected with a DMA interface.

pty Now includes a simple packet protocol to support flow controlled operation with mechanisms for flushing data to be read and/or written.

rx A driver for the DEC dual RX02 floppy disk unit.

ts Now supports TU80 tape drives.

tu The VAX-11/750 console cassette interface has been made somewhat usable when operating in single-user mode. The device driver employs assembly language pseudo-dma code for the reception of incoming packets from the cassette.

uda Now supports RA81, RA80, and RA60 disk drives.

un A network interface driver for an Ungermann-Bass network interface unit connected to the host via a DR-11W.

up Now supports ECC correction and bad sector handling. Also has improved logic for recognizing many different kinds of disk drives automatically at boot time.

uu A driver for DEC dual TU58 tape cartridges connected via a DL-11W interface.

va The Varian driver has been rewritten so that it may coexist on the same UNIBUS with devices which require exclusive use of the bus; i.e. RK07's.

vv A network interface driver for the Proteon proNET 10Mb/s ring network controller.

Section 5

dir Reflects the new directory format.

disktab Is a new file for maintaining disk geometry information. This is a temporary scheme until the information stored in this file for each disk is recorded on the disk pack itself.

dump Is a superset of that used in 4.1BSD.

fs Reflects the new file system organization.

gettytab Is a new file which idescribes terminal characteristics. Each entry in the file describes one of the possible arguments to the getty program.

hosts Is a database for mapping between host names and DARPA Internet host addresses.

mtab Has been modified to include a "type" field indicating whether the file system is mounted read-only, read-write, or read-write with disk quotas enabled.

networks Is a database for mapping between network names and DARPA standard network numbers.

phones Is a phone number data base for tip.

printcap Is a termcap clone for configuring printers.

protocols Is a database for mapping between protocol names and DARPA Internetwork standard protocol numbers.

remote Is a database of remote hosts for use with tip.

services Is a database in which DARPA Internet services are recorded. The information contained in this file indicates the name of the service, the protocol which is required to access it, and the port number at which a client should connect to utilize the service.

tar Is a new entry describing the format of a tar tape.

utmp Has been augmented to include a remote host from which a login session originates. The wtmp file is also used to record FTP sessions.

vgrindefs Is a file describing how to vgrind programs written in many languages.

Section 6

aardvark Does not work because it requires the "Dungeon Definition Language" processor which is a binary image requiring 4.1BSD compatibility mode; the DDL source is still present.

aliens The aliens have returned home, the game is no longer included in the distribution.

backgammon Is now screen oriented. A new program, teachgammon, instructs the new backgammon player. The old version is now called btlgammon.

canfield Is a new game which plays a brand of the popular game of solitaire. Betting is included, the program cfscores may be used to find out your current debt.

ching Now pipes its output through more. Thus the hacker placates the seekers.

chase No longer exists because the binary does not work under 4.2BSD.

factor Is a rewrite in C of the old version 7 assembly language program which finds the prime factors of a number.

fortune Has yet more adages.

hangman Is now screen oriented.

mille Now plays more intelligently.

primes Is a rewrite in C of the old version 7 assembly language program which finds prime numbers within a specified range.

rogue Has been made more of a scoundrel. The supplementary document "A Guide to the Dungeons of Doom", has been updated as well, and is now part of Volume 2C of the programmer's manual.

sail Is a new game which simulates sea battles of yore. The manual page is large enough to be a separate document and so has been left in its source directory.

trek The original trek has returned; trekies rejoice.

Section 7

hier Has been updated to reflect the reorganization to the user and system source.

mailaddr Is a new entry describing mail addressing syntax under sendmail (possibly too Berkeley specific).

ms The −ms macros have been extended to allow automatic creation of a table of contents. Support for the refer preprocessor is improved. Several bugs related to multi-column output and floating keeps have been fixed. Extensions to the accent mark string set are available by including the .AM macro. Footnotes can now be automatically numbered (in superscript) by −ms and referenced in the text with a ** string register. The manual page includes a summary of important number and string registers. A new document "Changes to −ms" is included in Volume 2C of the programmer's manual.

Section 8

Major changes affecting system operations include:

- The system now supports disk quotas. These allow system administrators to control users' disk space and file allocation on a per-file system basis. Utilities in this section exist for fixing, summarizing, and editing disk quota summary files.
- File systems are now made with a new program, newfs, which acts as front end to the old mkfs program. There no longer is a need to remember disk partition sizes, as newfs gets this information automatically from the /etc/disktab file. In addition, newfs attempts to lay out file systems according to the characteristics of the underlying disk drive (taking into account disk geometry information).
- DEC standard bad block forwarding is now supported on the RP06 and second vendor UNIBUS storage module disks. The bad144 program can now be used to mark sectors bad on many disks, though inclusion in the bad sector table is still somewhat risky due to requirements in the ordering of entries in the table.
- A new program, format, should be used to initialize all non-DEC storage modules before creating file systems. Format formats the sector headers and creates a bad sector table which is used in normal system operation. Format runs in a standalone mode.
- Getty has been rewritten to use a description file, /etc/gettytab. This allows sites to tailor terminal operation and configuration without making modifications to getty.
- The line printer system is totally new. A program to administer the operation of printers, lpc, is supplied, and printer accounting has been consolidated into a single program, pac.
- The program used to restore files from dump tapes is now called *restore*. This name change was done to reinforce the fact that it is completely rewritten and operates in a very different way than the old restor program. Restore operates on mounted file systems and uses only normal file system operations to restore files. Versions of both dump and restore which operate across a network are included as rdump and rrestore. Dump and restore (and their network oriented counterparts) now perform so efficiently (mostly because of the new file system), that disk to disk backups should no longer be an attractive alternative.

arff No longer asks if you want to clobber the floppy when manipulating archives which are not on the floppy.

bad144 Has been modified to use the /etc/disktab file. Can be used to create bad sector tables for the DEC RP06 and several new Winchester disk drives. Consult the source code for details and use with extreme care.

badsect Has been modified to work with the new file system and now must interact with fsck to perform its duties. Consult the manual page for more information.

bugfiler Is a new program for automatic filing and acknowledgement of bug reports submitted by the sendbug program. Intended to operate with the Rand MH software which is part of the user contributed software. Used at Berkeley to process bug reports on 4.2BSD.

chgrp Has been moved to section 1.

comsat Has been changed to filter the noise lines in message headers when displaying incoming mail. No longer uses a second process watchdog as it uses the more reliable socket facilities instead of the old mpx facilities.

config Has been extensively modified to handle the new root and swap device specification syntax. A new document, "Configuring 4.2BSD UNIX Systems with Config", describes its use, as well as other important information needed in configuring system images; this is part of Volume 2C of the programmer's manual.

diskpart Is a new program which may be used to generate disk partition tables according to the rules used at Berkeley. Can automatically generate entries required for device drivers and for the /etc/diskpart file. (Does not handle the new DEC DSA style drives properly because it tries to reserve space for the bad sector table.)

drtest Is a new standalone program which is useful in testing standalone disk device drivers and for pinpointing bad sectors on a disk.

dump Has been modified for the new file system organization. Mainly due to the new file system, it runs virtually at tape speed. Properly handles locking on the dumpdates file when multiple dumps are performed concurrently on the same machine.

dumpfs Is a new program for dumping out information about a file system such as the block size and disk layout information.

edquota Is a new program for editing user quotas. Operates by invoking your favorite editor on an ASCII representation of the information stored in the binary quota files. Edquota also has a "replication" mode whereby a quota template may be used to create quotas for a group of users.

fastboot Is a new shell script which reboots the system without checking the file systems; should be used with extreme care.

fasthalt Is a new script which is similar to fastboot.

format Is a new standalone program for formatting non-DEC storage modules and creating the appropriate bad sector table on the disk.

fsck Has been changed for the new file system. Fsck is more paranoid then ever in checking the disks, and has been sped up significantly. The accompanying Volume 2C document has been updated to reflect the new file system organization.

ftpd Is the DARPA File Transfer Protocol server program. It supports C shell style globbing of arguments and a large set of the commands in the specification (except the ABORT command!).

gettable Is a new program which can be used in aquiring up to date DARPA Internet host database files.

getty Has been rewritten to use a terminal description database, /etc/gettytab. Consult the manual entries for *getty*(8) and *gettytab*(5) for more information.

icheck Has been modified for the new file system.

init Has been significantly modified to use the new signal facilities. In doing so, several race conditions related to signal delivery have been fixed.

kgmon Is a new program for controlling running systems which have been created with kernel profiling. Using kgmon, profiling can be turned on or off and internal profiling buffers can dumped into a gmon.out file suiitable for interpretation by gprof.

lpc Is a new program controlling line printers and their associated spooling queues. Lpc can be used to enable and disable printers and/or their spooling queues. Lpc can also be used to rearrange existing jobs in a queue.

lpd Has been rewritten and now runs as a "server", using the interprocess communication facilities to service print requests. A supplementary document describing the line printer system is now part of Volume 2C of the programmer's manual.

MAKEDEV Is a new shell script which resides in /dev and is used to create special files there. MAKEDEV keeps commands for creating and manipulating local devices in a separate file MAKEDEV.local.

mkfs Has been virtually rewritten for the new file system. The arguments supplied are very different. For the most part, users now use the newfs program when creating file systems. Mkfs now automatically creates the lost+found directory.

mount Now indicates file systems which are mounted read-only or have disk quotas enabled.

newfs Is a new front-end to the mkfs program. Newfs figures out the appropriate parameters to supply to mkfs, invokes it, and then, if necessary, installs the boot blocks necessary to bootstrap UNIX on 11/750's.

pac Is a new program which can be used to do printer accounting on any printer. It subsumes the vpac program.

quot Now uses the information in the inode of each file to find out how many blocks are allocated to it.

quotacheck Is a new program which performs consistency checks on disk quota files. Quotacheck is normally run from the /etc/rc.local file after a system is rebooted, though it can also be run on mounted on file systems which are not in use.

quotaon Is a new program which enables disk quotas on file systems. A link to quotaon, named quotaoff, is used to disable disk quotas on file systems.

pstat Has been modified to understand new kernel data structures.

rc Has had system dependent startup commands moved to /etc/rc.local.

rdump Is a new program to dump file systems across a network.

renice Has been rewritten to use the new setpriority system call. As a result, you can now renice users and process groups.

repquota Is a new program which summarizes disk quotas on one or more file systems.

restor No longer exists. A new program, restore, is its successor.

restore Replaces restor. Restore operates on mounted file systems; it contains an interactive mode and can be used to restore files by name. Restore has become almost as flexible to use as tar in retrieving files from tape.

rexecd Is a network server for the *rexec*(3X) library routine. Supports remote command execution where authentication is performed using user accounts and passwords.

rlogind Is a network server for the *rlogin*(1C) command. Supports remote login sessions where authentication is performed using privileged port numbers and two files, /etc/hosts.equiv and .rhosts (in each users home directory).

rmt Is a program used by rrestore and rdump for doing remote tape operations.

route Is a program for manually manipulating network routing tables.

routed Is a routing daemon which uses a variant of the Xerox Routing Information Protocol to automatically maintain up to date routing tables.

rrestore Is a version of restore which works across a network.

rshd Is a server for the *rsh*(1C) command. It supports remote command execution using privileged port numbers and the /etc/hosts.equiv and .rhosts files in users' home directories.

rwhod Is a server which generates and listens for host status information on local networks. The information stored by rwhod is used by the *rwho*(1C) and*ruptime*(1C) programs.

rxformat Is a program for formatting floppy disks (this uses the *rx* device driver, not the console floppy interface).

savecore Has been modified to get many pieces of information from the running system and crash dump to avoid compiled in constants.

sendmail Is a new program replacing delivermail; it provides fully internetwork mail forwarding capabilities. Sendmail uses the DARPA standard SMTP protocol to send and receive mail. Sendmail uses a configuration file to control its operation, eliminating the compiled in description used in delivermail.

setifaddr Is a new program used to set a network interface's address. Calls to this program are normally placed in the /etc/rc.local file to configure the network hardware present on a machine.

syslog Is a server which receives system logging messages. Currently, only the sendmail program uses this server.

telnetd Is a server for the DARPA standard TELNET protocol.

tftpd Is a server for the DARPA Trivial File Transfer Protocol.

trpt Is a program used in debugging TCP. Trpt transliterates protocol trace information recorded by TCP in a circular buffer in kernel memory.

tunefs Is a program for modifying certain parameters in the super block of file systems.

vipw Is no longer a shell script and properly interacts with passwd, chsh, and chfn in locking the password file.

A Fast File System for UNIX*

Revised July 27, 1983

Marshall Kirk McKusick, William N. Joy†,
Samuel J. Leffler‡,Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

1. Introduction

This paper describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to affect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the user visible facilities. The paper concludes with a history of the software engineering of the project.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. No constraints other than available disk space are placed on file growth [Ritchie74], [Thompson79].

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications that need to do a small amount of processing on a large quantities of data such as VLSI design and image processing, need to have a high throughput from the file system. High throughput rates are also needed by programs with large address spaces that are constructed by mapping files from the file system into virtual memory. Paging data in and out of the file system is likely to occur frequently. This requires a file system providing higher bandwidth than the original 512 byte UNIX one which provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently users of the system have not been faced with massive software conversion.

---

* UNIX is a trademark of Bell Laboratories.

†William N. Joy is currently employed by: Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043 ‡Samuel J. Leffler is currently employed by: Lucasfilm Ltd., PO Box 2009, San Rafael, CA 94912

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a lisp environment [Symbolics81a].

A major goal of this project has been to build a file system that is extensible into a networked environment [Holler73]. Other work on network file systems describe centralized file servers [Accetta80], distributed file servers [Dion80], [Luniewski77], [Porcar82], and protocols to reduce the amount of information that must be transferred across a network [Symbolics81b], [Sturgis80].

2. Old File System

In the old file system developed at Bell Laboratories each disk drive contains one or more file systems.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to a list of free blocks. All the free blocks in the system are chained together in a linked list. Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in the inode structure itself*. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further single indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A traditional 150 megabyte UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from its inode to its data. Files in a single directory are not typically allocated slots in consecutive locations in the 4 megabytes of inodes, causing many non-consecutive blocks to be accessed when executing operations on all the files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by changing the file system so that all modifications of critical information were staged so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors; each disk transfer accessed twice as much data, and most files could be described without need to access through any indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system.*

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random causing files to have their blocks allocated randomly over the disk. This forced the disk to seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of randomization of their free block list. There was no way of restoring the performance an old file system except to dump, rebuild, and restore the file system. Another possibility would be to have a process that periodically reorganized the data on the disk to restore locality as suggested by [Maruyama76].

---

† A file system always resides on a single drive.

* The actual number may vary from system to system, but is usually in the range 5-13.

3. New file system organization

As in the old file system organization each disk drive contains one or more file systems. A file system is described by its super-block, that is located at the beginning of its disk partition. Because the super-block contains critical data it is replicated to protect against catastrophic loss. This is done at the time that the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To insure that it is possible to create files as large as 2^{32} bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block so it is possible for file systems with different block sizes to be accessible simultaneously on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. For each cylinder group a static number of inodes is allocated at file system creation time. The current policy is to allocate one inode for each 2048 bytes of disk space, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. Thus a single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transfer, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before initiating a seek.

The main problem with bigger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The machine measured to obtain these figures is one of our time sharing systems that has roughly 1.2 Gigabyte of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formated space. The space wasted is measured as the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16K or greater, because of the requirement that the cylinder group information must begin at a block boundary.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	512 byte block UNIX file system
866.5 Mb	11.8	1024 byte block UNIX file system
948.5 Mb	22.4	2048 byte block UNIX file system
1128.3 Mb	45.6	4096 byte block UNIX file system

Table 1 – Amount of wasted space as a function of block size.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can be optionally broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space availability at the fragment level; to determine block availability, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 – Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0–5, 10, and 11 are in use, while fragments 6–9, and 12–15 are free. Fragments of adjoining blocks cannot be used as a block, even if they are large enough. In this example, fragments 6–9 cannot be coalesced into a block; only fragments 12–15 are available for allocation as a block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and a 3072 byte fragment. If no 3072 byte fragments are available at the time the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file as needed.

The granularity of allocation is the *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block to hold the new data. The new data is written into the available space in the block.

2) Nothing has been allocated. If the new data contains more than 4096 bytes, a 4096 byte block is allocated and the first 4096 bytes of new data is written there. This process is repeated until less than 4096 bytes of new data remain. If the remaining new data to be written will fit in three or fewer 1024 byte pieces, an unallocated fragment is located,

* A program may be overwriting data in the middle of an existing file in which case space will already be allocated.

otherwise a 4096 byte block is located. The new data is written into the located piece.

3) A fragment has been allocated. If the number of bytes in the new data plus the number of bytes already in the fragment exceeds 4096 bytes, a 4096 byte block is allocated. The contents of the fragment is copied to the beginning of the block and the remainder of the block is filled with the new data. The process then continues as in (2) above. If the number of bytes in the new data plus the number of bytes already in the fragment will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The contents of the previous fragment appended with the new data is written into the allocated piece.

The problem with allowing only a single fragment on a 4096/1024 byte file system is that data may be potentially copied up to three times as its requirements grow from a 1024 byte fragment to a 2048 byte fragment, then a 3072 byte fragment, and finally a 4096 byte block. The fragment reallocation can be avoided if the user program writes a full block at a time, except for a partial block at the end of the file. Because file systems with different block sizes may coexist on the same system, the file system interface been extended to provide the ability to determine the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The space overhead in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of space overhead as the 512 byte block UNIX file system. The new file system is more space efficient than the 512 byte or 1024 byte file systems in that it uses the same amount of space for small files while requiring less indexing information for large files. This savings is offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when the new file systems fragment size equals the old file systems block size.

In order for the layout policies to be effective, the disk cannot be kept completely full. Each file system maintains a parameter that gives the minimum acceptable percentage of file system blocks that can be free. If the the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter can be changed at any time, even when the file system is mounted and active. The transfer rates to be given in section 4 were measured on file systems kept less than 90% full. If the reserve of free blocks is set to zero, the file system throughput rate tends to be cut in half, because of the inability of the file system to localize the blocks in a file. If the performance is impaired because of overfilling, it may be restored by removing enough files to obtain 10% free space. Access speed for files created during periods of little free space can be restored by recreating them once enough space is available. The amount of free space maintained must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, a site running the old 1024 byte UNIX file system wastes 11.8% of the space and one could expect to fit the same amount of data into a 4096/512 byte new file system with 5% free space, since a 512 byte old file system wasted 6.9% of the space.

3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given

installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can adapt to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be well positioned rotationally. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with a channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks often can be accessed without suffering lost time because of an intervening disk revolution. For processors without such channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation policy routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to schedule an interrupt. Given the previous block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in a file will be coming into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the availability of blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global

policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Files in a directory are frequently accessed together. For example the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the files in a directory in the same cylinder group. To ensure that files are allocated throughout the disk, a different policy is used for directory allocation. A new directory is placed in the cylinder group that has a greater than average number of free inodes, and the fewest number of directories in it already. The intent of this policy is to allow the file clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for each cylinder group can be read with 4 to 8 disk transfers. This puts a small and constant upper bound on the number of disk transfers required to access all the inodes for all the files in a directory as compared to the old file system where typically, one disk transfer is needed to get the inode for each file in a directory.

The other major resource is the data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all the data blocks for a file in the same cylinder group, preferably rotationally optimally on the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Using up all the space in a cylinder group has the added drawback that future allocations for any file in the cylinder group will also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The solution devised is to redirect block allocation to a newly chosen cylinder group when a file exceeds 32 kilobytes, and at every megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. If the requested block is not available, the allocator allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristic guesses based on partial information.

If a requested block is not available the local allocator uses a four level allocation strategy:

1) Use the available block rotationally closest to the requested block on the same cylinder.
2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.
3) If the cylinder group is entirely full, quadratically rehash among the cylinder groups looking for a free block.
4) Finally if the rehash fails, apply an exhaustive search.

The use of quadratic rehash is prompted by studies of symbol table strategies used in programming languages. File systems that are parameterized to maintain at least 10% free space almost never use this strategy; file systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random. Consequently the most important characteristic of the strategy used when the file system is low on space is that it be fast.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empiric studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories, the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate that user programs can transfer data to or from a file without performing any processing on it. These programs must write enough data to insure that buffering in the operating system does not affect the results. They should also be run at least three times in succession; the first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The methodology and test results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the cpu or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an Ampex Capricorn 330 Megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured.

Type of File System	Processor and Bus Measured	Read Speed	Read Bandwidth	Read % CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/1100 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/1100 20%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/1100 21%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	54%

Table 2a – Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Write Speed	Write Bandwidth	Write % CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/1100 4%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/1100 13%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/1100 19%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/1200 27%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	95%

Table 2b – Writing rates of the old and new UNIX file systems.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system run with 10% free space. Synthetic work loads suggest the performance deteriorates to about half the throughput rates given in Table 2 when no free space is maintained.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is measured by doing 65536* byte reads from contiguous tracks on the disk. The bandwidth is calculated by

† A UNIX command that is similar to the reading test that we used is, "cp file /dev/null", where "file" is eight Megabytes long.

* This number, 65536, is the maximal I/O size supported by the VAX hardware; it is a remnant of the

comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-4% of the disk bandwidth, while the new file system uses up to 39% of the bandwidth.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, and the processor is unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the *write* system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers build up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek order, the average seek between the scheduled disk writes is much less than they would be if the data blocks are written out in the order in which they are generated. However when the file is read, the *read* system call is processed synchronously so the disk blocks must be retrieved from the disk in the order in which they are allocated. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

The performance of the new file system is currently limited by a memory to memory copy operation because it transfers data from the disk into buffers in the kernel address space and then spends 40% of the processor cycles copying these buffers to user address space. If the buffers in both address spaces are properly aligned, this transfer can be affected without copying by using the VAX virtual memory management hardware. This is especially desirable when large amounts of data are to be transferred. We did not implement this because it would change the semantics of the file system in two major ways; user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow files to be allocated to contiguous disk blocks that could be read in a single disk transaction. Most disks contain either 32 or 48 512 byte sectors per track. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than fifty percent of the available bandwidth. Since each track has a multiple of sixteen sectors it holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. If the the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up the allocation the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79].

---

system's PDP-11 ancestry.

5. File system functional enhancements

The speed enhancements to the UNIX file system did not require any changes to the semantics or data structures viewed by the users. However several changes have been generally desired for some time but have not been introduced because they would require users to dump and restore all their file systems. Since the new file system already requires that all existing file systems be dumped and restored, these functional enhancements have been introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems that are not running the new file system, a set of directory access routines have been introduced that provide a uniform interface to directories on both old and new systems.

Directories are allocated in units of 512 bytes. This size is chosen so that each allocation can be transferred to disk in a single atomic operation. Each allocation unit contains variable-length directory entries. Each entry is wholly contained in a single allocation unit. The first three fields of a directory entry are fixed and contain an inode number, the length of the entry, and the length of the name contained in the entry. Following this fixed size information is the null terminated name, padded to a 4 byte boundary. The maximum length of a name in a directory is currently 255 characters.

Free space in a directory is held by entries that have a record length that exceeds the space required by the directory entry itself. All the bytes in a directory unit are claimed by the directory entries. This normally results in the last entry in a directory being large. When entries are deleted from a directory, the space is returned to the previous entry in the same directory unit by increasing its length. If the first entry of a directory unit is free, then its inode number is set to zero to show that it is unallocated.

5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to create a separate "lock" file to synchronize their updates. A process would try to create a "lock" file. If the creation succeeded, then it could proceed with its update; if the creation failed, then it would wait, and try again. This mechanism had three drawbacks. Processes consumed CPU time, by looping over attempts to create locks. Locks were left lying around following system crashes and had to be cleaned up by hand. Finally, processes running as system administrator are always permitted to create files, so they had to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight-forward, so a mechanism for locking files has been added.

The most general schemes allow processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to simply serialize access with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the applications that currently run on the system, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the decision of when to override them. A hard lock is always enforced whenever a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel, with advisory locks the policy is implemented by the user programs. In the UNIX system, programs with system administrator privilege can override any protection scheme. Because many of the programs that need to use locks run as system administrators, we chose to implement advisory locks rather than create a protection scheme that was contrary to the UNIX philosophy or could not be used by

system administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process has an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the open will block until the lock can be gained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process can override the lock by opening the same file without a lock.

Locks can be applied or removed on open files, so that locks can be manipulated without needing to close and reopen the file. This is useful, for example, when a process wishes to open a file with a shared lock to read some information, to determine whether an update is required. It can then get an exclusive lock so that it can do a read, modify, and write to update the file in a consistent manner.

A request for a lock will cause the process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to poll for a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since the lock is removed when the process exits or the system crashes, there is no problem with unintentional locks files that must be cleared by hand.

Almost no deadlock detection is attempted. The only deadlock detection made by the system is that the file descriptor to which a lock is applied does not currently have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail). Thus a process can deadlock itself by requesting locks on two separate file descriptors for the same object.

5.3. Symbolic links

The 512 byte UNIX file system allows multiple directory entries in the same file system to reference a single file. The link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated. This style of links does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* have been added similar to the scheme used by Multics [Feiertag71].

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. If the symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links, and seven system utilities were modified to use these calls.

In future Berkeley software distributions it will be possible to mount file systems from other machines within a local file system. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create new versions of data files typically create the new version as a temporary file and then rename the temporary file with the original name of the data file. In the old UNIX file systems the renaming required three calls to the system. If the program were interrupted or the system crashed between these calls, the data file could be left with only its temporary name. To eliminate this possibility a single system call has been added that performs the rename in an atomic fashion to guarantee the existence of the original name.

In addition, the rename facility allows directories to be moved around in the directory tree hierarchy. The rename system call performs special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the ancestry of the target directory to insure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Each resource is given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more space while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If they fail to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

6. Software engineering

The preliminary design was done by Bill Joy in late 1980; he presented the design at The USENIX Conference held in San Francisco in January 1981. The implementation of his design was done by Kirk McKusick in the summer of 1981. Most of the new system calls were implemented by Sam Leffler. The code for enforcing quotas was implemented by Robert Elz at the University of Melbourne.

To understand how the project was done it is necessary to understand the interfaces that the UNIX system provides to the hardware mass storage systems. At the lowest level is a *raw disk*. This interface provides access to the disk as a linear array of sectors. Normally this interface is only used by programs that need to do disk to disk copies or that wish to dump file systems. However, user programs with proper access rights can also access this interface. A disk is usually formated with a file system that is interpreted by the UNIX system to provide a directory hierarchy and files. The UNIX system interprets and multiplexes requests from user programs to create, read, write, and delete files by allocating and freeing inodes and data blocks. The interpretation of the data on the disk could be done by the user programs themselves. The reason that it is done by the UNIX system is to synchronize the user requests, so that two processes do not attempt to allocate or modify the same resource simultaneously. It also allows access to be restricted at the file level rather than at the disk level and allows the common file system routines to be shared between processes.

The implementation of the new file system amounted to using a different scheme for formating and interpreting the disk. Since the synchronization and disk access routines themselves were not being changed, the changes to the file system could be developed by moving the file system interpretation routines out of the kernel and into a user program. Thus, the first step was to extract the file system code for the old file system from the UNIX kernel and change its requests to the disk driver to accesses to a raw disk. This produced a library of routines that mapped what would normally be system calls into read or write operations on the raw disk. This library was then debugged by linking it into the system utilities that copy, remove, archive, and restore files.

A new cross file system utility was written that copied files from the simulated file system to the one implemented by the kernel. This was accomplished by calling the simulation library to do a read, and then writing the resultant data by using the conventional write system call. A similar utility copied data from the kernel to the simulated file system by doing a conventional read system call and then writing the resultant data using the simulated file system library.

The second step was to rewrite the file system simulation library to interpret the new file system. By linking the new simulation library into the cross file system copying utility, it was possible to easily copy files from the old file system into the new one and from the new one to the old one. Having the file system interpretation implemented in user code had several major benefits. These included being able to use the standard system tools such as the debuggers to set breakpoints and single step through the code. When bugs were discovered, the offending problem could be fixed and tested without the need to reboot the machine. There was never a period where it was necessary to maintain two concurrent file systems in the kernel. Finally it was not necessary to dedicate a machine entirely to file system development, except for a brief period while the new file system was boot strapped.

The final step was to merge the new file system back into the UNIX kernel. This was done in less than two weeks, since the only bugs remaining were those that involved interfacing to the synchronization routines that could not be tested in the simulated system. Again the simulation system proved useful since it enabled files to be easily copied between old and new file systems regardless of which file system was running in the kernel. This greatly reduced the number of times that the system had to be rebooted.

The total design and debug time took about one man year. Most of the work was done on the file system utilities, and changing all the user programs to use the new facilities. The code changes in the kernel were minor, involving the addition of only about 800 lines of code (including comments).

Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when files were less stable than they should have been. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

References

[Accetta80] Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M. "The Design of a Network-Based Central File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-134

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Dion80] Dion, J. "The Cambridge File Server", Operating Systems Review, 14, 4. Oct 1980. pp 26-35

[Eswaran74] Eswaran, K. "Placement of records in a file and file allocation in a computer network", Proceedings IFIPS, 1974. pp 304-307

[Holler73] Holler, J. "Files in Computer Networks", First European Workshop on Computer Networks, April 1973. pp 381-396

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Luniewski77] Luniewski, A. "File Allocation in a Distributed System", MIT Laboratory for Computer Science, Dec 1977.

[Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", Communications of the ACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Porcar82] Porcar, J. "File Migration in Distributed Computer Systems", Ph.D. Thesis, Lawrence Berkeley Laboratory Tech Report #LBL-14763.

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a] Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Sturgis80] Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System", Operating Systems Review, 14, 3. pp 55-79

[Symbolics81a] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Symbolics81b] "Chaosnet FILE Protocol". Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Sept 1981.

[Thompson79] Thompson, K. "UNIX Implementation", Section 31, Volume 2B, UNIX Programmers Manual, Bell Laboratory, Murray Hill, NJ 07974. Jan 1979

[Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-???

[Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

PART 2: MAINTENANCE AND ADMINISTRATION

The three articles in this part describe system administration utilities on ULTRIX-32. Two of the utilities, *quota* and the file system check program (*fsck*), will help you keep your system running efficiently. The third utility, *sendmail*, makes possible communication between users on computers that use different networking software.

Disk Quotas

The ULTRIX-32 system allows the system manager to impose limits on the amount of disk space and the number of files available to each user. Each category (disk space and the maximum number of files) has a hard limit and a soft limit. The hard limit for a user sets an absolute maximum that cannot be exceeded. The soft limit is a guideline: the number of blocks or files that the user should try not to exceed. The *quota* utility warns any user who exceeds his or her soft limit. If the user consistently ignores the warnings, the soft limit becomes a hard limit after a set number of warnings.

The article, "Disc Quotas in a UNIX Environment," by Elz, tells how the system manager can establish, disable, or check the limits and the number of warnings for any user. Elz also explains how a user can exit without loss from an editing session in which writing the edited material to a file would exceed one of the hard limits.

Fixing Corrupted File Systems

The ULTRIX-32 system includes a file system check program called *fsck*. You can use this utility to determine whether your file system is corrupted and to fix any inconsistencies you find.

Fsck runs in two modes: noninteractive and interactive. Normally the boot procedure calls *fsck* to run noninteractively after booting the operating system. In this mode, the utility checks for inconsistencies and corrects only those that it can handle without help from an operator. In general, these are problems associated with a system crash or improper shutdown procedure. When the utility finds a problem it can't deal with, it notifies the operator and stops. The operator can then run *fsck* interactively, deciding between the alternative measures presented by the utility.

The article by McKusick, "Fsck - The UNIX File System Check Program," gives an overview of the file system, the kinds of corruption that can occur, and the methods that *fsck* uses to check for inconsistencies. An appendix provides a comprehensive list of error messages together with explanations and appropriate responses. *Fsck* is essential to proper maintenance of the ULTRIX-32 system, and this article is essential to proper use of *fsck*.

Managing the *Sendmail* Utility

Sendmail is an internetwork mail utility transparent to most users. Once it is installed and running, you can send mail to users on foreign network systems in the same way that you send mail to users on the local network. The *sendmail* utility handles the protocol and message-routing differences between networks automatically.

The "Sendmail Installation and Operation Guide," by Allman, tells what you need to know to start up the utility and to keep it running correctly from day to day. A second article, "Sendmail - An Internetwork Mail Router," in Part 3 of this volume, gives background information that tells how *sendmail* works. Read the background article before using the installation and operating information included in this part.

The installation information in the "Sendmail Installation and Operation Guide" explains:

- How to use either of two off-the-shelf configuration files supplied with the software
- How to use a makefile to install *sendmail* automatically
- How to install *sendmail* by hand by building your own configuration file and setting up the *sendmail* startup procedure on your ULTRIX-32 system

The day-to-day *sendmail* operations explained include:

- Use of the system log for records and debugging
- Mail queue processing
- Treatment of address aliases
- The mail-forwarding feature
- Special headers for return receipts and error situations

The article describes parameters you can adjust to tune *sendmail* to suit a specific site. If you must build your own configuration file, you will find the list of configuration file rules and hints to be helpful. And for expert system managers, the appendixes list detailed *sendmail* information in five categories:

- Command line flags
- Configuration options
- Mailer flags
- Compilation options (other configuration)
- Support files

Disc Quotas in a UNIX* Environment

Robert Elz

Department of Computer Science
University of Melbourne,
Parkville,
Victoria,
Australia.

1. Users' view of disc quotas

To most users, disc quotas will either be of no concern, or a fact of life that cannot be avoided. The *quota*(1) command will provide information on any disc quotas that may have been imposed upon a user.

There are two individual possible quotas that may be imposed, usually if one is, both will be. A limit can be set on the amount of space a user can occupy, and there may be a limit on the number of files (inodes) he can own.

Quota provides information on the quotas that have been set by the system administrators, in each of these areas, and current usage.

There are four numbers for each limit, the current usage, soft limit (quota), hard limit, and number of remaining login warnings. The soft limit is the number of 1K blocks (or files) that the user is expected to remain below. Each time the user's usage goes past this limit, he will be warned. The hard limit cannot be exceeded. If a user's usage reaches this number, further requests for space (or attempts to create a file) will fail with an EDQUOT error, and the first time this occurs, a message will be written to the user's terminal. Only one message will be output, until space occupied is reduced below the limit, and reaches it again, in order to avoid continual noise from those programs that ignore write errors.

Whenever a user logs in with a usage greater than his soft limit, he will be warned, and his login warning count decremented. When he logs in under quota, the counter is reset to its maximum value (which is a system configuration parameter, that is typically 3). If the warning count should ever reach zero (caused by three successive logins over quota), the particular limit that has been exceeded will be treated as if the hard limit has been reached, and no more resources will be allocated to the user. The **only** way to reset this condition is to reduce usage below quota, then log in again.

1.1. Surviving when quota limit is reached

In most cases, the only way to recover from over quota conditions, is to abort whatever activity was in progress on the filesystem that has reached its limit, remove sufficient files to bring the limit back below quota, and retry the failed program.

However, if you are in the editor and a write fails because of an over quota situation, that is not a suitable course of action, as it is most likely that initially attempting to write the file will have truncated its previous contents, so should the editor be aborted without correctly writing the file not only will the recent changes be lost, but possibly much, or even all, of the data that previously existed.

---

* UNIX is a trademark of Bell Laboratories.

There are several possible safe exits for a user caught in this situation. He may use the editor ! shell escape command to examine his file space, and remove surplus files. Alternatively, using *csh*, he may suspend the editor, remove some files, then resume it. A third possibility, is to write the file to some other filesystem (perhaps to a file on /tmp) where the user's quota has not been exceeded. Then after rectifying the quota situation, the file can be moved back to the filesystem it belongs on.

2. Administering the quota system

To set up and establish the disc quota system, there are several steps necessary to be performed by the system administrator.

First, the system must be configured to include the disc quota sub-system. This is done by including the line:

options QUOTA

in the system configuration file, then running *config*(8) followed by a system configuration*.

Second, a decision as to what filesystems need to have quotas applied needs to be made. Usually, only filesystems that house users' home directories, or other user files, will need to be subjected to the quota system, though it may also prove useful to also include **/usr**. If possible, **/tmp** should usually be free of quotas.

Having decided on which filesystems quotas need to be set upon, the administrator should then allocate the available space amongst the competing needs. How this should be done is (way) beyond the scope of this document.

Then, the *edquota*(8) command can be used to actually set the limits desired upon each user. Where a number of users are to be given the same quotas (a common occurrence) the **−p** switch to edquota will allow this to be easily accomplished.

Once the quotas are set, ready to operate, the system must be informed to enforce quotas on the desired filesystems. This is accomplished with the *quotaon*(8) command. *Quotaon* will either enable quotas for a particular filesystem, or with the **−a** switch, will enable quotas for each filesystem indicated in **/etc/fstab** as using quotas. See *fstab*(5) for details. Most sites using the quota system, will include the line

/etc/quotaon -a

in **/etc/rc.local**.

Should quotas need to be disabled, the *quotaoff*(8) command will do that, however, should the filesystem be about to be dismounted, the *umount*(8) command will disable quotas immediately before the filesystem is unmounted. This is actually an effect of the *umount*(2) system call, and it guarantees that the quota system will not be disabled if the umount would fail because the filesystem is not idle.

Periodically (certainly after each reboot, and when quotas are first enabled for a filesystem), the records retained in the quota file should be checked for consistency with the actual number of blocks and files allocated to the user. The *quotachk*(8) command can be used to accomplish this. It is not necessary to dismount the filesystem, or disable the quota system to run this command, though on active filesystems inaccurate results may occur. This does no real harm in most cases, another run of *quotachk* when the filesystem is idle will certainly correct any inaccuracy.

The super-user may use the *quota*(1) command to examine the usage and quotas of any user, and the *repquota*(8) command may be used to check the usages and limits for all users on a filesystem.

---

* See also the document "Building 4.2BSD UNIX Systems with Config".

3. Some implementation detail

Disc quota usage and information is stored in a file on the filesystem that the quotas are to be applied to. Conventionally, this file is **quotas** in the root of the filesystem. While this name is not known to the system in any way, several of the user level utilities "know" it, and choosing any other name would not be wise.

The data in the file comprises an array of structures, indexed by uid, one structure for each user on the system (whether the user has a quota on this filesystem or not). If the uid space is sparse, then the file may have holes in it, which would be lost by copying, so it is best to avoid this.

The system is informed of the existence of the quota file by the *setquota*(2) system call. It then reads the quota entries for each user currently active, then for any files open owned by users who are not currently active. Each subsequent open of a file on the filesystem, will be accompanied by a pairing with its quota information. In most cases this information will be retained in core, either because the user who owns the file is running some process, because other files are open owned by the same user, or because some file (perhaps this one) was recently accessed. In memory, the quota information is kept hashed by user-id and filesystem, and retained in an LRU chain so recently released data can be easily reclaimed. Information about those users whose last process has recently terminated is also retained in this way.

Each time a block is accessed or released, and each time an inode is allocated or freed, the quota system gets told about it, and in the case of allocations, gets the opportunity to object.

Measurements have shown that the quota code uses a very small percentage of the system cpu time consumed in writing a new block to disc.

4. Acknowledgments

The current disc quota system is loosely based upon a very early scheme implemented at the University of New South Wales, and Sydney University in the mid 70's. That system implemented a single combined limit for both files and blocks on all filesystems.

A later system was implemented at the University of Melbourne by the author, but was not kept highly accurately, eg: chown's (etc) did not affect quotas, nor did i/o to a file other than one owned by the instigator.

The current system has been running (with only minor modifications) since January 82 at Melbourne. It is actually just a small part of a much broader resource control scheme, which is capable of controlling almost anything that is usually uncontrolled in unix. The rest of this is, as yet, still in a state where it is far too subject to change to be considered for distribution.

For the 4.2BSD release, much work has been done to clean up and sanely incorporate the quota code by Sam Leffler and Kirk McKusick at The University of California at Berkeley.

Fsck – The UNIX† File System Check Program

Revised July 28, 1983

Marshall Kirk McKusick

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

This document reflects the use of *fsck* with the 4.2BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. *Fsck* runs in two modes. Normally it is run non-interactively by the system after a normal boot. When running in this mode, it will only make changes to the file system that are known to always be correct. If an unexpected inconsistency is found *fsck* will exit with a non-zero exit status, leaving the system running single-user. Typically the operator then runs *fsck* interactively. When running in this mode, each problem is listed followed by a suggested corrective action. The operator must decide whether or not the suggested correction should be made.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of deterministic corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

UNIX is a trademark of Bell Laboratories.

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

2. Overview of the file system

The file system is discussed in detail in [Mckusick83]; this section gives a brief overview.

2.1. Superblock

A file system is described by its *super-block*. The super-block is built when the file system is created (*newfs*(8)) and never changes. The super-block contains the basic parameters of the file system, such as the number of data blocks it contains and a count of the maximum number of files. Because the super-block contains critical data, *newfs* replicates it to protect against catastrophic loss. The *default super block* always resides at a fixed offset from the beginning of the file system's disk partition. The *redundant super blocks* are not referenced unless a head crash or other hard disk error causes the default super-block to be unusable. The redundant blocks are sprinkled throughout the disk partition.

Within the file system are files. Certain files are distinguished as directories and contain collections of pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps indicating modification and access times for the file, and an array of indices pointing to the data blocks for the file. In this section, we assume that the first 12 blocks of the file are directly referenced by values stored in the inode structure itself†. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 4096 byte block size, a singly indirect block contains 1024 further block addresses, a doubly indirect block contains 1024 addresses of further single indirect blocks, and a triply indirect block contains 1024 addresses of further doubly indirect blocks.

In order to create files with up to 2↑32 bytes, using only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block, so it is possible for file systems of different block sizes to be accessible simultaneously on the same system. The block size must be decided when *newfs* creates the file system; the block size cannot be subsequently changed without rebuilding the file system.

2.2. Summary information

Associated with the super block is non replicated *summary information*. The summary information changes as the file system is modified. The summary information contains the number of blocks, fragments, inodes and directories in the file system.

2.3. Cylinder groups

The file system partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Each cylinder group includes inode slots for files, a *block map* describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. A fixed number of inodes is allocated for each cylinder group when the file system is created. The current policy is to allocate one inode for each 2048 bytes of disk space; this is expected to be far more inodes than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for the $i+1$st cylinder group is about one track further from the beginning of the cylinder group than it was for the ith cylinder group. In this way, the redundant information spirals down into

†The actual number may vary from system to system, but is usually in the range 5-13.

the pack; any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information stores data.

2.4. Fragments

To avoid waste in storing small files, the file system space allocator divides a single file system block into one or more *fragments*. The fragmentation of the file system is specified when the file system is created; each file system block can be optionally broken into 2, 4, or 8 addressable fragments. The lower bound on the size of these fragments is constrained by the disk sector size; typically 512 bytes is the lower bound on fragment size. The block map associated with each cylinder group records the space availability at the fragment level. Aligned fragments are examined to determine block availability.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. For example, consider an 11000 byte file stored on a 4096/1024 byte file system. This file uses two full size blocks and a 3072 byte fragment. If no fragments with at least 3072 bytes are available when the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file, as needed.

2.5. Updates to the file system

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. The file system stages all modifications of critical information; modification can either be completed or cleanly backed out after a crash. Knowing the information that is first written to the file system, deterministic procedures can be developed to repair a corrupted file system. To understand this process, the order that the update requests were being honored must first be understood.

When a user program does an operation to change the file system, such as a *write*, the data to be written is copied into an internal *in-core* buffer in the kernel. Normally, the disk update is handled asynchronously; the user process is allowed to proceed even though the data has not yet been written to the disk. The data, along with the inode information reflecting the change, is eventually written out to disk. The real disk write may not happen until long after the *write* system call has returned. Thus at any given time, the file system, as it resides on the disk, lags the state of the file system represented by the in-core information.

The disk information is updated to reflect the in-core information when the buffer is required for another use, when a *sync*(2) is done (at 30 second intervals) by */etc/update*(8), or by manual operator intervention with the *sync*(8) command. If the system is halted without writing out the in-core information, the file system on the disk will be in an inconsistent state.

If all updates are done asynchronously, several serious inconsistencies can arise. One inconsistency is that a block may be claimed by two inodes. Such an inconsistency can occur when the system is halted before the pointer to the block in the old inode has been cleared in the copy of the old inode on the disk, and after the pointer to the block in the new inode has been written out to the copy of the new inode on the disk. Here, there is no deterministic method for deciding which inode should really claim the block. A similar problem can arise with a multiply claimed inode.

The problem with asynchronous inode updates can be avoided by doing all inode deallocations synchronously. Consequently, inodes and indirect blocks are written to the disk synchronously (*i.e.* the process blocks until the information is really written to disk) when they are being deallocated. Similarly inodes are kept consistent by synchronously deleting, adding, or changing directory entries.

3. Fixing corrupted file systems

A file system can become corrupted in several ways. The most common of these ways are improper shutdown procedures and hardware failures.

File systems may become corrupted during an *unclean halt*. This happens when proper shutdown procedures are not observed, physically write-protecting a mounted file system, or a mounted file system is taken off-line. The most common operator procedural failure is forgetting to *sync* the system before halting the CPU.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

3.1. Detecting and correcting corruption

Normally *fsck* is run non-interactively. In this mode it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this paper we assume that *fsck* is being run interactively, and all possible errors can be encountered. When an inconsistency is discovered in this mode, *fsck* reports the inconsistency for the operator to chose a corrective action.

A quiescent‡ file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system, or computed from other known values. The file system **must** be in a quiescent state when *fsck* is run, since *fsck* is a multi-pass program.

In the following sections, we discuss methods to discover inconsistencies and possible corrective actions for the cylinder group blocks, the inodes, the indirect blocks, and the data blocks containing directory entries.

3.2. Super-block checking

The most commonly corrupted item in a file system is the summary information associated with the super-block. The summary information is prone to corruption because it is modified with every change to the file system's blocks or inodes, and is usually corrupted after an unclean halt.

The super-block is checked for inconsistencies involving file-system size, number of inodes, free-block count, and the free-inode count. The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The file-system size and layout information are the most critical pieces of information for *fsck*. While there is no way to actually check these sizes, since they are statically determined by *newfs*, *fsck* can check that these sizes are within reasonable bounds. All other file system checks require that these sizes be correct. If *fsck* detects corruption in the static parameters of the default super-block, *fsck* requests the operator to specify the location of an alternate super-block.

3.3. Free block checking

Fsck checks that all the blocks marked as free in the cylinder group block maps are not claimed by any files. When all the blocks have been initially accounted for, *fsck* checks that the number of free blocks plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

---

‡ I.e., unmounted and not being written on.

If anything is wrong with the block allocation maps, *fsck* will rebuild them, based on the list it has computed of allocated blocks.

The summary information associated with the super-block counts the total number of free blocks within the file system. *Fsck* compares this count to the number of free blocks it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-block count.

The summary information counts the total number of free inodes within the file system. *Fsck* compares this count to the number of free inodes it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-inode count.

3.4. Checking the inode state

An individual inode is not as likely to be corrupted as the allocation information. However, because of the great number of active inodes, a few of the inodes are usually corrupted.

The list of inodes in the file system is checked sequentially starting with inode 2 (inode 0 marks unused inodes; inode 1 is saved for future generations) and progressing through the last inode in the file system. The state of each inode is checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes must be one of six types: regular inode, directory inode, symbolic link inode, special block inode, special character inode, or socket inode. Inodes may be found in one of three allocation states: unallocated, allocated, and neither unallocated nor allocated. This last state suggests an incorrectly formated inode. An inode can get in this state if bad data is written into the inode list. The only possible corrective action is for *fsck* is to clear the inode.

3.5. Inode links

Each inode counts the total number of directory entries linked to the inode. *Fsck* verifies the link count of each inode by starting at the root of the file system, and descending through the directory structure. The actual link count for each inode is calculated during the descent.

If the stored link count is non-zero and the actual link count is zero, then no directory entry appears for the inode. If this happens, *fsck* will place the disconnected file in the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated. If this happens, *fsck* replaces the incorrect stored link count by the actual link count.

Each inode contains a list, or pointers to lists (indirect blocks), of all the blocks claimed by the inode. Since indirect blocks are owned by an inode, inconsistencies in indirect blocks directly affect the inode that owns it.

Fsck compares each block number claimed by an inode against a list of already allocated blocks. If another inode already claims a block number, then the block number is added to a list of *duplicate blocks*. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, *fsck* will perform a partial second pass over the inode list to find the inode of the duplicated block. The second pass is needed, since without examining the files associated with these inodes for correct content, not enough information is available to determine which inode is corrupted and should be cleared. If this condition does arise (only hardware failure will cause it), then the inode with the earliest modify time is usually incorrect, and should be cleared. If this happens, *fsck* prompts the operator to clear both inodes. The operator must decide which one should be kept and which one should be cleared.

Fsck checks the range of each block number claimed by an inode. If the block number is lower than the first data block in the file system, or greater than the last data block, then

the block number is a *bad block number*. Many bad blocks in an inode are usually caused by an indirect block that was not written to the file system, a condition which can only occur if there has been a hardware failure. If an inode contains bad block numbers, *fsck* prompts the operator to clear it.

3.6. Inode data size

Each inode contains a count of the number of data blocks that it contains. The number of actual data blocks is the sum of the allocated data blocks and the indirect blocks. *Fsck* computes the actual number of data blocks and compares that block count against the actual number of blocks the inode claims. If an inode contains an incorrect count *fsck* prompts the operator to fix it.

Each inode contains a thirty-two bit size field. The size is the number of data bytes in the file associated with the inode. The consistency of the byte size field is roughly checked by computing from the size field the maximum number of blocks that should be associated with the inode, and comparing that expected block count against the actual number of blocks the inode claims.

3.7. Checking the data associated with an inode

An inode can directly or indirectly reference three kinds of data blocks. All referenced blocks must be the same kind. The three types of data blocks are: plain data blocks, symbolic link data blocks, and directory data blocks. Plain data blocks contain the information stored in a file; symbolic link data blocks contain the path name stored in a link. Directory data blocks contain directory entries. *Fsck* can only check the validity of directory data blocks.

Each directory data block is checked for several types of inconsistencies. These inconsistencies include directory inode numbers pointing to unallocated inodes, directory inode numbers that are greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories that are not attached to the file system. If the inode number in a directory data block references an unallocated inode, then *fsck* will remove that directory entry. Again, this condition can only arise when there has been a hardware failure.

If a directory entry inode number references outside the inode list, then *fsck* will remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." must be the first entry in the directory data block. The inode number for "." must reference itself; e.g., it must equal the inode number for the directory data block. The directory inode number entry for ".." must be the second entry in the directory data block. Its value must equal the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory). If the directory inode numbers are incorrect, *fsck* will replace them with the correct values.

3.8. File system connectivity

Fsck checks the general connectivity of the file system. If directories are not linked into the file system, then *fsck* links the directory back into the file system in the *lost+found* directory. This condition only occurs when there has been a hardware failure.

Acknowledgements

I thank Bill Joy, Sam Leffler, Robert Elz and Dennis Ritchie for their suggestions and help in implementing the new file system. Thanks also to Robert Henry for his editorial input to get this document together. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235. (Kirk McKusick, July 1983)

I would like to thank Larry A. Wehr for advice that lead to the first version of *fsck* and Rick B. Brandt for adapting *fsck* to UNIX/TS. (T. Kowalski, July 1979)

References

[Dolotta78] Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1* (January 1978).

[Joy83] Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, M., and Mosher, D. *4.2BSD System Manual*, University of California at Berkeley, Computer Systems Research Group Technical Report #4, 1982.

[McKusick83] McKusick, M., Joy, W., Leffler, S., and Fabry, R. *A Fast File System for UNIX*, University of California at Berkeley, Computer Systems Research Group Technical Report #7, 1982.

[Ritchie78] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1905-29.

[Thompson78] Thompson, K., UNIX Implementation, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1931-46.

4. Appendix A – Fsck Error Conditions

4.1. Conventions

Fsck is a multi-pass file system check program. Each file system pass invokes a different Phase of the *fsck* program. After the initial setup, *fsck* performs successive Phases over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the map of free blocks, (possibly rebuilding it), and performs some cleanup.

Normally *fsck* is run non-interactively to *preen* the file systems after an unclean halt. While preen'ing a file system, it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this appendix many errors have several options that the operator can take. When an inconsistency is detected, *fsck* reports the error condition to the operator. If a response is required, *fsck* prints a prompt message and waits for a response. When preen'ing most errors are fatal. For those that are expected, the response taken is noted. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the *Phase* of the *fsck* program in which they can occur. The error conditions that may occur in more than one Phase will be discussed in initialization.

4.2. Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. This section lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file. All of the initialization errors are fatal when the file system is being preen'ed.

C option?
C is not a legal option to *fsck*; legal options are −b, −y, −n, and −p. *Fsck* terminates on this error condition. See the *fsck*(8) manual entry for further detail.

cannot alloc NNN bytes for blockmap
cannot alloc NNN bytes for freemap
cannot alloc NNN bytes for statemap
cannot alloc NNN bytes for lncntp
Fsck's request for memory for its virtual memory tables failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't open checklist file: *F*
The file system checklist file *F* (usually */etc/fstab*) can not be opened for reading. *Fsck* terminates on this error condition. Check access modes of *F*.

Can't stat root
Fsck's request for statistics about the root directory "/" failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't stat *F*
Can't make sense out of name *F*
Fsck's request for statistics about the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.

Can't open *F*
Fsck's request attempt to open the file system *F* failed. When running manually, it ignores

this file system and continues checking the next file system given. Check access modes of *F*.

***F*: (NO WRITE)**
Either the −n flag was specified or *fsck*'s attempt to open the file system *F* for writing failed. When running manually, all the diagnostics are printed out, but no modifications are attempted to fix them.

file is not a block or character device; OK
You have given *fsck* a regular file name by mistake. Check the type of the file specified.

Possible responses to the OK prompt are:

YES Ignore this error condition.

NO ignore this file system and continues checking the next file system given.

One of the following messages will appear:
MAGIC NUMBER WRONG
NCG OUT OF RANGE
CPG OUT OF RANGE
NCYL DOES NOT JIVE WITH NCG*CPG
SIZE PREPOSTEROUSLY LARGE
TRASHED VALUES IN SUPER BLOCK

and will be followed by the message:
F*: BAD SUPER BLOCK: *B
USE -b OPTION TO FSCK TO SPECIFY LOCATION OF AN ALTERNATE SUPER-BLOCK TO SUPPLY NEEDED INFORMATION; SEE fsck(8).
The super block has been corrupted. An alternative super block must be selected from among those listed by *newfs* (8) when the file system was created. For file systems with a blocksize less than 32K, specifying −b 32 is a good first choice.

INTERNAL INCONSISTENCY: *M*
Fsck's has had an internal panic, whose message is specified as *M*. This should never happen. See a guru.

CAN NOT SEEK: BLK *B* (CONTINUE)
Fsck's request for moving to a specified block number *B* in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT READ: BLK *B* (CONTINUE)
Fsck's request for reading a specified block number *B* in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT WRITE: BLK *B* (CONTINUE)
Fsck's request for writing a specified block number *B* in the file system failed. The disk is write-protected. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

4.3. Phase 1 – Check Blocks and Sizes

This phase concerns itself with the inode list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format. All errors in this phase except **INCORRECT BLOCK COUNT** are fatal if the file system is being preen'ed,

CG *C*: BAD MAGIC NUMBER The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group maps have been destroyed. When running manually the cylinder group is marked as needing to be reconstructed.

UNKNOWN FILE TYPE I=*I* (CLEAR) The mode word of the inode *I* indicates that the inode is not a special block inode, special character inode, socket inode, regular inode, symbolic link, or directory inode.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents. This will always invoke the UNALLOCATED error condition in Phase 2 for each directory entry pointing to this inode.

NO ignore this error condition.

LINK COUNT TABLE OVERFLOW (CONTINUE)
An internal table for *fsck* containing allocated inodes with a link count of zero has no more room. Recompile *fsck* with a larger value of MAXLNCNT.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO terminate the program.

B* BAD I=*I
Inode *I* contains block number *B* with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the **EXCESSIVE BAD BLKS** error condition in Phase 1 if inode *I* has too many block numbers outside the file system range. This error condition will always invoke the **BAD/DUP** error condition in Phase 2 and Phase 4.

EXCESSIVE BAD BLKS I=*I* (CONTINUE)
There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in

the file system associated with inode *I*.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO terminate the program.

B **DUP I=***I*

Inode *I* contains block number *B* which is already claimed by another inode. This error condition may invoke the **EXCESSIVE DUP BLKS** error condition in Phase 1 if inode *I* has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the **BAD/DUP** error condition in Phase 2 and Phase 4.

EXCESSIVE DUP BLKS I=*I* (CONTINUE)

There is more than a tolerable number (usually 10) of blocks claimed by other inodes.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO terminate the program.

DUP TABLE OVERFLOW (CONTINUE)

An internal table in *fsck* containing duplicate block numbers has no more room. Recompile *fsck* with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.

NO terminate the program.

PARTIALLY ALLOCATED INODE I=*I* (CLEAR)

Inode *I* is neither allocated nor unallocated.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

INCORRECT BLOCK COUNT I=*I* (*X* should be *Y*) (CORRECT)

The block count for inode *I* is *X* blocks, but should be *Y* blocks. When preen'ing the count is corrected.

Possible responses to the CORRECT prompt are:

YES replace the block count of inode *I* with *Y*.

NO ignore this error condition.

4.4. Phase 1B: Rescan for More Dups

When a duplicate block is found in the file system, the file system is rescanned to find the inode which previously claimed that block. This section lists the error condition when the duplicate block is found.

B* DUP I=*I
Inode *I* contains block number *B* that is already claimed by another inode. This error condition will always invoke the **BAD/DUP** error condition in Phase 2. You can determine which inodes have overlapping blocks by examining this error condition and the DUP error condition in Phase 1.

4.5. Phase 2 – Check Pathnames

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes. All errors in this phase are fatal if the file system is being preen'ed.

ROOT INODE UNALLOCATED. TERMINATING.
The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program will terminate.

NAME TOO LONG *F*
An excessively long path name has been found. This is usually indicative of loops in the file system name space. This can occur if the super user has made circular links to directories. The offending links must be removed (by a guru).

ROOT INODE NOT DIRECTORY (FIX)
The root inode (usually inode number 2) is not directory inode type.

Possible responses to the FIX prompt are:

YES replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a VERY large number of error conditions will be produced.

NO terminate the program.

DUPS/BAD IN ROOT INODE (CONTINUE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system.

Possible responses to the CONTINUE prompt are:

YES ignore the **DUPS/BAD** error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, then this may result in a large number of other error conditions.

NO terminate the program.

I OUT OF RANGE I=*I* NAME=*F* (REMOVE)
A directory entry *F* has an inode number *I* which is greater than the end of the inode list.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

UNALLOCATED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)
A directory entry *F* has a directory inode *I* without allocate mode bits. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

UNALLOCATED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* FILE=*F* (REMOVE)
A directory entry *F* has an inode *I* without allocate mode bits. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

DUP/BAD I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, directory inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

DUP/BAD I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* FILE=*F* (REMOVE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

ZERO LENGTH DIRECTORY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)
A directory entry *F* has a size *S* that is zero. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed; this will always invoke the BAD/DUP error condition in Phase 4.

NO ignore this error condition.

DIRECTORY TOO SHORT I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *F* has been found whose size *S* is less than the minimum size directory. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the FIX prompt are:

YES increase the size of the directory to the minimum directory size.

NO ignore this directory.

DIRECTORY CORRUPTED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (SALVAGE)
A directory with an inconsistent internal state has been found.

Possible responses to the FIX prompt are:

YES throw away all entries up to the next 512-byte boundary. This rather drastic action can throw away up to 42 entries, and should be taken only after other recovery efforts have failed.

NO Skip up to the next 512-byte boundary and resume reading, but do not modify the directory.

BAD INODE NUMBER FOR '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose inode number for '.' does does not equal *I*.

Possible responses to the FIX prompt are:

YES change the inode number for '.' to be equal to *I*.

NO leave the inode number for '.' unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose first entry is unallocated.

Possible responses to the FIX prompt are:

YES make an entry for '.' with inode number equal to *I*.

NO leave the directory unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, FIRST ENTRY IN DIRECTORY CONTAINS *F*
A directory *I* has been found whose first entry is *F*. *Fsck* cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, INSUFFICIENT SPACE TO ADD '.'
A directory *I* has been found whose first entry is not '.'. *Fsck* cannot resolve this problem as it should never happen. See a guru.

EXTRA '.' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '.'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '.'.

NO leave the directory unchanged.

BAD INODE NUMBER FOR '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose inode number for '..' does does not equal the parent of *I*.

Possible responses to the FIX prompt are:

YES change the inode number for '..' to be equal to the parent of *I*.

NO leave the inode number for '..' unchanged.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose second entry is unallocated.

Possible responses to the FIX prompt are:

YES make an entry for '..' with inode number equal to the parent of *I*.

NO leave the directory unchanged.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, SECOND ENTRY IN DIRECTORY CONTAINS *F*
A directory *I* has been found whose second entry is *F*. *Fsck* cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, INSUFFICIENT SPACE TO ADD '..'
A directory *I* has been found whose second entry is not '..'. *Fsck* cannot resolve this problem as it should never happen. See a guru.

EXTRA '..' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '..'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '..'.

NO leave the directory unchanged.

4.6. Phase 3 – Check Connectivity

This phase concerns itself with the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full *lost+found* directories.

UNREF DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)
The directory inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. When preen'ing, the directory is reconnected if its size is non-zero, otherwise it is cleared.

Possible responses to the RECONNECT prompt are:

YES reconnect directory inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 3 if there are problems connecting directory inode *I* to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.

NO ignore this error condition. This will always invoke the UNREF error condition in Phase 4.

SORRY. NO lost+found DIRECTORY
There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See *fsck*(8) manual entry for further detail. This error is fatal if the file system is being preen'ed.

SORRY. NO SPACE IN lost+found DIRECTORY
There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make *lost+found* larger. See *fsck*(8) manual entry for further detail. This error is fatal if the file system is being preen'ed.

DIR I=*I1* CONNECTED. PARENT WAS I=*I2*
This is an advisory message indicating a directory inode *I1* was successfully connected to the *lost+found* directory. The parent inode *I2* of the directory inode *I1* is replaced by the inode number of the *lost+found* directory.

4.7. Phase 4 – Check Reference Counts

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full *lost+found* directory, incorrect link counts for files, directories, symbolic links, or special files, unreferenced files, symbolic links, and directories, bad and duplicate blocks in files, symbolic links, and directories, and incorrect total free-inode counts. All errors in this phase are correctable if the file system is being preen'ed except running out of space in the lost+found directory.

UNREF FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)
Inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing the file is cleared if either its size or its link count is zero, otherwise it is reconnected.

Possible responses to the RECONNECT prompt are:

YES reconnect inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 4 if there are problems connecting inode *I* to *lost+found*.

NO ignore this error condition. This will always invoke the CLEAR error condition in Phase 4.

(CLEAR)
The inode mentioned in the immediately previous error condition can not be reconnected. This cannot occur if the file system is being preen'ed, since lack of space to reconnect files is a fatal error.

Possible responses to the CLEAR prompt are:

YES de-allocate the inode mentioned in the immediately previous error condition by zeroing its contents.

NO ignore this error condition.

SORRY. NO lost+found DIRECTORY
There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check access modes of *lost+found*. This error is fatal if the file system is being preen'ed.

SORRY. NO SPACE IN lost+found DIRECTORY
There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found*. This error is fatal if the file system is being preen'ed.

LINK COUNT FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for inode *I* which is a file, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of file inode *I* with *Y*.

NO ignore this error condition.

LINK COUNT DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for inode *I* which is a directory, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of directory inode *I* with *Y*.

NO ignore this error condition.

LINK COUNT *F* I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for *F* inode *I* is *X* but should be *Y*. The name *F*, owner *O*, mode *M*, size *S*, and modify time *T* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of inode *I* with *Y*.

NO ignore this error condition.

UNREF FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Inode *I* which is a file, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing, this is a file that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

UNREF DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Inode *I* which is a directory, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing, this is a directory that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

BAD/DUP FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. This error cannot arise when the file system is being preen'ed, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

BAD/DUP DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory inode

I. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. This error cannot arise when the file system is being preen'ed, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

FREE INODE COUNT WRONG IN SUPERBLK (FIX)
The actual count of the free inodes does not match the count in the super-block of the file system. When preen'ing, the count is fixed.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.

NO ignore this error condition.

4.8. Phase 5 - Check Cyl groups

This phase concerns itself with the free-block maps. This section lists error conditions resulting from allocated blocks in the free-block maps, free blocks missing from free-block maps, and the total free-block count incorrect.

CG *C*: BAD MAGIC NUMBER
The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group maps have been destroyed. When running manually the cylinder group is marked as needing to be reconstructed. This error is fatal if the file system is being preen'ed.

EXCESSIVE BAD BLKS IN BIT MAPS (CONTINUE)
An inode contains more than a tolerable number (usually 10) of blocks claimed by other inodes or that are out of the legal range for the file system. This error is fatal if the file system is being preen'ed.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the free-block maps and continue the execution of *fsck*.

NO terminate the program.

SUMMARY INFORMATION *T* BAD
where *T* is one or more of:
(INODE FREE)
(BLOCK OFFSETS)
(FRAG SUMMARIES)
(SUPER BLOCK SUMMARIES)
The indicated summary information was found to be incorrect. This error condition will always invoke the **BAD CYLINDER GROUPS** condition in Phase 6. When preen'ing, the summary information is recomputed.

***X* BLK(S) MISSING**
X blocks unused by the file system were not found in the free-block maps. This error condition will always invoke the **BAD CYLINDER GROUPS** condition in Phase 6. When preen'ing, the block maps are rebuilt.

BAD CYLINDER GROUPS (SALVAGE)
Phase 5 has found bad blocks in the free-block maps, duplicate blocks in the free-block maps, or blocks missing from the file system. When preen'ing, the cylinder groups are reconstructed.

Possible responses to the SALVAGE prompt are:

YES replace the actual free-block maps with a new free-block maps.

NO ignore this error condition.

FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)
The actual count of free blocks does not match the count in the super-block of the file system. When preen'ing, the counts are fixed.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.

NO ignore this error condition.

4.9. Phase 6 - Salvage Cylinder Groups

This phase concerns itself with the free-block maps reconstruction. No error messages are produced.

4.10. Cleanup

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

***V* files, *W* used, *X* free (*Y* frags, *Z* blocks)**
This is an advisory message indicating that the file system checked contained *V* files using *W* fragment sized blocks leaving *X* fragment sized blocks free in the file system. The numbers in parenthesis breaks the free count down into *Y* free fragments and *Z* free full sized blocks.

******* REBOOT UNIX *******
This is an advisory message indicating that the root file system has been modified by *fsck*. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps. When preen'ing, *fsck* will exit with a code of 4. The auto-reboot script interprets an exit code of 4 by issuing a reboot system call.

******* FILE SYSTEM WAS MODIFIED *******
This is an advisory message indicating that the current file system was modified by *fsck*. If this file system is mounted or is the current root file system, *fsck* should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps.

SENDMAIL

INSTALLATION AND OPERATION GUIDE

Eric Allman
Britton-Lee, Inc.

Version 4.2

Sendmail implements a general purpose internetwork mail routing facility under the UNIX* operating system. It is not tied to any one transport protocol – its function may be likened to a crossbar switch, relaying messages from one domain into another. In the process, it can do a limited amount of message header editing to put the message into a format that is appropriate for the receiving domain. All of this is done under the control of a configuration file.

Due to the requirements of flexibility for *sendmail*, the configuration file can seem somewhat unapproachable. However, there are only a few basic configurations for most sites, for which standard configuration files have been supplied. Most other configurations can be built by adjusting an existing configuration files incrementally.

Although *sendmail* is intended to run without the need for monitoring, it has a number of features that may be used to monitor or adjust the operation under unusual circumstances. These features are described.

Section one describes how to do a basic *sendmail* installation. Section two explains the day-to-day information you should know to maintain your mail system. If you have a relatively normal site, these two sections should contain sufficient information for you to install *sendmail* and keep it happy. Section three describes some parameters that may be safely tweaked. Section four has information regarding the command line arguments. Section five contains the nitty-gritty information about the configuration file. This section is for masochists and people who must write their own configuration file. The appendixes give a brief but detailed explanation of a number of features not described in the rest of the paper.

The references in this paper are actually found in the companion paper *Sendmail – An Internetwork Mail Router*. This other paper should be read before this manual to gain a basic understanding of how the pieces fit together.

1. BASIC INSTALLATION

There are two basic steps to installing sendmail. The hard part is to build the configuration table. This is a file that sendmail reads when it starts up that describes the mailers it knows about, how to parse addresses, how to rewrite the message header, and the settings of various options. Although the configuration table is quite complex, a configuration can usually be built by adjusting an existing off-the-shelf configuration. The second part is actually doing the installation, i.e., creating the necessary files, etc.

The remainder of this section will describe the installation of sendmail assuming you can use one of the existing configurations and that the standard installation parameters are

*UNIX is a trademark of Bell Laboratories.

acceptable. All pathnames and examples are given from the root of the *sendmail* subtree.

1.1. Off-The-Shelf Configurations

The configuration files are all in the subdirectory *cf* of the sendmail directory. The ones used at Berkeley are in *m4*(1) format; files with names ending ".m4" are *m4* include files, while files with names ending ".mc" are the master files. Files with names ending ".cf" are the *m4* processed versions of the corresponding ".mc" file.

Two off the shelf configuration files are supplied to handle the basic cases: *cf/arpaproto.cf* for Arpanet (TCP) sites and *cf/uucpproto.cf* for UUCP sites. These are *not* in *m4* format. The file you need should be copied to a file with the same name as your system, e.g.,

```
cp uucpproto.cf ucsfcgl.cf
```

This file is now ready for installation as */usr/lib/sendmail.cf*.

1.2. Installation Using the Makefile

A makefile exists in the root of the *sendmail* directory that will do all of these steps for a 4.2BSD system. It may have to be slightly tailored for use on other systems.

Before using this makefile, you should already have created your configuration file and left it in the file "cf/*system*.cf" where *system* is the name of your system (i.e., what is returned by *hostname*(1)). If you do not have *hostname* you can use the declaration "HOST=*system*" on the *make*(1) command line. You should also examine the file *md/config.m4* and change the *m4* macros there to reflect any libraries and compilation flags you may need.

The basic installation procedure is to type:

```
make
make install
```

in the root directory of the *sendmail* distribution. This will make all binaries and install them in the standard places. The second *make* command must be executed as the superuser (root).

1.3. Installation by Hand

Along with building a configuration file, you will have to install the *sendmail* startup into your UNIX system. If you are doing this installation in conjunction with a regular Berkeley UNIX install, these steps will already be complete. Many of these steps will have to be executed as the superuser (root).

1.3.1. lib/libsys.a

The library in lib/libsys.a contains some routines that should in some sense be part of the system library. These are the system logging routines and the new directory access routines (if required). If you are not running the new 4.2BSD directory code and do not have the compatibility routines installed in your system library, you should execute the commands:

```
cd lib
make ndir
```

This will compile and install the 4.2 compatibility routines in the library. You should then type:

```
cd lib      # if required
make
```

This will recompile and fill the library.

1.3.2. /usr/lib/sendmail

The binary for sendmail is located in /usr/lib. There is a version available in the source directory that is probably inadequate for your system. You should plan on recompiling and installing the entire system:

```
cd src
rm -f *.o
make
cp sendmail /usr/lib
```

1.3.3. /usr/lib/sendmail.cf

The configuration file that you created earlier should be installed in /usr/lib/sendmail.cf:

```
cp cf/system.cf /usr/lib/sendmail.cf
```

1.3.4. /usr/ucb/newaliases

If you are running delivermail, it is critical that the *newaliases* command be replaced. This can just be a link to *sendmail*:

```
rm -f /usr/ucb/newaliases
ln /usr/lib/sendmail /usr/ucb/newaliases
```

1.3.5. /usr/lib/sendmail.cf

The configuration file must be installed in /usr/lib. This is described above.

1.3.6. /usr/spool/mqueue

The directory */usr/spool/mqueue* should be created to hold the mail queue. This directory should be mode 777 unless *sendmail* is run setuid, when *mqueue* should be owned by the sendmail owner and mode 755.

1.3.7. /usr/lib/aliases*

The system aliases are held in three files. The file "/usr/lib/aliases" is the master copy. A sample is given in "lib/aliases" which includes some aliases which *must* be defined:

```
cp lib/aliases /usr/lib/aliases
```

You should extend this file with any aliases that are apropos to your system.

Normally *sendmail* looks at a version of these files maintained by the *dbm*(3) routines. These are stored in "/usr/lib/aliases.dir" and "/usr/lib/aliases.pag." These can initially be created as empty files, but they will have to be initialized promptly. These should be mode 666 if you are running a reasonably relaxed system:

```
cp /dev/null /usr/lib/aliases.dir
cp /dev/null /usr/lib/aliases.pag
chmod 666 /usr/lib/aliases.*
newaliases
```

1.3.8. /usr/lib/sendmail.fc

If you intend to install the frozen version of the configuration file (for quick startup) you should create the file /usr/lib/sendmail.fc and initialize it. This step may be safely skipped.

```
cp /dev/null /usr/lib/sendmail.fc
/usr/lib/sendmail -bz
```

1.3.9. /etc/rc

It will be necessary to start up the sendmail daemon when your system reboots. This daemon performs two functions: it listens on the SMTP socket for connections (to receive mail from a remote system) and it processes the queue periodically to insure that mail gets delivered when hosts come up.

Add the following lines to "/etc/rc" (or "/etc/rc.local" as appropriate) in the area where it is starting up the daemons:

```
if [ -f /usr/lib/sendmail ]; then
        (cd /usr/spool/mqueue; rm -f [lnx]f*)
        /usr/lib/sendmail -bd -q30m &
        echo -n ' sendmail' >/dev/console
fi
```

The "cd" and "rm" commands insure that all lock files have been removed; extraneous lock files may be left around if the system goes down in the middle of processing a message. The line that actually invokes *sendmail* has two flags: "-bd" causes it to listen on the SMTP port, and "-q30m" causes it to run the queue every half hour.

If you are not running a version of UNIX that supports Berkeley TCP/IP, do not include the **-bd** flag.

1.3.10. /usr/lib/sendmail.hf

This is the help file used by the SMTP **HELP** command. It should be copied from "lib/sendmail.hf":

```
cp lib/sendmail.hf /usr/lib
```

1.3.11. /usr/lib/sendmail.st

If you wish to collect statistics about your mail traffic, you should create the file "/usr/lib/sendmail.st":

```
cp /dev/null /usr/lib/sendmail.st
chmod 666 /usr/lib/sendmail.st
```

This file does not grow. It is printed with the program "aux/mailstats."

1.3.12. /etc/syslog

You may want to run the *syslog* program (to collect log information about sendmail). This program normally resides in */etc/syslog*, with support files */etc/syslog.conf* and */etc/syslog.pid*. The program is located in the *aux* subdirectory of the *sendmail* distribution. The file */etc/syslog.conf* describes the file(s) that sendmail will log in. For a complete description of syslog, see the manual page for *syslog*(8) (located in *sendmail/doc* on the distribution).

1.3.13. /usr/ucb/newaliases

If *sendmail* is invoked as "newaliases," it will simulate the **−bi** flag (i.e., will rebuild the alias database; see below). This should be a link to /usr/lib/sendmail.

1.3.14. /usr/ucb/mailq

If *sendmail* is invoked as "mailq," it will simulate the **−bp** flag (i.e., *sendmail* will print the contents of the mail queue; see below). This should be a link to /usr/lib/sendmail.

2. NORMAL OPERATIONS

2.1. Quick Configuration Startup

A fast version of the configuration file may be set up by using the **−bz** flag:

/usr/lib/sendmail −bz

This creates the file */usr/lib/sendmail.fc* ("frozen configuration"). This file is an image of *sendmail*'s data space after reading in the configuration file. If this file exists, it is used instead of */usr/lib/sendmail.cf* *sendmail.fc* must be rebuilt manually every time *sendmail.cf* is changed.

The frozen configuration file will be ignored if a **−C** flag is specified or if sendmail detects that it is out of date. However, the heuristics are not strong so this should not be trusted.

2.2. The System Log

The system log is supported by the *syslog*(8) program.

2.2.1. Format

Each line in the system log consists of a timestamp, the name of the machine that generated it (for logging from several machines over the ethernet), the word "sendmail:", and a message.

2.2.2. Levels

If you have *syslog*(8) or an equivalent installed, you will be able to do logging. There is a large amount of information that can be logged. The log is arranged as a succession of levels. At the lowest level only extremely strange situations are logged. At the highest level, even the most mundane and uninteresting events are recorded for posterity. As a convention, log levels under ten are considered "useful;" log levels above ten are usually for debugging purposes.

A complete description of the log levels is given in section 4.3.

2.3. The Mail Queue

The mail queue should be processed transparently. However, you may find that manual intervention is sometimes necessary. For example, if a major host is down for a period of time the queue may become clogged. Although sendmail ought to recover gracefully when the host comes up, you may find performance unacceptably bad in the meantime.

2.3.1. Printing the queue

The contents of the queue can be printed using the *mailq* command (or by specifying the **−bp** flag to sendmail):

mailq

This will produce a listing of the queue id's, the size of the message, the date the message entered the queue, and the sender and recipients.

2.3.2. Format of queue files

All queue files have the form *x***f***AA99999* where *AA99999* is the *id* for this file and the *x* is a type. The types are:

d The data file. The message body (excluding the header) is kept in this file.

l The lock file. If this file exists, the job is currently being processed, and a queue run will not process the file. For that reason, an extraneous **lf** file can cause a job to apparently disappear (it will not even time out!).

n This file is created when an id is being created. It is a separate file to insure that no mail can ever be destroyed due to a race condition. It should exist for no more than a few milliseconds at any given time.

q The queue control file. This file contains the information necessary to process the job.

t A temporary file. These are an image of the **qf** file when it is being rebuilt. It should be renamed to a **qf** file very quickly.

x A transcript file, existing during the life of a session showing everything that happens during that session.

The **qf** file is structured as a series of lines each beginning with a code letter. The lines are as follows:

D The name of the data file. There may only be one of these lines.

H A header definition. There may be any number of these lines. The order is important: they represent the order in the final message. These use the same syntax as header definitions in the configuration file.

R A recipient address. This will normally be completely aliased, but is actually realiased when the job is processed. There will be one line for each recipient.

S The sender address. There may only be one of these lines.

T The job creation time. This is used to compute when to time out the job.

P The current message priority. This is used to order the queue. Higher numbers mean lower priorities. The priority increases as the message sits in the queue. The initial priority depends on the message class and the size of the message.

M A message. This line is printed by the *mailq* command, and is generally used to store status information. It can contain any text.

As an example, the following is a queue file sent to "mckusick@calder" and "wnj":

```
DdfA13557
Seric
T404261372
P132
Rmckusick@calder
Rwnj
H?D?date: 23-Oct-82 15:49:32-PDT (Sat)
H?F?from: eric (Eric Allman)
H?x?full-name: Eric Allman
Hsubject: this is an example message
Hmessage-id: <8209232249.13557@UCBARPA.BERKELEY.ARPA>
Hreceived: by UCBARPA.BERKELEY.ARPA (3.227 [10/22/82])
        id A13557; 23-Oct-82 15:49:32-PDT (Sat)
Hphone: (415) 548-3211
HTo: mckusick@calder, wnj
```

This shows the name of the data file, the person who sent the message, the submission time (in seconds since January 1, 1970), the message priority, the message class, the recipients, and the headers for the message.

2.3.3. Forcing the queue

Sendmail should run the queue automatically at intervals. The algorithm is to read and sort the queue, and then to attempt to process all jobs in order. When it attempts to run the job, *sendmail* first checks to see if the job is locked. If so, it ignores the job.

There is no attempt to insure that only one queue processor exists at any time, since there is no guarantee that a job cannot take forever to process. Due to the locking algorithm, it is impossible for one job to freeze the queue. However, an uncooperative recipient host or a program recipient that never returns can accumulate many processes in your system. Unfortunately, there is no way to resolve this without violating the protocol.

In some cases, you may find that a major host going down for a couple of days may create a prohibitively large queue. This will result in *sendmail* spending an inordinate amount of time sorting the queue. This situation can be fixed by moving the queue to a temporary place and creating a new queue. The old queue can be run later when the offending host returns to service.

To do this, it is acceptable to move the entire queue directory:

```
cd /usr/spool
mv mqueue omqueue; mkdir mqueue; chmod 777 mqueue
```

You should then kill the existing daemon (since it will still be processing in the old queue directory) and create a new daemon.

To run the old mail queue, run the following command:

```
/usr/lib/sendmail -oQ/usr/spool/omqueue -q
```

The **-oQ** flag specifies an alternate queue directory and the **-q** flag says to just run every job in the queue. If you have a tendency toward voyeurism, you can use the **-v** flag to watch what is going on.

When the queue is finally emptied, you can remove the directory:

```
rmdir /usr/spool/omqueue
```

2.4. The Alias Database

The alias database exists in two forms. One is a text form, maintained in the file */usr/lib/aliases*. The aliases are of the form

name: name1, name2, ...

Only local names may be aliased; e.g.,

eric@mit-xx: eric@berkeley

will not have the desired effect. Aliases may be continued by starting any continuation lines with a space or a tab. Blank lines and lines beginning with a sharp sign ("#") are comments.

The second form is processed by the *dbm*(3) library. This form is in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag*. This is the form that *sendmail* actually uses to resolve aliases. This technique is used to improve performance.

2.4.1. Rebuilding the alias database

The DBM version of the database may be rebuilt explicitly by executing the command

newaliases

This is equivalent to giving *sendmail* the **−bi** flag:

/usr/lib/sendmail −bi

If the "D" option is specified in the configuration, *sendmail* will rebuild the alias database automatically if possible when it is out of date. The conditions under which it will do this are:

(1) The DBM version of the database is mode 666. -or-

(2) *Sendmail* is running setuid to root.

Auto-rebuild can be dangerous on heavily loaded machines with large alias files; if it might take more than five minutes to rebuild the database, there is a chance that several processes will start the rebuild process simultaneously.

2.4.2. Potential problems

There are a number of problems that can occur with the alias database. They all result from a *sendmail* process accessing the DBM version while it is only partially built. This can happen under two circumstances: One process accesses the database while another process is rebuilding it, or the process rebuilding the database dies (due to being killed or a system crash) before completing the rebuild.

Sendmail has two techniques to try to relieve these problems. First, it ignores interrupts while rebuilding the database; this avoids the problem of someone aborting the process leaving a partially rebuilt database. Second, at the end of the rebuild it adds an alias of the form

@: @

(which is not normally legal). Before sendmail will access the database, it checks to insure that this entry exists[1]. It will wait up to five minutes for this entry to appear, at which point it will force a rebuild itself[2].

---

[1]The "a" option is required in the configuration for this action to occur. This should normally be specified unless you are running *delivermail* in parallel with *sendmail*.

[2]Note: the "D" option must be specified in the configuration file for this operation to occur.

2.4.3. List owners

If an error occurs on sending to a certain address, say "*x*", *sendmail* will look for an alias of the form "owner-*x*" to receive the errors. This is typically useful for a mailing list where the submitter of the list has no control over the maintanence of the list itself; in this case the list maintainer would be the owner of the list. For example:

```
unix-wizards: eric@ucbarpa, wnj@monet, nosuchuser,
        sam@matisse
owner-unix-wizards: eric@ucbarpa
```

would cause "eric@ucbarpa" to get the error that will occur when someone sends to unix-wizards due to the inclusion of "nosuchuser" on the list.

2.5. Per-User Forwarding (.forward Files)

As an alternative to the alias database, any user may put a file with the name ".forward" in his or her home directory. If this file exists, *sendmail* redirects mail for that user to the list of addresses listed in the .forward file. For example, if the home directory for user "mckusick" has a .forward file with contents:

```
mckusick@ernie
kirk@calder
```

then any mail arriving for "mckusick" will be redirected to the specified accounts.

2.6. Special Header Lines

Several header lines have special interpretations defined by the configuration file. Others have interpretations built into *sendmail* that cannot be changed without changing the code. These builtins are described here.

2.6.1. Return-Receipt-To:

If this header is sent, a message will be sent to any specified addresses when the final delivery is complete. if the mailer has the **l** flag (local delivery) set in the mailer descriptor.

2.6.2. Errors-To:

If errors occur anywhere during processing, this header will cause error messages to go to the listed addresses rather than to the sender. This is intended for mailing lists.

2.6.3. Apparently-To:

If a message comes in with no recipients listed in the message (in a To:, Cc:, or Bcc: line) then *sendmail* will add an "Apparently-To:" header line for any recipients it is aware of. This is not put in as a standard recipient line to warn any recipients that the list is not complete.

At least one recipient line is required under RFC 822.

3. ARGUMENTS

The complete list of arguments to *sendmail* is described in detail in Appendix A. Some important arguments are described here.

3.1. Queue Interval

The amount of time between forking a process to run through the queue is defined by the **−q** flag. If you run in mode **f** or **a** this can be relatively large, since it will only be relevant when a host that was down comes back up. If you run in **q** mode it should be relatively short, since it defines the maximum amount of time that a message may sit in the queue.

3.2. Daemon Mode

If you allow incoming mail over an IPC connection, you should have a daemon running. This should be set by your */etc/rc* file using the **−bd** flag. The **−bd** flag and the **−q** flag may be combined in one call:

```
/usr/lib/sendmail −bd −q30m
```

3.3. Forcing the Queue

In some cases you may find that the queue has gotten clogged for some reason. You can force a queue run using the **−q** flag (with no value). It is entertaining to use the **−v** flag (verbose) when this is done to watch what happens:

```
/usr/lib/sendmail −q −v
```

3.4. Debugging

There are a fairly large number of debug flags built into *sendmail*. Each debug flag has a number and a level, where higher levels means to print out more information. The convention is that levels greater than nine are "absurd," i.e., they print out so much information that you wouldn't normally want to see them except for debugging that particular piece of code. Debug flags are set using the **−d** option; the syntax is:

```
debug-flag:    −d debug-list
debug-list:    debug-option [ , debug-option ]
debug-option:  debug-range [ . debug-level ]
debug-range:   integer | integer − integer
debug-level:   integer
```

where spaces are for reading ease only. For example,

```
−d12        Set flag 12 to level 1
−d12.3      Set flag 12 to level 3
−d3-17      Set flags 3 through 17 to level 1
−d3-17.4    Set flags 3 through 17 to level 4
```

For a complete list of the available debug flags you will have to look at the code (they are too dynamic to keep this documentation up to date).

3.5. Trying a Different Configuration File

An alternative configuration file can be specified using the **−C** flag; for example,

```
/usr/lib/sendmail −Ctest.cf
```

uses the configuration file *test.cf* instead of the default */usr/lib/sendmail.cf*. If the **−C** flag has no value it defaults to *sendmail.cf* in the current directory.

3.6. Changing the Values of Options

Options can be overridden using the **−o** flag. For example,

/usr/lib/sendmail −oT2m

sets the **T** (timeout) option to two minutes for this run only.

4. TUNING

There are a number of configuration parameters you may want to change, depending on the requirements of your site. Most of these are set using an option in the configuration file. For example, the line "OT3d" sets option "T" to the value "3d" (three days).

4.1. Timeouts

All time intervals are set using a scaled syntax. For example, "10m" represents ten minutes, whereas "2h30m" represents two and a half hours. The full set of scales is:

s	seconds
m	minutes
h	hours
d	days
w	weeks

4.1.1. Queue interval

The argument to the **−q** flag specifies how often a subdaemon will run the queue. This is typically set to between five minutes and one half hour.

4.1.2. Read timeouts

It is possible to time out when reading the standard input or when reading from a remote SMTP server. Technically, this is not acceptable within the published protocols. However, it might be appropriate to set it to something large in certain environments (such as an hour). This will reduce the chance of large numbers of idle daemons piling up on your system. This timeout is set using the **r** option in the configuration file.

4.1.3. Message timeouts

After sitting in the queue for a few days, a message will time out. This is to insure that at least the sender is aware of the inability to send a message. The timeout is typically set to three days. This timeout is set using the **T** option in the configuration file.

The time of submission is set in the queue, rather than the amount of time left until timeout. As a result, you can flush messages that have been hanging for a short period by running the queue with a short message timeout. For example,

/usr/lib/sendmail −oT1d −q

will run the queue and flush anything that is one day old.

4.2. Delivery Mode

There are a number of delivery modes that *sendmail* can operate in, set by the "d" configuration option. These modes specify how quickly mail will be delivered. Legal modes are:

i deliver interactively (synchronously)
b deliver in background (asynchronously)
q queue only (don't deliver)

There are tradeoffs. Mode "i" passes the maximum amount of information to the sender, but is hardly ever necessary. Mode "q" puts the minimum load on your machine, but means that delivery may be delayed for up to the queue interval. Mode "b" is probably a good compromise. However, this mode can cause large numbers of processes if you have a mailer that takes a long time to deliver a message.

4.3. Log Level

The level of logging can be set for sendmail. The default using a standard configuration table is level 9. The levels are as follows:

0 No logging.

1 Major problems only.

2 Message collections and failed deliveries.

3 Successful deliveries.

4 Messages being defered (due to a host being down, etc.).

5 Normal message queueups.

6 Unusual but benign incidents, e.g., trying to process a locked queue file.

9 Log internal queue id to external message id mappings. This can be useful for tracing a message as it travels between several hosts.

12 Several messages that are basically only of interest when debugging.

16 Verbose information regarding the queue.

4.4. File Modes

There are a number of files that may have a number of modes. The modes depend on what functionality you want and the level of security you require.

4.4.1. To suid or not to suid?

Sendmail can safely be made setuid to root. At the point where it is about to *exec*(2) a mailer, it checks to see if the userid is zero; if so, it resets the userid and groupid to a default (set by the **u** and **g** options). (This can be overridden by setting the **S** flag to the mailer for mailers that are trusted and must be called as root.) However, this will cause mail processing to be accounted (using *sa*(8)) to root rather than to the user sending the mail.

4.4.2. Temporary file modes

The mode of all temporary files that *sendmail* creates is determined by the "F" option. Reasonable values for this option are 0600 and 0644. If the more permissive mode is selected, it will not be necessary to run *sendmail* as root at all (even when running the queue).

4.4.3. Should my alias database be writable?

At Berkeley we have the alias database (/usr/lib/aliases*) mode 666. There are some dangers inherent in this approach: any user can add him-/her-self to any list, or can "steal" any other user's mail. However, we have found users to be basically trustworthy, and the cost of having a read-only database greater than the expense of finding and eradicating the rare nasty person.

The database that *sendmail* actually used is represented by the two files *aliases.dir* and *aliases.pag* (both in /usr/lib). The mode on these files should match the mode on /usr/lib/aliases. If *aliases* is writable and the DBM files (*aliases.dir* and *aliases.pag*) are not, users will be unable to reflect their desired changes through to the actual database. However, if *aliases* is read-only and the DBM files are writable, a slightly sophisticated user can arrange to steal mail anyway.

If your DBM files are not writable by the world or you do not have autorebuild enabled (with the "D" option), then you must be careful to reconstruct the alias database each time you change the text version:

newaliases

If this step is ignored or forgotten any intended changes will also be ignored or forgotten.

5. THE WHOLE SCOOP ON THE CONFIGURATION FILE

This section describes the configuration file in detail, including hints on how to write one of your own if you have to.

There is one point that should be made clear immediately: the syntax of the configuration file is designed to be reasonably easy to parse, since this is done every time *sendmail* starts up, rather than easy for a human to read or write. On the "future project" list is a configuration-file compiler.

An overview of the configuration file is given first, followed by details of the semantics.

5.1. The Syntax

The configuration file is organized as a series of lines, each of which begins with a single character defining the semantics for the rest of the line. Lines beginning with a space or a tab are continuation lines (although the semantics are not well defined in many places). Blank lines and lines beginning with a sharp symbol ('#') are comments.

5.1.1. R and S – rewriting rules

The core of address parsing are the rewriting rules. These are an ordered production system. *Sendmail* scans through the set of rewriting rules looking for a match on the left hand side (LHS) of the rule. When a rule matches, the address is replaced by the right hand side (RHS) of the rule.

There are several sets of rewriting rules. Some of the rewriting sets are used internally and must have specific semantics. Other rewriting sets do not have specifically assigned semantics, and may be referenced by the mailer definitions or by other rewriting sets.

The syntax of these two commands are:

S*n*

Sets the current ruleset being collected to *n*. If you begin a ruleset more than once it deletes the old definition.

R*lhs rhs comments*

The fields must be separated by at least one tab character; there may be embedded spaces in the fields. The *lhs* is a pattern that is applied to the input. If it matches, the input is rewritten to the *rhs*. The *comments* are ignored.

5.1.2. D – define macro

Macros are named with a single character. These may be selected from the entire ASCII set, but user-defined macros should be selected from the set of upper case letters only. Lower case letters and special symbols are used internally.

The syntax for macro definitions is:

D*x val*

where *x* is the name of the macro and *val* is the value it should have. Macros can be interpolated in most places using the escape sequence **$***x*.

5.1.3. C and F – define classes

Classes of words may be defined to match on the left hand side of rewriting rules. For example a class of all local names for this site might be created so that attempts to send to oneself can be eliminated. These can either be defined directly in the configuration file or read in from another file. Classes may be given names from the set of upper case letters. Lower case letters and special characters are reserved for system use.

The syntax is:

C*c word1 word2...*
F*c file* [*format*]

The first form defines the class *c* to match any of the named words. It is permissible to split them among multiple lines; for example, the two forms:

CHmonet ucbmonet

and

CHmonet
CHucbmonet

are equivalent. The second form reads the elements of the class *c* from the named *file*; the *format* is a *scanf*(3) pattern that should produce a single string.

5.1.4. M – define mailer

Programs and interfaces to mailers are defined in this line. The format is:

M*name*, {*field=value*}*

where *name* is the name of the mailer (used internally only) and the "field=name" pairs define attributes of the mailer. Fields are:

Path	The pathname of the mailer
Flags	Special flags for this mailer
Sender	A rewriting set for sender addresses
Recipient	A rewriting set for recipient addresses
Argv	An argument vector to pass to this mailer
Eol	The end-of-line string for this mailer
Maxsize	The maximum message length to this mailer

Only the first character of the field name is checked.

5.1.5. H – define header

The format of the header lines that sendmail inserts into the message are defined by the **H** line. The syntax of this line is:

H[**?***mflags***?**]*hname*: *htemplate*

Continuation lines in this spec are reflected directly into the outgoing message. The *htemplate* is macro expanded before insertion into the message. If the *mflags* (surrounded by question marks) are specified, at least one of the specified flags must be stated in the mailer definition for this header to be automatically output. If one of these headers is in the input it is reflected to the output regardless of these flags.

Some headers have special semantics that will be described below.

5.1.6. O – set option

There are a number of "random" options that can be set from a configuration file. Options are represented by single characters. The syntax of this line is:

O*o value*

This sets option *o* to be *value*. Depending on the option, *value* may be a string, an integer, a boolean (with legal values "t", "T", "f", or "F"; the default is TRUE), or a time interval.

5.1.7. T – define trusted users

Trusted users are those users who are permitted to override the sender address using the **–f** flag. These typically are "root," "uucp," and "network," but on some users it may be convenient to extend this list to include other users, perhaps to support a separate UUCP login for each host. The syntax of this line is:

T*user1 user2...*

There may be more than one of these lines.

5.1.8. P – precedence definitions

Values for the "Precedence:" field may be defined using the **P** control line. The syntax of this field is:

P*name=num*

When the *name* is found in a "Precedence:" field, the message class is set to *num*. Higher numbers mean higher precedence. Numbers less than zero have the special property that error messages will not be returned. The default precedence is zero. For example, our list of precedences is:

```
Pfirst-class=0
Pspecial-delivery=100
Pjunk=-100
```

5.2. The Semantics

This section describes the semantics of the configuration file.

5.2.1. Special macros, conditionals

Macros are interpolated using the construct **$***x*, where *x* is the name of the macro to be interpolated. In particular, lower case letters are reserved to have special semantics, used to pass information in or out of sendmail, and some special characters are reserved to provide conditionals, etc.

The following macros *must* be defined to transmit information into *sendmail:*

e The SMTP entry message
j The "official" domain name for this site
l The format of the UNIX from line
n The name of the daemon (for error messages)
o The set of "operators" in addresses
q default format of sender address

The **$e** macro is printed out when SMTP starts up. The first word must be the **$j** macro. The **$j** macro should be in RFC821 format. The **$l** and **$n** macros can be considered constants except under terribly unusual circumstances. The **$o** macro consists of a list of characters which will be considered tokens and which will separate tokens when doing parsing. For example, if "r" were in the **$o** macro, then the input "address" would be scanned as three tokens: "add," "r," and "ess." Finally, the **$q** macro specifies how an address should appear in a message when it is defaulted. For example, on our system these definitions are:

```
De$j Sendmail $v ready at $b
DnMAILER-DAEMON
DlFrom $g  $d
Do.:%@!^=/
Dq$g$?x ($x)$.
Dj$H.$D
```

An acceptable alternative for the **$q** macro is "$?x$x $.<$g>". These correspond to the following two formats:

```
eric@Berkeley (Eric Allman)
Eric Allman <eric@Berkeley>
```

Some macros are defined by *sendmail* for interpolation into argv's for mailers or for other contexts. These macros are:

a The origination date in Arpanet format
b The current date in Arpanet format
c The hop count
d The date in UNIX (ctime) format
f The sender (from) address
g The sender address relative to the recipient
h The recipient host
i The queue id
p Sendmail's pid
r Protocol used
s Sender's host name
t A numeric representation of the current time
u The recipient user
v The version number of sendmail
w The hostname of this site
x The full name of the sender
y The id of the sender's tty
z The home directory of the recipient

There are three types of dates that can be used. The **$a** and **$b** macros are in Arpanet format; **$a** is the time as extracted from the "Date:" line of the message (if there was one), and **$b** is the current date and time (used for postmarks). If no "Date:" line is found in the incoming message, **$a** is set to the current time also. The **$d** macro is equivalent to the **$a** macro in UNIX (ctime) format.

The **$f** macro is the id of the sender as originally determined; when mailing to a specific host the **$g** macro is set to the address of the sender *relative to the recipient*. For example, if I send to "bollard@matisse" from the machine "ucbarpa" the **$f** macro will be "eric" and the **$g** macro will be "eric@ucbarpa."

The **$x** macro is set to the full name of the sender. This can be determined in several ways. It can be passed as flag to *sendmail*. The second choice is the value of the "Full-name:" line in the header if it exists, and the third choice is the comment field of a "From:" line. If all of these fail, and if the message is being originated locally, the full name is looked up in the */etc/passwd* file.

When sending, the **$h**, **$u**, and **$z** macros get set to the host, user, and home directory (if local) of the recipient. The first two are set from the **$@** and **$:** part of the rewriting rules, respectively.

The **$p** and **$t** macros are used to create unique strings (e.g., for the "Message-Id:" field). The **$i** macro is set to the queue id on this host; if put into the timestamp line it can be extremely useful for tracking messages. The **$y** macro is set to the id of the terminal of the sender (if known); some systems like to put this in the Unix "From" line. The **$v** macro is set to be the version number of *sendmail*; this is normally put in timestamps and has been proven extremely useful for debugging. The **$w** macro is set to the name of this host if it can be determined. The **$c** field is set to the "hop count," i.e., the number of times this message has been processed. This can be determined by the **−h** flag on the command line or by counting the timestamps in the message.

The **$r** and **$s** fields are set to the protocol used to communicate with sendmail and the sending hostname; these are not supported in the current version.

Conditionals can be specified using the syntax:

$?x text1 $|text2 $.

This interpolates *text1* if the macro **$x** is set, and *text2* otherwise. The "else"(**$|**) clause may be omitted.

5.2.2. Special classes

The class **$=w** is set to be the set of all names this host is known by. This can be used to delete local hostnames.

5.2.3. The left hand side

The left hand side of rewriting rules contains a pattern. Normal words are simply matched directly. Metasyntax is introduced using a dollar sign. The metasymbols are:

$* Match zero or more tokens
$+ Match one or more tokens
$− Match exactly one token
$=*x* Match any token in class *x*
$~*x* Match any token not in class *x*

If any of these match, they are assigned to the symbol **$***n* for replacement on the right hand side, where *n* is the index in the LHS. For example, if the LHS:

$−:$+

is applied to the input:

UCBARPA:eric

the rule will match, and the values passed to the RHS will be:

$1 UCBARPA
$2 eric

5.2.4. The right hand side

When the right hand side of a rewriting rule matches, the input is deleted and replaced by the right hand side. Tokens are copied directly from the RHS unless they are begin with a dollar sign. Metasymbols are:

$*n*	Substitute indefinite token *n* from LHS
$>*n*	"Call" ruleset *n*
$#*mailer*	Resolve to *mailer*
$@*host*	Specify *host*
$:*user*	Specify *user*

The **$***n* syntax substitutes the corresponding value from a **$+**, **$−**, **$***, **$=**, or **$~** match on the LHS. It may be used anywhere.

The **$>***n* syntax causes the remainder of the line to be substituted as usual and then passed as the argument to ruleset *n*. The final value of ruleset *n* then becomes the substitution for this rule.

The **$#** syntax should *only* be used in ruleset zero. It causes evaluation of the ruleset to terminate immediately, and signals to sendmail that the address has completely resolved. The complete syntax is:

$#***mailer***$@*host***$:***user*

This specifies the {mailer, host, user} 3-tuple necessary to direct the mailer. If the mailer is local the host part may be omitted. The *mailer* and *host* must be a single word, but the *user* may be multi-part.

A RHS may also be preceeded by a **$@** or a **$:** to control evaluation. A **$@** prefix causes the ruleset to return with the remainder of the RHS as the value. A **$:** prefix causes the rule to terminate immediately, but the ruleset to continue; this can be used to avoid continued application of a rule. The prefix is stripped before continuing.

The **$@** and **$:** prefixes may preceed a **$>** spec; for example:

```
R$+     $:$>7$1
```

matches anything, passes that to ruleset seven, and continues; the **$:** is necessary to avoid an infinite loop.

5.2.5. Semantics of rewriting rule sets

There are five rewriting sets that have specific semantics. These are related as depicted by figure 2.

Ruleset three should turn the address into "canonical form." This form should have the basic syntax:

local-part@host-domain-spec

If no "@" sign is specified, then the host-domain-spec *may* be appended from the sender address (if the **C** flag is set in the mailer definition corresponding to the *sending* mailer). Ruleset three is applied by sendmail before doing anything with any address.

Ruleset zero is applied after ruleset three to addresses that are going to actually specify recipients. It must resolve to a {*mailer, host, user*} triple. The *mailer* must be defined in the mailer definitions from the configuration file. The *host* is

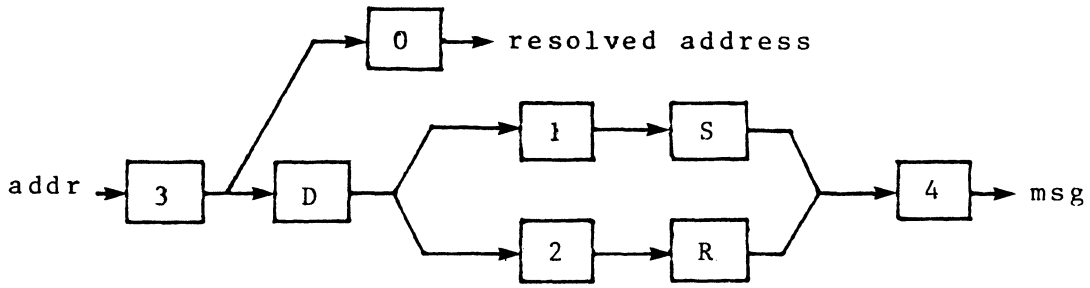

Figure 2 Rewriting Set Semantics

D--Sender domain addition S--mailer-specific sender rewritin
R--mailer-specific recipient rewriting

defined into the **$h** macro for use in the argv expansion of the specified mailer.

Rulesets one and two are applied to all sender and recipient addresses respectively. They are applied before any specification in the mailer definition. They must never resolve.

Ruleset four is applied to all addresses in the message. It is typically used to translate internal to external form.

5.2.6. Mailer flags etc.

There are a number of flags that may be associated with each mailer, each identified by a letter of the alphabet. Many of them are assigned semantics internally. These are detailed in Appendix C. Any other flags may be used freely to conditionally assign headers to messages destined for particular mailers.

5.2.7. The "error" mailer

The mailer with the special name "error" can be used to generate a user error. The (optional) host field is a numeric exit status to be returned, and the user field is a message to be printed. For example, the entry:

$#error$:Host unknown in this domain

on the RHS of a rule will cause the specified error to be generated if the LHS matches. This mailer is only functional in ruleset zero.

5.3. Building a Configuration File From Scratch

Building a configuration table from scratch is an extremely difficult job. Fortunately, it is almost never necessary to do so; nearly every situation that may come up may be resolved by changing an existing table. In any case, it is critical that you understand what it is that you are trying to do and come up with a philosophy for the configuration table. This section is intended to explain what the real purpose of a configuration table is and to give you some ideas for what your philosophy might be.

5.3.1. What you are trying to do

The configuration table has three major purposes. The first and simplest is to set up the environment for *sendmail*. This involves setting the options, defining a

few critical macros, etc. Since these are described in other places, we will not go into more detail here.

The second purpose is to rewrite addresses in the message. This should typically be done in two phases. The first phase maps addresses in any format into a canonical form. This should be done in ruleset three. The second phase maps this canonical form into the syntax appropriate for the receiving mailer. *Sendmail* does this in three subphases. Rulesets one and two are applied to all sender and recipient addresses respectively. After this, you may specify per-mailer rulesets for both sender and recipient addresses; this allows mailer-specific customization. Finally, ruleset four is applied to do any default conversion to external form.

The third purpose is to map addresses into the actual set of instructions necessary to get the message delivered. Ruleset zero must resolve to the internal form, which is in turn used as a pointer to a mailer descriptor. The mailer descriptor describes the interface requirements of the mailer.

5.3.2. Philosophy

The particular philosophy you choose will depend heavily on the size and structure of your organization. I will present a few possible philosophies here.

One general point applies to all of these philosophies: it is almost always a mistake to try to do full name resolution. For example, if you are trying to get names of the form "user@host" to the Arpanet, it does not pay to route them to "xyzvax!decvax!ucbvax!c70:user@host" since you then depend on several links not under your control. The best approach to this problem is to simply forward to "xyzvax!user@host" and let xyzvax worry about it from there. In summary, just get the message closer to the destination, rather than determining the full path.

5.3.2.1. Large site, many hosts – minimum information

Berkeley is an example of a large site, i.e., more than two or three hosts. We have decided that the only reasonable philosophy in our environment is to designate one host as the guru for our site. It must be able to resolve any piece of mail it receives. The other sites should have the minimum amount of information they can get away with. In addition, any information they do have should be hints rather than solid information.

For example, a typical site on our local ether network is "monet." Monet has a list of known ethernet hosts; if it receives mail for any of them, it can do direct delivery. If it receives mail for any unknown host, it just passes it directly to "ucbvax," our master host. Ucbvax may determine that the host name is illegal and reject the message, or may be able to do delivery. However, it is important to note that when a new ethernet host is added, the only host that *must* have its tables updated is ucbvax; the others *may* be updated as convenient, but this is not critical.

This picture is slightly muddied due to network connections that are not actually located on ucbvax. For example, our TCP connection is currently on "ucbarpa." However, monet *does not* know about this; the information is hidden totally between ucbvax and ucbarpa. Mail going from monet to a TCP host is transfered via the ethernet from monet to ucbvax, then via the ethernet from ucbvax to ucbarpa, and then is submitted to the Arpanet. Although this involves some extra hops, we feel this is an acceptable tradeoff.

An interesting point is that it would be possible to update monet to send TCP mail directly to ucbarpa if the load got too high; if monet failed to note a

host as a TCP host it would go via ucbvax as before, and if monet incorrectly sent a message to ucbarpa it would still be sent by ucbarpa to ucbvax as before. The only problem that can occur is loops, as if ucbarpa thought that ucbvax had the TCP connection and vice versa. For this reason, updates should *always* happen to the master host first.

This philosophy results as much from the need to have a single source for the configuration files (typically built using *m4*(1) or some similar tool) as any logical need. Maintaining more than three separate tables by hand is essentially an impossible job.

5.3.2.2. Small site – complete information

A small site (two or three hosts) may find it more reasonable to have complete information at each host. This would require that each host know exactly where each network connection is, possibly including the names of each host on that network. As long as the site remains small and the the configuration remains relatively static, the update problem will probably not be too great.

5.3.2.3. Single host

This is in some sense the trivial case. The only major issue is trying to insure that you don't have to know too much about your environment. For example, if you have a UUCP connection you might find it useful to know about the names of hosts connected directly to you, but this is really not necessary since this may be determined from the syntax.

5.3.3. Relevant issues

The canonical form you use should almost certainly be as specified in the Arpanet protocols RFC819 and RFC822. Copies of these RFC's are included on the *sendmail* tape as *doc/rfc819.lpr* and *doc/rfc822.lpr*.

RFC822 describes the format of the mail message itself. *Sendmail* follows this RFC closely, to the extent that many of the standards described in this document can not be changed without changing the code. In particular, the following characters have special interpretations:

< > ( ) " \

Any attempt to use these characters for other than their RFC822 purpose in addresses is probably doomed to disaster.

RFC819 describes the specifics of the domain-based addressing. This is touched on in RFC822 as well. Essentially each host is given a name which is a right-to-left dot qualified pseudo-path from a distinguished root. The elements of the path need not be physical hosts; the domain is logical rather than physical. For example, at Berkeley one legal host is "a.cc.berkeley.arpa"; reading from right to left, "arpa" is a top level domain (related to, but not limited to, the physical Arpanet), "berkeley" is both an Arpanet host and a logical domain which is actually interpreted by a host called ucbvax (which is actually just the "major" host for this domain), "cc" represents the Computer Center, (in this case a strictly logical entity), and "a" is a host in the Computer Center; this particular host happens to be connected via berknet, but other hosts might be connected via one of two ethernets or some other network.

Beware when reading RFC819 that there are a number of errors in it.

5.3.4. How to proceed

Once you have decided on a philosophy, it is worth examining the available configuration tables to decide if any of them are close enough to steal major parts of. Even under the worst of conditions, there is a fair amount of boiler plate that can be collected safely.

The next step is to build ruleset three. This will be the hardest part of the job. Beware of doing too much to the address in this ruleset, since anything you do will reflect through to the message. In particular, stripping of local domains is best deferred, since this can leave you with addresses with no domain spec at all. Since *sendmail* likes to append the sending domain to addresses with no domain, this can change the semantics of addresses. Also try to avoid fully qualifying domains in this ruleset. Although technically legal, this can lead to unpleasantly and unnecessarily long addresses reflected into messages. The Berkeley configuration files define ruleset nine to qualify domain names and strip local domains. This is called from ruleset zero to get all addresses into a cleaner form.

Once you have ruleset three finished, the other rulesets should be relatively trivial. If you need hints, examine the supplied configuration tables.

5.3.5. Testing the rewriting rules – the –bt flag

When you build a configuration table, you can do a certain amount of testing using the "test mode" of *sendmail*. For example, you could invoke *sendmail* as:

```
sendmail –bt –Ctest.cf
```

which would read the configuration file "test.cf" and enter test mode. In this mode, you enter lines of the form:

```
rwset address
```

where *rwset* is the rewriting set you want to use and *address* is an address to apply the set to. Test mode shows you the steps it takes as it proceeds, finally showing you the address it ends up with. You may use a comma separated list of rwsets for sequential application of rules to an input; ruleset three is always applied first. For example:

```
1,21,4 monet:bollard
```

first applies ruleset three to the input "monet:bollard." Ruleset one is then applied to the output of ruleset three, followed similarly by rulesets twenty-one and four.

If you need more detail, you can also use the "–d21" flag to turn on more debugging. For example,

```
sendmail –bt –d21.99
```

turns on an incredible amount of information; a single word address is probably going to print out several pages worth of information.

5.3.6. Building mailer descriptions

To add an outgoing mailer to your mail system, you will have to define the characteristics of the mailer.

Each mailer must have an internal name. This can be arbitrary, except that the names "local" and "prog" must be defined.

The pathname of the mailer must be given in the P field. If this mailer should be accessed via an IPC connection, use the string "[IPC]" instead.

The F field defines the mailer flags. You should specify an "f" or "r" flag to pass the name of the sender as a **−f** or **−r** flag respectively. These flags are only passed if they were passed to *sendmail,* so that mailers that give errors under some circumstances can be placated. If the mailer is not picky you can just specify "−f $g" in the argv template. If the mailer must be called as **root** the "S" flag should be given; this will not reset the userid before calling the mailer[3]. If this mailer is local (i.e., will perform final delivery rather than another network hop) the "l" flag should be given. Quote characters (backslashes and " marks) can be stripped from addresses if the "s" flag is specified; if this is not given they are passed through. If the mailer is capable of sending to more than one user on the same host in a single transaction the "m" flag should be stated. If this flag is on, then the argv template containing **$u** will be repeated for each unique user on a given host. The "e" flag will mark the mailer as being "expensive," which will cause *sendmail* to defer connection until a queue run[4].

An unusual case is the "C" flag. This flag applies to the mailer that the message is received from, rather than the mailer being sent to; if set, the domain spec of the sender (i.e., the "@host.domain" part) is saved and is appended to any addresses in the message that do not already contain a domain spec. For example, a message of the form:

From: eric@ucbarpa
To: wnj@monet, mckusick

will be modified to:

From: eric@ucbarpa
To: wnj@monet, mckusick@ucbarpa

if and only if the "C" flag is defined in the mailer corresponding to "eric@ucbarpa."

Other flags are described in Appendix C.

The S and R fields in the mailer description are per-mailer rewriting sets to be applied to sender and recipient addresses respectively. These are applied after the sending domain is appended and the general rewriting sets (numbers one and two) are applied, but before the output rewrite (ruleset four) is applied. A typical use is to append the current domain to addresses that do not already have a domain. For example, a header of the form:

From: eric

might be changed to be:

From: eric@ucbarpa

or

From: ucbvax!eric

depending on the domain it is being shipped into. These sets can also be used to do special purpose output rewriting in cooperation with ruleset four.

The E field defines the string to use as an end-of-line indication. A string containing only newline is the default. The usual backslash escapes (\r,\n,\f,\b) may be used.

Finally, an argv template is given as the E field. It may have embedded spaces. If there is no argv with a **$u** macro in it, *sendmail* will speak SMTP to the mailer. If the pathname for this mailer is "[IPC]," the argv should be

---

[3]*Sendmail* must be running setuid to root for this to work.

[4]The "c" configuration option must be given for this to be effective.

IPC $h [*port*]

where *port* is the optional port number to connect to.

For example, the specifications:

```
Mlocal, P=/bin/mail, F=rlsm  S=10, R=20, A=mail -d $u
Mether,P=[IPC],      F=meC, S=11, R=21, A=IPC $h, M=100000
```

specifies a mailer to do local delivery and a mailer for ethernet delivery. The first is called "local," is located in the file "/bin/mail," takes a picky **-r** flag, does local delivery, quotes should be stripped from addresses, and multiple users can be delivered at once; ruleset ten should be applied to sender addresses in the message and ruleset twenty should be applied to recipient addresses; the argv to send to a message will be the word "mail," the word "-d," and words containing the name of the receiving user. If a **-r** flag is inserted it will be between the words "mail" and "-d." The second mailer is called "ether," it should be connected to via an IPC connection, it can handle multiple users at once, connections should be deferred, and any domain from the sender address should be appended to any receiver name without a domain; sender addresses should be processed by ruleset eleven and recipient addresses by ruleset twenty-one. There is a 100,000 byte limit on messages passed through this mailer.

APPENDIX A

COMMAND LINE FLAGS

Arguments must be presented with flags before addresses. The flags are:

−f *addr* The sender's machine address is *addr*. This flag is ignored unless the real user is listed as a "trusted user" or if *addr* contains an exclamation point (because of certain restrictions in UUCP).

−r *addr* An obsolete form of **−f**.

−h *cnt* Sets the "hop count" to *cnt*. This represents the number of times this message has been processed by *sendmail* (to the extent that it is supported by the underlying networks). *Cnt* is incremented during processing, and if it reaches MAXHOP (currently 30) *sendmail* throws away the message with an error.

−F*name* Sets the full name of this user to *name*.

−n Don't do aliasing or forwarding.

−t Read the header for "To:", "Cc:", and "Bcc:" lines, and send to everyone listed in those lists. The "Bcc:" line will be deleted before sending. Any addresses in the argument vector will be deleted from the send list.

−b*x* Set operation mode to *x*. Operation modes are:

m	Deliver mail (default)
a	Run in arpanet mode (see below)
s	Speak SMTP on input side
d	Run as a daemon
t	Run in test mode
v	Just verify addresses, don't collect or deliver
i	Initialize the alias database
p	Print the mail queue
z	Freeze the configuration file

The special processing for the ARPANET includes reading the "From:" line from the header to find the sender, printing ARPANET style messages (preceded by three digit reply codes for compatibility with the FTP protocol [Neigus73, Postel74, Postel77]), and ending lines of error messages with <CRLF>.

−q*time* Try to process the queued up mail. If the time is given, a sendmail will run through the queue at the specified interval to deliver queued mail; otherwise, it only runs once.

−C*file* Use a different configuration file.

−d*level* Set debugging level.

−o*x value* Set option *x* to the specified *value*. These options are described in Appendix B.

There are a number of options that may be specified as primitive flags (provided for compatibility with *delivermail*). These are the e, i, m, and v options. Also, the f option may be specified as the **−s** flag.

APPENDIX B

CONFIGURATION OPTIONS

The following options may be set using the **−o** flag on the command line or the **O** line in the configuration file:

A*file* — Use the named *file* as the alias file. If no file is specified, use *aliases* in the current directory.

a — If set, wait for an "@:@" entry to exist in the alias database before starting up. If it does not appear in five minutes, rebuild the database.

c — If an outgoing mailer is marked as being expensive, don't connect immediately. This requires that queueing be compiled in, since it will depend on a queue run process to actually send the mail.

d*x* — Deliver in mode *x*. Legal modes are:

- i Deliver interactively (synchronously)
- b Deliver in background (asynchronously)
- q Just queue the message (deliver during queue run)

D — If set, rebuild the alias database if necessary and possible. If this option is not set, *sendmail* will never rebuild the alias database unless explicitly requested using **−bi**.

e*x* — Dispose of errors using mode *x*. The values for *x* are:

- p Print error messages (default)
- q No messages, just give exit status
- m Mail back errors
- w Write back errors (mail if user not logged in)
- e Mail back errors and give zero exit stat always

F*n* — The temporary file mode, in octal. 644 and 600 are good choices.

f — Save Unix-style "From" lines at the front of headers. Normally they are assumed redundant and discarded.

g*n* — Set the default group id for mailers to run in to *n*.

H*file* — Specify the help file for SMTP.

i — Ignore dots in incoming messages.

L*n* — Set the default log level to *n*.

M*x value* — Set the macro *x* to *value*. This is intended only for use from the command line.

m — Send to me too, even if I am in an alias expansion.

o — Assume that the headers may be in old format, i.e., spaces delimit names. This actually turns on an adaptive algorithm: if any recipient address contains a comma, parenthesis, or angle bracket, it will be assumed that commas already exist. If this flag is not on, only commas delimit names. Headers are always output with commas between the names.

Q*dir* — Use the named *dir* as the queue directory.

r*time* — Timeout reads after *time* interval.

S*file*	Log statistics in the named *file*.
s	Be super-safe when running things, i.e., always instantiate the queue file, even if you are going to attempt immediate delivery. *Sendmail* always instantiates the queue file before returning control the the client under any circumstances.
T*time*	Set the queue timeout to *time*. After this interval, messages that have not been successfully sent will be returned to the sender.
t*S,D*	Set the local timezone name to *S* for standard time and *D* for daylight time; this is only used under version six.
u*n*	Set the default userid for mailers to *n*. Mailers without the *S* flag in the mailer definition will run as this user.
v	Run in verbose mode.

APPENDIX C

MAILER FLAGS

The following flags may be set in the mailer description.

f The mailer wants a **−f** *from* flag, but only if this is a network forward operation (i.e., the mailer will give an error if the executing user does not have special permissions).

r Same as **f**, but sends a **−r** flag.

S Don't reset the userid before calling the mailer. This would be used in a secure environment where *sendmail* ran as root. This could be used to avoid forged addresses. This flag is suppressed if given from an "unsafe" environment (e.g, a user's mail.cf file).

n Do not insert a UNIX-style "From" line on the front of the message.

l This mailer is local (i.e., final delivery will be performed).

s Strip quote characters off of the address before calling the mailer.

m This mailer can send to multiple users on the same host in one transaction. When a **$u** macro occurs in the *argv* part of the mailer definition, that field will be repeated as necessary for all qualifying users.

F This mailer wants a "From:" header line.

D This mailer wants a "Date:" header line.

M This mailer wants a "Message-Id:" header line.

x This mailer wants a "Full-Name:" header line.

P This mailer wants a "Return-Path:" line.

u Upper case should be preserved in user names for this mailer.

h Upper case should be preserved in host names for this mailer.

A This is an Arpanet-compatible mailer, and all appropriate modes should be set.

U This mailer wants Unix-style "From" lines with the ugly UUCP-style "remote from <host>" on the end.

e This mailer is expensive to connect to, so try to avoid connecting normally; any necessary connection will occur during a queue run.

X This mailer want to use the hidden dot algorithm as specified in RFC821; basically, any line beginning with a dot will have an extra dot prepended (to be stripped at the other end). This insures that lines in the message containing a dot will not terminate the message prematurely.

L Limit the line lengths as specified in RFC821.

P Use the return-path in the SMTP "MAIL FROM:" command rather than just the return address; although this is required in RFC821, many hosts do not process return paths properly.

I This mailer will be speaking SMTP to another *sendmail* − as such it can use special protocol features. This option is not required (i.e., if this option is omitted the transmission will still operate successfully, although perhaps not as efficiently as possible).

C If mail is *received* from a mailer with this flag set, any addresses in the header that do not have an at sign ("@") after being rewritten by ruleset three will have the "@domain" clause from the sender tacked on. This allows mail with headers of the form:

From: usera@hosta
To: userb@hostb, userc

to be rewritten as:

From: usera@hosta
To: userb@hostb, userc@hosta

automatically.

APPENDIX D

OTHER CONFIGURATION

There are some configuration changes that can be made by recompiling *sendmail*. These are located in three places:

md/config.m4 These contain operating-system dependent descriptions. They are interpolated into the Makefiles in the *src* and *aux* directories. This includes information about what version of UNIX you are running, what libraries you have to include, etc.

src/conf.h Configuration parameters that may be tweaked by the installer are included in conf.h.

src/conf.c Some special routines and a few variables may be defined in conf.c. For the most part these are selected from the settings in conf.h.

Parameters in md/config.m4

The following compilation flags may be defined in the *m4CONFIG* macro in *md/config.m4* to define the environment in which you are operating.

V6 If set, this will compile a version 6 system, with 8-bit user id's, single character tty id's, etc.

VMUNIX If set, you will be assumed to have a Berkeley 4BSD or 4.1BSD, including the *vfork* (2) system call, special types defined in <sys/types.h> (e.g, u char), etc.

If none of these flags are set, a version 7 system is assumed.

You will also have to specify what libraries to link with *sendmail* in the *m4LIBS* macro. Most notably, you will have to include if you are running a 4.1BSD system.

Parameters in src/conf.h

Parameters and compilation options are defined in conf.h. Most of these need not normally be tweaked; common parameters are all in sendmail.cf. However, the sizes of certain primitive vectors, etc., are included in this file. The numbers following the parameters are their default value.

MAXLINE [256] The maximum line length of any input line. If message lines exceed this length they will still be processed correctly; however, header lines, configuration file lines, alias lines, etc., must fit within this limit.

MAXNAME [128] The maximum length of any name, such as a host or a user name.

MAXFIELD [2500] The maximum total length of any header field, including continuation lines.

MAXPV [40] The maximum number of parameters to any mailer. This limits the number of recipients that may be passed in one transaction.

MAXHOP [30] When a message has been processed more than this number of times, sendmail rejects the message on the assumption that there has been an aliasing loop. This can be determined from the **-h** flag or by counting the number of trace fields (i.e, "Received:" lines) in the message header.

MAXATOM [100] The maximum number of atoms (tokens) in a single address. For example, the address "eric@Berkeley" is three atoms.

MAXMAILERS [25]
The maximum number of mailers that may be defined in the configuration file.

MAXRWSETS [30]
The maximum number of rewriting sets that may be defined.

MAXPRIORITIES [25]
The maximum number of values for the "Precedence:" field that may be defined (using the **P** line in sendmail.cf).

MAXTRUST [30] The maximum number of trusted users that may be defined (using the **T** line in sendmail.cf).

A number of other compilation options exist. These specify whether or not specific code should be compiled in.

DBM — If set, the "DBM" package in UNIX is used (see DBM(3X) in [UNIX80]). If not set, a much less efficient algorithm for processing aliases is used.

DEBUG — If set, debugging information is compiled in. To actually get the debugging output, the **−d** flag must be used.

LOG — If set, the *syslog* routine in use at some sites is used. This makes an informational log record for each message processed, and makes a higher priority log record for internal system errors.

QUEUE — This flag should be set to compile in the queueing code. If this is not set, mailers must accept the mail immediately or it will be returned to the sender.

SMTP — If set, the code to handle user and server SMTP will be compiled in. This is only necessary if your machine has some mailer that speaks SMTP.

DAEMON — If set, code to run a daemon is compiled in. This code is for 4.2BSD if the NVMUNIX flag is specified; otherwise, 4.1a BSD code is used. Beware however that there are bugs in the 4.1a code that make it impossible for **sendmail** to work correctly under heavy load.

UGLYUUCP — If you have a UUCP host adjacent to you which is not running a reasonable version of *rmail*, you will have to set this flag to include the "remote from sysname" info on the from line. Otherwise, UUCP gets confused about where the mail came from.

NOTUNIX — If you are using a non-UNIX mail format, you can set this flag to turn off special processing of UNIX-style "From " lines.

Configuration in src/conf.c

Not all header semantics are defined in the configuration file. Header lines that should only be included by certain mailers (as well as other more obscure semantics) must be specified in the *HdrInfo* table in *conf.c*. This table contains the header name (which should be in all lower case) and a set of header control flags (described below), The flags are:

H ACHECK — Normally when the check is made to see if a header line is compatible with a mailer, *sendmail* will not delete an existing line. If this flag is set, *sendmail* will delete even existing header lines. That is, if this bit is set and the mailer does not have flag bits set that intersect with the required mailer flags in the header definition in sendmail.cf, the header line is *always* deleted.

H EOH — If this header field is set, treat it like a blank line, i.e., it will signal the end of the header and the beginning of the message text.

H FORCE — Add this header entry even if one existed in the message before. If a header entry does not have this bit set, *sendmail* will not add another header line if a header line of this name already existed. This would normally be used to stamp the message by everyone who handled it.

H TRACE — If set, this is a timestamp (trace) field. If the number of trace fields in a message exceeds a preset amount the message is returned on the assumption that it has an aliasing loop.

H RCPT — If set, this field contains recipient addresses. This is used by the **−t** flag to determine who to send to when it is collecting recipients from the message.

H FROM — This flag indicates that this field specifies a sender. The order of these fields in the *HdrInfo* table specifies *sendmail's* preference for which field to return error messages to.

Let's look at a sample *HdrInfo* specification:

```
struct hdrinfo          HdrInfo[] =
{
        /* originator fields, most to least significant */
    "resent-sender",    H FROM,
    "resent-from",      H FROM,
    "sender",           H FROM,
    "from",             H FROM,
    "full-name",        H ACHECK,
        /* destination fields */
    "to",               H RCPT,
    "resent-to",        H RCPT,
    "cc",               H RCPT,
        /* message identification and control */
    "message",          H EOH,
    "text",             H EOH,
        /* trace fields */
    "received",         H TRACE|H FORCE,

    NULL,               0,
};
```

This structure indicates that the "To:", "Resent-To:", and "Cc:" fields all specify recipient addresses. Any "Full-Name:" field will be deleted unless the required mailer flag (indicated in the configuration file) is specified. The "Message:" and "Text:" fields will terminate the header; these are specified in new protocols [NBS80] or used by random dissenters around the network world. The "Received:" field will always be added, and can be used to trace messages.

There are a number of important points here. First, header fields are not added automatically just because they are in the *HdrInfo* structure; they must be specified in the configuration file in order to be added to the message. Any header fields mentioned in the configuration file but not mentioned in the *HdrInfo* structure have default processing performed; that is, they are added unless they were in the message already. Second, the *HdrInfo* structure only specifies cliched processing; certain headers are processed specially by ad hoc code regardless of the status specified in *HdrInfo*. For example, the "Sender:" and "From:" fields are always scanned on ARPANET mail to determine the sender; this is used to perform the "return to sender" function. The "From:" and "Full-Name:" fields are used to determine the full name of the sender if possible; this is stored in the macro **$x** and used in a number of ways.

The file *conf.c* also contains the specification of ARPANET reply codes. There are four classifications these fall into:

```
char  Arpa Info[] =      "050"; /* arbitrary info */
char  Arpa TSyserr[] =   "455"; /* some (transient) system error */
char  Arpa PSyserr[] =   "554"; /* some (transient) system error */
char  Arpa Usrerr[] =    "554"; /* some (fatal) user error */
```

The class *Arpa Info* is for any information that is not required by the protocol, such as forwarding information. *Arpa TSyserr* and *Arpa PSyserr* is printed by the *syserr* routine. TSyserr is printed out for transient errors, whereas PSyserr is printed for permanent errors; the distinction is made based on the value of *errno*. Finally, *Arpa Usrerr* is the result of a user error and is generated by the *usrerr* routine; these are generated when the user has specified something wrong, and hence the error is permanent, i.e., it will not work simply by resubmitting the request.

If it is necessary to restrict mail through a relay, the *checkcompat* routine can be modified. This routine is called for every recipient address. It can return **TRUE** to indicate that the address is acceptable and mail processing will continue, or it can return **FALSE** to reject the recipient. If it returns false, it is up to *checkcompat* to print an error message (using *usrerr*) saying why the message is rejected. For example, *checkcompat* could read:

```
bool
checkcompat(to)
    register ADDRESS *to;
{
    if (MsgSize > 50000 && to->q mailer != LocalMailer)
    {
        usrerr("Message too large for non-local delivery");
        NoReturn = TRUE;
        return (FALSE);
    }
    return (TRUE);
}
```

This would reject messages greater than 50000 bytes unless they were local. The *NoReturn* flag can be sent to supress the return of the actual body of the message in the error return. The actual use of this routine is highly dependent on the implementation, and use should be limited.

APPENDIX E

SUMMARY OF SUPPORT FILES

This is a summary of the support files that *sendmail* creates or generates.

/usr/lib/sendmail
: The binary of *sendmail.*

/usr/bin/newaliases
: A link to /usr/lib/sendmail; causes the alias database to be rebuilt. Running this program is completely equivalent to giving *sendmail* the **−bi** flag.

/usr/bin/mailq
: Prints a listing of the mail queue. This program is equivalent to using the **−bp** flag to *sendmail.*

/usr/lib/sendmail.cf
: The configuration file, in textual form.

/usr/lib/sendmail.fc
: The configuration file represented as a memory image.

/usr/lib/sendmail.hf
: The SMTP help file.

/usr/lib/sendmail.st
: A statistics file; need not be present.

/usr/lib/aliases
: The textual version of the alias file.

/usr/lib/aliases.{pag,dir}
: The alias file in *dbm*(3) format.

/etc/syslog
: The program to do logging.

/etc/syslog.conf
: The configuration file for syslog.

/etc/syslog.pid
: Contains the process id of the currently running syslog.

/usr/spool/mqueue
: The directory in which the mail queue and temporary files reside.

/usr/spool/mqueue/qf*
: Control (queue) files for messages.

/usr/spool/mqueue/df*
: Data files.

/usr/spool/mqueue/lf*
: Lock files

/usr/spool/mqueue/tf*
: Temporary versions of the qf files, used during queue file rebuild.

/usr/spool/mqueue/nf*
: A file used when creating a unique id.

/usr/spool/mqueue/xf*
: A transcript of the current session.

PART 3: COMMUNICATIONS

The three articles in this part cover a range of communications topics, from general background information to detailed descriptions of program structures and protocols. They describe the interprocess communication software (this can be either interactive or batch) and *sendmail*, an internetwork mail server.

Ftp, *telnet*, and the *r*-command set are three other networking software utilities available on the ULTRIX-32 system but not mentioned in these articles. See the end of this introduction for a brief description of each.

Interprocess Communication

The first two articles describe the socket software, a set of system calls (new with the 4.2BSD distribution) used for interprocess communication. The communicating processes can be running on the same computer or on separate computers linked by the DARPA standard communication protocols.

Interprocess communication requires each process to set up one of three types of socket:

Stream socket	Communication is bidirectional, reliable, sequenced, and unduplicated.
Datagram socket	Communication is bidirectional but not promised to be reliable, sequenced, or unduplicated.
Raw socket	Communication is possible through access to underlying protocols.

"A 4.2BSD Interprocess Communication Primer" gives the format for each socket-related call and explains how to coordinate the calls to establish a connection and send and receive messages:

- *Create* a socket
- *Bind* a name to a socket
- *Connect* - initiate a connection
- *Listen* for a connect request
- *Accept* a connect request
- *Write* a message
- *Read* a message
- *Send* a message
- *Receive* a message
- *Sendto* - send a datagram message
- *Recvfrom* - receive a datagram message
- *Close* a connection
- *Shutdown* a connection
- *Select* - multiplex the transfer of messages

These commands are listed individually in the *ULTRIX-32 Programmer's Manual.* The article also tells how to use: a library of routines that manipulate addresses, server and client calls, and connectionless servers. And information on a variety of advanced topics is available for sophisticated users.

The second article, "4.2BSD Networking Implementation Notes," describes the internal structure of the interprocess communication software. This information should be useful to engineers who are developing new communication protocols and network utilities. The article explains:

- Support for multiple protocol families and addressing styles
- Structures for internal address representation
- Memory management for network functions
- Internal layering
- Protocol interfaces
- Gateways
- Routing tables
- Use of raw sockets for direct access to low level protocols
- Buffering issues
- Handling out-of-band data
- Use of trailer protocols

You can also find a description of the user interface to the interprocess communication software in Section 2.3 of the "4.2BSD System Manual" (in Volume II of this set). Prefer the more recent "4.2BSD Interprocess Communication Primer" when you find discrepancies.

Sendmail

The article by Allman, "Sendmail - An Internetwork Mail Router," offers good background information for people who install and maintain the *sendmail* utility. For actual instructions on installation, see the "Sendmail Installation and Operation Guide" in Part 2 of this volume.

Sendmail acts like a post office, enabling different networking systems to route mail between them. For example, people using the ARPANET and others using the ETHERNET can send mail to each other, and *sendmail* will cooperate with the network software at each end to make sure that the messages get through. The *sendmail* functions are transparent to people sending the messages; each sender or receiver needs to deal only with the interface to the local network used on his or her computer system.

A reading of this article is prerequisite to an understanding of the "Sendmail Installation and Operation Guide."

Standard Networking Utilities

Three other networking systems are available on ULTRIX-32:

ftp	File transfer program (a user interface to the ARPANET)
telnet	Remote login protocol
r-commands	New networking software layered on sockets

You can find information on these utilities in the *ULTRIX-32 Programmer's Manual. Ftp* and *telnet* are utilities that prompt you for commands when you run them. Users must give appropriate passwords when accessing information on remote systems. The command descriptions listed under *ftp* and *telnet* are comprehensive.

The *r*-commands, like the interprocess communication commands, are listed individually in the *ULTRIX-32 Programmer's Manual,* because you must call each one from the shell:

rcmd	Connect, then execute a command
rcp	Remote file copy
rdump	File system dump across a network
rexec	Remote execute
rlogin	Remote login
rmt	Remote magnetic tape dump
rrestore	Restore a system from a file system dump across a network
rshell	Remote shell; provide remote execution facilities
ruptime	Show how long a remote system has been up
rwho	Show who is on a remote system

However, the *r*-command software requires trust between system users, because remote access using the *r*-commands does not require use of passwords.

A 4.2BSD Interprocess Communication Primer

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

1. INTRODUCTION

One of the most important parts of 4.2BSD is the interprocess communication facilities. These facilities are the result of more than two years of discussion and research. The facilities provided in 4.2BSD incorporate many of the ideas from current research, while trying to maintain the UNIX* philosophy of simplicity and conciseness. It is hoped that the interprocess communication facilities included in 4.2BSD will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4.2BSD facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the ipc facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact these facilities have been tied to the UNIX file system; either through naming, or implementation. Consequently, the ipc facilities provided in 4.2BSD have been designed as a totally independent subsystem. The 4.2BSD ipc allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to included more than the simple byte stream provided by a pipe-like entity. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

The remainder of this document is organized in four sections. Section 2 introduces the new system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the ipc facilities.

---

* UNIX is a Trademark of Bell Laboratories.

2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.2BSD ipc supports two separate communication domains: the UNIX domain, and the Internet domain is used by processes which communicate using the the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the possible error conditions which are possible when operating in the Internet domain.

2.1. Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes*.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

Two potential socket types which have interesting properties are the *sequenced packet* socket and the *reliably delivered message* socket. A sequenced packet socket is identical to a stream socket with the exception that record boundaries are preserved. This interface is very similar to that provided by the Xerox NS Sequenced Packet protocol. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. While these two socket types have been loosely defined, they are currently unimplemented in 4.2BSD. As such, in this document we will concern ourselves only with the three socket types for which support exists.

---

* In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

2.2. Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use a sample call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To obtain a particular protocol one selects the protocol number, as defined within the communication domain. For the Internet domain the available protocols are defined in *<netinet/in.h>* or, better yet, one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p proto);
```

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. The *bind* call is used to assign a name to a socket:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). In the UNIX domain names are path names while in the Internet domain names contain an Internet address and port number. If one wanted to bind the name "/dev/foo" to a UNIX domain socket, the following would be used:

* The manifest constants are named AF whatever as they indicate the "address format" to use in interpreting names.

```
bind(s, "/dev/foo", sizeof ("/dev/foo") − 1);
```

(Note how the null byte in the name is not counted as part of the name.) In binding an Internet address things become more complicated. The actual call is simple,

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

2.4. Connection establishment

With a bound socket it is possible to rendezvous with an unrelated process. This operation is usually asymmetric with one process a "client" and the other a "server". The client requests services from the server by initiating a "connection" to the server's socket. The server, when willing to offer its advertised services, passively "listens" on its socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
connect(s, "server-name", sizeof ("server-name"));
```

while in the Internet domain,

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket; c.f. section 5.4. An error is returned when the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

Many errors can be returned when a connection attempt fails. The most common are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. When connecting to a host running 4.2BSD this is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down. In these cases the system is conservative and indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be zero.

Accept normally blocks. That is, the call to accept will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the accept call there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are useable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

SOF_OOB	send/receive out of band data
SOF_PREVIEW	look at data without reading
SOF_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When

SOF_PREVIEW is specified with a *recv* call, any data present is returned to the user, but treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received. Applying shutdown to a socket causes any data queued to be immediately discarded.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to it in order that the recipient of a message may identify the sender. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the intended recipient of the message. When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. Where information is present locally to recognize a message which may never be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket (i.e. no multi-casting). Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket where a connect request initiates establishment of an end to end connection). Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

Select takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or-ing bits of the form "1 << fd". That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The parameter *nfds* specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of −1 is returned along with the error number EINTR.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.2BSD networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we hope the same user interface will be maintained in accessing network-related address data bases. The only difference should be the values returned to the user. Since these values are normally supplied the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings is likely to vary between network architectures. For instance, it is desirable for a network to not require hosts be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

3.1. Host names

A host name to address mapping is represented by the *hostent* structure:

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    *h_addr;        /* address */
};
```

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine *gethostbyname*(3N) takes a host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps host addresses into a *hostent* structure. It is possible for a host to have many addresses, all having the same name. *Gethostybyname* returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine *gethostent*(3N) may be used. For example, to obtain a *hostent* structure for a host on a particular network the following routine might be used (for simplicity, only Internet addresses are considered):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h name)) {
                        for (cp = hp->h aliases; cp && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                goto found;
                        continue;
                }
        found:
                if (in netof(*(struct in addr *)hp->h addr)) == net)
                        break;
        }
        endhostent(0);
        return (hp);
}
```

(*in netof*(3N) is a standard routine which returns the network portion of an Internet address.)

3.2. Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char        *n name;        /* official name of net */
        char        **n aliases;    /* alias list */
        int         n addrtype;     /* net address type */
        int         n net;          /* network # */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above.

3.3. Protocol names

For protocols the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char        *p_name;        /* official protocol name */
        char        **p_aliases;    /* alias list */
        int         p_proto;        /* protocol # */
};
```

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the "gethostbynameandnet" routine described above. A service mapping is described by the *servent* structure,

```
struct  servent {
        char        *s_name;        /* official service name */
        char        **s_aliases;    /* alias list */
        int         s_port;         /* port # */
        char        *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

3.5. Miscellaneous

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile. Perhaps when the system is adapted to different network architectures the utilities will be reorganized more cleanly.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
main(argc, argv)
        char *argv[];
{
        struct sockaddr in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h addr, (char *)&sin.sin addr, hp->h length);
        sin.sin family = hp->h addrtype;
        sin.sin port = sp->s port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

usually reversed. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s port));
```

On machines other than the VAX these routines are defined as null macros.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well know address for service requests. Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a data base and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication. However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous. The *portal* notion described in the "4.2BSD System Manual" embodies many of the ideas found in Courier, with the rendezvous mechanism implemented internal to the system.

4.1. Servers

In 4.2BSD most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time it advertises it services by listening at a well know location. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its

```
main(argc, argv)
        int argc;
        char **argv;
{
        int f;
        struct sockaddr in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        <<disassociate server from controlling terminal>>
#endif
        ...
        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                perror("rlogind: accept");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2. Remote login server.

expected location. The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        if (fork() == 0) {
                close(f);
                doit(g, &from);
        }
        close(g);
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established. With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the accept is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides:

```
bzero((char *)&sin, sizeof (sin));
bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
...
if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

```
arpa      up   9:45,      5 users, load  1.15,  1.39,  1.31
cad       up   2+12:04,   8 users, load  4.67,  5.13,  4.59
calder    up   10:10,     0 users, load  0.27,  0.15,  0.14
dali      up   2+06:28,   9 users, load  1.04,  1.20,  1.65
degas     up   25+09:48,  0 users, load  1.49,  1.43,  1.41
ear       up   5+00:05,   0 users, load  1.51,  1.54,  1.56
ernie   down   0:24
esvax   down   17:04
ingres  down   0:26
kim       up   3+09:16,   8 users, load  2.03,  2.46,  3.11
matisse   up   3+06:18,   0 users, load  0.03,  0.03,  0.05
medea     up   3+09:39,   2 users, load  0.35,  0.37,  0.50
merlin  down   19+15:37
miro      up   1+07:20,   7 users, load  4.59,  3.28,  2.12
monet     up   1+00:43,   2 users, load  0.22,  0.09,  0.07
oz      down   16:09
statvax   up   2+15:57,   3 users, load  1.52,  1.81,  1.86
ucbvax    up   9:34,      2 users, load  6.08,  5.16,  3.28
```

Figure 3. ruptime output.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update

a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        bind(s, &sin, sizeof (sin));
        ...
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                perror("rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port\n",
                                ntohs(from.sin port));
                        continue;
                }
                ...
                if (!verify(wd.wd hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from %x\n",
                                ntohl(from.sin addr.s addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd hostname);
                whod = open(path, FWRONLY|FCREATE|FTRUNCATE, 0666);
                ...
                (void) time(&wd.wd recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet

does, however, indicates some problems with the current protocol.

Status information is broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

The second problem with the current scheme is that the rwho process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.2BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information†. Unfortunately, no straightforward mechanism currently exists for finding the collection of networks to which a host is directly connected. Thus the rwho server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is to interrogate the system data structures to locate those directly connected networks. A mechanism to acquire this information from the system would be a useful addition.

---

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

† An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the ipc the mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with the SIGURG signal. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out of band message the SOF_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data SOF_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5.

5.2. Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        signal(SIGURG, oob);
        /* flush local terminal input and output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, SOF_OOB);
        ...
}
```

Figure 5. Flushing terminal i/o on receipt of out of band data.

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

5.3. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a *pseudo terminal*. A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side. The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *killpg*(2) on the appropriate process group.

5.4. Internet address binding

Binding addresses to sockets in the Internet domain can be fairly complex. Communicating processes are bound by an *association*. An association is composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique. That is, there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The bind system call allows a process to specify half of an association, <local address, local port>, while the connect and accept primitives are used to complete a socket's

association. Since the association is created in two steps the association uniqueness requirement indicated above could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin family = AF_INET;
sin.sin addr.s addr = INADDR_ANY;
sin.sin port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, if a host is on a networks 46 and 10 and a socket is bound as above, then an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example:

```
sin.sin addr.s addr = MYADDRESS;
sin.sin port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. The first is that ports numbered 0 through 1023 are reserved for privileged users (i.e. the super user). The second is that the port number is not currently bound to some other socket. In order to find a free port number in the privileged range the following code is used by the remote shell server:

```
struct sockaddr_in sin;
...
lport = IPPORT_RESERVED - 1;
sin.sin addr.s addr = INADDR_ANY;
...
for (;;) {
        sin.sin port = htons((u short)lport);
        if (bind(s, (caddr t)&sin, sizeof (sin)) >= 0)
                break;
        if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
                perror("socket");
                break;
        }
        lport--;
        if (lport == IPPORT RESERVED/2) {
                fprintf(stderr, "socket: All ports in use\n");
                break;
        }
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is due to associations being created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket were around. To override the default port selection algorithm then an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SQ_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port (if an association already exists, the error EADDRINUSE is returned).

Local address binding by the system is currently done somewhat haphazardly when a host is on multiple networks. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 were arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46. This unfortunately, is not always the case. For reasons too complicated to discuss here, the local address bound may be appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected*.

* For example, if network 46 were unknown to the host on network 32, and the local address were bound to that located on network 46, then even though a route between the two hosts existed through network 10, a connection would fail.

5.5. Broadcasting and datagram sockets

By using a datagram socket it is possible to send broadcast packets on many networks supported by the system (the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software). Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to the super user.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
dst.sin_addr.s addr = INADDR_ANY;
dst.sin_port = DESTPORT;
```

and, finally, a sendto call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the senders address and port (datagram sockets are anchored before a message is allowed to go out).

5.6. Signals

Two new signals have been added to the system which may be used in conjunction with the interprocess communication facilities. The SIGURG signal is associated with the existence of an "urgent condition". The SIGIO signal is used with "interrupt driven i/o" (not presently implemented). SIGURG is currently supplied a process when out of band data is present at a socket. If multiple sockets have out of band data awaiting delivery, a select call may be used to determine those sockets with such data.

An old signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any children processes have changed state. Normally servers use the signal to "reap" child processes after exiting. For example, the remote login server loop shown in Figure 2 may be augmented as follows:

```
int reaper();
 ...
sigset(SIGCHLD, reaper);
listen(f, 10);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len, 0);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

4.2BSD Networking Implementation Notes

Revised July, 1983

Samuel J. Leffler, William N. Joy, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

(415) 642-7780

1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *4.2BSD System Manual* [Joy82a]. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

---

2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
        short sa_family; /* data format identifier */
        char  sa_data[14];      /* address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The *sa_family* field indicates which address family the address belongs to, the *sa data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats*.

* Later versions of the system support variable length addresses.

5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbuf*'s. An mbuf is a structure of the form:

```
struct mbuf {
        struct      mbuf *m_next;       /* next buffer in chain */
        u_long      m_off;              /* offset of data */
        short m_len;                /* amount of data in this mbuf */
        short m_type;               /* mbuf type (accounting) */
        u_char      m_dat[MLEN];        /* data storage */
        struct      mbuf *m_act;        /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbufs to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat*. The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define    mtod(x,t)  ((t)((int)(x) + (x)->m off))
```

(note the *t* parameter, a C type cast, is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. The virtual addresses of these data pages precede those of mbufs, so when pages of data are separated from an mbuf, the mbuf data offset is a negative value. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the relevant page table entries in the data page map and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware if data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following utility routines are available for manipulating mbuf chains:

m = m copy(m0, off, len);
: The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);
: The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);
: The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation, alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);
: After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is

guaranteed to have at least *size* bytes of data in contiguous memory (allowing access via a pointer, obtained using the *mtod* macro). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring mbufs always reside on 128 byte boundaries it is possible to always locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define     dtom(x)     ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets, as well as memory allocated for packets. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(1) program.

6. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces each must conform to.

6.1. Socket layer

The socket layer deals with the interprocess communications facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *4.2BSD System Manual* are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short           so_type;        /* generic type */
        short           so_options;     /* from socket call */
        short           so_linger;      /* time to linger while closing */
        short           so_state;       /* internal state flags */
        caddr_t         so_pcb;         /* protocol control block */
        struct          protosw *so_proto; /* protocol handle */
        struct          socket *so_head; /* back pointer to accept socket */
        struct          socket *so_q0;  /* queue of partial connections */
        short           so_q0len;       /* partials on so_q0 */
        struct          socket *so_q;   /* queue of incoming connections */
        short           so_qlen;        /* number of connections on so q */
        short           so_qlimit;      /* max number queued connections */
        struct          sockbuf so_snd; /* send queue */
        struct          sockbuf so_rcv; /* receive queue */
        short           so_timeo;       /* connection timeout */
        u_short         so_error;       /* error affecting connection */
        short           so_oobmark;     /* chars to oob mark */
        short           so_pgrp;        /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol specific data structure, the "protocol control block" is also present in the socket structure. Protocols control this data structure and it normally includes a back pointer to the parent socket structure(s) to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queueing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as an "wait channel' to be used in notification. When data arrives for the process and is placed in the

socket's queue, the blocked process is identified by the fact it is waiting "on the queue".

6.1.1. Socket state

A socket's state is defined from the following:

```
#define SS_NOFDREF              0x001   /* no file table ref any more */
#define SS_ISCONNECTED          0x002   /* socket connected to a peer */
#define SS_ISCONNECTING         0x004   /* in process of connecting to peer */
#define SS_ISDISCONNECTING      0x008   /* in process of disconnecting */
#define SS_CANTSENDMORE         0x010   /* can't send more data to peer */
#define SS_CANTRCVMORE          0x020   /* can't receive more data from peer */
#define SS_CONNAWAITING         0x040   /* connections awaiting acceptance */
#define SS_RCVATMARK            0x080   /* at mark on input */

#define SS_PRIV                 0x100   /* privileged */
#define SS_NBIO                 0x200   /* non-blocking ops */
#define SS_ASYNC                0x400   /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created the state is defined based on the type of input/output the user wishes to perform. "Non-blocking" I/O implies a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULDBLOCK (the service request may be partially fulfilled, e.g. a request for more data than is present).

If a process requested "asynchronous" notification of events related to the socket the SIGIO signal is posted to the process. An event is a change in the socket's state, examples of such occurances are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "priviledged" if it was created by the super-user. Only priviledged sockets may send broadcast packets, or bind addresses in priviledged portions of an address space.

6.1.2. Socket data queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
        short           sb_cc;          /* actual chars in buffer */
        short           sb_hiwat;       /* max actual char count */
        short           sb_mbcnt;       /* chars of mbufs used */
        short           sb_mbmax;       /* max chars of mbufs to use */
        short           sb_lowat;       /* low water mark */
        short           sb_timeo;       /* timeout */
        struct          mbuf *sb_mb;    /* the mbuf chain */
        struct          proc *sb_sel;   /* process selecting read/write */
        short           sb_flags;       /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of characters as well as high and low water marks are used by the protocols in controlling the flow of data. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but is assumed this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources;

```
#define   SB_LOCK    0x01   /* lock on data queue (so rcv only) */
#define   SB_WANT    0x02   /* someone is waiting to lock */
#define   SB_WAIT    0x04   /* someone is waiting for data/space */
#define   SB_SEL     0x08   /* buffer is selected */
#define   SB_COLL    0x10   /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

6.1.3. Socket connection queueing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two sides are considered distinct. One side is termed *active*, and generates connection requests. The other side is called *passive* and accepts connection requests.

From the passive side, a socket is created with the option SO_ACCEPTCONN specified, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn*(). When the connection is established, the socket structure is then transfered to *so_q*, making it available for an accept.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped.

6.2. Protocol layer(s)

Protocols are described by a set of entry points and certain socket visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
                        short   pr_type;            /* socket type used for */
                        short   pr_family;          /* protocol family */
                        short   pr_protocol;        /* protocol number */
                        short   pr_flags;           /* socket visible attributes */
/* protocol-protocol hooks */
                        int     (*pr_input)();      /* input to protocol (from below) */
                        int     (*pr_output)();     /* output to protocol (from above) */
                        int     (*pr_ctlinput)();   /* control input (from below) */
                        int     (*pr_ctloutput)();  /* control output (from above) */
/* user-protocol hook */
                        int     (*pr_usrreq)();     /* user request */
/* utility hooks */
                        int     (*pr_init)();       /* initialization routine */
                        int     (*pr_fasttimo)();   /* fast timeout (200ms) */
                        int     (*pr_slowtimo)();   /* slow timeout (500ms) */
                        int     (*pr_drain)();      /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any the arguments to these entries and must dispose of it.

The *pr_userreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define     PR_ATOMIC           0x01    /* exchange atomic messages only */
#define     PR_ADDR             0x02    /* addresses given with messages */
#define     PR_CONNREQUIRED     0x04    /* connection required by protocol */
#define     PR_WANTRCVD         0x08    /* want PRU RCVD calls */
#define     PR_RIGHTS           0x10    /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notfiy the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates the protocol supports the passing of capabilities; this is currently used only the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_family* field indicates which protocol family the protocol belongs to. The *pr_protocol* field contains the protocol number of the protocol, normally a well known value.

6.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and deencapsulation of any low level header information required to deliver a message to it's destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. Each interface normally identifies itself at boot time to the routing module so that it may be selected for packet delivery.

An interface is defined by the following structure,

```
struct ifnet {
        char            *if_name;       /* name, e.g. "en" or "lo" */
        short           if_unit;        /* sub-unit for lower level driver */
        short           if_mtu;         /* maximum transmission unit */
        int             if_net;         /* network number of interface */
        short           if_flags;       /* up/down, broadcast, etc. */
        short           if_timer;       /* time 'til if_watchdog called */
        int             if_host[2];     /* local net host number */
        struct          sockaddr if_addr; /* address of interface */
        union {
                        struct          sockaddr ifu_broadaddr;
                        struct          sockaddr ifu_dstaddr;
        } if ifu;
        struct          ifqueue if_snd; /* output queue */
        int             (*if_init)();   /* init routine */
        int             (*if_output)(); /* output routine */
        int             (*if_ioctl)();  /* ioctl routine */
        int             (*if_reset)();  /* bus reset routine */
        int             (*if_watchdog)(); /* timer routine */
        int             if_ipackets;    /* packets received on interface */
        int             if_ierrors;     /* input errors on interface */
        int             if_opackets;    /* packets sent on interface */
        int             if_oerrors;     /* output errors on interface */
        int             if_collisions;  /* collisions on csma interfaces */
        struct          ifnet *if_next;
};
```

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*, which should be called every *if_timer* seconds (if non-zero).

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define IFF_UP              0x1     /* interface is up */
#define IFF_BROADCAST       0x2     /* broadcast address valid */
#define IFF_DEBUG           0x4     /* turn on debugging */
#define IFF_ROUTE           0x8     /* routing entry installed */
#define IFF_POINTOPOINT     0x10    /* interface is point-to-point link */
#define IFF_NOTRAILERS      0x20    /* avoid use of trailers */
#define IFF_RUNNING         0x40    /* resources allocated */
#define IFF_NOARP           0x80    /* no address resolution protocol */
```

If the interface is connected to a network which supports transmission of *broadcast* packets,

the IFF_BROADCAST flag will be set and the *if_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point to point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *if_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets; *trailer* protocols are described in section 14. The IFF_NOARP flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses.

The information stored in an *ifnet* structure for point to point communication devices is not currently used by the system internally. Rather, it is used by the user level routing process in determining host network connections and in initially devising routes (refer to chapter 10 for more information).

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used to initially define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration.

6.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilizes a structure of the following form:

```
struct  ifuba {
        short       ifu_uban;               /* uba number */
        short       ifu_hlen;               /* local net header length */
        struct      uba_regs *ifu_uba;      /* uba regs, in vm */
        struct ifrw {
                    caddr_t     ifrw_addr;  /* virt addr of header */
                    int         ifrw_bdp;   /* unibus bdp */
                    int         ifrw_info;  /* value from ubaalloc */
                    int         ifrw_proto; /* map register prototype */
                    struct      pte *ifrw_mr;/* base of map registers */
        } ifu_r, ifu_w;
        struct      pte ifu_wmap[IF MAXNUBAMR];/* base pages for output */
        short       ifu_xswapd;             /* mask of clusters swapped */
        short       ifu_flags;              /* used during uballoc's */
        struct      mbuf *ifu_xtofree;      /* pages being dma'd out */
};
```

The *if_uba* structure describes UNIBUS resources held by an interface. IF NUBAMR map registers are held for datagram data, starting at *ifr_mr*. UNIBUS map register *ifr_mr*[−1] maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifr_bdp*. The prototype of the map registers for read and for write is saved in *ifr_proto*.

When write transfers are not full pages on page boundaries the data is just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is of a (1024 byte) page size and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifu_wmap* when the transfer completes.

When read transfers give whole pages of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers, all use the *ifuba* structure described above.

if_ubainit(ifu, uban, hlen, nmr);

if_ubainit allocates resources on UNIBUS adaptor *uban* and stores the resultant information in the *ifuba* structure pointed to by *ifu*. It is called only at boot time or after a UNIBUS reset. Two data paths (buffered or unbuffered, depending on the *ifu_flags* field) are allocated, one for reading and one for writing. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if wubaput* below). If *if_ubainit* is called with resources already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization is successful, 0 otherwise.

m = if_rubaget(ifu, totlen, off0);

if_rubaget pulls read data off an interface. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When page sized units of data are present and are page-aligned, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages; thus avoiding any copying. A 0 return value indicates a failure to allocate resources.

if_wubaput(ifu, m);

if_wubaput maps a chain of mbufs onto a network interface in preparation for output. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Any other mbufs which contained non-page sized data portions are also copied to the I/O space. Pages mapped from a previous output operation (no longer needed) are unmapped and returned to the network page pool.

7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define    PRU_ATTACH 0     /* attach protocol */
#define    PRU_DETACH 1     /* detach protocol */
#define    PRU_BIND   2     /* bind socket to address */
#define    PRU_LISTEN 3     /* listen for connection */
#define    PRU_CONNECT      4     /* establish connection to peer */
#define    PRU_ACCEPT 5     /* accept connection from peer */
#define    PRU_DISCONNECT 6     /* disconnect from peer */
#define    PRU_SHUTDOWN 7     /* won't send any more data */
#define    PRU_RCVD   8     /* have taken data; more room now */
#define    PRU_SEND   9     /* send this data */
#define    PRU_ABORT  10    /* abort (fast DISCONNECT, DETATCH) */
#define    PRU_CONTROL      11    /* control operations on protocol */
#define    PRU_SENSE  12    /* return status into m */
#define    PRU_RCVOOB 13    /* retrieve out of band data */
#define    PRU_SENDOOB      14    /* send out of band data */
#define    PRU_SOCKADDR     15    /* fetch socket's address */
#define    PRU_PEERADDR     16    /* fetch peer's address */
#define    PRU_CONNECT2     17    /* connect two sockets */
/* begin for protocols internal use */
#define    PRU_FASTTIMO     18    /* 200ms timeout */
#define    PRU_SLOWTIMO     19    /* 500ms timeout */
#define    PRU_PROTORCV     20    /* receive from below */
#define    PRU_PROTOSEND    21    /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr usrreq)(up, req, m, addr, rights);
int error; struct socket *up; int req; struct mbuf *m, *rights; caddr_t addr;
```

The mbuf chain, *m*, and the address are optional parameters. The *rights* parameter is an optional pointer to an mbuf chain containing user specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of both mbuf chains. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH

When a protocol is bound to a socket (with the *socket* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH

This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND

When a socket is initially created it has no address bound to it. This request indicates an address should be bound to an existing socket. The protocol module must verify the requested address is valid and available for use.

PRU_LISTEN

The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to

accept a connection.

PRU_CONNECT
: The "connect" request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel80b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel79], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT
: Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT
: Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN
: This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown.

PRU_RCVD
: This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND
: Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate a single user send request must be translated into a single PRU SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT
: This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL
: The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention).

PRU_SENSE
: The "sense" request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. There currently is no common format for the status returned. Information which might be returned includes per-connection statistics,

protocol state, resources currently in use by the connection, the optimal transfer size for the connection (based on windowing information and maximum packet size). The *addr* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB
: Any "out-of-band" data presently available is to be returned. An mbuf is passed in to the protocol module and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf.

PRU_SENDOOB
: Like PRU SEND, but for out-of-band data.

PRU_SOCKADDR
: The local address of the socket is returned, if any is currently bound to the it. The address format (protocol specific) is returned in the *addr* parameter.

PRU_PEERADDR
: The address of the peer to which the socket is connected is returned. The socket must be in a SS ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2
: The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU SLOWTIMO).

PRU_FASTTIMO
: A "fast timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO
: A "slow timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV
: This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND
: This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

8. Protocol/protocol interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, "*in*ternet *p*rotocol *c*ontrol *b*lock", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on to the IP module:

```
error = ip_output(m, opt, ro, allowbroadcast);
int error; struct mbuf *m, *opt; struct route *ro; int allowbroadcast;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller). The final parameter, *allowbroadcast* is a flag indicating if the user is allowed to transmit a broadcast packet. This may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occured which could be immediately detected (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m);
struct mbuf *m;
```

Each mbuf list passed is a single packet to be processed by the protocol module.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission.

8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data). This routine, and the *pr_ctloutput* routine, have not been extensively developed, and thus suffer from a "clumsiness" that can only be improved as more demands are placed on it.

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, info);
int req; caddr_t info;
```

The *req* parameter is one of the following,

```
#define  PRC_IFDOWN              0    /* interface transition */
#define  PRC_ROUTEDEAD           1    /* select new route if possible */
#define  PRC_QUENCH              4    /* some said to slow down */
#define  PRC_HOSTDEAD            6    /* normally from IMP */
#define  PRC_HOSTUNREACH         7    /* ditto */
#define  PRC_UNREACH_NET         8    /* no route to network */
#define  PRC_UNREACH_HOST        9    /* no route to host */
#define  PRC_UNREACH_PROTOCOL    10   /* dst says bad protocol */
#define  PRC_UNREACH_PORT        11   /* bad port # */
#define  PRC_MSGSIZE             12   /* message size forced drop */
#define  PRC_REDIRECT_NET        13   /* net routing redirect */
#define  PRC_REDIRECT_HOST       14   /* host routing redirect */
#define  PRC_TIMXCEED_INTRANS    17   /* packet lifetime expired in transit */
#define  PRC_TIMXCEED_REASS      18   /* lifetime expired on reass q */
#define  PRC_PARAMPROB           19   /* header incorrect */
```

while the *info* parameter is a "catchall" value which is request dependent. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This routine is not currently used by any protocol modules.

9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases to be concerned with, transmission of a packet, and receipt of a packet; each will be considered separately.

9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate, or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable mediums, such as the Ethernet, "successful" transmission simply means the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are normally trivial in nature (no buffer space, address format not handled, etc.).

9.2. Packet reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In our system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued up for the protocol module and a VAX software interrupt is posted to initiate processing.

Three macros are available for queueing and dequeueing packets,

IF_ENQUEUE(ifq, m)
: This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
: This places a pointer to the packet at the head of queue *ifq* in *m*. A zero value will be returned in *m* if the queue is empty.

IF_PREPEND(ifq, m)
: This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to bump a count of the number of packets dropped and the offending packet dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long          rt_hash;        /* hash key for lookups */
        struct          sockaddr rt_dst;    /* destination net or host */
        struct          sockaddr rt_gateway;/* forwarding agent */
        short           rt_flags;       /* see below */
        short           rt_refcnt;      /* no. of references to structure */
        u_long          rt_use;         /* packets sent using route */
        struct          ifnet *rt_ifp;  /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to an "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (who's at the other end of the route), a gateway to send the packet to, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept for use in deciding between multiple routes to the same destination (see below), and a count of "held references" to the dynamically allocated structure is maintained to insure memory reclamation occurs only when the route is not in use. Finally a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will indicate the address of the eventual destination, while the local network header will indicate the address of the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates the route is to an "indirect" gateway agent and the local network header should be filled in from the *rt_gateway* field instead of *rt_dst*, or from the internetwork destination address.

It is assumed multiple routes to the same destination will not be present unless they are deemed *equal* in cost (the current routing policy process never installs multiple routes to the same destination). However, should multiple routes to the same destination exist, a request for a route will return the "least used" route based on the total number of packets sent along this route. This can result in a "ping-pong" effect (alternate packets taking alternate routes), unless protocols "hold onto" routes until they no longer find them useful; either because the destination has changed, or because the route is lossy.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure,

```
struct route {
        struct      rtentry *ro_rt;
        struct      sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until disposed of with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, currently allocate and free routes on each transmission.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address and the new gateway to that destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accesible from the host are ignored.

10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is

normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

One routing policy process has already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up to date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet GGP (Gateway-Gateway Protocol), may be accomplished using a similar process.

11. Raw sockets

A raw socket is a mechanism which allows users direct access to a lower level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, and (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control blocks

Every raw socket has a protocol control block of the following form,

```
struct rawcb {
        struct      rawcb *rcb_next;        /* doubly linked list */
        struct      rawcb *rcb_prev;
        struct      socket *rcb_socket;     /* back pointer to socket */
        struct      sockaddr rcb_faddr;     /* destination address */
        struct      sockaddr rcb_laddr;     /* socket's address */
        caddr t     rcb_pcb;                /* protocol specific stuff */
        short       rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The addresses are also used to filter packets on input; this will be described in more detail shortly. If any protocol specific information is required, it may be attached to the control block using the *rcb_pcb* field.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Another flag, RAW_DONTROUTE, indicates if routing should be performed on outgoing packets. If it is, a route is expected to be allocated for each "new" destination address. That is, the first time a packet is transmitted a route is determined, and thereafter each time the destination address stored in *rcb_route* differs from *rcb_faddr*, or *rcb_route.ro_rt* is zero, the old route is discarded and a new one allocated.

11.2. Input processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives packets to the raw input routine with the call:

```
raw input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw header {
      struct      sockproto raw_proto;
      struct      sockaddr raw_dst;
      struct      sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

11.3. Output processing

On output the raw *pr_usrreq* routine passes the packet and raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory management

The basic memory allocation routines place no restrictions on the amount of space which may be allocated. Any request made is filled until the system memory allocator starts refusing to allocate additional memory. When the current quota of memory is insufficient to satisfy an mbuf allocation request, the allocator requests enough new pages from the system to satisfy the current request only. All memory owned by the network is described by a private page table used in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple copies of a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, if possible, it is copied or remapped into logically contiguous pages of memory from the network page pool. Data smaller than the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines, though it is clear if one imposed an upper bound on the total amount of physical memory allocated to the network, reserving memory would become important.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth

network to a low bandwidth network). As a "defensive" mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system generates a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocols perogative to support larger sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and usually not stored in the socket's send queue. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user's address space into the system it is deposited in pages (if sufficient data is present to fill an entire page). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA'ing the eventual data for the user into a page aligned area of memory is impossible without a priori knowledge of the format (e.g. supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol*, places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion, and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct {
        short protocol;   /* original protocol no. */
        short length;     /* length of trailer */
};
```

The processing of the trailer protocol is very simple. On output, the local network header indicates a trailer encapsulation is being used. The protocol identifier also includes an indication of the number of data pages present (before the trailer protocol header). The trailer protocol header is initialized to contain the actual protocol and variable length header size, and appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.

Acknowledgements

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

References

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture.* Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; *Specification for the Interconnection of Host and IMP.* BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP. Internet Working Group, IEN Draft Clark-2. March 1982.

[DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture – General Description.* Order No. AA-K179A-TK. October 1980.

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81] International Organization for Standardization. *ISO Open Systems Interconnection – Basic Reference Model.* ISO/TC 97/SC 16 N 719. August 1981.

[Joy82a] Joy, W.; Cooper, E.; Fabry, R.; Leffler, S.; and McKusick, M.; *4.2BSD System Manual.* Computer Systems Research Group, Technical Report 5. University of California, Berkeley. Draft of September 1, 1982.

[Postel79] Postel, J., ed. *DOD Standard User Datagram Protocol.* Internet Working Group, IEN 88. May 1979.

[Postel80a] Postel, J., ed. *DOD Standard Internet Protocol.* Internet Working Group, IEN 128. January 1980.

[Postel80b] Postel, J., ed. *DOD Standard Transmission Control Protocol.* Internet Working Group, IEN 129. January 1980.

[Xerox81] Xerox Corporation. *Internet Transport Protocols.* Xerox System Integration Standard 028112. December 1981.

[Zimmermann80] Zimmermann, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications. Com-28(4); 425-432. April 1980.

SENDMAIL – An Internetwork Mail Router

Eric Allman†

Britton-Lee, Inc.
1919 Addison Street, Suite 105.
Berkeley, California 94704.

ABSTRACT

Routing mail through a heterogenous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both domain-based addressing and old-style *ad hoc* addresses. The production system is powerful enough to rewrite addresses in the message header to conform to the standards of a number of common target networks, including old (NCP/RFC733) Arpanet, new (TCP/RFC822) Arpanet, UUCP, and Phonenet. Sendmail also implements an SMTP server, message queueing, and aliasing.

Sendmail implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characterstics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require point-to-point routing, which simplifies the database update problem since only adjacent hosts must be entered into the system tables, while others use end-to-end addressing. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, resource} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was quickly expanded to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes only the logical organization of the

†A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

address space.

Sendmail is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 discusses the design goals for *sendmail.* Section 2 gives an overview of the basic functions of the system. In section 3, details of usage are discussed. Section 4 compares *sendmail* to other internet mail routers, and an evaluation of *sendmail* is given in section 5, including future plans.

1. DESIGN GOALS

Design goals for *sendmail* include:

(1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail [UNIX83], Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78a, Nowitz78b]. ARPANET mail [Crocker77a, Postel77] was also required.

(2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formated addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with UNIX mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.

(3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.

(4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ether nets [Metcalfe76]). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use. For example, the ARPANET is bringing up the TCP protocol to replace the old NCP protocol. No host at Berkeley runs both TCP and NCP, so it is necessary to look at the ARPANET host name to determine whether to route mail to an NCP gateway or a TCP gateway.

(5) Configuration should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.

(6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

(7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and

facilitates specialized functions (such as returning an "I am on vacation" message).

(8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1. The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*, which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

2. OVERVIEW

2.1. System Organization

Sendmail neither interfaces with the user nor does actual mail delivery. Rather, it collects a message generated by a user interface program (UIP) such as Berkeley *Mail*, MS [Crocker77b], or MH [Borden79], edits the message as required by the destination network, and calls appropriate mailers to do mail delivery or queueing for network transmission[1]. This discipline allows the insertion of new mailers at minimum cost. In this sense *sendmail* resembles the Message Processing Module (MPM) of [Postel79b].

2.2. Interfaces to the Outside World

There are three ways *sendmail* can communicate with the outside world, both in receiving and in sending mail. These are using the conventional UNIX argument

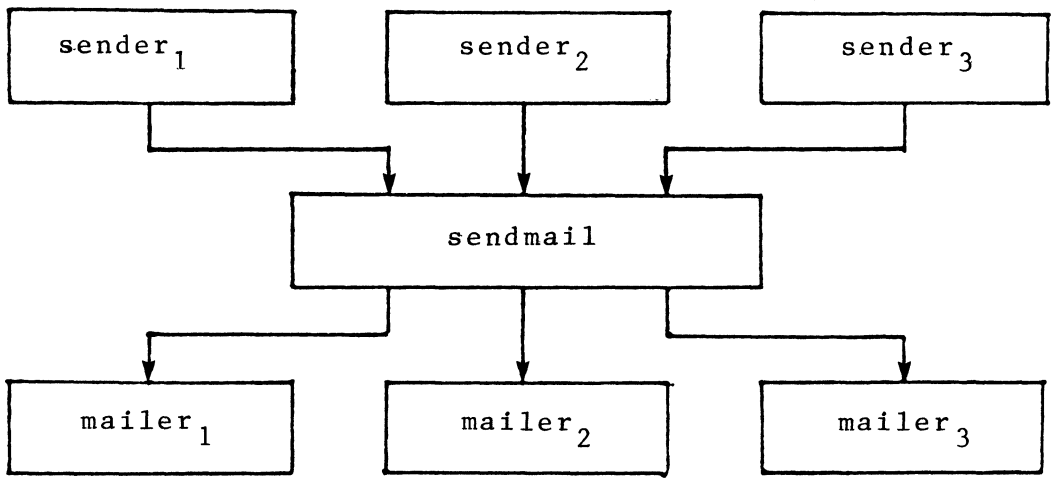

Figure 1 – Sendmail System Structure.

[1]except when mailing to a file, when *sendmail* does the delivery directly.

vector/return status, speaking SMTP over a pair of UNIX pipes, and speaking SMTP over an interprocess(or) channel.

2.2.1. Argument vector/exit status

This technique is the standard UNIX method for communicating with the process. A list of recipients is sent in the argument vector, and the message body is sent on the standard input. Anything that the mailer prints is simply collected and sent back to the sender if there were any problems. The exit status from the mailer is collected after the message is sent, and a diagnostic is printed if appropriate.

2.2.2. SMTP over pipes

The SMTP protocol [Postel82] can be used to run an interactive lock-step interface with the mailer. A subprocess is still created, but no recipient addresses are passed to the mailer via the argument list. Instead, they are passed one at a time in commands sent to the processes standard input. Anything appearing on the standard output must be a reply code in a special format.

2.2.3. SMTP over an IPC connection

This technique is similar to the previous technique, except that it uses a 4.2BSD IPC channel [UNIX83]. This method is exceptionally flexible in that the mailer need not reside on the same machine. It is normally used to connect to a sendmail process on another machine.

2.3. Operational Description

When a sender wants to send a message, it issues a request to *sendmail* using one of the three methods described above. *Sendmail* operates in two distinct phases. In the first phase, it collects and stores the message. In the second phase, message delivery occurs. If there were errors during processing during the second phase, *sendmail* creates and returns a new message describing the error and/or returns an status code telling what went wrong.

2.3.1. Argument processing and address parsing

If *sendmail* is called using one of the two subprocess techniques, the arguments are first scanned and option specifications are processed. Recipient addresses are then collected, either from the command line or from the SMTP RCPT command, and a list of recipients is created. Aliases are expanded at this step, including mailing lists. As much validation as possible of the addresses is done at this step: syntax is checked, and local addresses are verified, but detailed checking of host names and addresses is deferred until delivery. Forwarding is also performed as the local addresses are verified.

Sendmail appends each address to the recipient list after parsing. When a name is aliased or forwarded, the old name is retained in the list, and a flag is set that tells the delivery phase to ignore this recipient. This list is kept free from duplicates, preventing alias loops and duplicate messages deliverd to the same recipient, as might occur if a person is in two groups.

2.3.2. Message collection

Sendmail then collects the message. The message should have a header at the beginning. No formatting requirements are imposed on the message except that they must be lines of text (i.e., binary data is not allowed). The header is parsed and stored in memory, and the body of the message is saved in a temporary file.

To simplify the program interface, the message is collected even if no addresses were valid. The message will be returned with an error.

2.3.3. Message delivery

For each unique mailer and host in the recipient list, *sendmail* calls the appropriate mailer. Each mailer invocation sends to all users receiving the message on one host. Mailers that only accept one recipient at a time are handled properly.

The message is sent to the mailer using one of the same three interfaces used to submit a message to sendmail. Each copy of the message is prepended by a customized header. The mailer status code is caught and checked, and a suitable error message given as appropriate. The exit code must conform to a system standard or a generic message ("Service unavailable") is given.

2.3.4. Queueing for retransmission

If the mailer returned an status that indicated that it might be able to handle the mail later, *sendmail* will queue the mail and try again later.

2.3.5. Return to sender

If errors occur during processing, *sendmail* returns the message to the sender for retransmission. The letter can be mailed back or written in the file "dead.letter" in the sender's home directory[2].

2.4. Message Header Editing

Certain editing of the message header occurs automatically. Header lines can be inserted under control of the configuration file. Some lines can be merged; for example, a "From:" line and a "Full-name:" line can be merged under certain circumstances.

2.5. Configuration File

Almost all configuration information is read at runtime from an ASCII file, encoding macro definitions (defining the value of macros used internally), header declarations (telling sendmail the format of header lines that it will process specially, i.e., lines that it will add or reformat), mailer definitions (giving information such as the location and characteristics of each mailer), and address rewriting rules (a limited production system to rewrite addresses which is used to parse and rewrite the addresses).

To improve performance when reading the configuration file, a memory image can be provided. This provides a "compiled" form of the configuration file.

3. USAGE AND IMPLEMENTATION

3.1. Arguments

Arguments may be flags and addresses. Flags set various processing options. Following flag arguments, address arguments may be given, unless we are running in SMTP mode. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

(1) Anything in parentheses is thrown away (as a comment).

---

[2]Obviously, if the site giving the error is not the originating site, the only reasonable option is to mail back to the sender. Also, there are many more error disposition options, but they only effect the error message – the "return to sender" function is always handled in one of these two ways.

(2) Anything in angle brackets ("<>") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

user name <machine-address>

will send to the electronic "machine-address" rather than the human "user name."

(3) Double quotes (") quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently – for example, *user* and *"user"* are equivalent, but *\user* is different from either of them.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing[3].

3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program, or the MARS system [Sattley78]).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("|") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("/") the name is used as a file name, instead of a login name.

Files that have setuid or setgid bits set but no execute bits set have those bits honored if *sendmail* is running as root.

3.3. Aliasing, Forwarding, Inclusion

Sendmail reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

3.3.1. Aliasing

Aliasing maps names to address lists using a system-wide file. This file is indexed to speed access. Only names that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

3.3.2. Forwarding

After aliasing, recipients that are local and valid are checked for the existence of a ".forward" file in their home directory. If it exists, the message is *not* sent to that user, but rather to the list of users in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

"|/usr/local/newmail myname"

will use a different incoming mailer.

---

[3]Disclaimer: Some special processing is done after rewriting local names; see below.

3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77a] syntax:

:Include: pathname

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

project: :include:/usr/project/userlist

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line.

The header is formatted as a series of lines of the form

field-name: field-value

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

The body is a series of text lines. It is completely uninterpreted and untouched, except that lines beginning with a dot have the dot doubled when transmitted over an SMTP channel. This extra dot is stripped by the receiver.

3.5. Message Delivery

The send queue is ordered by receiving host before transmission to implement message batching. Each address is marked as it is sent so rescanning the list is safe. An argument list is built as the scan proceeds. Mail to files is detected during the scan of the send list. The interface to the mailer is performed using one of the techniques described in section 2.2.

After a connection is established, *sendmail* makes the per-mailer changes to the header and sends the result to the mailer. If any mail is rejected by the mailer, a flag is set to invoke the return-to-sender function after all delivery completes.

3.6. Queued Messages

If the mailer returns a "temporary failure" exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other salient parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is just the temporary file that was originally collected.

3.7. Configuration

Configuration is controlled primarily by a configuration file read at startup. *Sendmail* should not need to be recomplied except

(1) To change operating systems (V6, V7/32V, 4BSD).

(2) To remove or insert the DBM (UNIX database) library.

(3) To change ARPANET reply codes.

(4) To add headers fields requiring special processing.

Adding mailers or changing parsing (i.e., rewriting) or routing information does not require recompilation.

If the mail is being sent by a local user, and the file ".mailcf" exists in the sender's home directory, that file is read as a configuration file after the system configuration file. The primary use of this feature is to add header lines.

The configuration file encodes macro definitions, header definitions, mailer definitions, rewriting rules, and options.

3.7.1. Macros

Macros can be used in three ways. Certain macros transmit unstructured textual information into the mail system, such as the name *sendmail* will use to identify itself in error messages. Other macros transmit information from *sendmail* to the configuration file for use in creating other fields (such as argument vectors to mailers); e.g., the name of the sender, and the host and user of the recipient. Other macros are unused internally, and can be used as shorthand in the configuration file.

3.7.2. Header declarations

Header declarations inform *sendmail* of the format of known header lines. Knowledge of a few header lines is built into *sendmail*, such as the "From:" and "Date:" lines.

Most configured headers will be automatically inserted in the outgoing message if they don't exist in the incoming message. Certain headers are suppressed by some mailers.

3.7.3. Mailer declarations

Mailer declarations tell *sendmail* of the various mailers available to it. The definition specifies the internal name of the mailer, the pathname of the program to call, some flags associated with the mailer, and an argument vector to be used on the call; this vector is macro-expanded before use.

3.7.4. Address rewriting rules

The heart of address parsing in *sendmail* is a set of rewriting rules. These are an ordered list of pattern-replacement rules, (somewhat like a production system, except that order is critical), which are applied to each address. The address is rewritten textually until it is either rewritten into a special canonical form (i.e., a (mailer, host, user) 3-tuple, such as {arpanet, usc-isif, postel} representing the address "postel@usc-isif"), or it falls off the end. When a pattern matches, the rule is reapplied until it fails.

The configuration file also supports the editing of addresses into different formats. For example, an address of the form:

ucsfcgl!tef

might be mapped into:

tef@ucsfcgl.UUCP

to conform to the domain syntax. Translations can also be done in the other

direction.

3.7.5. Option setting

There are several options that can be set from the configuration file. These include the pathnames of various support files, timeouts, default modes, etc.

4. COMPARISON WITH OTHER MAILERS

4.1. Delivermail

Sendmail is an outgrowth of *delivermail.* The primary differences are:

(1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.

(2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.

(3) Forwarding and :include: features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).

(4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.

(5) A mail queue is provided in *sendmail.* Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.

(6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

4.2. MMDF

MMDF [Crocker79] spans a wider problem set than *sendmail.* For example, the domain of MMDF includes a "phone network" mailer, whereas *sendmail* calls on preexisting mailers in most cases.

MMDF and *sendmail* both support aliasing, customized mailers, message batching, automatic forwarding to gateways, queueing, and retransmission. MMDF supports two-stage timeout, which *sendmail* does not support.

The configuration for MMDF is compiled into the code[4].

Since MMDF does not consider backwards compatibility as a design goal, the address parsing is simpler but much less flexible.

It is somewhat harder to integrate a new channel[5] into MMDF. In particular, MMDF must know the location and format of host tables for all channels, and the channel must speak a special protocol. This allows MMDF to do additional verification (such as verifying host names) at submission time.

---

[4]Dynamic configuration tables are currently being considered for MMDF; allowing the installer to select either compiled or dynamic tables.

[5]The MMDF equivalent of a *sendmail* "mailer."

MMDF strictly separates the submission and delivery phases. Although *sendmail* has the concept of each of these stages, they are integrated into one program, whereas in MMDF they are split into two programs.

4.3. Message Processing Module

The Message Processing Module (MPM) discussed by Postel [Postel79b] matches *sendmail* closely in terms of its basic architecture. However, like MMDF, the MPM includes the network interface software as part of its domain.

MPM also postulates a duplex channel to the receiver, as does MMDF, thus allowing simpler handling of errors by the mailer than is possible in *sendmail.* When a message queued by *sendmail* is sent, any errors must be returned to the sender by the mailer itself. Both MPM and MMDF mailers can return an immediate error response, and a single error processor can create an appropriate response.

MPM prefers passing the message as a structured object, with type-length-value tuples[6]. Such a convention requires a much higher degree of cooperation between mailers than is required by *sendmail.* MPM also assumes a universally agreed upon internet name space (with each address in the form of a net-host-user tuple), which *sendmail* does not.

5. EVALUATIONS AND FUTURE PLANS

Sendmail is designed to work in a nonhomogeneous environment. Every attempt is made to avoid imposing unnecessary constraints on the underlying mailers. This goal has driven much of the design. One of the major problems has been the lack of a uniform address space, as postulated in [Postel79a] and [Postel79b].

A nonuniform address space implies that a path will be specified in all addresses, either explicitly (as part of the address) or implicitly (as with implied forwarding to gateways). This restriction has the unpleasant effect of making replying to messages exceedingly difficult, since there is no one "address" for any person, but only a way to get there from wherever you are.

Interfacing to mail programs that were not initially intended to be applied in an internet environment has been amazingly successful, and has reduced the job to a manageable task.

Sendmail has knowledge of a few difficult environments built in. It generates ARPANET FTP/SMTP compatible error messages (prepended with three-digit numbers [Neigus73, Postel74, Postel82]) as necessary, optionally generates UNIX-style "From" lines on the front of messages for some mailers, and knows how to parse the same lines on input. Also, error handling has an option customized for BerkNet.

The decision to avoid doing any type of delivery where possible (even, or perhaps especially, local delivery) has turned out to be a good idea. Even with local delivery, there are issues of the location of the mailbox, the format of the mailbox, the locking protocol used, etc., that are best decided by other programs. One surprisingly major annoyance in many internet mailers is that the location and format of local mail is built in. The feeling seems to be that local mail is so common that it should be efficient. This feeling is not born out by our experience; on the contrary, the location and format of mailboxes seems to vary widely from system to system.

The ability to automatically generate a response to incoming mail (by forwarding mail to a program) seems useful ("I am on vacation until late August....") but can create problems such as forwarding loops (two people on vacation whose programs send notes

[6]This is similar to the NBS standard.

back and forth, for instance) if these programs are not well written. A program could be written to do standard tasks correctly, but this would solve the general case.

It might be desirable to implement some form of load limiting. I am unaware of any mail system that addresses this problem, nor am I aware of any reasonable solution at this time.

The configuration file is currently practically inscrutable; considerable convenience could be realized with a higher-level format.

It seems clear that common protocols will be changing soon to accommodate changing requirements and environments. These changes will include modifications to the message header (e.g., [NBS80]) or to the body of the message itself (such as for multimedia messages [Postel80]). Experience indicates that these changes should be relatively trivial to integrate into the existing system.

In tightly coupled environments, it would be nice to have a name server such as Grapvine [Birrell82] integrated into the mail system. This would allow a site such as "Berkeley" to appear as a single host, rather than as a collection of hosts, and would allow people to move transparently among machines without having to change their addresses. Such a facility would require an automatically updated database and some method of resolving conflicts. Ideally this would be effective even without all hosts being under a single management. However, it is not clear whether this feature should be integrated into the aliasing facility or should be considered a "value added" feature outside *sendmail* itself.

As a more interesting case, the CSNET name server [Solomon81] provides an facility that goes beyond a single tightly-coupled environment. Such a facility would normally exist outside of *sendmail* however.

ACKNOWLEDGEMENTS

Thanks are due to Kurt Shoens for his continual cheerful assistance and good advice, Bill Joy for pointing me in the correct direction (over and over), and Mark Horton for more advice, prodding, and many of the good ideas. Kurt and Eric Schmidt are to be credited for using *delivermail* as a server for their programs (*Mail* and BerkNet respectively) before any sane person should have, and making the necessary modifications promptly and happily. Eric gave me considerable advice about the perils of network software which saved me an unknown amount of work and grief. Mark did the original implementation of the DBM version of aliasing, installed the VFORK code, wrote the current version of *rmail*, and was the person who really convinced me to put the work into *delivermail* to turn it into *sendmail*. Kurt deserves accolades for using *sendmail* when I was myself afraid to take the risk; how a person can continue to be so enthusiastic in the face of so much bitter reality is beyond me.

Kurt, Mark, Kirk McKusick, Marvin Solomon, and many others have reviewed this paper, giving considerable useful advice.

Special thanks are reserved for Mike Stonebraker at Berkeley and Bob Epstein at Britton-Lee, who both knowingly allowed me to put so much work into this project when there were so many other things I really should have been working on.

REFERENCES

[Birrell82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M. 25,* 4, April 82.

[Borden79] Borden, S., Gaines, R. S., and Shapiro, N. Z., *The MH Message Handling System: Users' Manual.* R-2367-PAF. Rand Corporation. October 1979.

[Crocker77a] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages.* RFC 733, NIC 41952. In [Feinler78]. November 1977.

[Crocker77b] Crocker, D. H., *Framework and Functions of the MS Personal Message System.* R-2134-ARPA, Rand Corporation, Santa Monica, California. 1977.

[Crocker79] Crocker, D. H., Szurkowski, E. S., and Farber, D. J., *An Internetwork Memo Distribution Facility – MMDF.* 6th Data Communication Symposium, Asilomar. November 1979.

[Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages.* RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Metcalfe76] Metcalfe, R., and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM 19,* 7. July 1976.

[Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook.* NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.

[NBS80] National Bureau of Standards, *Specification of a Draft Message Format Standard.* Report No. ICST/CBOS 80-2. October 1980.

[Neigus73] Neigus, N., *File Transfer Protocol for the ARPA Network.* RFC 542, NIC 17759. In [Feinler78]. August, 1973.

[Nowitz78a] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Nowitz78b] Nowitz, D. A., *Uucp Implementation Description.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. October, 1978.

[Postel74] Postel, J., and Neigus, N., Revised FTP Reply Codes. RFC 640, NIC 30843. In [Feinler78]. June, 1974.

[Postel77] Postel, J., *Mail Protocol.* NIC 29588. In [Feinler78]. November 1977.

[Postel79a] Postel, J., *Internet Message Protocol.* RFC 753, IEN 85. Network Information Center, SRI International, Menlo Park, California. March 1979.

[Postel79b] Postel, J. B., *An Internetwork Message Structure.* In *Proceedings of the Sixth Data Communications Symposium,* IEEE. New York. November 1979.

[Postel80] Postel, J. B., *A Structured Format for Transmission of Multi-Media Documents.* RFC 767. Network Information Center, SRI International, Menlo Park, California. August 1980.

[Postel82] Postel, J. B., *Simple Mail Transfer Protocol.* RFC821 (obsoleting RFC788). Network Information Center, SRI International, Menlo Park, California. August 1982.

[Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network.* University of California, Berkeley California. 1979.

[Shoens79] Shoens, K., *Mail Reference Manual.* University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

[Sluizer81] Sluizer, S., and Postel, J. B., *Mail Transfer Protocol.* RFC 780. Network Information Center, SRI International, Menlo Park, California. May 1981.

[Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., "The Design of the CSNET Name Server." CS-DN-2, University of Wisconsin, Madison. November 1981.

[Su82] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications.* RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.

[UNIX83] *The UNIX Programmer's Manual, Seventh Edition,* Virtual VAX-11 Version, Volume 1. Bell Laboratories, modified by the University of California, Berkeley, California. March, 1983.

PART 4: SECURITY CONSIDERATIONS

Security on the ULTRIX-32 system is flexible and reasonably comprehensive. These two articles describe a number of measures you can take to make your installation moderately secure. The first article in this part, "On the Security of UNIX," explains the major features and weaknesses of ULTRIX-32 system security. The second article, "Password Security: A Case History," tells how the password facility used on the ULTRIX-32 system was developed.

Protection Against Crashes and Unauthorized Access

Unrestricted use of disk space on the ULTRIX-32 system may cripple or stop the operating system, and Ritchie indicates in "On the Security of UNIX" that the software cannot be protected from this type of abuse. However, the ULTRIX-32 system does include the *quota* utility, which enables the system manager to control the use of resources by limiting the number of blocks and the number of files available to each user. See "Disk Quotas in a UNIX Environment" in Part 2 of this volume.

The Ritchie article explains the functions of the file protection bits, the user identification number (UID), and the user-group identification number (GID); these functions allow users to control access to their files.

In addition, Ritchie outlines the ULTRIX-32 system schemes for:

- Data encryption
- Password security
- Precautions concerning the superuser account, set-UID programs, mail, and the *mount* command

Password Security Development

"Password Security: A Case History," by Morris and Thompson, outlines the objectives of the password system on the ULTRIX-32 system:

- To protect the system against unauthorized users
- To prevent logged-in users from performing unauthorized functions
- To minimize inconvenience to legitimate users

The case history describes the early, rejected password schemes and their weaknesses, and it shows how they evolved to the current scheme. You may find the article useful as well as interesting, because it offers well-researched precautions to users and tested recommendations to system managers.

On the Security of UNIX

Dennis M. Ritchie

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX† system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls— there may be bugs in this area, but none are known— but rather in lack of checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
while : ; do
    mkdir x
    cd x
done
```

Either a panic will occur because all the i-nodes on the device are used up, or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

In this version of the system, users are prevented from creating more than a set number of processes simultaneously, so unless users are in collusion it is unlikely that any one can stop the system altogether. However, creation of 20 or so CPU or disk-bound jobs leaves few resources available for others. Also, if many large jobs are run simultaneously, swap space may run out, causing a panic.

It should be evident that excessive consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group identification number (UID and GID). Nine of the protection bits are used to specify independently permission to read, to write, and to

† UNIX is a trademark of Bell Laboratories.

execute the file to the user himself, to members of the user's group, and to all other users. Each process generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file but ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a "super user" who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Traditionally, UNIX software has been exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. In the current version, this policy may be easily adjusted to suit the needs of the installation or the individual user. Associated with each process and its descendants is a mask, which is in effect *and*-ed with the mode of every file and directory created by that process. In this way, users can arrange that, by default, all their files are no more accessible than they wish. The standard mask, set by *login,* allows all permissions to the user himself and to his group, but disallows writing by others.

To maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's files inaccessible to others. The lack of sufficiency could follow from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below). For greater protection, an encryption scheme is available. Since the editor is able to create encrypted documents, and the *crypt* command can be used to pipe such documents into the other text-processing programs, the length of time during which cleartext versions need be available is strictly limited. The encryption scheme used is not one of the strongest known,

but it is judged adequate, in the sense that cryptanalysis is likely to require considerably more effort than more direct methods of reading the encrypted files. For example, a user who stores data that he regards as truly secret should be aware that he is implicitly trusting the system administrator not to install a version of the crypt command that stores every typed password in a file.

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. In the current version, it is based on a slightly defective version of the Federal DES; it is purposely defective so that easily-available hardware is useless for attempts at exhaustive key-search. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is still feasible up to a point. We have observed that users choose passwords that are easy to guess: they are short, or from a limited alphabet, or in a dictionary. Passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out "login:" on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user *(su)* command is writable, anyone can copy the shell onto it and get a password-free version of *su.* A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. To take an obsolete example, the previous version of the *mail* command was set-UID and owned by the super-user. This version sent mail to the recipient's own directory. The notion was that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble was that *mail* was rather dumb: anyone could mail someone else's private file to himself. Much more serious is the following scenario: make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ".mail" to the password file in some writable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp/.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and tapes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su;* other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.

Password Security: A Case History

Robert Morris

Ken Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

Password security on the UNIX† time-sharing system [1] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e. prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so called "super-user" password, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual unauthorized access and accidental disclosure.

† UNIX is a trademark of Bell Laboratories.

PROLOGUE

The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons.

The technique is excessively vulnerable to lapses in security. Temporary loss of protection can occur when the password file is being edited or otherwise modified. There is no way to prevent the making of copies by privileged users. Experience with several earlier remote-access systems showed that such lapses occur with frightening frequency. Perhaps the most memorable such occasion occurred in the early 60's when a system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

Once such a lapse in security has been discovered, everyone's password must be changed, usually simultaneously, at a considerable administrative cost. This is not a great matter, but far more serious is the high probability of such lapses going unnoticed by the system administrators.

Security against unauthorized disclosure of the passwords was, in the last analysis, impossible with this system because, for example, if the contents of the file system are put on to magnetic tape for backup, as they must be, then anyone who has physical access to the tape can read anything on it with no restriction.

Many programs must get information of various kinds about the users of the system, and these programs in general should have no special permission to read the password file. The information which should have been in the password file actually was distributed (or replicated) into a number of files, all of which had to be updated whenever a user was added to or dropped from the system.

THE FIRST SCHEME

The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted. Such a scheme was first described in [3, p.91ff.]. It also seemed advisable to devise a system in which neither the password file nor the password program itself needed to be protected against being read by anyone.

All that was needed to implement these ideas was to find a means of encryption that was very difficult to invert, even when the encryption program is available. Most of the standard encryption methods used (in the past) for encryption of messages are rather easy to invert. A convenient and rather good encryption program happened to exist on the system at the time; it simulated the M-209 cipher machine [4] used by the U.S. Army during World War II. It turned out that the M-209 program was usable, but with a given key, the ciphers produced by this program are trivial to invert. It is a much more difficult matter to find out the key given the cleartext input and the enciphered output of the program. Therefore, the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key. The encrypted result was entered into the password file.

ATTACKS ON THE FIRST APPROACH

Suppose that the bad guy has available the text of the password encryption program and the complete password file. Suppose also that he has substantial computing capacity at his disposal.

One obvious approach to penetrating the password mechanism is to attempt to find a general method of inverting the encryption algorithm. Very possibly this can be done, but few successful results have come to light, despite substantial efforts extending over a period of more than five years. The results have not proved to be very useful in penetrating systems.

Another approach to penetration is simply to keep trying potential passwords until one succeeds; this is a general cryptanalytic approach called *key search.* Human beings being what they are, there is a strong tendency for people to choose relatively short and simple passwords that they can remember. Given free choice, most people will choose their passwords from a restricted character set (e.g. all lower-case letters), and will often choose words or names. This human habit makes the key search job a great deal easier.

The critical factor involved in key search is the amount of time needed to encrypt a potential password and to check the result against an entry in the password file. The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed. It is takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations.

If we want to check all passwords of length n that consist entirely of lower-case letters, the number of such passwords is 26^n. If we suppose that the password consists of printable characters only, then the number of possible passwords is somewhat less than 95^n. (The standard system "character erase" and "line kill" characters are, for example, not prime candidates.) We can immediately estimate the running time of a program that will test every password of a given length with all of its characters chosen from some set of characters. The following table gives estimates of the running time required on a PDP-11/70 to test all possible character strings of length n chosen from various sets of characters: namely, all lower-case letters, all lower-case letters plus digits, all alphanumeric characters, all 95 printable ASCII characters, and finally all 128 ASCII characters.

n	26 lower-case letters	36 lower-case letters and digits	62 alphanumeric characters	95 printable characters	all 128 ASCII characters
1	30 msec.	40 msec.	80 msec.	120 msec.	160 msec.
2	800 msec.	2 sec.	5 sec.	11 sec.	20 sec.
3	22 sec.	58 sec.	5 min.	17 min.	43 min.
4	10 min.	35 min.	5 hrs.	28 hrs.	93 hrs.
5	4 hrs.	21 hrs.	318 hrs.		
6	107 hrs.				

One has to conclude that it is no great matter for someone with access to a PDP-11 to test all lower-case alphabetic strings up to length five and, given access to the machine for, say, several weekends, to test all such strings up to six characters in length. By using such a program against a collection of actual encrypted passwords, a substantial fraction of all the passwords will be found.

Another profitable approach for the bad guy is to use the word list from a dictionary or to use a list of names. For example, a large commercial dictionary contains typicallly about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some "good" things to try are:

- The dictionary with the words spelled backwards.
- A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.
- The above with initial upper-case letters.

- All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
- Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time;

15 were a single ASCII character;

72 were strings of two ASCII characters;

464 were strings of three ASCII characters;

477 were string of four alphamerics;

706 were five letters, all upper-case or all lower-case;

605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831, or 86% of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

AN ANECDOTE

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudo-random number generator with only 2^{15} starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only 2^{15} of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time.

IMPROVEMENTS TO THE FIRST APPROACH

1. Slower Encryption

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

2. Less Predictable Passwords

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters,

then the program asks him to enter a longer password. This further reduces the efficacy of key search.

These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

3. Salted Passwords

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 (2^{12}). The reason for this is that there are 4096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be unthinkable.

5. A Subtle Point

To login successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in a invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.

CONCLUSIONS

On the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

It is also worth some effort to conceal even the encrypted passwords. Some UNIX systems have instituted what is called an "external security code" that must be typed when dialing into the system, but before logging in. If this code is changed periodically, then someone with an old password will likely be prevented from using it.

Whenever any security procedure is instituted that attempts to deny access to unauthorized persons, it is wise to keep a record of both successful and unsuccessful attempts to get at the secured resource. Just as an out-of-hours visitor to a computer center normally must not only identify himself, but a record is usually also kept of his entry. Just so, it is a wise precaution to make and keep a record of all attempts to log into a remote-access time-sharing system, and certainly all unsuccessful attempts.

Bad guys fall on a spectrum whose one end is someone with ordinary access to a system and whose goal is to find out a particular password (usually that of the super-user) and, at the other end, someone who wishes to collect as much password information as possible from as many systems as possible. Most of the work reported here serves to frustrate the latter type; our experience indicates that the former type of bad guy never was very successful.

We recognize that a time-sharing system must operate in a hostile environment. We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

References

[1] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* **17** (July 1974), pp. 365-375.

[2] *Proposed Federal Information Processing Data Encryption Standard.* Federal Register (40FR12134), March 17, 1975

[3] Wilkes, M. V. *Time-Sharing Computer Systems.* American Elsevier, New York, (1968).

[4] U. S. Patent Number 2,089,603.

PART 5: SUPPORTING DOCUMENTS

Of the seven articles in this part, only the first is current and directly applicable to the ULTRIX-32 system. "Changes to the Kernel in 4.2BSD," by Leffler, describes the differences between the 4.1BSD and 4.2BSD software distributions. The ULTRIX-32 system is based on 4.2BSD. Be sure to use this article if you are converting an existing UNIX 4.1BSD system to ULTRIX-32. The information provided covers:

- Revising local software so that it is compatible with 4.2BSD
- New directory organization
  - Machine-independent code
  - Machine-dependent code
- New file names
- Changes to old files
- Changes to stand-alone and bootstrap utilities

The six remaining articles in this part are included for historical reasons. They provide background information, some of which you may find useful in installing or tuning your ULTRIX-32 system. However, much of the information in these articles is obsolete. Note in particular that the articles on *uucp* do not refer to the current implementation. Check the *ULTRIX-32 Installation Manual* and the *ULTRIX-32 System Management Guide* for installation and maintenance procedures.

Changes to the Kernel in 4.2BSD

July 25, 1983

Samuel J. Leffler

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

This document summarizes the changes to the kernel between the September 1981 4.1BSD release and the July 1983 4.2BSD distribution. The information is presented in both overall terms (e.g. organizational changes), and as specific comments about individual files. See the source code itself for more details.

The system has undergone too many changes to detail everything. Instead the major areas of change will pointed out, followed by a brief description of the contents of files present in the 4.1BSD release. Where important changes and/or bug fixes were applied they are described. The networking support is not discussed in this document, refer to "4.2BSD Networking Implementation Notes" for a discussion of the internal structure of the network facilities.

Major changes include:

- organizational changes to isolate VAX specific portions of the system
- changes to support the new file system organization
- changes to support the new interprocess communication facilities
- changes for the new networking support; in particular, the DARPA standard Internet protocols TCP, UDP, IP, and ICMP, and the *network interface drivers* which provide hardware support
- changes for the new signal facilities
- changes for the new time and interval timer facilities
- changes to eliminate references to global variables; in particular, the global variables *u.u base*, *u.u offset*, *u.u segflg*, and *u.u count* have been almost completely replaced by *uio* structures which are passed by reference; the *u.u error* variable has not been completely purged from low level portions of the system, but is in many places now returned as a function value; the *uio* changes were necessitated by the new scatter-gather i/o facilities
- changes for the new disk quota facilities
- changes for more flexible configuration of the disk space used for paging and swapping

1. Carrying over local software

With the massive changes made to the system, both in organization and in content, it may take some time to understand how to carry over local software. The majority of this document is devoted to describing the contents of each important source file in the system. If you have local software other than device drivers to incorporate in the system you should first read this document completely, then study the source code to more fully understand the changes as they affect you.

Locally written device drivers will need to be converted to work in the new system. The changes required of device drivers are:

1) The calling convention for the driver *ioctl* routine has changed. Any data copied in or out of the system is now done at the highest level inside *ioctl*(). The third parameter to the driver *ioctl* routine is a data buffer passed by reference. Values to be returned by a driver must be copied into the associated buffer from which the system then copies them into the user address space.
2) The *read*, *write*, and *ioctl* entry points in device drivers must return 0 or an error code from *<errno.h>*.
3) The *read* and *write* entry points should no longer reference global variables out of the user area. A new *uio* parameter is passed to these routines which should, in turn, be passed to the *physio*() routine if the driver supports raw i/o.
4) Disk drivers which are to support swapping/paging must have a new routine which returns the size, in sectors, of a disk partition. This value is used in calculating the size of swapping/paging areas at boot time.
5) Code which previously used the *iomove*, *passc*, or *cpass* routines will have to be modified to use the new *uiomove*, *ureadc*, and *uwritec* routines. The new routines all use a *uio* structure to communicate the i/o base, offset, count, and segflag values previously passed globally in the user area.
6) Include files have been rearranged and new ones have been created. Common machine-dependent files such as *mtpr.h*, *pte.h*, *reg.h*, and *psl.h* are no longer in the "h" directory; see below under organizational changes.
7) The handling of UNIBUS resets has changed. The reset routine should no longer deallocate UNIBUS resources allocated to pending i/o requests (this is done in the *ubareset* routine). For most drivers this means the reset routine simply needs to invalidate any *ub info* values stored in local data structures to insure new UNIBUS resources will be allocated the next time the "device start" routine is entered.

2. Organizational changes

The directory organization and file names are very different from 4.1BSD. The new directory layout breaks machine-specific and network-specific portions of the system out into separate directories. A new file, *machine* is a symbolic link to a directory for the target machine, e.g. *vax*. This allows a single set of sources to be shared between multiple machine types (by including header files as "../machine/file"). The directory naming conventions, as they relate to the network support, are intended to allow expansion in supporting multiple "protocol families". The following directories comprise the system sources for the VAX:

/sys/h	machine independent include files
/sys/sys	machine independent system source files
/sys/conf	site configuration files and basic templates
/sys/net	network independent, but network related code
/sys/netinet	DARPA Internet code
/sys/netimp	IMP support code
/sys/netpup	PUP-1 support code
/sys/vax	VAX specific mainline code
/sys/vaxif	VAX network interface code
/sys/vaxmba	VAX MASSBUS device drivers and related code
/sys/vaxuba	VAX UNIBUS device drivers and related code

Files indicated as *machine independent* are shared among 4.2BSD systems running on the VAX and Motorola 68010. Files indicated as *machine dependent* are located in directories indicative of the machine on which they are used; the 4.2BSD release from Berkeley

contains support only for the VAX. Files marked *network independent* form the "core" of the networking subsystem, and are shared among all network software; the 4.2BSD release from Berkeley contains complete support only for the DARPA Internet protocols IP, TCP, UDP, and ICMP.

3. Bug fixes and changes

This section contains a brief description of each file which is not part of the network subsystem, and also indicates important changes and bug fixes applied to the source code contained in the file.

3.1. /sys/h

Files residing here are intended to be machine independent. Consequently, the header files for device drivers which were present in this directory in 4.1BSD have been moved to other directories; e.g. /sys/vaxuba. Many files which had been duplicated in /usr/include are now present only in /sys/h. Further, the 4.1BSD /usr/include/sys directory is now normally a symbolic link to this directory. By having only a single copy of these files the "multiple update" problem no longer occurs. (It is still possible to have /usr/include/sys be a copy of the /sys/h for sites where it is not feasible to allow the general user community access to the system source code.)

The following files are new to /sys/h in 4.2BSD:

domain.h describes the internal structure of a communications domain; part of the new ipc facilities

errno.h had previously been only in /usr/include; the file /usr/include/errno.h is now a symbolic link to this file

fs.h replaces the old filsys.h description of the file system organization

gprof.h describes various data structures used in profiling the kernel; see *gprof*(1) for details

kernel.h is an offshoot of systm.h and param.h; contains constants and definitions related to the logical UNIX "kernel"

mbuf.h describes the memory managment support used mostly by the network; see "4.2BSD Networking Implementation Notes" for more information

mman.h contains definitions for planned changes to the memory management facilities (not implemented in 4.2BSD)

nami.h defines various structures and manifest constants used in conjunctions with the *namei* routine (part of this file reflects future plans for changes to *namei* rather than current use)

protosw.h contains a description of the protocol switch table and related manifest constants and data structures use in communicating with routines located in the table

quota.h contains definitions related to the new disk quota facilities

resource.h contains definitions used in the *getrusage*, *getrlimit*, and *getpriority* system calls (among others)

socket.h contains user-visible definitions related to the new socket ipc facilities

socketvar.h contains implementation definitions for the socket ipc facilities

ttychars.h contains definitions related to tty character handling; in particular, manifest constants for the system standard erase, kill, interrupt, quit, etc. characters are stored here (all the appropriate user programs use these manifest definitions)

ttydev.h contains definitions related to hardware specific portions of tty handling (such as baud rates); to be expanded in the future

uio.h contains definitions for users wishing to use the new scatter-gather i/o facilities; also contains the kernel *uio* structure used in implementing scatter-gather i/o

un.h contains user-visible definitions related to the "unix" ipc domain

unpcb.h contains the definition of the protocol control block used in the "unix" ipc domain

wait.h contains definitions used in the *wait* and *wait3*(2) system calls; previously in /usr/include/wait.h

The following files have undergone significant change:

buf.h reflects the changes made to the buffer cache for the new file system organization — buffers are variable sized with pages allocated to buffers on demand from a pool of pages dedicated to the buffer cache; one new structure member has been added to eliminate overloading of a commonly unreferenced structure member; a new flag B CALL, when set, causes the function *b iodone* to be called when i/o completes on a buffer (this is used to wakeup the pageout daemon); macros have been added for manipulating the buffer queues, these replace the previous subroutines used to insert and delete buffers from the queues

conf.h reflects changes made in the handling of swap space and changes made for the new *select*(2) system call; the block device table has a new member, *d psize*, which returns the size of a disk partition, in sectors, given a major/minor value; the character device table has a new member, *d select*, which is passed a *dev t* value and an FREAD (FWRITE) flag and returns 1 when data may be read (written), and 0 otherwise; the *swdevt* structure now includes the size, in sectors, of a swap partition

dir.h is completely different since directory entries are now variable length; definitions for the user level interface routines described in *directory*(3) are also present

file.h has a very different *file* structure definition and definitions for the new *open* and *flock* system calls; symbolic definitions for many constants commonly supplied to *access* and *lseek*, are also present

inode.h reflects the new hashed cacheing scheme as well additions made to the on-disk and in-core inodes; on-disk inodes now contain a count of the actual number of disk blocks allocated a file (used mostly by the disk quota facilities), larger time stamps (for planned changes), more direct block pointers, and room for future growth; in-core inodes have new fields for the advisory locking facilities, a back pointer to the file system super block information (to eliminate lookups), and a pointer to a structure used in implementing disk quotas.

ioctl.h has all request codes constructed from IO, IOR, IOW, and IOWR macros which encode whether the request requires data copied in, out, or in and out of the kernel address space; the size of the data parameter (in bytes) is also encoded in the request, allowing the *ioctl*() routine to perform all user-kernel address space copies

mount.h the *mount* structure has a new member used in the disk quota facilities

param.h has had numerous items deleted from it; in particular, many definitions logically part of the "kernel" have been moved to kernel.h, and machine-dependent values and definitions are now found in param.h files located in machine/param.h; contains a manifest constant, NGROUPS, which defines

the maximum size of the group access list

proc.h has changed extensively as a result of the new signals, the different resource usage structure, the disk quotas, and the new timers; in addition, new members are present to simplify searching the process tree for siblings; the SDLYU and SDETACH bits are gone, the former is replaced by a second parameter to *pagein*, the latter is no longer needed due to changes in the handling of open's on /dev/tty by processes which have had their controlling terminal revoked with *vhangup*

signal.h reflects the new signal facilities; several new signals have been added: SIGIO for signal driven i/o; SIGURG for notification when an urgent condition arises; and SIGPROF and SIGVTALRM for the new timer facilities; structures used in the *sigvec*(2) and *sigstack*(2) system calls, as well as signal handler invocations are defined here

stat.h has been updated to reflect the changes to the inode structure; in addition a new field *st blksize* contains an "optimal blocking factor" for performing i/o (for files this is the block size of the underlying file system)

systm.h has been trimmed back a bit as various items were moved to kernel.h

time.h contains the definitions for the new time and interval timer facilities; time zone definitions for the half dozen time zones understood by the system are also included here

tty.h reflects changes made to the internal structure of the terminal handler; the "local" structures have been merged into the standard flags and character definitions though the user interface is virtually identical to that of 4.1BSD; the TTYHOG value has been changed from 256 to 255 to account for a counting problem in the terminal handler on input buffer overflow

user.h has been extensively modified; members have been grouped and categorized to reflect the "4.2BSD System Manual" presentation; new members have been added and existing members changed to reflect: the new groups facilities, changes to resource accounting and limiting, new timer facilities, and new signal facilities

vmmac.h has had many macro definitions changed to eliminate assumptions about the hardware virtual memory support; in particular, the stack and user area page table maps are no longer assumed to be adjacent or mapped by a single page table base register

vmparam.h now includes machine-dependent definitions from a file machine/vmparam.h.

vmsystm.h has had several machine-dependent definitions moved to machine/vmparam.h

3.2. /sys/sys

This directory contains the "mainstream" kernel code. Files in this directory are intended to be shared between 4.2BSD implementations on all machines. As there is little correspondence between the current files in this directory and those which were present in 4.1BSD a general overview of each files's contents will be presented rather than a file-by-file comparison.

Files in the *sys* directory are named with prefixes which indicate their placement in the internal system layering. The following table summarizes these naming conventions.

init	system initialization
kern	kernel (authentication, process management, etc.)
quota	disk quotas
sys	system calls and similar
tty	terminal handling
ufs	file system
uipc	interprocess communication
vm	virtual memory

3.2.1. Initialization code

init main.c contains system startup code

init sysent.c contains the definition of the *sysent* table – the table of system calls supported by 4.2BSD

3.2.2. Kernel-level support

kern acct.c contains code used in per-process accounting

kern clock.c contains code for clock processing; work was done here to minimize time spent in the *hardclock* routine; support for kernel profiling and statistics collection from an alternate clock source have been added; a bug which caused the system to lose time has been fixed; the code which drained terminal multiplexor silos has been made the default mode of operation and moved to locore.s

kern descrip.c contains code for management of descriptors; descriptor related system calls such as *dup* and *close* (the upper-most levels) are present here

kern exec.c contains code for the *exec* system call

kern exit.c contains code for the *exit* system call

kern fork.c contains code for the *fork* (and *vfork*) system call

kern mman.c contains code for memory management related calls; the contents of this file is expected to change when the revamped memory management facilities are added to the system

kern proc.c contains code related to process management; in particular, support routines for process groups

kern prot.c contains code related to access control and protection; the notions of user ID, group ID, and the group access list are implemented here

kern resource.c code related to resource accounting and limits; the *getrusage* and "get" and "set" resource limit system calls are found here

kern sig.c the signal facilities; in particular, kernel level routines for posting and processing signals

kern subr.c support routines for manipulating the *uio* structure: *uiomove*, *ureadc*, and *uwritec*

kern synch.c code related to process synchonization and scheduling: *sleep* and *wakeup* among others

kern time.c code related to processing time; the handling of interval timers and time of day

kern xxx.c miscellaneous system facilities and code for supporting 4.1BSD compatibility mode (kernel level)

3.2.3. Disk quotas

quota kern.c "kernel" of disk quota suppport

quota subr.c miscellaneous support routines for disk quotas

quota sys.c disk quota system call routines

quota ufs.c portions of the disk quota facilities which interface to the file system routines

3.2.4. General subroutines

subr mcount.c code used when profiling the kernel

subr prf.c *printf* and friends; also, code related to handling of the diagnostic message buffer

subr rmap.c subroutines which manage resource maps

subr xxx.c miscellaneous routines and code for routines implemented with special VAX instructions, e.g. bcopy

3.2.5. System level support

sys generic.c code for the upper-most levels of the "generic" system calls: *read*, *write*, *ioctl*, and *select*; a "must read" file for the system guru trying to shake out 4.1BSD bad habits

sys inode.c code supporting the "generic" system calls of sys generic.c as they apply to inodes; the guts of the byte stream file i/o interface

sys process.c code related to process debugging: *ptrace* and its support routine *procxmt*; this file is expected to change as better process debugging facilities are developed

sys socket.c code supporting the "generic" system calls of sys generic.c as they apply to sockets

3.2.6. Terminal handling

tty.c the terminal handler proper; both 4.1BSD and version 7 terminal interfaces have been merged into a single set of routines which are selected as line disciplines; a bug which caused new line delays past column 127 to be calculated incorrectly has been fixed; the high water marks for terminals running in tandem mode at 19.2 or 38.4 kilobaud have been upped

tty bk.c the old Berknet line discipline (defunct)

tty conf.c initialized data structures related to terminal handling;

tty pty.c support for pseudo-terminals; actually two device drivers in one; additions over 4.1BSD pseudo-terminals include a simple "packet protocol" used to support flow-control and output flushing on interrupt, as well as a "transparent" mode used in programs such as emacs

tty subr.c c-list support routines

tty tb.c two line disciplines for supporting RS232 interfaces to Genisco and Hitachi tablets

tty tty.c trivial support routines for "/dev/tty"

3.2.7. File system support

ufs alloc.c code which handles allocation and deallocation of file system related resources: disk blocks, on-disk inodes, etc.

ufs bio.c block i/o support; the buffer cache proper; see description of buf.h and "A Fast File System for UNIX" for information

ufs bmap.c code which handles logical file system to logical disk block number mapping; understands structure of indirect blocks and files with holes; handles automatic extension of files on write

ufs dsort.c sort routine implementing prioritized seek sort algorithm for disk i/o operations

ufs fio.c code handling file system specific issues of access control and protection

ufs inode.c inode management routines; in-core inodes are now hashed and cached; inode synchronization has been revamped since 4.1BSD to eliminate race conditions present in 4.1

ufs mount.c code related to demountable file systems

ufs nami.c the *namei* routine (and related support routines) — the routine that maps pathnames to inode numbers

ufs subr.c miscellaneous subroutines: this code is shared with certain user programs such as *fsck* (8); for a good time look at the *bufstats* routine in this file

ufs syscalls.c file system related system calls, everything from *open* to *unlink*; many new system calls are found here: *rename*, *mkdir*, *rmdir*, *truncate*, etc.

ufs tables.c static tables used in block and fragment accounting; this file is shared with user programs such as *fsck* (8)

ufs xxx.c miscellaneous routines and 4.1BSD compatibility code; all of the code which still understands the old inode format is in here

3.2.8. Interprocess communication

uipc domain.c code implementing the "communication domain" concept; this file must be augmented to incorporate new domains

uipc mbuf.c memory management routines for the ipc and network facilities; refer to the document "4.2BSD Networking Implementation Notes" for a detailed description of the routines in this file

uipc pipe.c leftover code for connecting two sockets into a pipe; actually a special case of the code for the *socketpair* system call

uipc proto.c UNIX ipc communication domain configuration definitions; contains UNIX domain data structure initialization

uipc socket.c top level socket support routines; these routines handle the interface to the protocol request routines, move data between user address space and socket data queues, understand the majority of the logic in process synchronization as it relates to the ipc facilities

uipc socket2.c lower level socket support routines; provide nitty gritty bit twiddling of socket data structures; manage placement of data on socket data queues

uipc syscalls.c user interface code to ipc system calls: *socket*, *bind*, *connect*, *accept*, etc.; concerned exclusively with system call argument passing and validation

uipc usrreq.c UNIX ipc domain support; user request routine and supporting utility routines

3.2.9. Virtual memory support

The code in the virtual memory subsystem has changed very little from 4.1BSD; changes made in these files were either to gain portability, handle the new swap space configuration scheme, or fix bugs.

vm drum.c code for the management of disk space used in paging and swapping

vm mem.c management of physical memory; the "core map" is implemented here as well as the routines which lock down pages for physical i/o (the latter will have to change when the memory management facilities are modified to support sharing of pages); a sign extension bug on block numbers extracted from the core map has been fixed (this caused the system to crash with certain disk partition layouts on RA81 disks)

vm mon.c support for virtual memory monitoring; code in this file is included in the system only if the PGINPROF and/or TRACE options are configured

vm page.c the code which handles and processes page faults: *pagein*; race conditions in accessing pages in transit and requests to lock pages for raw i/o have been fixed in this code; a major path through *pagein* whose sole purpose was to implement the software simulated reference bit has been "parallel coded" in assembly language (this appears to decrease system time by at least 5% when a system is paging heavily); *pagein* now has a second parameter indicating if the page to be faulted in should be left locked (this eliminated the need for the SDLYU flag in the *proc* structure)

vm proc.c mainly code to manage virtual memory allocation during process creation and destruction (the virtual memory equivalent of "passing the buck" is done here).

vm pt.c code for manipulating process page tables; knowledge of the user area is found here as it relates to the user address space page tables

vm sched.c the code for process 0, the scheduler, lives here; other routines which monitor and meter virtual memory activity (used in implementing high level scheduling policies) also are present; this code has been better parameterized to isolate machine-dependent heuristics used in the scheduling policies

vm subr.c miscellaneous routines: some for manipulating accessability of virtual memory, others for mapping virtual addresses to logical segments (text, data, stack)

vm sw.c indirect driver for interleaved, multi-controller, paging area; modified to support interleaved partitions of different sizes

vm swap.c code to handle process related issues of swapping

vm swp.c code to handle swap i/o

vm text.c code to handle shared text segments – the "text" table

3.3. /sys/conf

This directory contains files used in configuring systems. The format of configuration files has changed slightly; it is described completely in a new document "Building 4.2BSD UNIX Systems with Config". Several new files exist for use by the *config*(8) program, and several old files have had their meaning changed slightly.

LINT a new configuration file for use in linting kernels

devices.vax maps block device names to major device numbers (on the VAX)

files now has only files containing machine-independent code

files.*xxx* (where *xxx* is a system name) optional, *xxx*-specific *files* files

files.vax new file describing files which contain machine-dependent code

makefile.vax makefile template specific to the VAX

param.c updated calculations of *ntext* and *nfile* to reflect network requirements; new quantities added for disk quotas

3.3.1. /sys/vaxuba

This directory contains UNIBUS device drivers and their related include files. The latter have moved from /sys/h in an effort to isolate machine-dependent portions of the system. The following device drivers were not present in the 4.1BSD release.

ad.c a driver for the Data Translation A/D converter

ik.c an Ikonas frame buffer graphics interphase; user access to the device is implemented by mapping the device registers directly into the virtual address space of a user (the routines to map memory are included in uba.c only if an Ikonas is configured in the system)

kgclock.c a driver for a DL11-W or KL11-W used as an auxiliary real-time clock source for kernel profiling and/or statistics gathering; if this device is present, the system will automatically collect its i/o statistics (and if profiling, pc samples) off the secondary clock; very useful in kernel profiling as the second clock source eliminates most of the statistical anomalies and shows the true time spent in the clock routine

ps.c driver for an Evans and Sutherland Picture System 2

rl.c driver for RL11 controller with RL02 cartridge disks; does not support RL01 disks though it should only require additions to disk geometry and partition tables

rx.c driver for RX211 floppy disk controller; provides both block and character device interfaces; *ioctl* calls support floppy disk formatting and "deleted data mark" sensing and writing; makes a great paging device

ut.c driver for tape controllers which emulate a TU45 on the UNIBUS; in particular, the System Industries Model 9700 triple density tape drive

uu.c driver for dual UNIBUS TU58 cartridge tape cassettes accessed through a DL11 serial line; uses assembly language code in locore.s which provides pseudo-DMA on input (necessary to avoid data overruns); using this driver while the system runs multi-user degrades response severely (developed at Berkeley exclusively to produce distribution TU58 cassettes)

In addition to the above device drivers, many drivers present in 4.1BSD now sport corresponding include files which contain device register definitions. For example, the DH11 driver is now broken into three files: dh.c, dhreg.h, and dmreg.h.

The following drivers have been significantly modified, or had bugs fixed in them, since the 4.1BSD release:

dh.c changes to reflect the revised tty data organization

dmf.c a bug where device register accesses caused unwitting modification of certain status bits has been fixed; modem control has been fixed; a remnant of the DH11 include file which caused incorrect definitions for even/odd parity has been fixed; changes to reflect the revised tty data organization

dz.c now supports the DZ32; changes to reflect the revised tty data organization

lp.c now takes a non-zero flags value specified in the configuration file as the printer width (default is 132 columns); thus, to configure an 80 column printer, include "flags 80" in the device specification

rk.c a race condition has been fixed where a seek finishing on one drive appeared as an i/o transfer completeing on another (this bug actually was present in all UNIBUS disk drivers); changes for *uio* and swap space configuration

tm.c a typo which made the system crash with multiple slaves on a single controller has been fixed; an incorrect priority level change in the watchdog timer routine which caused the system to crash when a device operation timed out has been fixed; changes for *uio* processing of raw i/o

ts.c changes for *uio* processing of raw i/o

uba.c a new support routine for allocating UNIBUS memory for memory-mapped devices such as the 3Com Ethernet interface; the handling of UNIBUS resets has been changed, all UNIBUS resources are now reclaimed in the *ubareset* routine prior to calling individual device driver reset routines – this implies driver reset routines should no longer free up allocated UNIBUS resources; new routines for mapping UNIBUS memory into the virtual address space of a process have been added to support the Ikonas device driver; changes to fix the race condition described above in the RK07 device driver; processes awaiting UNIBUS map registers now sleep on a different event than those waiting for buffered data paths

uda.c the problem with multiplexing buffered data paths on an 11/750 has been fixed; a bug in the setup of the *ui dk* field has been fixed; now properly defines the field indicating the disk transfer rate; changes for *uio* processing and swap space configuration

up.c now supports ECC correction and bad sector forwarding; significant changes have been made to make configuration of various disk drives simple (by probing the holding register and using the resultant value indicating the number of tracks on the disk); the race condition described under rk.c has been fixed; references to UNIBUS map registers are now done with longword instructions so the device driver does not cause the system to crash when an ECC or bad sector error occurs on a disk attached to a 730 UNIBUS; the upSDIST/upRDIST parameters which control the use of search and seek operations on controllers with multiple drives have been made drive dependent; a bug whereby the probe routine would belive certain non-existant drives were present has been fixed; changes for *uio* processing and swap space configuration

va.c has been rewritten to honor the software support for exclusive access to the UNIBUS so that the device may coexist on the same UNIBUS with RK07 disk drives; the driver now works with controllers which have a GO bit

3.3.2. /sys/vax

The following files are new in 4.2BSD:

crt0.ex edit script for creating a profiled kernel

frame.h copied from /usr/include

in cksum.c checksum routine for the DARPA Internet protocols

param.h machine-dependent portion of /sys/h/param.h

pup cksum.c checksum routine for PUP-I protocols

rsp.h protocol definitions for communicating with a TU58

sys machdep.c machine-dependent portion of the "sys *" files of /sys/sys

ufs machdep.c machine-dependent portion of the "ufs *" files of /sys/sys

vm machdep.c machine-dependent portion of the "vm *" files of /sys/sys

vmparam.h machine-dependent portion of /sys/h/vmparam.h

The following files have been modified for 4.2BSD:

Locore.c includes new definitions for linting the network and ipc code

asm.sed now massages *insque*, *remque*, and various routines which do byte swapping into assembly language

autoconf.c handles MASSBUS drives which come on-line after the initial autoconfiguration process; sizes and configures swap space at boot time in addition to calculating the swap area allocation parameters *dmtext*, *dmmax*, and *dmmin* (which were manifest constants in 4.1BSD); calculates the disk

partition offset for system dumps at boot time to take into account variable sized swap areas; now uses the per-driver array of standard control status register addresses when probing for devices on the UNIBUS; now allows MASSBUS tapes and disks to be wildcarded across controllers

conf.c uses many "local" spaces for new and uncommon device drivers

genassym.c generates several new definitions for use in locore.s

locore.s includes code to vector software interrupts to protocol processing modules; assembly language assist routines for the console and UNIBUS TU58 cassette drives; a new routine, *Fastreclaim* is a fast coding of a major path through the *pagein* routine; copyin and copyout now handle greater than 64Kbyte data copies and return EFAULT on failure; understands the new signal trampoline code; now contains code for draining terminal multiplexor silos at clock time; a bug where a the translation buffer was sometimes being improperly flushed during a *resume* operation has been fixed

machdep.c a bug which caused memory errors to not be reported on 11/750's has been fixed; has new code for handling the new signals; recovers from translation buffer parity fault machine checks apparently caused by substandard memory chips used in many 11/750's; includes optional code to pinpoint bad memory chips on Trendata memory boards; the machine check routine now calls the *memerr* routine to print out the memory controller status registers in case the fault occurred because of a memory error

mem.c now has correct definitions to enable correctable memory error reporting on 11/750's: DEC documentation incorrectly specifies use of the ICRD bit

pcb.h has changes related to the new signal trampoline code

swapgeneric.c supports more devices which can be used as a generic root device; interacts with the new swap configuration code to size the swap area properly when running a generic system; understands the special "swap on root" device syntax used when installing the system

trap.c can be compiled with a SYSCALLTRACE define to allow system calls to be traced when the variable *syscalltrace* is non-zero;

tu.c includes (limited) support for the TU58 console cassette on the 11/750, sufficient for use in single-user mode; supports the use of the MRSP ROM on the 11/750.

3.3.3. /sys/vaxmba

The following bug fixes and modifications have been applied to the MASSBUS device drivers:

hp.c a large number of disk drives attached to second vendor disk controllers are now automatically recognized at boot time by probing the holding register and using disk geometry information to decide what kind of drive is present; the hpSDIST/hpRDIST parameters that control seek and search operations on controllers with multiple drives have been made a per-drive parameter; a bug where the sector number reported on a hard error was off by one has been fixed; the error recovery code now searches the bad sector table when a header CRC error occurs; the error recovery code now handles bad sectors on tracks which also have skip sectors; a bug in the handling of ECC errors has been fixed; many separate driver data structures have been consolidated into the software carrier structure; the driver handles the ML-11 solid-state disk

mba.c now autoconfigures MASSBUS tapes and disks which "come on-line" after the initial boot

4. Standalone support

This section describes changes made to the standalone i/o facilities and the new methods used in system bootstrapping.

4.1. Disk formatting

A new disk formatting program has been developed for use with non-DEC UNIBUS and MASSBUS disk controllers. The *format* (8V) program has been tested mainly with disk drives attached to Emulex MASSBUS and UNIBUS disk controllers, but should operate with any controller which handles bad sector forwarding in an identical fashion to DEC RM03/RM05 or RM80 (but not RP06) disk controllers. The program runs standalone formatting disk headers and creating a bad sector table in the DEC standard 144 format.

4.2. Standalone i/o library

Changes to support more complex standalone i/o applications as well as changes for the new file system organization, have resulted in significant revisions to the standalone i/o library. Device drivers now support a new entry point for *ioctl* requests and library routines now return error codes a la the UNIX system calls. In addition, standalone i/o library routines now make many more internal consistency checks to verify data structures have not been corrupted by faulty device drivers and that i/o errors have not occurred when reading critical file system information. In conjunction with the new disk formatter, the *up* and *hp* standalone drivers have been rewritten to support ECC correction and bad sector handling. These drivers are used in bootstrapping from the console media on 11/780's and 11/730's thereby eliminating the requirement for error free root partitions on disks attached to *hp* and *up* controllers. Many bugs in the standalone tape drivers have been fixed.

4.3. System bootstrapping

On 11/780's and 11/730's, the console device is still used to load the "boot" program. This in turn loads the system image from the root file system.

The method by which the system bootstraps on 11/750's is different in 4.2BSD. The system is still bootstrapped from disk using a boot block in sector 0 of the root file system partition, but now this boot block simply reads in the next 7.5 kilobytes. The 7.5 kilobyte program is a version of the "/boot" program loaded only with the device driver required to read the "/boot" program from the root file system. The "/boot" program then reads in the system image, as done on 11/780's and 11/730's.

The additional level of bootstrap code was done to simplify the sector 0 boot programs and minimize the total amount of assembly language code which had to be maintained. It was also expected that 7.5 kilobytes would be sufficient to allow the new *hp* and *up* standalone drivers which support ECC correction and bad sector handling to be used. Unfortunately, the standalone system has not yet been trimmed down to allow the second level boot programs, loaded with the new drivers, to fit in the space provided. Sites which have Winchester disk drives with bad sectors in the root file system partition and which require this support should be able to trim the size of the second level boot program to make it fit.

Installing and Operating 4.2BSD on the VAX

July 21, 1983

Samuel J. Leffler

William N. Joy

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

1. INTRODUCTION

This document explains how to install the 4.2BSD release of the Berkeley version of UNIX for the VAX on your system. Due to the new file system organization used in 4.2BSD, no matter what version of UNIX you may currently be running you will have to perform a full bootstrap from the distribution tape; the techniques for converting "old" systems are discussed in a chapter 3 of this document.

1.1. Hardware supported

This distribution can be booted on a VAX 11/780, VAX 11/750, or VAX 11/730 cpu with any of the following disks:

DEC MASSBUS:	RM03, RM05, RM80, RP06, RP07
EMULEX MASSBUS:	AMPEX 300M, 330M, CDC 300M, FUJITSU 404M
DEC UNIBUS:	RK07, RA80, RA81, RA60
EMULEX SC-21V UNIBUS*:	AMPEX 300M, 330M, CDC 300M, FUJITSU 160M, 404M
DEC IDC:	R80, RL02

The tape drives supported by this distribution are:

DEC MASSBUS:	TE16, TU45, TU77, TU78
DEC UNIBUS:	TS11, TU80
EMULEX TC-11 UNIBUS:	KENNEDY 9300, CIPHER
TU45 UNIBUS*:	SI 9700

The tapes and disks may be on any available UNIBUS or MASSBUS adapter at any slot with the proviso that the tape device must be slave number 0 on the formatter if it is a MASSBUS tape drive.

DEC, VAX, IDC, UNIBUS and MASSBUS are trademarks of Digital Equipment Corporation.
UNIX is a Trademark of Bell Laboratories.
* Other UNIBUS controllers and drives may be easily usable with the system, but will likely require minor modifications to the system to allow bootstrapping. The EMULEX disk and SI tape controllers, and the drives shown here are known to work as bootstrap devices.

1.2. Distribution format

The basic distribution contains the following items:

(2) 1600bpi 2400' magnetic tapes,
(1) TU58 console cassette, and
(1) RX01 console floppy disk.

Installation on any machine requires a tape unit. Since certain standard VAX packages do not include a tape drive, this means one must either borrow one from another VAX system or one must be purchased separately. The console media distributed with the system are not suitable for use as the standard console media; their intended use is only for installation.

The distribution does not fit on several standard VAX configurations which contain only small disks. If your hardware configuration does not provide at least 75 Megabytes of disk space you can still install the distribution, but you will probably have to operate without source for the user level commands and, possibly, the source for the operating system. The previous RK07-only distribution format provided by our group is no longer available. Further, no attempt has ever been made to install the system on the standard VAX-11/730 hardware configuration from DEC which contains only dual RL02 disk drives (though the distribution tape may be bootstrapped on an RL11 controller and the system provides support for RL02 disk drives either on an IDC or an RL11). The labels on the two distribution tapes indicate the amount of disk space each tape file occupies, these should be used in selecting file system layouts on systems with little disk space.

If you have the facilities, it is a good idea immediately to copy the magnetic tapes in the distribution kit to guard against disaster. The tapes are 9-track 1600 BPI and contain some 512-byte records followed by many 10240-byte records. There are interspersed tape marks; end-of-tape is signaled by a double end-of-file.

The basic bootstrap material is present in three short files at the beginning of the bootstrap tape. The first file on the tape contains preliminary bootstrapping programs. This is followed by a binary image of a 400 kilobyte "mini root" file system. Following the mini root file is a full dump of the root file system (see *dump*(8)**). Additional files on the first and second tapes contain tape archive images (see *tar*(1)): the fourth file on the first tape contains source for the system (/sys); the fifth file on the first tape contains most of the files in the file system /usr, except the source (/usr/src) which is in the first file on the second tape. The second file on the second tape contains software contributed by the user community, refer to the accompanying documentation for a description of its contents and an explanation of how it should be installed.

1.3. VAX hardware terminology

This section gives a short discussion of VAX hardware terminology to help you get your bearings.

If you have MASSBUS disks and tapes it is necessary to know the MASSBUS they are attached to, at least for the purposes of bootstrapping and system description. The MASSBUSes can have up to 8 devices attached to them. A disk counts as a device. A tape *formatter* counts as a device, and several tape drives may be attached to a formatter. If you have a separate MASSBUS adapter for a disk and one for a tape then it is conventional to put the disk as unit 0 on the MASSBUS with the lowest "TR" number, and the tape formatter as unit 0 on the next MASSBUS. On a 11/780 this would correspond to having the disk on "mba0" at "tr8" and the tape on "mba1" at "tr9". Here the MASSBUS adapter with the lowest TR number has been called "mba0" and the one with the next lowest number is called "mba1".

** References of the form X(Y) mean the subsection named X in section Y of the UNIX programmer's manual.

To find out the MASSBUS your tape and disk are on you can examine the cabling and the unit numbers or your site maintenance guide. Do not be fooled into thinking that the number on the front of the tape drive is a device number; it is a *slave* number, one of several possible tapes on the single tape formatter. For bootstrapping the slave number **must** be 0. The formatter unit number may be anything distinct from the other numbers on the same MASSBUS, but you must know what it is.

The MASSBUS devices are known by several different names by DEC software and by UNIX. At various times it is necessary to know both names. There is, of course, the name of the device like "RM03" or "RM80"; these are easy to remember because they are printed on the front of the device. DEC also gives devices names by the names of the driver in the system using a naming convention that reflects the interconnect topology of the machine. The first letter of such a name is a "D" for a disk, the second letter depends on the type of the drive, "DR" for RM03, RM05, and RM80's, "DB" for RP06's. The next letter is related to the interconnect; DEC calls the first MASSBUS adapter "A", the second "B", etc. Thus "DRA" is a RM drive on the first MASSBUS adapter. Finally, the name ends in a digit corresponding to the unit number for the device on the MASSBUS, i.e. "DRA0" is a disk at the first device slot on the first MASSBUS adapter and is a RM disk.

1.4. UNIX device naming

UNIX has a set of names for devices, which are different from the DEC names for the devices, viz.:

RM/RP disks	hp
TE/TU tapes	ht
TU78 tape	mt

The normal standalone system, used to bootstrap the full UNIX system, uses device names:

xx(y,z)

where *xx* is either **hp**, **ht**, or **mt**. The value *y* specifies the MASSBUS to use and also the device. It is computed as

8 * mba + device

Thus mba0 device 0 would have a *y* value of 0 while mba1 device 0 would have a *y* value of 8. The *z* value is interpreted differently for tapes and disks: for disks it is a disk *partition* (in the range 0-7), and for tapes it is a file number on the tape.

Each UNIX physical disk is divided into 8 logical disk partitions, each of which may occupy any consecutive cylinder range on the physical device. The cylinders occupied by the 8 partitions for each drive type are specified in section 4 of the programmers manual and in the disk description file /etc/disktab (c.f. *disktab*(5)).* Each partition may be used for either a raw data area such as a paging area or to store a UNIX file system. It is conventional for the first partition on a disk to be used to store a root file system, from which UNIX may be bootstrapped. The second partition is traditionally used as a paging area, and the rest of the disk is divided into spaces for additional "mounted file systems" by use of one or more additional partitions.

The third logical partition of each physical disk also has a conventional usage: it allows access to the entire physical device, including the bad sector forwarding information recorded at the end of the disk (one track plus 126 sectors). It is occasionally used to store a single

* It is possible to change the partitions by changing the code for the table in the disk driver; since it is often desirable to do this it is clear that these tables should be read off each pack; they may be in a future version of the system.

large file system or to access the entire pack when making a copy of it on another. Care must be taken when using this partition to not overwrite the last few tracks and thereby clobber the bad sector information.

The disk partitions have names in the standalone system of the form "hp(x,y)" with varying *y* as described above. Thus partition 1 of a RM05 on mba0 at drive 0 would be "hp(0,1)". When not running standalone, this partition would normally be available as "/dev/hp0b". Here the prefix "/dev" is the name of the directory where all "special files" normally live, the "hp" serves an obvious purpose, the "0" identifies this as a partition of hp drive number "0" and the "b" identifies this as the first partition (where we number from 0, the 0'th partition being "hp0a".)

In all simple cases, a drive with unit number 0 (in its unit plug on the front of the drive) will be called unit 0 in its UNIX file name. This is not, however, strictly necessary, since the system has a level of indirection in this naming. This can be taken advantage of to make the system less dependent on the interconnect topology, and to make reconfiguration after hardware failure extremely easy. We will not discuss that now.

Returning to the discussion of the standalone system, we recall that tapes also took two integer parameters. In the normal case where the tape formatter is unit 0 on the second mba (mba1), the files on the tape have names "ht(8,0)", "ht(8,1)", etc. Here "file" means a tape file containing a single data stream. The distribution tapes have data structures in the tape files and though the tapes contain only 6 tape files, they contain several thousand UNIX files.

For the UNIBUS, there are also conventional names. The important DEC names to know are DM?? for RK07 drives and DU?? for drives on a UDA50. For example, RK07 drive 0 on a controller on the first UNIBUS on the machine is "DMA0". UNIX calls such a device a "hk" and the standalone name for the first partition of such a device is "hk(0,0)". If the controller were on the second UNIBUS its name would be "hk(8,0)". If we wished to access the first partition of a RK07 drive 1 on uba0 we would use "hk(1,0)".

The UNIBUS disk and tape names used by UNIX are:

RK disks	hk
TS tapes	ts
UDA disks	ra
IDC disks	rb
SMD disks	up
TM tapes	tm
TU tapes	ut

Here SMD disks are disks on an RM emulating controller on the UNIBUS, and TM tapes are tapes on a controller that emulates the DEC TM-11. TU tapes are tapes on a controller that emulates the DEC TU45. IDC disks are disks on an 11/730 Integral Disk Controller. TS tapes are tapes on a controller that emulates the DEC TS-11 (e.g. a TU80). The naming conventions for partitions in UNIBUS disks and files in UNIBUS tapes are the same as those for MASSBUS disks and tapes.

1.5. UNIX devices: block and raw

UNIX makes a distinction between "block" and "raw" (character) devices. Each disk has a block device interface where the system makes the device byte addressable and you can write a single byte in the middle of the disk. The system will read out the data from the disk sector, insert the byte you gave it and put the modified data back. The disks with the names "/dev/xx0a", etc are block devices. There are also raw devices available. These have names like "/dev/rxx0a", the "r" here standing for "raw". In the bootstrap procedures we will often suggest using the raw devices, because these tend to work faster in some cases. In general, however, the block devices are used. They are where file systems are "mounted".

You should be aware that it is sometimes important to use the character device (for efficiency) or not (because it wouldn't work, e.g. to write a single byte in the middle of a sector). Don't change the instructions by using the wrong type of device indiscriminately.

2. BOOTSTRAP PROCEDURE

This section explains the bootstrap procedure that can be used to get the kernel supplied with this tape running on your machine. Even if you are currently running UNIX you will have to do a full bootstrap.

If you are already running UNIX you should first save your existing files on magnetic tape. 4.2BSD uses a totally different file system organization than previous versions of the system; it is thus necessary to rebuild the file system format before restoring the data. The easiest way to save the current files on tape is by doing a full dump and then restoring under the new system. Refer to chapter 3 in understanding how to upgrade an existing 4BSD system.

Booting from tape

The tape bootstrap procedure used to create a working system involves the following major steps:

1) Format a disk pack with the *format* program.
2) Copy a "mini root" file system from the tape onto the swap area of the disk.
3) Boot the UNIX system on the "mini root".
4) Restore the full root file system using *restore*(8).
5) Build a console floppy or cassette for bootstrapping.
6) Reboot the completed root file system.
7) Build and restore the /usr file system from tape with *tar*(1).

Certain of these steps are dependent on your hardware configuration. Formatting the disk pack used for the root file system may require using the DEC standard formatting programs. Also, if you are bootstrapping the system on an 11/750, no console cassette is created.

The following sections describe the above steps in detail. In these sections references to disk drives are of the form *xx*(*n*,*m*) and references to files on tape drives are of the form *yy*(*n*,*m*) where *xx* and *yy* are one of the names described in section 1.4 and *n* and *m* are the unit and offset numbers described in section 1.4. Commands you are expected to type are shown in roman, while that information printed by the system is shown emboldened. Throughout the installation steps the reboot switch on an 11/780 or 11/730 should be set to off; on an 11/750 set the power-on action to halt. (In normal operation an 11/780 or 11/730 will have the reboot switch on and an 11/750 will have the power-on action set to reboot/restart.)

If you encounter problems in following the instructions in this part of the document, refer to Appendix C for help in troubleshooting.

2.1. Step 1: formatting the disk

All disks used with 4.2BSD should be formatted to insure the proper handling of physically corrupted disk sectors. If you have DEC disk drives, you should use the standard DEC formatter to format your disks. If not, the *format* program included in the distribution, or a vendor supplied formatting program, may be used to format disks. The *format* program is capable of formatting any of the following supported distribution devices:

EMULEX MASSBUS:	AMPEX 300M, 330M, CDC 300M, FUJITSU 404M
EMULEX SC-21V UNIBUS:	AMPEX 300M, 330M, CDC 300M, FUJITSU 160M, 404M

If you have run a pre-4.1BSD version of UNIX on the packs you are planning to use for bootstrapping it is likely that the bad sector information on the packs has been destroyed, since it was accessible as normal data in the last several tracks of the disk. You should

therefore run the formatter again to make sure the information is valid.

On an 11/750, to use a disk pack as a bootstrap device, sectors 0 through 15, the disk sectors in the files "/vmunix" (the system image) and "/boot" (the program that loads the system image), and the file system indices that lead to these two files must not have any errors. On an 11/780 or 11/730, the "boot" program is loaded from the console medium and includes device drivers for the "hp" and "up" disks which perform ECC correction and bad sector forwarding; consequently, on these machines the system may be bootstrapped on these disks even if the disk is not error free in critical locations. In general, if the first 15884 sectors of your disk are clean you are safe; if not you can take your chances.

To load the *format* program, insert the distribution TU58 cassette or RX01 floppy disk in the appropriate console device (on the 11/730 use cassette 0) and perform the following steps.

If you have an 11/780 give the commands:

```
>>>HALT
>>>UNJAM
>>>LOAD FORMAT
>>>START 2
```

If you have an 11/750 give the commands:

```
>>>I
>>>B DDA0
=format
```

If you have an 11/730 give the commands:

```
>>>H
>>>I
>>>L DD0:FORMAT
>>>S 2
```

The *format* program should now be running and awaiting your input:

Disk format/check utility

Enable debugging (1=bse, 2=ecc, 3=bse+ecc)?

If you made a mistake loading the program off the TU58 cassette the "=" prompt should reappear and you can retype the program name. If something else happened, you may have a bad distribution cassette or floppy, or your hardware may be broken; refer to Appendix C for help in troubleshooting. If you are unable to load programs off the distributed medium, consult Appendix B for an alternate (more painful) approach.

Format will create sector headers and verify the integrity of each sector formatted by using the disk controller's "write check" command. Remember *format* runs only on the **up** and **hp** drives indicated above. *Format* will prompt for the information required as shown below. If you make a mistake in answering questions, "#" erases the last character typed, and "@" erases the current input line.

Enable debugging (0=none, 1=bse, 2=ecc, 3=bse+ecc)?
Device to format? *xx*(0,0)
...(the old bad sector table is read; ignore any errors that occur here)...
Formatting drive *xx*0 on adaptor 0: verify (yes/no)? *yes*
Device data: #cylinders=842, #tracks=20, #sectors=48
Available test patterns are:
1 - (f00f) RH750 worst case
2 - (ec6d) media worst case
3 - (a5a5) alternating 1's and 0's
4 - (ffff) Severe burnin (takes several hours)
Pattern (one of the above, other to restart)? *2*
Start formatting...make sure the drive is online
...(soft ecc's and other errors are reported as they occur)...
...(if 4 write check errors were found, the program terminates like this)...
Errors:
Write check: 4
Bad sector: 0
ECC: 0
Skip sector: 0
Total of 4 hard errors found.
Writing bad sector table at block 524256
(524256 is the block # of the first block in the bad sector table)
Done

Once the root device has been formatted, *format* will prompt for another disk to format. Halt the machine by typing "control-P" and "H" (the "H" is necessary only on an 11/780, but does not hurt on the other machines).

Enable debugging (1=bse, 2=ecc, 3=bse+ecc)?^P
>>>H

It may be necessary to format other drives before constructing file systems on them; this can be done at a later time with the steps just performed. *Format* can also be used in an extended test mode (pattern 4) that uses numerous test patterns in 46 passes to detect as many disk surface errors as possible; this test runs for many hours, depending on the CPU and controller. On an 11/780, this can be speeded up significantly by setting the clock fast.

2.2. Step 2: copying the mini-root file system

The second step is to run a simple program, *copy*, which copies a very small root file system into the second partition of the disk. This file system will serve as the base for creating the actual root file system to be restored. The version of the operating system maintained on the "mini-root" file system understands not to swap on top of itself, thereby allowing double use of the disk partition. *Copy* is loaded just as the *format* program was loaded; for example, on an 11/780:

(copy mini root file system)
>>>LOAD COPY
>>>START 2
From: *yy*(*y*,**1**) (unit *y*, second tape file)
To: *xx*(*x*,**1**) (mini root is on drive *x*; second partition)
Copy completed: 205 records copied
From:

while for an 11/750:

```
(copy mini root file system)
>>>B DDA0
=copy
From: yy(y,1)                    (unit y, second tape file)
To: xx(x,1)                      (mini root is on drive x; second partition)
Copy completed: 205 records copied
From:
```

and for an 11/730:

```
(copy mini root file system)
>>>L DD0:COPY
>>>S 2
From: yy(y,1)                    (unit y, second tape file)
To: xx(x,1)                      (mini root is on drive x; second partition)
Copy completed: 205 records copied
From:
```

(As above, '#' erases characters and '@' erases lines.)

2.3. Step 3: booting from the mini-root file system

You now have the minimal set of tools necessary to create a root file system and restore the file system contents from tape. To access this file system load the bootstrap program and boot the version of unix which has been placed in the "mini-root":

```
(load bootstrap program)
>>>LOAD BOOT
>>>START 2
Boot
: xx(x,1)vmunix                  (bring in vmunix off mini root)
```

or, on an 11/750:

```
(load bootstrap program)
>>>B DDA0
=boot
Boot
: xx(x,1)vmunix                  (bring in vmunix off mini root)
```

or, on an 11/730:

```
(load bootstrap program)
>>>L DD0:BOOT
>>>S 2
Boot
: xx(x,1)vmunix                  (bring in vmunix off mini root)
```

(As above, '#' erases characters and '@' erases lines.)

The standalone boot program should then read the system from the mini root file system you just created, and the system should boot:

215564+64088+69764 start 0xf98
4.2 BSD UNIX #1: Sun Feb 6 15:02:15 PST 1983
real mem = *xxx*
avail mem = *yyy*
... information about available devices ...
root device?

The first three numbers are printed out by the bootstrap programs and are the sizes of different parts of the system (text, initialized and uninitialized data). The system also allocates several system data structures after it starts running. The sizes of these structures are based on the amount of available memory and the maximum count of active users expected, as declared in a system configuration description. This will be discussed later.

UNIX itself then runs for the first time and begins by printing out a banner identifying the release and version of the system that is in use and the date it was compiled.

Next the *mem* messages give the amount of real (physical) memory and the memory available to user programs in bytes. For example, if your machine has only 512K bytes of memory, then xxx will be 523264, 1024 bytes less than 512K. The system reserves the last 1024 bytes of memory for use in error logging and doesn't count it as part of real memory.

The messages that came out next show what devices were found on the current processor. These messages are described in *autoconf*(4). The distributed system may not have found all the communications devices you have (dh's and dz's), or all the mass storage peripherals you have if you have more than two of anything. This will be corrected soon, when you create a description of your machine to configure UNIX from. The messages printed at boot here contain much of the information that will be used in creating the configuration. In a correctly configured system most of the information present in the configuration description is printed out at boot time as the system verifies that each device is present.

The "root device?" prompt was printed by the system and is now asking you for the name of the root file system to use. This happens because the distribution system is a *generic* system. It can be bootstrapped on any VAX cpu and with its root device and paging area on any available disk drive. You should respond to the root device question with *xx*0*. This response supplies two pieces of information: first, *xx*0 indicates the disk it is running on is drive 0 of type *xx*, secondly the "*" indicates the system is running "atop" the paging area. The latter is most important, otherwise the system will attempt to page on top of itself and chaos will ensue. You will later build a system tailored to your configuration that will not ask this question when it is bootstrapped.

root device? *xx*0*
WARNING: preposterous time in file system -- CHECK AND RESET THE DATE!
erase ^?, kill ^U, intr ^C
#

The "erase ..." message is part of /.profile that was executed by the root shell when it started. This message is present to remind you that the line character erase, line erase, and interrupt characters are set to be what is standard on DEC systems; this insures things are consistent with the DEC console interface characters.

2.4. Step 4: restoring the root file system

UNIX is now running, and the 'UNIX Programmer's manual' applies. The '#' is the prompt from the shell, and lets you know that you are the super-user, whose login name is "root". To complete installation of the bootstrap system two steps remain. First, the root file system must be created, and second a boot floppy or cassette must be constructed.

To create the root file system the shell script "xtr" should be run as follows:

#disk=*xx0* type=*tt* tape=*yy* xtr

where *xx0* is the name of the disk on which the root file system is to be restored (unit 0), *tt* is the type of drive on which the root file system is to be restored (see the table below), and *yy* is the name of the tape drive on which the distribution tape is mounted.

If the root file system is to reside on a disk other than unit 0 (as shown in the information printed out during autoconfiguration), you will have to create the necessary special files in /dev and use the appropriate value. For example, if the root should be placed on hp1, you must create /dev/rhp1a and /dev/hp1a using *mknod*(8).

Drive	Type	Drive	Type
DEC RM03	type=rm03	DEC RM05	type=rm05
DEC RM80	type=rm80	DEC RP06	type=rp06
DEC RP07	type=rp07	DEC RK07	type=rk07
DEC RA80	type=ra80	DEC RA60	type=ra60
DEC RA81	type=ra81	DEC R80	type=rb80
CDC 9766	type=9766	CDC 9775	type=9775
AMPEX 300M	type=9300	AMPEX 330M	type=capricorn
FUJITSU 160M	type=fuji160	FUJITSU 404M	type=eagle

This will generate many messages regarding the construction of the file system and the restoration of the tape contents, but should eventually terminate with the messages:

```
...
Root filesystem extracted

If this is a 780, update floppy
If this is a 730, update the cassette
#
```

2.5. Step 5: creating a boot floppy or cassette

If the machine is an 11/780 or 11/730, a boot floppy or cassette should be constructed according to the instructions in chapter 4. For 11/750's, bootstrapping is performed by using a boot prom and special code located in sectors 0-15 of the root file system. The *newfs* program automatically installs the needed code, so you may continue on to the next step. On an 11/780 with interleaved memory, or other configurations that require alteration of the standard boot files, this step may be left for later.

2.6. Step 6: rebooting the completed root file system

With the above work completed, all that is left is to reboot:

```
#sync                                   (synchronize file system state)
#^P                                     (halt machine)
>>>HALT                                 (for 11/780's only)
>>>UNJAM                                (for 11/780's only)
>>>I                                    (initialize processor state)
>>>B xxS                                (on an 11/750, use B/2)
...(boot program is eventually loaded)...
Boot
: xx(x,0)vmunix                         (vmunix brought in off root)
215564+64088+69764 start 0xf98
4.2 BSD UNIX #1: Sun Feb 6 15:02:15 PST 1983
real mem  = xxx
avail mem = yyy
... information about available devices ...
root on xx0
WARNING: preposterous time in file system -- CHECK AND RESET THE DATE!
erase ^?, kill ^U, intr ^C
#
```

(see section 6.1 if the system does not reboot properly)

The system is now running single user on the installed root file system. The next section tells how to complete the installation of distributed software on the /usr file system.

2.7. Step 7: setting up the /usr file system

First set a shell variable to the name of your disk, so the commands we give will work regardless of the disk you have; do one of

```
# disk=hp    (if you have an RP06, RM03, RM05, RM80, or other MASSBUS drive)
# disk=hk    (if you have RK07s)
# disk=ra    (if you have UDA50 storage module drives)
# disk=up    (if you have UNIBUS storage module drives)
# disk=rb    (if you have IDC storage module drives)
```

The next thing to do is to extract the rest of the data from the tape. You might wish to review the disk configuration information in section 4.4 before continuing; the partitions used below are those most appropriate in size. Find the disk you have in the following table and execute the commands in the right hand portion of the table:

DEC RM03	# name=hp0g; type=rm03
DEC RM05	# name=hp0g; type=rm05
DEC RM80	# name=hp0g; type=rm80
DEC RP06	# name=hp0g; type=rp06
DEC RP07	# name=hp0h; type=rp07
DEC RK07	# name=hk0g; type=rk07
DEC RA80	# name=ra0h; type=ra80
DEC RA60	# name=ra0h; type=ra60
DEC RA81	# name=ra0h; type=ra81
DEC R80	# name=rb0h; type=rb80
UNIBUS CDC 9766	# name=up0g; type=9766
UNIBUS AMPEX 300M	# name=up0g; type=9300
UNIBUS AMPEX 330M	# name=up0g; type=capricorn
UNIBUS FUJITSU 160M	# name=up0g; type=fuji160
UNIBUS FUJITSU 404M	# name=up0h; type=eagle
MASSBUS CDC 9766	# name=hp0g; type=9766
MASSBUS AMPEX 300M	# name=hp0g; type=9300
MASSBUS AMPEX 330M	# name=hp0g; type=capricorn
MASSBUS FUJITSU 404M	# name=hp0h; type=eagle

Find the tape you have in the following table and execute the commands in the right hand portion of the table:

DEC TE16/TU45/TU77	# cd /dev; MAKEDEV ht0; sync
DEC TU78	# cd /dev; MAKEDEV mt0; sync
DEC TS11	# cd /dev; MAKEDEV ts0; sync
EMULEX TC11	# cd /dev; MAKEDEV tm0; sync
SI 9700	# cd /dev; MAKEDEV ut0; sync

Then execute the following commands

```
# date yymmddhhmm                (set date, see date(1))
....
# passwd root                    (set password for super-user)
New password:                    (password will not echo)
Retype new password:
# newfs ${name} ${type}          (create empty user file system)
(this takes a few minutes)
# mount /dev/${name} /usr        (mount the usr file system)
# cd /usr                        (make /usr the current directory)
# mkdir sys                      (make directory for system source)
# cd sys                         (make /usr/sys the current directory)
# mt fsf
# tar xpbf 20 /dev/rmt12         (extract the system source)
(this takes about 5-10 minutes)
# cd ..                          (back to /usr)
# mt fsf
# tar xpbf 20 /dev/rmt12         (extract all of usr except usr/src)
(this takes about 15-20 minutes)
# cd /                           (back to root)
# chmod 755  /  /usr  /usr/sys
# rm -f sys
# ln -s /usr/sys sys             (make a symbolic link to the system source)
# umount /dev/${name}            (unmount /usr)
```

The data on the fourth and fifth tape files has now been extracted and the first reel of the distribution is no longer needed. The remainder of the installation procedure uses the second reel of tape which should be mounted in place of the first.

You can check the consistency of the /usr file system by doing

```
# fsck /dev/r${name}
```

The output from *fsck* should look something like:

```
** /dev/rxx0h
** Last Mounted on /usr
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
671 files, 3497 used, 137067 free (75 frags, 34248 blocks)
```

If there are inconsistencies in the file system, you may be prompted to apply corrective action; see the document describing *fsck* for information.

To use the /usr file system, you should now remount it by saying

```
# /etc/mount /dev/${name} /usr
```

You can now extract the first file on the second tape (the source for the commands). If you have RK07's you must first put a formatted pack in drive 1 and set up a UNIX file system on it by doing:

```
# newfs hk1g rk07
(this takes a few minutes)
# mount /dev/hk1g /usr/src
# cd /usr/src
```

In any case you can then extract the source code for the commands (except on RK07's this will fit in the /usr file system):

```
# mkdir /usr/src
# chmod 755 /usr/src
# cd /usr/src
# tar xpb 20
```

If you get an error at this point, you can reposition the tape with the following command and try the above commands again.

```
# mt rew
```

2.8. Additional software

There are three extra tape files on the distribution tapes which have not been installed to this point. They are a font library for use with Varian and Versatec printers, the Ingres database system, and user contributed software. All three tapes files are in *tar*(1) format and can be installed by positioning the tape and reading in the files as was done for /usr/src above. As distributed, the fonts should be placed in a directory /usr/lib/vfont, the Ingres system should be placed in /usr/ingres, and the user contributed software should be placed in /usr/src/new. The exact contents of the user contributed software is given in a separate document.

3. UPGRADING A 4BSD SYSTEM

Begin by reading the other parts of this document to see what has changed since the last time you bootstrapped the system. Also read the "Changes in 4.2BSD" document, and look at the new manual sections provided to you. If you have local system modifications to the kernel to install, look at the document "Kernel changes in 4.2BSD" to get an idea of how the system changes will affect your local mods.

If you are running a version of the system distributed prior to 4.0BSD, you are pretty much on your own. Sites running 3BSD or 32/V may be able to modify the restor program to understand the old 512 byte block file system, but this has never been tried. This section assumes you are running 4.1BSD.

3.1. Step 1: what to save

No matter what version of the system you may be running, you will have to rebuild your root and usr file systems. The easist way to do this is to save the important files on your existing system, perform a bootstrap as if you were installing 4.2BSD on a brand new machine, then merge the saved files into the new system. The following list enumerates the standard set of files you will want to save and indicates directories in which site specific files should be present. This list will likely be augmented with non-standard files you have added to your system; be sure to do a tar of the directories /etc, /lib, and /usr/lib to guard against your missing something the first time around.

/.profile	root sh startup script
/.login	root csh startup script
/.cshrc	root csh startup script
/dev/MAKE	for the LOCAL case for making devices
/etc/fstab	disk configuration data
/etc/group	group data base
/etc/passwd	user data base
/etc/rc	for any local additions
/etc/ttys	terminal line configuration data
/etc/ttytype	terminal line to terminal type mapping data
/etc/termcap	for any local entries which may have been added
/lib	for any locally developed language processors
/usr/dict/*	for local additions to words and papers
/usr/include/*	for local additions
/usr/lib/aliases	mail forwarding data base
/usr/lib/crontab	cron daemon data base
/usr/lib/font/*	for locally developed font libraries
/usr/lib/lint/*	for locally developed lint libraries
/usr/lib/tabset/*	for locally developed tab setting files
/usr/lib/term/*	for locally developed nroff drive tables
/usr/lib/tmac/*	for locally developed troff/nroff macros
/usr/lib/uucp/*	for local uucp configuration files
/usr/man/manl	for manual pages for locally developed programs
/usr/msgs	for current msgs
/usr/spool/*	for current mail, news, uucp files, etc.
/usr/src/local	for source for locally developed programs

As 4.1BSD binary images will run unchanged under 4.2BSD you should be certain to save any programs such as compilers which you will need in bootstrapping to 4.2BSD.*

* 4.2BSD can support a "4.1BSD compatibility mode" of system operation whereby system calls from

Once you have saved the appropriate files in a convenient format, the next step is to dump your file systems with *dump*(8). For the utmost of safety this should be done to magtape. However, if you enjoy gambling with your life (or you have a VERY friendly user community) and you have sufficient disk space, you can try converting your file systems in-place by using a disk partition. If you select the latter tact, a version of the 4.1BSD dump program which runs under 4.2 is provided in /etc/dump.4.1; be sure to read through this entire document before beginning the conversion. Beware that file systems created under 4.2BSD will use about 5-10% more disk space for file system related information than under 4.1BSD. Thus, before dumping each file system it is a good idea to remove any files which may be easily regenerated. Since most all programs will likely be recompiled under the new system your best bet is to remove any object files. File systems with at least 10% free space on them should restore into an equivalently sized 4.2BSD file system without problem.

Once you have dumped the file systems you wish to convert to 4.2BSD, install the system from the bootstrap tape as described in chapter 2, then proceed to the next section.

3.2. Step 2: merging

When your system is booting reliably and you have the 4.2BSD root and /usr file systems fully installed you will be ready to proceed to the next step in the conversion process: merging your old files into the new system.

Using the tar tape, or tapes, you created in step 1 extract the appropriate files into a scratch directory, say /usr/convert:

```
# mkdir /usr/convert
# cd /usr/convert
# tar x
```

Certain data files, such as those from the /etc directory, may simply be copied into place.

```
# cp passwd group fstab ttys ttytype /etc
# cp crontab /usr/lib
```

Other files, however, must be merged into the distributed versions by hand. In particular, be careful with /etc/termcap.

The commands kept under the LOCAL entry in /dev/MAKE should be placed in the new shell script /dev/MAKEDEV.local so that saying "MAKEDEV LOCAL" will create the appropriate local devices and device names. If you have any homegrown device drivers which use major device numbers reserved by the system you will have to modify the commands used to create the devices or alter the system device configuration tables in /sys/vax/conf.c.

The spooling directories saved on tape may be restored in their eventual resting places without too much concern. Be sure to use the 'p' option to tar so that files are recreated with the same file modes:

```
# cd /usr
# tar xp msgs spool/mail spool/uucp spool/uucppublic spool/news
```

Whatever else is left is likely to be site specific or require careful scrutiny before placing in its eventual resting place. Refer to the documentation and source code before arbitrarily overwriting a file.

4.1BSD are either emulated or safely ignored. There are only two exceptions; programs which read directories or use the old jobs library will not operate properly. However, while 4.1BSD binaries will execute under 4.2BSD it is **STRONGLY RECOMMENDED** that the programs be recompiled under the new system. Refer to the document "Changes in 4.2BSD" for elaboration on this point.

3.3. Step 3: converting file systems

The dump format used in 4.0 and 4.1BSD is upward compatible with that used in 4.2BSD. That is, the 4.2BSD *restore* program understands how to read old dump tapes, although 4.2BSD dump tapes may not be properly restored under 4.0BSD or 4.1BSD. To convert a file system dumped to magtape, simply create the appropriate file system and restore the data. Note that the 4.2BSD *restore* program does its work on a mounted file system using normal system operations (unlike the older *restor* which accessed the raw file system device and deposited inodes in the appropriate locations on disk). This means that file system dumps may be restored even if the characteristics of the file system changed. To restore a dump tape for, say, the /a file system something like the following would be used:

```
# mkdir /a
# newfs hp1g eagle
# mount /dev/hp1g /a
# cd /a
# restore r
```

If tar images were written instead of doing a dump, you should be sure to use the 'p' option when reading the files back. No matter how you restore a file system, be sure and check its integrity with fsck when the job is complete.

3.4. Bootstrapping language processors

To convert a compiler from 4.1BSD to 4.2BSD you should simply have to recompile and relink the various parts. If the processor is written in itself, for instance a PASCAL compiler written in PASCAL, the important step in converting is to save a working copy of the 4.1BSD binary before converting to 4.2BSD. Then, once the system has been changed over, the 4.1BSD binary should be used in the rebuilding process. In order to do this, you should enable the 4.1 compatibility option when you configure the kernel (below).

If no working 4.1BSD binary exists, or the language processor uses some nonstandard system call, you will likely have to compile the language processor into an intermediate form, such as assembly language, on a 4.1BSD system, then bring the intermediate form to 4.2BSD for assembly and loading.

4. SYSTEM SETUP

This section describes procedures used to setup a VAX UNIX system. Procedures described here are used when a system is first installed or when the system configuration changes. Procedures for normal system operation are described in the next section.

4.1. Making a UNIX boot floppy

If you have an 11/780 you will want to create a UNIX boot floppy by adding some files to a copy of your current DEC console floppy, using *flcopy*(8) and *arff*(8). This floppy will make standalone system operations such as bootstrapping much easier.

First change into the directory where the console floppy information is stored:

```
# cd /sys/floppy
```

then set up the default boot device. If you have an RK07 as your primary root do:

```
# cp defboo.hk defboo.cmd
```

If you have a drive on a UDA50 (e.g. an RA81) as your primary root do:

```
# cp defboo.ra defboo.cmd
```

If you have a second vendor UNIBUS storage module as your primary root do:

```
# cp defboo.up defboo.cmd
```

Otherwise:

```
# cp defboo.hp defboo.cmd
```

If the local configuration requires any changes in restar.cmd or defboo.cmd (e.g., for interleaved memory controllers), these should be made now. The following command will then copy your DEC local console floppy, updating the copy appropriately.

```
# make update
Change Floppy, Hit return when done.
(waits for you to put clean floppy in console)
Are you sure you want to clobber the floppy? yes
```

More copies of this floppy can be made using *flcopy*(8).

4.2. Making a UNIX boot cassette

If you have an 11/730 you will want to create a UNIX boot cassette by adding some files to a copy of your current DEC console cassette, using *flcopy*(8) and *arff*(8). This cassette will make standalone system operations such as bootstrapping much easier.

First change into the directory where the console cassette information is stored:

```
# cd /sys/cassette
```

then set up the default boot device. If you have an IDC storage module as your primary root do:

```
# cp defboo.rb defboo.cmd
```

If you have an RK07 as your primary root do:

```
# cp defboo.hk defboo.cmd
```

If you have a drive on a UDA50 as your primary root do:

```
# cp defboo.ra defboo.cmd
```

Otherwise:

```
# cp defboo.up defboo.cmd
```

To complete the procedure place your DEC local console cassette in drive 0 (the drive at front of the CPU); the following command will then copy it, updating the copy appropriately.

```
# make update
Change Floppy, Hit return when done.
(waits for you to put clean cassette in console drive 0)
Are you sure you want to clobber the floppy? yes
```

More copies of this cassette can best be made using *dd*(1).

4.3. Kernel configuration

This section briefly describes the layout of the kernel code and how files for devices are made. For a full discussion of configuring and building system images, consult the document "Building 4.2BSD UNIX Systems with Config".

4.3.1. Kernel organization

As distributed, the kernel source is in a separate tar image. The source may be physically located anywhere within any file system so long as a symbolic link to the location is created for the file /sys (many files in /usr/include are normally symbolic links relative to /sys). In further discussions of the system source all path names will be given relative to /sys.

The directory /sys/sys contains the mainline machine independent operating system code. Files within this directory are conventionally named with the following prefixes.

init_	system initialization
kern_	kernel (authentication, process management, etc.)
quota_	disk quotas
sys_	system calls and similar
tty_	terminal handling
ufs_	file system
uipc_	interprocess communication
vm_	virtual memory

The remaining directories are organized as follows.

/sys/h	machine independent include files
/sys/conf	site configuration files and basic templates
/sys/net	network independent, but network related code
/sys/netinet	DARPA Internet code
/sys/netimp	IMP support code
/sys/netpup	PUP-1 support code
/sys/vax	VAX specific mainline code
/sys/vaxif	VAX network interface code
/sys/vaxmba	VAX MASSBUS device drivers and related code
/sys/vaxuba	VAX UNIBUS device drivers and related code

Many of these directories are referenced through /usr/include with symbolic links. For example, /usr/include/sys is a symbolic link to /sys/h. The system code, as distributed, is totally independent of the include files in /usr/include. This allows the system to be recompiled from scratch without the /usr file system mounted.

4.3.2. Devices and device drivers

Devices supported by UNIX are implemented in the kernel by drivers whose source is kept in /sys/vax, /sys/vaxuba, or /sys/vaxmba. These drivers are loaded into the system when included in a cpu specific configuration file kept in the conf directory. Devices are accessed through special files in the file system, made by the *mknod*(8) program and normally kept in the /dev directory. For all the devices supported by the distribution system, the files in /dev are created by the /dev/MAKEDEV shell script.

Determine the set of devices that you have and create a new /dev directory by running the MAKEDEV script. First create a new directory /newdev, copy MAKEDEV into it, edit the file MAKEDEV.local to provide an entry for local needs, and run it to generate a /newdev directory. For instance, if your machine has a single dz-11, a single dh-11, a single dmf-32, an rm03 disk, an EMULEX controller, an AMPEX-9300 disk, and a te16 tape drive you would do:

```
# cd /
# mkdir newdev
# cp dev/MAKEDEV newdev/MAKEDEV
# cd newdev
# MAKEDEV dz0 dh0 dmf0 hp0 up0 ht0 std LOCAL
```

Note the "std" argument causes standard devices such as /dev/console, the machine console, /dev/floppy, the console floppy disk interface for the 11/780, and /dev/tu0 and /dev/tu1, the console cassette interfaces for the 11/750 and 11/730, to be created.

You can then do

```
# cd /
# mv dev olddev ; mv newdev dev
# sync
```

to install the new device directory.

4.3.3. Building new system images

The kernel configuration of each UNIX system is described by a single configuration file, stored in the /sys/conf directory. To learn about the format of this file and the procedure used to build system images, start by reading "Building 4.2BSD UNIX Systems with Config", look at the manual pages in section 4 of the UNIX manual for the devices you have, and look at the configuration files in the /sys/conf directory.

The configured system image "vmunix" should be copied to the root, and then booted to try it out. It is best to name it /newvmunix so as not to destroy the working system until you're sure it does work:

```
# cp vmunix /newvmunix
# sync
```

It is also a good idea to keep the old system around under some other name. In particular, we recommend that you save the generic distribution version of the system permanently as /genvmunix for use in emergencies.

To boot the new version of the system you should follow the bootstrap procedures outlined in section 6.1. A systematic scheme for numbering and saving old versions of the system is best.

4.4. Disk configuration

This section describes how to layout file systems to make use of the available space and to balance disk load for better system performance.

4.4.1. Initializing /etc/fstab

Change into the directory /etc and copy the appropriate file from:

fstab.rm03
fstab.rm05
fstab.rm80
fstab.ra60
fstab.ra80
fstab.ra81
fstab.rb80
fstab.rp06
fstab.rp07
fstab.rk07
fstab.up160m (160Mb up drives)
fstab.up300m (300Mb up drives)
fstab.hp400m (400Mb hp drives)
fstab.up (other up drives)
fstab.hp (other hp drives)

to the file /etc/fstab, i.e.:

```
# cd /etc
# cp fstab.xxx fstab
```

This will set up the initial information about the usage of disk partitions, which we see how to update more below.

4.4.2. Disk naming and divisions

Each physical disk drive can be divided into up to 8 partitions; UNIX typically uses only 3 or 4 partitions. For instance, on an RM03 or RP06, the first partition, hp0a, is used for a root file system, a backup thereof, or a small file system like, /tmp; the second partition, hp0b, is used for paging and swapping; and the third partition hp0g holds a user file system. On an RM05, the first three partitions are used as for the RM03, and the fourth partition, hp0h, is used to hold the /usr file system, including source code.

The disk partition sizes for a drive are based on a set of four default partition tables; c.f. *diskpart* (8). The particular table used is dependent on the size of the drive. The "a" partition is the same size across all drives, 15884 sectors. The "b" partition, used for paging and swapping, is sized according to the total space on the disk. For drives less than about 400 megabytes the partition is 33440 sectors, while for larger drives the partition size is doubled to 66880 sectors. The "c" partition is always used to access the entire physical disk, including the space at the back of the disk reserved for the bad sector forwarding table. If the disk is larger than about 250 megabytes, an "h" partition is created with size 291346 sectors, and no matter whether the "h" partition is created or not, the remainder of the drive is allocated to the "g" partition. Sites which want to split up the "g" partition into a number of smaller file systems may use the "d", "e", and "f" partitions which overlap the "g" partition. The default sizes for these partitions are 15884, 55936, and the remainder of the disk, respectively*.

4.4.3. Space available

The space available on a disk varies per device. The amount of space available on the common disk partitions is listed in the following table. Not shown in the table are the partitions of each drive devoted to the root file system and the paging area.

---

* These rules are, unfortunately not evenly applied to all disks. Drives on DEC UDA50 and IDC controllers do not completely follow these rules; in particular, the swap partition on an RA81 is only 33440 sectors, and no "d", "e", or "f" partitions are available on an RA60 or RA80. Consult *uda* (4) for more information.

Type	Name	Size	Name	Size
rk07	hk?g	13 Mb		
rm03	hp?g	41 Mb		
rp06	hp?g	145 Mb		
rm05	hp?g	80 Mb	hp?h	145 Mb
rm80	hp?g	96 Mb		
ra60	ra?g	41 Mb	ra?h	139 Mb
ra80	ra?g	41 Mb	ra?h	56 Mb
ra81	ra?g	41 Mb	ra?h	380 Mb
rb80	rb?g	41 Mb	rb?h	56 Mb
rp07	hp?g	315 Mb	hp?h	145 Mb
up300	up?g	80 Mb	up?h	145 Mb
hp400	hp?g	216 Mb	hp?h	145 Mb
up160	up?g	106 Mb		

Here up300 refers to either an AMPEX or CDC 300 Megabyte disk on a UNIBUS disk controller, up160 refers to a FUJITSU 160 Megabyte disk on the UNIBUS, and hp400 refers to a FUJITSU Eagle 400 Megabyte disk on a MASBUS disk controller. Consult the manual pages for the specific controllers for other supported disks or other partitions.

Each disk also has a paging area, typically of 16 Megabytes, and a root file sytem of 8 Megabytes. The distributed system binaries occupy about 22 Megabytes while the major sources occupy another 25 Megabytes. This overflows dual RK07 and dual RL02 systems, but fits easily on most other hardware configurations.

Be aware that the disks have their sizes measured in disk sectors (512 bytes), while the UNIX file system blocks are variable sized. All user programs report disk space in kilobytes and, where needed, disk sizes are always specified in terms of sectors. The /etc/disktab file used in making file systems specifies disk partition sizes in sectors; the default sector size of 512 bytes may be overridden with the "se" attribute.

4.4.4. Layout considerations

There are several considerations in deciding how to adjust the arrangement of things on your disks: the most important is making sure there is adequate space for what is required; secondarily, throughput should be maximized. Paging space is an important parameter. The system, as distributed, sizes the configured paging areas each time the system is booted. Further, multiple paging areas of different size may be interleaved. Drives smaller than 400 megabytes have swap partitions of 16 megabytes while drives larger than 400 megabytes have 32 megabytes. These values may be changed to get more paging space by changing the appropriate partition table in the disk driver.

Many common system programs (C, the editor, the assembler etc.) create intermediate files in the /tmp directory, so the file system where this is stored also should be made large enough to accommodate most high-water marks; if you have several disks, it makes sense to mount this in a "root" (i.e. first partition) file system on another disk. All the programs that create files in /tmp take care to delete them, but are not immune to rare events and can leave dregs. The directory should be examined every so often and the old files deleted.

The efficiency with which UNIX is able to use the CPU is often strongly affected by the configuration of disk controllers. For general time-sharing applications, the best strategy is to try to split the root file system (/), system binaries (/usr), the temporary files (/tmp), and the user files among several disk arms, and to interleave the paging activity among a several arms.

It is critical for good performance to balance disk load. There are at least five components of the disk load that you can divide between the available disks:

1. The root file system.
2. The /tmp file system.
3. The /usr file system.
4. The user files.
5. The paging activity.

The following possibilities are ones we have used at times when we had 2, 3 and 4 disks:

	disks		
what	2	3	4
/	1	2	2
tmp	1	3	4
usr	1	1	1
paging	1+2	1+3	1+3+4
users	2	2+3	2+3
archive	x	x	4

The most important things to consider are to even out the disk load as much as possible, and to do this by decoupling file systems (on separate arms) between which heavy copying occurs. Note that a long term average balanced load is not important... it is much more important to have instantaneously balanced load when the system is busy.

Intelligent experimentation with a few file system arrangements can pay off in much improved performance. It is particularly easy to move the root, the /tmp file system and the paging areas. Place the user files and the /usr directory as space needs dictate and experiment with the other, more easily moved file systems.

4.4.5. File system parameters

Each file system is parameterized according to its block size, fragment size, and the disk geometry characteristics of the medium on which it resides. Inaccurate specification of the disk characteristics or haphazard choice of the file system parameters can result in substantial throughput degradation or significant waste of disk space. As distributed, file systems are configured according to the following table.

File system	Block size	Fragment size
/	8 Kbytes	1 Kbytes
usr	4 Kbytes	512 bytes
users	4 Kbytes	1 Kbytes

The root file system block size is made large to optimize bandwidth to the associated disk; this is particularly important since the /tmp directory is normally part of the root file. The large block size is also important as many of the most heavily used programs are demand paged out of the /bin directory. The fragment size of 1 Kbytes is a "nominal" value to use with a file system. With a 1 Kbyte fragment size disk space utilization is approximately the same as with the earlier versions of the file system.

The usr file system uses a 4 Kbyte block size with 512 byte fragment size in an effort to get high performance while conserving the amount of space wasted by a large fragment size. Space compaction has been deemed important here because the source code for the system is normally placed on this file system.

The file systems for users have a 4 Kbyte block size with 1 Kbyte fragment size. These parameters have been selected based on observations of the performance of our user file systems. The 4 Kbyte block size provides adequate bandwidth while the 1 Kbyte fragment size provides acceptable space compaction and disk fragmentation.

Other parameters may be chosen in constructing file systems, but the factors involved in choosing a block size and fragment size are many and interact in complex ways. Larger block sizes result in better throughput to large files in the file system as larger i/o requests will then be performed by the system. However, consideration must be given to the average file sizes found in the file system and the performance of the internal system buffer cache. The system currently provides space in the inode for 12 direct block pointers, 1 single indirect block pointer, and 1 double indirect block pointer.* If a file uses only direct blocks, access time to it will be optimized by maximizing the block size. If a file spills over into an indirect block, increasing the block size of the file system may decrease the amount of space used by eliminating the need to allocate an indirect block. However, if the block size is increased and an indirect block is still required, then more disk space will be used by the file because indirect blocks are allocated according to the block size of the file system.

In selecting a fragment size for a file system, at least two considerations should be given. The major performance tradeoffs observed are between an 8 Kbyte block file system and a 4 Kbyte block file system. Due to implementation constraints, the block size / fragment size ratio can not be greater than 8. This means that an 8 Kbyte file system will always have a fragment size of at least 1 Kbytes. If a file system is created with a 4 Kbyte block size and a 1 Kbyte fragment size, then upgraded to an 8 Kbyte block size and 1 Kbyte fragment size, identical space compaction will be observed. However, if a file system has a 4 Kbyte block size and 512 byte fragment size, converting it to an 8K/1K file system will result in significantly more space being used. This implies that 4 Kbyte block file systems which might be upgraded to 8 Kbyte blocks for higher performance should use fragment sizes of at least 1 Kbytes to minimize the amount of work required in conversion.

A second, more important, consideration when selecting the fragment size for a file system is the level of fragmentation on the disk. With a 512 byte fragment size, storage fragmentation occurs much sooner, particularly with a busy file system running near full capacity. By comparison, the level of fragmentation in a 1 Kbyte fragment file system is an order of magnitude less severe. This means that on file systems where many files are created and deleted the 512 byte fragment size is more likely to result in apparent space exhaustion due to fragmentation. That is, when the file system is nearly full, file expansion which requires locating a contiguous area of disk space is more likely to fail on a 512 byte file system than on a 1 Kbyte file system. To minimize fragmentation problems of this sort, a parameter in the super block specifies a minimum acceptable free space threshhold. When normal users (i.e. anyone but the super-user) attempt to allocate disk space and the free space threshold is exceeded, the user is returned an error as if the file system were actually full. This parameter is nominally set to 10%; it may be changed by supplying a parameter to *newfs*, or by patching the super block of an existing file system.

In general, unless a file system is to be used for a special purpose application (for example, storing image processing data), we recommend using the default values supplied. Remember that the current implementation limits the block size to at most 8 Kbytes and the ratio of block size / fragment size must be in the range 1-8.

The disk geometry information used by the file system affects the block layout policies employed. The file /etc/disktab, as supplied, contains the data for most all drives supported by the system. When constructing a file system you should use the *newfs*(8) program and specify the type of disk on which the file system resides. This file also contains the default file system partition sizes, and default block and fragment sizes. To override any of the default values you can modify the file or use one of the options to *newfs*.

* A triple indirect block pointer is also reserved, but not currently supported.

4.4.6. Implementing a layout

To put a chosen disk layout into effect, you should use the *newfs*(8) command to create each new file system. Each file system must also be added to the file /etc/fstab so that it will be checked and mounted when the system is bootstrapped.

As an example, consider a system with rm03's. On the first rm03, hp0, we will put the root file system in hp0a, and the /usr file system in hp0g, which has enough space to hold it and then some. The /tmp directory will be part of the root file system, as no file system will be mounted on /tmp. If we had only one rm03, we would put user files in the hp0g partition with the system source and binaries.

If we had a second rm03, we would create a file system in hp1g and put user files there, calling the file system /mnt. We would also interleave the paging between the 2 rm03's. To do this we would build a system configuration that specified:

```
config      vmunix    root on hp0 swap on hp0 and hp1
```

to get the swap interleaved, and add the lines

```
/dev/hp1b::sw::
/dev/hp1g:/mnt:rw:1:2
```

to the /etc/fstab file. We would keep a backup copy of the root file system in the **hp1a** disk partition.

To make the /mnt file system we would do:

```
# cd /dev
# MAKEDEV hp1
# newfs hp1g rm03
(information about file system prints out)
# mkdir /mnt
# mount /dev/hp1g /mnt
```

4.5. Configuring terminals

If UNIX is to support simultaneous access from more than just the console terminal, the file /etc/ttys (*ttys*(5)) has to be edited.

Terminals connected via dz interfaces are conventionally named **ttyDD** where DD is a decimal number, the "minor device" number. The lines on dz0 are named /dev/tty00, /dev/tty01, ... /dev/tty07. Lines on dh or dmf interfaces are conventionally named **ttyh**X, where X is a hexadecimal digit. If more than one dh or dmf interface is present in a configuration, successive terminals would be named **ttyi**X, **ttyj**X, etc.

To add a new terminal, be sure the device is configured into the system and that the special file for the device has been made by /dev/MAKEDEV. Then, set the first character of the appropriate line of /etc/ttys to 1 (or add a new line).

The second character of each line in the /etc/ttys file lists the speed and initial parameter settings for the terminal. The commonly used choices are:

```
0    300-1200-150-110
2    9600
3    1200-300
5    300-1200
```

Here the first speed is the speed a terminal starts at, and "break" switches speeds. Thus a newly added terminal /dev/tty00 could be added as

```
12tty00
```

if it was wired to run at 9600 baud. The definition of each "terminal type" is located in the

file /etc/gettytab and read by the *getty* program. To make custom terminal types, consult *gettytab*(5) before modifying this file.

Dialup terminals should be wired so that carrier is asserted only when the phone line is dialed up. For non-dialup terminals from which modem control is not available, you must either wire back the signals so that the carrier appears to always be present, or show in the system configuration that carrier is to be assumed to be present. See *dh*(4), *dz*(4), and *dmf*(4) for details.

You should also edit the file /etc/ttytype placing the type of each new terminal there (see *ttytype*(5)).

When the system is running multi-user, all terminals that are listed in /etc/ttys having a 1 as the first character of their line are enabled. If, during normal operations, it is desired to disable a terminal line, you can edit the file /etc/ttys and change the first character of the corresponding line to be a 0 and then send a hangup signal to the *init* process, by doing

```
# kill -1 1
```

Terminals can similarly be enabled by changing the first character of a line from a 0 to a 1 and sending a hangup signal to *init*.

Note that several programs, /usr/src/etc/init.c and /usr/src/etc/comsat.c in particular, will have to be recompiled if there are to be more than 100 terminals. Also note that if a special file is inaccessible when *init* tries to create a process for it, init will print a message on the console and try to reopen the terminal every minute, reprinting the warning message every 10 minutes.

Finally note that you should change the names of any dialup terminals to ttyd? where ? is in [0-9a-f], as some programs use this property of the names to determine if a terminal is a dialup. Shell commands to do this should be put in the /dev/MAKEDEV.local script.

While it is possible to use truly arbitrary strings for terminal names, the accounting and noticeably the *ps*(1) command make good use of the convention that tty names (by default, and also after dialups are named as suggested above) are distinct in the last 2 characters. Change this and you may be sorry later, as the heuristic *ps*(1) uses based on these conventions will then break down and ps will run MUCH slower.

4.6. Adding users

New users can be added to the system by adding a line to the password file /etc/passwd. The procedure for adding a new user is described in *adduser*(8).

You should add accounts for the initial user community, giving each a directory and a password, and putting users who will wish to share software in the same groups.

A number of guest accounts have been provided on the distribution system; these accounts are for people at Berkeley, DEC and at Bell Laboratories who have done major work on UNIX in the past. You can delete these accounts, or leave them on the system if you expect that these people would have occasion to login as guests on your system.

4.7. Site tailoring

All programs which require the site's name, or some similar characteristic, obtain the information through system calls or from files located in /etc. Aside from parts of the system related to the network, to tailor the system to your site you must simply select a site name, then edit the file

```
/etc/rc.local
```

The first line in /etc/rc.local,

```
/bin/hostname mysitename
```

defines the value returned by the *gethostname*(2) system call. Programs such as *getty*(8),

mail(1), *wall*(1), *uucp*(1), and *who*(1) use this system call so that the binary images are site independent.

4.8. Setting up the line printer system

The line printer system consists of at least the following files and commands:

/usr/ucb/lpq	spooling queue examination program
/usr/ucb/lprm	program to delete jobs from a queue
/usr/ucb/lpr	program to enter a job in a printer queue
/etc/printcap	printer configuration and capability data base
/usr/lib/lpd	line printer daemon, scans spooling queues
/etc/lpc	line printer control program

The file /etc/printcap is a master data base describing line printers directly attached to a machine and, also, printers accessible across a network. The manual page *printcap*(5) describes the format of this data base and also indicates the default values for such things as the directory in which spooling is performed. The line printer system handles multiple printers, multiple spooling queues, local and remote printers, and also printers attached via serial lines which require line initialization such as the baud rate. Raster output devices such as a Varian or Versatec, and laser printers such as an Imagen, are also supported by the line printer system.

Remote spooling via the network is handled with two spooling queues, one on the local machine and one on the remote machine. When a remote printer job is initiated with *lpr*, the job is queued locally and a daemon process created to oversee the transfer of the job to the remote machine. If the destination machine is unreachable, the job will remain queued until it is possible to transfer the files to the spooling queue on the remote machine. The *lpq* program shows the contents of spool queues on both the local and remote machines.

To configure your line printers, consult the printcap manual page and the accompanying document, "4.2BSD Line Printer Spooler Manual". A call to the *lpd* program should be present in /etc/rc.

4.9. Setting up the mail system

The mail system consists of the following commands:

/bin/mail	old standard mail program (from 32/V)
/usr/ucb/mail	UCB mail program, described in mail(1)
/usr/lib/sendmail	mail routing program
/usr/spool/mail	mail spooling directory
/usr/spool/secretmail	secure mail directory
/usr/bin/xsend	secure mail sender
/usr/bin/xget	secure mail receiver
/usr/lib/aliases	mail forwarding information
/usr/ucb/newaliases	command to rebuild binary forwarding database
/usr/ucb/biff	mail notification enabler
/etc/comsat	mail notification daemon
/etc/syslog	error message logger, used by sendmail

Mail is normally sent and received using the *mail*(1) command, which provides a front-end to edit the messages sent and received, and passes the messages to *sendmail*(8) for routing. The routing algorithm uses knowledge of the network name syntax, aliasing and forwarding information, and network topology, as defined in the configuration file /usr/lib/sendmail.cf, to process each piece of mail. Local mail is delivered by giving it to the program /usr/bin/mail which adds it to the mailboxes in the directory /usr/spool/mail/*username*, using a locking

protocol to avoid problems with simultaneous updates. After the mail is delivered, the local mail delivery daemon /etc/comsat is notified, which in turn notifies users who have issued a "biff y" command that mail has arrived.

Mail queued in the directory /usr/spool/mail is normally readable only by the recipient. To send mail which is secure against any possible perusal (except by a code-breaker) you should use the secret mail facility, which encrypts the mail so that no one can read it.

To setup the mail facility you should read the instructions in the file READ ME in the directory /usr/src/usr.lib/sendmail and then adjust the necessary configuration files. You should also set up the file /usr/lib/aliases for your installation, creating mail groups as appropriate. Documents describing *sendmail*'s operation and installation are also included in the distribution.

4.9.1. Setting up a uucp connection

The version of *uucp* included in 4.2BSD is an enhanced version of that originally distributed with 32/V*. The enhancements include:

- support for many auto call units other than the DEC DN11,
- breakup of the spooling area into multiple subdirectories,
- addition of an *L.cmds* file to control the set of commands which may be executed by a remote site,
- enhanced "expect-send" sequence capabilities when logging in to a remote site,
- new commands to be used in polling sites and obtaining snap shots of *uucp* activity.

This section gives a brief overview of *uucp* and points out the most important steps in its installation.

To connect two UNIX machines with a *uucp* network link using modems, one site must have an automatic call unit and the other must have a dialup port. It is better if both sites have both.

You should first read the paper in volume 2B of the Unix Programmers Manual: "Uucp Implementation Description". It describes in detail the file formats and conventions, and will give you a little context. In addition, the document setup.tblms, located in the directory /usr/src/usr.bin/uucp/UUAIDS, may be of use in tailoring the software to your needs.

The *uucp* support is located in three major directories: /usr/bin, /usr/lib/uucp, and /usr/spool/uucp. User commands are kept in /usr/bin, operational commands in /usr/lib/uucp, and /usr/spool/uucp is used as a spooling area. The commands in /usr/bin are:

/usr/bin/uucp	file-copy command
/usr/bin/uux	remote execution command
/usr/bin/uusend	binary file transfer using mail
/usr/bin/uuencode	binary file encoder (for *uusend*)
/usr/bin/uudecode	binary file decoder (for *uusend*)
/usr/bin/uulog	scans session log files
/usr/bin/uusnap	gives a snap-shot of *uucp* activity
/usr/bin/uupoll	polls remote system until an answer is received

The important files and commands in /usr/lib/uucp are:

* The *uucp* included in this distribution is the result of work by many people; we gratefully acknowledge their contributions, but refrain from mentioning names in the interest of keeping this document current.

/usr/lib/uucp/L-devices	list of dialers and hardwired lines
/usr/lib/uucp/L-dialcodes	dialcode abbreviations
/usr/lib/uucp/L.cmds	commands remote sites may execute
/usr/lib/uucp/L.sys	systems to communicate with, how to connect, and when
/usr/lib/uucp/SEQF	sequence numbering control file
/usr/lib/uucp/USERFILE	remote site pathname access specifications
/usr/lib/uucp/uuclean	cleans up garbage files in spool area
/usr/lib/uucp/uucico	*uucp* protocol daemon
/usr/lib/uucp/uuxqt	*uucp* remote execution server

while the spooling area contains the following important files and directories:

/usr/spool/uucp/C.	directory for command, "C." files
/usr/spool/uucp/D.	directory for data, "D.", files
/usr/spool/uucp/X.	directory for command execution, "X.", files
/usr/spool/uucp/D.*machine*	directory for local "D." files
/usr/spool/uucp/D.*machine*X	directory for local "X." files
/usr/spool/uucp/TM.	directory for temporary, "TM.", files
/usr/spool/uucp/LOGFILE	log file of *uucp* activity
/usr/spool/uucp/SYSLOG	log file of *uucp* file transfers

To install *uucp* on your system, start by selecting a site name (less than 8 characters). A *uucp* account must be created in the password file and a password set up. Then, create the appropriate spooling directories with mode 755 and owned by user *uucp*, group *daemon*.

If you have an auto-call unit, the L.sys, L-dialcodes, and L-devices files should be created. The L.sys file should contain the phone numbers and login sequences required to establish a connection with a *uucp* daemon on another machine. For example, our L.sys file looks something like:

```
adiron Any ACU 1200 out0123456789- ogin-EOT-ogin uucp
cbosg Never Slave 300
cbosgd Never Slave 300
chico Never Slave 1200 out2010123456
```

The first field is the name of a site, the second indicates when the machine may be called, the third field specifies how the host is connected (through an ACU, a hardwired line, etc.), then comes the phone number to use in connecting through an auto-call unit, and finally a login sequence. The phone number may contain common abbreviations which are defined in the L-dialcodes file. The device specification should refer to devices specified in the L-devices file. Indicating only ACU causes the *uucp* daemon, *uucico*, to search for any available auto-call unit in L-devices. Our L-dialcodes file is of the form:

```
ucb 2
out 9%
```

while our L-devices file is:

```
ACU cul0 unused 1200 ventel
```

Refer to the README file in the *uucp* source directory for more information about installation.

As *uucp* operates it creates (and removes) many small files in the directories underneath /usr/spool/uucp. Sometimes files are left undeleted; these are most easily purged with the *uuclean* program. The log files can grow without bound unless trimmed back; *uulog* is used to maintain these files. Many useful aids in maintaining your *uucp* installation are included in a subdirectory UUAIDS beneath /usr/src/usr.bin/uucp. Peruse this directory and read the "setup" instructions also located there.

5. NETWORK SETUP

4.2BSD provides support for the DARPA standard Internet protocols IP, ICMP, TCP, and UDP. These protocols may be used on top of a variety of hardware devices ranging from the IMP's used in the ARPANET to local area network controllers for the Ethernet. Network services are split between the kernel (communication protocols) and user programs (user services such as TELNET and FTP). This section describes how to configure your system to use the networking support.

5.1. System configuration

To configure the kernel to include the Internet communication protocols, define the INET option and include the pseudo-devices "inet", "pty", and "loop" in your machine's configuration file. The "pty" pseudo-device forces the pseudo terminal device driver to be configured into the system, see *pty*(4), while the "loop" pseudo-device forces inclusion of the software loopback interface driver. The loop driver is used in network testing and also by the mail system.

If you are planning to use the network facilities on a 10Mb/s Ethernet, the pseudo-device "ether" should also be included in the configuration; this forces inclusion of the Address Resolution Protocol module used in mapping between 48-bit Ethernet and 32-bit Internet addresses. Also, if you have an imp, you will need to include the pseudo-device "imp."

Before configuring the appropriate networking hardware, you should consult the manual pages in section 4 of the programmer's manual. The following table lists the devices for which software support exists.

Device name	Manufacturer and product
acc	ACC LH/DH interface to IMP
css	DEC IMP-11A interface to IMP
dmc	DEC DMC-11 (also works with DMR-11)
ec	3Com 10Mb/s Ethernet
en	Xerox 3Mb/s prototype Ethernet (not a product)
hy	NSC Hyperchannel, w/ DR-11B and PI-13 interfaces
il	Interlan 10Mb/s Ethernet
pcl	DEC PCL-11
un	Ungermann-Bass network w/ DR-11W interface
vv	Proteon ring network (V2LNI)

All network interface drivers require some or all of their host address be defined at boot time. This is accomplished with *ifconfig*(8C) commands included in the /etc/rc.local file. Interfaces which are able to dynamically deduce the host part of an address, but not the network number, take the network number from the address specified with *ifconfig*. Hosts which use a more complex address mapping scheme, such as the Address Resolution Protocol, *arp*(4), require the full address. The manual page for each network interface describes the method used to establish a host's address. *Ifconfig*(8) can also be used to set options for the interface at boot time. These options include disabling the use of the Address Resolution Protocol and/or the use of trailer encapsulation; this is useful if a network is shared with hosts running software which is unable to perform these functions. Options are set independently for each interface, and apply to all packets sent using that interface. An alternative approach to ARP is to divide the address range, using ARP only for those addresses below the cutoff and using another mapping above this constant address; see the source (/sys/netinet/if ether.c) for more information.

In order to use the pseudo terminals just configured, device entries must be created in the /dev directory. To create 16 pseudo terminals (plenty, unless you have a heavy network load) perform the following commands.

```
# cd /dev
# MAKEDEV pty0
```

More pseudo terminals may be made by specifying *pty1*, *pty2*, etc. The kernel normally includes support for 32 pseudo terminals unless the configuration file specifies a different number. Each pseudo terminal actually consists of two files in /dev: a master and a slave. The master pseudo terminal file is named /dev/pty?, while the slave side is /dev/ttyp?. Pseudo terminals are also used by the *script*(1) program. In addition to creating the pseudo terminals, be sure to install them in the */etc/ttys* file (with a '0' in the first column so no *getty* is started), and in the */etc/ttytype* file (with type "network").

When configuring multiple networks some thought must be given to the ordering of the devices in the configuration file. The first network interface configured in the system is used as the default network when the system is forced to assign a local address to a socket. This means that your most widely known network should always be placed first in the configuration file. For example, hosts attached to both the ARPANET and our local area network have devices configured in the order show below.

```
device      acc0    at uba? csr 0167600 vector accrint accxint
device      en0     at uba? csr 0161000 vector enxint enrint encollide
```

5.2. Network data bases

A number of data files are used by the network library routines and server programs. Most of these files are host independent and updated only rarely.

File	Manual reference	Use
/etc/hosts	*hosts*(5)	host names
/etc/networks	*networks*(5)	network names
/etc/services	*services*(5)	list of known services
/etc/protocols	*protocols*(5)	protocol names
/etc/hosts.equiv	*rshd*(8C)	list of "trusted" hosts
/etc/rc.local	*rc*(8)	command script for starting servers
/etc/ftpusers	*ftpd*(8C)	list of "unwelcome" ftp users

The files distributed are set up for ARPANET or other Internet hosts. Local networks and hosts should be added to describe the local configuration; the Berkeley entries may serve as examples (see also the next section). Network numbers will have to be chosen for each ethernet. For sites not connected to the Internet, these can be chosen more or less arbitrarily, otherwise the normal channels should be used for allocation of network numbers.

5.2.1. Regenerating /etc/hosts and /etc/networks

The host and network name data bases are normally derived from a file retrieved from the Internet Network Information Center at SRI. To do this you should use the program /etc/gettable to retrieve the NIC host data base, and the program /etc/htable to convert it to the format used by the libraries.

```
# cd /usr/src/ucb/netser/htable
# /etc/gettable sri-nic
Connection to sri-nic opened.
Host table received.
Connection to sri-nic closed.
# /etc/htable hosts.txt
Warning, no localgateways file.
#
```

The *htable* program generates two files of interest in the local directory: *hosts* and *networks*. If a file "localhosts" is present in the working directory its contents are first copied to the output file. Similarly, a "localnetworks" file may be prepended to the output created by *htable*. It is usually wise to run *diff*(1) on the new host and network data bases before installing them in /etc.

5.2.2. /etc/hosts.equiv

The remote login and shell servers use an authentication scheme based on trusted hosts. The hosts.equiv file contains a list of hosts which are considered trusted and/or, under a single administrative control. When a user contacts a remote login or shell server requesting service, the client process passes the user's name and the official name of the host on which the client is located. In the simple case, if the hosts's name is located in hosts.equiv and the user has an account on the server's machine, then service is rendered (i.e. the user is allowed to log in, or the command is executed). Users may constrain this "equivalence" of machines by installing a .rhosts file in their login directory. The root login is handled specially, bypassing the hosts.equiv file, and using only the /.rhosts file.

Thus, to create a class of equivalent machines, the hosts.equiv file should contain the *official* names for those machines. For example, most machines on our major local network are considered trusted, so the hosts.equiv file is of the form:

```
ucbarpa
ucbcalder
ucbdali
ucbernie
ucbkim
ucbmatisse
ucbmonet
ucbvax
ucbmiro
ucbdegas
```

5.2.3. /etc/rc.local

Most network servers are automatically started up at boot time by the command file /etc/rc (if they are installed in their presumed locations). These include the following:

/etc/rshd	shell server
/etc/rexecd	exec server
/etc/rlogind	login server
/etc/rwhod	system status daemon

To have other network servers started up as well, commands of the following sort should be placed in the site dependent file /etc/rc.local.

```
if [ -f /etc/telnetd ]; then
        /etc/telnetd & echo -n ' telnetd'              >/dev/console
fi
```

The following servers are included with the system and should be installed in /etc/rc.local as the need arises.

/etc/telnetd	TELNET server
/etc/ftpd	FTP server
/etc/tftpd	TFTP server
/etc/syslog	error logging server
/etc/sendmail	SMTP server
/etc/courierd	Courier remote procedure call server
/etc/routed	routing table management daemon

Consult the manual pages and accompanying documentation (particularly for sendmail) for details about their operation.

5.2.4. /etc/ftpusers

The FTP server included in the system provides support for an anonymous FTP account. Due to the inherent security problems with such a facility you should read this section carefully if you consider providing such a service.

An anonymous account is enabled by creating a user *ftp*. When a client uses the anonymous account a *chroot*(2) system call is performed by the server to restrict the client from moving outside that part of the file system where the user ftp home directory is located. Because a chroot call is used, certain programs and files must be supplied the server process for it to execute properly. Further, one must be sure that all directories and executable images are unwritable. The following directory setup is recommended.

```
# cd ~ftp
# chmod 555 .; chown ftp .; chgrp ftp .
# mkdir bin etc pub
# chown root bin etc
# chmod 555 bin etc
# chown ftp pub
# chmod 777 pub
# cd bin
# cp /bin/sh /bin/ls .
# chmod 111 sh ls
# cd ../etc
# cp /etc/passwd /etc/group .
# chmod 444 passwd group
```

When local users wish to place files in the anonymous area, they must be placed in a subdirectory. In the setup here, the directory ~ftp/pub is used.

Aside from the problems of directory modes and such, the ftp server may provide a loophole for interlopers if certain user accounts are allowed. The file /etc/ftpusers is checked on each connection. If the requested user name is located in the file, the request for service is denied. This file normally has the following names on our systems.

```
uucp
root
```

Accounts with nonstandard shells and no passwords (e.g., who or finger) should also be listed in this file to prevent their use as anonymous accounts with ftp.

5.3. Routing and gateways/bridges

If your environment allows access to networks not directly attached to your host you will need to set up routing information to allow packets to be properly routed. Two schemes are supported by the system. The first scheme employs the routing table management daemon /etc/routed to maintain the system routing tables. The routing daemon uses a variant of the Xerox Routing Information Protocol to maintain up to date routing tables in a cluster of local area networks. By using the /etc/gateways file created by /etc/htable, the routing daemon can also be used to initialize static routes to distant networks. When the routing daemon is started up (usually from /etc/rc.local) it reads /etc/gateways and installs those routes defined there, then broadcasts on each local network to which the host is attached to find other instances of the routing daemon. If any responses are received, the routing daemons cooperate in maintaining a globally consistent view of routing in the local environment. This view can be extended to include remote sites also running the routing daemon by setting up suitable entries in /etc/gateways; consult *routed*(8C) for a more thorough discussion.

The second approach is to define a wildcard route to a smart gateway and depend on the gateway to provide ICMP routing redirect information to dynamically create a routing data base. This is done by adding an entry of the form

/etc/route add 0 *smart-gateway* 1

to /etc/rc.local; see *route*(8C) for more information. The wildcard route, indicated by a 0 valued destination, will be used by the system as a "last resort" in routing packets to their destination. Assuming the gateway to which packets are directed is able to generate the proper routing redirect messages, the system will then add routing table entries based on the information supplied. This approach has certain advantages over the routing daemon, but is unsuitable in an environment where their are only bridges (i.e. pseudo gateways which, for instance, do not generate routing redirect messages). Further, if the smart gateway goes down there is no alternative, save manual alteration of the routing table entry, to maintaining service.

The system always listens, and processes, routing table redirect information, so it is possible to combine both the above facilities. For example, the routing table management process might be used to maintain up to date information about routes to geographically local networks, while employing the wildcard routing techniques for "distant" networks. The *netstat*(1) program may be used to display routing table contents as well as various routing oriented statistics. For example,

#netstat −r

will display the contents of the routing tables, while

#netstat −r −s

will show the number of routing table entries dynamically created as a result of routing redirect messages, etc.

6. SYSTEM OPERATION

This section describes procedures used to operate a VAX UNIX system. Procedures described here are used periodically, to reboot the system, analyze error messages from devices, do disk backups, monitor system performance, recompile system software and control local changes.

6.1. Bootstrap and shutdown procedures

In a normal reboot, the system checks the disks and comes up multi-user without intervention at the console. Such a reboot can be stopped (after it prints the date) with a ^C (interrupt). This will leave the system in single-user mode, with only the console terminal active.

If booting from the console command level is needed, then the command

```
>>> B
```

will boot from the default device. On an 11/780 (11/730) the default device is determined by a "DEPOSIT" command stored on the floppy (cassette) in the file "DEFBOO.CMD"; on an 11/750 the default device is determined by the setting of a switch on the front panel.

You can boot a system up single user on a 780 or 730 by doing

```
>>> B XXS
```

where *XX* is one of HP, HK, UP, RA, or RB for a 730. The corresponding command on an 11/750 is

```
>>> B/1
```

For second vendor storage modules on the UNIBUS or MASSBUS of an 11/750 you will need to have a boot prom. Most vendors will sell you such proms for their controllers; contact your vendor if you don't have one.

Other possibilities are:

```
>>> B ANY
```

or, on a 750

```
>>> B/3
```

These commands boot and ask for the name of the system to be booted. They can be used after building a new test system to give the boot program the name of the test version of the system.

To bring the system up to a multi-user configuration from the single-user status after, e.g., a "B HPS" on an 11/780, "B RBS" on a 730, or a "B/1" on an 11/750 all you have to do is hit ^D on the console. The system will then execute /etc/rc, a multi-user restart script (and /etc/rc.local), and come up on the terminals listed as active in the file /etc/ttys. See *init*(8) and *ttys*(5). Note, however, that this does not cause a file system check to be performed. Unless the system was taken down cleanly, you should run "fsck −p" or force a reboot with *reboot*(8) to have the disks checked.

To take the system down to a single user state you can use

```
# kill 1
```

or use the *shutdown*(8) command (which is much more polite, if there are other users logged in.) when you are up multi-user. Either command will kill all processes and give you a shell on the console, as if you had just booted. File systems remain mounted after the system is taken single-user. If you wish to come up multi-user again, you should do this by:

```
# cd /
# /etc/umount -a
# ^D
```

Each system shutdown, crash, processor halt and reboot is recorded in the file /usr/adm/shutdownlog with the cause.

6.2. Device errors and diagnostics

When errors occur on peripherals or in the system, the system prints a warning diagnostic on the console. These messages are collected regularly and written into a system error log file /usr/adm/messages.

Error messages printed by the devices in the system are described with the drivers for the devices in section 4 of the programmer's manual. If errors occur indicating hardware problems, you should contact your hardware support group or field service. It is a good idea to examine the error log file regularly (e.g. with "tail −r /usr/adm/messages").

6.3. File system checks, backups and disaster recovery

Periodically (say every week or so in the absence of any problems) and always (usually automatically) after a crash, all the file systems should be checked for consistency by *fsck*(1). The procedures of *reboot*(8) should be used to get the system to a state where a file system check can be performed manually or automatically.

Dumping of the file systems should be done regularly, since once the system is going it is easy to become complacent. Complete and incremental dumps are easily done with *dump*(8). You should arrange to do a towers-of-hanoi dump sequence; we tune ours so that almost all files are dumped on two tapes and kept for at least a week in most every case. We take full dumps every month (and keep these indefinitely). Operators can execute "dump w" at login that will tell them what needs to be dumped (based on the /etc/fstab information). Be sure to create a group **operator** in the file /etc/group so that dump can notify logged-in operators when it needs help.

More precisely, we have three sets of dump tapes: 10 daily tapes, 5 weekly sets of 2 tapes, and fresh sets of three tapes monthly. We do daily dumps circularly on the daily tapes with sequence '3 2 5 4 7 6 9 8 9 9 ...'. Each weekly is a level 1 and the daily dump sequence level restarts after each weekly dump. Full dumps are level 0 and the daily sequence restarts after each full dump also.

Thus a typical dump sequence would be:

tape name	level number	date	opr	size
FULL	0	Nov 24, 1979	jkf	137K
D1	3	Nov 28, 1979	jkf	29K
D2	2	Nov 29, 1979	rrh	34K
D3	5	Nov 30, 1979	rrh	19K
D4	4	Dec 1, 1979	rrh	22K
W1	1	Dec 2, 1979	etc	40K
D5	3	Dec 4, 1979	rrh	15K
D6	2	Dec 5, 1979	jkf	25K
D7	5	Dec 6, 1979	jkf	15K
D8	4	Dec 7, 1979	rrh	19K
W2	1	Dec 9, 1979	etc	118K
D9	3	Dec 11, 1979	rrh	15K
D10	2	Dec 12, 1979	rrh	26K
D1	5	Dec 15, 1979	rrh	14K
W3	1	Dec 17, 1979	etc	71K
D2	3	Dec 18, 1979	etc	13K

FULL 0 Dec 22, 1979 etc 135K

We do weekly's often enough that daily's always fit on one tape and never get to the sequence of 9's in the daily level numbers.

Dumping of files by name is best done by *tar*(1) but the amount of data that can be moved in this way is limited to a single tape. Finally if there are enough drives entire disks can be copied with *dd*(1) using the raw special files and an appropriate blocking factor; the number of sectors per track is usually a good value to use, consult /etc/disktab.

It is desirable that full dumps of the root file system be made regularly. This is especially true when only one disk is available. Then, if the root file system is damaged by a hardware or software failure, you can rebuild a workable disk doing a restore in the same way that the initial root file system was created.

Exhaustion of user-file space is certain to occur now and then; disk quotas may be imposed, or if you prefer a less facist approach, try using the programs *du*(1), *df*(1), *quot*(8), combined with threatening messages of the day, and personal letters.

6.4. Moving filesystem data

If you have the equipment, the best way to move a file system is to dump it to magtape using *dump*(8), use *newfs*(8) to create the new file system, and restore the tape, using *restore*(8). If for some reason you don't want to use magtape, dump accepts an argument telling where to put the dump; you might use another disk. The restore program uses an "in-place" algorithm which allows file system dumps to be restored without concern for the original size of the file system. Further, portions of a file system may be selectively restored in a manner similar to the tape archive program.

If you have to merge a file system into another, existing one, the best bet is to use *tar*(1). If you must shrink a file system, the best bet is to dump the original and restore it onto the new file system. If you are playing with the root file system and only have one drive, the procedure is more complicated. What you do is the following:

1. GET A SECOND PACK!!!!
2. Dump the root file system to tape using *dump*(8).
3. Bring the system down and mount the new pack.
4. Load the distribution tape and install the new root file system as you did when first installing the system.
5. Boot normally using the newly created disk file system.

Note that if you change the disk partition tables or add new disk drivers they should also be added to the standalone system in /sys/stand and the default disk partition tables in /etc/disktab should be modified.

6.5. Monitoring System Performance

The *vmstat* program provided with the system is designed to be an aid to monitoring systemwide activity. Together with the *ps*(1) command (as in "ps av"), it can be used to investigate systemwide virtual memory activity. By running *vmstat* when the system is active you can judge the system activity in several dimensions: job distribution, virtual memory load, paging and swapping activity, disk and cpu utilization. Ideally, there should be few blocked (b) jobs, there should be little paging or swapping activity, there should be available bandwidth on the disk devices (most single arms peak out at 30-35 tps in practice), and the user cpu utilization (us) should be high (above 60%).

If the system is busy, then the count of active jobs may be large, and several of these jobs may often be blocked (b). If the virtual memory is active, then the paging demon will be running (sr will be non-zero). It is healthy for the paging demon to free pages when the virtual memory gets active; it is triggered by the amount of free memory dropping below a

threshold and increases its pace as free memory goes to zero.

If you run *vmstat* when the system is busy (a "vmstat 1" gives all the numbers computed by the system), you can find imbalances by noting abnormal job distributions. If many processes are blocked (b), then the disk subsystem is overloaded or imbalanced. If you have a several non-dma devices or open teletype lines that are "ringing", or user programs that are doing high-speed non-buffered input/output, then the system time may go high (60-70% or higher). It is often possible to pin down the cause of high system time by looking to see if there is excessive context switching (cs), interrupt activity (in) or system call activity (sy). Cumulatively on one of our large machines we average about 60 context switches and interrupts per second and about 90 system calls per second.

If the system is heavily loaded, or if you have little memory for your load (1M is little in most any case), then the system may be forced to swap. This is likely to be accompanied by a noticeable reduction in system performance and pregnant pauses when interactive jobs such as editors swap out. If you expect to be in a memory-poor environment for an extended period you might consider administratively limiting system load.

6.6. Recompiling and reinstalling system software

It is easy to regenerate the system, and it is a good idea to try rebuilding pieces of the system to build confidence in the procedures. The system consists of two major parts: the kernel itself (/sys) and the user programs (/usr/src and subdirectories). The major part of this is /usr/src.

The three major libraries are the C library in /usr/src/lib/libc and the FORTRAN libraries /usr/src/usr.lib/libI77 and /usr/src/usr.lib/libF77. In each case the library is remade by changing into the corresponding directory and doing

```
# make
```

and then installed by

```
# make install
```

Similar to the system,

```
# make clean
```

cleans up.

The source for all other libraries is kept in subdirectories of /usr/src/usr.lib; each has a makefile and can be recompiled by the above recipe.

If you look at /usr/src/Makefile, you will see that you can recompile the entire system source with one command. To recompile a specific program, find out where the source resides with the *whereis*(1) command, then change to that directory and remake it with the makefile present in the directory. For instance, to recompile "date", all one has to do is

```
# whereis date
date: /usr/src/bin/date.c /bin/date /usr/man/man1/date.1
# cd /usr/src/bin
# make date
```

this will create an unstripped version of the binary of "date" in the current directory. To install the binary image, use the install command as in

```
# install -s date /bin/date
```

The -s option will insure the installed version of date has its symbol table stripped. The install command should be used instead of mv or cp as it understands how to install programs even when the program is currently in use.

If you wish to recompile and install all programs in a particular target area you can override the default target by doing:

```
# make
# make DESTDIR=pathname install
```

To regenerate all the system source you can do

```
# cd /usr/src
# make
```

If you modify the C library, say to change a system call, and want to rebuild and install everything from scratch you have to be a little careful. You must insure the libraries are installed before the remainder of the source, otherwise the loaded images will not contain the new routine from the library. The following steps are recommended.

```
# cd /usr/src
# cd lib; make install
# cd ..
# make usr.lib
# cd usr.lib; make install
# cd ..
# make bin etc usr.bin ucb games local
# for i in bin etc usr.bin ucb games local; do (cd $i; make install); done
```

This will take about 12 hours on a reasonably configured 11/750.

6.7. Making local modifications

To keep track of changes to system source we migrate changed versions of commands in /usr/src/bin, /usr/src/usr.bin, and /usr/src/ucb in through the directory /usr/src/new and out of the original directory into /usr/src/old for a time before removing them. Locally written commands that aren't distributed are kept in /usr/src/local and their binaries are kept in /usr/local. This allows /usr/bin, /usr/ucb, and /bin to correspond to the distribution tape (and to the manuals that people can buy). People wishing to use /usr/local commands are made aware that they aren't in the base manual. As manual updates incorporate these commands they are moved to /usr/ucb.

A directory /usr/junk to throw garbage into, as well as binary directories /usr/old and /usr/new are useful. The man command supports manual directories such as /usr/man/manj for junk and /usr/man/manl for local to make this or something similar practical.

6.8. Accounting

UNIX optionally records two kinds of accounting information: connect time accounting and process resource accounting. The connect time accounting information is stored in the file /usr/adm/wtmp, which is summarized by the program *ac*(8). The process time accounting information is stored in the file /usr/adm/acct, and analyzed and summarized by the program *sa*(8).

If you need to implement recharge for computing time, you can implement procedures based on the information provided by these commands. A convenient way to do this is to give commands to the clock daemon /etc/cron to be executed every day at a specified time. This is done by adding lines to /usr/adm/crontab; see *cron*(8) for details.

6.9. Resource control

Resource control in the current version of UNIX is fairly elaborate compared to most UNIX systems. The disc quota facilities developed at the University of Melbourne have been incorporated in the system and allow control over the number of files and amount of disc space each user may use on each file system. In addition, the resources consumed by any single process can be limited by the mechanisms of *setrlimit*(2). As distributed, the latter mechanism is voluntary, though sites may choose to modify the login mechanism to impose

limits not covered with disc quotas.

To utilize the disc quota facilities, the system must be configured with "options QUOTA". File systems may then be placed under the quota mechanism by creating a null file *quotas* at the root of the file system, running *quotacheck*(8), and modifying /etc/fstab to indicate the file system is read-write with disc quotas (a "rq" type field). The *quotaon*(8) program may then be run to enable quotas.

Individual quotas are applied by using the quota editor *edquota*(8). Users may view their quotas (but not those of other users) with the *quota*(1) program. The *repquota*(8) program may be used to summarize the quotas and current space usage on a particular file system or file systems.

Quotas are enforced with *soft* and *hard* limits. When a user first reaches a soft limit on a resource, a message is generated on his/her terminal. If the user fails to lower the resource usage below the soft limit the next time they log in to the system the *login* program will generate a warning about excessive usage. Should three login sessions go by with the soft limit breached the system then treats the soft limit as a *hard* limit and disallows any allocations until enough space is reclaimed to bring the user back below the soft limit. Hard limits are enforced strictly resulting in errors when a user tries to create or write a file. Each time a hard limit is exceeded the system will generate a message on the user's terminal.

Consult the auxiliary document, "Disc Quotas in a UNIX Environment" and the appropriate manual entries for more information.

6.10. Network troubleshooting

If you have anything more than a trivial network configuration, from time to time you are bound to run into problems. Before blaming the software, first check your network connections. On networks such as the Ethernet a loose cable tap or misplaced power cable can result in severely deteriorated service. The *netstat*(1) program may be of aid in tracking down hardware malfunctions. In particular, look at the −i and −s options in the manual page.

Should you believe a communication protocol problem exists, consult the protocol specifications and attempt to isolate the problem in a packet trace. The SO DEBUG option may be supplied before establishing a connection on a socket, in which case the system will trace all traffic and internal actions (such as timers expiring) in a circular trace buffer. This buffer may then be printed out with the *trpt*(8C) program. Most all the servers distributed with the system accept a −d option forcing all sockets to be created with debugging turned on. Consult the appropriate manual pages for more information.

6.11. Files which need periodic attention

We conclude the discussion of system operations by listing the files that require periodic attention or are system specific

/etc/fstab	how disk partitions are used
/etc/disktab	disk partition sizes
/etc/printcap	printer data base
/etc/gettytab	terminal type definitions
/etc/remote	names and phone numbers of remote machines for *tip*(1)
/etc/group	group memberships
/etc/motd	message of the day
/etc/passwd	password file; each account has a line
/etc/rc.local	local system restart script; runs reboot; starts daemons
/etc/hosts	host name data base
/etc/networks	network name data base
/etc/services	network services data base
/etc/hosts.equiv	hosts under same administrative control

/etc/securetty	restricted list of ttys where root can log in
/etc/ttys	enables/disables ports
/etc/ttytype	terminal types connected to ports
/usr/lib/crontab	commands that are run periodically
/usr/lib/aliases	mail forwarding and distribution groups
/usr/adm/acct	raw process account data
/usr/adm/messages	system error log
/usr/adm/shutdownlog	log of system reboots
/usr/adm/wtmp	login session accounting

APPENDIX A – BOOTSTRAP DETAILS

This appendix contains pertinent files and numbers regarding the bootstrapping procedure for 4.2BSD. You should never have to look at this appendix. However, if there are problems in installing the distribution on your machine, the material contained here may prove useful.

Contents of the distribution tapes

The distribution normally consists of two 1600bpi 2400' magnetic tapes. The first tape contains the following files on it. All tape files are blocked in 10 kilobytes records, except for the first file on the first tape which has 512 byte records.

Tape file	Records*	Contents
one	194	8 bootstrap monitor programs and a *tp*(1) file containing *boot*, *format*, and *copy*
two	205	"mini root" file system
three	380	*dump*(8) of distribution root file system
four	440	*tar*(1) image of /sys, including GENERIC system
five	2111	*tar*(1) image of binaries and libraries in /usr
six	576	*tar*(1) image of /usr/lib/vfont

The second tape contains the following files.

Tape file	# Records	Contents
one	2100	*tar*(1) image of /usr/src
two	973	*tar*(1) image of user contributed software
three	420	*tar*(1) image of /usr/ingres

The distribution tape is made with the shell scripts located in the directory /sys/dist. To construct a distribution tape one must first build a mini root file system with the *buildmini* shell script.

* The number of records in each tape file may not be precisely that shown in this table; these values reflect the contents of the distribution tape at the time this document was written.

```
#! /bin/sh
#     @(#)buildmini  4.4   7/9/83
#
miniroot=hp0g
minitype=rm80
#
date
umount /dev/${miniroot}
newfs -s 4096 ${miniroot} ${minitype}
fsck /dev/r${miniroot}
mount /dev/${miniroot} /mnt
cd /mnt; sh /sys/dist/get
cd /sys/dist; sync
umount /dev/${miniroot}
fsck /dev/${miniroot}
date
```

The *buildmini* script uses the *get* script to construct the actual file system.

```
#! /bin/sh
#     @(#)get   4.13  7/19/83
#
# Shell script to build a mini-root file system
# in preparation for building a distribution tape.
# The file system created here is image copied onto
# tape, then image copied onto disk as the "first"
# step in a cold boot of 4.2 systems.
#
DISTROOT=/nbsd
#
if [ `pwd` = '/' ]
then
        echo You just '(almost)' destroyed the root
        exit
fi
cp $DISTROOT/a/sys/GENERIC/vmunix .
rm -rf bin; mkdir bin
rm -rf etc; mkdir etc
rm -rf a; mkdir a
rm -rf tmp; mkdir tmp
rm -rf usr; mkdir usr usr/mdec
rm -rf sys; mkdir sys sys/floppy sys/cassette
cp $DISTROOT/etc/disktab etc
cp $DISTROOT/etc/newfs etc; strip etc/newfs
cp $DISTROOT/etc/mkfs etc; strip etc/mkfs
cp $DISTROOT/etc/restore etc; strip etc/restore
cp $DISTROOT/etc/init etc; strip etc/init
cp $DISTROOT/etc/mount etc; strip etc/mount
cp $DISTROOT/etc/mknod etc; strip etc/mknod
cp $DISTROOT/etc/fsck etc; strip etc/fsck
cp $DISTROOT/etc/umount etc; strip etc/umount
cp $DISTROOT/etc/arff etc; strip etc/arff
cp $DISTROOT/etc/flcopy etc; strip etc/flcopy
cp $DISTROOT/bin/mt bin; strip bin/mt
cp $DISTROOT/bin/ls bin; strip bin/ls
cp $DISTROOT/bin/sh bin; strip bin/sh
cp $DISTROOT/bin/mv bin; strip bin/mv
cp $DISTROOT/bin/sync bin; strip bin/sync
cp $DISTROOT/bin/cat bin; strip bin/cat
cp $DISTROOT/bin/mkdir bin; strip bin/mkdir
cp $DISTROOT/bin/stty bin; strip bin/stty; ln bin/stty bin/STTY
cp $DISTROOT/bin/echo bin; strip bin/echo
cp $DISTROOT/bin/rm bin; strip bin/rm
cp $DISTROOT/bin/cp bin; strip bin/cp
cp $DISTROOT/bin/expr bin; strip bin/expr
cp $DISTROOT/bin/awk bin; strip bin/awk
cp $DISTROOT/bin/make bin; strip bin/make
cp $DISTROOT/usr/mdec/* usr/mdec
cp $DISTROOT/a/sys/floppy/[Ma-z0-9]* sys/floppy
cp $DISTROOT/a/sys/cassette/[Ma-z0-9]* sys/cassette
cp $DISTROOT/a/sys/stand/boot boot
cp $DISTROOT/.profile .profile
cat >etc/passwd <<EOF
```

```
root::0:10::/:/bin/sh
EOF
cat >etc/group <<EOF
wheel:*:0:
staff:*:10:
EOF
cat >etc/fstab <<EOF
/dev/hp0a:/a:xx:1:1
/dev/up0a:/a:xx:1:1
/dev/hk0a:/a:xx:1:1
/dev/ra0a:/a:xx:1:1
/dev/rb0a:/a:xx:1:1
EOF
cat >xtr <<'EOF'
: ${disk?'Usage: disk=xx0 type=tt tape=yy xtr'}
: ${type?'Usage: disk=xx0 type=tt tape=yy xtr'}
: ${tape?'Usage: disk=xx0 type=tt tape=yy xtr'}
echo 'Build root file system'
newfs ${disk}a ${type}
sync
echo 'Check the file system'
fsck /dev/r${disk}a
mount /dev/${disk}a /a
cd /a
echo 'Rewind tape'
mt -t /dev/${tape}0 rew
echo 'Restore the dump image of the root'
restore rsf 3 /dev/${tape}0
cd /
sync
umount /dev/${disk}a
sync
fsck /dev/r${disk}a
echo 'Root filesystem extracted'
echo
echo 'If this is a 780, update floppy'
echo 'If this is a 730, update the cassette'
EOF
chmod +x xtr
rm -rf dev; mkdir dev
cp $DISTROOT/sys/dist/MAKEDEV dev
chmod +x dev/MAKEDEV
cp /dev/null dev/MAKEDEV.local
cd dev
cd ..
sync
```

The mini root file system must have enough space to hold the files found on a floppy or cassette.

Once a mini root file system is constructed, the *maketape* script is used to make a distribution tape.

```
#! /bin/sh
#     @(#)maketape  4.12  8/4/83
#
miniroot=hp0g
#
trap "rm -f /tmp/tape.$$; exit" 0 1 2 3 13 15
mt rew
date
umount /dev/hp2g /dev/hp2h
umount /dev/hp2a
mount -r /dev/hp2a /nbsd
mount -r /dev/hp2g /nbsd/usr
mount -r /dev/hp2h /nbsd/a
cd /nbsd/tp
tp cmf /tmp/tape.$$ boot copy format
cd /nbsd/sys/mdec
echo "Build 1st level boot block file"
cat tsboot htboot tmboot mtboot utboot noboot noboot /tmp/tape.$$ |\
      dd of=/dev/rmt12 bs=512 conv=sync
cd /nbsd
sync
echo "Add dump of mini-root file system"
dd if=/dev/r${miniroot} of=/dev/rmt12 bs=20b count=205 conv=sync
echo "Add full dump of real file system"
/etc/dump 0uf /dev/rmt12 /nbsd
echo "Add tar image of system sources"
cd /nbsd/a/sys; tar cf /dev/rmt12 .
echo "Add tar image of /usr"
cd /nbsd/usr; tar cf /dev/rmt12 adm bin dict doc games\
      guest hosts include lib local man mdec msgs new\
      old preserve pub spool tmp ucb
echo "Add varian fonts"
cd /usr/lib/vfont; tar cf /dev/rmt12 .
echo "Done, rewinding first tape"
mt rew
echo "Mount second tape and hit return when ready"; read x
echo "Add user source code"
cd /nbsd/usr/src; tar cf /dev/rmt12 .
echo "Add user contributed software"
cd /usr/src/new; tar cf /dev/rmt12 .
echo "Add ingres source"
cd /nbsd/usr/ingres; tar cf /dev/rmt12 .
echo "Done, rewinding second tape"
mt rew
```

Summarizing then, to construct a distribution tape you can use the above scripts and the following commands.

```
# buildmini
# maketape
...
Done, rewinding first tape
Mount second tape and hit return when ready
(remove the first tape and place a fresh one on the drive)
...
Done, rewinding second tape
```

Control status register addresses

The distribution uses many standalone device drivers which presume the location of a UNIBUS device's control status register (CSR). The following table summarizes these values.

Device name	Controller	CSR address (octal)
ra	DEC UDA50	0172150
rb	DEC 730 IDC	0175606
rk	DEC RK11	0177440
rl	DEC RL11	0174400
tm	EMULEX TC-11	0172520
ts	DEC TS11	0172520
up	EMULEX SC-21V	0176700
ut	SI 9700	0172440

All MASSBUS controllers are located at standard offsets from the base address of the MASSBUS adapter register bank.

Generic system control status register addresses

The *generic* version of the operating system supplied with the distribution contains the UNIBUS devices indicated below. These devices will be recognized if the appropriate control status registers respond at any of the indicated UNIBUS addresses.

Device name	Controller	CSR addresses (octal)
hk	DEC RK11	0177440
tm	EMULEX TC-11	0172520
ut	SI 9700	0172440
up	EMULEX SC-21V	0176700, 0174400, 0176300
ra	DEC UDA-50	0172150, 0172550, 0177550
rb	DEC 730 IDC	0175606
rl	DEC RL11	0174400
dn	DEC DN11	0160020
dm	DM11 equivalent	0170500
dh	DH11 equivalent	0160040
dz	DEC DZ11	0160100, 0160110, ... 0160170
ts	DEC TS11	0172520
dmf	DEC DMF32	0160340
lp	DEC LP11	0177514

If devices other than the above are located at any of the addresses indicated, the system may not bootstrap properly.

APPENDIX B – LOADING THE TAPE MONITOR

This section indicates how the bootstrap monitor located on the first tape of the distribution tape set may be loaded. This should not be necessary, but has been included as a fallback measure in case it is not possible to read the distributed console medium. **WARNING**: the bootstraps supplied below may not work, in certain instances on an 11/730 because they use a buffered data path for transferring data from tape to memory; consult our group if you are unable to load the monitor on an 11/730.

To load the tape bootstrap monitor, first mount the magnetic tape on drive 0 at load point, making sure that the write ring is not inserted. Temporarily set the reboot switch on an 11/780 or 11/730 to off; on an 11/750 set the power-on action to halt. (In normal operation an 11/780 or 11/730 will have the reboot switch on, and an 11/750 will have the power-on action set to boot/restart.)

If you have an 11/780 give the commands:

```
>>>HALT
>>>UNJAM
```

Then, on any machine, give the init command:

```
>>>I
```

and then key in at location 200 and execute either the TS, HT, TM, or MT bootstrap that follows, as appropriate. The machine's printouts are shown in boldface, explanatory comments are within ( ). (You can use 'delete' to delete a character and 'control U' to kill the whole line.)

TS bootstrap

```
>>>D/P 200 3AEFD0
>>>D + D05A0000
>>>D + 3BEF
>>>D + 800CA00
>>>D + 32EFC1
>>>D + CA010000
>>>D + EFC10804
>>>D + 24
>>>D + 15508F
>>>D + ABB45B00
>>>D + 2AB9502
>>>D + 8FB0FB18
```

```
>>>D + 956B024C
>>>D + FB1802AB
>>>D + 25C8FB0
>>>D + 6B
      (The next two deposits set up the addresses of the UNIBUS)
      (adapter and its memory; the 20006000 here is the address of)
      (the 11/780 uba0 and the 2013E000 the address of the 11/780 umem0)
>>>D + 20006000          (780 uba0)
      (780 uba1: 20008000, 750 uba: F30000, 730 uba: F26000)
>>>D + 2013E000          (780 umem0)
      (780 umem1: 2017E000, 750 umem: FFE000, 730 umem: FFE000)
>>>D + 80000000
>>>D + 254C004
>>>D + 80000
>>>D + 264
>>>D + E
>>>D + C001
>>>D + 2000000
>>>S 200
```

HT bootstrap

```
>>>D/P 200 3EEFD0
>>>D + C55A0000
>>>D + 3BEF
>>>D + 808F00
>>>D + C15B0000
>>>D + C05B5A5B
>>>D + 4008F
>>>D + D05B00
>>>D + 9D004AA
>>>D + C08F326B
>>>D + D424AB14
>>>D + 8FD00CAA
>>>D + 80000000
>>>D + 320800CA
>>>D + AAFE008F
>>>D + 6B39D010
>>>D + 0
      (The next two deposits set up the addresses of the MASSBUS)
      (adapter and the drive number for the tape formatter)
      (the 20012000 here is the address of the 11/780 mba1 and the 0)
      (reflects that the formatter is drive 0 on mba1)
>>>D + 20012000          (780 mba1) (780 mba0: 20010000, 750 mba0. F28000)
>>>D + 0                       (Formatter unit number in range 0-7)
>>>S 200
>>>S 200
```

TM bootstrap

```
>>>D/P 200 2AEFD0
>>>D + D0510000
>>>D + 2000008F
>>>D + 800C180
>>>D + 804C1D4
```

```
>>>D + 1AEFD0
>>>D + C8520000
>>>D + F5508F
>>>D + 8FAE5200
>>>D + 4A20200
>>>D + B006A2B4
>>>D + 2A203
      (The following two numbers are uba0 and umem0; see TS above)
      (for some hints on other values if your TM isn't on UBA0 on a 780)
>>>D + 20006000        (780 uba0)
>>>D + 2013E000        (780 umem0)
>>>S 200
>>>S 200
>>>S 200
```

MT bootstrap

```
>>>D/P 200 46EFD0
>>>D + C55A0000
>>>D + 43EF
>>>D + 808F00
>>>D + C15B0000
>>>D + C05B5A5B
>>>D + 4008F
>>>D + 19A5B00
>>>D + 49A04AA
>>>D + AAD408AB
>>>D + 8FD00C
>>>D + CA800000
>>>D + 8F320800
>>>D + 10AAFE00
>>>D + 2008F3C
>>>D + ABD014AB
>>>D + FE15044
>>>D + 399AF850
>>>D + 6B
      (The next two deposits set up the addresses of the MASSBUS)
      (adapter and the drive number for the tape formatter)
      (the 20012000 here is the address of the 11/780 mba1 and the 0)
      (reflects that the formatter is drive 0 on mba1)
>>>D + 20012000
>>>D + 0
>>>S 200
>>>S 200
>>>S 200
>>>S 200
```

(no toggle-in code exists for the UT device)

If the tape doesn't move the first time you start the bootstrap program with "S 200" you probably have entered the program incorrectly (but also check that the tape is online). Start over and check your typing. For the HT, MT, and TM bootstraps you will not be able to see the tape motion as you advance through the first few blocks as the tape motion is all within the vacuum columns.

Next, deposit in RA the address of the tape MBA/UBA and in RB the address of the device registers or unit number from one of:

```
>>>D/G A 20006000      (for tapes on 780 uba0)
>>>D/G A 20008000      (for tapes on 780 uba1)
>>>D/G A 20012000      (for tapes on 780 mba1)
>>>D/G A 20010000      (for tapes on 780 mba0)
>>>D/G A F30000        (for tapes on 750 uba0)
>>>D/G A F2A000        (for tapes on 750 mba1)
>>>D/G A F28000        (for tapes on 750 mba0)
>>>D/G A F26000        (for tapes on 730 uba0)
```

and for register B:

```
>>>D/G B 0            (for tm03/tm78 formatters at mba? drive 0)
>>>D/G B 1            (for tm03/tm78 formatters at mba? drive 1)
>>>D/G B 2013F550     (for tm11/ts11/tu80 tapes on 780 uba0)
>>>D/G B FFF550       (for tm11/ts11/tu80 tapes on 750 or 730 uba0)
```

Then start the bootstrap program with

```
>>>S 0
```

The console should type

```
=
```

You are now talking to the tape bootstrap monitor. At any point in the following procedure you can return to this section, reload the tape bootstrap, and restart the procedure. The console monitor is identical to that loaded from a TU58 console cassette, follow the instructions in section 2 as they apply to this device. The only exception is that when using programs loaded from the tape bootstrap monitor, programs will always return to the monitor (the "=" prompt). This saves your having to type in the above toggle-in code for each program to be loaded.

APPENDIX C – INSTALLATION TROUBLESHOOTING

This appendix lists and explains certain problems which might be encountered while trying to install the 4.2BSD distribution. The information provided here is limited to the early steps in the installation process; i.e. up to the point where the root file system is installed. If you have a problem installing the release consult this section for an indication of the problem before contacting our group.

Using the distribution console medium.

This section describes problems which may occur when using the programs provided on the distributed console medium: TU58 cassette or RX01 floppy disk.

program can not be loaded.

Check to make sure the correct floppy or cassette is being used. If using a floppy, be sure it is not in upside down. If using a cassette on an 11/730, be certain drive 0 is being used. If a hard i/o error occurred while reading a floppy, try reseting the console LSI-11 by powering it on and off. If you can not boot the cassette's bootstrap monitor, verify the standard DEC console cassette can be read; if it can not, your cassette is broken – not uncommon.

program halts without warning.

Check to make sure you have specified the correct disk to format; consult sections 1.3 and 1.4 for a discussion of the VAX and UNIX device naming conventions. On 11/750's, specifying a non-existent MASSBUS device will cause the program to halt as it receives an interrupt (standalone programs operate by polling devices).

If using a floppy, try reading the floppy under your current system. If this works, copy the floppy to a new one and begin again. If using a cassette on an 11/730, do likewise.

format prints "Known devices are ...".

You have requested *format* to work on a device for which it has no driver, or which does not exist; only the indicated devices are supported.

format, boot, or copy prints "unknown drive type".

A MASSBUS disk was specified, but the associated MASSBUS drive type register indicates a drive of unknown type. This probably means you typed something wrong or your hardware is incorrectly configured.

format, boot, or copy prints "unknown device".

The device specified is probably not one of those supported by the distribution; consult section 1.1. If the device is listed in section 1.1, the drive may be dual-ported, or for some other reason the driver was unable to decipher its characteristics. If this is a MASSBUS drive, try powering the MASSBUS adapter and/or controller on and off to clear the drive type register.

copy does not copy 205 records

If a tape read error occurred, clean your tape drive heads. If a disk write error occurred, the disk formatting may have failed. If the disk pack is removable, try another one. If you are currently running UNIX, you can reboot your old system and use *dd* to copy the mini-root file system into a disk partition (assuming the destination is not in use by the running system).

boot prints "not a directory"

The *boot* program was unable to find the requested program because it encountered something other than a directory while searching the file system. This usually indicates no file system is present on the disk partition supplied, or the file system has been corrupted. First check to make sure you typed the correct line to boot. If this is the case and you are booting off the mini-root file system, the mini-root was probably not copied correctly off the tape (perhaps it was not placed in the correct disk partition). Try reinstalling the mini-root file system or, if trying to boot the true root file system, try booting off the mini-root file system and run *fsck* on the restored root file system to insure its integrity. Finally, as a last resort, copy the *boot* program from the mini-root file system to the newly installed root file system.

boot prints "bad format"

The program you requested *boot* to load did not have a 407, 410, or 413 magic number in its header. This should never happen on a distribution system. If you were trying to boot off the root file system, reboot the system on the mini-root file system and look at the program on the root file system. Try copying the copy of vmunix on the mini-root to the root file system also.

boot prints "read short"

The file header for the program indicated a size larger than the actual size of the file located on disk. This is probably the result of file system corruption (or a disk i/o error). Try booting again or creating a new copy of the program to be loaded (see above).

Booting the generic system

This section contains common problems encountered when booting the generic version of the system.

system panics with "panic: iinit"

This occured because the system was unable to locate the program /etc/init. The root file system supplied at the "root device?" prompt was probably incorrect. Remember that when running on the mini-root file system, this question must be answered with something of the form "hp0*". If the answer had been "hp0", the system would have used the "a" partition on unit 0 of the "hp" drive, where presumably no file system exists.

Alternatively, the file system on which you were trying to run is corrupted, or simply missing /etc/init. Try reinstalling the appropriate file system or installing a version of init.

system selects incorrect root device

That is, you try to boot the system single user with "B/2" or "B xxS" but do not get the root file system in the expected location. This is most likely caused by your having many disks available more suited to be a root file system than the one you wanted. For example, if you have a "up" disk and an "hk" disk and install the system on the "hk", then try and boot the system to single-user mode, the heuristic used by the generic system to select the root file system will choose the "up" disk. The following list gives, in descending order, those disks thought most suitable to be a root file system: "hp", "up", "ra", "rb", "rl", "hk" (the position of "rl" is subject to argument). To get the root device you want you must boot using "B/3" or "B ANY", then supply the root device at the prompt.

system crashes during autoconfiguration

This is almost always caused by an unsupported UNIBUS device being present at a location where a supported device was expected. You must disable the device in some way, either by pulling it off the bus, or by moving the location of the console status register (consult

Appendix A for a complete list of UNIBUS csr's used in the generic system).

system does not find device(s)

The UNIBUS device is not at a standard location. Consult the list of control status register addresses in Appendix A, or wait to configure a system to your hardware.

Alternatively, certain devices are difficult to locate during autoconfiguration. A classic example is the TS11 tape drive which does not autoconfigure properly if it is rewinding when the system is rebooted. Tape drives should configure properly if they are off-line, or are not performing a tape movement. Disks which are dual-ported should autoconfigure properly if the drive is not being simultaneously accessed through the alternate port.

Building console cassettes

This sections describes common problems encountered while constructing a console bootstrap cassette.

system crashes

You are trying to build a cassette for an 11/750. On an 11/750 the system is booted by using a bootstrap prom and sector 0 of the root file system. Refer to section 2.1.5 or *tu*(4) for the appropriate reprimand.

system hangs

You are using an MRSP prom on an 11/750 and think you can ignore the instructions in this document. The problem here is that the generic system only supports the MRSP prom on an 11/730. Using it on an 11/750 requires a special system configuration; consult *tu*(4) for more information.

Building 4.2BSD UNIX† Systems with Config

June, 1983

Samuel J. Leffler

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

1. INTRODUCTION

Config is a tool used in building 4.2BSD system images. It takes a file describing a system's tunable parameters and hardware support, and generates a collection of files which are then used to build a copy of UNIX appropriate to that configuration. *Config* simplifies system maintenance by isolating system dependencies in a single, easy to understand, file.

This document describes the content and format of system configuration files and the rules which must be followed when creating these files. Example configuration files are constructed and discussed.

Later sections suggest guidelines to be used in modifying system source and explain some of the inner workings of the autoconfiguration process. Appendix D summarizes the rules used in calculating the most important system data structures and indicates some inherent system data structure size limitations (and how to go about modifying them).

---

†UNIX is a Trademark of Bell Laboratories.

2. CONFIGURATION FILE CONTENTS

A system configuration must include at least the following pieces of information:

- machine type
- cpu type
- system identification
- timezone
- maximum number of users
- location of the root file system
- available hardware

Config allows multiple system images to be generated from a single configuration description. Each system image is configured for identical hardware, but may have different locations for the root file system and, possibly, other system devices.

2.1. Machine type

The *machine type* indicates if the system is going to operate on a DEC VAX-11 computer, or some other machine on which 4.2BSD operates. The machine type is used to locate certain data files which are machine specific and, also, to select rules used in constructing the resultant configuration files.

2.2. Cpu type

The *cpu type* indicates which, of possibly many, cpu's the system is to operate on. For example, if the system is being configured for a VAX-11, it could be running on a VAX-11/780, VAX-11/750, or VAX-11/730. Specifying more than one cpu type implies the system should be configured to run on all the cpu's specified. For some types of machines this is not possible and *config* will print a diagnostic indicating such.

2.3. System identification

The *system identification* is a moniker attached to the system, and often the machine on which the system is to run. For example, at Berkeley we have machines named Ernie (Co-VAX), Kim (No-VAX), and so on. The system identifier selected is used to create a global C "#define" which may be used to isolate system dependent pieces of code in the kernel. For example, Ernie's Varian driver used to be special cased because its interrupt vectors were wired together. The code in the driver which understood how to handle this non-standard hardware configuration was conditionally compiled in only if the system was for Ernie.

The system identifier "GENERIC" is given to a system which will run on any cpu of a particular machine type; it should not otherwise be used for a system identifier.

2.4. Timezone

The timezone in which the system is to run is used to define the information returned by the *gettimeofday*(2) system call. This value is specified as the number of hours east or west of GMT. Negative numbers indicate a value east of GMT. The timezone specification may also indicate the type of daylight savings time rules to be applied.

2.5. Maximum number of users

The system allocates many system data structures at boot time based on the maximum number of users the system will support. This number is normally between 8 and 40, depending on the hardware and expected job mix. The rules used to calculate system data structures are discussed in Appendix D.

2.6. Root file system location

When the system boots it must know the location of the root of the file system tree. This location and the part(s) of the disk(s) to be used for paging and swapping must be specified in order to create a complete configuration description. *Config* uses many rules to calculate default locations for these items; these are described in Appendix B.

When a generic system is configured, the root file system is left undefined until the system is booted. In this case, the root file system need not be specified, only that the system is a generic system.

2.7. Hardware devices

When the system boots it goes through an *autoconfiguration* phase. During this period, the system searches for all those hardware devices which the system builder has indicated might be present. This probing sequence requires certain pieces of information such as register addresses, bus interconnects, etc. A system's hardware may be configured in a very flexible manner or be specified without any flexibility whatsoever. Most people do not configure hardware devices into the system unless they are currently present on the machine, expect them to be present in the near future, or are simply guarding against a hardware failure somewhere else at the site (it is often wise to configure in extra disks in case an emergency requires moving one off a machine which has hardware problems).

The specification of hardware devices usually occupies the majority of the configuration file. As such, a large portion of this document will be spent understanding it. Section 6.3 contains a description of the autoconfiguration process, as it applies to those planning to write, or modify existing, device drivers.

2.8. Optional items

Other than the mandatory pieces of information described above, it is also possible to include various optional system facilities. For example, 4.2BSD can be configured to support binary compatibility for programs built under 4.1BSD. Also, optional support is provided for disk quotas and tracing the performance of the virtual memory subsystem. Any optional facilities to be configured into the system are specified in the configuration file. The resultant files generated by *config* will automatically include the necessary pieces of the system.

3. SYSTEM BUILDING PROCESS

In this section we consider the steps necessary to build a bootable system image. We assume the system source is located in the "/sys" directory and that, initially, the system is being configured from source code.

Under normal circumstances there are 5 steps in building a system.

1) Create a configuration file for the system.
2) Make a directory for the system to be constructed in.
3) Run *config* on the configuration file to generate the files required to compile and load the system image.
4) Construct the source code interdependency rules for the configured system.
5) Compile and load the system with *make*(1).

Steps 1 and 2 are usually done only once. When a system configuration changes it usually suffices to just run *config* on the modified configuration file, rebuild the source code dependencies, and remake the system. Sometimes, however, configuration dependencies may not be noticed in which case it is necessary to clean out the relocatable object files saved in the system's directory; this will be discussed later.

3.1. Creating a configuration file

Configuration files normally reside in the directory "/sys/conf". A configuration file is most easily constructed by copying an existing configuration file and modifying it. The 4.2BSD distribution contains a number of configuration files for machines at Berkeley, one may be suitable or, in worst case, you may take the generic configuration file and edit that.

The configuration file must have the same name as the directory in which the configured system is to be built. Further, *config* assumes this directory is located in the parent directory of the directory in which it is run. For example, the generic system has a configuration file "/sys/conf/GENERIC" and an accompanying directory named "/sys/GENERIC". In general it is unwise to move your configuration directories out of "/sys" as most of the system code and the files created by *config* use pathnames of the form "../". If you are running out of space on the file system where the configuration directories are located there is a mechanism for sharing relocatable object files between systems; this is described later.

When building your configuration file, be sure to include the items described in section 2. In particular, the machine type, cpu type, timezone, system identifier, maximum users, and root device must be specified. The specification of the hardware present may take a bit of work; particularly if your hardware is configured at non-standard places (e.g. device registers located at funny places or devices not supported by the system). Section 4 of this document gives a detailed description of the configuration file syntax, section 5 explains some sample configuration files, and section 6 discusses how to add new devices to the system. If the devices to be configured are not already described in one of the existing configuration files you should check the manual pages in section 4 of the UNIX Programmers Manual. For each supported device, the manual page synopsis entry gives a sample configuration line.

Once the configuration file is complete, run it through *config* and look for any errors. Never try and use a system which *config* has complained about; the results are unpredictable. For the most part, *config*'s error diagnostics are self explanatory. It may be the case that the line numbers given with the error messages are off by one.

A successful run of *config* on your configuration file will generate a number of files in the configuration directory. These files are:

- A file to be used by *make*(1) in compiling and loading the system.

- One file for each possible system image for your machine which describes where swapping, the root file system, and other miscellaneous system devices are located.
- A collection of header files, one per possible device the system supports, which define the hardware configured.
- A file containing the i/o configuration tables used by the system during its *autoconfiguration* phase.
- An assembly language file of interrupt vectors which connect interrupts from your machine's external buses to the main system path for handling interrupts.

Unless you have reason to doubt *config*, or are curious how the system's autoconfiguration scheme works, you should never have to look at any of these files.

3.2. Constructing source code dependencies

When *config* is done generating the files needed to compile and link your system it will terminate with a message of the form "Don't forget to run make depend". This is a reminder that you should change over to the configuration directory for the system just configured and type "make depend" to build the rules used by *make* to recognize interdependencies in the system source code. This will insure that any changes to a piece of the system source code will result in the proper modules being recompiled the next time *make* is run.

This step is particularly important if your site makes changes to the system include files. The rules generated specify which source code files are dependent on which include files. Without these rules, *make* will not recognize when it must rebuild modules due to a system header file being modified. Note that dependency rules created by this step only reflect directly included files. That is, if file "a" includes another file "b", which includes yet another, say "c", and then "c" is modified, *make* will not recognize that "a" should be recompiled. It is best to keep include file dependencies only one level deep.

3.3. Building the system

The makefile constructed by *config* should allow a new system to be rebuilt by simply typing "make image-name". For example, if you have named your bootable system image "vmunix", then "make vmunix" will generate a bootable image named "vmunix". Alternate system image names are used when the root file system location and/or swapping configuration is done in more than one way. The makefile which *config* creates has entry points for each system image defined in the configuration file. Thus, if you have configured "vmunix" to be a system with the root file system on an "hp" device and "hkvmunix" to be a system with the root file system on an "hk" device, then "make vmunix hkvmunix" will generate binary images for each.

Note that the name of a bootable image is different from the system identifier. All bootable images are configured for the same system; only the information about the root file system and paging devices differ. (This is described in more detail in section 4.)

The last step in the system building process is to rearrange certain commonly used symbols in the symbol table of the system image; the makefile generated by *config* does this automatically for you. This is advantageous for programs such as *ps*(1) and *vmstat*(1), which run much faster when the symbols they need are located at the front of the symbol table. Remember also that many programs expect the currently executing system to be named "/vmunix". If you install a new system and name it something other than "/vmunix", many programs are likely to give strange results.

3.4. Sharing object modules

If you have many systems which are all built on a single machine there are at least two approaches to saving time in building system images. The best way is to have a single system image which is run on all machines. This is attractive since it minimizes disk space used and time required to rebuild systems after making changes. However, it is often the case that one

or more systems will require a separately configured system image. This may be due to limited memory (building a system with many unused device drivers can be expensive), or to configuration requirements (one machine may be a development machine where disk quotas are not needed, while another is a production machine where they are), etc. In these cases it is possible for common systems to share relocatable object modules which are not configuration dependent; most of the module in the directory "/sys/sys" are of this sort.

To share object modules, a generic system should be built. Then, for each system configure the system as before, but before recompiling and linking the system, type "make links". This will cause the system to be searched for source modules which are safe to share between systems and generate symbolic links in the current directory to the appropriate object modules in the directory "../GENERIC". A shell script, "makelinks" is generated with this request and may be checked for correctness. The file "/sys/conf/defines" contains a list of symbols which we believe are safe to ignore when checking the source code for modules which may be shared. Note that this list includes the definitions used to conditionally compile in the virtual memory tracing facilities, and the trace point support used only rarely (even at Berkeley). It may be necessary to modify this file to reflect local needs. Note further, that as described previously, interdependencies which are not directly visible in the source code are not caught. This means that if you place per-system dependencies in an include file, they will not be recognized and the shared code may be selected in an unexpected fashion.

3.5. Building profiled systems

It is simple to configure a system which will automatically collect profiling information as it operates. The profiling data may be collected with *kgmon*(8) and processed with *gprof*(1) to obtain information regarding the system's operation. Profiled systems maintain histograms of the program counter as well as the number of invocations of each routine. The *gprof*(1) command will also generate a dynamic call graph of the executing system and propagate time spent in each routine along the arcs of the call graph (consult the gprof documentation for elaboration). The program counter sampling can be driven by the system clock, or if you have an alternate real time clock this can be used. The latter is highly recommended as use of the system clock will result in statistical anomalies and time spent in the clock routine will not be accurately accounted for.

To configure a profiled system, the **-p** option should be supplied to *config*. A profiled system is about 5-10% larger in its text space due to the calls to count the subroutine invocations. When the system executes, the profiling data is stored in a buffer which is 1.2 times the size of the text space. The overhead for running a profiled system varies; under normal load we see anywhere from 5-25% of the system time spent in the profiling code.

Note that systems configured for profiling should not be shared as described above unless all the other shared systems are also to be profiled.

4. CONFIGURATION FILE SYNTAX

In this section we consider the specific rules used in writing a configuration file. A complete grammar for the input language can be found in Appendix A and may be of use if you should have problems with syntax errors.

A configuration file is broken up into three logical pieces:

- configuration parameters global to all system images specified in the configuration file,
- parameters specific to each system image to be generated, and
- device specifications.

4.1. Global configuration parameters

The global configuration parameters are the type of machine, cpu types, options, timezone, system identifier, and maximum users. Each is specified with a separate line in the configuration file.

machine *type*
The system is to run on the machine type specified. No more than one machine type can appear in the configuration file. Legal values are **vax** and **sun**.

cpu "*type*"
This system is to run on the cpu type specified. More than one cpu type specification can appear in a configuration file. Legal types for a **vax** machine are **VAX780**,**VAX750**, and **VAX730**.

options *optionlist*
Compile the listed optional code into the system. Options in this list are separated by commas. Possible options are listed at the top of the generic makefile. A line of the form "options FUNNY,HAHA" generates global "#define"s -DFUNNY -DHAHA in the resultant makefile. An option may be given a value by following its name with "=", then the value enclosed in (double) quotes. None of the standard options use such a value. The following options are currently in use: COMPAT (include code for compatiblity with 4.1BSD binaries), INET (Internet communication protocols), PUP (support for a PUP raw interface), and QUOTA (enable disk quotas). There are additional options which are associated with certain peripheral devices; those are listed in the Synopsis section of the manual page for the device.

timezone *number* [**dst** [*number*]]
Specifies the timezone you are in. This is measured in the number of hours your timezone is west of GMT. EST is 5 hours west of GMT, PST is 8. Negative numbers indicate hours east of GMT. If you specify **dst**, the system will operate under daylight savings time. An optional integer or floating point number may be included to specify a particular daylight saving time correction algorithm; the default value is 1, indicating the United States. Other values are: 2 (Australian style), 3 (Western European), 4 (Middle European), and 5 (Eastern European). See *gettimeofday*(2) and *ctime*(3) for more information.

ident *name*
This system is to be known as *name*. This is usually a cute name like ERNIE (short for Ernie Co-Vax) or VAXWELL (for Vaxwell Smart).

maxusers *number*
The maximum expected number of simultaneously active user on this system is *number*. This number is used to size several system data structures.

4.2. System image parameters

Multiple bootable images may be specified in a single configuration file. The systems will have the same global configuration parameters and devices, but the location of the root file system and other system specific devices may be different. A system image is specified with a "config" line:

config *sysname config-clauses*

The *sysname* field is the name given to the loaded system image; almost everyone names their standard system image "vmunix". The configuration clauses are one or more specifications indicating where the root file system is located, how many paging devices there are and where they go. The device used by the system to process argument lists during *execve*(2) calls may also be specified, though in practice this is almost always selected by *config* using one of its rules for selecting default locations for system devices.

A configuration clause is one of the following

root [**on**] *root-device*
swap [**on**] *swap-device* [**and** *swap-device*]
dumps [**on**] *dump-device*
args [**on**] *arg-device*

(the "on" is optional.) Multiple configuration clauses are separated by white space; *config* allows specifications to be continued across multiple lines by beginning the continuation line with a tab character. The "root" clause specifies where the root file system is located, the "swap" clause indicates swapping and paging area(s), the "dumps" clause can be used to force system dumps to be taken on a particular device, and the "args" clause can be used to specify that argument list processing for *execve* should be done on a particular disk.

The device names supplied in the clauses may be fully specified as a device, unit, and file system partition; or underspecified in which case *config* will use builtin rules to select default unit numbers and file system partitions. The defaulting rules are a bit complicated as they are dependent on the overall system configuration. For example, the swap area need not be specified at all if the root device is specified; in this case the swap area is placed in the "b" partition of the same disk where the root file system is located. Appendix B contains a complete list of the defaulting rules used in selecting system configuration devices.

The device names are translated to the appropriate major and minor device numbers on a per-machine basis. A file, "/sys/conf/devices.machine" (where "machine" is the machine type specified in the configuration file), is used to map a device name to its major block device number. The minor device number is calculated using the standard disk partitioning rules: on unit 0, partition "a" is minor device 0, partition "b" is minor device 1, and so on; for units other than 0, add 8 times the unit number to get the minor device.

If the default mapping of device name to major/minor device number is incorrect for your configuration, it can be replaced by an explicit specification of the major/minor device. This is done by substituting

major *x* **minor** *y*

where the device name would normally be found. For example,

config vmunix **root on major** 99 **minor** 1

Normally, the areas configured for swap space are sized by the system at boot time. If a non-standard partition size is to be used for one or more swap areas, this can also be specified. To do this, the device name specified for a swap area should have a "size" specification appended. For example,

config vmunix **root on** hp0 **swap on** hp0b **size** 1200

would force swapping to be done in partition "b" of "hp0" and the swap partition size would be set to 1200 sectors. A swap area sized larger than the associated disk partition is trimmed to the partition size.

To create a generic configuration, only the clause "swap generic" should be specified; any extra clauses will cause an error.

4.3. Device specifications

Each device attached to a machine must be specified to *config* so that the system generated will know to probe for it during the autoconfiguration process carried out at boot time. Hardware specified in the configuration need not actually be present on the machine where the generated system is to be run. Only the hardware actually found at boot time will be used by the system.

The specification of hardware devices in the configuration file parallels the interconnection hierarchy of the machine to be configured. On the VAX, this means a configuration file must indicate what MASSBUS and UNIBUS adapters are present, and to which *nexi* they might be connected*. Similarly, devices and controllers must be indicated as possibly being connected to one or more adapters. A device description may provide a complete definition of the possible configuration parameters or it may leave certain parameters undefined and make the system probe for all the possible values. The latter allows a single device configuration list to match many possible physical configurations. For example, a disk may be indicated as present at UNIBUS adapter 0, or at any UNIBUS adapter which the system locates at boot time. The latter scheme, termed *wildcarding*, allows more flexibility in the physical configuration of a system; if a disk must be moved around for some reason, the system will still locate it at the alternate location.

A device specification takes one of the following forms:

master *device-name device-info*
controller *device-name device-info* [*interrupt-spec*]
device *device-name device-info interrupt-spec*
disk *device-name device-info*
tape *device-name device-info*

A "master" is a MASSBUS tape controller; a "controller" is a disk controller, a UNIBUS tape controller, a MASSBUS adapter, or a UNIBUS adapter. A "device" is an autonomous device which connects directly to a UNIBUS adapter (as opposed to something like a disk which connects through a disk controller). "Disk" and "tape" identify disk drives and tape drives connected to a "controller" or "master".

The *device-name* is one of the standard device names, as indicated in section 4 of the UNIX Programmers Manual, concatenated with the *logical* unit number to be assigned the device (the *logical* unit number may be different than the *physical* unit number indicated on the front of something like a disk; the *logical* unit number is used to refer to the UNIX device, not the physical unit number). For example, "hp0" is logical unit 0 of a MASSBUS storage device, even though it might be physical unit 3 on MASSBUS adapter 1.

The *device-info* clause specifies how the hardware is connected in the interconnection hierarchy. On the VAX, UNIBUS and MASSBUS adapters are connected to the internal system bus through a *nexus*. Thus, one of the following specifications would be used:

controller mba0 **at nexus** *x*
controller uba0 **at nexus** *x*

To tie a controller to a specific nexus, "x" would be supplied as the number of that nexus; otherwise "x" may be specified as "?", in which case the system will probe all nexi present looking for the specified controller.

The remaining interconnections on the VAX are:

* While VAX-11/750's and VAX-11/730 do not actually have nexi, the system treats them as having *simulated nexi* to simplify device configuration.

- a controller may be connected to another controller (e.g. a disk controller attached to a UNIBUS adapter),
- a master is always attached to a controller (a MASSBUS adaptor),
- a tape is always attached to a master (for MASSBUS tape drives),
- a disk is always attached to a controller, and
- devices are always attached to controllers (e.g. UNIBUS controllers attached to UNIBUS adapters).

The following lines give an example of each of these interconnections:

```
controller      hk0     at uba0 ...
master          ht0     at mba0 ...
tape            tu0     at ht0 ...
disk            rk1     at hk0 ...
device          dz0     at uba0 ...
```

Any piece of hardware which may be connected to a specific controller may also be wildcarded across multiple controllers.

The final piece of information needed by the system to configure devices is some indication of where or how a device will interrupt. For tapes and disks, simply specifying the *slave* or *drive* number is sufficient to locate the control status register for the device. For controllers, the control status register must be given explicitly, as well how many interrupt vectors are used and the names of the routines to which they should be bound. Thus the example lines given above might be completed as:

```
controller      hk0     at uba0 csr 0177440     vector rkintr
master          ht0     at mba0 drive 0
tape            tu0     at ht0 slave 0
disk            rk1     at hk0 drive 1
device          dz0     at uba0 csr 0160100     vector dzrint dzxint
```

Certain device drivers require extra information passed to them at boot time to tailor their operation to the actual hardware present. The line printer driver, for example, needs to know how many columns are present on each non-standard line printer (i.e. a line printer with other than 80 columns). The drivers for the terminal multiplexors need to know which lines are attached to modem lines so that no one will be allowed to use them unless a connection is present. For this reason, one last parameter may be specified to a *device*, a *flags* field. It has the syntax

flags *number*

and is usually placed after the *csr* specification. The *number* is passed directly to the associated driver. The manual pages in section 4 should be consulted to determine how each driver uses this value (if at all). Communications interface drivers commonly use the flags to indicate whether modem control signals are in use.

The exact syntax for each specific device is given in the Synopsis section of its manual page in section 4 of the manual.

4.4. Pseudo-devices

A number of drivers and software subsystems are treated like device drivers without any associated hardware. To include any of these pieces, a "pseudo-device" specification must be used. A specification for a pseudo device takes the form

pseudo-device *device-name* [*howmany*]

Examples of pseudo devices are **bk**, the Berknet line discipline, **pty**, the pseudo terminal driver (where the optional *howmany* value indicates the number of pseudo terminals to configure, 32 default), and **inet**, the DARPA Internet protocols (one must also specify INET in the "options"). Other pseudo devices for the network include **loop**, the software loopback

interface, **imp** (required when a CSS or ACC imp is configured), and **ether** (used by the Address Resolution Protocol on 10 Mb/sec ethernets). More information on configuring each of these can also be found in section 4 of the manual.

5. SAMPLE CONFIGURATION FILES

In this section we will consider how to configure a sample VAX-11/780 system on which the hardware can be reconfigured to guard against various hardware mishaps. We then study the rules needed to configure a VAX-11/750 to run in a networking environment.

5.1. VAX-11/780 System

Our VAX-11/780 is configured with hardware recommended in the document "Hints on Configuring a VAX for 4.2BSD" (this is one of the high-end configurations). Table 1 lists the pertinent hardware to be configured.

Item	Vendor	Connection	Name	Reference
cpu	DEC		VAX780	
MASSBUS controller	Emulex	nexus ?	mba0	hp(4)
disk	Fujitsu	mba0	hp0	
disk	Fujitsu	mba0	hp1	
MASSBUS controller	Emulex	nexus ?	mba1	
disk	Fujitsu	mba1	hp2	
disk	Fujitsu	mba1	hp3	
UNIBUS adapter	DEC	nexus ?		
tape controller	Emulex	uba0	tm0	tm(4)
tape drive	Kennedy	tm0	te0	
tape drive	Kennedy	tm0	te1	
terminal multiplexor	Emulex	uba0	dh0	dh(4)
terminal multiplexor	Emulex	uba0	dh1	
terminal multiplexor	Emulex	uba0	dh2	

Table 1. VAX-11/780 Hardware support.

We will call this machine ANSEL and construct a configuration file one step at a time.

The first step is to fill in the global configuration parameters. The machine is a VAX, so the *machine type* is "vax". We will assume this system will run only on this one processor, so the *cpu type* is "VAX780". The options are empty since this is going to be a "vanilla" VAX. The system identifier, as mentioned before, is "ANSEL" and the maximum number of users we plan to support is about 40. Thus the beginning of the configuration file looks like this:

```
#
# ANSEL VAX (a picture perfect machine)
#
machine         vax
cpu             VAX780
timezone        8 dst
ident           ANSEL
maxusers        40
```

To this we must then add the specifications for three system images. The first will be our standard system with the root on "hp0" and swapping on the same drive as the root. The second will have the root file system in the same location, but swap space interleaved among drives on each controller. Finally, the third will be a generic system, to allow us to boot off any of the four disk drives.

```
config          vmunix      root on hp0
config          hpvmunix    root on hp0 swap on hp0 and hp2
config          genvmunix   swap generic
```

Finally, the hardware must be specified. Let us first just try transcribing the information from Table 1.

```
controller      mba0        at nexus ?
disk            hp0         at mba0 disk 0
disk            hp1         at mba0 disk 1
controller      mba1        at nexus ?
disk            hp2         at mba1 disk 2
disk            hp3         at mba1 disk 3
controller      uba0        at nexus ?
controller      tm0         at uba0 csr 0172520     vector tmintr
tape            te0         at tm0 drive 0
tape            te1         at tm0 drive 1
device          dh0         at uba0 csr 0160020     vector dhrint dhxint
device          dm0         at uba0 csr 0170500     vector dmintr
device          dh1         at uba0 csr 0160040     vector dhrint dhxint
device          dh2         at uba0 csr 0160060     vector dhrint dhxint
```

(Oh, I forgot to mention one panel of the terminal multiplexor has modem control, thus the "dm0" device.)

This will suffice, but leaves us with little flexibility. Suppose our first disk controller were to break. We would like to recable the drives normally on the second controller so that all our disks could still be used without reconfiguring the system. To do this we wildcard the MASSBUS adapter connections and also the slave numbers. Further, we wildcard the UNIBUS adapter connections in case we decide some time in the future to purchase another adapter to offload the single UNIBUS we currently have. The revised device specifications would then be:

```
controller      mba0        at nexus ?
disk            hp0         at mba? disk ?
disk            hp1         at mba? disk ?
controller      mba1        at nexus ?
disk            hp2         at mba? disk ?
disk            hp3         at mba? disk ?
controller      uba0        at nexus ?
controller      tm0         at uba? csr 0172520     vector tmintr
tape            te0         at tm0 drive 0
tape            te1         at tm0 drive 1
device          dh0         at uba? csr 0160020     vector dhrint dhxint
device          dm0         at uba? csr 0170500     vector dmintr
device          dh1         at uba? csr 0160040     vector dhrint dhxint
device          dh2         at uba? csr 0160060     vector dhrint dhxint
```

The completed configuration file for ANSEL is shown in Appendix C.

5.2. VAX-11/750 with network support

Our VAX-11/750 system will be located on two 10Mb/s Ethernet local area networks and also the DARPA Internet. The system will have a MASSBUS drive for the root file system and two UNIBUS drives. Paging is interleaved among all three drives. We have sold our standard DEC terminal multiplexors since this machine will be accessed solely through the network. This machine is not intended to have a large user community, it does not have a

great deal of memory. First the global parameters:

```
#
# UCBVAX (Gateway to the world)
#
machine         vax
cpu             "VAX780"
cpu             "VAX750"
ident           UCBVAX
timezone        8 dst
maxusers        32
options         INET
```

The multiple cpu types allow us to replace UCBVAX with a more powerful cpu without reconfiguring the system. The value of 32 given for the maximum number of users is done to force the system data structures to be over-allocated. That is desirable on this machine because, while it is not expected to support many users, it is expected to perform a great deal of work. Upping this value results in a larger disk buffer cache than would normally be allocated if the true number of users were given. The "INET" indicates we plan to use the DARPA standard Internet protocols on this machine.

The system images and disks are configured in next.

```
config          vmunix      root on hp swap on hp and rk0 and rk1
config          upvmunix    root on up
config          hkvmunix    root on hk swap on rk0 and rk1

controller      mba0        at nexus ?
controller      uba0        at nexus ?
disk            hp0         at mba? drive 0
disk            hp1         at mba? drive 1
controller      sc0         at uba? csr 0176700     vector upintr
disk            up0         at sc0 drive 0
disk            up1         at sc0 drive 1
controller      hk0         at uba? csr 0177440     vector rkintr
disk            rk0         at hk0 drive 0
disk            rk1         at hk0 drive 1
```

UCBVAX requires heavy interleaving of its paging area to keep up with all the mail traffic it handles. The limiting factor on this system's performance is usually the number of disk arms, as opposed to memory or cpu cycles. The extra UNIBUS controller, "sc0", is in case the MASSBUS controller breaks and a spare controller must be installed (most of our old UNIBUS controllers have been replaced with the newer MASSBUS controllers, so we have a number of these around as spares).

Finally, we add in the network support. The Internet protocols require an "inet" *pseudo-device* in addition to the global "INET" option specified above. Pseudo terminals are needed to allow users to log in across the network (remember the only hardwired terminal is the console). The connection to the Internet is through an IMP, this requires yet another *pseudo-device* (in addition to the actual hardware device used by the IMP software). And, finally, there are the two Ethernet devices. These use a special protocol, the Address Resolution Protocol (ARP), to map between Internet and Ethernet addresses. Thus, yet another *pseudo-device* is needed. The additional device specifications are show below.

```
pseudo-device       inet
pseudo-device       pty
# software loopback device for testing
pseudo-device       loop
pseudo-device       imp
device              acc0        at uba? csr 0167600     vector accrint accxint
pseudo-device       ether
device              ec0         at uba? csr 0164330     vector ecrint eccollide ecxin
device              il0         at uba? csr 0164000     vector ilrint ilcint
```

The completed configuration file for UCBVAX is shown in Appendix C.

5.3. Miscellaneous comments

It should be noted in these examples that neither system was configured to use disk quotas or the 4.1BSD compatibility mode. To use these optional facilities, and others, we would probably clean out our current configuration, reconfigure the system, then recompile and relink the system image(s). This could, of course, be avoided by figuring out which relocatable object files are affected by the reconfiguration, then reconfiguring and recompiling only those files affected by the configuration change. This technique should be used carefully.

6. ADDING NEW SYSTEM SOFTWARE

This section is not for the novice, it describes some of the inner workings of the configuration process as well as the pertinent parts of the system autoconfiguration process. It is intended to give those people who intend to install new device drivers and/or other system facilities sufficient information to do so in the manner which will allow others to easily share the changes.

This section is broken into four parts:

- general guidelines to be followed in modifying system code,
- how to add a device driver to 4.2BSD,
- how UNIBUS device drivers are autoconfigured under 4.2BSD on the VAX, and
- how to add non-standard system facilities to 4.2BSD.

6.1. Modifying system code

If you wish to make site-specific modifications to the system it is best to bracket them with

```
#ifdef SITENAME
...
#endif
```

to allow your source to be easily distributed to others, and also to simplify *diff*(1) listings. If you choose not to use a source code control system (e.g. SCCS, RCS), and perhaps even if you do, it is recommended that you save the old code with something of the form:

```
#ifndef SITENAME
...
#endif
```

We try to isolate our site-dependent code in individual files which may be configured with pseudo-device specifications.

Indicate machine specific code with "#ifdef vax". 4.2BSD has undergone extensive work to make it extremely portable to machines with similar architectures – you may someday find yourself trying to use a single copy of the source code on multiple machines.

Use *lint* periodically if you make changes to the system. The 4.2BSD release has only one line of lint in it. It is very simple to lint the kernel. Use the LINT configuration file, designed to pull in as much of the kernel source code as possible, in the following manner.

```
$ cd /sys/conf
$ mkdir ../LINT
$ config LINT
$ cd ../LINT
$ make depend
$ make assym.s
$ make -k lint > linterrs 2>&1 &
(or for users of csh(1))
% make -k >& linterrs
```

This takes about 45 minutes on a lightly loaded VAX-11/750, but is well worth it.

6.2. Adding device drivers to 4.2BSD

The i/o system and *config* have been designed to easily allow new device support to be added. As described in "Installing and Operating 4.2BSD on the VAX", the system source directories are organized as follows:

/sys/h	machine independent include files
/sys/sys	machine independent system source files
/sys/conf	site configuration files and basic templates
/sys/net	network independent, but network related code
/sys/netinet	DARPA Internet code
/sys/netimp	IMP support code
/sys/netpup	PUP-1 support code
/sys/vax	VAX specific mainline code
/sys/vaxif	VAX network interface code
/sys/vaxmba	VAX MASSBUS device drivers and related code
/sys/vaxuba	VAX UNIBUS device drivers and related code

Existing block and character device drivers for the VAX reside in "/sys/vax", "/sys/vaxmba", and "/sys/vaxuba". Network interface drivers reside in "/sys/vaxif". Any new device drivers should be placed in the appropriate source code directory and named so as not to conflict with existing devices. Normally, definitions for things like device registers are placed in a separate file in the same directory. For example, the "dh" device driver is named "dh.c" and its associated include file is named "dhreg.h".

Once the source for the device driver has been placed in a directory, the file "/sys/conf/files.machine", and possibly "/sys/conf/devices.machine" should be modified. The *files* files in the conf directory contain a line for each source or binary-only file in the system. Those files which are machine independent are located in "/sys/conf/files" while machine specific files are in "/sys/conf/files.machine". The "devices.machine" file is used to map device names to major block device numbers. If the device driver being added provides support for a new disk you will want to modify this file (the format is obvious).

The format of the *files* file has grown somewhat complex over time. Entries are normally of the form

vaxuba/foo.c **optional** foo **device-driver**

where the keyword *optional* indicates that to compile the "foo" driver into the system it must be specified in the configuration file. If instead the driver is specified as *standard*, the file will be loaded no matter what configuration is requested. This is not normally done with device drivers. The fact that the file is specified as a *device-driver* results, on the VAX, in the compilation including a **-i** option for the C optimizer. This is required when pointer references are made to memory locations in the VAX i/o address space.

Aside from including the driver in the *files* file, it must also be added to the device configuration tables. These are located in "/sys/vax/conf.c", or similar for machines other than the VAX. If you don't understand what to add to this file, you should study an entry for an existing driver. Remember that the position in the block device table specifies what the major block device number is, this is needed in the "devices.machine" file if the device is a disk.

With the configuration information in place, your configuration file appropriately modified, and a system reconfigured and rebooted you should incorporate the shell commands needed to install the special files in the file system to the file "/dev/MAKEDEV" or "/dev/MAKEDEV.local". This is discussed in the document "Installing and Operating 4.2BSD on the VAX".

6.3. Autoconfiguration on the VAX

4.2BSD (and 4.1BSD) require all device drivers to conform to a set of rules which allow the system to:

1) support multiple UNIBUS and MASSBUS adapters,

2) support system configuration at boot time, and

3) manage resources so as not to crash when devices request resources which are unavailable.

In addition, devices such as the RK07 which require everyone else to get off the UNIBUS when they are running need cooperation from other DMA devices if they are to work. Since it is unlikely that you will be writing a device driver for a MASSBUS device, this section is devoted exclusively to describing the i/o system and autoconfiguration process as it applies to UNIBUS devices.

Each UNIBUS on a VAX has a set of resources:

- 496 map registers which are used to convert from the 18 bit UNIBUS addresses into the much larger VAX address space.
- Some number of buffered data paths (3 on an 11/750, 15 on an 11/780, 0 on an 11/730) which are used by high speed devices to transfer data using fewer bus cycles.

There is a structure of type *struct uba_hd* in the system per UNIBUS adapter used to manage these resources. This structure also contains a linked list where devices waiting for resources to complete DMA UNIBUS activity have requests waiting.

There are three central structures in the writing of drivers for UNIBUS controllers; devices which do not do DMA i/o can often use only two of these structures. The structures are *struct uba_ctlr*, the UNIBUS controller structure, *struct uba_device* the UNIBUS device structure, and *struct uba_driver*, the UNIBUS driver structure. The *uba_ctlr* and *uba_device* structures are in one-to-one correspondence with the definitions of controllers and devices in the system configuration. Each driver has a *struct uba_driver* structure specifying an internal interface to the rest of the system.

Thus a specification

```
controller sc0 at uba0 csr 0176700 vector upintr
```

would cause a *struct uba_ctlr* to be declared and initialized in the file *ioconf.c* for the system configured from this description. Similarly specifying

```
disk up0 at sc0 drive 0
```

would declare a related *uba_device* in the same file. The *up.c* driver which implements this driver specifies in its declarations:

```
int     upprobe(), upslave(), upattach(), updgo(), upintr();
struct  uba_ctlr *upminfo[NSC];
struct  uba_device *updinfo[NUP];
u_short upstd[] = { 0776700, 0774400, 0776300, 0 };
struct  uba_driver scdriver =
    { upprobe, upslave, upattach, updgo, upstd, "up", updinfo, "sc", upminfo };
```

initializing the *uba_driver* structure. The driver will support some number of controllers named *sc0*, *sc1*, etc, and some number of drives named *up0*, *up1*, etc. where the drives may be on any of the controllers (that is there is a single linear name space for devices, separate from the controllers.)

We now explain the fields in the various structures. It may help to look at a copy of *vaxuba/ubareg.h*, *h/ubavar.h* and drivers such as *up.c* and *dz.c* while reading the descriptions of the various structure fields.

uba_driver structure

One of these structures exists per driver. It is initialized in the driver and contains functions used by the configuration program and by the UNIBUS resource routines. The fields of the structure are:

ud_probe

A routine which is given a *caddr_t* address as argument and should cause an interrupt on the device whose control-status register is at that address in virtual memory. It may be the case that the device does not exist, so the probe routine should use delays (via the DELAY(n) macro which delays for *n* microseconds) rather than waiting for specific events to occur. The routine must **not** declare its argument as a *register* parameter, but **must** declare

```
register int br, cvec;
```

as local variables. At boot time the system takes special measures that these variables are "value-result" parameters. The *br* is the IPL of the device when it interrupts, and the *cvec* is the interrupt vector address on the UNIBUS. These registers are actually filled in in the interrupt handler when an interrupt occurs.

As an example, here is the *up.c* probe routine:

```
upprobe(reg)
        caddr_t reg;
{
        register int br, cvec;

#ifdef lint
        br = 0; cvec = br; br = cvec;
#endif
        ((struct updevice *)reg)->upcs1 = UP_IE|UP_RDY;
        DELAY(10);
        ((struct updevice *)reg)->upcs1 = 0;
        return (sizeof (struct updevice));
}
```

The definitions for *lint* serve to indicate to it that the *br* and *cvec* variables are value-result. The statements here interrupt enable the device and write the ready bit UP_RDY. The 10 microsecond delay insures that the interrupt enable will not be canceled before the interrupt can be posted. The return of "sizeof (struct updevice)" here indicates that the probe routine is satisfied that the device is present (the value returned is not currently used, but future plans dictate you should return the amount of space in the device's register bank). A probe routine may use the function "badaddr" to see if certain other addresses are accessible on the UNIBUS (without generating a machine check), or look at the contents of locations where certain registers should be. If the registers contents are not acceptable or the addresses don't respond, the probe routine can return 0 and the device will not be considered to be there.

One other thing to note is that the action of different VAXen when illegal addresses are accessed on the UNIBUS may differ. Some of the machines may generate machine checks and some may cause UNIBUS errors. Such considerations are handled by the configuration program and the driver writer need not be concerned with them.

It is also possible to write a very simple probe routine for a one-of-a-kind device if probing is difficult or impossible. Such a routine would include statements of the form:

```
br = 0x15;
cvec = 0200;
```

for instance, to declare that the device ran at UNIBUS br5 and interrupted through vector 0200 on the UNIBUS. The current UDA-50 driver does something similar to this because the device is so difficult to force an interrupt on that it hardly seems worthwhile.

ud_slave

This routine is called with a *uba_device* structure (yet to be described) and the address of the device controller. It should determine whether a particular slave device of a

controller is present, returning 1 if it is and 0 if it is not. As an example here is the slave routine for *up.c*.

```
upslave(ui, reg)
        struct uba_device *ui;
        caddr_t reg;
{
        register struct updevice *upaddr = (struct updevice *)reg;

        upaddr->upcs1 = 0;              /* conservative */
        upaddr->upcs2 = ui->ui_slave;
        if (upaddr->upcs2&UPCS2 NED) {
                upaddr->upcs1 = UP_DCLR|UP GO;
                return (0);
        }
        return (1);
}
```

Here the code fetches the slave (disk unit) number from the *ui_slave* field of the *uba_device* structure, and sees if the controller responds that that is a non-existent driver (NED). If the drive a drive clear is issued to clean the state of the controller, and 0 is returned indicating that the slave is not there. Otherwise a 1 is returned.

ud_attach

The attach routine is called after the autoconfigure code and the driver concur that a peripheral exists attached to a controller. This is the routine where internal driver state about the peripheral can be initialized. Here is the *attach* routine from the *up.c* driver:

```
upattach(ui)
        register struct uba_device *ui;
{
        register struct updevice *upaddr;

        if (upwstart == 0) {
                timeout(upwatch, (caddr_t)0, hz);
                upwstart++;
        }
        if (ui->ui_dk >= 0)
                dk mspw[ui->ui_dk] = .0000020345;
        upip[ui->ui ctlr][ui->ui_slave] = ui;
        up_softc[ui->ui ctlr].sc_ndrive++;
        ui->ui_type = upmaptype(ui);
}
```

The attach routine here performs a number of functions. The first time any drive is attached to the controller it starts the timeout routine which watches the disk drives to make sure that interrupts aren't lost. It also initializes, for devices which have been assigned *iostat* numbers (when ui->ui_dk >= 0), the transfer rate of the device in the array *dk_mspw*, the fraction of a second it takes to transfer 16 bit word. It then initializes an inverting pointer in the array *upip* which will be used later to determine, for a particular *up* controller and slave number, the corresponding *uba_device*. It increments the count of the number of devices on this controller, so that search commands can later be avoided if the count is exactly 1. It then attempts to decipher the actual type of drive attached to the controller in a controller-specific way. On the EMULEX SC-21 it may ask for the number of tracks on the device and use this to decide what the drive type is. The drive type is used to setup disk partition mapping tables and other device specific information.

ud_dgo
Is the routine which is called by the UNIBUS resource management routines when an operation is ready to be started (because the required resources have been allocated). The routine in *up.c* is:

```
updgo(um)
        struct uba ctlr *um;
{
        register struct updevice *upaddr = (struct updevice *)um->um_addr;

        upaddr->upba = um->um ubinfo;
        upaddr->upcs1 = um->um cmd|((um->um ubinfo>>8)&0x300);
}
```

This routine uses the field *um_ubinfo* of the *uba_ctlr* structure which is where the UNIBUS routines store the UNIBUS map allocation information. In particluar, the low 18 bits of this word give the UNIBUS address assigned to the transfer. The assignment to *upba* in the go routine places the low 16 bits of the UNIBUS address in the disk UNIBUS address register. The next assignment places the disk operation command and the extended (high 2) address bits in the device control-status register, starting the i/o operation. The field *um_cmd* was initialized with the command to be stuffed here in the driver code itself before the call to the *ubago* routine which eventually resulted in the call to *updgo*.

ud_addr
Are the conventional addresses for the device control registers in UNIBUS space. This information is used by the system to look for instances of the device supported by the driver. When the system probes for the device it first checks for a control-status register located at the address indicated in the configuration file (if supplied), then uses the list of conventional addresses pointed to be *ud_addr*.

ud_dname
Is the name of a *device* supported by this controller; thus the disks on a SC-21 controller are called *up0*, *up1*, etc. That is because this field contains *up*.

ud_dinfo
Is an array of back pointers to the *uba device* structures for each device attached to the controller. Each driver defines a set of controllers and a set of devices. The device address space is always one-dimensional, so that the presence of extra controllers may be masked away (e.g. by pattern matching) to take advantage of hardware redundancy. This field is filled in by the configuration program, and used by the driver.

ud_mname
The name of a controller, e.g. *sc* for the *up.c* driver. The first SC-21 is called *sc0*, etc.

ud_minfo
The backpointer array to the structures for the controllers.

ud_xclu
If non-zero specifies that the controller requires exclusive use of the UNIBUS when it is running. This is non-zero currently only for the RK611 controller for the RK07 disks to map around a hardware problem. It could also be used if 6250bpi tape drives are to be used on the UNIBUS to insure that they get the bandwidth that they need (basically the whole bus).

uba_ctlr structure

One of these structures exists per-controller. The fields link the controller to its UNIBUS adapter and contain the state information about the devices on the controller. The fields are:

um_driver
A pointer to the *struct uba_device* for this driver, which has fields as defined above.

um_ctlr
The controller number for this controller, e.g. the 0 in *sc0*.

um_alive
Set to 1 if the controller is considered alive; currently, always set for any structure encountered during normal operation. That is, the driver will have a handle on a *uba_ctlr* structure only if the configuration routines set this field to a 1 and entered it into the driver tables.

um_intr
The interrupt vector routines for this device. These are generated by *config* and this field is initialized in the *ioconf.c* file.

um_hd
A back-pointer to the UNIBUS adapter to which this controller is attached.

um_cmd
A place for the driver to store the command which is to be given to the device before calling the routine *ubago* with the devices *uba_device* structure. This information is then retrieved when the device go routine is called and stuffed in the device control status register to start the i/o operation.

um_ubinfo
Information about the UNIBUS resources allocated to the device. This is normally only used in device driver go routine (as *updgo* above) and occasionally in exceptional condition handling such as ECC correction.

um_tab
This buffer structure is a place where the driver hangs the device structures which are ready to transfer. Each driver allocates a buf structure for each device (e.g. *updtab* in the *up.c* driver) for this purpose. You can think of this structure as a device-control-block, and the buf structures linked to it as the unit-control-blocks. The code for dealing with this structure is stylized; see the *rk.c* or *up.c* driver for the details. If the *ubago* routine is to be used, the structure attached to this *buf* structure must be:

- A chain of *buf* structures for each waiting device on this controller.
- On each waiting *buf* structure another *buf* structure which is the one containing the parameters of the i/o operation.

uba_device structure

One of these structures exist for each device attached to a UNIBUS controller. Devices which are not attached to controllers or which perform no buffered data path DMA i/o may have only a device structure. Thus *dz* and *dh* devices have only *uba device* structures. The fields are:

ui_driver
A pointer to the *struct uba_driver* structure for this device type.

ui_unit
The unit number of this device, e.g. 0 in **up0**, or 1 in **dh1**.

ui_ctlr
The number of the controller on which this device is attached, or −1 if this device is not on a controller.

ui_ubanum
The number of the UNIBUS on which this device is attached.

ui_slave
The slave number of this device on the controller which it is attached to, or −1 if the

device is not a slave. Thus a disk which was unit 2 on a SC-21 would have *ui_slave* 2; it might or might not be *up2*, that depends on the system configuration specification.

ui_intr
The interrupt vector entries for this device, copied into the UNIBUS interrupt vector at boot time. The values of these fields are filled in by *config* to small code segments which it generates in the file *ubglue.s*.

ui_addr
The control-status register address of this device.

ui_dk
The iostat number assigned to this device. Numbers are assigned to disks only, and are small positive integers which index the various *dk* * arrays in *<sys/dk.h>*.

ui_flags
The optional "flags *xxx*" parameter from the configuration specification was copied to this field, to be interpreted by the driver. If *flags* was not specified, then this field will contain a 0.

ui_alive
The device is really there. Presently set to 1 when a device is determined to be alive, and left 1.

ui_type
The device type, to be used by the driver internally.

ui_physaddr
The physical memory address of the device control-status register. This is used in the device dump routines typically.

ui_mi
A *struct uba_ctlr* pointer to the controller (if any) on which this device resides.

ui_hd
A *struct uba_hd* pointer to the UNIBUS on which this device resides.

UNIBUS resource management routines

UNIBUS drivers are supported by a collection of utility routines which manage UNIBUS resources. If a driver attempts to bypass the UNIBUS routines, other drivers may not operate properly. The major routines are: *uballoc* to allocate UNIBUS resources, *ubarelse* to release previously allocated resources, and *ubago* to initiate DMA. When allocating UNIBUS resources you may request that you

NEEDBDP
if you need a buffered data path,

HAVEBDP
if you already have a buffered data path and just want new mapping registers (and access to the UNIBUS), and

CANTWAIT
if you are calling (potentially) from interrupt level

If the presentation here does not answer all the questions you may have, consult the file /sys/vaxuba/uba.c

Autoconfiguration requirements

Basically all you have to do is write a *ud_probe* and a *ud_attach* routine for the controller. It suffices to have a *ud_probe* routine which just initializes *br* and *cvec*, and a *ud_attach* routine which does nothing. Making the device fully configurable requires, of course, more work, but is worth it if you expect the device to be in common usage and want to share it with others.

If you managed to create all the needed hooks, then make sure you include the necessary header files; the ones included by *vaxuba/ct.c* are nearly minimal. Order is important here, don't be surprised at undefined structure complaints if you order the includes wrongly. Finally if you get the device configured in, you can try bootstrapping and see if configuration messages print out about your device. It is a good idea to have some messages in the probe routine so that you can see that you are getting called and what is going on. If you do not get called, then you probably have the control-status register address wrong in your system configuration. The autoconfigure code notices that the device doesn't exist in this case and you will never get called.

Assuming that your probe routine works and you manage to generate an interrupt, then you are basically back to where you would have been under older versions of UNIX. Just be sure to use the *ui ctlr* field of the *uba device* structures to address the device; compiling in funny constants will make your driver only work on the CPU type you have (780, 750, or 730).

Other bad things that might happen while you are setting up the configuration stuff:

- You get "nexus zero vector" errors from the system. This will happen if you cause a device to interrupt, but take away the interrupt enable so fast that the UNIBUS adapter cancels the interrupt and confuses the processor. The best thing to do it to put a modest delay in the probe code between the instructions which should cause and interrupt and the clearing of the interrupt enable. (You should clear interrupt enable before you leave the probe routine so the device doesn't interrupt more and confuse the system while it is configuring other devices.)
- The device refuses to interrupt or interrupts with a "zero vector". This typically indicates a problem with the hardware or, for devices which emulate other devices, that the emulation is incomplete. Devices may fail to present interrupt vectors because they have configuration switches set wrong, or because they are being accessed in inappropriate ways. Incomplete emulation can cause "maintenance mode" features to not work properly, and these features are often needed to force device interrupts.

6.4. Adding non-standard system facilities

This section considers the work needed to augment *config*'s data base files for non-standard system facilities.

As far as *config* is concerned non-standard facilities may fall into two categories. *Config* understands that certain files are used especially for kernel profiling. These files are indicated in the *files* files with a *profiling-routine* keyword. For example, the current profiling subroutines are sequestered off in a separate file with the following entry:

sys/subr mcount.c **optional profiling-routine**

The *profiling-routine* keyword forces *config* to not compile the source file with the **−pg** option.

The second keyword which can be of use is the *config-dependent* keyword. This causes *config* to compile the indicated module with the global configuration parameters. This allows certain modules, such as *machdep.c* to size system data structures based on the maximum number of users configured for the system.

APPENDIX A. CONFIGURATION FILE GRAMMAR

The following grammar is a compressed form of the actual *yacc*(1) grammar used by *config* to parse configuration files. Terminal symbols are shown all in upper case, literals are emboldened; optional clauses are enclosed in brackets, "[" and "]"; zero or more instantiations are denoted with "*".

Configuration ::= [Spec ;]*

Spec ::= Config spec
| Device spec
| **trace**
| /* lambda */

/* configuration specifications */

Config spec ::= **machine** ID
| **cpu** ID
| **options** Opt list
| **ident** ID
| System spec
| **timezone** [−] NUMBER [**dst** [NUMBER]]
| **timezone** [−] FPNUMBER [**dst** [NUMBER]]
| **maxusers** NUMBER

/* system configuration specifications */

System spec ::= **config** ID System parameter [System parameter]*

System parameter ::= swap spec | root spec | dump spec | arg spec

swap spec ::= **swap** [**on**] swap dev [**and** swap dev]*

swap dev ::= dev spec [**size** NUMBER]

root spec ::= **root** [**on**] dev spec

dump spec ::= **dumps** [**on**] dev spec

arg spec ::= **args** [**on**] dev spec

dev spec ::= dev name | major minor

major minor ::= **major** NUMBER **minor** NUMBER

dev name ::= ID [NUMBER [ID]]

/* option specifications */

Opt list ::= Option [, Option]*

Option ::= ID [= Opt value]

Opt value ::= ID | NUMBER

/* device specifications */

Device spec ::= **device** Dev name Dev info Int spec
| **master** Dev name Dev info
| **disk** Dev name Dev info
| **tape** Dev name Dev info
| **controller** Dev name Dev info [Int spec]
| **pseudo-device** Dev [NUMBER]

Dev name ::= Dev NUMBER

Dev ::= **uba** | **mba** | ID

Dev info ::= Con info [Info]*

Con info ::= **at** Dev NUMBER
| **at nexus** NUMBER

Info ::= **csr** NUMBER
| **drive** NUMBER
| **slave** NUMBER
| **flags** NUMBER

Int spec ::= **vector** ID [ID]*
| **priority** NUMBER

Lexical Conventions

The terminal symbols are loosely defined as:

ID
: One or more alphabetics, either upper or lower case, and underscore, " ".

NUMBER
: Approximately the C language specification for an integer number. That is, a leading "0x" indicates a hexadecimal value, a leading "0" indicates an octal value, otherwise the number is expected to be a decimal value. Hexadecimal numbers may use either upper or lower case alphabetics.

FPNUMBER
: A floating point number without exponent. That is a number of the form "nnn.ddd", where the fractional component is optional.

In special instances a question mark, "?", can be substituted for a "NUMBER" token. This is used to effect wildcarding in device interconnection specifications.

Comments in configuration files are indicated by a "#" character at the beginning of the line; the remainder of the line is discarded.

A specification is interpreted as a continuation of the previous line if the first character of the line is tab.

APPENDIX B. RULES FOR DEFAULTING SYSTEM DEVICES

When *config* processes a "config" rule which does not fully specify the location of the root file system, paging area(s), device for system dumps, and device for argument list processing it applies a set of rules to define those values left unspecified. The following list of rules are used in defaulting system devices.

1) If a root device is not specified, the swap specification must indicate a "generic" system is to be built.
2) If the root device does not specify a unit number, it defaults to unit 0.
3) If the root device does not include a partition specification, it defaults to the "a" partition.
4) If no swap area is specified, it defaults to the "b" partition of the root device.
5) If no device is specified for processing argument lists, the first swap partition is selected.
6) If no device is chosen for system dumps, the first swap partition is selected (see below to find out where dumps are placed within the partition).

The following table summarizes the default partitions selected when a device specification is incomplete, e.g. "hp0".

Type	Partition
root	"a"
swap	"b"
args	"b"
dumps	"b"

Multiple swap/paging areas

When multiple swap partitions are specified, the system treats the first specified as a "primary" swap area which is always used. The remaining partitions are then interleaved into the paging system at the time a *swapon*(2) system call is made. This is normally done at boot time with a call to *swapon*(8) from the /etc/rc file.

System dumps

System dumps are automatically taken after a system crash, provided the device driver for the "dumps" device supports this. The dump contains the contents of memory, but not the swap areas. Normally the dump device is a disk in which case the information is copied to a location near the back of the partition. The dump is placed in the back of the partition because the primary swap and dump device are commonly the same device and this allows the system to be rebooted without immediately overwriting the saved information. When a dump has occurred, the system variable *dumpsize* is set to a non-zero value indicating the size (in bytes) of the dump. The *savecore*(8) program then copies the information from the dump partition to a file in a "crash" directory and also makes a copy of the system which was running at the time of the crash (usually "/vmunix"). The offset to the system dump is defined in the system variable *dumplo* (a sector offset from the front of the dump partition). The *savecore* program operates by reading the contents of *dumplo*, *dumpdev*, and *dumpmagic* from /dev/kmem, then comparing the value of *dumpmagic* read from /dev/kmem to that located in corresponding location in the dump area of the dump partition. If a match is found, *savecore* assumes a crash occurred and reads *dumpsize* from the dump area of the dump partition. This value is then used in copying the system dump. Refer to *savecore*(8) for more information about its operation.

The value *dumplo* is calculated to be

dumpdev-size – DUMPDEV

where *dumpdev-size* is the size of the disk partition where system dumps are to be placed, and DUMPDEV is 10 Megabytes. If the disk partition is not large enough to hold a 10 Megabyte dump, *dumplo* is set to 0 (the front of the partition). For sites with more than 10 Megabytes of memory the definition of DUMPDEV in /sys/vax/autoconf.c will have to be changed.

APPENDIX C. SAMPLE CONFIGURATION FILES

The following configuration files are developed in section 5; they are included here for completeness.

```
#
# ANSEL VAX (a picture perfect machine)
#
machine         vax
cpu             VAX780
timezone        8 dst
ident           ANSEL
maxusers        40

config          vmunix      root on hp0
config          hpvmunix    root on hp0 swap on hp0 and hp2
config          genvmunix   swap generic

controller      mba0        at nexus ?
disk            hp0         at mba? disk ?
disk            hp1         at mba? disk ?
controller      mba1        at nexus ?
disk            hp2         at mba? disk ?
disk            hp3         at mba? disk ?
controller      uba0        at nexus ?
controller      tm0         at uba? csr 0172520     vector tmintr
tape            te0         at tm0 drive 0
tape            te1         at tm0 drive 1
device          dh0         at uba? csr 0160020     vector dhrint dhxint
device          dm0         at uba? csr 0170500     vector dmintr
device          dh1         at uba? csr 0160040     vector dhrint dhxint
device          dh2         at uba? csr 0160060     vector dhrint dhxint
```

```
#
# UCBVAX - Gateway to the world
#
machine         vax
cpu             "VAX780"
cpu             "VAX750"
ident           UCBVAX
timezone        8 dst
maxusers        32
options         INET

config          vmunix      root on hp swap on hp and rk0 and rk1
config          upvmunix    root on up
config          hkvmunix    root on hk swap on rk0 and rk1

controller      mba0        at nexus ?
controller      uba0        at nexus ?
disk            hp0         at mba? drive 0
disk            hp1         at mba? drive 1
controller      sc0         at uba? csr 0176700     vector upintr
disk            up0         at sc0 drive 0
disk            up1         at sc0 drive 1
controller      hk0         at uba? csr 0177440     vector rkintr
disk            rk0         at hk0 drive 1
disk            rk1         at hk0 drive 1
pseudo-device   inet
pseudo-device   pty
# software loopback device for testing
pseudo-device   loop
pseudo-device   imp
device          acc0        at uba? csr 0167600     vector accrint accxint
pseudo-device   ether
device          ec0         at uba? csr 0164330     vector ecrint eccollide ecxint
device          il0         at uba? csr 0164000     vector ilrint ilcint
```

APPENDIX D. VAX KERNEL DATA STRUCTURE SIZING RULES

Certain system data structures are sized at compile time according to the maximum number of simultaneous users expected, while others are calculated at boot time based on the physical resources present; e.g. memory. This appendix lists both sets of rules and also includes some hints on changing built-in limitations on certain data structures.

Compile time rules

The file */sys/conf/param.c* contains the definitions of almost all data structures sized at compile time. This file is copied into the directory of each configured system to allow configuration-dependent rules and values to be maintained. The rules implied by its contents are summarized below (here MAXUSERS refers to the value defined in the configuration file in the "maxusers" rule).

nproc
The maximum number of processes which may be running at any time. It is defined to be 20 + 8 * MAXUSERS and referred to in other calculations as NPROC.

ntext
The maximum number of active shared text segments. Defined as 24 + MAXUSERS + NETSLOP, where NETSLOP is 20 when the Internet protocols are configured in the system and 0 otherwise. The added size for supporting the network is to take into account the numerous server processes which are likely to exist.

ninode
The maximum number of files in the file system which may be active at any time. This includes files in use by users, as well as directory files being read or written by the system and files associated with bound sockets in the UNIX ipc domain. This is defined as (NPROC + 16 + MAXUSERS) + 32.

nfile
The number of "file table" structures. One file table structure is used for each open, unshared, file descriptor. Multiple file descriptors may reference a single file table entry when they are created through a *dup* call, or as the result of a *fork*. This is defined to be

16 * (NPROC + 16 + MAXUSERS) / 10 + 32 + 2 * NETSLOP

where NETSLOP is defined as for **ntext**.

ncallout
The number of "callout" structures. One callout structure is used per internal system event handled with a timeout. Timeouts are used for terminal delays, watchdog routines in device drivers, protocol timeout processing, etc. This is defined as 16 + NPROC.

nclist
The number of "c-list" structures. C-list structures are used in terminal i/o. This is defined as 100 + 16 * MAXUSERS.

nmbclusters
The maximum number of pages which may be allocated by the network. This is defined as 256 (a quarter megabyte of memory) in /sys/h/mbuf.h. In practice, the network rarely uses this much memory. It starts off by allocating 64 kilobytes of memory, then requesting more as required. This value represents an upper bound.

nquota
The number of "quota" structures allocated. Quota structures are present only when disc quotas are configured in the system. One quota structure is kept per user. This is defined to be (MAXUSERS * 9) / 7 + 3.

ndquot

The number of "dquot" structures allocated. Dquot structures are present only when disc quotas are configured in the system. One dquot structure is required per user, per active file system quota. That is, when a user manipulates a file on a file system on which quotas are enabled, the information regarding the user's quotas on that file system must be in-core. This information is cached, so that not all information must be present in-core all the time. This is defined as (MAXUSERS * NMOUNT) / 4 + NPROC, where NMOUNT is the maximum number of mountable file systems.

In addition to the above values, the system page tables (used to map virtual memory in the kernel's address space) are sized at compile time by the SYSPTSIZE definition in the file /sys/vax/param.h. This is defined to be 20 + MAXUSERS pages of page tables. Its definition affects the size of many data structures allocated at boot time because it constrains the amount of virtual memory which may be addressed by the running system. This is often the limiting factor in the size of the buffer cache.

Run-time calculations

The most important data structures sized at run-time are those used in the buffer cache. Allocation is done by swiping physical memory (and the associated virtual memory) immediately after the system has been started up; look in the file /sys/vax/machdep.c. The amount of physical memory which may be allocated to the buffer cache is constrained by the size of the system page tables, among other things. While the system may calculate a large amount of memory to be allocated to the buffer cache, if the system page table is too small to map this physical memory into the virtual address space of the system, only as much as can be mapped will be used.

The buffer cache is comprised of a number of "buffer headers" and a pool of pages attached to these headers. Buffer headers are divided into two categories: those used for swapping and paging, and those used for normal file i/o. The system tries to allocate 10% of available physical memory for the buffer cache (where *available* does not count that space occupied by the system's text and data segments). If this results in fewer than 16 pages of memory allocated, then 16 pages are allocated. This value is kept in the initialized variable *bufpages* so that it may be patched in the binary image (to allow tuning without recompiling the system). A sufficient number of file i/o buffer headers are then allocated to allow each to hold 2 pages each, and half as many swap i/o buffer headers are then allocated. The number of swap i/o buffer headers is constrained to be no more than 256.

System size limitations

As distributed, the sum of the virtual sizes of the core-resident processes is limited to 64M bytes. The size of the text, and data segments of a single process are currently limited to 6M bytes each, and the stack segment size is limited to 512K bytes as a soft, user-changeable limit, and may be increased to 6M with the *setrlimit*(2) system call. If these are insufficient, they can be increased by changing the constants MAXTSIZ, MAXDSIZ and MAXSSIZ in the file /sys/vax/vmparam.h, while changing the definitions in /sys/h/dmap.h and /sys/h/text.h. You must be careful in doing this that you have adequate paging space. As normally configured , the system has only 16M bytes per paging area. The best way to get more space is to provide multiple, thereby interleaved, paging areas.

To increase the amount of resident virtual space possible, you can alter the constant USRPTSIZE (in /sys/vax/vmparam.h). To allow 128 megabytes of resident virtual space one would change the 8 to a 16.

Because the file system block numbers are stored in page table *pg blkno* entries, the maximum size of a file system is limited to 2^19 1024 byte blocks. Thus no file system can be larger than 512M bytes.

The count of mountable file systems is limited to 15. This should be sufficient. If you have many disks it makes sense to make some of them single file systems, and the paging

areas don't count in this total. To increase this it will be necessary to change the core-map /sys/h/cmap.h since there is a 4 bit field used here. The size of the core-map will then expand to 16 bytes per 1024 byte page. (Don't forget to change MSWAPX and NMOUNT in /sys/h/param.h also.)

The maximum value NOFILE (open files per process limit) can be raised to is 30 because of a bit field in the page table entry in /sys/machine/pte.h.

Setting Up Version 1.0 of Unix/32V Operating System

Thomas B. London
John F. Reiser

Bell Laboratories
Holmdel, New Jersey 07733

The distribution tape can be used only on a DEC VAX-11/780 with RP06 or RM03 disks and with TE16 tape drives. The tape consists of some preliminary bootstrapping programs followed by one filesystem image and one tape archive image (see tar(1)); if needed, individual files can be extracted after the initial construction of the filesystems.

If you are set up to do it, it is a good idea immediately to make a copy of the tape to guard against disaster. The tape is 9-track 800 BPI and contains some 512-byte records followed by many 10240-byte records. There are interspersed tapemarks; end-of-tape is signalled by a double end-of-file.

The tape contains binary images of the system and all the user level programs, along with source and manual sections for them. There are about 2100 UNIX files altogether. The first two tape files contain binary images, along with other things needed to flesh out the filesystem enough so UNIX will run. The second tape file is to be put on one filesystem called the 'root filesystem'. The filesystem size required is about 9600 blocks. The third tape file has all of the source and documentation. Altogether it requires about 20,000 512-byte disk blocks.

Making a Disk From Tape

This description is an annotated version of the 'sysgen' manual page in section 8 of the UNIX Programmer's Manual.

Perform the following bootstrap procedure to obtain a disk with a root filesystem on it.

1. Mount the magtape on drive 0 at load point. [Make sure that the ring is not inserted.]
2. Mount a disk pack on drive 0.
3. Key in at 30000 and execute the following boot program: [You may enter in lower-case, the LSI-11 will echo in upper-case. The machine's printouts are shown in italic, explanatory comments are within ( ). Terminate each line you type by carriage return or line-feed.]

```
>>> HALT
>>> UNJAM
>>> INIT
>>> D 30000 20009FDE
>>> D + D0512001
>>> D + 3204A101
>>> D + C113C08F
>>> D + A1D40424
>>> D + 008FD00C
>>> D + C1800000
>>> D + 8F320800
>>> D + 10A1FE00
>>> D + 00C139D0
>>> D + 00000004
>>> START 30000
```

The tape should move and the CPU should halt at location 3002A. If it doesn't, you probably have entered the program incorrectly. Start over and check your typing.

4. Start the CPU with

```
>>> START 0
```

5. The console should type

```
=
```

If the disk pack is already formatted, skip to step 6. Otherwise, format the pack with

```
(bring in standalone RP06 formatter)
=rp6fmt
rp6fmt : Format RP06 Disk

MBA no. : 0             (format spindle on mba # 0)
unit : 0          (format unit zero)
(this procedure should take about 20 minutes)
(some diagnostic messages may appear here)

unit : -1         (exit from formatter)
=                       (back at tape boot level)
```

6. Next, verify the readability of the pack via

```
(bring in RP06 verifier)
=rpread
rpread : Read RP06 Disk

disk unit : 0           (specify unit zero)
start block : 0         (start at block zero)
no. blocks :            (default is entire pack)

(this procedure should take about 10 minutes)
(some diagnostic messages may appear here)
# Data Check errors : nn        (number of soft errors)
# Other errors : xx             (number of hard errors)
disk unit : -1                  (exit verifier)
=                       (back to tape boot)
```

If the number of 'Other errors' is not zero, consideration should be given to obtaining a

clean pack before proceeding further.

7. Copy the magtape to disk by the following procedure.

```
(bring in the tape to disk program)
=tdcopy
tdcopy : TM03 tape-to-RP06 disk copy

tape MBA # : 1          (tape mba is normally 1)
tape unit # : 0         (tape unit is normally 0)
tape file offset : 1    (skip over tp tape file)
tape block offset : 0

disk MBA # : 0          (disk mba is normally 0)
disk unit : 0           (disk unit is normally 0)
disk block offset : 0   (start at block zero)

no. of input blocks : 480
10240 = tape block size

normal termination
480 input blocks read
9600 output blocks written
=                       (back at tape boot level)
```

You now have a UNIX root filesystem.

Booting UNIX

Since DEC does not provide a program on the console floppy which boots the VAX from a program located at block zero of a disk spindle, we provide one here.

If the console is not in 'LSI mode' (i.e. >>> prompt), type the 'CONTROL-p' key (i.e. hold the control key down while you hit the 'p' key). Perform the following sequence.

```
>>> HALT
>>> LINK              (save the following sequence on the floppy)
                      (the prompt should change to <<<)
<<< HALT
<<< UNJAM
<<< INIT
<<< D 30000 00009FDE      (boot pgm for MBA 0, drive 0)
<<< D + D0512001
<<< D + D004A101
<<< D + 0400C113
<<< D + 10008F32
<<< D + D40424C1
<<< D + 8FD00CA1
<<< D + 80000000
<<< D + 320800C1
<<< D + A1FE008F
<<< D + 28C1D410
<<< D + 14C1D404
<<< D + C139D004
<<< D + 00000400
<<< START 30000
<<< START 2
(to exit from linking mode type 'control-c')
<<< 'control-c'
>>>
```

You are now ready to boot UNIX (yea!). Each time it is necessary to boot (or reboot) UNIX, one simply follows the sequence

```
                      (we should now be in 'LSI mode')
                      (i.e. >>> prompt)
                      (if not, it may be necessary to type 'control-p')
                      (and 'HALT\r' i.e. HALT followed by return key)
>>> PERFORM           (this executes the commands saved in floppy)
                      (link file)
                      (the console should echo each command in the file)
file : unix                 (load and execute /unix)
```

The machine should type the following:

```
real mem  = xxx
avail mem = yyy
#
```

The *mem* messages give the amount of real (physical) memory and the memory available to user programs in bytes. For example, if your machine has 512K bytes of memory, then xxx will be 524228, i.e. exactly 512K.

UNIX is now running, and the 'UNIX Programmer's manual' applies; references below of the form X(Y) mean the subsection named X in section Y of the manual. The '#' is the prompt from the Shell, and indicates you are the super-user. The user name of the super-user is 'root' if you should find yourself in multi-user mode and need to log in. There is no password provided for 'root'; provide one by using passwd(1). In the future, when you reboot from 'LSI mode' (i.e. >>> prompt), you can type just

```
>>> PERFORM           (let the LSI-11 boot the system)
file : unix           (as above)
```

You now need to make some special file entries in the dev directory. These specify what sort of disk you are running on, what sort of tape drive you have, and where the filesystems are. For simplicity, this recipe creates fixed device names. These names will be used below, and some of them are built into various programs, so they are most convenient. For example, 'rp0a' will be used for the name of the root filesystem, and 'rp0h' will be used for the name of the filesystem. Also, this sequence will put the user filesystem on the same disk drive as the root, which is not the best place if you have more than one drive. Thus the prescription below should be taken only as one example of where to put things. See also the section on 'Disk layout' below.

In any event, change to the dev directory (via cd /dev) and, if you like, examine and perhaps change the entries there (use rm(1) and mknod(1)). The file 'rp0a' refers to the root file system; 'swap' to the swap-space filesystem; 'rp0h' to the user filesystem. The devices 'rrp0a' and 'rrp0h' are the 'raw' versions of the disks. Also, 'mt0' is tape drive 0, at 800 BPI; 'rmt0' is the raw tape, on which large records can be read and written; 'rmt4' is raw tape with the quirk that it does not rewind on close, which permits multifile tapes to be handled.

The next thing to do is to extract the rest of the data from the tape. Comments are enclosed in ( ); don't type these. The number in the first command is the size of the filesystem.

```
        (in the following, xxx should be 322278 if
         you are using RP06's, 113280 if RM03's)
        (the following command creates an empty filesystem)
#/etc/mkfs /dev/rp0h xxx
isize = 65496   (this is the number of available inodes)
m/n = 3 500     (freelist interleave parameters)
#/etc/mount /dev/rp0h /usr   (mount the usr filesystem)
#cd /usr        (make /usr the current directory)
#cp /dev/rmt4 /dev/null      (skip first tape file (tp format))
#cp /dev/rmt4 /dev/null      (skip second tape file (root))
#tar xbf 20 /dev/rmt0 (extract the usr filesystem)
#cd /           (back to root)
#/etc/umount /dev/rp0h       (unmount /usr)
```

All of the data on the tape has been extracted. The tape will rewind automatically.

You may at this point mount the source filesystem (mount(1)). To do this type the following:

```
/etc/mount /dev/rp0h /usr
```

The source and manual pages are now available in subdirectories of /usr.

The above mount command is only needed if you intend to play around with source on a single user system. The filesystem is mounted automatically when multi-user mode is entered, by a command in the file /etc/rc. (See 'Disk Layout' below).

Before UNIX is turned up completely, a few configuration dependent exercises must be performed. At this point, it would be wise to read all of the manuals (especially 'Regenerating System Software') and to augment this reading with hand to hand combat.

Reconfiguration

The UNIX system running is configured to run with the given disk and tape, a console, up to 1 megabyte of main memory, and 8 DZ11 lines. This is probably not the correct configuration. You will have to correct the configuration table to reflect the true state of your machine.

It is wise at this point to know how to recompile the system. Print the file /usr/src/sys/sys/makefile using the command cat /usr/src/sys/sys/makefile. This file is input

to the program 'make(1)' which if invoked with 'make unix', will recompile all of the system source.

There are certain magic numbers and configuration parameters imbedded in various device drivers that you may want to change. The device addresses of each device are defined in each driver. In case you have any non-standard device addresses, just change the address and recompile. Also, if the devices's interrupt vector address(es) are not currently known to the system (this is likely), then the file /usr/src/sys/sys/univec.c must be modified appropriately: namely, the proper interrupt routine addresses must be placed in the table 'UNIvec'. Use the DZ11 as an example (as distributed, the DZ11 vectors are assumed to be at locations c0 and c4 (hexadecimal)).

The DZ11 driver is set to run 8 lines. This can be changed in dz.c

The DC11 driver is set to run 4 lines. This can be changed in dc.c.

The DH11 driver is set to handle 3 DH11's with a full complement of 48 lines. If you have less, or more, you may want to edit dh.c.

The DN11 driver will handle 4 DN's. Edit dn.c.

The DU11 driver can only handle a single DU. This cannot be easily changed.

The KL/DL driver is set up to run a single DL11-A, -B, and no DL11-E's. To change this, edit kl.c to have NKL11 reflect the total number of DL11-AB's and NDL11 to reflect the number of DL11-E's. So far as the driver is concerned, the difference between the devices is their address.

The disk and tape drivers (hp.c, ht.c) are set up to run 1 drive and should be changed if you have more. The disk driver (hp.c) has a partition table which you may want to experiment with.

After all the corrections have been made, use 'make(1)' to recompile the system (or recompile individually if you wish: use the makefile as a guide). If you compiled individually, say 'make unix' in the directory /usr/src/sys/sys. The final object file (unix) should be moved to the root, and then booted to try it out. It is best to name it /nunix so as not to destroy the working system until you're sure it does work. See Boot Procedures(8) for a discussion of booting. Note: before taking the system down, always (!!) perform a sync(1) to force delayed output to the disk.

Special Files

Next you must put in special files for the new devices in the directory /dev using mknod(1). Print the configuration file /usr/src/sys/sys/conf.c. This is the major device switch of each device class (block and character). There is one line for each device configured in your system and a null line for place holding for those devices not configured. The essential block special files were installed above; for any new devices, the major device number is selected by counting the line number (from zero) of the device's entry in the block configuration table. Thus the first entry in the table bdevsw would be major device zero. This number is also printed in the table along the right margin.

The minor device is the drive number, unit number or partition as described under each device in section 4. For tapes where the unit is dial selectable, a special file may be made for each possible selection. You can also add entries for other disk drives.

In reality, device names are arbitrary. It is usually convenient to have a system for deriving names, but it doesn't have to be the one presented above.

Some further notes on minor device numbers. The hp driver uses the 0100 bit of the minor device number to indicate whether or not to interleave a filesystem across more than one physical device. See hp(4) for more detail. The ht driver uses the 04 bit to indicate whether or not to rewind the tape when it is closed. The 010 bit indicates the density of the tape on TE16 drives. Again, see ht(4).

The naming of character devices is similar to block devices. Here the names are even more arbitrary except that devices meant to be used for teletype access should (to avoid confusion, no other reason) be named /dev/ttyX, where X is some string (as in '00' or 'library'). The files console, mem, kmem, and null are already correctly configured.

The disk and magtape drivers provide a 'raw' interface to the device which provides direct transmission between the user's core and the device and allows reading or writing large records. The raw device counts as a character device, and should have the name of the corresponding standard block special file with 'r' prepended. Thus the raw magtape files would be called /dev/rmtX. These special files should be made.

When all the special files have been created, care should be taken to change the access modes (chmod(1)) on these files to appropriate values.

Time Conversion

If your machine is not in the Eastern time zone, you must edit (ed(1)) the file /usr/src/sys/h/param.h to reflect your local time. The manifest 'TIMEZONE' should be changed to reflect the time difference between local time and GMT in minutes. For EST, this is 5*60; for PST it would be 8*60. Finally, there is a 'DSTFLAG' manifest; when it is 1 it causes the time to shift to Daylight Savings automatically between the last Sundays in April and October (or other algorithms in 1974 and 1975). Normally this will not have to be reset. When the needed changes are done, recompile and load the system using make(1) and install it. (As a general rule, when a system header file is changed, the entire system should be recompiled. As it happens, the only uses of these flags are in /usr/src/sys/sys/sys4.c, so if this is all that was changed it alone needs to be recompiled.)

You may also want to look at timezone(3) (/usr/src/libc/gen/timezone.c) to see if the name of your timezone is in its internal table. If needed, edit the changes in. After timezone.c has been edited it should be compiled and installed in its library. (See /usr/src/libc/Makefile). Then you should (at your leisure) recompile and reinstall all programs that use it (such as date(1)).

Disk Layout

If there are to be more filesystems mounted than just the root and /usr, use mkfs(1) to create any new filesystem and put its mounting in the file /etc/rc (see init(8) and mount(1)). (You might look at /etc/rc anyway to see what has been provided for you.)

There are two considerations in deciding how to adjust the arrangement of things on your disks: the most important is making sure there is adequate space for what is required; secondarily, throughput should be maximized. Swap space is a critical parameter. The system as distributed has 8778 blocks for swap space. This should be large enough for most sites. You may want to change these if local wisdom indicates otherwise.

The system as distributed has many of the binaries in /bin. Some of them should be moved to /usr/bin, leaving only the ones required for system maintenance (such as icheck, dcheck, cc, ed, tar, restor, etc.) and the most heavily used in /bin. This will speed things up a bit if you have only one disk, and also free up space on the root filesystem for temporary files. (See below).

Many common system programs (C, the editor, the assembler etc.) create intermediate files in the /tmp directory, so the filesystem where this is stored also should be made large enough to accommodate most high-water marks. If you leave the root filesystem as distributed (except as discussed above) there should be no problem. All the programs that create files in /tmp take care to delete them, but most are not immune to events like being hung up upon, and can leave dregs. The directory should be examined every so often and the old files deleted.

Exhaustion of user-file space is certain to occur now and then; the only mechanisms for controlling this phenomenon are occasional use of du(1), df(1), quot(1), threatening messages of the day, and personal letters.

The efficiency with which UNIX is able to use the CPU is largely dictated by the configuration of disk controllers. For general time-sharing applications, the best strategy is to try to split user files, the root directory (including the /tmp directory) and the swap area among three controllers.

Once you have decided how to make best use of your hardware, the question is how to initialize it. If you have the equipment, the best way to move a filesystem is to dump it (dump(1)) to magtape, use mkfs(1) to create the new filesystem, and restore (restor(1)) the tape. If for some reason you don't want to use magtape, dump accepts an argument telling where to put the dump; you might use another disk. Sometimes a filesystem has to be increased in logical size without copying. The super-block of the device has a word giving the highest address which can be allocated. For relatively small increases, this word can be patched using the debugger (adb(1)) and the free list reconstructed using icheck(1). The size should not be increased very greatly by this technique, however, since although the allocatable space will increase the maximum number of files will not (that is, the i-list size can't be changed). Read and understand the description given in filesystem(5) before playing around in this way. You may want to see section rp(4) for some suggestions on how to lay out the information on RP disks.

If you have to merge a filesystem into another, existing one, the best bet is to use tar(1). If you must shrink a filesystem, the best bet is to dump the original and restor it onto the new filesystem. However, this might not work if the i-list on the smaller filesystem is smaller than the maximum allocated inode on the larger. If this is the case, reconstruct the filesystem from scratch on another filesystem (perhaps using tar(1)) and then dump it. If you are playing with the root filesystem and only have one drive the procedure is more complicated. What you do is the following:

1. GET A SECOND PACK!!!!
2. Create an image of the new root filesystem using mkfs(1), dump(1), and restor(1).
3. Make a binary tape image of the new filesystem using dd(1).
4. Bring the system down and mount the new pack.
5. Retrieve the WECo distribution tape and perform steps 1 through 4 at the beginning of this document, then skip to step 7, substituting the desired filesystem size instead of 480 when asked for 'no. of input blocks'.
6. Boot(8) using the newly created disk filesystem.

New Users

Install new users by editing the password file /etc/passwd (passwd(5)). This procedure should be done before multi-user mode is entered (see init(8)). You'll have to make a current directory for each new user and change its owner to the newly installed name. Login as each user to make sure the password file is correctly edited. For example:

```
ed /etc/passwd
$a
joe::47:13::/usr/joe:
.
w
q
mkdir /usr/joe
chown joe /usr/joe
login joe
ls -la
login root
```

This will make a new login entry for joe, who should be encouraged to use passwd(1) to give himself a password. His default current directory is /usr/joe which has been created. The

delivered password file has the user *bin* in it to be used as a prototype.

Multiple Users

If UNIX is to support simultaneous access from more than just the console terminal, the file /etc/ttys (ttys(5)) has to be edited. To add a new terminal be sure the device is configured and the special file exists, then set the first character of the appropriate line of /etc/ttys to 1 (or add a new line). Note that init.c will have to be recompiled if there are to be more than 100 terminals. Also note that if the special file is inaccessible when init tries to create a process for it, the system will thrash trying and retrying to open it.

File System Health

Periodically (say every day or so) and always after a crash, you should check all the filesystems for consistency (icheck, dcheck(1)). It is quite important to execute sync (8) before rebooting or taking the machine down. This is done automatically every 30 seconds by the update program (8) when a multiple-user system is running, but you should do it anyway to make sure.

Dumping of the filesystem should be done regularly, since once the system is going it is very easy to become complacent. Complete and incremental dumps are easily done with dump(1). Dumping of files by name is best done by tar(1) but the number of files is somewhat limited. Finally if there are enough drives entire disks can be copied using cp(1), or preferably with dd(1) using the raw special files and an appropriate block size.

Converting Sixth Edition Filesystems

The best way to convert filesystems from 6th edition (V6) to 7th edition (V7) format is to use tp(1) or tar(1).

Odds and Ends

The programs dump, icheck, quot, dcheck, ncheck, and df (source in /usr/source/cmd) should be changed to reflect your default mounted filesystem devices. Print the first few lines of these programs and the changes will be obvious. Tar should be changed to reflect your desired default tape drive.

Good Luck

Thomas B. London
John F. Reiser

REGENERATING SYSTEM SOFTWARE

For UNIX/32V

Thomas B. London

John F. Reiser

Bell Laboratories
Holmdel, New Jersey 07733

Introduction

This document discusses how to assemble or compile various parts of the UNIX/32V† system software. This may be necessary because a command or library is accidentally deleted or otherwise destroyed; also, it may be desirable to install a modified version of some command or library routine. A few commands depend to some degree on the current configuration of the system; thus in any new system modifications to some commands are advisable. Most of the likely modifications relate to the standard disk devices contained in the system. For example, the df(1) ('disk free') command has built into it the names of the standardly present disk storage drives (e.g. '/dev/rf0', '/dev/rp0'). Df(1) takes an argument to indicate which disk to examine, but it is convenient if its default argument is adjusted to reflect the ordinarily present devices. The companion document 'Setting up UNIX' discusses which commands are likely to require changes.

Where Commands and Subroutines Live

The source files for commands and subroutines reside in several subdirectories of the directory /usr/src. These subdirectories, and a general description of their contents, are

cmd	Source files for commands.
libc/stdio	Source files making up the 'standard i/o package'.
libc/sys	Source files for the C system call interfaces.
libc/gen	Source files for most of the remaining routines described in section 3 of the manual.
libc/crt	Source files making up the C runtime support package, as in call save-return and long arithmetic.
libc/csu	Source for the C startup routines.
games	Source for (some of) the games. No great care has been taken to try to make it obvious how to compile these; treat it as a game.
libF77	Source for the Fortran 77 runtime library, exclusive of IO.
libI77	Source for the Fortran 77 IO runtime routines.
libdbm	Source for the 'data-base manager' package *dbm* (3).
libm	Source for the mathematical library.
libnm	Source for the assembler language mathematical library.
libplot	Source for plotting routines.

† UNIX is a trademark of Bell Laboratories.

Commands

The regeneration of most commands is straightforward. The 'cmd' directory will contain either a source file for the command or a subdirectory containing the set of files that make up the command. If it is a single file the command

```
cd /usr/src/cmd/Admin
Mk cmd name.c
```

suffices. (Cmd name is the name of the command you are playing with.) The result of the Mk command will be an executable version, copied to /bin (or perhaps /etc or other places if appropriate). If you want the result placed somewhere else, the command

```
cd /usr/src/cmd/Admin
DESTDIR=mydir Mk cmd name.c
```

where mydir is a full pathname of some destination directory (e.g. /usr/tbl/newcmds), will compile the command and place the result in mydir/bin (or perhaps mydir/etc or mydir/usr/bin, etc.).

If the source files are in a subdirectory there will be a 'makefile' (see make(1)) to control the regeneration. After changing to the proper directory (cd(1)) you type one of the following:

make	The program is compiled and loaded; the executable is left in the current directory.
make install	The program is compiled and loaded, and the executable is installed.
make clean	Everything is cleanup; for example .o files are deleted.

Some of the makefiles have other options. Print (cat(1)) the ones you are interested in to find out.

Alternately, to compiler and install a subdirectory command, one may perform the following

```
cd /usr/src/cmd/Admin
Mk cmd name
```

which combines all three of the above make options.

The Assembler

The assembler consists of one executable file: /bin/as. The source files for /bin/as are named '/usr/src/cmd/as/as?.c'. Considerable care should be exercised in replacing the assembler. Remember that if the assembler is lost, the only recourse is to replace it from some backup storage; a broken assembler cannot assemble itself.

The C Compiler

The C compiler consists of six routines: '/bin/cc', which calls the phases of the compiler proper, the compiler control line expander '/lib/cpp', the assembler ('as'), and the loader ('ld'). The C compiler proper is '/lib/ccom'; '/lib/c2' is the optional assembler-language optimizer. The loss of the C compiler is as serious as that of the assembler.

The source for /bin/cc resides in '/usr/src/cmd/cc.c'. Its loss alone (or that of c2) is not fatal. If needed, prog.c can be compiled by

```
/lib/cpp prog.c >temp0
/lib/ccom temp0 temp1
as temp3
ld /lib/crt0.o a.out -lc
```

The source for the compiler proper is in the directories /usr/src/cmd/mip and /usr/src/cmd/pcc. The /usr/src/cmd/mip directory contains files which are (relatively)

machine independent; the machine dependent files reside in the directory /usr/src/cmd/pcc. The compiler is 'made' by the makefile (see make(1)) in the directory /usr/src/cmd/pcc. To make a new /lib/ccom use

```
cd /usr/src/cmd/pcc
make
```

which produces the compiler (named /usr/src/cmd/pcc/comp). Before installing the new compiler, it is prudent to save the old one someplace.

In a similar manner, the optimizer phase of the C compiler (/lib/c2) is made up from the files c20.c, c21.c, and c22.c together with c2.h. Its loss is not critical since it is completely optional.

UNIX

The source and object programs for UNIX are kept in two subdirectories of */usr/src/sys*. In the subdirectory *h* there are several files ending in '.h'; these are header files which are picked up (via '#include ...') as required by each system module. The subdirectory *sys* is the rest of the system.

The file *conf.c* contains the tables which control device configuration of the system. *Locore.s* specifies the contents of the interrupt vectors, and all the machine-language code in the system.

To recreate the system, use

```
cd /usr/src/sys/sys
make unix
```

See 'Setting Up UNIX' for other information about configuration and such.

When the make is done, the new system is present in the current directory as 'unix'. It should be tested before destroying the currently running '/unix', this is best done by doing something like

```
mv /unix /ounix
mv unix /unix
```

If the new system doesn't work, you can still boot 'ounix' and come up (see boot(8)). When you have satisfied yourself that the new system works, remove /ounix.

To install a new device driver, copy its source to /usr/src/sys/sys, and edit the 'makefile' and the file 'loadall' if necessary (see make(1)).

Next, the I/O interrupt fielding mechanism must be modified to properly handle the new device. If the device is connected via the UNIBUS, then one only need add the device's interrupt handling routine address(s) in the proper position in the table 'UNIvec' in the file /usr/src/sys/sys/univec.c. 'UNIvec' should be modified by placing a pointer to a callout routine in the proper vector. Use some other device (like the DZ11) as a guide. Notice that the entries in 'UNIvec' must be in order. Bits 27-31 of the vector address will be available as the first argument of the interrupt routine. This stratagem is used when several similar devices share the same interrupt routine (as in dz11's).

If the device is connected via the MASSBUS, then /usr/src/sys/sys/univec.c is not to be modifed. Instead, code must be added to /usr/src/sys/sys/locore.s to actually transfer to the interrupt routine. Use the RP06 interrupt routine 'Xmba0int' in locore.s as a guide. As an aside, note that external names in C programs have an underscore (' ') prepended to them.

The second step which must be performed to add a new device is to add it to the configuration table /usr/src/sys/sys/conf.c. This file contains two subtables, 'bdevsw' and 'cdevsw', one for block-type devices, and one for character-type devices. Block devices include disks, and magtape. All other devices are character devices. A line in each of these tables gives all the information the system needs to know about the device handler; the ordinal

position of the line in the table implies its major device number, starting at 0.

There are four subentries per line in the block device table, which give its open routine, close routine, strategy routine, and device table. The open and close routines may be nonexistent, in which case the name 'nulldev' is given; this routine merely returns. The strategy routine is called to do any I/O, and the device table contains status information for the device.

For character devices, each line in the table specifies a routine for open, close, read, and write, and one which sets and returns device-specific status (used, for example, for stty and gtty on typewriters). If there is no open or close routine, 'nulldev' may be given; if there is no read, write, or status routine, 'nodev' may be given. Nodev sets an error flag and returns.

The final step which must be taken to install a device is to make a special file for it. This is done by mknod(1), to which you must specify the device class (block or character), major device number (relative line in the configuration table) and minor device number (which is made available to the driver at appropriate times).

The documents 'Setting up Unix' and 'The Unix IO system' may aid in comprehending these steps.

The Library libc.a

The library /lib/libc.a is where most of the subroutines described in sections 2 and 3 of the manual are kept. This library can be remade using the following commands:

```
cd /usr/src/libc
make libc.a
make install
make clean
```

If single routines need to be recompiled and replaced, use

```
cc -c -O x.c
ar vr /lib/libc.a x.o
rm x.o
```

The above can also be used to put new items into the library. See ar(1), lorder(1), and tsort(1).

The routines in /usr/src/cmd/libc/csu (C start up) are not in libc.a. These are separately assembled and put into /lib. The commands to do this are

```
cd /usr/src/libc
for i in csu/*.s
do
        j=`basename $i .s`
        as -o $j.o $i
        mv $j.o /lib
done
```

or, if you need only redo one routine,

```
cd /usr/src/libc/csu
as -o x.o x.s
mv x.o /lib
```

where x is the routine you want.

Other Libraries

Likewise, the directories containing the source for the other libraries have makefiles.

System Tuning

There are several tunable parameters in the system. These set the size of various tables and limits. They are found in the file /usr/sys/h/param.h as manifests ('#define's). Their values are rather generous in the system as distributed. Our typical maximum number of users is about 20, but there are many daemon processes.

When any parameter is changed, it is prudent to recompile the entire system, as discussed above. A brief discussion of each follows:

NBUF This sets the size of the disk buffer cache. Each buffer is 512 bytes. This number should be around 25 plus NMOUNT, or as big as can be if the above number of buffers cause the system to not fit in memory.

NFILE This sets the maximum number of open files. An entry is made in this table every time a file is 'opened' (see open(2), creat(2)). Processes share these table entries across forks (fork(2)). This number should be about the same size as NINODE below. (It can be a bit smaller.)

NMOUNT This indicates the maximum number of mounted file systems. Make it big enough that you don't run out at inconvenient times.

MAXMEM This specifies the number of page-frames of real memory at this installation. It is currently set at 1024 (512K bytes), and should be increased if you have more (otherwise the additional memory will not be utilized).

MAXUMEM This sets an administrative limit on the amount of memory a process may have. It is currently set at MAXMEM-128 (i.e. 896). It will be increased automatically by increasing MAXMEM. Note, however, that the current upper bound on MAXUMEM is 128*12 (i.e. 1536) which limits user process space to 768K bytes.

PHYSPAGES This indicates the number of pages which can be represented on the memory freelist. Its current value is 2048, and is sufficient for systems of up to one megabyte. If this value is too small (i.e. more memory than freelist) then system will only use PHYSPAGES page frames.

MAXUPRC This sets the maximum number of processes that any one user can be running at any one time. This should be set just large enough that people can get work done but not so large that a user can hog all the processes available (usually by accident!).

NPROC This sets the maximum number of processes that can be active. It depends on the demand pattern of the typical user; we seem to need about 8 times the number of terminals.

NINODE This sets the size of the inode table. There is one entry in the inode table for every open device, current working directory, sticky text segment, open file, and mounted device. Note that if two users have a file open there is still only one entry in the inode table. A reasonable rule of thumb for the size of this table is

NPROC + NMOUNT + (number of terminals)

SSIZE The initial size of a process stack. This may be made bigger if commonly run processes have large data areas on the stack.

SINCR The size of the stack growth increment.

NOFILE This sets the maximum number of files that any one process can have open. 20 is plenty.

CANBSIZ This is the size of the typewriter canonicalization buffer. It is in this buffer that erase and kill processing is done. Thus this is the maximum size of an input typewriter line. 256 is usually plenty.

SMAPSIZ The number of fragments that secondary (swap) memory can be broken into. This should be big enough that it never runs out. The theoretical maximum is twice the number of processes, but this is a vast overestimate in practice. 70 seems enough.

NCALL This is the size of the callout table. Callouts are entered in this table when some sort of internal system timing must be done, as in carriage return delays for terminals. The number must be big enough to handle all such requests.

NTEXT The maximum number of simultaneously executing pure programs. This should be big enough so as to not run out of space under heavy load. A reasonable rule of thumb is about

(number of terminals) + (number of sticky programs)

NCLIST The number of clist segments. A clist segment is 12 characters. NCLIST should be big enough so that the list doesn't become exhausted when the machine is busy. The characters that have arrived from a terminal and are waiting to be given to a process live here. Thus enough space should be left so that every terminal can have at least one average line pending (about 30 or 40 characters).

TIMEZONE The number of minutes westward from Greenwich. See 'Setting Up UNIX'.

DSTFLAG See 'Setting Up UNIX' section on time conversion.

MSGBUFS The maximum number of characters of system error messages saved. This is used as a circular buffer.

NCARGS The maximum number of characters in an exec(2) arglist. This number controls how many arguments can be passed into a process. 5120 is practically infinite.

HZ Set to the desired frequency of the system clock (e.g., 50 for a 50 Hz. clock). The current value is 60 (i.e. 60 Hz. clock).

A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

1. Purpose

The widespread use of the UNIX† system[1] within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back an forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only.[1,2] Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes; in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there; this makes

† UNIX is a trademark of Bell Laboratories.

a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud; hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.[3,4] Thus, systems typically join our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustratives the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

3. Processing

The user has two commands which set up communications, *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into

the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp* or *uux* programs,

c) by a remote system.

Scan For Work

The file names in the spool directory are constructed to allow the daemon programs *(uucico, uuxqt)* to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number,

[6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same "phone number" to be stored at every site, despite local variations in telephone services and dialing conventions.

A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the uucp transmission program. Other protocols may be added by individual installations.

Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

S	send a file,
R	receive a file,
C	copy complete,
H	hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, a *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

4. Present Uses

One application of this software is remote mail. Normally, a UNIX system user writes "mail dan" to send mail to user "dan". By writing "mail usg!dan" the mail is sent to user "dan" on system "usg".

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a "stockroom" which allows remote users to call in

and request software. He keeps a "stock list" of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's "diff" program which compares two text files and indicates the differences, line by line, between them.[5] Only lines which are not identical are printed. Similarly, the program "uudiff" compares files (or directories) on two machines. One of these directories may be on a passive system. The "uudiff" program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The "uux" command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

Nominal speed	Characters/sec.
300 baud	27
1200 baud	100-110
9600 baud	200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.

The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages.[6] There are distribution errors and delays, resulting in old software running at remote sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system "mail" and "diff," plus the many implied commands represented by "uux." However, we still need inter-system "write" (real-time inter-user communication) and "who" (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.
2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.
3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.
4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver and protocol, and A. S. Cohen, J. Lions, and P. F. Long for their suggestions and assistance.

References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1905-1929, 1978.
2. G. L. Chesson, "The Network UNIX System," *Operating Systems Review*, vol. 9, no. 5, pp. 60-66, 1975. Also in *Proc. 5th Symp. on Operating Systems Principles.*
3. A. G. Fraser, "Spider — An Experimental Data Communications System," *Proc. IEEE Conf. on Communications*, p. 21F, June 1974. IEEE Cat. No. 74CH0859-9-CSCB.
4. A. G. Fraser, "A Virtual Channel Network," *Datamation*, pp. 51-56, February 1975.
5. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey, June 1976.
6. F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, Mass., 1975.

Uucp Implementation Description

D. A. Nowitz

Introduction

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution. The first version of the system was designed and implemented by M. E. Lesk.[1] This paper describes the current (second) implementation of the system.

Uucp is a batch type operation. Files are created in a spool directory for processing by the uucp demons. There are three types of files used for the execution of work. *Data files* contain data for transfer to remote systems. *Work files* contain directions for file transfers between systems. *Execution files* are directions for UNIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

uucp — This program creates work and gathers data files in the spool directory for the transmission of files.

uux — This program creates work files, execute files and gathers data files for the remote execution of UNIX commands.

uucico — This program executes the work files for data transmission.

uuxqt — This program executes the execution files for UNIX command execution.

The secondary programs are:

uulog — This program updates the log file with new entries and reports on the status of uucp requests.

uuclean — This program removes old files from the spool directory.

The remainder of this paper will describe the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

1. Uucp - UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The *uucp* command was designed to look like *cp* to the user. The syntax is

uucp [option] ... source ... destination

where the source and destination may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

−d — Make directories when necessary for copying the file.

† UNIX is a trademark of Bell Laboratories.

1 M. E. Lesk and A. S. Cohen, UNIX Software Distribution by Communication Link, private communication.

−c Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.

−g*letter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)

−m Send mail on completion of the work.

The following options are used primarily for debugging:

−r Queue the job but do not start *uucico* program.

−s*dir* Use directory *dir* for the spool directory.

−x*num* *Num* is the level of debugging output desired.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as "*?*[]*". If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

uucp *.c usg!/usr/dan

will set up the transfer of all files whose names end with ".c" to the "/usr/dan" directory on the"usg" machine.

The source and/or destination names may also contain a ~*user* prefix. This translates to the login directory on the specified system. For names with partial path-names, the current directory is prepended to the file name. File names with ../ are not permitted.

The command

uucp usg!~dan/*.h ~dan

will set up the transfer of files whose names end with ".h" in dan's login directory on system "usg" to dan's local login directory.

For each source file, the program will check the source and destination file-names and the system-part of each to classify the work into one of five types:

[1] Copy source to destination on local system.

[2] Receive files from other systems.

[3] Send files to a remote systems.

[4] Send files from remote systems to another remote system.

[5] Receive files from remote systems when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the −r option was specified).

Type 1

A *cp* command is used to do the work. The −*d* and the −*m* options are not honored in this case.

Type 2

A one line *work file* is created for each file requested and put in the spool directory with the following fields, each separated by a blank. (All *work files* and *execute files* use a blank as the field separator.)

[1] R

[2] The full path-name of the source or a ~user/path-name. The ~*user* part will be expanded on the remote system.

[3] The full path-name of the destination file. If the ~*user* notation is used, it will be immediately expanded to be the login directory for the user.

[4] The user's login name.

[5] A "−" followed by an option list. (Only the −m and −d options will appear in this list.)

Type 3

For each source file, a *work file* is created and the source file is copied into a *data file* in the spool directory. (A "−c" option on the *uucp* command will prevent the *data file* from being made.) In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

[1] S

[2] The full-path name of the source file.

[3] The full-path name of the destination or ~user/file-name.

[4] The user's login name.

[5] A "−" followed by an option list.

[6] The name of the *data file* in the spool directory.

[7] The file mode bits of the source file in octal print format (e.g. 0666).

Type 4 and Type 5

Uucp generates a *uucp* command and sends it to the remote machine; the remote *uucico* executes the *uucp* command.

2. Uux - UNIX To UNIX Execution

The *uux* command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of the uux command is

uux [−] [option] ... command-string

where the command-string is made up of one or more arguments. All special shell characters such as "<>|^" must be quoted either by quoting the entire command-string or quoting the character as a separate argument. Within the command-string, the command and file names may contain a *system-name!* prefix. All arguments which do not contain a "!" will not be treated as files. (They will not be copied to the execution machine.) The "−" is used to indicate that the standard input for *command-string* should be inherited from the standard input of the *uux* command. The options, essentially for debugging, are:

−r Don't start *uucico* or *uuxqt* after queuing the job;

−x*num* Num is the level of debugging output desired.

The command

pr abc | uux − usg!lpr

will set up the output of "pr abc" as standard input to an lpr command to be executed on system "usg".

Uux generates an *execute file* which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed by the *uucico* program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The *execute file* will be processed by the *uuxqt* program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

User Line

U user system

where the *user* and *system* are the requester's login name and system.

Required File Line

F file-name real-name

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the *execute file*. The *uuxqt* program will check for the existence of all required files before the command is executed.

Standard Input Line

I file-name

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the *uux* command if the "−" option is used. If a standard input is not specified, "/dev/null" is used.

Standard Output Line

O file-name system-name

The standard output is specified by a ">" within the command-string. If a standard output is not specified, "/dev/null" is used. (Note − the use of ">>" is not implemented.)

Command Line

C command [arguments] ...

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All *required files* will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command is executed using the Shell specified in the *uucp.h* header file. In addition, a shell "PATH" statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

3. Uucico - Copy In, Copy Out

The *uucico* program will perform the following major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp, uux, uuxqt* or *uucico* programs,

c) directly by the user (this is usually for testing),

d) by a remote system. (The uucico program should be specified as the "shell" field in the "/etc/passwd" file for the "uucp" logins.)

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If no system name is specified (−s option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

−r1 Start the program in *MASTER* mode. This is used when *uucico* is started by a program or "cron" shell.

−s*sys* Do work only for system *sys*. If −*s* is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

−d*dir* Use directory *dir* for the spool directory.

−x*num* *Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the *uucico* program.

Scan For Work

The names of the work related files in the spool directory have format

type . system-name grade number

where:

Type is an upper case letter, (*C* - copy command file, *D* - data file, *X* - execute file);

System-name is the remote system;

Grade is a character;

Number is a four digit, padded sequence number.

The file

C.res45n0031

would be a *work file* for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for *work files* (files with prefix "C."). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* file. The information contained for each system is;

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number if field [3] is *ACU* or the device name (same as field [3]) if not *ACU,*

[6] login information (multiple fields),

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins. The *SLAVE* can also reply with a "call-back required" message in which case, the current conversation is terminated.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where proto-list is a string of characters, each representing a line protocol.

The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or *N* which means there is no common protocol.

Work Processing

The initial roles (*MASTER* or *SLAVE*) for the work processing are the mode in which each program starts. (The *MASTER* has been specified by the "−r1" uucico option.) The *MASTER* program does a work search similar to the one used in the "Scan For Work" section.

There are five messages used during the work processing, each specified by the first character of the message. They are;

S send a file,

R receive a file,

C copy complete,

X execute a *uucp* command,

H hangup.

The *MASTER* will send *R*, *S* or *X* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY, SN, RY, RN, HY, HN, XY, XN, corresponding to yes or no for each request.*

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a *CN* message is sent. (In the

case of *CN*, the transferred file will be in the spool directory with a name beginning with "TM'.) The requests and results are logged on both systems.

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, an *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other. The original *SLAVE* program will clean up and terminate. The *MASTER* will proceed to call other systems and process work as long as possible or terminate if a *−s* option was specified.

4. Uuxqt - Uucp Command Execution

The *uuxqt* program is used to execute *execute files* generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the spool directory for *execute files* (prefix "X."). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The *execute file* is described in the "Uux" section above.

Command Execution

The execution is accomplished by executing a *sh −c* of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

5. Uulog - Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to prepend these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

−s*sys* Print entries where *sys* is the remote system name;

−u*user* Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

6. Uuclean - Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

−d*dir* The directory to be scanned is *dir*.

−m Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the uucp programs.)

−n*hours* Change the aging time from 72 hours to *hours* hours.

−p*pre* Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)

−x*num* This is the level of debugging output desired.

7. Security

The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

- The login for uucp does not get a standard shell. Instead, the *uucico* program is started. Therefore, the only work that can be done is through *uucico*.
- A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require callback for certain login-ids. (See the "Files required for execution" section for the file description.)
- A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.
- The *uuxqt* program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specifed in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (Programs uucp, uucico, uux, uuxqt should be also owned by uucp and have the setuid bit set.)

8. Uucp Installation

There are several source modifications that may be required before the system programs are compiled. These relate to the directories used during compilation, the directories used during execution, and the local *uucp system-name.*

The four directories are:

lib	(/usr/src/cmd/uucp) This directory contains the source files for generating the *uucp* system.
program	(/usr/lib/uucp) This is the directory used for the executable system programs and the system files.
spool	(/usr/spool/uucp) This is the spool directory used during *uucp* execution.
xqtdir	(/usr/spool/uucp/.XQTDIR) This directory is used during execution of *execute files*.

The names given in parentheses above are the default values for the directories. The italicized named *lib, program, xqtdir,* and *spool* will be used in the following text to represent the appropriate directory names.

There are two files which may require modification, the *makefile* file and the *uucp.h* file. The following paragraphs describe the modifications. The modes of *spool* and *xqtdir* should be made "0777".

Uucp.h modification

Change the *program* and the *spool* names from the default values to the directory names to be used on the local system using global edit commands.

Change the *define* value for *MYNAME* to be the local *uucp* system-name.

makefile modification

There are several *make* variable definitions which may need modification.

INSDIR This is the *program* directory (e.g. INSDIR=/usr/lib/uucp). This parameter is used if "make cp" is used after the programs are compiled.

IOCTL This is required to be set if an appropriate *ioctl* interface subroutine does not exist in the standard "C" library; the statement "IOCTL=ioctl.o" is required in this case.

PKON The statement "PKON=pkon.o" is required if the packet driver is not in the kernel.

Compile the system The command

```
make
```

will compile the entire system. The command

```
make cp
```

will copy the commands to the to the appropriate directories.

The programs *uucp*, *uux*, and *uulog* should be put in "/usr/bin". The programs *uuxqt*, *uucico*, and *uuclean* should be put in the *program* directory.

Files required for execution

There are four files which are required for execution, all of which should reside in the *program* directory. The field separator for all files is a space unless otherwise specified.

L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp*. The special device files are assumed to be in the */dev* directory. The format for each entry is

```
line  call-unit  speed
```

where;

line is the device for the line (e.g. cul0),

call-unit is the automatic call unit associated with *line* (e.g. cua0), (Hardwired lines have a number "0" in this field.),

speed is the line speed.

The line

```
cul0  cua0  300
```

would be set up for a system which had device cul0 wired to a call-unit cua0 for use at 300 baud.

L-dialcodes

This file contains entries with location abbreviations used in the *L.sys* file (e.g. py, mh, boston). The entry format is

```
abb  dial-seq
```

where;

abb is the abbreviation,

dial-seq is the dial sequence to call that location.

The line

py 165–

would be set up so that entry py7777 would send 165–7777 to the dial-unit.

LOGIN/SYSTEM NAMES

It is assumed that the *login name* used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique *system name* which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

USERFILE

This file contains user accessibility information. It specifies four types of constraint;

[1] which files can be accessed by a normal user of the local machine,

[2] which files can be accessed from a remote computer,

[3] which login name is used by a particular remote computer,

[4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format

login,sys [c] path-name [path-name] ...

where;

login	is the login name for a user or the remote computer,
sys	is the system name for a remote computer,
c	is the optional *call-back required* flag,
path-name	is a path-name prefix that is acceptable for *user*.

The constraints are implemented as follows.

[1] When the program is obeying a command stored on the local machine, *MASTER* mode, the path-names allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

[2] When the program is responding to a command from a remote machine, *SLAVE* mode, the path-names allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a *null* system name is used.

[3] When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name.

[4] If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

u,m /usr/xyz

allows machine *m* to login with name *u* and request the transfer of files whose names start with "/usr/xyz".

The line

dan, /usr/dan

allows the ordinary user *dan* to issue commands for files whose name starts with "/usr/dan".

The lines

```
u,m /usr/xyz  /usr/spool
u,  /usr/spool
```

allows any remote machine to login with name *u*, but if its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool".

The lines

```
root, /
,  /usr
```

allows any user to transfer files beginning with "/usr" but the user with login *root* can transfer any file.

L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

system name

The name of the remote system.

time

This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800–1730).

The day portion may be a list containing some of

Su Mo Tu We Th Fr Sa

or it may be *Wk* for any week-day or *Any* for any day.

The time should be a range of times (e.g. 0800–1230). If no time portion is specified, any time of day is assumed to be ok for the call.

device

This is either *ACU* or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

speed

This is the line speed for the call (e.g. 300).

phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. mh5900, boston995–9980).

For the hardwired devices, this field contains the same string as used for the *device* field.

login

The login information is given as a series of fields and subfields in the format

expect send [expect send] ...

where; *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect[−send−expect]...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a BREAK character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the L.sys file would be

sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm looks at the last part of the string as illustrated in the password field.

9. Administration

This section indicates some events and files which must be administered for the *uucp* system. Some administration can be accomplished by *shell files* which can be initiated by *crontab* entries. Others will require manual intervention. Some sample *shell files* are given toward the end of this section.

SQFILE − sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

TM − temporary data files

These files are created in the *spool* directory while files are being copied from a remote machine. Their names have the form

TM.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

uuclean −pTM

will remove all *TM* files older than three days.

LOG − log entry files

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command

uulog

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new LOG entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

STST – system status files

These files are created in the spool directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a *TALKING* status when to machines are conversing. The form of the file name is

STST.sys

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a *TALKING* status. In this case, the file must be removed before a conversation is attempted.

LCK – lock files

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

LCK..str

where *str* is either a device or system name. The files may be left in the spool directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but the starting of *uucico* will sometimes fail. (No devices available, login failures etc.). Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a "shell" file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

program/uulog
program/uucico −r1

Note that the "−r1" option is required to start the *uucico* program in *MASTER* mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C.* or *D.* files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

program/uuclean −pTM −pC. −pD.
program/uuclean −pST −pLCK −n12

can be used. Note the "−n12" option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the "−n" option will use a three day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILE*s. A shell like

cp *spool*/LOGFILE *spool*/o.LOGFILE
rm *spool*/LOGFILE

can be used.

Login Entry

One or more logins should be set up for *uucp*. Each of the "/etc/passwd" entries should have the "*program*/uucico" as the shell to be executed. The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the *shell* argument limits the login to the use of uucp (*uucico*) only.

File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp*, *uux*, *uucico* and *uuxqt* should be owned by the *uucp* login with the "setuid" bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard *shell* for the *uucp* logins.

The *L.sys*, *SQFILE* and the *USERFILE* which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

UNIX MASTER INDEX

The **UNIX Master Index** is a cumulative index; it brings together the indexes of all the UNIX volumes. The Master Index appears at the end of each volume.

Each entry is followed by one or more shortened volume titles, indicating the volumes in which the topic is discussed and the pages containing the information. The volumes and their shortened titles are shown in the following table:

Shortened	**Volume Title**
General use	*GEN*
Programming	*PGM*
System manager	*SYS*

If a topic is discussed in two or more volumes, the shortened volume names are presented in alphabetical order. For example, an entry in the Master Index might appear in the following way:

ed line editor

description, *GEN* 4-8 to 4-9, *SYS* 4-6

ed__.hup file

saving text, *GEN* 2-6

This entry indicates that a description of the ed line editor can be found on pages 4-8 through 4-9 of the *GEN* volume and page 4-6 of the *SYS* volume. The ed__.hup file is discussed on page 3-43 of the *GEN* volume.

ACRONYMS AND MNEMONICS

The acronym (or mnemonic) is the preferred entry. The acronym is cross-referred from the complete form.

DEFINITIONS

Defined terms and glossary terms are indexed.

HOMONYMS

Things of the same name but different meaning are followed by a descriptive word or by an abbreviation in parentheses.

KEYS FOR EXAMPLES, FIGURES, TABLES, AND FOOTNOTES

Page references for example, figure, and table index entries are keyed. Example:

Example	4-13E
Figure	4-13F
Table	4-13T
Footnote	4-13n

NONALPHABETIC CHARACTERS

Entries containing leading nonalphabetic characters (symbols, numbers, and punctuation) are placed at the beginning of the index. Nonalphabetic characters within index entries are sorted before alphabetic characters.

Nonalphabetic characters that serve as terms are indexed in a spelled-out form whenever possible.

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Notes:

Notes: