Ptask Library: A Quick Guide

Giorgio Buttazzo and Giuseppe Lipari
RETIS Lab - *Scuola Superiore Sant'Anna*

Version 0.4

Summary

Ptask is a C-library for fast development of periodic and aperiodic real-time tasks under Linux. It is written on top of the **Pthread** library with the aim of simplifying the creation of threads with typical timing parameters, like periods and deadlines. Ptask functions allow programmers to quickly

- create periodic and aperiodic tasks;
- specify timing constraints such periods and relative deadlines;
- monitor deadline misses;
- monitor average and worst-case execution times;
- activate tasks with specific offsets;
- manage task groups;
- handle mode changes.

Table of contents

This document includes the following sections:

1. New data types.
2. System functions
3. Task functions
4. Measuring execution times
5. Handling mode changes
6. Examples

1. New data types

The following new types are defined in the ptask library:

ptime	this is the type used for the time variables. It is basically a shortcut for a **long** integer.
tspec	this type is used for specifying a precise time, and it is used by the library for internal time representation. It is a shortcut for **struct timespec**. The Ptask library provides appropriate functions to operate on **tspec** and to convert a **ptime** into a **tspec**, and viceversa.
tpars	this type of structure is used to store all task parameters and it is initialized at task creation.

ptask this type is used for defining the task code. It is a shortcut for **void**.

2. System functions

```
void   ptask_init(int scheduler, int schedtype, int protocol);
```

Initializes the **ptask** library, resets the system time, set the scheduler for all the tasks and the resource access protocol for all the semaphores.

`scheduler`	can be SCHED_OTHER, SCHED_FIFO, SCHED_RR, SCHED_DEADLINE.
`schedtype`	can be PARTITIONED or GLOBAL, and it is only useful for multicore systems.
`protocol`	can be NO_PROTOCOL for classical semaphores, INHERITANCE for Priority Inheritance, or CEILING for Immediate Priority Ceiling.

Note (from version 0.4): in order to use SCHED_DEADLINE in PARTITIONED mode, the user has first to disable the admission control in Linux. This can be done by invoking the script no-admission.sh (included in the distribution) with root privileges.

```
ptime  ptask_gettime(int unit);
```

Returns the current time (from the system start time) in the specified `unit`, which can be SEC, MILLI, MICRO, or NANO.

```
int    ptask_getnumcores();
```

Returns the number of available cores in the system.

3. Task functions

The Ptask library maintains a Task Control Block (TCB) for every task, used to store the task state, the current task parameters, and some information collected about the task during its execution. The content of the TCB is for internal use only, therefore it is not reported in this manual. Please refer to the source code for more information.

```
int    ptask_create(void (*body)(void), int period, int prio, int flag);
```

Creates a concurrent task and returns the task index that can be used to differentiate multiple instances of the same task. The arguments have the following meaning:

`body`	is the name of the function containing the task body;
`period`	specifies the task period (equal to the relative deadline) in milliseconds.
`prio`	specifies the task priority between 1 (low) and 99 (high);
`flag`	specifies the activation mode of the task (NOW or DEFERRED): if set to NOW, the task is immediately activated; if set to DEFERRED, the task will block on the **wait_for_activation**() until a **ptask_activate**() is invoked by another task.

If task creation cannot be performed or an error occurs, the function returns the value -1.

Note (from version 0.4): this function has been deprecated since 0.4 because it is not compatible with the SCHED_DEADLINE scheduler.

```
int ptask_create_prio(void (*body)(void), int period, int prio,
                      int aflag);
```

Creates a concurrent task and returns the task index that can be used to differentiate multiple instances of the same task. This function is supposed to be used with fixed priority scheduling (SCHED_FIFO or SCHED_RR) only. The arguments have the following meaning:

`body`	is the name of the function containing the task body;
`period`	specifies the task period (equal to the relative deadline) in milliseconds.
`prio`	specifies the task priority between 1 (low) and 99 (high);
`flag`	specifies the activation mode of the task (NOW or DEFERRED): if set to NOW, the task is immediately activated; if set to DEFERRED, the task will block on the **wait_for_activation**() until a **ptask_activate**() is invoked by another task.

If task creation cannot be performed or an error occurs, the function returns the value -1.

Note: since version 0.4

```
int  ptask_create_edf(void (*task)(void), int period, int runtime,
                      int dline, int aflag);
```

Creates a concurrent task for the SCHED_DEADLINE scheduler, and returns the task index that can be used to differentiate multiple instances of the same task. The arguments have the following meaning:

`body`	is the name of the function containing the task body;
`period`	specifies the task period (equal to the relative deadline) in milliseconds;
`runtime`	specifies the budget of the task in milliseconds, it must be less than the period.;
`dline`	specifies the relative deadline of the task in milliseconds;
`flag`	specifies the activation mode of the task (NOW or DEFERRED): if set to NOW, the task is immediately activated; if set to DEFERRED, the task will block on the **wait_for_activation**() until a **ptask_activate**() is invoked by another task.

If task creation cannot be performed or an error occurs, the function returns the value -1.

In the current version of Linux (4.4), there is no way to create a task so that it is immediately scheduled by SCHED_DEADLINE: the only possibility is to create the task with some other scheduler (for example: SCHED_OTHER), and then modify its scheduling parameters once the task has been created.

Since ptask uses the Linux API, function ptask_create_edf() will create the task in SCHED_OTHER; when the task starts executing, it will change its scheduling parameters to SCHED_DEADLINE. Therefore, it is recommended to leave the possibility for the task to get a chance to execute and go into SCHED_DEADLINE during the initialization phase, for example by setting its activation flag to DEFERRED, and the activate the task with a certain offset in the future.

Also, if the SCHED_DEADLINE parameters cannot be set, the user will not receive any error at the time of task creation, but the error will be communicated later when the task effectively tries to set its scheduling parameters.

```
void   ptask_activate(int tid);
```

Activates the task with index `tid`.

```
void    ptask_activate_at(int tid, ptime t);
```

Activates the task with index `tid` at the absolute time `t`. If `t` has already passed, the task is immediately activated.

```
void    ptask_wait_for_period();
```

It suspends the calling task until the beginning of its next period. The typical usage of this call in a task body is shown in Figure 1.

```
void ptask_wait_for_activation();
```

It suspends the calling task until an explicit activation is invoked by another task. The typical usage of this call in a task body is shown in Figure 2.

```
ptask    my_periodic_task()
{
int      i;

         i = ptask_get_index();

         while (1) {

                <do useful things as a function of i>

                ptask_wait_for_period();
         }
}
```

Figure 1: General structure of a periodic task.

```
ptask    my_aperiodic_task()
{
int      i;

         i = ptask_get_index();
         ptask_wait_for_activation();

         while (1) {

                <do useful things as a function of i>

                ptask_wait_for_activation();
         }
}
```

Figure 2: General structure of an aperiodic task.

The example illustrated in Figure 3 shows how to define a periodic task that starts executing upon an explicit activation.

```
ptask   activated_periodic_task()
{
int     i;

        i = ptask_get_index();
        ptask_wait_for_activation();

        while (1) {

               <do useful things as a function of i>

               ptask_wait_for_period();
        }
}
```

Figure 3: General structure of a periodic task with an explicit activation.

The following functions are used to obtain and modify the parameters of a running task.

```
int     ptask_get_index();
```

Returns the index of the calling task.

```
int     ptask_get_priority(int i);
```

Returns the priority of the task with index i.

```
void    ptask_set_priority(int i, int prio);
```

Sets the priority of the task with index i to the value specified by `prio`, which must be a value between 1 (the lowest priority) and 99 (the highest priority).

```
int     ptask_get_period(int i, int units);
```

Returns the period of the task with index `i` (in `units`).

```
void    ptask_set_period(int i, int myper, int units);
```

Sets the period of the task with index `i` to `myper` (in `units`).

```
ptime   ptask_get_deadline(int i, int units);
```

Returns the relative deadline of the task with index `i` (in `units`).

```
void    ptask_set_deadline(int i, int mydline, int units);
```

Sets the relative deadline of the task with index `i` to `mydline` (in `units`).

```
int    ptask_deadline_miss();
```

Returns 1 if the current time is greater than the absolute deadline of the current job, 0 otherwise.

```
int    ptask_migrate_to(int i, int core_id);
```

Moves the task with index `i` to the core specified by `core_id`.

More specific parameters can be passed to a task at creation time through the following structure:

```
typedef  struct {
         tspec      runtime;        // task budget (only for SCHED_DEADLINE)
         tspec      period;         // task period (in ms)
         tspec      rdline;         // relative deadline (in ms)
         int        priority;       // from 1 (low) to 99 (high) (not used in SCHED_DEADLINE)
         int        processor;      // processor where task should be allocated
         int        act_flag;       // activation flag (NOW, DEFERRED)
         int        measure_flag;   // enable/disable exec. time measurements
         void       *arg;           // pointer to a task argument
         rtmode_t   *modes;         // pointer to the mode handler
         int        mode_list[RTMODE_MAX_MODES];      // the maximum number of modes
         int        nmodes;         // num of modes in which the task is active
} tpars;
```

In particular:

runtime	specifies the budget of the task (only for SCHED_DEADLINE)
period	specifies the task period in ms;
rdline	specifies the task relative deadline in ms (by default it is set equal to period);
priority	specifies the task priority between 1 (low) and 99 (high); this is only used for fixed priority scheduling (SCHED_FIFO and SCHED_RR);
processor	specifies the processor where the task has to be allocated (default value is 0);
act_flag	if set to NOW, the task is immediately activated, if set to DEFERRED (default value), the task will block on the **wait_for_activation**() until a **ptask_activate**() is invoked by another task;
measure_flag	if set to a non-zero the library automatically profiles the execution time of the task;
arg	pointer to a memory area used to pass arguments to the task; the structure and the content of such a memory are user-defined;
modes	used to manage mode changes (see Section 5);
nmodes	number of modes of the task;
mode_list	list of task modes.

Such parameters can be set either directly or by using the following functions (for efficiency reasons, and following a common practice in C programming, these functions are actually implemented as macros).

```
void    ptask_param_init(tpars tp);
```

Initializes the task parameters in the `tp` structure with the default values.

```
void    ptask_param_period(tpars tp, int myper, int units);
```

Initializes the task period in the `tp` structure with the value specified by `myper` expressed in given `units` (SEC, MILLI, MICRO, or NANO).

```
void    ptask_param_deadline(tpars tp, int mydline, int units);
```

Initializes the task relative deadline in the `tp` structure with the value specified by `mydline` expressed in given `units` (SEC, MILLI, MICRO, or NANO).

```
void    ptask_param_priority(tpars tp, int myprio);
```

Initializes the task priority in the `tp` structure with the value specified by `myprio`, which must be a value between 1 (the lowest priority) and 99 (the highest priority).

```
void    ptask_param_activation(tpars tp, int myact);
```

Initializes the task activation mode in the `tp` structure with the value specified by `myact`, which can be either NOW, for immediately activation, or DEFERRED; in this case the task will block on the **wait_for_activation**() until a **ptask_activate**() is invoked by another task.

```
void    ptask_param_processor(tpars tp, int proc_id);
```

Specifies the index of the processor on which the task is supposed to run. Note that this is valid only if the PARTITIONED strategy has been set by **ptask_init()**. This call has no effect when the scheduling strategy is set to GLOBAL.

```
void    ptask_param_measure(tpars tp);
```

Sets the measuring flag to 1, so enabling **ptask** to automatically profile the execution time of the task.

```
void    ptask_param_argument(tpars tp, void *arg);
```

Passes to the `tp` structure the user-defined arguments pointed by `arg`.

```
void    ptask_param_modes(tpars tp, rtmode *modes, int nmodes);
```

Allows specifying the set of execution modes in the task that is going to be created. The argument `modes` is a pointer to a structure that defines the system modes and **`nmodes`** specifies the number of modes in which this task is going to be active.

```
void    ptask_param_mode_add(tpars tp, int mode_num);
```

Specifies that the current task is active in mode `mode_num`. See Section 5 for an example of use of this function.

Once all the specific parameters are set, the task can be created using the following function.

```
int     ptask_create_param(void (*body)(void), tpars *tp);
```

Creates a concurrent task and returns the task index that can be used to differentiate multiple instances of the same task. The arguments have the following meaning:

body	is the name of the function containing the task body;
tp	is a pointer to the task parameter structure; if tp is set to NULL, then the task is created with the following default values:

type:	APERIODIC
period	1000 ms
rdline	1000 ms
priority	1
act_flag	DEFERRED
processor	0
measure	NO
arg	NULL
modes	NULL
nmodes	0
mode_list	an empty array

4. Measuring execution times

To measure the execution time of a task it is necessary to set its `measure_flag` when the task is created. Then, after the task has completed its execution, it is possible to obtain its execution time by calling the following functions.

WARNING: calling these functions while the task is executing may give inconsistent values, because the internal data structures are not protected by semaphores for containing the overhead. Therefore, it is up to the user to make sure that the task is no executing before calling this function.

```
tspec   ptask_stat_getwcet(int i)
```

Returns the maximum execution time among all the jobs of task `i` since its first activation.

```
tspec   ptask_stat_getavg(int i)
```

Returns the average execution time among all the jobs of task `i` since its first activation.

```
int     ptask_stat_getnuminstances(int i)
```

Returns the number of jobs of task `i` activated since its creation.

```
tspec   ptask_stat_gettotal(int i)
```

Returns the total execution time consumed by all the jobs of task `i` since its creation.

5. Handling mode changes

The Ptask library allows the user to specify a set of execution modes for the whole system and handle mode changes transparently through the following functions.

```
int     rtmode_init(rtmode_t *g, int nmodes);
```

Initializes the mode manager and the data structure pointed by `g`, which will contain the task groups involved in each mode. The second parameter nmodes is the total number of system modes.

```
void    rtmode_changemode(rtmode_t *g, int new_mode_id);
```

This function has to be called every time we want the system to perform a mode change. The parameter `new_mode_id` specifies the new mode in which the system must switch. The mode change is performed by an internal mode manager that executes the mode change protocol. In the current implementation, before activating the tasks involved in the new mode, the mode manager waits for the largest absolute deadline of those tasks that are only present in the old mode. An example of mode change is shown in Figure 4.

```
#define    MODE_OFF      0
#define    MODE_ON       1
```

```
#define     MODE_FAIL    2

ptask       taskbody()
{
            ptask_wait_for_activation();

            while (1) {
                  printf("Task T%d is running\n", ptask_get_index());
                  ptask_wait_for_period();
            }
}

int         main()
{
rtmode_t    mymodes;
tpars       param;
int         res;

            ptask_init(SCHED_FIFO, GLOBAL, PRIO_INHERITANCE);
            res = rtmode_init(&mymodes, 3);     // System with 3 modes

            ptask_param_init(param);
            ptask_param_period(param, 1, SEC);
            ptask_param_priority(param, 4);

            // this task is present in two modes
            ptask_param_modes(param, &mymodes, 2);
            // the task is present in mode MODE_ON
            ptask_param_mode_add(param, MODE_ON);
            // The task is present in mode MODE_FAIL
            ptask_param_mode_add(param, MODE_FAIL);
            // The task is NOT present in MODE_OFF
            res = ptask_create_param(taskbody, &param);

            // create the other tasks in a similar way

            rtmode_changemode(&mymodes, MODE_OFF);   // set initial mode

            if (condition)
                  // activates MODE_ON; all tasks not in this mode
                  // are suspended; after that, the tasks in MODE_ON
                  // not already active, are activated
                  rtmode_changemode(&mymodes, MODE_ON);
            else if (error_condition)
                  // activates MODE_FAIL; all tasks not in this mode
                  // are suspended; after that, the tasks in MODE_FAIL
                  // not yet active, are activated
                  rtmode_changemode(&mymodes, MODE_FAIL);
            ...
}
```

Figure 4: example of mode change.