Ptask Library: A Quick Guide

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

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. Main functions allow the programmer to quickly

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

Table of contents

1. New 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 ,ore precise time, and it is used by the library for internal representation. It is a shortcut for **struct timespec**. Appropriate helper functions allow to convert a ptime into a tspec, and viceversa, and to operate on tspec.
tpars	this is the type of structure used to store all task parameters, used at task creation
ptask	this is the type used for defining the code of a task. It is a shortcut for **void**.

2. System functions

void ptask_init(int scheduler, int partioning, 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[1]. Partioning 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.

ptime ptask_get_time(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_get_numcores();

Returns the number of available cores in the system.

3. Task functions

The library maintains internally a Task Control Block (TCB)for every task, to store the state of the task, the current task parameters, and collect information about the task. The contents of the TCB are for internal use and are subject to change, therefore we will not report them here. Please refer to the source code for more information.

int ptask_create(void (*body)(void), int period, int priority);

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.
priority	specifies the task priority (1-low, 99-high);

If it is not possible to create the task, or if an error occurred, the function returns -1[2]. The task index is used by many other functions of the library, therefore the user should check and store the return of this function.

void ptask_activate(int tid);

Activates the task with index `tid`.

voidptask_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.

1 A version with support for EDF is under preparation.

2 A more complete support for error treatment and error codes is planned for the next version of the library.

voidptask_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.

voidptask_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_perdiodic_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_aperdiodic_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    my_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.

4. Additional task parameters and full creation mode

More specific parameters can be passed through the following structure:

```
typedef struct {
   tspec period;
   tspec rdline;
```

```
    int priority;             /* from 0 to 99                                 */
    int processor;            /* processor id                                 */
    int act_flag;             /* NOW if the create activates the task         */

    int measure_flag;         /* if 1, activates measure of exec time         */
    void *arg;                /* pointer to a task argument                   */
    rtmode_t *modes;          /* a 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:

period	specifies the task period;
rdline	specifies the task relative deadline in milliseconds. By default it is set equal to period.
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 the a **ptask_activate**() is invoked by another task.
measure_flag	if set to a value different from 0 the library automatically profiles the execution time of the task.
arg	a pointer to a memory that is used to pass arguments to the task. The content of the memory and its meaning are user-defined.
modes	used for managing mode changes (see Section xx)
nmodes	the number of modes for this task.
mode_list	the list of modes for this task.

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 asd macros).

void ptask_param_init(tpars tp);

Initializes the task parameters structure **tp** with the default values.

void ptask_param_period(tpars tp, int myper);

Initializes the task period with myper in the structure **tp**.

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

Initializes the task relative deadline with mydline in the structure **tp**.

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

Initializes the task priority with myprio in the structure **tp**.

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

Initializes the task activation mode with myact in the structure **tp**.

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

Specifies the index of the processor on which the task will run. Notice that this is only valid if the global PARTITIONED strategy has been set in the ptask_init() function call. It has no effect when the scheduling strategy is set equal to GLOBAL.

```
void ptask_param_measure(tpars p);
```

Set the measuring flag to 1.

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

Sets the argument of the task.

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

Specifies the number of modes for the task. Parameter modes is a structure that defines the modes for the system.

```
void ptask_param_mode_add(tpars p, int n, int mode_num);
```

Specifies that the task is present in mode mode_num. Parameter n must be an increasing number for each mode specified for the task. See Section xx for an example of use of this function.

To create a task then it is sufficient to call 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
rel_dline	1000 ms
priority	1
actmode	DEFERRED
processor	0
measure	NO
...	

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

```
int ptask_get_index();
```

Returns the index of the calling task.

```
int ptask_get_period(int tid, int unit);
```

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

```
void ptask_set_period(int tid, int myper, int unit);
```

Set the period of the task with index **`tid`** to **`myper`** (in units).

```
ptime ptask_get_deadline(int tid, int unit);
```

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

void ptask_set_deadline(int tid, int mydline, int unit);

Set the relative deadline of the task with index **`tid`** to **`mydline`** (in milliseconds).

ptime ptask_get_absdeadline(int tid, int unit);

Returns the current absolute deadline of the task with index `tid`, in units, since the start of the library.

void ptask_set_absdeadline(int tid, ptime adline, int units);

Set the absolute deadline of the current job of the task with index **`tid`** to **`adline`** **since the start of the library**.

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 core_id);

Move the calling task to the core specified by `core_id`.

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.

5. Measuring execution times

To measure the execution time of a task, it is necessary to enable ther measure flag when creating the task. Then, after the task has completed execution, it is possible to obtain its execution time by calling the following functions.

WARNING: calling these functions while the task executes may give inconsistent values (because the internal data structures are not protected by semaphores for minimal overhead).

```
tspec       ptask_stat_getwcet(i)
```

Returns the maximum execution time monitored since the first task activation.

```
tspec       ptask_stat_getavg(i)
```

Returns the average execution time computed since the first task activation.

```
int   ptask_stat_getnuminstances(i)
```

Returns the number of jobs activated since the task creation time.

```
tspec       ptask_stat_gettotal(i)
```

Returns the total execution time performed by all the jobs since the task creation time.

6. Handling mode changes

It is possible to specify a set of modes, and handling mode changes transparently. The functions of the interface are the following.

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

This function initializes a data structure of type rtmode_t which will represent the group of task that participate to the mode change. It is necessary to specify the number of modes nmodes.

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

This function is called when we want to perform a mode change. It is necessary to specify the new mode_id in which we want to the system to transition, and it returns the id of the old mode. The mode change is performed by an internal manager task that performs the mode change protocol. In the current implementation, the mode manager will wait for the largest absolute deadline of the tasks that will block because they are only present in the old mode before activating the task that are present in the new 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);
    ptask_param_init(param);
    ptask_param_period(param, 1, SEC);
    ptask_param_priority(param, 4);
    ptask_param_modes(param, &mymodes, 2);
    ptask_param_mode_add(param, 0, MODE_ON);
    ptask_param_mode_add(param, 1, MODE_FAIL);
    res = ptask_create_param(taskbody, &param);
    // creates the other tasks
    rtmode_changemode(&mymodes, MODE_OFF);  // sets initial mode
    if (condition)
       rtmode_changemode(&mymodes, MODE_ON);
    else if (error_condition)
       rtmode_changemode(&mymodes, MODE_FAIL);
    …
}
```

Figure 4: example of mode change.