Proc. of the 18th IEEE Conf. on Emerging Technologies and Factory Automation (ETFA 2013), Cagliari, Italy, September 10-13, 2013.

Ptask: an Educational C Library for Programming Real-Time Systems on Linux

Giorgio Buttazzo[*], Giuseppe Lipari[†]
[*]Scuola Superiore Sant'Anna, Italy
g.buttazzo@sssup.it
[†]LSV - ENS, France
giuseppe.lipari@lsv.ens-cachan.fr

***Abstract*—When learning real-time programming, the novice is faced with many technical difficulties due to low-level C libraries that require considerable programming effort even for implementing a simple periodic task. For example, the POSIX Real-Time standard only provides a low level notion of thread, hence programmers usually build higher level code on top of the POSIX API, every time re-inventing the wheel.**

In this paper we present a simple C library that simplifies real-time programming in Linux by hiding low-level details of task creation, allocation and synchronization, and provides utilities for more high-level functionalities, like support for mode-change and adaptive systems. The library is released as open-source and it is currently being employed to teach real-time programming in university courses in embedded systems.

I. Introduction

A huge number of embedded control applications requires the execution of periodic activities, which cyclically perform the same computation on different data at specific rates determined by the system characteristics. A periodic task typically consists of three main phases: input, processing, and output. In the input phase, the data to be processed are accessed from a shared memory buffer or acquired from an external device (e.g., a camera, a keyboard, or an interconnection network) and stored in some local data structure. In the processing phase, data are elaborated according to a given algorithm (for example for filtering, classification, or recognition), while in the output phase data are stored in another shared buffer or transferred into an external device (e.g., a motor, a transceiver, or a graphic display).

To generate a periodic execution at a precise activation rate, such a sequence of phases is normally inserted in a loop, whose last instruction is a synchronization call that suspends the task until the arrival of an event generated by a timer. Figure 1 illustrates a typical structure of a periodic activity implemented with a `while` loop terminating when a given condition becomes true. The `wait_for_activation()` call inserted before entering the `while` loop is a synchronization primitive that suspends the execution until an explicit activation is invoked; hence, it allows activating the task upon a given instruction that can be executed by the system or by another task. The `wait_for_period()` call inserted at the end of the loop is another synchronization primitive that suspends the execution until the next activation time. In this case the re-activation must be triggered by a timer, properly programmed to interrupt at the next activation time.

```
TASK sample_task()
{
  < local variables >
  wait_for_activation();
  while (condition) {
    < task body >
    wait_for_period();
  }
}
```

Figure 1. Sample structure of a periodic task.

In more complex real-time systems, many periodic tasks run concurrently, together with aperiodic tasks that deal with external or internal events. Tasks may be activated, suspended or killed dynamically depending on the state of the system; they may need to access shared memory; in some cases, they require the dynamic adaptation of run-time parameters.

Linux is one of the most popular operating systems, and it is being widely used in embedded systems domain for implementing real-time applications. The advantages of Linux are well known: it is widely available as open source; it is supported by a large community of developers; and it implements the POSIX interface, including the real-time extensions. Many "real-time improvements" of Linux have been proposed as academic open source projects (e.g., RT-Linux [17], RTAI [16], XENOMAI [20], Linux Preemption patch [18], and `SCHED_DEADLINE` [11]). In this paper we concentrate on projects that use standard APIs, like the POSIX real-time interface [3].

Unfortunately, programming real-time applications in Linux using the POSIX interface is cumbersome. In fact, POSIX threads represent a general, but also a low-level programming paradigm. Higher level abstractions must be implemented by the programmer, and even a simple entity, like a periodic thread, requires careful coding. An example of periodic thread implemented with the POSIX RT API is shown in Figure 2. The programmer must spend a significant effort to implement the periodic activation mechanism, passing parameters to the thread, dealing with time operations (there is no interface for summing `struct timespec` variables), identifying deadline miss situations (not shown in the code), or measuring the

```
void *thread_body(void *arg)
{
  struct timespec next, period;
  sem_t *s_act;
  < local variables >
  < read parameters from arg >
  sem_wait(s_act);

  clock_gettime(CLOCK_MONOTONIC, &next);
  while (condition) {
    < task body >
    < next = next + period >
    clock_nanosleep(CLOCK_MONOTONIC,
                    TIMER_ABSTIME, &next, 0);
  }
}
```

Figure 2. A periodic thread in POSIX RT.

execution time of task instances.

Another problem is that typical task models and abstractions used in the real-time scheduling theory do not find a corresponding abstraction in the library functions. There is no abstraction for a "periodic" task, nor for the concept of deadline. Also, as we will discuss in Section II-A, the mechanisms provided by the kernel are sometimes too general and do not have a corresponding theoretical problem. Such mismatch between theory and practice is, in our opinion, one of the reasons that has slowed down the introduction of proper real-time techniques in everyday programming.

To overcome such difficulties, programmers write their own libraries for supporting higher level abstractions, every time reinventing the wheel. This is particularly annoying when teaching real-time systems programming to novices. Linux is perfect as a platform for educational purposes. However, when using the bare POSIX API, students must necessarily focus on the low-level details and put extra care on identifying all corner cases of thread synchronization. As a consequence, explaining complex programming techniques, like *mode changes* [15] or *adaptive tasks* [6], [8], becomes extremely difficult.

a) Contributions of this paper: In this paper, we present a library for real-time programming in Linux, called `Ptask`, to manage a set of periodic and aperiodic soft real-time tasks under the Linux operating system. The main purpose of this library is to teach real-time programming in university courses on embedded systems. Therefore, we aimed at simplifying real-time programming by providing a clean interface, rather than achieving efficiency and performance optimization. To achieve these objectives, we designed our library trying to provide a close mapping between the provided API and the task models and techniques found in the real-time scheduling theory.

Library functions are implemented on top of the POSIX thread (Pthread) library [3]. The library exploits the Linux priority scheduler to execute tasks under the Rate Monotonic or Deadline Monotonic priority assignments. In addition, it provides convenient functions for dealing with time, support different types of tasks, allocate tasks on multi-core systems, checking deadlines, and measuring tasks' execution time. Finally, it also provides support for higher level programming techniques, like group scheduling and mode-changes.

Since the library is released as open source code, the interested students and practitioners can improve their code or take inspiration for their own situations. The library is currently used in several courses at the University of Pisa.

The rest of the paper is organized as follows. Section II provides a brief overview of the Linux schedulers, summarizing the available kernel mechanisms for managing tasks, handling timers, and accessing shared resources under priority inheritance protocols. Section IV-B presents the adopted task model and parameters. Section IV describes the `Ptask` library with all the available functions. Section V concludes the chapter and states some perspectives.

II. Background on the Linux scheduler

The current standard Linux kernel (version 3.8) provides a modular and flexible scheduling architecture because it can support many different *scheduling modules*, managed as a prioritized hierarchy. When the kernel needs to select a task to be executed on a processor, it looks at each scheduling module in a priority order, until it finds a task to be executed. The official kernel provided by Linus Torvalds includes at least 2 scheduling modules: the Completely Fair Scheduler (CFS) and the real-time scheduler, which has priority over CFS.

The CFS is a non real-time, best effort fair scheduler that can be selected by specifying the constant `SCHED_OTHER` when creating processes and threads. It is the standard Linux time-sharing scheduler that is intended for all processes that do not require real-time service. It uses a peculiar priority aging mechanism to ensure fairness among tasks.

The real-time scheduling module provides two scheduling policies, compliant with the POSIX RT standard:

- `SCHED_FIFO`: is a priority-based scheduler where threads with the same priority are managed by a FIFO policy. Under this scheduler, a thread runs until either it terminates, it is preempted by a higher priority thread, it is blocked by an I/O request, or it executes a cancellation call.
- `SCHED_RR` is a priority-based scheduler where threads with the same priority are managed by a Round-Robin policy. Under this scheduler, a thread runs until either it terminates, it is preempted by a higher priority thread, it is blocked by an I/O request, it executes a cancellation call, or it consumes the available time quantum. The Round-Robin time quantum depends on the system and cannot be defined by the user. However, the length of the time quantum can be retrieved by calling the function `sched_rr_get_interval()`.

Linux provides 99 priority levels, where level 1 denotes the lowest priority and level 99 the highest priority (note, however, that the POSIX standard requires to ensure only 32 levels). Each priority is associated with a queue, in which all threads with the same priority are enqueued. The thread at the head of the queue with the highest priority level is selected as the running task.

SCHED_FIFO and SCHED_RR are referred to as real-time policies and can be used only from the superuser (root). For example, a Rate Monotonic preemptive scheduler [13] can be easily implemented by assigning each periodic task a priority inversely proportional to its period. Similarly, a Deadline Monotonic preemptive scheduler [12] can be easily implemented by assigning each periodic task a priority inversely proportional to its relative deadline.

Recently, a new scheduling module, namely SCHED_DEADLINE [11], has been proposed as a patch to the Linux kernel to provide the Earliest Deadline First (EDF) algorithm [13] along with a Constant Bandwidth Server (CBS) [1], [2] to support resource reservation. The SCHED_DEADLINE module is supposed to run at the highest priority level in the sequence of scheduling modules.

A. *Multi-processor scheduling*

The POSIX RT interface does not directly address multi-processor scheduling. The reason is probably due to the fact that, when the interface was standardized, multi-processor RT systems where not widely popular. Recently, multi-core platforms are available even in embedded systems. For this reason, Linux provides non-standard extensions to the POSIX API to support multi-core scheduling, through the concept of *affinity*. Every thread is associated with a *CPU bit-mask* that tells the system the processors on which the thread can be executed. Each bit of the mask represents a processor and a bit set to 1 means that the thread can execute on the corresponding processor.

By default, a thread is associated with a mask having all bits set to 1. This means that by default Linux implements *global scheduling*, i.e. a thread can migrate across all processors. In particular, the RT scheduling module implements global fixed priority scheduling. The user can set a different mask at creation time or during the thread life-time by using the following functions:

```
int pthread_setaffinity_np(pthread_t thread,
    size_t cpusetsize, const cpu_set_t *cpuset);
int pthread_attr_setaffinity_np(pthread_attr_t *attr,
    size_t cpusetsize, const cpu_set_t *cpuset);
```

where cpuset is the *affinity mask*.

In our opinion, such a generic interface is ill-conceived. In the real-time literature, multi-processor schedulers can be categorized into *global*, *partitioned*, or *clustered* schedulers [10].

In global scheduling, a task can execute in any processor and all ready tasks are enqueued in one single logical ready queue: this corresponds to the default behavior of Linux, where all bits in the CPU mask are set to 1. In partitioned scheduling each task is assigned only one processor: this corresponds to setting only one bit to 1 in the mask.

In clustered scheduling, the set of processors is divided into disjoint *clusters* of processors, and every task is assigned to a single cluster. This corresponds to setting a subsets of bits of the mask to 1; moreover, two threads belonging to the same cluster are assigned the same mask; two threads belonging to different clusters do not share any processor, so their masks have no bits in common.

Global scheduling has been extensively studied in the literature and efficient schedulability tests for fixed priority scheduling have been proposed [4]. Partitioned scheduling instead reduces to single processor schedulability for each task.

In Linux, the CPU bit mask can be set in an arbitrary way, but such a generic model has never been studied in the literature, and we doubt it has ever been used in practical situations, due to the difficulty of analyzing the resulting schedule. Nevertheless, it can be a source of confusion for novice programmers that are not aware of the theoretical background in multi-processor scheduling.

B. *Time management in POSIX*

POSIX provides the struct timespec data type to store time information.

```
struct timespec {
  time_t tv_sec;   // seconds
  long   tv_nsec;  // nanoseconds
};
```

However, it does not provide any interface for manipulating such a data structure. The POSIX RT interface provides timers and clocks:

- CLOCK_REALTIME: it maintains a value that is as close as possible to the absolute time. However, it may be discontinuous, because it can be adjusted by the system and by the user.
- CLOCK_MONOTONIC: it represents the elapsed time from an unspecified initial instant. It is not affected by adjustments, hence it is the best solution for measuring the time elapsed between two events. However, it can be subject to automatic adjustments by the NTP protocol.
- CLOCK_MONOTONIC_RAW: it is similar to CLOCK_MONOTONIC, but it gives access to a low-level raw hardware timer and it is not subject to adjustments by the NTP protocol.

It also provides clocks specific to processes and threads. In particular, CLOCK_THREAD_CPUTIME_ID measure the execution time of a thread. Time can be read by using the following function:

```
int clock_gettime(clockid_t  clk_id,
                  struct timespec *t);
```

which stores in t the value of the clock specified by clk_id. Periodic behavior can be implemented by using the following function:

```
int clock_nanosleep(clockid_t clock_id, int flags,
                    const struct timespec *request,
                    struct timespec *remain);
```

which suspends the execution of the calling thread until clock clk_id reaches the time specified by t. If flag is equal to zero, the time t is interpreted as relative to the current time; If flag is equal to TIMER_ABSTIME, the time t is interpreted as an absolute value. If the thread is awakened before the set time, the remaining time is stored in rem. An example of implementation of a periodic task using this interface is shown in Figure 2.

C. Resource Access Protocols

The pthread library permits to model critical sections of code to be executed in mutual exclusion through the mutex mechanism, which uses a semaphore variable of type `pthread_mutex_t` to protect a shared resource from simultaneous accesses. If `mux` is a mutex semaphore, `pthread_mutex_init(&mux, NULL)` initializes the semaphore with default values. Then, critical sections can be accessed according to the following scheme:

```
pthread_mutex_lock(&mux);
< critical section >
pthread_mutex_unlock(&mux);
```

A mutex semaphore can be initialized in three different modes using specific attributes. To do so, a variable, say `myatt`, of type `pthread_mutexattr_t`, must be defined and then initialized by `pthread_mutexattr_init(&myatt)`. Then a protocol can be set using `pthread_mutexattr_setprotocol(&myatt, protocol)`, where `protocol` can have one of the following values:

- `PTHREAD_PRIO_NONE`
  This is the classical mutual exclusion mechanism using binary semaphores for accessing critical sections. This method suffers from priority inversion phenomena [7] that may introduce unbounded blocking in the execution of high-priority tasks.
- `PTHREAD_PRIO_INHERIT`
  This method accesses shared resources using the Priority Inheritance Protocol [19], which prevents priority inversion by increasing the priority of a task holding a resource to the maximum priority among those tasks blocked on the same resource.
- `PTHREAD_PRIO_PROTECT`
  This method accesses shared resources using the Immediate Priority Ceiling Protocol (also known as Highest Locker Priority [7]). According to this method each resource R_k is assigned a ceiling $C(R_k)$ equal to the highest priority among the tasks using R_k. Then, a task entering a critical section related to R_k executes at a priority level equal to the ceiling $C(R_k)$.

If using Immediate Priority Ceiling, a *ceiling* value must be associated with each semaphore and it must be equal to the highest priority among the threads using it. The code fragment in Figure 3 shows the set of pthread calls required to initialize a semaphore using the Immediate Priority Ceiling protocol. As you can see, the code is cumbersome to write and to read. Unfortunately, this is true for many other parts of the POSIX API, and it one of the reasons that forces many programmers to implement their own wrapper interface to POSIX.

```
pthread_mutex_t mux;
pthread_mutexattr_t myatt;
pthread_mutexattr_init(&myatt);
pthread_mutexattr_setprotocol(&matt,
                              PTHREAD_PRIO_PROTECT);
pthread_mutexattr_setprioceiling(&myatt, ceiling);
pthread_mutex_init(&mux, &myatt);
pthread_mutexattr_destroy(&myatt);
```

Figure 3. Creating and setting the protocol for a mutex.

III. Task Model

A real-time periodic task, denoted by τ_i, is a portion of code cyclically executed several times on different data. Each execution instance, identified as a job and denoted by $\tau_{i,j}$ ($j = 1, 2, \ldots$) is triggered at a precise time instant, called the *job activation time*. Note that the activation of any consecutive jobs of a periodic task τ_i is exactly separated by the same interval, called the *task period*. In general, the following parameters are typically defined on a periodic task:

- *Worst-case execution time* (WCET) C_i of task τ_i: it is the longest possible execution time of any job on the considered hardware platform.
- *Activation time* $a_{i,j}$: it is the absolute time at which job $\tau_{i,j}$ becomes active (i.e., ready to execute).
- *Period* T_i: it is the separation interval between any two consecutive job activation times of task τ_i.
- *Relative deadline* D_i: it is the maximum time (relative to the activation time) within which any job of task τ_i should complete its execution.
- *Priority* P_i: it is a number specifying the relative importance of task τ_i with respect to the others and used by the scheduler to select the task to execute among the set of active tasks ready to run.
- *Phase* Φ_i: it is the activation time of the first job of task τ_i ($\Phi_i = a_{i,1}$). Note that, all the subsequent jobs of a periodic task are activated at precise time instants given by
$$a_{i,j} = \Phi_i + (j-1)T_i.$$
- *Absolute deadline* $d_{i,j}$: it is the maximum absolute time within which job $\tau_{i,j}$ should complete its execution. It is computed as
$$d_{i,j} = a_{i,j} + D_i.$$

A periodic task τ_i activated at time Φ_i is said to be *schedulable* by a given scheduling algorithm $\mathcal{A}$ if and only if both the following conditions are met:

1) each job of τ_i is activated at time $a_{i,j} = \Phi_i + (j-1)T_i$;
2) each job of τ_i is completed no later than $d_{i,j} = a_{i,j} + D_i$.

A set Γ of n periodic tasks is said to be schedulable by a given scheduling algorithm $\mathcal{A}$ if and only if all tasks are schedulable.

Note that meeting the first condition is a responsibility of the operating system, which has to manage dedicated timers to wake up each job at the given activation time. On the other hand, meeting the second condition is a responsibility of the application designer, who has to perform an appropriate schedulability analysis to guarantee that all the tasks are schedulable by the selected scheduling algorithm.

IV. The ptask library

In this section we present the ptask library along with some example of usage. The library is divided into modules, so our presentation will follow the same division. We first state the design principles that guided our implementation:

- Simplicity of usage: the library must be easy to use, so that the students can concentrate on the theoretical concepts and test their knowledge with simple examples;
- Task model presented in the research literature must be immediately mapped on simple code structures, so that there is a direct correspondence between theoretical model and implementation;
- Advanced techniques must be readily available in the library, so that students can use them and test the mechanisms without particular regard to the low-level synchronization;
- The library must be open-source and accessible to the students that can then study the internal mechanisms and a standard and safe implementation.

The ptask library has been written in standard C (ISO/IEC 9899:2011). The reason for choosing C is that it is by far the dominant language in embedded system development; therefore the students are exposed to a language and a development environment similar to what they will find in industry. The ptask library avoids the use of dynamic memory, except for a few initialization functions that are meant to be called before the application starts executing.

In order to achieve the second objective, when necessary ptask restricts the power of the POSIX library, so that the theoretical concepts find a more direct correspondence with the abstractions provided by the library. For example, ptask forces the programmer to select a scheduler for the entire application: in fact, the library needs to be initialized by calling function `ptask_init(scheduler)`, where `scheduler` is the scheduling policy to be used in the program (see Section IV-D). Therefore, all tasks will be created using the same scheduling policy.

This restriction corresponds to what is usually explained in the first part of a course in Real-Time Systems, in which there is only one scheduling discipline for the entire system. In this way, the students are introduced to the concepts in a step-by-step fashion. Once the students progress in their knowledge, they can change the scheduling discipline for single tasks by using appropriate functions, thus implementing multi-level scheduling applications.

A. Time management

The library provides a set of simple functions to manipulate timing data in module `ptime.h`. The basic time data type is `tspec_t`, which is simply mapped into a `struct timespec`. In addition time can be expressed by a `ptime_t`, which is mapped into a `long` and needs an (implicit) unit. It is possible to sum and subtract `tspec_t` variables, and

```
typedef struct _tst {
  tspec_t period;  /* period                      */
  tspec_t rdline;  /* relative deadline           */
  tspec_t phase;   /* initial phase               */
  int priority;    /* from 0 to 99                */
  int processor;   /* processor id                */
  int act_flag;    /* ACT activates the task      */
  int measure;     /* if 1, measures exec time    */
  void *arg;       /* pointer to a task argument */
  rtmode_t *modes; /* pointer to the mode struct */
  int mode_list[RTMODE_MAX_MODES];  /* mode list */
  int nmodes;      /*< num of modes for the task */
} task_spec_t;
```

Figure 4. Task parameters.

to convert to and from `ptime_t` by using the following functions:

```
tspec_t tspec_add_delta(const tspec_t *a,
                        ptime_t delta, int units);
tspec_t tspec_add(const tspec_t *a, const tspec_t *b);
ptime_t tspec_to(const tspec_t *t, int unit);
tspec_t tspec_from(ptime_t t, int unit);
```

where `unit` is the time granularity and can be `SEC` (seconds), `MILLI`, `MICRO`, or `NANO`. In this way, time specification is more natural and readable. For example, a time variable of 125 milliseconds can be specified simply as `tspec_from(125, MILLI)`.

B. Task model and structure

In the ptask library, the code of a task is specified using a C function similar to the one shown in Figure 1, where the keyword `TASK` is an alias for `void`. A task is created by calling one of the following two functions:

```
int task_create(void (*task)(void),
    int period, int drel, int prio, int aflag);

int task_create_ex(void (*task)(void),
                   task_spec_t *tp);
```

The first function allows the user to create a task by passing the most important parameters directly as arguments. This function is presented first to the students, so that they can immediately implement a simple periodic thread in their first test programs.

The second function allows a more advanced use by taking as input the `task_spec_t` structure, which contains all task parameters. Both functions return a integer that represents the task id (a number in `[0, MAX_TASKS]`), or a negative value in the case of an error.

The complete list of parameters in the `task_spec_t` structure is shown in Figure 4. They will be described as we progress in the presentation of the library.

When creating a task, the library actually creates a thread by calling the `pthread_create()` passing the address of an internal function called `ptask_std_body()`, which in turns calls the user-specified function. The body of such an internal function is shown in Figure 5. The library keeps a list of task descriptors `struct task_par` in the array `_tp[]` and the index of the current task in the thread-specific variable

```
1  static void *ptask_std_body(void *arg)
2  {
3    struct task_par *pdes = (struct task_par *)arg;
4    ptask_idx = pdes->index;
5    if (_tp[ptask_idx].measure_flag)
6      tstat_init(ptask_idx);
7
8    pthread_cleanup_push(ptask_exit_handler, 0);
9
10   if (_tp[ptask_idx].group != NULL)
11     tgroup_wait(_tp[ptask_idx].group,
12                 _tp[ptask_idx].phase);
13   else if (_tp[ptask_idx].act_flag == 1)
14     wait_for_activation();
15   else
16     clock_gettime(CLOCK_MONOTONIC,
17                   &_tp[ptask_idx].at);
18
19   (*pdes->body)();
20
21   pthread_cleanup_pop(1);
22   return 0;
23 }
```

Figure 5. Code of the internal thread function.

```
int main ()
{
  tgroup_t group;

  ptask_init(SCHED_FIFO);
  tgroup_init(&group, 2);

  task_spec_t param = TASK_SPEC_DFL;
  param.period = tspec_from(300, MILLI);
  param.priority = 10;
  param.group = &group;
  param.phase = tspec_from(0, MILLI);
  int T1 = task_create_ex(task_body, &param);

  param.period = tspec_from(600, MILLI);
  param.priority = 10;
  param.group = &group;
  param.phase = tspec_from(100, MILLI);
  int T2 = task_create_ex(task_body, &param);
  ...
  tgroup_activate(&group);
  ...
}
```

Figure 6. Task group

`ptask_idx` (line 4 in Figure 5). A thread-specific variable optimizes the access to the internal descriptor.

Each task is associated with an internal private semaphore that is used to control the task activation and suspension. Parameter `act_flag` allows to configure the way the task is started. If `act_flag` is set to 0, the task is started immediately (as with the pthread API); if `act_flag` is set to 1, the task immediately blocks on its private semaphore waiting for an explicit activation.

After checking all relevant flags, the user-specified function is called (line 19). The `ptask_exit_handler()` is a function that cleans and releases all resources that have been allocated in the library for creating this tasks, like internal descriptors, the task semaphore, etc. It is automatically called when the task exits, thanks to the cleanup mechanism provided by the pthread API (line 8 and 21).

Periodic tasks can be grouped together to implement synchronous or asynchronous periodic task sets. To implement both models, ptask provides the `tgroup_t` structure and an appropriate synchronization mechanisms is implemented using function `tgroup_wait()` (line 11 and 12). The use of such structure is better explained with a simple example. In Figure 6 we implemented two periodic tasks belonging to the same group. Task T1 has period equal to 300 msec, deadline equal to period and phase equal 0; T2 has period equal to 600 msec, and phase 100 milliseconds. Initially both tasks block on a special semaphore for the group using the `tgroup_wait()` function (see Figure 5).

The group is activated when the user calls `tgroup_activate()`; from that instant, each task waits for an interval equal to its phase before starting execution.

C. Measurements and deadline checks

Worst-case and average case execution times of a task can be measured by setting the `measure` flag equal to 1. Then, the statistical data of task τ_i can be obtained by calling:

```
tspec_t tstat_getwcet(int i);
tspec_t tstat_getavg(int i);
```

Measurement is based on the use of the `CLOCK_THREAD_CPUTIME_ID` clock id of the POSIX API and it is performed automatically within the `wait_for_period()` and the `wait_for_activation()` functions.

These functions also detect and count deadline misses by simply measuring the time elapsed from the job activation time until the time the job is suspended at the end of its cycle. Therefore, when a deadline miss occurs, the task continues executing and the event is detected only when one of the two functions is invoked by the task.

The current version of the library does not allow detecting a deadline miss at the deadline instant. Also, currently it is not possible to constrain the execution time of a task to be within a certain budget. Implementing such features requires the use of POSIX RT signals along with the `setjump()` and `longjump()` functions. Work in this direction has been carried out by Cucinotta and Faggioli [9], who proposed the OML library for supporting `timing exceptions` in the C language and provided an implementation over the Linux kernel.

D. Multi-processor scheduling

In the ptask library we decided to restrict to two possible models: global scheduling and fully-partitioned scheduling. To use global scheduling the `SCHED_GLOBAL_FP` constant must be passed to the `ptask_init()` initialization function. To select partitioned scheduling, the `SCHED_PART_FP` constant has to be passed to `ptask_init()`. In this second case, it is possible to specify the processor on which the task is allocated by setting the `processor` parameter in the `task_spec_t` structure and using the `task_create_ex()` function. By default, the created task is allocated on processor 0. The number of available processors can be obtained by calling `ptask_getnumcores()`. Later, it is possible to migrate a task by using the function `task_migrate(int i, int p)`.

E. Mutual exclusion

In Section II-C we discussed the POSIX RT interface for creating a mutex and setting a protocol for accessing shared resources. However, once again the interface does not specify the system behavior in a multi-processor systems. Recently, Brandenburg and Bastoni [5] have shown that using simple priority inheritance in a multi-processor environment does not reduce priority inversion: they illustrated an example in which using priority inheritance in a multi-core Linux based system the blocking time can be as high as in a single processor environment without priority inheritance. They proposed two solutions: *migratory priority inheritance* and *priority boosting*. The first one consists of migrating the blocking task on the processor that hosts the highest priority blocked task. However, this approach requires support from the kernel, and hence it is outside the scope of this paper.

Priority boosting consists in using the priority ceiling protocol and assigning each resource a ceiling higher than the priority of any other task. This increases the blocking time of higher priority tasks, even if they do not use any resource, and hence it is not optimal. However, this is a safe method that can be analyzed using the same methodology as in the M-PCP protocol [14].

The ptask library provides two simple functions to initialize mutex semaphores, one for the priority inheritance protocol and one for the priority ceiling protocol:

```
int pmux_create_pi(pthread_mutex_t *m);
int pmux_create_pc(pthread_mutex_t *m, int c);
```

The computation of the correct ceiling for a certain resource is difficult to automatize using a library. Therefore, choosing simplicity over transparency, we let the user select the best ceiling depending on the task accessing the resource and the platform (single or multiprocessor).

F. Mode change

Many control systems can be modelled as a set of *operating modes* [15]. Each mode represents a state of the system, and produces a different behavior. A finite state machine governs the transition between different modes.

Each mode is associated a set of tasks, and each task can be associated with different modes. Let $\mathcal{M} = \{M_1, \ldots, M_n\}$ be the set of modes of the system, and let $\mathcal{T}_i = \{\tau_{i,1}, \ldots, \tau_{i,n_i}\}$ be the set of tasks associated with mode M_i. When the system is in a certain mode M_i, the tasks in $\mathcal{T}_i$ are active, while the other tasks are suspended. When the system *changes mode* from M_i to M_j, the transition involves the following operations:

1) the tasks in $\mathcal{T}_i - \mathcal{T}_j$ must be suspended;
2) the tasks in $\mathcal{T}_j - \mathcal{T}_i$ must be activated;
3) the tasks in $\mathcal{T}_i \cap \mathcal{T}_j$ remain active.

The exact sequence of suspensions and activations can be different, depending on the specific *mode change protocol*.

Usually, mode changes are difficult to implement, since they require a careful coordination between the tasks by using non trivial synchronization protocols. In the ptask library we provide an automatic mechanism for supporting mode changes using the *idle-protocol*. The mechanism is implemented by a *task manager* that runs at the highest priority and upon a mode change request performs the following steps:

1) All tasks in $\mathcal{T}_i - \mathcal{T}_j$ are suspended at the end of their job cycle;
2) The task manager waits for the latest end of the period of any suspended task;
3) The new tasks in $\mathcal{T}_j - \mathcal{T}_j$ are activated.

The interface for implementing the mode change is very simple and it is better explained by the example reported in in Figure 7. In this example the system consists of two modes, `MODE_ON` and `MODE_FAIL`. In mode `MODE_ON`, two tasks are present; in mode `MODE_FAIL` only the first task is present.

The mode structure is initialized in the main, by declaring and initializing a data structure `rtmode_t` with 2 modes (lines 12 and 15). Then, when creating the task, we need to pass a parameter structure, where the `param.modes` field points to the `rtmode_t` structure (lines 18 and 26); the `param.nmodes` field contains the number of modes associated with the task (lines 19 and 27) and the `param.mode_list[]` field contains the identifiers of the modes associated with the task (lines 20, 21 and 28). Then, the first time we have to set the system in the initial mode by calling `rtmode_change()` (line 31) and specifying the initial mode; this activates all tasks in that mode. Then, every time we need to change mode (according to some state machine), we need to call `rtmode_change()`.

Note that the code of the task remains the same: the mode change protocol is implemented within functions `wait_for_period()` and `wait_for_activation()`, automatically and transparently.

Function `rtmode_change()` sends the id of the new mode over a communication channel, implemented as a circular array. On the other side of the channel, the task manager is awaken and performs the mode change protocol. The communication channel ensures that requests for mode changes are enqueued and served in a FIFO order; no new request of change can be performed before the previous one has completed.

Different mode change protocols can be implemented by changing the task manager; the possibility of implementing some of the more complicated mode change protocols described in [15] is currently under investigation.

```
void taskbody()
{
  wait_for_activation();
  while (1) {
    printf("Task T%d is running\n",
           get_taskindex());
    wait_for_period();
  }
}

int main() {
  rtmode_t mymodes;
  task_spec_t param;

  rtmode_init(&mymodes, 2);
  param = TASK_SPEC_DFL;
  < set main task parameters >
  param.modes = &mymodes;
  param.nmodes = 2;
  param.mode_list[0] = MODE_ON;
  param.mode_list[1] = MODE_FAIL;
  int T1 = task_create_ex(taskbody, &param);

  param = TASK_SPEC_DFL;
  < set main task parameters >
  param.modes = &mymodes;
  param.nmodes = 1;
  param.mode_list[0] = MODE_ON;
  int T2 = task_create_ex(taskbody, &param);
  ...
  rtmode_change(&mymodes, MODE_ON);

  < state machine code >
}
```

Figure 7. Mode change interface

V. Conclusions and perspectives

The current trend in embedded systems shows that hardware is evolving more rapidly than software, causing a strong need for methodologies able to achieve portability, modularity, and scalability of performance. Surprisingly, such a growth in hardware complexity was not balanced by a corresponding evolution of the control software for a predictable an efficient management of the computational resources. In particular, the POSIX RT interface for programming real-time systems suffers from aging and it is not adequate to today complex requirements, as it does not permit to easily implement concepts and abstractions available in the real-time theory.

We presented ptask , a C library that provides a simple interface to program soft real-time activities and express high-level concepts, like periodic tasks, time synchronization, and mode changes, through simple and transparent abstractions. The library has been originally designed for educational purposes, but we believe it can also be quite useful to practitioners. The library is available as open source code at https://github.com/glipari/ptask.

As a future work, we plan to extend the library to include other mechanisms and abstractions. One one hand, we would like to support other schedulers, like `SCHED_DEADLINE` [11]; on the other hand, we plan to integrate ptask with the OML library [9] for supporting timing exceptions. In addition, we plan to extend the interface to support clustered scheduling.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] L. Abeni and G. Buttazzo. Resource reservations in dynamic real-time systems. *Real-Time Systems*, 27(2):123–165, 2004.

[3] B. Barney. Posix threads programming, January 2013. Lawrence Livermore National Laboratory, URL: https://computing.llnl.gov/tutorials/pthreads/.

[4] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[5] B. B. Brandenburg and A. Bastoni. The case for migratory priority inheritance in linux: Bounded priority inversions on multiprocessors. In *Proceedings of 14th the Real-Time Linux Workshop*, Department of Computer Science, University of North Carolina at Chapel Hill, October 2012.

[6] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, March 2002.

[7] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*. Springer, New York, 2011.

[8] G. C. Buttazzo and L. Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23(1):7–24, July 2002.

[9] T. Cucinotta and D. Faggioli. An exception based approach to timing constraints violations in real-time and multimedia applications. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 136–145, 2010.

[10] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, October 2011.

[11] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta. An efficient and scalable implementation of global EDF in linux. In *Proceedings of the 7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2011)*, Porto, Portugal, 2011.

[12] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[13] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[14] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 259–269, 1988.

[15] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[16] http://www.aero.polimi.it/ rtai/. RTAI Official Website.

[17] http://www.fsmlabs.com. FSMLabs - RTLinux Official website.

[18] Ingo molnar's rt tree. http://people.redhat.com/mingo/realtime-preempt.

[19] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[20] XENOMAI home page. http://www.xenomai.org.