Technical Report

Sporadic Multiprocessor Linux Scheduler

Paulo Baltarejo Sousa

HURRAY-TR-090102

Version: 0

Date: 01-15-2009

Sporadic Multiprocessor Linux Scheduler

Paulo Baltarejo Sousa

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: pbsousa@dei.isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

The advent of the multicore systems renewed the interest of research community in real-time scheduling on multiprocessor systems. Real-time scheduling theory for uniprocessors is considered a mature research field, but real-time scheduling theory for multiprocessors is an emerging research field. Being part of this research community we have decided to implement the Sporadic Multiprocessor Linux Scheduler (SMLS) for a new real-time scheduling algorithm that was designed to schedule real-time sporadic tasks on multiprocessor systems.This technical report describes the implementation of the SMLS.

Sporadic Multiprocessor Linux Scheduler

Paulo Baltarejo Sousa
IPP-HURRAY! Research Group,
Polytechnic Institute of Porto (ISEP-IPP),
Rua Dr. António Bernardino de Almeda 431,
4200-072 Porto, Portugal
pbsousa@dei.isep.ipp.pt

Abstract

The advent of multicore systems has renewed the interest of research community on real-time scheduling on multiprocessor systems. Real-time scheduling theory for uniprocessors is considered a mature research field, but real-time scheduling theory for multiprocessors is an emerging research field. Being part of this research community I have decided to implement the Sporadic Multiprocessor Linux Scheduler that implements a new real-time scheduling algorithm, which was designed to schedule real-time sporadic tasks on multiprocessor systems. This technical reports describes the implementation of the SMLS.

1 Introduction

Multicore platforms are being commercialized coming with an increasing number of cores, expecting to reach hundreds of processors per chip in the future [1]. These new platforms triggered the interest of the research community for real-time scheduling on multiprocessor systems. Actually, they have renewed the interest of the research community for this issue, because real-time scheduling on multiprocessors has started a long time ago [2, 3].

Real-time scheduling theory is well-developed for uniprocessors and is considered a mature research field, but real-time scheduling theory for multicore systems is an emerging research field. I have chosen to study real-time multicore scheduling theory with practice. So, I want to know whether predictions made by that scheduling theory are valid in practice. Usually these theoretical algorithms assume a set of assumption that do not have correspondence in a real system. For instance, they usually consider negligible the time for context switching and the execution time of the scheduler.

This technical report describes the implementation of the Sporadic Multiprocessor Linux Scheduler (SMLS), which implements the recently published Sporadic Multiprocessor Scheduling (SMS) algorithm [4] that was designed to schedule real-time sporadic tasks on multiprocessor systems.

According to our best knowledge there are not any previous implementations of this algorithm, so this is the first attempt to deeply study this algorithm in practice.

The SMLS has been implemented by modifying the general purpose Linux 2.6.28 kernel version downloaded from `www.kernel.org`. Our choice did not fall with any real-time Linux version, because (i) Linux 2.6.28 kernel version is provided with features that I believe to be relevant for implementing this algorithm, such as high resolution timers, preemption and dynamic ticks, and (ii) there are too many versions of real-time Linux that shows there are no consensus on what constitutes a real-time Linux [5].

This document is structured as follows. In Section 2 I begin an overview of the most relevant concepts of real-time multiprocessor scheduling and I also introduce the Linux modular scheduling framework. Based mainly on material from [4], I describe the main concepts of the Sporadic Multiprocessor Scheduling (SMS) algorithm in Section 3. Next, I proceed with my contributions to improve the performance of the SMS algorithm, in Section 4. In Section 5 I describe the main issues related to the SMLS implementation. And I conclude this document discussing my plans for studying the SMS algorithm in Section 7.

2 Background

The purpose of this section is to give to the reader the necessary background to understand the content of this document.

2.1 Real-Time Scheduling Algorithms on Multiprocessors

The most common definition of real-time systems is: *Real-Time Systems* are defined as those systems in which the correctness of the system depends on the logical result of computation, but also on the time at which the results are produced. This time is usually referred to as *deadline*.

Real-time applications are usually composed by multiple tasks. Depending on the criticality level, tasks can be classified as: (i) *hard real-time*, when missing a deadline produces undesirable or fatal results and (ii) *soft real-time*, where missing a deadline is not desirable but the system can still work correctly [6].

Another characteristic of real-time tasks is the periodicity, which defines the frequency in which they are activated or appear in the system. They can be classified as: (i) *periodic*, which appear regularly with at some known rate, (ii) *aperiodic*, which appear irregularly with at some unknown rate and (iii) *sporadic*, which appear irregularly with at some known rate.

The real-time scheduling algorithm should schedule tasks according to their demands such that their deadlines are met. One of the most used for uniprocessor systems is the *Earliest-Deadline-First* (EDF) [2]. The EDF

scheduling algorithm is a dynamic priority driven algorithm in which higher priority is assigned to the task that has earlier deadline.

Advances in process technology allow integration of multiple processors on a single chip, called *multicores*. Multicore processors are now mainstream, with the number of cores increasing, expecting to reach hundreds of processors per chip in the future [1, 7].

Unfortunately, real-time scheduling on multiprocessors did not enjoy such a success as it did on a uniprocessor. As early as in the 1960s, it was observed by the inventor of EDF that [3]: "*Few of the results obtained for a single processor generalize directly to the multiple processor ...*".

The research community has focused it interests on developing new scheduling algorithms [4, 8] for multiprocessors systems, which in many cases use concepts and principles used in the scheduling algorithms for uniprocessor.

The multiprocessor scheduling algorithms have traditionally been categorized as *global* or *partitioned*. Global scheduling algorithms store tasks in one global queue, shared by all processors. At any moment, the m (assuming that the system is composed by m processors) highest-priority tasks among those are selected for execution on the m processors. Tasks can migrate from one processor to another during the execution, that is, a execution of a task can be preempted in one processor and resume its execution on another processor. In contrast, partitioned scheduling algorithms partition the task set such that all tasks in a partition are assigned to the same processor. Tasks may not migrate from one processor to another. An important issue is, in both systems, one task can be executed by only one processor at any given time instant.

2.2 Linux Modular Scheduling Framework

The introduction of scheduling classes, in the Linux 2.6.23 kernel version, made the core scheduler quite extensible. The scheduling classes encapsulate scheduling policies and are implemented as modules [9]. Then, the kernel consists of a scheduler core and various modules. These modules are hierarchically organized by priority and the scheduler dispatcher will look for runnable task of each module in a decreasing order priority.

Fig. 1 shows the three native scheduler modules. *RT* and *CFS* denote the Real-Time and Completely Fair Scheduling scheduler modules, respectively. Thus, in this system the dispatcher will always look in the runqueue of *RT* for a runnable task. If the runqueue is empty, only then will it moves to the *CFS* runqueue and so on. Note that, every processor has an idle task in its runqueue that is executed when there is no other runnable task to be executed.

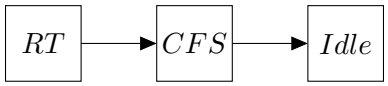


Figure 1: Linux native scheduler modules

3 Sporadic Multiprocessor Scheduling Algorithm

The Sporadic Multiprocessor Scheduling (SMS) algorithm [4], was designed to schedule real-time sporadic tasks. The SMS algorithm tries to clamp down on the number of preemptions (which involve operating system overheads) and can be configured to achieve different levels of utilization bound and migration costs. This algorithm can be categorized as semi-partitioned, since it assigns $m-1$ tasks (assuming that there are m processors in the systems) to two processors and the rest to only one processor. Next, the details of the algorithm will be discussed.

3.1 System Model

The SMS algorithm consider the problem of preemptively scheduling n sporadic tasks on m identical processors. A task τ_i is uniquely indexed in the range $1..n$ and a processor in the range $1..m$. Each task τ_i is characterized by worst-case execution time C_i and minimum inter-arrival time T_i and by the time that the execution must be completed, the deadline (D_i). In this algorithm it is assumed that T_i and C_i are real numbers and $0 \leq C_i \leq T_i$ and $D_i = T_i$.

A processor p executes at most one task at a time and no task may execute on multiple processors simultaneously. The system utilization is defined as $U_s = \frac{1}{m} \cdot \Sigma_{i=1}^{n} \frac{C_i}{T_i}$. The SMS algorithm divides time into slot of length $S = \frac{TMIN}{\delta}$. Where $TMIN$ is the minimal inter-arrival time of all tasks ($TMIN = min(T_1, T_2, \cdots, T_n)$) and δ is a parameter assigned by the designer to control the frequency of migration of tasks assigned to two processors.

α is an inflation parameter and is computed as follows: $\alpha = \frac{1}{2} - \sqrt{\delta \cdot (\delta + 1)} + \delta$. Later in this document the purpose of α will be explained. SEP is the utilization bound of SMS algorithm and is computed as follows: $SEP = 1 - (4 \cdot \alpha)$.

The SMS algorithm can be divided into two algorithms. An offline algorithm for task assignment and an online dispatching algorithm. These algorithms will be detailed in the next sections.

3.2 Tasks Assigning Algorithm

The first step of the algorithm is to sort the task set by task utilization ($U_i = \frac{C_i}{T_i}$) in descending order, such that τ_1 is the heaviest and τ_n is the

lightest tasks, respectively. Tasks whose utilization exceed SEP (henceforth called *heavy tasks*) are each assigned to a dedicated processor. Then, the remaining tasks are assigned to the remaining processors in a manner similar to next-fit bin packing [10]. Assignment is done in such a manner that the utilization of processors is exactly SEP. Task splitting is performed whenever a task causes the utilization of the processor to exceed SEP. In this case, this task (henceforth called a *split task*) is split by the current processor p and by the next one $p + 1$. Then, in these processors there are time window (called *reserves*) where this split task has priority over other tasks (henceforth called *non-split tasks*) assigned to these processors. The length of the reserves are chosen such that no overlap occurs, the split task can be scheduled, and also all non-split tasks can meet deadlines. The non-split tasks are scheduled under EDF.

Consider a system with four processors ($m = 4$) and seven tasks ($n = 7$). Table 1 shows the worst case execution time (C_i), minimum inter-arrival time (T_i) and utilization ($U_i = \frac{C_i}{T_i}$) of each task τ_i. As a matter of simplicity, the task set is already sorted in descending order by U_i. Assume also that $\delta = 4$ and consequently the utilization of the SMS algorithm is 88.85 % ($SEP = 0.8885$). The time units are intentionally omitted in Table 1, because they are not important for understanding the algorithm.

Task	C	T	U
τ_1	9	10	0.9000
τ_2	7	12	0.5833
τ_3	7	13	0.5385
τ_4	8	16	0.5000
τ_5	6	14	0.4286
τ_6	6	16	0.3750
τ_7	3	17	0.1765

Table 1: Task set

The task assignment algorithm works as follows: since τ_1 is a heavy task it is assigned to a dedicated processor (P_1). τ_2 is assigned to processor (P_2), but assigning task τ_3 to processor P_2 would cause the utilization of processor P_2 to exceed SEP ($0.5833 + 0.5385 > 0.8885$). Therefore, task τ_3 is split between processor P_2 and processor P_3. A portion of task τ_3 is assigned to processor P_2, just enough to make the utilization of processor P_2 equal to SEP, that is 0.3052. This part is referred as $hi\_split[P_2]$ and the remaining portion (0.2332) of task τ_3 is assigned to processor P_3, which is referred as $lo\_split[P_3]$. The task set assignment to processors is shown on Table 2. Note that, the U_p of each processor is shown in the last column.

Processor	Tasks and Utilization			U
	lo_split		*hi_split*	
P_1		τ_1: 0.9000		0.9000
P_2		τ_2: 0.5833	τ_3: 0.3052	0.8885
P_3	τ_3: 0.2332	τ_4: 0.5000	τ_5: 0.1533	0.8885
P_4	τ_5: 0.2733	τ_6: 0.3750 and τ_7: 0.1765		0.8247

Table 2: Task set assignment

3.3 Dispatching Algorithm

On a dedicated processor, the dispatching algorithm is very simple, whenever there is one task ready to be executed, the processor executes this task. Recall that the time is divided into timeslot of length $S = \frac{TMIN}{\delta}$ and non-dedicated processors usually execute split and non-split tasks. For that, the timeslot might be divided in three parts. The first time units x are reserved for executing the *lo_split*$[p]$ and the last time units y are reserved for executing the *hi_split*$[p]$. The remainder is reserved for executing non-split tasks. However, it is important to note that one split task executes one portion on processor p and the remaining portion on another processor $p+1$. This means that a split task τ_i will execute on both processors but not simultaneously (Fig. 2).

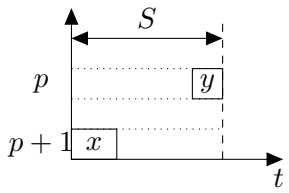


Figure 2: x and y reserves for one task

Reserves x and y for each split task must be sized such that $\frac{x+y}{S} = \frac{C_i}{T_i}$. Depending on the phasing of the arrival and deadline of τ_i relative to timeslot boundaries, the fraction of time available for τ_i between its arrival and deadline may differ from $\frac{x+y}{S}$, since a split task only executes during the reserves. Consequently, it is necessary to inflate reserves by α in order to always meet deadlines: $x = S \cdot (\alpha + lo\_split[p+1])$ and $y = S \cdot (\alpha + hi\_split[p])$. Note that, the timeslot composition is usually different for every processors.

The online dispatching algorithm works over timeslot of each processor and whenever the dispatcher is running, it checks to find the time elapsed in the current timeslot: (i) if the current time falls within a reserve (x or y) and if the assigned split task is ready to be executed, then the split task is scheduled to run on processor. Otherwise, the non-split task with the earliest deadline is scheduled to execute on processor. (ii) If the current time

does not fall within a reserve, the non-split task with the earliest deadline is scheduled to run on processor.

Table 3 presents the timeslot composition for every processor for the system model presented on Section 3.2 and Fig. 3 shows a simplified execution timeline. The timeslot length is $S = 2.5000$ and the inflation factor is $\alpha = 0.2786$. In the execution timeline presented on Fig. 3 only one activation of each task is assumed and also the release time of all tasks is at the same instant. The execution of the tasks is represented by rectangles labeled with the task's name. A black circle states the end of execution of a task. As one can see, the split tasks execute only within reserves. For instance, task τ_3 on processor P_2 executes only on reserves. Outside its reserves it does not use the processor, even if it is free. In contrast, the non-split tasks execute mainly outside the reserves but potentially also within the reserves, namely, when there is no split task ready to be executed. There are two clear situations in the Fig. 3 that illustrate this. First, task τ_7 executes at the beginning of the timeslot, which begins at 12.50, because the split task τ_5 has finished its execution on the previous timeslot. Second, split task τ_5 ends its execution a little bit before the end of timeslot that finishes at 12.50 and there is some time available on the reserve, which is used by non-split task τ_4.

Processor	x	**non-reserve**	y
P_1	0.0000	2.5000	0.0000
P_2	0.0000	1.6673	0.8327
P_3	0.6528	1.3893	0.4579
P_4	0.7529	1.7471	0.0000

Table 3: Timeslot composition

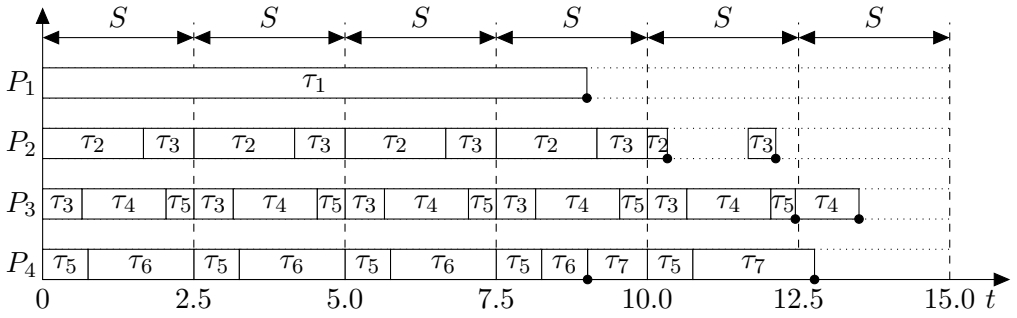


Figure 3: Execution timeline

4 Improvements to the SMS Algorithm

Usually, from theory to practice there are some obstacles and barriers that depend on the platform used to. In this section, two improvements to the SMS algorithm are described: a new timeslot compostion and a new way to compute the $TMIN$ parameter.

4.1 New Timeslot Selection

The algorithm defines at most two reserves x and y in the timeslot of each processor and one split task τ_i executes one portion on reserve y of the processor p and the other portion on reserve x of the processor $p+1$. Nevertheless, looking for two consecutive timeslots I realize that whenever a split task finishes the execution on processor p, the task has to immediately resume execution on its reserve on processor $p+1$. Actually, this situation can imply more overhead and more preemptions. This will be explained assuming the task set presented in Section 3.2, more specifically the situation illustrated in Fig. 4.

Before the explanation it is important state that, there is always some time clock drift between processors. Let us assume that the current processor is processor P_3 and the current time is a little bit more than 2.5, which falls in the x reserves. Then, the next step is to assign task τ_3 to the processor P_3 to be executed on. However, this can not be done before checking if task τ_3 is running on the processor P_2. And if it is, on one hand, processor P_3 sends an interprocessor interrupt to force rescheduling on processor P_2 to stop the execution of the task τ_3. On the other hand, processor P_3 according to the dispatching algorithm selects a non-split task to be executed on it.

This causes additional overhead and more preemptions, that could be avoided if the x reserve was available some time units later. So, the timeslot composition in such manner that x reserve is available M time units later (Fig. 5). However, this introduce a new detail, which must be the length of the M? I inquire the authors of [4] and they stated that $M < 2 \cdot \alpha \cdot S$ time units and using this value the scheduling analysis presented in [4] is still valid, thus every system that is schedulable under the previously published SMS algorithm is also schedulable under this changed algorithm.

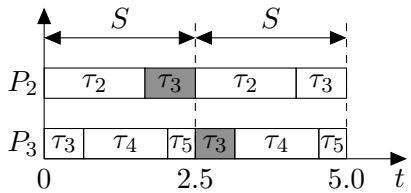


Figure 4: Timeline execution of the split task τ_3

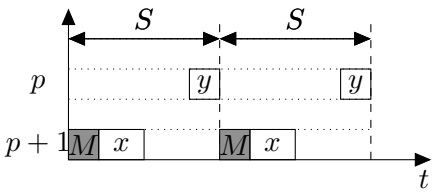


Figure 5: Position of M component in the timeslot composition

Fig 6 shows the new timeslot composition: M,x,N and y for each cpu. Where $N = S - M - x - y$. Note that, the timesolt's components M, x and y could be equal to zero.

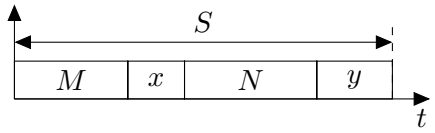


Figure 6: New timeslot composition

4.2 Improving Performance

According to the original SMS algorithm $TMIN$ is computed as the minimal arrival time of all tasks ($TMIN = min(T_1, T_2, \cdots, T_n)$) and the timeslot length $S = \frac{TMIN}{\delta}$. However, I realize that the T_i of the heavy tasks could be excluded. Thus the S could be potentially larger, if T_i of these tasks were the smaller. Note that, larger timeslot imply fewer preemptions and each task executes more time in each timeslot, consequently, the performance increases.

Using the original SMS algorithm the timeslot length is $S = 2.5000$ for the task set presented in Section 3.2, since $TMIN = 10$, which is precisely the minimal inter arrival time of a heavy task (τ_1). So, if the $TMIN$ was computed without the T_1, then the value of $TMIN = 12$, and consequently the timeslot length was also larger ($S = 3.0000$).

5 Sporadic Multiprocessor Linux Scheduler

In this section, the implementation of the Sporadic Multiprocessor Linux Scheduler (SMLS) will be described. The SMLS implementation consist on a set of modifications on the Linux 2.6.28 kernel version in order to support real-time sporadic tasks that will be scheduled according to the SMS algorithm. To differentiate these tasks from other tasks present in the system, in this document, these tasks will be referred as SMS tasks.

Note that SMS tasks are sporadic tasks and typically these kind of tasks are always present in the system waiting for events. I assume that, the SMS tasks algorithm is as shown in Listing 1.

```
while(true)
{
  wait_for_event();
  execute();
}
```

Listing 1: SMS task algorithm

5.1 SCHED_SMS macro

In order to identify the new scheduling policy it is required to define a new macro. For that, I have to change */kernel source code*/include/linux/sched.h and /usr/include/bits/sched.h files in order to define the SCHED_SMS macro (Listing 2).

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL  0
#define SCHED_FIFO    1
#define SCHED_RR      2
...
#define SCHED_SMS     7
...
```

Listing 2: Definition of the SCHED_SMS macro

5.2 Dispatching Algorithm

In this section a detailed description of the dispatching algorithm implementation of the SMS algorithm is provided.

5.2.1 Data Structures

A Linux *process* is an instance of a program in execution [11]. To manage processes, the kernel maintains information about each process in a *process descriptor*. The information stored in each process descriptor (struct task_struct, defined in */kernel source code*/include/linux/sched.h) concerns with run-state of process, address space, list of open files, process scheduling class, just to mention some. All process descriptors are stored in a circular doubly-linked list. Note that, in the context of this document, the meaning of a process or a task is the same.

To support the SMS algorithm some additional fields must be added to this data structure. Listing 3 shows the most important fields added. Fields cpu1 and cpu2 are used to set the logical identifier of processor(s) in which the task will be executed. Note that, according to the SMS algorithm each non-split task executes only on one processor, and each split task executes on two processors. In the former, these fields are set with the same identifier, in the latter, the $lo\_split[\tau_i]$ and $hi\_split[\tau_i]$ are executed on processors which identifiers are set on cpu1 and cpu2, respectively. The relative deadline of each task is set on the deadline field. Each SMS task has a specifc identifier, which is stored in the task_id field.

```
struct task_struct {
  ...
  int cpu1;
  int cpu2;
  unsigned long long deadline;
  int task_id;
```

```
    ...
};
```

Listing 3: Fields added to the `struct task_struct` data structure

Each processor holds a run-queue of all runnable processes assigned to it. The scheduling algorithm uses this run-queue to select the "best" process to be executed. The information for these processes is stored in a per-processor data structure called `struct rq`, which is declared in the */kernel source code*`/kernel/sched.c`. Listing 4 shows new data structures required by the SMS algorithm: these data structure are defined in the */kernel source code*`/kernel/sched.c` and were added to the `struct rq` data structure.

The information about each SMS task is stored using the `struct sms_task` data structure. Thus, `task` field is a pointer to the process descriptor. The absolute deadline is stored on the `deadline` field. A data type `struct rb_node` field is required for using SMS tasks on a red-black tree (`node_edf`). The linux kernel has already implemented red-black tree (*/kernel source code*`/include/linux/rbtree.h`). Basically, red-black trees are balanced binary trees whose external nodes are sorted by a key, the most operations are done in $O(log_{(n)})$ time, thus a red-black tree is indicated in situations where nodes come and go frequently.

All SMS tasks assigned to one processor are managed using the `struct sms_rq` data structure (Listing 4). The root of the red-black tree is the field `rb_edf`. All non-split tasks are organized in a red-black tree by the absolute deadline. The composition of the timeslot is defined using four fields: `m`,`x`, `n` and `y`. The `x` and `y` are used for split tasks and the others for non-split tasks. However, the non-split tasks can also be executed in the `x` and `y` reserves, if there are no split task to be executed.

Two pointers for process descriptor are used for the split tasks (`lo_split` and `hi_split`). Since split tasks are shared by two processors then, synchronization is required and is achieved by the `lock` field, which can have only two values: locked and unlocked.

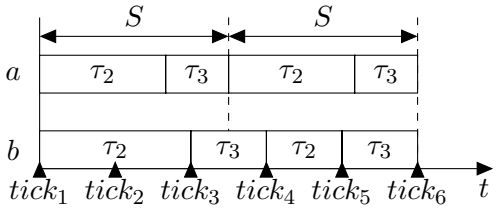
Tipically, a periodic timer interrupt mechanism, called *tick* is used by the kernel to get the control of the system. Because, the kernel code wil be executed when the tick expires. In our system, the `HZ` macro is set to 1000,so, the frequency of the tick is approximately of 1 ms, more precisely $999,848$ ns. This means that the granularity of our system is 1 ms. I did not experiment higher timer frequencies, but according to [5] experimentation with higher timer frequencies resulted in an unstable system.

To get a better grasp of the SMS timing behavior in a real system, Fig. 7 shows a timeline execution of task of processor P_2 according to the task set presented in Section 3.2, in which I now assume the time unit in milliseconds. As one can see, the theoretical and the practical timing behaviors of the SMS algorithm are different. The timeslot length is equal to 2.5 ms and the tick occurs every 1 ms, therefore the timeslot length is not a multiple of the tick.

This could lead to undesirable behavior of the scheduler. As one can see in the Fig. 7, tasks do not execute the same amount of time in every timeslot. On the other hand, the context switch does not occur at the correct time on the practical.

To solve the identified problem I need a mechanism by which timer interrupts are allowed to occur with nanosecond precision, which I think that is the ideal granularity for SMLS, but not necessarily on every nanosecond.

For each cpu is used a high resolution timer (`struct hrtimer timer`) to state the begining of each part that compose the timeslot. That is, the timer expires at the begining of the `m`,`x`, `n` and `y` parts in order to force the invocation of the schedule core.



a) Theoretical b) Pratical

Figure 7: Theoretical and practical timing behavior of the SMS algorithm

```
struct sms_task {
  struct task_struct *task;
  unsigned long long deadline;
  struct rb_node node_edf;
  ...
};

struct sms_rq {
  struct rb_root rb_edf;
  struct split_task {
    spinlock_t lock;
    struct task_struct *lo_split;
    struct task_struct *hi_split;
  }split_task;
  unsigned long long m;
  unsigned long long x;
  unsigned long long n;
  unsigned long long y;
  struct sms_timer{
    struct hrtimer timer;
    ...
  }timer;
  ...
};

struct rq {
  ...
  struct sms_rq sms_rq;
  ...
```

```
};
```

Listing 4: New data structures

5.2.2 New Scheduling Policy

To add a new scheduling policy to the Linux kernel it is necessary to create a new module. In this implementation, the SMS module was added on the top of the modules hierarchy, thus it is the highest priority module. Our system is hierarchically organized as it is shown in the Fig. 8.

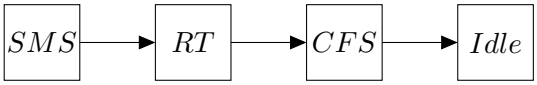


Figure 8: Priority hierarchy of scheduler modules

Note that, each scheduler module is coded in a file. The RT, CFS and Idle are coded in the */kernel source code*`/kernel/sched_rt.c`, */kernel source code*`/kernel/sched_fair.c` and */kernel source code*`/kernel/sched_idletask.c` files, respectively. Then, to implement the SMS module I have created the */kernel source code*`/kernel/sched_sms.c` file.

According to the modular scheduling framework rules each module must implement the set of functions specified in the `sched_class` structure. Listing 5 shows the definition of `sms_sched_class`, which implements the SMS module. The first field (`next`) of this structure is a pointer to `sched_class` which is pointing to the `rt_sched_class` that implements the RT module.

The other fields are functions that act as callbacks to specific events, which will be described next.

```
const struct sched_class sms_sched_class = {
  .next       = &rt_sched_class,
  .enqueue_task   = enqueue_task_sms,
  .dequeue_task   = dequeue_task_sms,
  ...
  .check_preempt_curr = check_preempt_curr_sms,
  .pick_next_task   = pick_next_task_sms,
  ...
  .task_tick      = task_tick_sms,
};
```

Listing 5: `sms_sched_class` definition

The `enqueue_task_sms` is called whenever a SMS task enters in a runnable state.

When a SMS task is no longer runnable, then the `dequeue_task_sms` function is called that undoes the work of the `enqueue_task_sms` function.

As the name suggests, `check_preempt_curr_sms` function, checks whether the currently running task must be preempted. This function is called following the enqueueing or dequeueing of a task and also following any inter-

ruption. This function only sets a flag that indicates to the scheduler core that the currently running task must be preempted.

`pick_next_task_sms` function selects the task to be executed by the current processor. This function is called by the scheduler core whenever the currently running task is marked to be preempted.

`task_tick_sms` function is mostly called from time tick functions. In the current implementation this function calls the `check_preempt_curr_sms` function, to check, if the current task must be preempted.

5.3 Task Assigning Algorithm

In this section a description of the tash assigning algorithm implementation of the SMS algorithm is provided. In the previous subsections it was described the implementation of the SMS module. However, to schedule a task, first it has to be present in the system. Initially, a SMS task is created as any task in the system, using `fork` or `clone` system calls. After that, in order to be a SMS task, there is the need to change the scheduling policy. In this section I will describe this process.

5.3.1 `struct sched_param` data structure

In order to set the scheduling parameters for a SMS task I have to change the `struct sched_param` data structure, in two files `/usr/include/bits/sched.h` and */kernel source code*`/include/linux/sched.h`. Listing 6 shows the fields added to the `struct sched_param` data structure. In the `struct sms_task_sched_param` data structure are defined a set of fields related to the SMS tasks, like a specific identifier `sms_pid`, the processor identifiers where a specific task is assigned to (`processor1` and `processor2`) and the relative deadline (`deadline`). On the other hand, `struct sms_global_sched_param` data structure is composed by global information, like timeslot size `slot_time` and also processor reserves size `reserve[NR_CPUS][4]`.

```
struct sched_param {
  int sched_priority;

  struct sms_task_sched_param{
    int task_id;
    int cpu1;
    int cpu2;
    unsigned long long deadline;
  }task;
  struct sms_global_sched_param{
    unsigned long long reserve[CPUS][4];
    unsigned long long slot_time;
    ...
  }global;
  ...
};
```

Listing 6: Changes on the `struct sched_param` data structure

5.3.2 `sched_setscheduler` function

The static priority is the priority assigned to the process when it was started. It can be modified with the `nice` and `sched_setscheduler` system calls. This is done by setting the `sched_class` field of the `struct task_struct` variable that represents the task in the system with the address of new scheduling class variable. Listing 7 shows how this is done in the `__setscheduler` function. `__setscheduler` function that is called by the `sched_setscheduler` function.

```
static void __setscheduler(struct rq *rq, struct task_struct *p, int
    policy, int prio)
{
  ...
  p->policy = policy;
  switch (p->policy) {
  ...
    case SCHED_FIFO:
    case SCHED_RR:
      p->sched_class = &rt_sched_class;
    break;
    case SCHED_SMS:
      p->sched_class = &sms_sched_class;
    break;
  }
  ...
}
```

Listing 7: Changes on the `setscheduler` function

The `sched_setscheduler` function (Listing 8) sets the scheduling policy and scheduling parameters of the process specified by the pointer `p` and the parameters specified by the pointer `param`, respectively.

```
int sched_setscheduler(struct task_struct *p, int policy,struct
    sched_param *param)
{
  ...
  if (policy == SCHED_SMS) {
    if((int)param->global.slot_time!=0){
      init_sms_rq_global(param);
    }
    p->task_id = param->task.task_id;
    p->cpu1 = param->task.cpu1;
    p->cpu2 = param->task.cpu2;
    p->deadline = param->task.deadline;
  }
  ...
  __setscheduler(rq, p, policy, param->sched_priority);
  ...
  return 0;
}
```

Listing 8: Changes on the `setscheduler` function

5.4 SMLS Logging System

In this section I will describe the logging system used by SMLS. This system is based on a circular queue implemented in the kernel space. In order to get the data stored in the circular for the user space it is used a char device driver. Each SMS task related event is stored using a tuple {`event, timestamp, data`}. I assume several kind of events: `ENQUEUE, DEQUEUE, SWITCH_TO, SWITCH_AWAY, START_M, START_X, START_N` and `START_Y`. `ENQUEUE` and `DEQUEUE` are related to the enqueue and dequeue events of a task. Concerning to the context swicth, `SWITCH_AWAY` is used to state when the task is relinquished from processor and `SWITCH_TO` is used to the task that is starting executing. `START_M, START_X, START_N` and `START_Y` are used to indicate the beginning of each timeslot part. The timestamp field states the time instant when the event hapenned and the data field holds the information related to the specific event.

Assuming the task set presented in the Table 1, Listing 9 shows the output generated by the SMLS for the processor P_3. Note that, there are one instance of each task assigned to this processor in the system: task τ_3 is a split task that runs on the x reserve; task τ_5 is a split task that runs on y reserve and task τ_4 is a non-split task. As mentioned before, each SMS tasks related event is stored using a tuple {`even, timestamp, data`}. Concerning to the events: `2` means `SWITCH_TO`; `3` means `SWITCH_AWAY` and `5`, `6`,`7` and `8` mean `START_M, START_X, START_N` and `START_Y`, respectively. The data format for `SWITCH_TO` and for `SWITCH_AWAY` events is the SMS task identifier (`task_id`) followed by the job counter. For the other events, the data field is usedt to specify when the event must be occured. Looking for the first lines, we can realize the scheduling algorithm behaviour. In the first line, it is shown the begin of M (event `2`) and according to the scheduling algorithm, in this part of the timeslot the task τ_4 must be executed by processor. Therefore, the job 1 of task τ_5 is removed from processor (event `3`) and the the job 1 of task τ_4 is assigned to the processor (event `2`).

```
...
5,468431346391,468431341873
3,468431347868,5,1
2,468431347868,4,1
6,468431349227,468431346873
3,468431350093,4,1
2,468431350093,3,1
7,469042342415,469042338373
3,469042343716,3,1
2,469042343716,4,1
8,470515248286,470515245374
3,470515249431,4,1
2,470515249431,5,1
...
```

Listing 9: Output file

6 SMS task implementation

In this section I expain how to create a SMS task showing the required code for that.

As previously referred the system call `sched_setscheduler` allows changing the scheduling policy of a process. However, the invocation of this system call must be performed by a process with root rights. Listing 10 shows the elementary SMS task code. As one can see, there is the need to set the `task_id`,`cpu1`,`cpu2` and `deadline` fields of the `struct sched_param` data structure. Note that, the `deadline` field must be set using nanoseconds unit. `sched_setscheduler` has three parameters, the first is the *pid* of the process. If *pid* is 0, the scheduler of the calling process will be set. The second argument is the policy and the last one is used to set the parameters. Note that there is the need to include the `sched.h` header file.

```
...
#include <sched.h>
...
int loop=1;

int main(int argc, char* argv[]) {

  struct sched_param param;
  ...
  param.task.task_id=3;
  param.task.cpu1=2;
  param.task.cpu2=3;
  param.task.deadline=13000000000;
  ...
  if ( sched_setscheduler( 0, SCHED_SMS , &param ) )
  {
    perror("ERROR");
  }
  ...

  while(loop)
  {
    wait_for_event();
    execute();
  }
  ...
  return 0;
}
```

Listing 10: SMS task code

7 Conclusion and Future Work

Nowadays, real-time scheduling for uniprocessor is considered mature, but real-time scheduling theory for multiprocessors has been triggered by the advent of the multicore systems, so is an emerging research field. Multiprocessor systems are much more complex than the uniprocessor systems. Usually, real-time scheduling analysis is based on a set of assumptions that

in a real implementation are not possible. For instance, they do not consider some source of overheads like the context switch and the execution of the scheduler, just to mention some. So, I have decided to study emerging real-time scheduling theory for multiprocessors with practice, that is, using real implementations of the algorithms. Motivated by this idea, I have implemented a Sporadic Multiprocessor Linux Scheduler (SMLS) based on a recently published real-time scheduling algorithm, the Sporadic Multiprocessor Scheduling (SMS) algorithm. The SMS algorithm schedules sporadic tasks for multiprocessor systems in order to met their deadlines. This algorithm tries to clamp down on the number of preemptions (which involve operating system overheads) and can be configured to achieve different levels of utilization bound and migration costs.

This first implementation gives us a better understanding of the algorithm as well as the platform used to (Linux 2.6.28 kernel version), in a such way that I have proposed a set of improvements to the algorithm in order to get better performance. Further, I am now aware of the specific features and details related to the implementation of the algorithm in this specific platform.

Future work will include investigations into timing behavior of the SMLS to get near of the SMS algorithm. The source of overheads will be explored in order to reduce some latencies. Finally, performance comparisons will be done with other algorithms to evaluate the performance of the SMS algorithm and also of the SMLS.

8 Acknowledgments

I would like to thank Björn Andersson and Konstantinos Bletsas for useful discussions.

References

[1] J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. White paper, Intel Corporation, 2006.

[2] C. L. Liu. Scheduling algorithms for hard-real-time multiprogramming of a single processor. *JPL Space Programs Summary*, II(1):37–60, 1969.

[3] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[4] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *ECRTS '08: Proceedings of the 2008 Euromicro*

Conference on Real-Time Systems, pages 243–252, Washington, DC, USA, 2008. IEEE Computer Society.

[5] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. H. Anderson. LITMUS$^{\mathrm{RT}}$: A status reports. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123. Real-Time Linux Foundation, 2007.

[6] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[7] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[8] J. H. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *Journal Computer and System Sciences*, 68(1):157–204, 2004.

[9] A. Kumar. Multiprocessing with the completely fair scheduler. Technical report, IBM, 2008.

[10] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[11] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O Reilly & Associates Inc, 2005.