Virginia Tech ❖ Bradley Department of Electrical and Computer Engineering

ECE 5984 Linux Kernel Programming
Spring 2017

Large Project: Process Scheduling

1 Introduction

Process scheduling is one of the essential functionalities of any operating system which facilitates multiple programs to share common resources between them. It enables the processes to be loaded into the executable memory and maintains their execution routine based on the scheduling algorithms. As there is no single best scheduling algorithm, different operating systems have their own scheduling policies which are based on different scheduling algorithms. For example, most general purpose Linux operating systems come with multiple scheduling policies: normal, batch, idle, FIFO, Round-Robin, etc.

In this project, you are going to implement a new scheduling policy in the kernel. An example of a new simple scheduling algorithm added to Linux will be given to you to help with the implementation.

The following concepts from the course will be put in practice in that project:

- Scheduling;
- Performance evaluation.

2 Project steps

The project can be divided into several steps:

1. Choose a simple scheduling algorithm and implement it on Linux 2.6.23.27;
2. Write a test program to validate your scheduling algorithm, i.e. prove that the runtime is stable and corresponds to the algorithm specifications;
3. Evaluate the scalability of your algorithm according to the number of tasks managed by the scheduler.

Along with the developed code, it is also needed to write and deliver a project report.

3 Steps details

3.1 Algorithm selection

It is asked to define a *simple* scheduling algorithm. This algorithm can be completely different from the ones currently existing in Linux (CFS, FIFO, RR). It can also be an adaptation from an existing one, but in that case it is asked to implement from scratch your own version of the basic algorithm (RR or FIFO as CFS is probably too complex given the project timeframe), then augment it with some original feature.

Note that a performance improvement compared to existing Linux scheduling algorithms is not expected: the primary goal is to make it functional.

Concerning the choice of the algorithm, some simple ideas can be found here:

- `http://wiki.osdev.org/Scheduling_Algorithms`
- `https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm`

If in doubt wehter the chosen algorithm will satisfy the assignment requirements, it is recommended to contact the instructor or the TA.

3.2 Algorithm implementation

Ressources about how to implement a new scheduling algorithm for Linux are very scarce. To help you with the implementation, you are given an example of a new scheduler implemented in the kernel: `SCHED_CASIO`, which is an implementation of the classic real-time EDF scheduling algorithm. This example should serve as a skeleton for your scheduler implementation. The `SCHED_CASIO` implementation is described in a guide available here: `http://www.embedded.com/design/operating-systems/4204929/Real-Time-Linux-Scheduling-Part-1`. That guide is attached at the bottom of this assignment for reference.

The scheduler should not be implemented as a module but directly in the kernel sources.

3.3 Validation

Validation should show that the implementation can handle large number of tasks without crashes. In addition, concordance to the theoritical algorithm must be demonstrated: for example, when implementing a preemptive, priority-based algortihm, it should be showed that a high priority task entering the system is preempting a currently running low priority task.

3.4 Scalability

Evaluating the scalability of an algorithm consists in measuring its performance while increasing the processed dataset. For this project, the dataset is the number of tasks. An example of scalability evaluation process would be to measure the execution time of some critical functions of the implemented scheduler (ex: the function picking the next task to run) according to the number of ready-to-run tasks.

4 Virtual machine for this assignment

You can find a link to a virtual box development environment here: `http://bit.ly/2iN2ySH`. The login/pw for this VM are "user"/"a". Root access is available through `sudo`. Inside the VM, the modified kernel will run under a Qemu emulated environment, documented in a README file. Its usage will also be presented during the course lectures. This environment is emulating a single-core CPU so that the algorithm do not need to support multicores.

The VM comes with these files on the desktop:

1. README.txt: Contains commands to use GDB and Qemu for kernel development;
2. CASIO Patch: Contains a reference implementation of the CASIO scheduler on top of linux-2.6.32.27 to help with your own implementation. It can be applied to vanilla 2.6.32.27 sources downloaded from `kernel.org`.

5 Project report

The report should be at least 5 pages and should contain:

1. A description of your chosen algorithm and a description of the implementation;
2. A description of the validation program and some comments about the validation results;
3. Description, results and interpretation of the scalability performance evaluation.

6 Group work and results to be handed - Deadline: 2017-03-03 11:59PM

This assignment should be performed by groups of 3 students.
The following is expected to be handed by 2017-03-03 11:59PM:

1. A patch for the kernel containing the modifications representing the new scheduler implementation. It should be applicable on a vanilla 2.6.32.27 kernel downloaded from `kernel.org`;
2. The sources of the validation and benchmarking programs;
3. The project report.

All of this should be contained in an archive. One archive per group should be submitted. Please register your group on Canvas (LKP → People → Group).

A `SCHED_CASIO` implementation guide

The next pages represent the guide to implement the `SCHED_CASIO` scheduling policy in Linux, and they should be used as a reference for understanding the sample code that is given to help with your own implementation. Theses pages are directly taken from `http://www.embedded.com/design/operating-systems/4204929/Real-Time-Linux-Scheduling-Part-1`.

Implementing a new real-time scheduling policy for Linux: Part 1

Paulo Baltarejo Sousa and Luis Lino Ferreira, Polytechnic Institute of Porto - July 26, 2010

Modifying any part of the Linux kernel source code is usually a challenging task most software developers would prefer to avoid, composed as it is thousands of code lines divided by hundred of files.

In this article we describe some of the techniques we have used to simplify this job during the implementation of a new scheduling policy for Linux 2.6.24 kernel version, based on the well known real-time earliest deadline first(EDF) scheduling algorithm. (*Source code for the scheduling policy implementation described in this article can be downloaded from sourceforge.net/*

The introduction of scheduling classes in the Linux 2.6.23 kernel version has, of course, made the core scheduler quite extensible. The scheduling classes encapsulate scheduling policies and are implemented as modules [1]. Then, the kernel consists of a core scheduler and various modules. These modules are hierarchically organized by priority and the scheduler dispatcher looks for a runnable task of each module in a decreasing priority order.

By in large, with the introduction of scheduling classes, implementing a new scheduling policy for Linux kernel is a much easier task. Nevertheless, there are problems in the small details, and in determining which parts of the kernel code are necessary to "touch" and which are not.

Our purpose here is to neither describe the Linux scheduler itself nor the Linux kernel. For that you must use these two references: [2] [3]. Here our focus will be on the implementation details we employed.

Our SCHED_CASIO 1 Linux Scheduler (SCLS) is the platform we used to implement the EDF scheduling algorithm by modifying the general purpose Linux 2.6.24 kernel version. (*The CASIO name comes from the name of a discipline called Conceitos Avancados de Sistemas Operativos (CASIO) of the Master course in Informatics Engineering of the Polytechnic Institute of Porto*.

Scheduling in Real-Time Systems

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced [5]. That is, the computation results must be delivered within a time bound. This time bound is referred to as deadline. Typically, a real-time system consists of a controlling (computer) and a controlled (environment) systems.

The controlling system interacts with its environment based on information available about the environment. So, the controlling system receives periodically information about the environment through sensor devices and acts on the environment through the actuator devices.

A real-time application is normally composed of multiple tasks with different levels of criticality. Hard real-time tasks cannot miss any deadline, otherwise, undesirable or fatal results would occur, while soft real-time tasks can miss some deadlines and the system can still work correctly.

For example, heart pacemaker is a hard real-time system because a delayed signal may cause the death of person whereas a live audio-video system is, usually, categorized as soft real-time system because missing a deadline results in degraded quality to the user, but the system can continue operating.

Real-time Task Parameters

Before we get into the details of the scheduler implementation it is important to understand some of the more important real-time task parameters. A real-time task (τ_i) is characterized by the following parameters: Release time (or ready time) (R_i), Worst case execution time (C_i), Deadline (D_J and Periodicity (T_{ii}).

R_i is the time at which the task is ready for processing. C_i is the processor time required for executing the task without interruption. D_i is the time at which a task should be completed to avoid damage to the system. T_i is the time interval at

which tasks are released in the system.

As a consequence of the periodicity, a real-time task T_i generates an infinite sequence of jobs (J_{ij}), where j is used to identified the j^{th} instance of task τ_i. According to the periodicity, there are basically three kinds of tasks: periodic, sporadic and aperiodic.

Periodic tasks are those that are released regularly at fixed periods. Sporadic tasks are activated irregularly but its minimum inter-arrival period is known, which is, a minimum interval between two consecutive job releases. Aperiodic tasks are activated with an unknown periodicity.

Aperiodic tasks are behind the scope of this document and the sporadic tasks can be assumed as a subset of the periodic tasks. Since, the minimum inter-arrival period can be assumed as their period.

Scheduling Algorithms
For a given set of jobs the main goal of a scheduling algorithm is to determine an execution order according to which the requirements of each job are satisfied [6].

In a real-time system the main purpose of the scheduling algorithm is to complete the execution of all jobs before their deadlines. There are several scheduling algorithms for real-time tasks. However, the most used and studied are: Rate Monotonic [4] and Earliest Deadline First [4].

Rate Monotonic (RM) assigns priorities statically according to the periodicity. The shorter period the higher is the task's priority. The Earliest Deadline First (EDF) is a dynamic scheduling algorithm that assigns the highest priority to the job with the earliest deadline.

This is an optimal scheduling algorithm on preemptive uniprocessors. The EDF algorithm can achieve an utilization of 100% if the task set presents periods equal to the deadlines (Ti = Di) for all tasks. The utilization of a system is computed as follows: $\mathbf{U = \sum^{n}_{i=1} x (C_i/T_i).}$

Let us consider the following task set example (**Table 1 below**), which utilization of the task set is 72.2%. Note that, in the last column of the Table 1 the offset (O_i) of the first job of each task is specified. That is, the time instant at which the first job is released.

Task	C	T	D	O
τ_1	2	11	11	3
τ_2	3	13	13	2
τ_3	2	15	15	1
τ_4	3	17	17	0

Table 1. Task set

Figure 1 below shows the timeline execution for the first job (only) of each task of the task set presented in Table l. The execution of the jobs is represented by gray rectangles and a black circle states the end of execution of a job.

As we can see, the priority assignment is done according to the absolute deadline (which is the sum of $R_i + D_i$) . That is, at a specific time instant the job that is executing is the one with the earliest deadline of all active jobs.

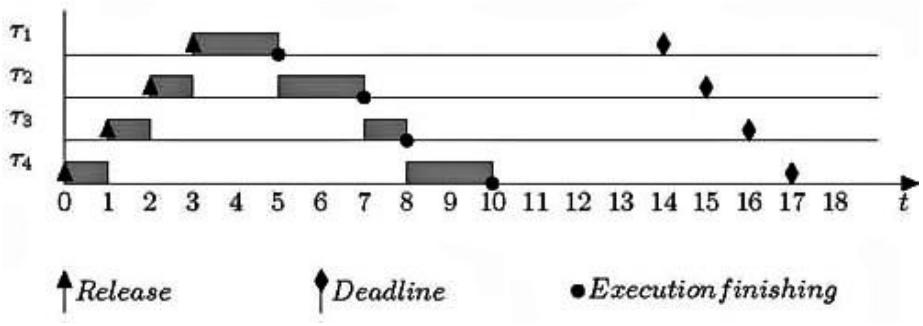

Figure l. Timeline execution according to the EDF algorithm

SCHED CASIO Linux Scheduler
The SCHED_CASIO Linux Scheduler (SCLS) implementation consist on a set of modifications to the Linux 2.6.24 kernel version to support real-time tasks scheduled according to the EDF scheduling algorithm.

To differentiate these tasks from other tasks present in the system, in this document, we refer to these tasks as CASIO tasks or CASIO jobs. Note that CASIO tasks are periodic tasks and are always present in the system. We also assume that, a CASIO tasks code structure must be similar to the algorithm presented in **Listing 1 below**.

```
start←clock();
while(true)
{
  execute();
  start←start + period;
  delay until start;
}
```

Listing 1. CASIO task algorithm

SCHED CASIO_POLICY Configuration Option
The first step is to add a new configuration option entry that is used to wrap all the SCLS required code. Since the host system is based on x86 architecture, this configuration option entry must be added to the */kernel_source_code/arch/x86/kconfig* file.

Listing 2 below shows the content of the new configuration option entry. A clarifying name (SCHED_CASIO_POLICY) defines what this option entry represents. However, this configuration option entry is referred in the code with CONFIG_SCHED_CASIO_POLICY. The CONFIG_ prefix is assumed but is not written.

The bool directive states that this option entry is a feature and can only be set using two values (y or n). The quoted text following the directive provides the name of this option in the various configuration utilities, like make menuconfig. The default value of this option is defined using default directive.

```
...
menu "CASIO scheduler"
config SCHED_CASIO_POLICY

  bool "CASIO scheduling policy"
  default y
endmenu
...
```

Listing 2. SCHEDCASIO POLICY configuration option entry

SCHED CASIO macro
The second step is to define a macro to identify the scheduling policy. For that, we have to change */kernel-source-code/include/linux/sched.h* and */usr/include/bits/sched.h* files in order to define this macro (**Listing 3 below**).

Note that, the */usr/include/bits/sched.h* file is outside of the kernel code, then the CONFIG_SCHED_CASIOPOLICY is unknown and consequently must be commented or removed.

```
...
#define SCHED_NORMAL    0
#define SCHED_FIFO      1
#define SCHED_RR        2
...
#ifdef  CONFIG_SCHED_CASIO_POLICY
#define SCHED_CASIO     6
#endif
...
```

Listing 3. Definition of the SCHED_CASIO macro

Data Structures
struct task_struct and *struct rq* are two central data structures in the system. In this section the changes made to these two data structures are explained in detail.

A Linux process is an instance of a program in execution [2]. 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 the run-state of a process, its address space, the list of open files, the process priority and its 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. In order to deal with CASIO tasks some fields must be added to this data structure.

Listing 4 below shows the fields added. Field casio_id is used to set the logical identifier of a CASIO task. The relative deadline (using nanosecond time unit) of each CASIO task is set on the deadline.

```
struct task_struct {
  ...
#ifdef  CONFIG_SCHED_CASIO_POLICY
  unsigned int casio_id;
  unsigned long long deadline;
#endif
};
```

Listing 4. Fields added to the struct task_struct data structure

Each processor holds a run-queue of all runnable processes assigned to it. The scheduling policy 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*.

The information about each CASIO task is stored using the struct casio task data structure (**Listing 5 below**). Thus, task field is a pointer to the process descriptor entry. The absolute deadline is stored on the absolute-deadline field.

A data type *struct rb* node field is required for organizing CASIO tasks on a red-black tree (casio node). 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 nodes are sorted by a key, consequently, most operations are done in $O(\log_{(n)})$ time, thus a red-black tree is adequate for situations where nodes come and go frequently. In the SCLS implementation, the key for the red-black tree is the absolute deadline.

As we can see in the Listing 5 below, a struct list head *casio_list_node* field is defined. This field is required to organize all CASIO tasks present on the system in a double linked list.

The struct list-head data structure is defined in the */kernel_source_code/include/ linux/list.h* file, which implements an easy-to use double linked list using C programming language.

All CASIO tasks assigned to one processor are managed using the *struct casio_rq* data structure. They are stored in a linked list, which list head is the casio list field.

The root of the red-black tree is the field casio task root. and *nr_running* field is used to specify the number of CASIO tasks on the run-queue. A data field of type *struct casio_rq* had to be added to the *struct rq rq* data structure.

```
#ifdef CONFIG_SCHED_CASIO_POLICY
struct casio_task{
  struct rb_node casio_node;
  unsigned long long absolute_deadline;
  struct list_head casio_list_node;
  struct task_struct *task;

};
struct casio_rq {
  struct rb_root casio_root;
  struct list_head casio_list;
  atomic_t nr_running;
};
#endif

struct rq {
  ...
#ifdef  CONFIG_SCHED_CASIO_POLICY
  struct casio_rq casio_rq;
#endif
  ...
};
```

Listing 5. Definition of CASIO specific data structures: struct casio task and struct casio_rq

The data structures of the scheduler and the run-queue are initialized in the *sched_init* function that is defined in */kernel_source_ code/kernel/sched.c* file. Since new field was added to the *struct rq rq*, then there is the need to initialize this data field.

```
void __init sched_init(void)
{
  ...
  for_each_possible_cpu(i) {
    ...
    struct rq *rq;
    ...
```

```
    rq = cpu_rq(i);
    ...
#ifdef CONFIG_SCHED_CASIO_POLICY
    init_casio_rq(&rq->casio_rq);
#endif
  ...
}
```

Listing 6. sched_init function

Listing 6 above shows the invocation of the *init_casio_rq* within the *sched_init* function, which initializes the *casio_rq* fields.

Next in Part 2: **Building a new scheduling policy module.**

Resources:
1. Source code for the SCHED CASIO Linux Scheduler

References:
1. A. Kumar. Multiprocessing with the completely fair scheduler. Technical report, IBM, 2008.
2. D. Bovet and M. Cesati. Understanding The Linux Kernel. O Reilly & Associates Inc, 2005.
3. Wolfgang Mauerer. ProfessionalLinux Kernel Architecture. Wiley Publishing, Inc., 2008.
4. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM, 20(1):46-61, 1973.
5. John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. Computer, 21(10):10-19, 1988.
6. Arezou Mohammadi and Selim G. Akl. Technical report no. 2005-499 scheduling algorithms for real-time systems. 2005.
7. C. L. Liu. Scheduling algorithms for hard-real-time multiprogramming of a single processor. JPL Space Programs Summary, II(1):37-60, 1969.

***(Paulo Baltarejo** is a researcher on scheduling algorithms for multicore processors at CISTER Research Group and also a Professor at the Polytechnic Institute of Porto in Portugal.*

***Luis Lino Ferreira** is a researcher on QoS Architectures and algorithms for mobile distributed systems at CISTER Research Group and Professor at the Polytechnic Institute of Porto.)*

Implementing a new real-time scheduling policy for Linux: Part 2

Paulo Baltarejo Sousa and Luis Lino Ferreira, Polytechnic Institute of Porto - July 27, 2010

To add a new scheduling policy to the Linux kernel it is necessary to create a new module. In the SCLS implementation, the CASIO module was added on the top of the modules hierarchy, thus it is the highest priority module. Therefore, scheduler modules becomes hierarchically organized as it is shown in the **Figure 2 below.**

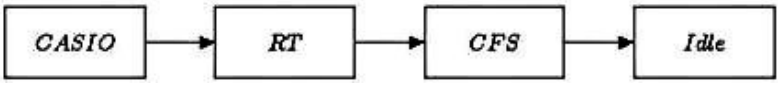

Figure 2. Priority hierarchy of scheduler modules

Note that each scheduler module is coded in a file. Currently, Linux kernel has three native scheduler modules: RT (Real-Time), CFS (Completely Fair Scheduling) and Idle.

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 CASIO module we have created the /kerneI_source_ code/kernel/sched_casio.c file.

According to the modular scheduling framework rules each module must implement a set of functions specified in the sched_class structure. **Listing 7 below** shows the definition of casio_sched_class, which implements the CASIO module.

The first field of this structure (next) is a pointer to sched_class that is used to organize the scheduler modules by priority in a linked list and the scheduler core, starting by the highest priority scheduler module, will look for a runnable task of each module in a decreasing order priority.

In this case, as casio_sched_class is the highest priority scheduler module, then this field points to the next low priority scheduler module that is struct rt_sched_class which in turn implements the RT module. (*Later we will explain how to declare the highest priority scheduler module*.)

```
const struct sched_class casio_sched_class = {
  .next       = &rt_sched_class,
  .enqueue_task   = enqueue_task_casio,
  .dequeue_task   = dequeue_task_casio,
  .check_preempt_curr = check_preempt_curr_casio,
  .pick_next_task   = pick_next_task_casio,
  ...
};
```

Listing 7. casio_sched_class scheduling class

The other fields are functions that act as callbacks to specific events, which are described in the following sections. In the description of the following functions it is assumed that the CASIO tasks are already stored in the linked list, which head is the casio list field of the struct casio_rq.

Later, we will explain how and when the tasks are added to that list. For right now, note that, some auxiliary functions are used to explain the behavior of these functions, but, their source code is not shown in this document.

The enqueue_task_casio (**Listing 8 below**) is called whenever a CASIO task enters in a runnable state. It receives two pointers, one for the run-queue of the processor that is running this code (rq) and another to the task that is entering in a runnable state (p).

This function, invoking the find_casio task list function gets the pointer to the struct casio task stored in the linked list of the struct casio_rq that points to the task p. Then, it updates the absolute-deadline and inserts the casio_task on the red-black tree (insert_casio_task_rb_tree).

The kernel native sched_clock returns the current time in nanoseconds (this function is defined in the / kernel-source-code/kernel/sched.c file). Additionally, it registers this event on the CASIO logging system (described later in this series).

```
static void enqueue_task_casio(struct rq *rq, struct task_struct *p, int wakeup)
{
  struct casio_task *t=NULL;
  char msg[CASIO_MSG_SIZE];
  if(p){
    t=find_casio_task_list(&rq->casio_rq,p);
    if(t){
      t->absolute_deadline=sched_clock()+p->deadline;
      insert_casio_task_rb_tree(&rq->casio_rq, t);
      atomic_inc(&rq->casio_rq.nr_running);
      snprintf(msg,CASIO_MSG_SIZE,"(%d:%d:%llu)",p->casio_id,p->pid,t->deadline);
      register_casio_event(sched_clock(), msg, CASIO_ENQUEUE);
    }
  }
}
```

Listing 8. enque_task_casio function

When a CASIO task is no longer runnable, then the dequeue_task_casio function is called that undoes the work of the enqueue_task_casio function (**Listing 9 below**). Note that, if the task is not being executed, then it is removed from the linked list (rem_casio_task_list).

```
static void dequeue_task_casio(struct rq *rq, struct task_struct *p, int sleep)
{
  struct casio_task *t=NULL;
  char msg[CASIO_MSG_SIZE];
  if(p){
    t=find_casio_task_list(&rq->casio_rq,p);
    if(t){
      snprintf(msg,CASIO_MSG_SIZE,"(%d:%d:%llu)",t->task->casio_id,t->task->pid,t
          ->deadline);
      register_casio_event(sched_clock(), msg, CASIO_DEQUEUE);
      remove_casio_task_rb_tree(&rq->casio_rq, t);
      atomic_dec(&rq->casio_rq.nr_running);
      if(t->task->state==TASK_DEAD || t->task->state==EXIT_DEAD || t->task->state
          ==EXIT_ZOMBIE){
        rem_casio_task_list(&rq->casio_rq,t->task);
      }
    }
  }
}
```

Figure 9

As the name suggests, check_preempt_curr_casio function, checks whether the currently running task must be preempted. This function is invoked following the enqueuing or dequeuing of a task and only sets a flag that indicates to the scheduler core that the currently running task must be preempted (through the kernel native resched_task function).

Note that, the struct task_struct pointer rq->curr points to the task that is currently running on the processor.

The currently running task must be preempted (Listing 10 below): if there is at least one CASIO task on the run-queue and the currently assigned task to the processor (rq->curr) it is not a CASIO task and also if the currently running task is a CASIO task and there is at least one CASIO task on the red-black tree with the earlier deadline.

```
static void check_preempt_curr_casio(struct rq *rq, struct task_struct *p)
{
  struct casio_task *t=NULL,*curr=NULL;
  if(atomic_read(&rq->casio_rq.nr_running) && rq->curr->policy!=SCHED_CASIO){
    resched_task(rq->curr);
  }
  else{
    t=earliest_deadline_casio_task_rb_tree(&rq->casio_rq);
    if(t){
      curr=find_casio_task_list(&rq->casio_rq,rq->curr);
      if(curr){
        if(t->absolute_deadline < curr->absolute_deadline)
```

Listing 10. check preempt-curr_casio function

The pick_next_task_casio function (**Listing 11 below**) selects the task to be executed by the current processor. This function is invoked by the scheduler core whenever the currently running task is marked to be preempted.

The CASIO task with the earliest absolute deadline (returned by earliest_deadline_casio_task_rb_tree function) is elected to be executed. Note that, if there is no CASIO task to be executed on the processor, then this function returns NULL, and this way the scheduler core tries to find one task on the next low priority scheduling class.

```
static struct task_struct *pick_next_task_casio(struct rq *rq)
{
  struct casio_task *t=NULL;
  t=earliest_deadline_casio_task_rb_tree(&rq->casio_rq);
  if(t){
    return t->task;
  }
  return NULL;
}
```

Listing 11. pick next task_casio function

Declare the Highest Priority Scheduler Module
After coding the new scheduler module, there is the need to include this module on the scheduler core file (/kernel_source_ code/kernel/sched.c) and also to inform the scheduler core which is the the highest priority scheduler module.

Listing 12 below shows the changes on the / kernel-source- code/kernel/sched.c file to include the file that implements the CASIO module (sched_casio. c) and also to inform the scheduler core that casio_sched_class is now the highest priority scheduler module.

```
...
#include "sched_rt.c"
#ifdef  CONFIG_SCHED_CASIO_POLICY
#include "sched_casio.c"
#endif

#ifdef  CONFIG_SCHED_CASIO_POLICY
  #define sched_class_highest (&casio_sched_class)
#else
  #define sched_class_highest (&rt_sched_class)
#endif
```

Listing 12. Inclusion and definition of the highest scheduling class

Defining a CASIO task
In the previous sections we described the implementation of the CASIO scheduler module. However, to schedule a task, first, it has to be present in the system.

Initially, a CASIO task is created as any task in the system, using fork or clone system calls. After that, in order to be a CASIO task, there is the need to change its scheduling policy. This section describes how to do that.

In order to set the required scheduling parameters of a CASIO task the struct sched_param data structure must be changed. This has to be done in two files /usr/include/bits/sched .h and /kernel_source- code/include/linux/sched. h.

Listing 13 below shows the fields added to the struct sched_param data structure. Once again, the /usr/include/bits/sched.h file is outside of the kernel code then the CONFIG_SCHED_CASIOPOLICY configuration option must be commented or removed.

Two fields were added to struct sched_param data structure. One is an identifier (casio_id and the other one is used to set the relative deadline (deadline) of a CASIO task.

```
struct sched_param {
  int sched_priority;
#ifdef  CONFIG_SCHED_CASIO_POLICY
  unsigned int  casio_id;
  unsigned long long deadline;
#endif
};
```

Listing 13. Changes on the struct sched_param data structure

The kernel native rt_policy function (defined in the /kernel-source- code/kernel/sched.c file) is used to decide if a given scheduling policy belongs to the real-time class (SCHED_RR and SCHED_FIFO) or not.

We changed this function in order to include the SCHED_CASIO as belonging to the real-time class, once SCHED_CASIO has higher priority than SCHED_RR and SCHEDFIFO. **Listing 14 below** shows the changes made to the rt_policy function.

```
static inline int rt_policy(int policy)
{
  if (unlikely(policy == SCHED_FIFO) || unlikely(policy == SCHED_RR)
#ifdef CONFIG_SCHED_CASIO_POLICY
      || unlikely(policy == SCHED_CASIO)
#endif
   )
    return 1;
  return 0;
}
```

Listing 14. Changes on the rt_policy function

The static priority is the priority assigned to the process when it is 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.

```
static void __setscheduler(struct rq *rq, struct task_struct *p, int policy, int
    prio)
{
  ...
  p->policy = policy;
  switch (p->policy) {
  ...
#ifdef  CONFIG_SCHED_CASIO_POLICY
  case SCHED_CASIO:
    p->sched_class = &casio_sched_class;
  break;
#endif
  }
  ...
}
```

Listing 15. Changes on the setscheduler function

Listing 15 above shows how this is done in the __setscheduler function. __setscheduler function is called by the sched_setscheduler function as we can see in **Listing 16 below.**

The sched_setscheduler function 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)
{
  ...
#ifdef  CONFIG_SCHED_CASIO_POLICY
  if(policy==SCHED_CASIO){
    p->deadline=param->deadline;
    p->casio_id=param->casio_id;
  }
#endif
  ...
#ifdef  CONFIG_SCHED_CASIO_POLICY
  if(policy==SCHED_CASIO){
    add_casio_task_2_list(&rq->casio_rq, p);
  }
#endif
  ...
  __setscheduler(rq, p, policy, param->sched_priority);
  ...
  return 0;
```

Listing 16. Changes on the sched_setscheduler function

Besides these actions, in this function, the add_casio_task_2_list function is invoked to add the task pointed by p to the linked list of the struct casio_rq data structure.

Next in Part 3: **The SCHED_CASIO Logging System**
To read **Part 1**, go to: A new Linux Scheduling Policy

Resources:
1. Source code for the SCHED CASIO Linux Scheduler

References:
1. A. Kumar. Multiprocessing with the completely fair scheduler. Technical report, IBM, 2008.
2. D. Bovet and M. Cesati. Understanding The Linux Kernel. O Reilly & Associates Inc, 2005.
3. Wolfgang Mauerer. ProfessionalLinux Kernel Architecture. Wiley Publishing, Inc., 2008.

4. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM, 20(1):46-61, 1973.
5. John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. Computer, 21(10):10-19, 1988.
6. Arezou Mohammadi and Selim G. Akl. Technical report no. 2005-499 scheduling algorithms for real-time systems. 2005.
7. C. L. Liu. Scheduling algorithms for hard-real-time multiprogramming of a single processor. JPL Space Programs Summary, II(1):37-60, 1969.

***(Paulo Baltarejo** is a researcher on scheduling algorithms for multicore processors at CISTER Research Group and also a Professor at the Polytechnic Institute of Porto in Portugal.*

***Luis Lino Ferreira** is a researcher on QoS Architectures and algorithms for mobile distributed systems at CISTER Research Group and Professor at the Polytechnic Institute of Porto.)*

Implementing a new real-time scheduling policy for Linux: Part 3

Paulo Baltarejo Sousa and Luis Lino Ferreira, Polytechnic Institute of Porto - July 28, 2010

Described in the third part in this series is the logging system used by SCLS. The logging system is based on a file entry in the /proc directory called casio_event. Therefore, to get the information from this file we can use the cat command and redirect the output to a file, for instance, typing in a terminal the following command:

bash$ cat /proc/casio events > event

Each CASIO tasks related event is stored using a tuple {action, timestamp, message}. We assume four actions: CASIO_ENQUEUE, CASIO_DEQUEUE, CASIO_CONTEXT_SWITCH and CASIO_MSG. CASIO_ENQUEUE, CASIO_DEQUEUE and CASIO_CONTEXT_SWITCH are related to the enqueue, dequeue and context switch of a task. CASIO_MSG is used for debug purpose.

The timestamp field refers to the instant time at which the action happened and msg is a string used to write the message. **Listing 17 below** shows the data structures and functions implemented for the logging system and are defined in the /kernel_source- code/include/linux/sched.h file.

```
#ifdef  CONFIG_SCHED_CASIO_POLICY

#define CASIO_MSG_SIZE       200
#define CASIO_MAX_EVENT_LINES 10000

#define CASIO_ENQUEUE      1
#define CASIO_DEQUEUE      2
#define CASIO_CONTEXT_SWITCH  3
#define CASIO_MSG          4

struct casio_event{
  int action;
  unsigned long long timestamp;
  char msg[CASIO_MSG_SIZE];
};

struct casio_event_log{
  struct casio_event casio_event[CASIO_MAX_EVENT_LINES];
  unsigned long lines;
  unsigned long cursor;
};
void init_casio_event_log();
struct casio_event_log * get_casio_event_log();
void register_casio_event(unsigned long long t, char *m, int a);
```

Listing 17. Data structures and functions for system log

The system log information is stored on the vector struct casio_event casio_event, which is managed as a circular queue. The fields lines and cursor are used to manage the information, namely the insertion and remotion of that information.

In order to get this information using an user space program like cat command, the /proc /casio_event file is created. For that purpose, we created a structure with all information required for the /proc file, including pointers to any functions in the /kernel source-code/ fs/proc/proc_misc.c (**Listing 18 below**). Then, void __init proc_misc_init registers the structure within the kernel.

```
...
//global variable
struct casio_event_log casio_event_log;
...
```

Listing 18. Creating the /proc/casio_event file

Next, is described which events are registered and where we have to write the code to catch those events. The first step is to initialize the data structures related to logging system. This must be done in the sched_init function. **Listing 19 below** shows the invocation of init_casio_event_log by the sched_init function to initialize the data fields of the logging system.

```
#ifdef  CONFIG_SCHED_CASIO_POLICY
...
static const struct file_operations proc_casio_operations = {
  .open    = casio_open,
  .read    = casio_read,
  .release  = casio_release,
};
#endif
...
void __init proc_misc_init(void)
{
  ...
#ifdef  CONFIG_SCHED_CASIO_POLICY
  {
    struct proc_dir_entry *casio_entry;
    casio_entry = create_proc_entry("casio_event", 0666, &proc_root);
    if (casio_entry){
      casio_entry->proc_fops = &proc_casio_operations;
    }
  }
#endif
}
```

Listing 19. sched_init function calling the init_casio event log

The enqueue and the dequeue events are registered in the enqueue_task_casio (Listing 8 earlier) and dequeue_task_casio (Listing 9 earlier) functions.

Another important event is the context switch, that is, when a task that is running on a processor is replaced by other. The context_switch function is the interface to the architecture-specific methods that perform a low-level context switch.

This function is invoked in the schedule function, which is the main scheduler core function. schedule function is invoked directly at many points in the kernel to allocate the processor to a different process.

Therefore, the place to register the context switch event is within the schedule function. The context switch registration is shown in the **Listing 20 below**.

The variable prev is a pointer to the task that is being replaced by the other task pointed by next variable. Note that, the next variable catchs the return of the pick_next_task function that calls the pick_next_task of each scheduling class, starting by the highest priority (in this case by the casio_sched_class) and returning when it found a task. Note that, it will always find a task. In the worst case returns the pointer to the swapper process, the idle task.

```
void __init sched_init(void)
{
  ...
  #ifdef  CONFIG_SCHED_CASIO_POLICY
    init_casio_event_log();
  #endif
  ...
}
```

Listing 20. schedule function

Assuming the task set presented in the **Table 1 below**, shown **Listing 21 below** shows part of the output generated by the SCLS.

Task	C	T	D	O
τ_1	2	11	11	3
τ_2	3	13	13	2
τ_3	2	15	15	1
τ_4	3	17	17	0

Table 1. Task set.

As mentioned before, each CASIO tasks related event is stored using a tuple {action, timestamp, message}.

```
asmlinkage void __sched schedule(void)
{
  ...
  prev->sched_class->put_prev_task(rq, prev);
  next = pick_next_task(rq, prev);

#ifdef  CONFIG_SCHED_CASIO_POLICY
  char msg[CASIO_MSG_SIZE];
  if(prev->policy==SCHED_CASIO || next->policy==SCHED_CASIO){
    snprintf(msg,CASIO_MSG_SIZE,"prev->(%d:%d),next->(%d:%d)",prev->casio_id,prev
        ->pid,next->casio_id,next->pid);
    register_casio_event(sched_clock(), msg, CASIO_CONTEXT_SWITCH);
  }
#endif
  ...
  context_switch(rq, prev, next); /* unlocks the rq */
  ...
}
```

Listing 21. Output file

Concerning to the actions: 1 means enqueueing; 2 dequeueing, 3 context swicthing and 4 debugging. The time unit is nanoseconds and specifies the time instant when the event occurred.

The message format for enqueueing and for dequeueing actions is as follows: between parenthesis and separated by colon is: CASIO identifier (casio_id); kernel Linux process identifier (pid) and the last one is the absolute deadline of the CASIO task.

The message format for the context switching is: the casio_id and pid of the process that is being replaced is coming next to the word prey and following the word *next* it is specified the casio_id and pid of the selected process to be executed by processor. Note that, when one of them it is not a CASIO task the casio_id field appears equal to -1.

Looking for the first lines of Listing 21, we can check the scheduling algorithm behaviour. In the first line, it is shown the enqueue of the CASIO task 4. According to the scheduling algorithm if there is one CASIO task in the system and the currently executing it is not a CASIO task then, it must be preempted.

The context switch appears in the second line, where task with pid equal to 0 is replaced by CASIO task with casio_id equal to 4. In the third line appears another CASIO task with casio_id equal to 3.

As its absolute deadline is earlier than the task that is running (CASIO task with casio-id equal to 4) then, this CASIO task must be preempted. The context switch appears in the fourth line.

CASIO Tasks Implementation
As previously referred the system call sched_setscheduler allows changing the scheduling policy of a process and setting scheduling parameters (using the struct sched-param data structure).

Listing 22 below shows the basic CASIO task code. As one can see, sched_setscheduler has three parameters, the first is the pid of the process. If pid is equal to 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. However, the invocation of this system call must be performed by a process with root permissions.

Note that, according to the EDF scheduling algorithm, the deadline is a fundamental item. Then, it must be set through the deadline field of the struct sched_param data structure and using nanosecond time unit. casio-id field is used to set CASIO task identifier. Note that there is the need to include the sched.h header file.

```
...
1,691479690839,(4:6227:708479686993)
3,691479697502,prev->(-1:0),next->(4:6227)
1,692471590677,(3:6226:707471526099)
3,692471667495,prev->(4:6227),next->(3:6226)
1,693472243918,(2:6225:706472229872)
3,693472249404,prev->(3:6226),next->(2:6225)
1,694464254492,(1:6223:705464249778)
3,694464270334,prev->(2:6225),next->(1:6223)
...
```

Listing 22. CASIO task code

Tools and source code for this project
To complement the source code supplied with this series of articles, following are some directions on how to set up your system in order to be able to compile the kernel as well as the way to get the patch file of the SCLS.

Tree directories. The structure of directories used by the authors to implement the SCLS is as follows: In the **$HOME** we have created two directories: ***scheduler_dev*** and ***scheduler***. The scheduler_dev was used to develop the SCLS. In the other hand, scheduler was used to compile the kernel source code.

Software required. Before you configure kernel make sure you have the minimum software versions for a kernel build:

```
bash$ sudo apt-get update
bash$ sudo apt-get install build-essential
bash$ sudo apt-get install libncurses5-dev
bash$ sudo apt-get install kernel-package
```

***Getting the required Linux kernel source code*.** Visit **http://kernel.org/** and download the linux-2.6.24.tar.bz2 source code, which represents 2.6.24 kernel version and follow these directions:

1) Use wget command to download kernel source code:

```
bash$ cd /scheduler_dev
bash$ sudo wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.24.tar.bz2
```

2) Type the following command:

```
bash$ sudo tar -xjvf linux-2.6.24.tar.bz2 -C .
```

3) Change the linux-2.6.24 directory name to linux-2.6.24-casio:

```
bash$ mv linux-2.6.24 linux-2.6.24-casio
```

4) Create a config file from the current system configuration:

```
bash$ cd linux-2.6.24-casio
bash$ sudo cp /boot/config-2.6.XX .config
```

5) Change the EXTRAVERSION variable in the Makefile to differentiate this kernel version from others present in your system (**Listing 23 below**).

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 24
EXTRAVERSION = -casio
...
```

Listing 23. kernel Makefile

6) Download the SCHED_CASIO.patch patch file from the ***https://moodle.isep.ipp.pt/mod/resource/view.php?id=34521*** to the scheduler_dev directory.

7) Apply the SCHED_CASIO patch. (Note that, at this moment the work directory is *linux-2.6.24-casio*):

```
bash$ patch -p1 < ../SCHED_CASIO.patch
```

Compiling the kernel. Assuming that the currently work directory is the $HOME directory **Listing 24 below** shows a script file with a sequence of commands to compile the kernel.

```
#!/bin/bash
sudo rm -R /lib/modules/2.6.24-casio.old
sudo mv /lib/modules/2.6.24-casio /lib/modules/2.6.24-casio.old
sudo rm -r /$HOME/scheduler/linux-*
cp -a scheduler_dev/linux-2.6.24-casio /$HOME/scheduler/
cd /$HOME/scheduler/linux-2.6.24-casio
sudo make oldconfig
sudo make-kpkg --initrd kernel_image 2>../errors
```

Listing 24. Compile script

First, we save the modules. Next, we remove any directory from scheduler directory, then we copy all source code to the scheduler directory. Therefore, the path of the source code to be compiled is in the /$HOME/scheduler/ linux-2.6.24-casio directory. We change the directory and select different options according to the needs.

Finally, the kernel compilation starts and if everything goes well a compressed kernel image is created, otherwise you can check the errors in the error file created in the /$HOME directory. Install the compiled kernel version generated by the previous step. For that, in the /$HOME/scheduler directory type:

```
bash$ sudo dpkg i kernel_image—2.6.24_xxxx.deb
```

To finish this process the system must be rebooted:

```
bash$ sudo reboot —n
```

A new kernel image will boot in your system (**Figure 3 below**).

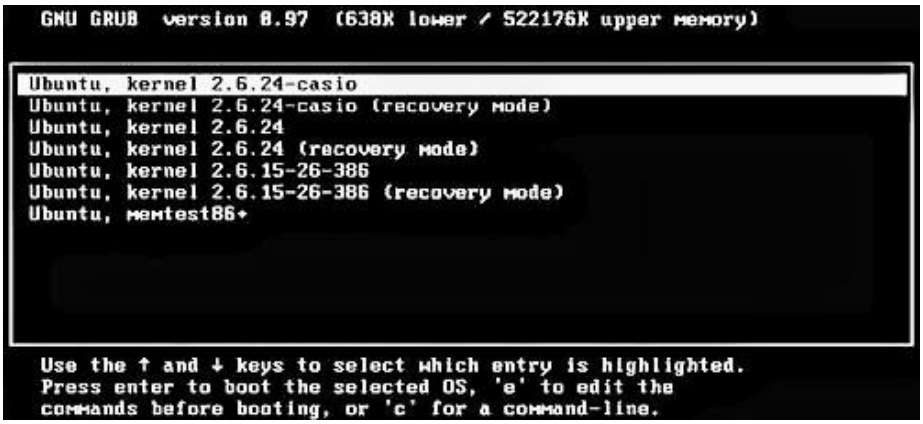

Figure 3. System rebooting

Conclusions
The Linux modular scheduling framework came with Linux 2.6.23 kernel version. This framework became the implementation of the new scheduling policies for Linux easier than the previous one.

However, in the literature we did not find documents that explain how to implement a new scheduling policy for Linux. Usually, general purpose documentation where all parts of the Linux are deeply explained can be found. The other way is hacking the Linux kernel source code.

In this document, we have presented in a depth description all steps required to implement a new scheduling policy. For that, we have modified the Linux 2.6.24 kernel version in order to support real-time tasks scheduled according to the EDF scheduling algorithm.

This is a simple implementation of that scheduling algorithm. However, advanced issues, like interruptions, timers and multiprocessor systems, just to mention some, are out of the scope of this article

To read **Part 1**, go to: Scheduling in real-time Systems.
To read **Part 2**, go to: Building a new scheduling policy module.

Resources:
1. Source code for the SCHED CASIO Linux Scheduler

References:
1. A. Kumar. Multiprocessing with the completely fair scheduler. Technical report, IBM, 2008.
2. D. Bovet and M. Cesati. Understanding The Linux Kernel. O Reilly & Associates Inc, 2005.
3. Wolfgang Mauerer. ProfessionalLinux Kernel Architecture. Wiley Publishing, Inc., 2008.
4. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM, 20(1):46-61, 1973.
5. John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems.

Computer, 21(10):10-19, 1988.
6. Arezou Mohammadi and Selim G. Akl. Technical report no. 2005-499 scheduling algorithms for real-time systems. 2005.
7. C. L. Liu. Scheduling algorithms for hard-real-time multiprogramming of a single processor. JPL Space Programs Summary, II(1):37-60, 1969.

*(**Paulo Baltarejo** is a researcher on scheduling algorithms for multicore processors at CISTER Research Group and also a Professor at the Polytechnic Institute of Porto in Portugal.*

***Luis Lino Ferreira** is a researcher on QoS Architectures and algorithms for mobile distributed systems at CISTER Research Group and Professor at the Polytechnic Institute of Porto.)*