UNIVERSITÀ DI PISA

Scuola di Ingegneria

Master of Science in Embedded Computing Systems

Novel Resource Management Mechanisms for Real-Time Scheduling in the Linux Kernel

Supervisors:
Prof. Giorgio C. Buttazzo
Ing. Mauro Marinoni

By:
Alessio Balsini

Academic Year 2013/2014

Abstract

The purpose of this work is the design and development of a new bandwidth reservation scheduling algorithm for managing time-critical tasks that may temporarily suspend their execution, waiting for events (self-suspending tasks). The resulting algorithm has been implemented in the Linux kernel.

The current scheduling algorithm used in Linux to manage CPU bandwidth reservation (SCHED_DEADLINE) is based on a server mechanism called Hard Constant Bandwidth Server (H-CBS). However, this mechanism was not designed to manage self-suspending tasks and may lead to possible deadline misses. To solve this problem, a new server mechanism, called the H-CBS-SO algorithm, has been studied. This thesis addresses the design and implementation of a Linux scheduling algorithm based on the H-CBS-SO reservation server.

The introduction illustrates the basic notions of real-time systems and presents an overview the actual SCHED_DEADLINE policy and its implementation. Then, the thesis focuses on the explanation of the H-CBS-SO algorithm and how it is developed in the Linux kernel.

The full exploitation of the scheduling policy developed in the thesis requires an infrastructure for handling "periodic tasks" that is currently missing in Linux. Therefore, this thesis proposes a new mechanism for managing periodic tasks inside the Linux kernel and a user-space library for exploiting such a new feature.

Special attention is dedicated to the description of the techniques used for checking the correctness of the kernel functions and the tools developed for measuring the achieved performance.

Contents

List of Figures **ix**

Introduction **1**

1 State of the Art **5**

1.1 Introduction to Real-Time Scheduling 6

1.1.1 Real-Time Computing Systems 7

1.1.2 Scheduling with EDF and H-CBS 8

1.2 H-CBS Issues in Real Systems Implementation 13

2 Linux Scheduler **15**

2.1 Scheduling Classes 16

2.2 Important Data Structures 18

2.2.1 sched_class 18

2.2.2 task_struct 19

2.2.3 rq 20

3 SCHED_DEADLINE **21**

3.1 Introduction 22

3.2 Data Structures and Functions 22

3.2.1 sched_dl_entity 22

3.2.2 dl_rq 23

3.2.3 dl_sched_class 24

3.3 H-CBS in SCHED_DEADLINE 26

3.3.1 Admission Control 27

3.3.2 Parameters Constraints 27

3.4 Userspace Perspective 28

3.4.1 Switching Scheduling Class 28

4 H_CBS_SO **31**
4.1 Problem and Solution 32
4.1.1 Unverified Schedulability Condition 32
4.1.2 Suspension Oblivious Analysis for H-CBS 34
4.1.3 Managing Self-Suspending Tasks Budget 35
4.2 The H-CBS-SO Algorithm 35
4.2.1 Idle 36
4.2.2 Ready 36
4.2.3 Self-Suspended 37
4.2.4 Suspended 37
4.3 Implementation 38
4.3.1 SS_QUEUE 38
4.3.2 Detecting Self-Suspension 40
4.3.3 Exhausting Budget While Self-Suspended 41
4.3.4 Replenishing Budget 41
4.3.5 Leaving Self-Suspension 42
4.3.6 Dying Tasks 42
4.3.7 Updating budget of Self-Suspended Tasks 42

5 Periodic Tasks in Linux **45**
5.1 Intro 46
5.2 Classical Periodic Real-Time Tasks 46
5.3 Kernel-Aware Periodic Real-Time Tasks 47
5.3.1 Userspace Library 47

6 Tools **53**
6.1 Ftrace 54
6.1.1 Kernel Events Tracing 54
6.1.2 Kernel Functions Profiling 56
6.2 trace-cmd and Kernelshark 56
6.3 Task Spawner 56
6.4 Generator 59
6.4.1 Generating Utilization Factors 60

7 Experimental Results **63**
7.1 Experiment Setup 64
7.2 Overhead Results 65

A Code 69

A.1 include/linux/sched.h 70

A.1.1 sched_dl_entity 70

A.2 kernel/sched/core.c 71

A.2.1 SS_QUEUE Allocation Macro 71

A.2.2 Setting Default Values to sched_dl_entity 71

A.2.3 SS_QUEUE Initialization 72

A.3 kernel/sched/deadline.c 72

A.3.1 Insertion and Removal of SS_QUEUE Elements 72

A.3.2 Getting SS_QUEUE Associated to a sched_dl_entity . 74

A.3.3 SS_QUEUE Red-Black Tree Initialization 74

A.3.4 Budget Replenishment 74

A.3.5 Managing Budget from SS_QUEUE Head 75

A.3.6 Decrementing Budget 75

A.3.7 Enqueuing SCHED_DEADLINE Tasks 77

A.3.8 Dequeuing SCHED_DEADLINE Tasks 78

A.3.9 Cleaning SS_QUEUE When Tasks Die 79

A.4 trace/events/sched.h 79

A.4.1 Macros for Generating new Ftrace Events 79

A.4.2 Macros for Accessing SS_QUEUE 87

A.5 kernel/sched/sched.h 87

A.5.1 ss_queue 87

A.6 kernel/trace/ftrace.c 87

A.6.1 ftrace_profile 87

A.6.2 Showing Ftrace Statistics 87

A.6.3 Updating Ftrace Statistics 89

List of Figures

1.1 State transition diagram for the H-CBS algorithm. 10

2.1 Linux scheduling classes. 18

4.1 Deadline miss with H-CBS algorithm. 33
4.2 H-CBS algorithm schedule with busy wait. 34
4.3 State transition diagram for the H-CBS-SO algorithm. [2] . . 36

6.1 Kernelshark Graphical User Interface. 57

7.1 *update_curr_dl()* average execution times. 66
7.2 *enqueue_task_dl()* average execution times. 66
7.3 *dequeue_task_dl()* average execution times. 67
7.4 *dl_task_timer()* average execution times. 67

Introduction

Real-time systems are computing systems that must react to their environment events with precise timing constraints. As a consequence, the correct behavior of these systems depends not only on the correctness of the computation results, but also on the time at which those results are produced [9]. Out of time results may be useless or even dangerous.

The popularity of real-time systems has increased in the last decades because of their wide-spreading success, not only in the industry, like control systems, but also for everyday use, like multimedia applications.

The system requirements may be strongly different from application to application: shared resources, computational power, reactiveness, power management and heterogeneous hardware architectures are the ordinary characteristics that drive the application development, without jeopardizing the timing constraints.

Due to the heterogeneous requirements, real-time systems were usually developed as dedicated, special purpose systems. Moreover, a large portion of real-time systems is developed in the form of embedded systems, where the commonly used hardware has limited capabilities in terms of computational power, memory and sometimes energy. This approach of developing real-time systems resulted in very optimized applications, but showed some disadvantages like tedious programming, difficult code understanding, difficult maintainability, difficult verification of timing constraints [3].

The success of real-time gave birth to a self-feeding phenomenon. Time critical applications are more easily manageable with a proper real-time operating systems and these new operating systems opened the doors also to brand new applications. The increasing trend of real-time applications requires continuous improvements of the real-time operating systems and this generates a loop between supply and demand.

More precise and realistic models and algorithms are continuously developed to meet the demanding applications requirements.

To ensure the correct behavior of the system with respect to the timing constraints, a real-time operating system must be able, if possible, to execute the tasks in the proper order. The operation of choosing which task must be executed at a given time is called *scheduling* and the entity that performs these decisions is called *scheduler*.

The hardware improvements of the last years broke the embedded systems limitations, providing enough resources to make them able to run a general purpose operating system. That allowed the Linux developers to port their kernel to embedded devices. This revolution attracted a huge number of professional and amateur developers to the real-time systems world. The Linux kernel scheduler architecture, presented in chapter 2, shows that the original purpose of the Linux kernel was completely different from the management of time sensitive tasks, so, for accomplishing the new goal, it requires several, non-trivial adjustments.

A new scheduling class, called *SCHED_DEADLINE*, was implemented into the Linux kernel to provide a *Hard Constant Bandwidth Server*s [1] (*H-CBS*) for bandwidth reservations on top of *Earliest Deadline First* [5] (*EDF*). *SCHED_DEADLINE*'s kernel implementation and management from user side will be described in chapter 3, while *EDF* and *H-CBS* scheduling algorithms will be explained in chapter 1.

The algorithm implemented by *SCHED_DEADLINE* scheduling class showed its limitations for managing certain tasks, causing mulfunctioning to the timings of the tasks into the system. Those tasks are the so called *self-suspending tasks*: tasks which may suspend their execution for an undetermined amount of time.

Moreover, real-time tasks are ofter periodic, but in the Linux kernel the concept of periodic task is still missing: currently, the only way to implement periodic tasks is at user level. For this reason, the kernel is not aware if a task is periodic or not. This results in a deep lack of the system, because the kernel cannot be able to provide an optimal real-time scheduling of the taskset.

In order to obtain a complete and reliable real-time architecture based on the Linux kernel, the first step is to implement a new cutting edge bandwidth reservation policy, able to properly handle self-suspending tasks. The chosen policy is *H-CBS-SO*, whose algorithm and implementation details for adapting *SCHED_DEADLINE* to it are described in chapter 4.

Secondly, it is required to introduce the concept of periodic tasks at kernel level and to provide the interfaces for managing those mechanisms from the user level. A solution to this, through kernel modifications and a user-space library, is proposed in chapter 5.

Verifying the correctness of the behavior of a real-time system and evaluating its performances is a difficult task. It becomes even more difficult on a complex systems like the Linux kernel.

For this purpose, as shown in chapter 6, several testing and tracing mechanisms will be used to stimulate the kernel and extract statistical data.

In the end, chapter 7 will present the performance of the SCHED_DEADLINE functions, modified for implementing the H-CBS-SO algorithm, with respect to the their original version.

Chapter 1

State of the Art

This chapter will describe the basic concepts of real-time systems and will show a well-known algorithm for managing periodic tasks with resources reservations and its limitations.

The first part describes what a Real-Time system is, which are its main requirements and will introduce the reader to the scheduling theory.

Then, will be presented a more detailed description of the Hard Constant Bandwidth Server, a scheduling algorithm for real-time systems with resource reservations constraints.

In the end, will be explained the limitations of the described algorithm, when applied to a real system.

1.1 Introduction to Real-Time Scheduling

In a multiprogrammed computing system, it is possible to distinguish three main kind of tasks:

- processor-bound;
- I/O-bound;
- real-time.

Processor-bound tasks perform intensive computations, with few usage of I/O resources. For this reason, the main target for handling those tasks is to maximize the system computing power (throughput).

I/O-bound tasks are instead tasks performing a lot of I/O operations, spending most of their time waiting for resources availability. Those tasks are often associated to the management of hadware interfaces interacting with the users, and then, must be fast to react to the actions peroformed by the users (responsiveness).

Real-time tasks are different from processor-bound or I/O-bound tasks. Their peculiarity is the relevance of the time at which computational results are produced, time that cannot get over well-defined values (deadlines). Those tasks may seem similar to I/O bound tasks, but while I/O tasks must be executed in the shortest time, real-time tasks must just be executed before their deadlines. The event of providing the computational results after the deadline is called deadline miss.

What mostly affects the system performances is the order in which the tasks are executed. The action of deciding to which task the CPU must be associated at which time is called scheduling and the kernel subsystem that performs this decision is the scheduler.

The operation performed by the CPU of switching from the execution of a task to another is called context switching and requires some time.

Depending on which tasks are running on the system and the system purpose, a scheduling algorithm can be more effective than another. For example, a system with a high number of context switches results performing for I/O bound tasks, while a system with processor-bound tasks would prefere to waste no time in context switches.

1.1.1 Real-Time Computing Systems

In real-time computing systems, the scheduler must organize the tasks executions in such a way to minimize the number of deadline misses.

There is a variety of scheduling algorithms. It is possible to distinguish between fixed and dynamic priority scheduling algorithms:

- fixed priority scheduling: tasks priorities are computed off-line. Those algorithms ensures the assignment of the CPU to the highest priority ready task, but provides few flexibility. Moreover, may be difficult to choose the right priorities in order to guarantee the timing constraints of every real-time task.
- dynamic priority scheduling: tasks priorities are computed at runtime. Those algorithms are able to dynamically adapt the tasks schedule to the system changes.

In real-time systems is common to find periodic tasks, meaning that they execute periodically the same bunch of code (job).

For the next sections, the following notation will be used for describing the parameters associated to the i-th task:

- arrival time (a_{ij}): time at which the j-th job is ready to execute;
- computation time (C_{ij}): total computation time necessary to the CPU to execute the j-th job;
- absolute deadline (d_{ij}): j-th job's deadline;
- relative deadline (D_{ij}): difference between absolute deadline and arrival time: $D_{ij} = d_{ij} - a_{ij}$;
- starting time (s_{ij}): time at which the j-th job obtains the CPU the first time;
- finishing time (f_{ij}): time at which the j-th job finishes its execution;
- response time (R_{ij}): time required by the j-th job to finish its execution: $R_{ij} = f_{ij}a_{ij}$;
- period (P_{ij}): for periodic tasks, the task is activated periodically, every P_{ij}.

Another important parameter is the task utilization factor

$$U_i = \frac{C_i}{P_i}$$

This is a fractional value representing how much CPU is used by a certain task.

The sum of all the utilization factors of the running tasks represents the CPU load.

1.1.2 Scheduling with EDF and H-CBS

For a deeper understanding of the ideas behind the SCHED_DEADLINE implementation, the next chapters will provide a brief presentation of the theory behind the mechanisms used by this scheduling class.

Earliest Deadline First

The Earliest Deadline First (EDF) is a dynamic priority scheduling rule that assigns tasks priorities depending on their absolute dedlines. Specifically, the tasks priorities are inversely proportional to their absolute deadlines.

The EDF rule can be used for scheduling periodic and aperiodic tasks without any additional effort. This is because the priority depends only on absolute deadlines. For periodic tasks, the absolute deadline of each job of each task can be calculated as

$$d_{ij} = \Phi_i + (j-1)\,P_i + D_i$$

As dynamic priority scheduling, it is typically executed in preemptive mode, so, the scheduler can preempt the current executing task with a newly activated higher priority tasks.

As demonstrated by Dertouzos [4], EDF is optimal in the sense of feasibility: if there exists a feasible schedule for a taskset, then EDF is able to find it.

One more point in favor of EDF is the easiness of feasibility verification: a taskset is schedulable by EDF if and only if the CPU utilization factor

$$U = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{C_i}{P_i} \leq 1$$

In theory, it should be possible to reach a CPU utilization factor of 1.

Hard Constant Bandwidth Server

As presented in the previous section, the schedulability verification of a taskset requires the utilization factors of each tasks, depending on the computation time value, which is not easy to compute.

For critical systems, the verification is done by considering the Worst Case Computation Time (WCET) of each job.

Using the WCET is a strongly pessimistic approach and, for this reason, it is common to prefere more relaxed assumptions. This approach may result in overload conditions that must be properly handled for limiting disastrous phenomena like domino effects on missing other tasks deadlines. In other words, the system must ensure *temporal isolation* among different tasks.

Resource reservation algorithms were developed for achieving this system property.

A typical resource reservation implementation is obtained by assigning a dedicated real-time server, called *reservation server*. Each server is characterized by a budget Q and a period P, meaning that each server provides Q units of execution time to its tasks every P time units. From those two values it is possible to obtain the server bandwidth $\alpha = \frac{Q}{P}$.

It is possible to abstract the server mechanisms by looking at each server as a separate processor, whose speed is α times the speed of the real processor.

Originally, Abeni and Buttazzo presented the Constant Bandwidth Server (CBS) algorithm [6] for handling multimedia applications in real-time systems. This resource reservation mechanism was developed on top of an EDF scheduling policy.

Hard-CBS (H-CBS), proposed by Marzario et al. [8] and whose presentation was given by Biondi, Melani and Bertogna [1] is an evolution of the CBS algorithm.

It is possible to demonstrate that, gien a set of N H-CBS servers, all the servers are schedulable by EDF if and only if:

$$U = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{Q_i}{P_i} \leq 1$$

In H-CBS, each server is characterized by three dynamic variables, updated at runtime:

- *deadline* d;
- *virtual time* v;
- *reactivation time* z.

Each server is defined as *backlogged* if it has any active job, awaiting to be executed or *non-backlogged* otherwise.

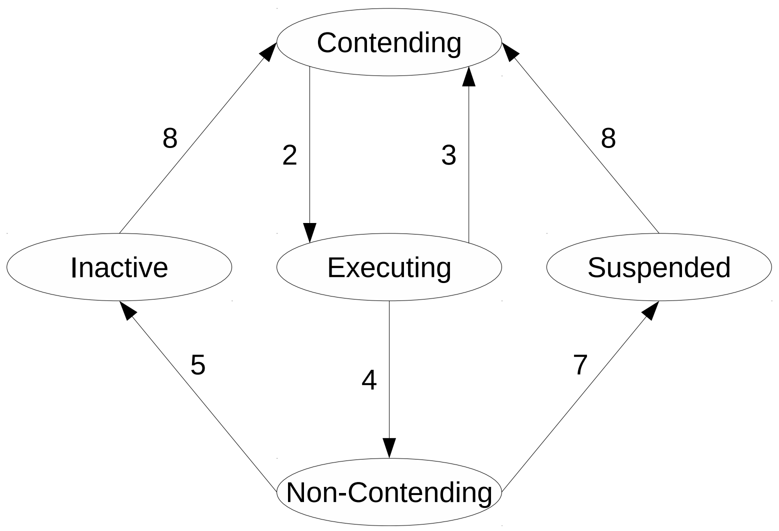


Figure 1.1: State transition diagram for the H-CBS algorithm.

At each time instant t, a server can be in one of the following possible statuses, as illustrated in figure 1.1:

- *inactive*: when
  - non-backlogged;
  - $v \leq t$;
- *non-contending*: when
  - non-backlogged;
  - $v > t$;
- *contending*: when

  - backlogged;
  - eligible to execute;

- *executing*: when
  - backlogged;
  - currently running;

- *suspended*: when
  - backlogged;
  - $v = d$;
  - $t < z$.

It is possible to notice that the comparisons between t and v are associated to the bandwidth violation checking.

The transitions between server statuses are:

1. the server is initially in the *inactive* state and, when it wishes to condtend for execution, switches to *contending* state. At the same time, the following updates are performed:
   - $d = t + P$;
   - $v = t$;

2. because of the EDF policy, the earliest deadline server is chosen for execution and switches to *executing* state. While the server is executing, its virtual time v is incremented;

3. the *executing* server is preempted by a higher priority server and switches back to the *contending* state;

4. the *executing* server has no more pending jobs to execute, it switches to *non-contending* state and remains there as long as $v > t$;

5. a *non-contending* server did not overcome its available bandwidth ($v \leq t$), so it switches to *inactive* state;

6. if the virtual time v of an *executing* server reaches the deadline d, then, the server switches to *suspended* state and the following updates are performed:

- $z = v$;
- $d = v + P$;

7. a *non-contending* server wants to contend, but its bandwidth is over $v > t$, so, it transits to the *suspended* state;
8. a *suspended* server switches back to the *contending* statewhen the reactivation time z is reached.

By introducing the two variables:

- absolute deadline: d;
- remaining budget: q;

it is possible to formulate the H-CBS rules in terms of period and budget.

1. the server starts with $q = 0$ and $d = 0$;
2. when H-CBS is idle and a job arrives at time t, a replenishment time is computed as $t_r = d - \frac{q}{\alpha}$:
   (a) if $t < t_r$, the server suspends until t_r. At time t_r, the server returns active, replenishing the budget to Q and updating the deadline as $d = t_r + P$;
   (b) otherwise, the budget is immediately replenished;
3. when $q = 0$, the server is suspended until time d. At time d, the budget is replenished and the deadline postponed: $d = d + P$.

Thanks to those rules, a server cannot run for a time longer than what is guaranteed by its budget.

1.2 H-CBS Issues in Real Systems Implementation

The H-CBS server shows its limitation in the management of self-suspending tasks.

Self-suspending tasks are tasks that for any reason stop their execution before finishing their computation and before exhausting tieir budget. Common causes that may lead a task to self-suspend are:

- mutual exclusion mechanisms;
- explicit sleep;
- I/O operations.

As Biondi, Parri and Marinoni showed [2], a taskset where tasks are allowed to self-suspend cannot be guaranteed to be schedulable with the H-CBS algorithm. Then, they proposed an alternative algorithm, called H-CBS-SO.

The reason is that, when a task self-suspends, the scheduler chooses another task to be executed. Meanwhile, the self-suspended task's budget is not consumed.

The budget maintained during the self-suspension can still be used by the task after the self-suspension. What may happen is that this posticipation of execution may cause deadline misses of lower priority tasks.

Chapter 2

Linux Scheduler

This chapter describes the architecture of the Linux kernel scheduler, from the hi-level structure to the implementation details.

The first part shows the modular organization of the scheduler and its partitioning into scheduling classes for managing different kind of tasks.

The second part presents the implementation details of the Linux kernel scheduler, describing which are the main used data structures and how those data structure are managed.

2.1 Scheduling Classes

Being Linux a General Purpose Operating System (GPOS), there's a variety of different typologies of tasks that can be executed.

For this reason, the system should behave in different ways to improve the performance of every task typology.

The Linux kernel approached this problem by introducing scheduling classes.

Scheduling classes are organized in a hierarchical way, ordered by scheduling class priority. Those classes are, in order of descending priority:

- *stop*;
- *dl*;
- *rt*;
- *normal*.

Stop

This is the scheduling class with the highest priority.

Tasks belonging to the *SCHED_STOP* scheduling class preempt everything and cannot be preempted by anything.

Deadline

This is the SCHED_DEADLINE scheduling class, which provides a resource reservation policy with a dynamic priority assignment.

All the details of this scheduling class will be described in detail in the next chapters.

Real-Time

The Linux kernel provides also some basic mechanisms for scheduling tasks requiring a priori awareness of their execution order.

This category includes two scheduling classes:

- *SCHED_FIFO*;
- *SCHED_RR*;

SCHED_FIFO is a simple scheduling algorithm without time slicing. The following rules apply for *SCHED_FIFO* tasks:

- when a task becomes runnable, it is inserted in the tail of the *SCHED_FIFO* queue;
- the same happens when a task calls *sched_yield()*;
- a call to *sched_setscheduler()*, *sched_setparam()*, or *sched_setattr()* will put the task identified by the passed *pid* at the start of the list if it was runnable;
- a task is preempted only if another higher priority task arrives or if the task suspends.

With the *SCHED_FIFO* scheduling class, a tasks that aims to run forever, can do it, causing the starvation of all the other *SCHED_FIFO* tasks and the lower priority tasks.

SCHED_RR class uses the same rules of *SCHED_FIFO*, but adds the concept of *time quantum.* A *SCHED_RR* task is also preempted in favor of another *SCHED_RR* task after occupying the CPU for the *time quantum.*

Normal

Tasks having no real-time requirements are managed through those classes:

- *SCHED_OTHER*;
- *SCHED_BATCH*;
- *SCHED_IDLE*;

SCHED_OTHER is the default scheduling class used by Linux. It tries to maximize the execution fairness among all *SCHED_OTHER* threads. The fairness is driven by a dynamic priority: the *nice* value. The nicer is a task, the less time it will use the CPU, in favor of the other tasks.

SCHED_BATCH policy is similar to *SCHED_OTHER*, but in this case the tasks are assumed to be CPU-bound. As a consequence, the number of preemptions is limited.

The *SCHED_IDLE* scheduling class is assigned the lower priority. This class is intended for the scheduling of very low priority background tasks.

2.2 Important Data Structures

The next paragraphs will describe how the Linux kernel implements this modular framework for managing several scheduling classes.

2.2.1 sched_class

The Linux kernel uses a single interface for all the scheduling classes. This provides a generalization level for making the scheduling classes easy to be handled by the system.

Scheduling classes are organized as a list, having the *stop* class as head.

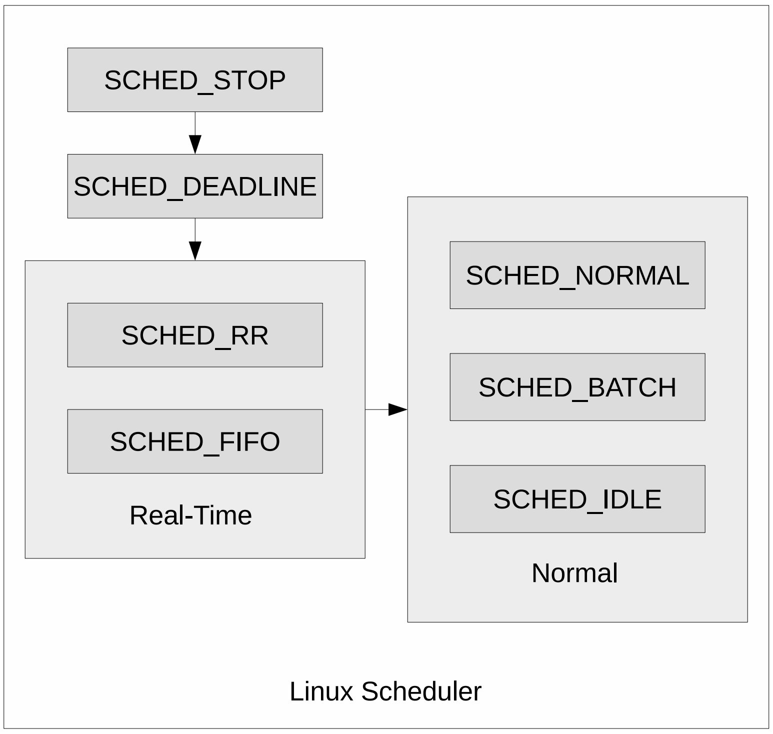


Figure 2.1: Linux scheduling classes.

The Linux scheduler traverses this list and asks the scheduling classes if there are available runnable tasks. The first scheduling class that has a task a runnable process wins and that task is picked for execution.

sched_class is the data structure that defines all the function hooks needed for managing the scheduling class.

Some of the function pointers defined by *sched_class* are the following:

- *enqueue_task*: invoked when a task becomes runnable;

- *dequeue_task*: invoked when the task is no more runnable;
- *yield_task*: the task deliberately leaves the processor in favor of another task;
- *check_preempt_curr*: when the task that wakes up and the currently executing task are in the same class, this function checks if a preemption is required;
- *pick_next_task*: returns the task that the scheduling class wants to be executed, depending on its internal scheduling policy;
- *put_prev_task*: the currently running task is going to be replaced by another one;
- *task_tick*: invoked for updating task parameters;
- *task_fork*: a new task was spawned;
- *task_dead*: the task died;
- *prio_changed*: the scheduling parameters were changed;
- *switched_from*: the task is leaving this scheduling class;
- *switched_to*: the task is chosen to be scheduled with this scheduling class;
- *update_curr*: updates the task's parameters.

The *sched_class* data structure defines also the pointer to the next scheduling class: the *next* field.

2.2.2 task_struct

The kernel uses the *task_struct* data structure for storing all the data required for managing a task. Important fields are:

- *pid*: the task identifier;
- *state*: defines the task status:
  - −1: not runnable;
  - 0: runnable;

  - > 0: stopped;

- *sched_class*: pointer to the scheduling class used for scheduling the task;
- *dl*: scheduling entity for the "dl" scheduling;
- *policy*: scheduling policy.

2.2.3 rq

The processor has associated a runqueue data structure, containing the runqueues of all the scheduling classes.

This modular approach allows the runqueue to be independent from the scheduling classes implementations of their runqueues.

The processor's runqueue is defined by the *rq* data structure. In multiprocessor systems, there is a runqueue for each processor.

Chapter 3

SCHED_DEADLINE

This chapter describes the Linux kernel's SCHED_DEADLINE scheduling class, by showing how the services it provides are implemented and how the user can interact with them.

The first part describes the main data structures and functions managed by SCHED_DEADLINE.

The second part shows how those functions and data structures are able to implement the H-CBS scheduling policy.

The last part describes how the SCHED_DEADLINE class can be used by the user, so, which are the parameters that the user can set and the system call used to notify the kernel the wish to switch scheduling class.

3.1 Introduction

SCHED_DEADLINE is a scheduling class provided by the Linux kernel that implements a resource reservations policy which follows the H-CBS algorithm.

As already shown, the H-CBS lies on top of an EDF scheduling, so, this scheduling class was originally named SCHED_EDF. The name was later modified to SCHED_DEADLINE for keeping the class implementation-independant.

All the next sections will take as reference the SCHED_DEADLINE code included in the Linux kernel versions that goes from *3.14* to *3.19*.

3.2 Data Structures and Functions

The next sections describe which are the data structures that are involved in the SCHED_DEADLINE algorithm and the functions from which those are managed.

3.2.1 sched_dl_entity

The *sched_dl_entity* data structure contains all the data required by SCHED_DEADLINE for managing the tasks.

This structure contains the fixed SCHED_DEADLINE parameters, chosen by the user side when performing the *sched_setattr()* system call:

- *dl_runtime*: maximum runtime for each instance;
- *dl_deadline*: relative deadline of each instance;
- *dl_period*: separation of two instances;
- *dl_bw*: the result of the division of *dl_runtime* by *dl_deadline.*

It also defines some dynamic data, which is continuously updated during task execution.

Those data are:

- *runtime*: remaining runtime;
- *deadline*: next absolute deadline;

- *flags*: specify the scheduler behavior;
- *dl_throttled*: is set when the budget has been exhausted, so, the task is waiting to get again in a ready queue;
- *dl_new*: tells that this is a new instance, so it must be correctly initialized with full runtime and the correct absolute deadline;
- *dl_boosted*: tells if the task priority has been boosted due to deadline inheritance. If so, under the critical section, the task overcomes the bandwidth enforcement mechanism;
- *dl_yielded*: the task left the CPU before finishing its budget during its last job. This flag can be used in future implementations for budget reclaiming;
- *dl_timer*: each task has its own timer for implementing the bandwidth enforcement mechanism.

3.2.2 dl_rq

This runqueue contains the runnable SCHED_DEADLINE tasks.

Those tasks are managed in this data structure depending on their absolute deadline: the priority depends on the absolute deadline, so, the runqueue must be organized in such a way that the element with smallest absolute deadline should be easily accessible.

Then, the operations that will be performed on this data structure are:

- insertion of a new element;
- removal (usually) of the task with smallest absolute deadline;

Implementing this data structure as a list requires an insertion complexity of $O\left(n\right)$ and a removal complexity of $O\left(1\right)$

A well-performing implementation for managing this data structure is through a red-black tree.

Red-black trees have a complexity of $O\left(\log\left(n\right)\right)$ for any operation performed on the data structure. Moreover, by considering that the red-black tree's lower value element is the the leftmost leaf, it is possible to develop some mechanism to reduce the operation for managing the leftmost element to $O\left(1\right)$ complexity.

3.2.3 dl_sched_class

All the function pointers for the SCHED_DEADLINE *sched_class* structure, introduced in subsection 2.2.1, are statically allocated by the *dl_sched_class* declaration.

Those hooks are:

- enqueue_task (enqueue_task_dl),
- dequeue_task (dequeue_task_dl),
- yield_task (yield_task_dl),
- check_preempt_curr (check_preempt_curr_dl),
- pick_next_task (pick_next_task_dl),
- put_prev_task (put_prev_task_dl),
- set_curr_task (set_curr_task_dl),
- task_tick (task_tick_dl),
- task_fork (task_fork_dl),
- task_dead (task_dead_dl),
- prio_changed (prio_changed_dl),
- switched_from (switched_from_dl),
- switched_to (switched_to_dl),
- update_curr (update_curr_dl),

The *sched_class* data structure also contains the pointer to the next scheduling class: *rt_sched_class*.

In the next pages will be described the *dl_sched_class* structure elements with a peculiar behavior.

yield_task_dl()

This function suspends the running task by consuming all its budget. The next activation will happen at next deadline.

This function may be useful in budget reclaiming algorithms, currently not yet implemented by SCHED_DEADLINE.

check_preempt_curr_dl()

This function is called when the task that wakes up and the currently executing task are both SCHED_DEADLINE tasks.

It checks if the absolute deadline of the incoming task is smaller than the executing task and, if so, performs the preemption.

For SMP systems, the problem arises when the new task has a deadline equal to the deadline of the executing task in that ready queue. In that case, depending on the CPU affinity, it is chosen if the task has to be scheduled and which is the best CPU to use.

pick_next_task_dl()

This function returns the highest priority task from the SCHED_DEADLINE scheduling class.

If no runnable task is available, then NULL is returned.

task_tick_dl()

This function is periodically invoked to update SCHED_DEADLINE parameters.

It calls the *update_curr_dl()* function.

task_fork_dl()

This function is not directly implemented, because SCHED_DEADLINE tasks are not allowed to fork.

The fork operation can be achieved through the *sched_fork()* function.

task_dead_dl()

The task timer is deleted and the total available bandwidth is freed from the bandwidth occupied by the task.

switched_from_dl()

This function is similar to *task_dead_dl()*.

switched_to_dl()

A new SCHED_DEADLINE task arrived, so, it overload the runqueue.

For this reason, it tries to push some task off from it, if possible.

update_curr_dl()

This function calculates the execution time of the task and updates its budget. If the task exhausted its budget, then is preempted in favor of the next highest priority task, by performing a reschedule operation.

If the task is preempted for budget exhaust (*suspended* state), the task's *dl_timer* is started to fire at the task deadline. When the *dl_timer* fires, then the *dl_task_timer()* function is executed.

3.3 H-CBS in SCHED_DEADLINE

The next paragraphs will explain how SCHED_DEADLINE implements the concepts introduced in chapter 1.1.2.

In SCHED_DEADLINE exists a strong relationship between tasks and servers: each SCHED_DEADLINE server must have one and only one associated task. For this reason, in SCHED_DEADLINE, the terms "task" and "server" can be used with the same meaning.

When a task is getting scheduled with SCHED_DEADLINE, the kernel initializes all the required data structures and decides if the currently running task must be preempted.

The preemption depends on:

- the comparison between the deadlines: if the newly arrived task has a deadline smaller the the running task, then it wins the CPU;
- the scheduling class of the running task: SCHED_DEADLINE is the highest priority class, after the *stop* class.

While the tasks are executed, the kernel prioriodically updates server budgets, by calling the *update_curr_dl()* function. This function subtracts the execution time of the task from the server budget. After that, it is checked if the task exhausted its budget. If the budget is less than or equal to zero, this condition is verified, then the task becomes throttled and is removed from the runqueue associated to SCHED_DEADLINE. At this point, the task's timer is set with the *start_dl_timer()* function to fire at the task's deadline.

When the timer fires, it runs the *dl_task_timer()*. This function runs only when the task was throttled and needs a budget replenishment, so, the task is enqueued into the SCHED_DEADLINE runqueue with the *ENQUEUE_REPLENISH* flag. Thanks to this flag, the server budget is replenished and the task is again able to run. The replenishment not only refills the budget, but also updates the absolute deadline of the task. In particular, the kernel should increment the value of the deadline with the value of its period.

It may happen, in case of a low frequency of task parameters update, that the budget is still negative, also after the replenishment. The replenishment is then implemented as a loop that:

- increments the deadline d with the period value P;
- increments the actual budget q with the maximum task budget Q.

until the budget switches back to a positive value.

Then, it is checked if the replenished task is the highest priority task, and, if it is, a preemption will occur.

3.3.1 Admission Control

The necessary and sufficient condition for the schedulability of EDF tasks is to have a CPU utilization factor less than or equal to 1.

The current utilization factor is updated by SCHED_DEADLINE everytime a task enters or leaves this scheduling class. Before allowing a task to be scheduled by SCHED_DEADLINE, the kernel checks if the new task's utilization factor, added to the sum of the utilization factors of the other scheduled tasks, is below a certain threshold. This threshold is chosen as 0.95, slightly lower than the theoretical maximum value of 1.

3.3.2 Parameters Constraints

As will be described later in section 3.4, the user performs a system call for switching its tasks' scheduling class. This system call requires also the SCHED_DEADLINE parameters required by the H-CBS server.

When the chosen scheduling class is SCHED_DEADLINE, the server parameters are checked to be correct.

This parameters validation is performed by the function *__checkparam_dl()*, defined in *kernel/sched/core.c*.

The basic rules are the following:

- $deadline > 0$;
- $runtime > 2^{DL\_SCALE} = 2^{10} = 1024$
- if $period > 0$, then $runtime \leq deadline \leq period$.

If at least one of the rules is not followed, the system call returns an error.

3.4 Userspace Perspective

This section describes the steps that the programmer follows to define the scheduling class of a task to SCHED_DEADLINE.

3.4.1 Switching Scheduling Class

A task cannot directly be scheduled with SCHED_DEADLINE: it has to explicitly switch scheduling class by using the proper system call.

This is performed by the *sched_setattr()* function, that will be provided by a future release of *sched.h* library.

Listing 3.1 : sched_setattr

```
#include <sched.h>
int sched_setattr(pid_t pid,
                  const struct sched_attr *attr,
                  unsigned int flags);
```

In the current implementation, this function updates the scheduling attributes defined by *attr* to the task identified by *pid*. The name *pid* is fraintendible, because recalls the Process ID, while the scheduling class can be set to any kind of task: processes or threads. Then the correct name should be *tid*, recalling the Thread ID, which identifies the entity that is going to be scheduled. The just mentioned *tid* has no relationship with the POSIX Thread ID.

The *flags* field has been left for the future: currently it has no meaning and is left to 0.

The *attr* field is a *sched_attr* structure defined as follows.

Listing 3.2 : sched_attr

```
struct sched_attr {
        u32 size;               /* Size of this structure */
        u32 sched_policy;       /* Policy (SCHED_*) */
        u64 sched_flags;        /* Flags */
        s32 sched_nice;         /* Nice value (SCHED_OTHER,
                                SCHED_BATCH) */
        u32 sched_priority;     /* Static priority (SCHED_FIFO,
                                SCHED_RR) */

        /* Remaining fields are for SCHED\_DEADLINE */

        u64 sched_runtime;
        u64 sched_deadline;
        u64 sched_period;
};
```

The relevant fields of this data structure are

- *size*: the size of this data structure;
- *sched_policy*: the desired scheduling class, so, it is possible to use the value defined by *SCHED_DEADLINE* or directly use its value: 6;
- *sched_runtime*: the server budget Q;
- *sched_deadline*: the server relative deadline D;
- *sched_period*: the server period P.

In case of success, the function returns 0, otherwise returns -1 and sets the *errno* value as follows:

- *EBUSY*: the admission control failed;
- *EINVAL*: the parameters chosen by the user are invalid.

Chapter 4

H_CBS_SO

This chapter describes a more detailed picture of the limitations demonstrated by the H-CBS algorithm. It also presents the new H-CBS-SO algorithm and the adaptations of SCHED_DEADLINE required for its implementation.

The first part recalls the issues that are present in the H-CBS scheduling policy and introduces the considerations that drove the development of H-CBS-SO algorithm. This is done by showing the scheduling difficulties that arise in the mamagement of self-suspending tasks.

The second part provides a full description of the H-CBS-SO algorithm.

The last part shows how SCHED_DEADLINE scheduling class has been modified to implement the H-CBS-SO policy.

4.1 Problem and Solution

As briefly introduced in section 1.2, the H-CBS algorithm has strong limitations in managing real systems.

The next sections will go deeper in detail of what are the issues of this algorithm by providing a scheduling example.

In the end, will be shown a possible solution for solving this problem.

4.1.1 Unverified Schedulability Condition

Consider an H-CBS based system, scheduling two tasks:

- τ_1:
  - $C = 2$;
  - $P = 5$;
  - $D = 5$;
  - managed by server S_1:
    - $Q = 2$;
    - $P = 5$;
- τ_2:
  - $C = 4$;
  - $P = 10$;
  - $D = 10$;
  - managed by server S_2:
    - $Q = 4$;
    - $P = 10$;

This taskset should result schedulable under EDF:

$$\sum_{i=1}^{2} \frac{C_i}{P_i} = \frac{2}{5} + \frac{4}{10} = 0.8 \leq 1$$

The scheduling of this taskset is shown in figure 4.1. In particular, at time:

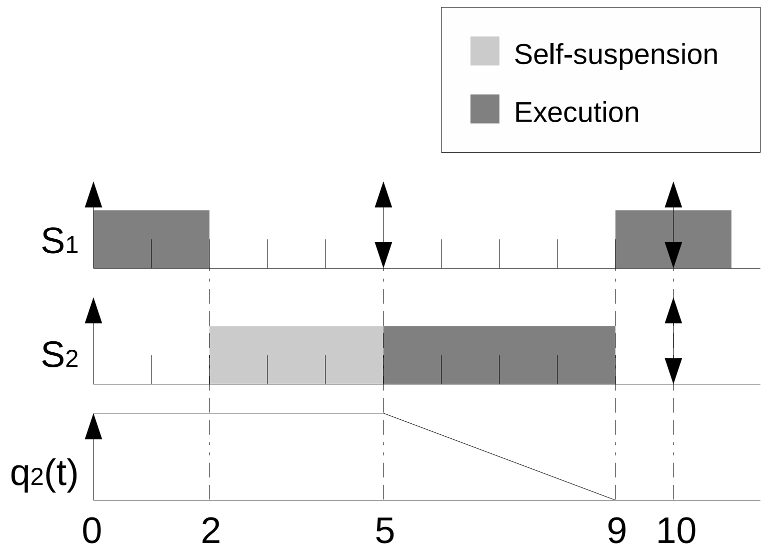


Figure 4.1: Deadline miss with H-CBS algorithm.

- $t = 0$: tasks τ_1 and τ_2 are activated and τ_1 starts executing because of its earlier deadline;

- $t = 2$: tasks τ_2 completes its job and τ_2 starts, but its first operation self-suspends the task. Then, the task leaves the CPU and the server keeps the budget unchanged;

- $t = 5$: τ_2 turns back to a runnable state just before the arrival of τ_1. For this reason, there's no reason for preemption and τ_2 continues its execution;

- $t = 9$: τ_2 completes its execution and leaves the CPU to τ_1;

- $t = 10$: τ_1 misses its deadline.

This example shows how a self-suspending task may cause the deadline miss of another task. In particular, even though the guilty task is τ_2, the task which misses its deadline is τ_1.

It is then demonstrated that the necessary and sufficient condition for the schedulability of a taskset under EDF cannot be applied to the H-CBS policy.

4.1.2 Suspension Oblivious Analysis for H-CBS

A possible approach for verifying the schedulability of the system could be to consider also the WCET of the self-suspension (by now, identified with S_i) in the server budgets.

The new checking will result as:

$$U = \sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{Q_i}{P_i} = \sum_{i=1}^{N} \frac{C_i + S_i}{P_i} \leq 1$$

This approach is able to recognize that the taskset shown in section 4.1.1 is not schedulable:

$$\sum_{i=1}^{N} \frac{C_i + S_i}{P_i} = \frac{2}{5} + \frac{7}{10} = 1.1 > 1$$

This is equivalent of considering the self-suspension as a busy wait.

The example seen in section 4.1.1 can be modified by transforming the self-suspension into a busy wait and the resulting schedule is shown in figure 4.2

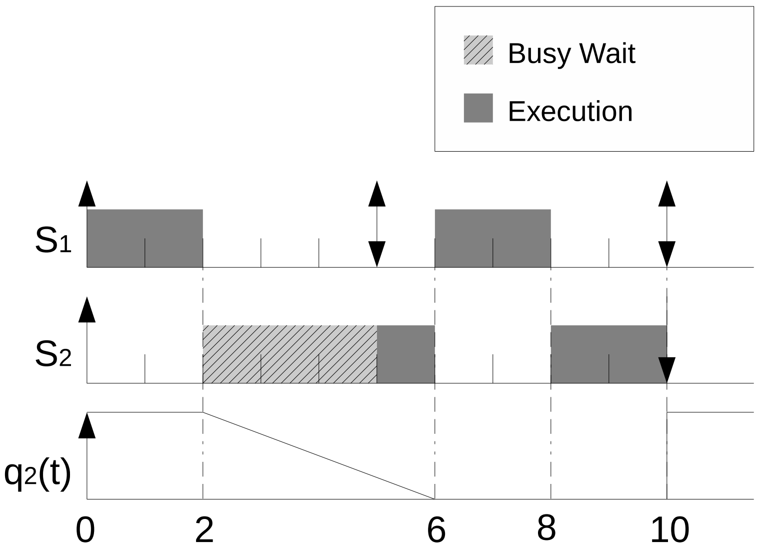


Figure 4.2: H-CBS algorithm schedule with busy wait.

In this case, τ_2 misses its deadline instead of τ_1. This is because of the resource reservation policy: τ_2 tried to use more budget than expected, so its server suspends, providing temporal isolation among tasks.

Considering self-suspensions as busy waits is a strongly pessimistic assumption, because the CPU remains inactive also if there are other runnable tasks.

Moreover, the final result of using busy waits instead of self-suspensions is a degradation of the performances in terms of response times. Another problem of this approach is that the system resources, in particular the CPU utilization factor, are wasted in vain.

4.1.3 Managing Self-Suspending Tasks Budget

Merging the considerations made in section 4.1.1 and section 4.1.2, a new scheduling policy which is able to manage self-suspending tasks without the need of migrating to busy execution is required.

The idea is to create a mechanism that allows the preemption of self-suspended tasks in favor of other runnable tasks to obtain good response time performances. At the same time, for providing temporal isolation, self-suspended tasks' budgets must be consumed also if those tasks are not executing.

It is possible to notice that the problem of interference between self-suspended tasks arises only when the suspended task allows other tasks, with lower or equal priority, to run.

In other words, the budget of the self-suspended task should be consumed when, in case of implementing the self-suspension with a busy wait, that task would be executed.

4.2 The H-CBS-SO Algorithm

The H-CBS-SO algorithm uses the considerations made in section 4.1.3 for managing the server budgets in a more strict manner with respect to H-CBS, but avoiding the use of busy waits instead of self-suspensions.

The figure 4.3 shows the new transitions introduced by the H-CBS-SO algorithm.

- *A-transition*: a running task self-suspends;
- *B-transition*: a self-suspended task becomes ready;
- *C-transition*: a self-suspended task becomes suspended;

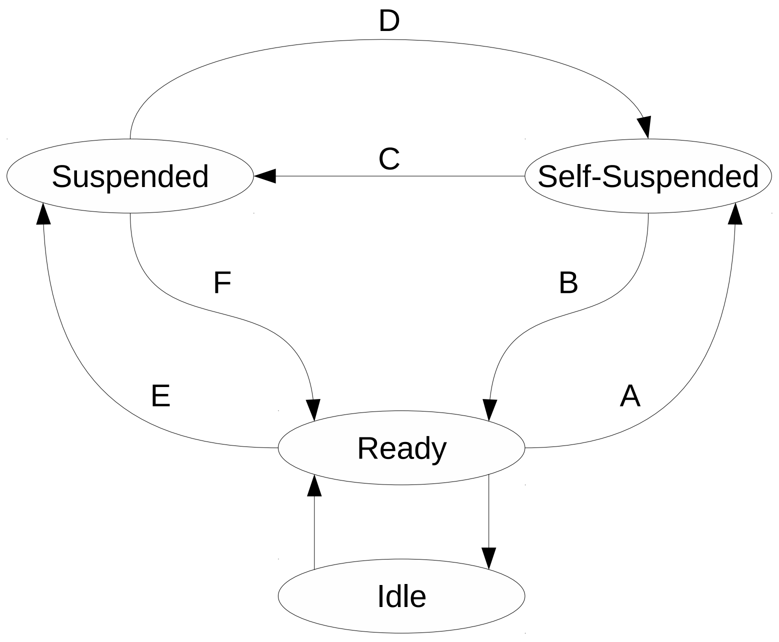


Figure 4.3: State transition diagram for the H-CBS-SO algorithm. [2]

- *D-transition*: a suspended task returns self-suspended.

The following chapters will describe the states that an H-CBS-SO server can reach and the transitions that may happen, following the rules presented in [2].

4.2.1 Idle

Initially the server is idle and its budget and period are null.

The transitions from the *idle* state happen when a job arrives, but because of the strong relationship between tasks and servers in SCHED_DEADLINE (each task has its own assciated server), this state is never reached.

4.2.2 Ready

The server is ready and can be executed.

Whenever a task reaches this state, its descriptor is inserted in the ready queue, awaiting for the CPU.

A-transition

A task suspends itself, waiting some event.

E-transition

When a task exhausts its budget, it becomes suspended.

4.2.3 Self-Suspended

Whenever a task self-suspends, it is inserted into the SS_QUEUE.

When a task executes, its budget is consumed accordingly. If the task in the head of the SS_QUEUE has equivalent or higher priority than the executing task, its budget is decreased of the same amount.

B-transition

When a task resumes its execution, it must be removed from the SS_QUEUE. The associated server becomes ready and the task is inserted into the ready queue.

C-transition

If a self-suspended server exhaust its budget, it switches to the *suspended* state and is removed from the SS_QUEUE.

A flag that keeps trace of this transition must be set. This flag will be used for choosing the exit transition from the *suspended* state.

4.2.4 Suspended

A server is in *suspended* state when it exhausted its budget and is waiting for a replenishment.

When the server's budget is replenished, the flag described in *C-transition* is checked for choosing the next transition.

D-transition

The server was suspended from a self-suspension and is still suspended.

So, it returns back to the self-suspended state and is inserted in the SS_QUEUE.

F-transition

This transition happens if the server budget is replenished and the server is not (or no more) suspended.

4.3 Implementation

This section describes how the code of the Linux kernel has been adapted to the H-CBS-SO resource reservation policy.

4.3.1 SS_QUEUE

Allocation

For allocating the SS_QUEUE, it is possible to use the *DEFINE_PER_CPU_SHARED_ALIGNED* macro provided by the kernel. This is a good choice for future implementations, because it creates an SS_QUEUE for every processor in the system. The current H-CBS-SO algorithm is developed for single core machines, but this macro makes it easy to be extended to multiprocessor systems.

This macro is used to allocate the *ss_queue* data structure, shown in subsection A.5.1, which contains a pointer to the root of the Red-Black tree and the pointer to the leftmost element, better described later.

sched_dl_entity

This data structure, already described in subsection 3.2.1, requires three more fields:

- *rb_ss_queue_node*: used for implementing the SS_QUEUE red-black tree;
- *in_ss_queue*: a pointer to the SS_QUEUE to which the task is associated;
- *dl_blocked*: a flag required for keeping trace of if the task is self-suspended or not.

The code of the final version of the data structure is shown in subsection A.1.1.

The *C* language guarantees a well-defined memory allocation of the data structures. Thanks to this property, it is possible to implement complex data structures, like lists, queues, trees not by creating a data structure which contains all the elements, but by modifying the elements themselves to contain the data structure properties.

And this is exactly what the *rb_ss_queue_node* is used for.

Red-Black Tree Management

The SS_QUEUE must be ordered by increasing absolute deadline, in such a way that the highest priority task is easily accessible. So, it is possible to use the red-black tree, previously introduced in subsection 3.2.2.

Red-black trees are self-balancing binary search trees, that are ensured to be kept approximately balanced.

The elements are disposed in the tree in such a way that, with an in-order traversal, the elements are returned in an ascending order. It means that all the left-children nodes must have a value lower than their parent and all the right-children nodes a higher value.

The elements required to create a red-black tree with the libraries provided by the Linux kernel are:

- the root of the tree: defined by *rb_tree* in *ss_queue*;
- the element of the tree: defined by *rb_ss_queue_node* in *sched_dl_entity*.

When inserting a new element, the tree must be traversed through the path which meets the just described rule.

The new element becomes finally a new leaf of the tree.

After any insertion, the tree is no more guaranteed to be balanced, so, a rebalancing operation must be performed. This is fortunately already implemented by the library.

The operations performed on the SS_QUEUE are:

- insertion: performed by *dl_ss_queue_insert()*;
- removal: performed by *dl_ss_queue_remove()*;

whose code is shown in subsection A.3.1.

The most important element for the SS_QUEUE is the value with the smallest absolute deadline, which, for the red-black tree ordering property, is always the leftmost leaf. For providing a quick access to this value, the SS_QUEUE is implemented with an additional field which points to its leftmost leaf.

When a new element is inserted in the tree, it becomes the new leftmost leaf if the traversal for the insertion never passed through a right-child node. When removing an element from the tree, if the removed element was the lefmost leaf, the pointer to the leftmost leaf is updated with the next element of an in-order traversal.

4.3.2 Detecting Self-Suspension

For implementing the *A-transition*, the crucial step is to identify the self-suspension of a task.

There are several possible approaches that can be followed for achieving this:

- modifying all the system calls that may suspend a process;
- checking at the common return point of the system calls;
- checking into the *dequeue_task_dl()* function of SCHED_DEADLINE.

Exhaustive System Calls Modification

The first option requires a high number of modifications to the kernel source for making all the system calls that may suspend a process compatible with this new scheduling algorithm.

There is also no guarantee that system calls that will be developed in the future will be correctly implemented for this purpose.

This solution provides a low code maintainability.

Common System Calls Return Point Improvement

The number of modifications can be reduced and code maintainability improved with the solution described by the second option.

This requires additional checkings into the piece of code that returns from the system calls, which is shared among all of them.

The problem of this solution is that, for every system call return, the kernel should also check if it caused the suspension of the task and if that task is scheduled with SCHED_DEADLINE.

All these operations are executef for every system call of the system, and so, will cause a big loss in terms of system performance.

SCHED_DEADLINE Runqueue Management Checking

The last option, which uses the event of removing tasks from the SCHED_DEADLINE ready queue, is chosen for tracing the self-suspension.

It has the following advantages:

- *dequeue_task()* is surely associated to SCHED_DEADLINE tasks, and then, no additional scheduling class checking is required;
- it is easy to understand if what caused the removal from the ready queue is due to a self suspension or not.

The *dequeue_dl_entity()* is the function that will be used as bottleneck for checking if a SCHED_DEADLINE task self-suspends.

This function is called in the following cases:

- the task runs out of budget;
- the task terminates;
- the task changes scheduling class;
- the task self-suspends.

So, additional checkings are required to confirm that the task was remove from the runqueue because of a self-suspension or not.

The final code developed for implementing the *A-transition* is shown in subsection A.3.8.

4.3.3 Exhausting Budget While Self-Suspended

A self-suspended task may exhaust its budget, switching from the *self-suspended* state to a *suspended* state through the *C-transition.*

A new field for keeping trace of this transition is required, because, when the budget is replenished, the kernel must bring the task back to the *self-suspended* state.

The field responsible for this transition is *dl_blocked*, defined in *sched_dl_entity* data structure and set during the *C-transition.*

Those operations are performed when the budget is being updated, when the function *update_curr_dl()*, shown in subsection A.3.6. is executed.

4.3.4 Replenishing Budget

When a task is *self-suspended* and its timer causes a budget replenishment, the kernel must decide which transition will be chosen, depending on the *dl_blocked* value:

- if *dl_blocked* is set, then the task is still suspended and a *D-transition* occurs, bringing back the task to the *self-suspended* state;
- otherwise, the task is no more *self-suspended* and returns back to the *ready* state, by following the *F-transition.*

Those operations are performed by the *dl_task_timer()* function, shown in subsection A.3.4.

4.3.5 Leaving Self-Suspension

Another problem is to detect when a task is no more *self-suspended.*

This happens when a task which was suspended returns to the SCHED_DEADLINE ready queue, through the *enqueue_task_dl()* function.

This function has been modified for checking if the task was in the SS_QUEUE and, if so:

- if was *suspended*, then resets the *dl_blocked* flag, which will be used by the *dl_task_timer()* function. If *dl_task_timer()* finds the *dl_blocked* flag off will perform a *F-transition*;
- otherwise, brings the task back to the *ready* state through a *B-transition.*

The final code developed for implementing the *B-transition* or *F-transition* is shown in subsection A.3.7.

4.3.6 Dying Tasks

When a task dies, it is first dequeued with no particular flag, so this behavior is incorrectly interpreted as a self-suspension.

To solve this problem and keeping the SS_QUEUE consistent, when the function *task_dead_dl()* is invoked, the associated task must be removed from SS_QUEUE, if present.

This behavior is handled in the *task_dead_dl()* function, described in subsection A.3.9.

4.3.7 Updating budget of Self-Suspended Tasks

When a tasks executes, its budget is consumed, as happens for any resource reservation policy.

In H-CBS-SO, at the same time, it is consumed also the budget from the self suspended task with highest priority. It is equivalent to say that the budget is consumed also from the head of the SS_QUEUE.

This operation is performed by the *update_ss_queue()* function, called by *update_curr_dl()*.

Their codes are shown in subsection A.3.5. and subsection A.3.6.

Chapter 5

Periodic Tasks in Linux

This chapter describes the need of introducing the concept of periodic task in the Linux kernel and the design of a userspace library for its management.

The first part introduces the problems that arise in H-CBS-SO algorithm without the kernel awareness of the tasks' periodicity.

The second part shows which is the classical Linux programmers approach for developing periodic tasks.

In the last part will be presented a simple library exemple for managing periodic tasks.

5.1 Intro

The current Linux kernel does not provide any functionality for managing periodic tasks. The programmer is forced to implement periodic tasks completely from user level.

For this reason, the kernel is not able to distinguish if a running task is periodic or not. This is a problem for scheduling algorithms because there is no way to retrieve the next activation times of the jobs.

Moreover, as discussed in the next chapter, the waiting for next activation is obtained by the tasks with standard sleep operations. Those operations may be misinterpreted by the H-CBS-SO algorithm as self-suspensions. This self-suspension misunderstanding does not cause any problem to the H-CBS-SO algorithm, but introduces some overhead in the management of the SS_QUEUE, where the number of elements becomes usually greater than necessary. In other words, this results in a not so clean usage of the H-CBS-SO algorithm itself.

5.2 Classical Periodic Real-Time Tasks

The common implementation of a periodic real-time task in Linux is the following:

- *initialization*;
- compute *next activation* time;
- *infinite loop*, which:
  - performs some elaboration;
  - waits for *next activation*;
  - computes *next activation* time.

In the *initialization* phase, the task prepares the environment.

In the *infinite loop* phase, the task launches its jobs. Each job performs some operation.

In a control system, common operations are:

- reading data from sensors;
- elaborating data through a control algorithm;

- sending commands to actuators.

When the job finishes, the task must wait for the *next activation* time before launching the next job.

The operation of waiting for *next activation* is performed by using a *sleep* operation, for exemple through the standard *clock_nanosleep()*. This function is a good candidate, because preempts the task and removes it from the ready queue, allowing other tasks to execute.

5.3 Kernel-Aware Periodic Real-Time Tasks

For solving the problems described in subsection 5.1, the Linux kernel should implement proper system calls and mechanisms for managing periodic tasks.

At the same time, a userspace library would be useful to the programmer for easily implementing periodic real-time tasks.

5.3.1 Userspace Library

For creating a library that the programmer can use for implementing periodic tasks, it is possible to use an interface similar to the one provided by the POSIX Threads.

The functionalities required for the periodic tasks management are:

- data structures for defining periodicity and scheduling class parameters;

- periodic task section: this function must periodically execute the jobs of the task;

- periodic section exit: when for some reason the periodic part of the task needs to terminate, it must return a value to its parent.

Data Structures

The element necessary for creating a periodic task is its period.

There should be no limitation on the period that a task can choose, so, it is possible to use the timespec structure defined by *time.h* library:

Listing 5.1 : timespec

```
struct timespec {
        time_t tv_sec;          /* seconds */
        long   tv_nsec;         /* nanoseconds [0 .. 999999999] */
};
```

Moreover, it can be useful to directly change the task's scheduling class for its periodic section. As introduced in subsection 3.4, the *sched_attr* data structure contains all the parameters required for choosing the new scheduling class and for setting its parameters.

It is then possible to create the *periodic_attr_t* data structure as follows.

Listing 5.2 : periodic_attr_t

```
struct periodic_attr_t {
        timespec period;
        timespec deadline;

        sched_attr s_attr;
};
```

This data structure will be easily

Creating Periodic Sections

The idea is to follow a Pthread-style library function for simplifying the programmer migration to this coding style.

The function that creates the periodic section will result similar to the *pthread_create()* function.

Listing 5.3 : pthread_create()

```
int pthread_create(pthread_t *thread,
                const pthread_attr_t *attr,
                void *(*start_routine)(void*),
                void *arg);
```

So, the *periodic_create()* function is required.

Listing 5.4 : periodic_create()

```
int periodic_create(const periodic_attr_t *p_attr,
  int (*routine)(void*),
  void *arg);

int periodic_create(const periodic_attr_t *p_attr,
  int (*routine)(void*),
  void *arg)
{
  int ret;
  sched_attr *s_attr = p_attr->s_attr;
  sched_attr attr_old;
  int flags = 0;

  get_sched_attr(&attr_old);

  set_sched_attr(s_attr);

  set_periodic_attr(p_attr, flags);

  while (ret = routine(arg))
    periodic_wait();

  periodic_clear();

  set_sched_attr(&attr_old);

  return ret;
}
```

This function's first step is to change the scheduling parameters for the current tread. The user has the freedom of choosing its desired scheduling class, without the restriction of using SCHED_DEADLINE.

Then, it invokes the system call *set_periodic_attr()*, required for notifying the kernel that the current thread is going to run a periodic section.

After that, it runs periodically the specified *routine*, whose parameters are passed through the *arg* field, until the *routine* returns the 0 value.

Every cycle is automatically terminated with the *periodic_wait()* system call, which suspends the task until the activation of the next job.

When the periodic section terminates, the kernel must be advertised, so, the *periodic_clear()* system call is called for cleaning all the kernel data structures associated with it.

Usage

The following piece of code shows how to create a periodic section inside a task.

The periodic section terminates when the task receives an external signal.

Listing 5.5 : periodic_create() usage

```
#include <periodic.h>
#include <signal.h>

int go_on;

void periodic_sig_handler(int signo)
{
  if (signo == SIGPERSTOP)
    go_on = 0;
}

int periodic_body(void *arg)
{
  // Read data from sensors
  // Apply control algorithm
  // Send commands to actuators

  return go_on;
}

int main()
{
  periodic_attr_t attr;
  int arg = 0;

  if (signal(SIGPERSTOP, sig_handler) == SIG_ERR) {
    printf("\ncan't␣catch␣SIGPERSTOP\n");
    return -1;
  }

  go_on = 1;

  attr.period.tv_sec = 1;
  attr.period.tv_nsec = 0;

  attr.deadline.tv_sec = 1;
  attr.deadline.tv_nsec = 0;

  attr.s_attr.size = sizeof(attr.s_attr);
  attr.s_attr.sched_policy =   SCHED_DEADLINE;
  attr.s_attr.period = 1 * 1000 * 1000;
  attr.s_attr.deadline = 1 * 1000 * 1000;
  attr.s_attr.runtime = 1000 * 1000;

  // Set periodic task parameters
  // Set periodic function argument
  periodic_create(&attr, periodic_body, &arg);

  return 0;
}
```

By using this library, the user will be also able to create periodic sections inside multiple threads, so, a single process will be able to spawn several periodic sections running at the same time.

Chapter 6

Tools

This chapter describes important tools that can help the Linux kernel programming.

The first part presents the *Ftrace* tool, useful for getting kernel events and kernel functions execution times.

The second part shows *trace-cmd*, a tool that interacts *Ftrace* for storing the kernel traces. This provides also a graphical user interface called *Kernelshark* for visualizing those events.

In the end, will be described an utility that executes a set of self-suspending tasks that are scheduled by SCHED_DEADLINE for verifying the correctness of the H-CBS-SO implementation.

6.1 Ftrace

Ftrace is a tracer provided by the Linux kernel that helps developers and designers to retrieve important data from the kernel, for understanding its behavior.

This tool integrates several tracing utilities, including the kernel events tracing and kernel function profiling.

6.1.1 Kernel Events Tracing

For retrieving events from the Linux kernel, especially for statistical or debugging purposes, the *Ftrace* system will be used.

Ftrace uses the *debugfs* file system for holding:

- the control files for the tracing system management;
- the buffer files from which the traces can be retrieved.

In particular, all these files can be accessed from the */sys/kernel/debug/-tracing* directory.

Inside this folder there are two important files: *trace* and *trace_pipe*.

The *trace* file holds the output of the trace in a human readable format, while the *trace_pipe* file is a sequential file, where each read operation consumes the data. This last file purpose is to provide traces to tools that perform live tracing operations.

An example of the content of the *trace* file is the following.

Listing 6.1 : */sys/kernel/debug/tracing/trace* file output

```
# tracer: nop
#
# entries-in-buffer/entries-written: 154/154   #P:8
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          <idle>-0     [002] d...  3092.423854: sched_switch: pre...
       trace-cmd-6624  [002] d.s.  3092.423862: sched_wakeup: com...
       trace-cmd-6624  [002] d...  3092.423866: sched_switch: pre...
          <idle>-0     [002] d...  3092.438571: sched_switch: pre...
  kworker/u16:6-196   [002] d...  3092.438584: sched_switch: pre...
          <idle>-0     [000] d...  3092.443778: sched_switch: pre...
       trace-cmd-6622  [000] d.s.  3092.443785: sched_wakeup: com...
       trace-cmd-6622  [000] d...  3092.443788: sched_switch: pre...
          <idle>-0     [002] d...  3092.443830: sched_switch: pre...
  kworker/u16:6-196   [002] d...  3092.443837: sched_switch: pre...
          <idle>-0     [000] d...  3092.445849: sched_switch: pre...
chromium-browse-5817  [000] d...  3092.445889: sched_wakeup: com...
chromium-browse-5817  [000] d...  3092.445944: sched_switch: pre...
```

Creting New Traces

For verifying the correctness of the SCHED_DEADLINE with H-CBS-SO implementation, it is necessary to have a set of traces that are triggered for every transition of the server state, as shown in chapter 4.2.

Those traces are the following:

- H-CBS-SO transitions:
  - *A-transition*: *trace_sched_dl_to_ss()*;
  - *B-transition*: *trace_sched_dl_from_ss()*;
  - *C-transition*: *trace_sched_dl_ss_to_susp()*;
  - *D-transition*: *trace_sched_dl_ss_from_susp()*;
  - *E-transition*: *trace_sched_dl_to_susp()*;
  - *F-transition*: *trace_sched_dl_from_susp()*.
- H-CBS-SO events:
  - a task is inserted in the SS_QUEUE: *trace_sched_dl_ss_queue_new()*;
  - a task is removed from the SS_QUEUE: *trace_sched_dl_ss_queue_delete()*;
- original SCHED_DEADLINE:
  - the task ran out of budget: *trace_sched_dl_to_ss()*;
  - the server budget has been replenished: *trace_sched_dl_fillbudget()*;
  - a new task is being scheduled with SCHED_DEADLINE: *trace_sched_dl_new()*;

All those traces are implemented by adding new entries to *include/-trace/events/sched.h*, using the macros provided by the Linux kernel as shown in subsection A.4.1.

6.1.2 Kernel Functions Profiling

Another important feature provided by Ftrace is the function profiling.

Ftrace allows to get statistics on functions execution times and on the number of times the functions were called.

This functionality is important to verify kernel function performance and to check which are the most called functions.

The following is an example of the output generated by the function profiler provided by Ftrace.

Listing 6.2 : ***/sys/kernel/debug/tracing/trace_stat/function0* file output**

```
Function              Hit   Time            Avg         s^2
--------              ---   ----            ---         ---
update_curr_dl    1147913   144970.1  us    0.126  us   26459.15  us
dequeue_task_dl     76113   35443.80  us    0.465  us   466.768  us
dl_task_timer       46304   29558.98  us    0.638  us   660.723  us
enqueue_task_dl    122420   20011.60  us    0.163  us   394.298  us
```

The Ftrace code has been improved as shown in subsection A.6 for providing also the minimum and maximum function execution times.

6.2 trace-cmd and Kernelshark

Kernelshark is a graphical tool that comes together with *trace-cmd* software.

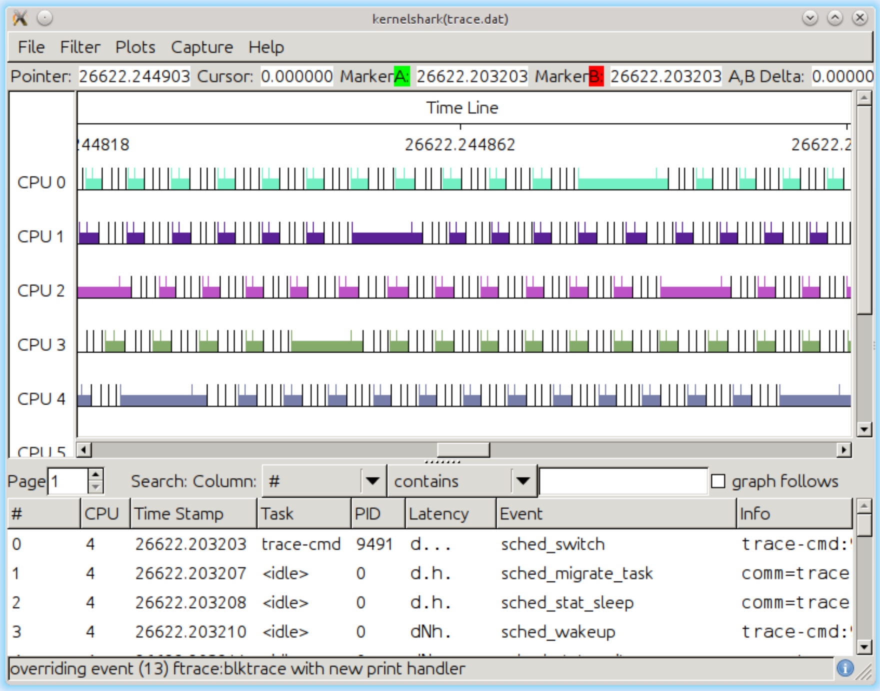
These are useful for recording kernel traces and visualizing them through an interactive interface, as shown in figure 6.1.

The interface shows not only the events that are generated by the kernel, but also the status of the task. For this reason, this tool should be improved for managing the new statuses introduced by H-CBS-SO. Moreover, for SCHED_DEADLINE tasks, it is really important to also plot the budget of the servers.

6.3 Task Spawner

Experimental tests must be performed for testing or debugging purposes. Those tests require a tool for launching the correct taskset.

For this purpose, a specific tool called *spawner* has been developed.



Pointer: 26622.244903 Cursor: 0.000000 Marker A 26622.203203 Marker B 26622.203203 A,B Delta: 0.00000

44818 26622.244862 26622.2

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	4	26622.203203	trace-cmd	9491	d...	sched_switch	trace-cmd:'
1	4	26622.203207	<idle>	0	d.h.	sched_migrate_task	comm=trace
2	4	26622.203208	<idle>	0	d.h.	sched_stat_sleep	comm=trace
3	4	26622.203210	<idle>	0	dNh.	sched_wakeup	trace-cmd:'

overriding event (13) ftrace:blktrace with new print handler

Figure 6.1: Kernelshark Graphical User Interface.

This tool launches a certain taskset composed by periodic tasks scheduled by SCHED_DEADLINE, whose parameters are passed to through JavaScript Object Notation (JSON) configuration files.

The parameters that must be defined for each task are the following:

- *jobs*: the number of jobs that the task executes;
- *ss_every*: the self-suspension happens every *ss_every* jobs;
- *ss*: the relative duration of the self-suspension;
- *c0*: the absolute busy wait before the self-suspension;
- *c1*: the absolute busy wait after the self-suspension;
- *period*: temporal distance between the activation of two consecutive jobs;
- *deadline*: relative deadline;

- *s_period*: server period;
- *s_deadline*: server deadline;
- *s_runtime*: server budget.

The following is an example of a JSON configuration file that can be parsed by the *spawner*.

Listing 6.3 : spawnerConfigurationExample.json

```
{
        "tasks" : {
                "thread0" : {
                        "jobs":         10000.0,
                        "ss_every":     0.0,
                        "ss":           79514.0,
                        "c0":           15902.0,
                        "c1":           15902.0,
                        "period":       447270.0,
                        "deadline":     447270.0,
                        "s_period":     447270.0,
                        "s_deadline":   447270.0,
                        "s_runtime":    119272.0
                },
                "thread1" : {
                        "jobs":         10000.0,
                        "ss_every":     0.0,
                        "ss":           1440163.0,
                        "c0":           288032.0,
                        "c1":           288032.0,
                        "period":       4050460.0,
                        "deadline":     4050460.0,
                        "s_period":     4050460.0,
                        "s_deadline":   4050460.0,
                        "s_runtime":    2160245.0
                }
        }
}
```

The *spawner* receives the configuration files from the *stdin* stream, parses the data and fills the data structures associated to each task. At that point, it can generate the tasks.

Each task is composed by two main parts:

- *initialization*: the task initializes its parameters and switches its scheduling class to SCHED_DEADLINE;
- *body*: the task performs periodically some operations as a common periodic task does.

The *body* is a loop of *jobs* cycles, which performs periodically, with period equal to *period*, the following actions:

- busy waits for *c0* nanoseconds;
- self-suspends for *ss* nanoseconds.
- busy waits for *c1* nanoseconds

The busy wait operations are implemented by cyclically checking the CLOCK_THREAD_CPUTIME_ID, until the desired time is reached. This clock is the CPU-time clock of the calling thread, representing the amount of execution time of the thread associated with the clock. This is a simple approach for ensuring the task to work for a fixed time.

For implementing the self-suspension, the thread must perform some action that causes a preemption. In such a way, the scheduler is able to put in execution tasks with lower priority. The function that is going to be used is *clock_nanosleep()*, using CLOCK_MONOTONIC clock. This is a good candidate for simulating the locking on mutexes or hardware resource, whose release time depends on external phenomena.

6.4 Generator

The taskset parameters are randomly generated by a tool called *generator*.

This takes as input:

- the total desired CPU utilization factor U_{tot};
- the minimum job utilization factor U_{lb};
- the period range $[T_{min}, T_{max}]$;
- the number of tasks;
- the number of jobs for each task;

The results are sent to the *stdout* stream in a JSON format, containing all the tasks parameters manageable by the *spawner*.

6.4.1 Generating Utilization Factors

To generate the set of U_i values for n tasks, the first step is to generate random u_i values in the range $[0, 1]$. Let's suppose that those values are ascending ordered, so, the minimum value will be u_1.

There are two constraints for generating the correct u_i values. The lower bound constraint

$$\min_i \{u_i\} \geq U_{lb}$$

and the total utilization factor constraint

$$\sum_{i=1}^{n} u_i = U_{tot}$$

For meeting the constraints, a possible way is to increase all the U_i of a certain constant value q and normalize all the set for a certain scale factor m.

The utilization factor constraint becomes

$$\sum_{i=1}^{n} (u_i + q)\, m = U_{tot}$$

and the lower bound constraint

$$(u_1 + q)\, m \geq U_{lb}$$

In order to ensure the minimum value to u_1, this inequality becomes equality and, by manipulating this constraint, it is possible to obtain the q value.

This value still depends on m

$$q = \frac{U_{lb}}{m} - u_1$$

Now, substituting this result to the utilization factor constraint, the result is the following

$$\sum_{i=1}^{n} \left(u_i + \frac{U_{lb}}{m} - u_1 \right) m = U_{tot}$$

With simple manipulation, it is possible to obtain the m value

$$m\sum_{i=1}^{n} u_i + m\sum_{i=1}^{n}\frac{U_{lb}}{m} - m\sum_{i=1}^{n} u_1 = U_{tot}$$

$$m\sum_{i=1}^{n} u_i + nU_{lb} - mnu_1 = U_{tot}$$

$$m\left(\sum_{i=1}^{n} u_i - nu_1\right) + nU_{lb} = U_{tot}$$

$$m = \frac{U_{tot} - nU_{lb}}{\sum_{i=1}^{n} u_i - nu_1}$$

Now that m is a known value, it can be used to compute q.

The final step is to use q and m for generating the U_i values.

$$U_i = (u_i + q)\, m \; \forall i = 1..n$$

Chapter 7

Experimental Results

This chapter shows the experiments that were performed to evaluate the extended version of the SCHED_DEADLINE scheduling class implementing the H-CBS-SO features.

Initially, the parameters used to generate the taskset are described. Then, the results of the experiments are presented.

7.1 Experiment Setup

The experiment aims at verifying the overhead introduced by the new features of SCHED_DEADLINE implementing the H-CBS-SO policy.

The performance tests are obtained by launching a script which sequentially executes the *spawner* tool with different tasksets.

Each taskset has a number N of tasks and the JSON configuration files are generated by the *generator* in such a way that:

- number of tasks: $N = \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$;
- number of jobs: $J = 10'000$;
- CPU total utilization factor: $U = 0.8$;
- minimum job utilization factor: $U_{lb} = \frac{U}{N+1}$;
- minimum job period: $P_{min} = 100000$ ns;
- maximum job period: $P_{max} = 10000000$ ns.

Before and after launching the *spawner*, the script enables the Ftrace function profiling through the *debugfs* filesystem.

The SCHED_DEADLINE functions that were modified are:

- *update_curr_dl()*;
- *enqeueue_task_dl()*;
- *dequeue_task_dl()*;
- *dl_task_timer()*.

The profiling function can be activated like shown in listing 7.1.

Listing 7.1 : Activation of Ftrace Function Profiling

```
echo nop > /sys/kernel/debug/tracing/current_tracer
echo 0 > /sys/kernel/debug/tracing/options/sleep-time

echo dl_task_timer > /sys/kernel/debug/tracing/set_ftrace_filter
echo update_curr_dl >> /sys/kernel/debug/tracing/set_ftrace_filter
echo enqueue_task_dl >> /sys/kernel/debug/tracing/set_ftrace_filter
echo dequeue_task_dl >> /sys/kernel/debug/tracing/set_ftrace_filter
echo 0 > /sys/kernel/debug/tracing/function_profile_enabled
echo 1 > /sys/kernel/debug/tracing/function_profile_enabled
```

At the end of every *spawner* execution, the obtained results are stored in a separate folder. Those results can be later manipulated and analyzed by other tools to obtain more precise statistical information.

7.2 Overhead Results

The new features introduced for managing the H-CBS-SO policy depend on the SS_QUEUE that stores the suspended tasks and has been implemented through a red-black tree. Thus, the tests to evaluate the overhead introduced with the new feature are performed using the number of tasks as reference parameter.

For each function described in the first part of this chapter has been measured the execution time, which varies as a function of the number of tasks.

The computational cost of the new functions is comparable to the original SCHED_DEADLINE implementation, as shown in the following figures.

As a result, the additional SCHED_DEADLINE features for implementing the H-CBS-SO algorithm do not cause an worsening of the system load.

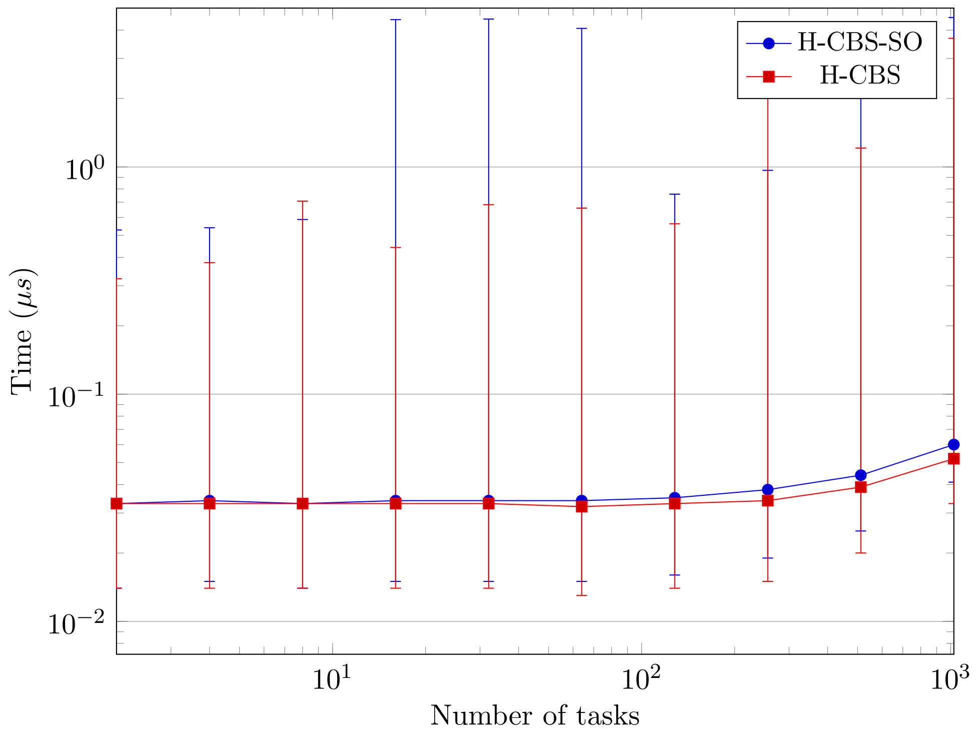


Figure 7.1: *update_curr_dl()* average execution times.

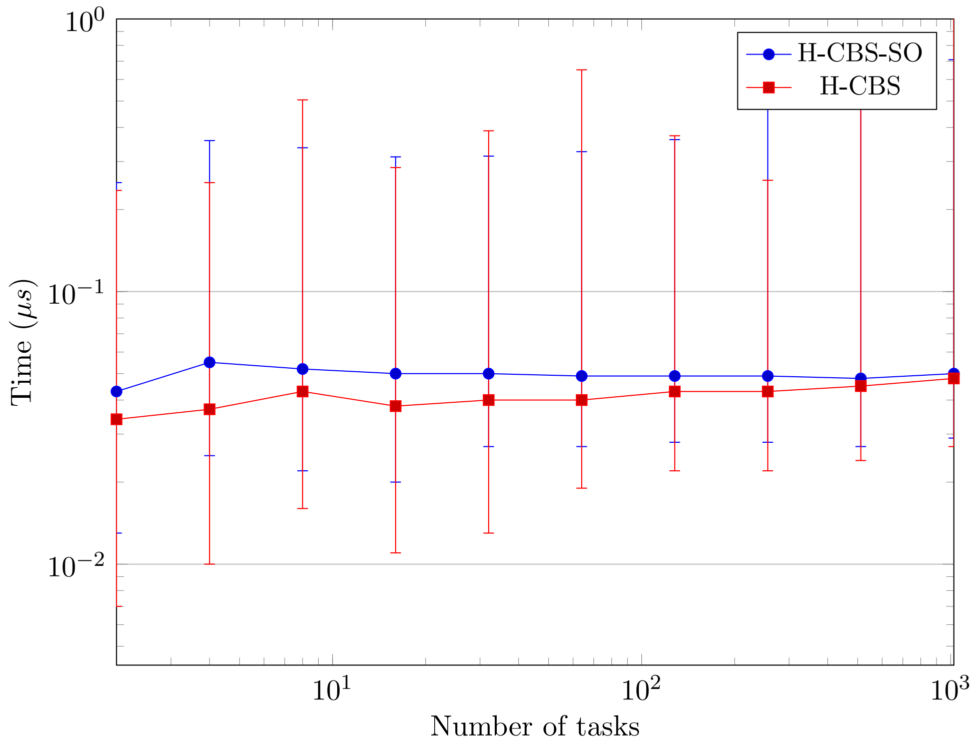


Figure 7.2: *enqueue_task_dl()* average execution times.

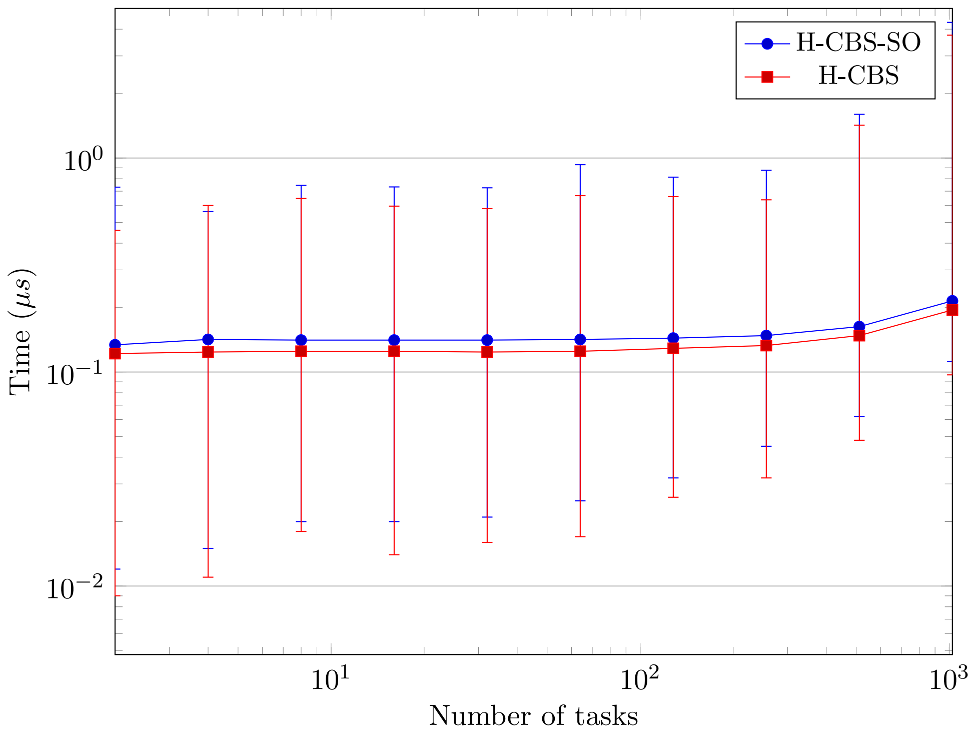


Figure 7.3: *dequeue_task_dl()* average execution times.

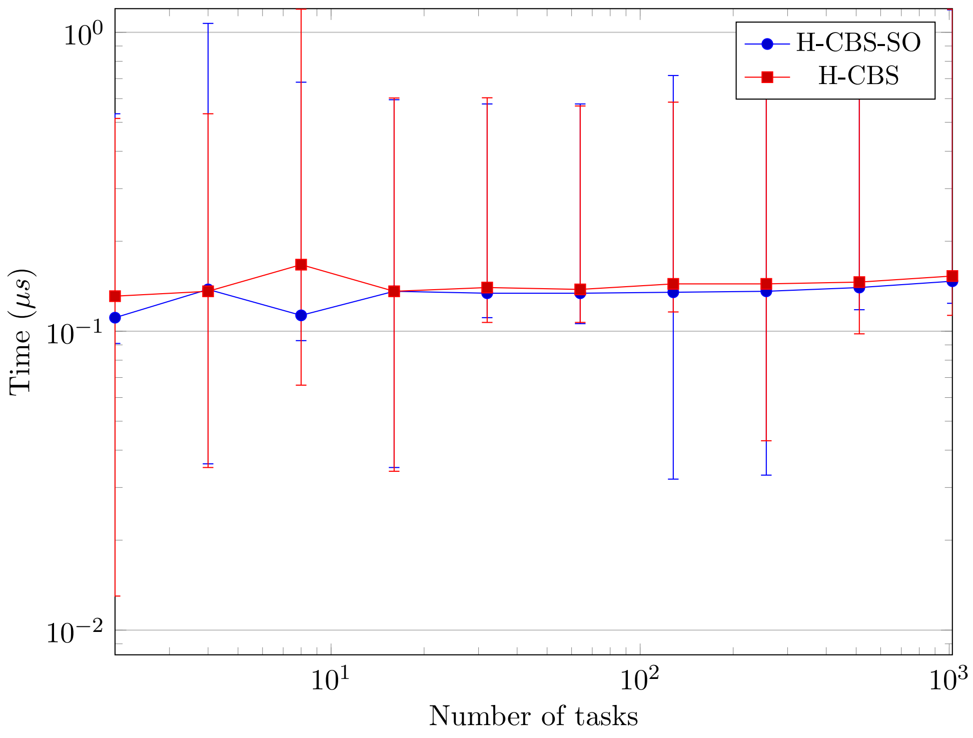


Figure 7.4: *dl_task_timer()* average execution times.

Appendix A

Code

This chapter shows the code modifications made on the Linux kernel in order to:

- impelment the H-CBS-SO features described in chapter 4;
- add the new tracing events shown in subsection 6.1.1;
- get more detailed information of the function performance with the Ftrace functionalities presented in subsection 6.1.2.

A.1 include/linux/sched.h

A.1.1 sched_dl_entity

Listing A.1 : sched_dl_entity

```
struct sched_dl_entity {
        struct rb_node  rb_node;
        struct rb_node  rb_ss_queue_node;

        struct ss_queue *in_ss_queue;

        /*
         * Original scheduling parameters. Copied here from sched_attr
         * during sched_setattr(), they will remain the same until
         * the next sched_setattr().
         */
        u64 dl_runtime;         /* maximum runtime for each instance    */
        u64 dl_deadline;        /* relative deadline of each instance   */
        u64 dl_period;          /* separation of two instances (period) */
        u64 dl_bw;              /* dl_runtime / dl_deadline             */

        /*
         * Actual scheduling parameters. Initialized with the values above,
         * they are continously updated during task execution. Note that
         * the remaining runtime could be < 0 in case we are in overrun.
         */
        s64 runtime;            /* remaining runtime for this instance  */
        u64 deadline;           /* absolute deadline for this instance  */
        unsigned int flags;     /* specifying the scheduler behaviour   */

        /*
         * Some bool flags:
         *
         * @dl_throttled tells if we exhausted the runtime. If so, the
         * task has to wait for a replenishment to be performed at the
         * next firing of dl_timer.
         *
         * @dl_new tells if a new instance arrived. If so we must
         * start executing it with full runtime and reset its absolute
         * deadline;
         *
         * @dl_boosted tells if we are boosted due to DI. If so we are
         * outside bandwidth enforcement mechanism (but only until we
         * exit the critical section);
         *
         * @dl_yielded tells if task gave up the cpu before consuming
         * all its available runtime during the last job.
         */
        int dl_throttled, dl_new, dl_boosted, dl_yielded, dl_blocked;

        /*
         * Bandwidth enforcement timer. Each -deadline task has its
         * own bandwidth to be enforced, thus we need one timer per task.
         */
        struct hrtimer dl_timer;
};
```

A.2 kernel/sched/core.c

A.2.1 SS_QUEUE Allocation Macro

Listing A.2 : Macro for Allocating SS_QUEUE

```
DEFINE_PER_CPU_SHARED_ALIGNED(struct ss_queue, ssqueues);
```

A.2.2 Setting Default Values to sched_dl_entity

Listing A.3 : dl_clear_params()

```
/*
 * This function clears the sched_dl_entity static params.
 */
void __dl_clear_params(struct task_struct *p)
{
        struct sched_dl_entity *dl_se = &p->dl;

        dl_se->dl_runtime = 0;
        dl_se->dl_deadline = 0;
        dl_se->dl_period = 0;
        dl_se->flags = 0;
        dl_se->dl_bw = 0;
        dl_se->in_ss_queue = 0;

        dl_se->dl_blocked = 0;
        dl_se->dl_throttled = 0;
        dl_se->dl_new = 1;
        dl_se->dl_yielded = 0;
}
```

A.2.3 SS_QUEUE Initialization

Listing A.4 : sched_init()

```
void __init sched_init(void)
{
        ...

        for_each_possible_cpu(i) {
                struct rq *rq;
                struct ss_queue *ss_queue;

                ss_queue = cpu_ss_queue(i);
                rq = cpu_rq(i);
                raw_spin_lock_init(&rq->lock);
                rq->nr_running = 0;
                rq->calc_load_active = 0;
                rq->calc_load_update = jiffies + LOAD_FREQ;
                init_cfs_rq(&rq->cfs);
                init_rt_rq(&rq->rt, rq);
                init_dl_rq(&rq->dl, rq);
                init_dl_ss_queue(ss_queue);

        ...
}
```

A.3 kernel/sched/deadline.c

A.3.1 Insertion and Removal of SS_QUEUE Elements

Listing A.5 : SS_QUEUE Elements Insertion and Removal

```
/*
 * Inserts a new element in the SS_QUEUE
 */
static int dl_ss_queue_insert(struct ss_queue *ss_queue,
                              struct sched_dl_entity *data)
{
        struct rb_node **new = &(ss_queue->rb_tree.rb_node), *parent = NULL;
        struct sched_dl_entity *this;
        int leftmost = 1;

        while (*new) {
                parent = *new;
                this = container_of(parent, struct sched_dl_entity,
                                    rb_ss_queue_node);

                if (data == this) {
                        // The process is already in the list, so
                        // no insertion is required
                        return -1;
                }

                if (data->deadline < this->deadline) {
                        new = &((*new)->rb_left);
                } else {
                        new = &((*new)->rb_right);
                        leftmost = 0;
                }
        }

        trace_sched_dl_ss_queue_new(data);

        if (leftmost)
                ss_queue->rb_leftmost = &data->rb_ss_queue_node;

        // Add new node and rebalance tree.
        rb_link_node(&data->rb_ss_queue_node, parent, new);
        rb_insert_color(&data->rb_ss_queue_node, &ss_queue->rb_tree);

        return 0;
}

/*
 * Removes the selected element from the SS_QUEUE
 */
static void dl_ss_queue_remove(struct ss_queue *ss_queue,
                               struct sched_dl_entity *data)
{
        if (RB_EMPTY_NODE(&data->rb_ss_queue_node))
                return;

        trace_sched_dl_ss_queue_delete(data);
        if (data->in_ss_queue->rb_leftmost == &data->rb_ss_queue_node)
                data->in_ss_queue->rb_leftmost = rb_next(&data->rb_ss_queue_node);

        rb_erase(&data->rb_ss_queue_node, &data->in_ss_queue->rb_tree);
        RB_CLEAR_NODE(&data->rb_ss_queue_node);
}
```

A.3.2 Getting SS_QUEUE Associated to a sched_dl_entity

Listing A.6 : ss_queue_of_se()

```
static inline struct ss_queue *ss_queue_of_se(struct sched_dl_entity *dl_se)
{
        return task_ss_queue(dl_task_of(dl_se));
}
```

A.3.3 SS_QUEUE Red-Black Tree Initialization

Listing A.7 : init_dl_ss_queue()

```
void init_dl_ss_queue(struct ss_queue *ss_queue)
{
        ss_queue->rb_tree = RB_ROOT;
}
```

A.3.4 Budget Replenishment

Listing A.8 : dl_task_timer()

```
static enum hrtimer_restart dl_task_timer(struct hrtimer *timer)
{
        struct sched_dl_entity *dl_se = container_of(timer,
                                                     struct sched_dl_entity,
                                                     dl_timer);
        struct task_struct *p = dl_task_of(dl_se);
        struct rq *rq;

        // D-transition
        if (dl_se->in_ss_queue) {
                if (dl_se->dl_blocked) {
                        trace_sched_dl_ss_from_susp(dl_se);
                        dl_ss_queue_insert(dl_se->in_ss_queue, dl_se);

                        return HRTIMER_NORESTART;
                }
                dl_se->in_ss_queue = 0;
        }

        ...
}
```

A.3.5 Managing Budget from SS_QUEUE Head

Listing A.9 : update_ss_queue()

```
static void update_ss_queue(struct rq *rq, struct sched_dl_entity *dl_se, u64 delt
{
        struct sched_dl_entity *ss_queue_head;

        if (this_ss_queue()->rb_leftmost) {

                ss_queue_head = container_of(this_ss_queue()->rb_leftmost,
                                        struct sched_dl_entity,
                                rb_ss_queue_node);

                // The budget is consumed only if the running task has lower prior
                // than the head of the SS_QUEUE

                if (dl_se->deadline >= ss_queue_head->deadline) {
                        ss_queue_head->runtime -= delta_exec;

                        if (dl_runtime_exceeded(rq, ss_queue_head)) {
                                trace_sched_dl_ss_to_susp(ss_queue_head);

                                dl_ss_queue_remove(ss_queue_head->in_ss_queue, ss_

                                if (unlikely(!start_dl_timer(ss_queue_head, ss_queu
                                        dl_ss_queue_insert(ss_queue_head->in_ss_que
                        }
                }
        }
}
```

A.3.6 Decrementing Budget

Listing A.10 : update_curr_dl()

```
/*
 * Update the current task's runtime statistics (provided it is still
 * a -deadline task and has not been removed from the dl_rq).
 */
static void update_curr_dl(struct rq *rq)
{
        struct task_struct *curr = rq->curr;
        struct sched_dl_entity *dl_se = &curr->dl;
        u64 delta_exec;

        if (!dl_task(curr) || !on_dl_rq(dl_se))
                return;

        /*
         * Consumed budget is computed considering the time as
         * observed by schedulable tasks (excluding time spent
         * in hardirq context, etc.). Deadlines are instead
         * computed using hard walltime. This seems to be the more
         * natural solution, but the full ramifications of this
         * approach need further study.
         */
        delta_exec = rq_clock_task(rq) - curr->se.exec_start;
        if (unlikely((s64)delta_exec <= 0))
                return;

        schedstat_set(curr->se.statistics.exec_max,
                      max(curr->se.statistics.exec_max, delta_exec));

        curr->se.sum_exec_runtime += delta_exec;
        account_group_exec_runtime(curr, delta_exec);

        curr->se.exec_start = rq_clock_task(rq);
        cpuacct_charge(curr, delta_exec);

        sched_rt_avg_update(rq, delta_exec);

        // Consume budget also from the head of SS_QUEUE
        update_ss_queue(rq, dl_se, delta_exec);

        ...
}
```

A.3.7 Enqueuing SCHED_DEADLINE Tasks

Listing A.11 : enqueue_task_dl()

```
static void enqueue_task_dl(struct rq *rq,
                                struct task_struct *p,
                                int flags)
{
        struct task_struct *pi_task = rt_mutex_get_top_task(p);
        struct sched_dl_entity *pi_se = &p->dl;

        /*
         * Use the scheduling parameters of the top pi-waiter
         * task if we have one and its (relative) deadline is
         * smaller than our one... OTW we keep our runtime and
         * deadline.
         */
        if (pi_task && p->dl.dl_boosted &&
                dl_prio(pi_task->normal_prio)) {
                pi_se = &pi_task->dl;
        } else if (!dl_prio(p->normal_prio)) {
                /*
                 * Special case in which we have a !SCHED_DEADLINE task
                 * that is going to be deboosted, but exceedes its
                 * runtime while doing so. No point in replenishing
                 * it, as it's going to return back to its original
                 * scheduling class after this.
                 */
                BUG_ON(!p->dl.dl_boosted || flags != ENQUEUE_REPLENISH);
                return;
        }

        /*
         * If p is throttled, we do nothing. In fact, if it exhausted
         * its budget it needs a replenishment and, since it now is on
         * its rq, the bandwidth timer callback (which clearly has not
         * run yet) will take care of this.
         */
        if (p->dl.dl_throttled) {
                if (p->dl.in_ss_queue)
                        p->dl.dl_blocked = 0;
                return;
        }
        if (p->dl.in_ss_queue) {

                trace_sched_dl_from_ss(p);

                // Remove task from SS_QUEUE
                // The task is self suspended, so,
                //place it into the SS_QUEUE

                dl_ss_queue_remove(p->dl.in_ss_queue, &p->dl);
                p->dl.in_ss_queue = 0;
        }

        enqueue_dl_entity(&p->dl, pi_se, flags);

        if (!task_current(rq, p) && p->nr_cpus_allowed > 1)
                enqueue_pushable_dl_task(rq, p);
}
```

A.3.8 Dequeuing SCHED_DEADLINE Tasks

Listing A.12 : dequeue_task_dl()

```
static void dequeue_task_dl(struct rq *rq,
                                struct task_struct *p,
                                int flags)
{
        update_curr_dl(rq);
        __dequeue_task_dl(rq, p, flags);

        if (!dl_runtime_exceeded(rq, &p->dl)) {
                if (flags & 1) {
                        if (!p->dl.in_ss_queue) {

                                trace_sched_dl_to_ss(&p->dl);

                                // The task is self suspended, so,
                                // place it into the SS_QUEUE

                                p->dl.in_ss_queue = this_ss_queue();
                                dl_ss_queue_insert(p->dl.in_ss_queue,
                                &p->dl);
                                p->dl.dl_blocked = 1;
                        }
                }
        }
}
```

A.3.9 Cleaning SS_QUEUE When Tasks Die

Listing A.13 : task_dead_dl()

```
static void task_dead_dl(struct task_struct *p)
{
        struct hrtimer *timer = &p->dl.dl_timer;
        struct dl_bw *dl_b = dl_bw_of(task_cpu(p));

        struct ss_queue * ss_q = p->dl.in_ss_queue;

        if (ss_q) {
          dl_ss_queue_remove(ss_q, &p->dl);
          p->dl.in_ss_queue = 0;
        }

        //dl_ss_queue_print_ordered(ss_q);

        /*
         * Since we are TASK_DEAD we won't slip out of the domain!
         */
        raw_spin_lock_irq(&dl_b->lock);
        /* XXX we should retain the bw until 0-lag */
        dl_b->total_bw -= p->dl.dl_bw;
        raw_spin_unlock_irq(&dl_b->lock);

        hrtimer_cancel(timer);
}
```

A.4 trace/events/sched.h

A.4.1 Macros for Generating new Ftrace Events

Listing A.14 : Event Macros for Ftrace

```

/*
 * Tracepoint for newborn SCHED_DEADLINE task.
 */
TRACE_EVENT(sched_dl_new,
        TP_PROTO(struct sched_dl_entity *dl_se),

        TP_ARGS(dl_se),

        TP_STRUCT__entry(
                __field(        u64,    dl_runtime)
                __field(        u64,    dl_deadline)
                __field(        u64,    dl_period)
                __field(        u64,    dl_bw)
                __field(        u64,    runtime)
                __field(        u64,    deadline)
        ),

        TP_fast_assign(
                __entry->dl_runtime     = dl_se->dl_runtime;
                __entry->dl_deadline    = dl_se->dl_deadline;
                __entry->dl_period      = dl_se->dl_period;
                __entry->dl_bw          = dl_se->dl_bw;
                __entry->runtime        = dl_se->runtime;
                __entry->deadline       = dl_se->deadline;
        ),

        TP_printk("deadline=%llu␣period=%llu␣runtime=%llu",
                  __entry->dl_deadline,
           __entry->dl_period,
           __entry->dl_runtime)
);

/*
 * Tracepoint for server budget replenishment in SCHED_DEADLINE.
 */
TRACE_EVENT(sched_dl_fillbudget,

        TP_PROTO(struct sched_dl_entity *dl_se),

        TP_ARGS(dl_se),

        TP_STRUCT__entry(
                __field(        u64,    dl_runtime)
                __field(        u64,    dl_deadline)
                __field(        u64,    dl_period)
                __field(        u64,    dl_bw)
                __field(        u64,    runtime)
                __field(        u64,    deadline)
        ),

        TP_fast_assign(
                __entry->dl_runtime     = dl_se->dl_runtime;
                __entry->dl_deadline    = dl_se->dl_deadline;
                __entry->dl_period      = dl_se->dl_period;
                __entry->dl_bw          = dl_se->dl_bw;
                __entry->runtime        = dl_se->runtime;
                __entry->deadline       = dl_se->deadline;
        ),

        TP_printk("deadline=%llu␣period=%llu␣runtime=%llu",
                  __entry->deadline,
           __entry->dl_period,
           __entry->runtime)
);

TRACE_EVENT(sched_dl_ss_queue_new,

        TP_PROTO(struct sched_dl_entity *dl_se),

        TP_ARGS(dl_se),

        TP_STRUCT__entry(
                __field(        u64,    dl_runtime)
                __field(        u64,    dl_deadline)
                __field(        u64,    dl_period)
                __field(        u64,    dl_bw)
                __field(        u64,    runtime)
                __field(        u64,    deadline)
        ),

        TP_fast_assign(
                __entry->dl_runtime     = dl_se->dl_runtime;
                __entry->dl_deadline    = dl_se->dl_deadline;
                __entry->dl_period      = dl_se->dl_period;
                __entry->dl_bw          = dl_se->dl_bw;
                __entry->runtime        = dl_se->runtime;
                __entry->deadline       = dl_se->deadline;
        ),

        TP_printk("deadline=%llu␣period=%llu␣runtime=%llu",
                  __entry->dl_deadline,
                __entry->dl_period,
                __entry->dl_runtime)
);

TRACE_EVENT(sched_dl_ss_queue_delete,

        TP_PROTO(struct sched_dl_entity *dl_se),

        TP_ARGS(dl_se),

        TP_STRUCT__entry(
                __field(        u64,    dl_runtime)
                __field(        u64,    dl_deadline)
                __field(        u64,    dl_period)
                __field(        u64,    dl_bw)
                __field(        u64,    runtime)
                __field(        u64,    deadline)
        ),

        TP_fast_assign(
                __entry->dl_runtime     = dl_se->dl_runtime;
                __entry->dl_deadline    = dl_se->dl_deadline;
                __entry->dl_period      = dl_se->dl_period;
                __entry->dl_bw          = dl_se->dl_bw;
                __entry->runtime        = dl_se->runtime;
                __entry->deadline       = dl_se->deadline;
        ),

        TP_printk("deadline=%llu␣period=%llu␣runtime=%llu",
                  __entry->dl_deadline,
                __entry->dl_period,
                __entry->dl_runtime)
);

/*
 * Tracepoint for server transition from Ready to Suspended.
 */
TRACE_EVENT(sched_dl_to_susp,

        TP_PROTO(struct sched_dl_entity *dl_se),

        TP_ARGS(dl_se),

        TP_STRUCT__entry(
                __field(        u64,    dl_runtime)
                __field(        u64,    dl_deadline)
                __field(        u64,    dl_period)
                __field(        u64,    dl_bw)
                __field(        u64,    runtime)
                __field(        u64,    deadline)
        ),

        TP_fast_assign(
                __entry->dl_runtime     = dl_se->dl_runtime;
                __entry->dl_deadline    = dl_se->dl_deadline;
                __entry->dl_period      = dl_se->dl_period;
                __entry->dl_bw          = dl_se->dl_bw;
                __entry->runtime        = dl_se->runtime;
                __entry->deadline       = dl_se->deadline;
        ),

        TP_printk("deadline=%llu␣period=%llu␣runtime=%llu",
                  __entry->deadline,
           __entry->dl_period,
           __entry->runtime)
);

/*
 * Tracepoint for server transition from Suspended to Ready.
 */
TRACE_EVENT(sched_dl_from_susp,

        TP_PROTO(struct task_struct *p),

        TP_ARGS(p),

        TP_STRUCT__entry(
                __field(        pid_t,  pid)
                __field(        u64,    deadline)
                __field(        s64,    runtime)
        ),

        TP_fast_assign(
                __entry->pid            = p->pid;
                __entry->deadline       = p->dl.deadline;
                __entry->runtime        = p->dl.runtime;
        ),
        TP_printk("pid=%d␣deadline=%lld␣runtime=%lld",
                  __entry->pid,
           __entry->deadline,
           __entry->runtime)
);

/*
 * Tracepoint for server transition from Self-Suspended to Suspended.
 */
TRACE_EVENT(sched_dl_ss_to_susp,

        TP_PROTO(struct sched_dl_entity *dl_se),

        TP_ARGS(dl_se),

        TP_STRUCT__entry(
                __field(        u64,    dl_runtime)
                __field(        u64,    dl_deadline)
                __field(        u64,    dl_period)
                __field(        u64,    dl_bw)
                __field(        u64,    runtime)
                __field(        u64,    deadline)
        ),

        TP_fast_assign(
                __entry->dl_runtime     = dl_se->dl_runtime;
                __entry->dl_deadline    = dl_se->dl_deadline;
                __entry->dl_period      = dl_se->dl_period;
                __entry->dl_bw          = dl_se->dl_bw;
                __entry->runtime        = dl_se->runtime;
                __entry->deadline       = dl_se->deadline;
        ),

        TP_printk("deadline=%llu␣period=%llu␣runtime=%llu",
                  __entry->dl_deadline,
           __entry->dl_period,
           __entry->dl_runtime)
);

/*
 * Tracepoint for server transition from Suspended to Self-Suspended.
 */
TRACE_EVENT(sched_dl_ss_from_susp,
        TP_PROTO(struct sched_dl_entity *dl_se),

        TP_ARGS(dl_se),

        TP_STRUCT__entry(
                __field(        u64,    dl_runtime)
                __field(        u64,    dl_deadline)
                __field(        u64,    dl_period)
                __field(        u64,    dl_bw)
                __field(        u64,    runtime)
                __field(        u64,    deadline)
        ),

        TP_fast_assign(
                __entry->dl_runtime     = dl_se->dl_runtime;
                __entry->dl_deadline    = dl_se->dl_deadline;
                __entry->dl_period      = dl_se->dl_period;
                __entry->dl_bw          = dl_se->dl_bw;
                __entry->runtime        = dl_se->runtime;
                __entry->deadline       = dl_se->deadline;
        ),

        TP_printk("deadline=%llu␣period=%llu␣runtime=%llu",
                  __entry->dl_deadline,
           __entry->dl_period,
           __entry->dl_runtime)
);

/*
 * Tracepoint for server transition from Self-Suspended to Ready.
 */
TRACE_EVENT(sched_dl_from_ss,

        TP_PROTO(struct task_struct *p),

        TP_ARGS(p),

        TP_STRUCT__entry(
                __field(        pid_t,  pid)
                __field(        u64,    deadline)
                __field(        s64,    runtime)
        ),

        TP_fast_assign(
                __entry->pid            = p->pid;
                __entry->deadline       = p->dl.deadline;
                __entry->runtime        = p->dl.runtime;
        ),

        TP_printk("pid=%d␣deadline=%lld␣runtime=%lld",
                  __entry->pid,
           __entry->deadline,
           __entry->runtime)
);

/*
 * Tracepoint for server transition from Ready to Self-Suspended.
 */
TRACE_EVENT(sched_dl_to_ss,

        TP_PROTO(struct sched_dl_entity *dl_se),

        TP_ARGS(dl_se),

        TP_STRUCT__entry(
                __field(        u64,    dl_runtime)
                __field(        u64,    dl_deadline)
                __field(        u64,    dl_period)
                __field(        u64,    dl_bw)
                __field(        u64,    runtime)
                __field(        u64,    deadline)
        ),

        TP_fast_assign(
                __entry->dl_runtime     = dl_se->dl_runtime;
                __entry->dl_deadline    = dl_se->dl_deadline;
                __entry->dl_period      = dl_se->dl_period;
                __entry->dl_bw          = dl_se->dl_bw;
                __entry->runtime        = dl_se->runtime;
                __entry->deadline       = dl_se->deadline;
        ),

        TP_printk("deadline=%llu␣period=%llu␣runtime=%llu",
                  __entry->dl_deadline,
           __entry->dl_period,
           __entry->dl_runtime)
);
```

A.4.2 Macros for Accessing SS_QUEUE

Listing A.15 : Macros that Allows to Easily get SS_QUEUEs pointers

```
DECLARE_PER_CPU_SHARED_ALIGNED(struct ss_queue, ssqueues);

#define cpu_ss_queue(cpu)        (&per_cpu(ssqueues, (cpu)))
#define this_ss_queue()          this_cpu_ptr(&ssqueues)
#define task_ss_queue(p)         cpu_ss_queue(task_cpu(p))
#define raw_ss_queue()           raw_cpu_ptr(&ssqueues)
```

A.5 kernel/sched/sched.h

A.5.1 ss_queue

Listing A.16 : ss_queue

```
struct ss_queue {
        struct rb_root rb_tree;
        struct rb_node *rb_leftmost;
};
```

A.6 kernel/trace/ftrace.c

A.6.1 ftrace_profile

Listing A.17 : ftrace_profile

```
struct ftrace_profile {
        struct hlist_node                node;
        unsigned long                    ip;
        unsigned long                    counter;
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
        char                             not_first_time;
        unsigned long long               time;
        unsigned long long               time_min;
        unsigned long long               time_max;
        unsigned long long               time_squared;
#endif
};
```

A.6.2 Showing Ftrace Statistics

Listing A.18 : function_stat_show()

```
static int function_stat_show(struct seq_file *m, void *v)
{
        struct ftrace_profile *rec = v;
        char str[KSYM_SYMBOL_LEN];
        int ret = 0;
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
        static struct trace_seq s;
        unsigned long long avg;
        unsigned long long stddev;
#endif
        mutex_lock(&ftrace_profile_lock);

        /* we raced with function_profile_reset() */
        if (unlikely(rec->counter == 0)) {
                ret = -EBUSY;
                goto out;
        }

        kallsyms_lookup(rec->ip, NULL, NULL, NULL, str);
        seq_printf(m, "␣␣%-30.30s␣␣%10lu", str, rec->counter);

#ifdef CONFIG_FUNCTION_GRAPH_TRACER
        seq_puts(m, "␣␣␣␣");
        avg = rec->time;
        do_div(avg, rec->counter);

        /* Sample standard deviation (s^2) */
        if (rec->counter <= 1)
                stddev = 0;
        else {
                /*
                 * Apply Welford's method:
                 * s^2 = 1 / (n * (n-1)) * (n * \Sum (x_i)^2 - (\Sum x_i)^2)
                 */
                stddev = rec->counter * rec->time_squared -
                         rec->time * rec->time;

                /*
                 * Divide only 1000 for ns^2 -> us^2 conversion.
                 * trace_print_graph_duration will divide 1000 again.
                 */
                do_div(stddev, rec->counter * (rec->counter - 1) * 1000);
        }

        trace_seq_init(&s);
        trace_print_graph_duration(rec->time, &s);
        trace_seq_puts(&s, "␣␣␣␣");
        trace_print_graph_duration(avg, &s);
        trace_seq_puts(&s, "␣␣␣␣");
        trace_print_graph_duration(stddev, &s);
        trace_seq_puts(&s, "␣␣␣␣");
        trace_print_graph_duration(rec->time_min, &s);
        trace_seq_puts(&s, "␣␣␣␣");
        trace_print_graph_duration(rec->time_max, &s);
        trace_print_seq(m, &s);
#endif
        seq_putc(m, '\n');
out:
        mutex_unlock(&ftrace_profile_lock);

        return ret;
}
```

A.6.3 Updating Ftrace Statistics

Listing A.19 : profile_graph_return()

```
static void profile_graph_return(struct ftrace_graph_ret *trace)
{
        struct ftrace_profile_stat *stat;
        unsigned long long calltime;
        struct ftrace_profile *rec;
        unsigned long flags;

        local_irq_save(flags);
        stat = this_cpu_ptr(&ftrace_profile_stats);
        if (!stat->hash || !ftrace_profile_enabled)
                goto out;

        /* If the calltime was zero'd ignore it */
        if (!trace->calltime)
                goto out;

        calltime = trace->rettime - trace->calltime;

        if (!(trace_flags & TRACE_ITER_GRAPH_TIME)) {
```

```
		int index;

		index = trace->depth;

		/* Append this call time to the parent time to subtract */
		if (index)
			current->ret_stack[index - 1].subtime += calltime;

		if (current->ret_stack[index].subtime < calltime)
			calltime -= current->ret_stack[index].subtime;
		else
			calltime = 0;
	}

	rec = ftrace_find_profiled_func(stat, trace->func);
	if (rec) {
		rec->time += calltime;
		rec->time_squared += calltime * calltime;
		if (unlikely(!rec->not_first_time)) {
			rec->not_first_time = 1;

			rec->time_max = calltime;
			rec->time_min = calltime;
		}
		if (rec->time_min > calltime)
			rec->time_min = calltime;
		if (rec->time_max < calltime)
			rec->time_max = calltime;
	}

 out:
	local_irq_restore(flags);
}
```

Bibliography

[1] M. Bertogna. A. Biondi, A. Melani. Hard Constant Bandwidth Server: Comprehensive Formulation and Critical Scenarios. *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, June 18-20, 2014.

[2] M. Marinoni A. Biondi, A. Parri. Resource Reservation for Real-Time Self-Suspending Tasks. 2015.

[3] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* 2011.

[4] M. L. Dertouzos. Control robotics: the procedural control of physical processes. *Information Processing, 74*, 1974.

[5] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 1974.

[6] G. Buttazzo L. Abeni. Integrating multimedia applications in hard real-time systems. *Proceeding of the IEEE Real-Tiime Systems Symposium*, Madrid, Spain, December 2-4 1998.

[7] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM, 20(1):4661*, January 1973.

[8] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A new reclaiming algorithm for server-based real-time systems. *In Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada, May 25-28*, 2004.

[9] J. Stankovic and K. Ramamritham. Tutorial on Hard Real-Time Systems. *IEEE Computer Society Press*, 1988.