



UNIVERSITÀ DEGLI STUDI
DI TRENTO
Dipartimento di Ingegneria
e Scienza dell'Informazione



Scuola Superiore
Sant'Anna
di Studi Universitari e di Perfezionamento

GRADUATE PROGRAM IN COMPUTER SCIENCE AND
ENGINEERING

Master Thesis

**Design and Development of
Real-Time Multi-Processor
Bandwidth Control
Mechanisms in General
Purpose Operating Systems**

Supervisors:

Prof. Giuseppe Lipari

Scuola Superiore Sant'Anna

Prof. Luigi Palopoli

University of Trento

Candidate:

Sun Youcheng

Academic year 2011/2012

*Alla mia famiglia
e a Liang.*

Abstract

Attentions have been being paid to extend general purpose operating systems (GPOS) with real time functionalities. In this thesis, the Open-Extension Container(OXC) scheduling framework, which can control Central Processing Unit (CPU) bandwidth distribution on multi-processor platforms in a real-time way, is proposed in Linux. The ox container is a new data structure defined in our work. It has a good feature that from a Linux scheduler's aspect, it behaves as a virtual CPU. Under the framework, CPU bandwidths from different processors are reserved according to Constant Bandwidth Server (CBS) rules. Yet as for how to utilize the reserved bandwidths to schedule tasks, this is not the interest of the framework. They can be scheduled by policies used in the Linux system or scheduling algorithms that are implemented for a specific purpose. A subset of these tasks can get a portion of the reservation using the same rule. Furthermore, under the framework, scheduling policies can be applied in a controlled scale instead of the whole system.

In current implementation, both normal tasks and real-time tasks in Linux can work under the framework. Experiments show that our hierarchical CPU bandwidth control is feasible in Linux.

Contents

Introduction	x
0.1 CPU reservations	xi
0.2 Multi-core systems	xii
0.3 The Linux scheduler	xii
0.4 Contributions of this thesis	xii
0.5 State of the Art	xiv
0.6 Organization of the thesis	xv
1 Background	1
1.1 The Constant Bandwidth Server theory	1
1.2 The Linux Scheduler	3
1.2.1 Scheduling classes	3
1.2.2 Runqueue centered scheduling	6
1.2.3 Completely Fair scheduler	8
1.2.4 Real time scheduler	11
1.3 Related work	12
1.3.1 RT throttling	12
1.3.2 CFS bandwidth control	13
1.3.3 AQuoSA	14
1.3.4 Sched_deadline patch	14
1.3.5 IRMOS real-time framework	15
2 Design of OXC Scheduling	16
2.1 Open-Extension Container Structure	16
2.2 The oxc scheduling	17

2.2.1	Scheduling routes in oxc enabled Linux	17
2.2.2	Features of the oxc scheduling	19
2.3	Movtivation for oxc scheduling framework	20
3	Development of OXC Framework	21
3.1	Implementation of ox container structure	21
3.2	Extensions on original data structures	24
3.3	To direct a task to a per ox container runqueue	25
3.3.1	To build the scheduling route in mainline Linux	26
3.3.2	To build the scheduling route in oxc enabled Linux	26
3.4	Run tasks under OXC scheduling framework	32
3.4.1	To obtain the runqueue of a task	33
3.4.2	To enqueue an oxc task	34
3.4.3	To dequeue an oxc task	36
3.4.4	To check the preemption condition	38
3.4.5	To pick up an oxc task	39
3.4.6	put_prev_task_oxc	43
3.4.7	set_curr_task_oxc	43
3.4.8	task_tick_oxc	45
3.5	SMP support in oxc framework	45
3.6	User interfaces provided by OXC framework	46
3.7	Cooperation with scheduling mechanisms inside Linux	48
4	Experiments	51
4.1	Ftrace in Linux kernel	52
4.2	Tbench	54
4.3	Experiment A	54
4.3.1	The experiment design	54
4.3.2	Experiment results	55
4.4	Experiment B	59
4.4.1	Experiment design	59
4.4.2	Experiment results	60
4.5	Experiment feedbacks	64

<i>CONTENTS</i>	vi
5 Conclusions and Future Work	65
Acknowledgments	68

List of Figures

1.1	Scheduling classes in Linux	5
1.2	CFS scheduling when CFS group scheduling is enabled.	10
1.3	Scheduling scheme without group scheduling	10
1.4	RT scheduling when RT group scheduling is enabled.	12
2.1	Open-Extension Container	17
2.2	Extended scheduling routes with task group scheduling	18
2.3	Extended scheduling route without task group scheduling	19
3.1	The creation of a task group in original Linux	27
3.2	The creation of a task group in oxc enabled Linux	29
4.1	Measured execution time for <code>pick_next_task_oxc</code>	57
4.2	Measured execution time for <code>put_prev_task_oxc</code>	57
4.3	Measured execution time for <code>task_tick_oxc</code>	58
4.4	<i>oxc control</i> vs. <i>RT throttling</i>	62
4.5	<i>oxc control</i> vs. <i>CFS bandwidth control</i>	63
4.6	<i>oxc control + RT throttling</i> vs. <i>oxc control + CFS scheduling</i>	63

Listings

1.1	Scheduling operations for a scheduler	3
1.2	A task embeds scheduling entities	5
1.3	A task group contains both RT and CFS tasks	6
1.4	The runqueue structure	6
1.5	Specific runqueue information within a task group	8
1.6	The CFS runqueue	9
1.7	The CFS scheduling entity	9
1.8	The RT runqueue	11
1.9	The RT scheduling entity	12
3.1	The ox container : struct oxc_rq	21
3.2	The EDF tree	23
3.3	The hyper ox container	23
3.4	Extensions on struct rq	24
3.5	Extensions on struct task_group	24
3.6	is_oxc_task field in struct task_struct	25
3.7	To associate tasks with a per CPU runqueue in mainline Linux	26
3.8	The extended set_task_rq	27
3.9	OXC scheduling related initialization during task group creation	29
3.10	To explicitly direct a task group (CFS part) to an OXC local runqueue	30
3.11	Codes to obtain the runqueue of a task	33
3.12	To enqueue an task into the oxc local runqueue	34
3.13	To update the number of tasks inside a container	35
3.14	To put an ox container in the EDF tree if necessary	35
3.15	To remove a task from the oxc local runqueue	36

3.16	To remove an ox container from the EDF tree if necessary . .	37
3.17	The preemption check for an oxc task	38
3.18	To pick up the most eligible oxc task	40
3.19	To update an ox container's runtime information	41
3.20	To check if an ox container should be throttled	41
3.21	Conclusion work before an oxc task is switched out of a CPU .	43
3.22	Another point to update reservation information	44
3.23	The most frequently called entry to update a container's runtime	45
4.1	The simple body of <code>task_tick_rt</code>	59

Introduction

Linux is the most widely deployed open source GPOS. Linux flavors are used in a very wide range of systems and application area, from servers for cloud computing, to desktop computing, to tablet and smart phones, to industrial embedded systems.

One critical part of every Operating System is the task scheduler. For Linux, the scheduling system is also very critical given the wide range of application scenario in which this system is used. In facts, every scenario has different requirements and there is no single scheduler that can optimally satisfy them all.

Traditionally, one of the aim of Linux scheduling is to distribute CPU cycles fairly among tasks and task groups according to their relative importance. Another important requirement is to make *interactive applications* reactive to user input. There is no precise definition of an *interactive task*: the kernel uses heuristics to identify interactive tasks from batch processing tasks, in order to give higher priority to interactive tasks, thus reducing their response time.

One important class of applications consists of hard real-time or soft real-time systems. In order to work well, these applications have requirements in terms of the CPU cycles they receive, or on the time they need to react to external events. Hard and soft real-time applications needs to be guaranteed prior to their execution that they will be able to meet all their timing requirements (if hard real-time) or most of their timing requirements (if soft real-time). Unfortunately, such timing guarantee (or *real-time guarantee*) cannot be provided just by using an heuristic scheduler.

Examples of soft real-time applications are multi-media applications. Mul-

multimedia finds its application in various areas including DVD player, video conference, virtual reality, computer games, etc. Multimedia activities include video and audio streams, which are the challenge for the scheduler in a GPOS. Because such media streams are characterized by an implicit timing requirement. More specifically, video/audio streams in a computer are digitalized and divided into frames. In order to reproduce the stream, the sequence of frames must be processed respecting some temporal constraints. As an example, a rate of 25 frames per second (fps) is allowed in MPEG-2 video, which means that every 40ms a frame is expected to be loaded, decoded, filtered and played. If a frame is lost or played too late, the user may not notice any degradation. But if the skipped frames or frame delays are over some threshold, users start perceiving a certain degradation on the quality of the service. In Linux, a multimedia application is normally treated as interactive tasks. Even though, the scheduler has no concept of the timing requirement for a stream and there is no way to guarantee some degree of quality for a multimedia application.

0.1 CPU reservations

The temporal constraint of multimedia applications can be solved by a class of *resource reservation* (RR) algorithms in real-time scheduling literature. The feature of a RR algorithm is temporal isolation. That is, a fraction of CPU time can be reserved to a set of tasks. By applying resource reservation in Linux, multimedia applications can meet their timing requirements. Our work exploits the RR mechanism for a more general use in managing CPU bandwidth distribution. The RR algorithm we use is CBS [1]. More on it will be introduced in section 1.1.

In many cases, people prefer to distributing the computing power in a privileged and predictable way. For instance, to give 10% of the total CPU cycles to a set of tasks; furthermore, take half of this 10% and assign it to a subset of the tasks. This example shows a particular scenario of CPU bandwidth management.

0.2 Multi-core systems

Nowadays, multi-core architectures are successfully used to boost computation capability for computing devices. Common computing systems with multiple processors are becoming mainstream: it is no surprise to see an embedded device with more than one CPU inside. So, it is necessary for a computing power distribution scheme to support multi-core platforms. Additionally, high speed processors and more cores rather than solving the CPU timing guarantee problem do bring more challenges for the developer. Also, no matter how powerful the platform is, there is always time when people need more. These are all considerations when designing the CPU bandwidth management work.

0.3 The Linux scheduler

In mainline Linux kernel, there are two so called real-time (RT) scheduling policies: `SCHED_FIFO` and `SCHED_RR`. Tasks scheduled by them are called “real-time (RT) tasks”. They are required in POSIX standard for POSIX-compliant operating systems like Linux. Unfortunately, despite of the name, these two policies can only provide real-time guarantee in very limited conditions. In mainline Linux, there exist two non real-time mechanisms to control CPU bandwidth distribution: real time (RT) throttling and complete fairness scheduler (CFS) bandwidth control. In principle, they are the same technique working for different types of tasks. For more on Linux scheduling, please read chapter 1.

0.4 Contributions of this thesis

This thesis will present the design and development of a hierarchical multi-processor reservation mechanism in Linux that is not constrained with specific scheduling algorithms implemented in the kernel. This work is started from scratch.

Before discussing details on our work, let’s think a little bit what is the

good mechanism to manage the CPU bandwidth in a GPOS, taking Linux as the study case.

- Compatibility

In principle, a scheduler does distribute CPU time. There are several schedulers in Linux and each deals with its own type of tasks. So, the constraint for a scheduler to manage CPU bandwidth distribution is that it can only handle a subset of tasks. A general mechanism that is compatible with different shedulers are better.

- Hierarchical distribution

Tasks in the Linux system are organized in a flat or hierarchy, depending on the kernel compilation option. In modern systems, the hierarchical way is more preferred. The recommended method should be able to hierarchically distribute the CPU cycles.

- Predictability

Because there are various kinds of tasks inside Linux, among which some require (hard or soft) timing guarantee and some do not. The CPU bandwidth management should be predictable. When it comes to predictability, a guarantee test or admission control is always necessary since the predictability can only be available under a condition. For instance, the CPU utilization is no larger than 1.

- Support multi-processor platforms

In this article, the above four items are called *golden rules* for CPU computation power management work in the GPOS. We design and develop a RR based framework that fulfils all golden rules. We call the framework Open-Extension Container (OXC) scheduling framework. The Open-Extension (OX) Container is a new data structure in Linux that is introduced by our work. It is the fundamental element in the framework. More on an ox container is in chapter 2. In oxc scheduling framework, each ox container can reserve an amount of bandwidth from a CPU through CBS rules. Several ox containers can work together and reserve CPU bandwidths from multiple

processors. From the Linux scheduler’s point of view, an ox container is regarded as a virtual CPU. This is why the reserved bandwidth through an ox container can be utilized by different kinds of tasks in Linux. Details on the development of the oxc framework is in chapter 3.

The oxc framework has two features:

- It can predictably distribute the CPU cycles of a multi-processor platform to a set of tasks and its subsets, without requirements for details how they are scheduled.
- Under the framework, scheduling policies can be applied in a fine-grained way.

This first feature is actually a conclusion of four golden rules. The second feature has nothing to do with CPU bandwidth reservation. Usually, when a scheduling algorithm is applied in a system, it is indeed applied in the whole system level. One by-product of our framework is that the working scale of a scheduling algorithm can be constrained in part of the system. The basic unit to apply a scheduling policy under oxc framework is an ox container.

0.5 State of the Art

There is work that extends Linux with real-time capabilities to fulfil the temporal guarantee requirement: RTAI [2], AQuoSA [3], `sched_deadline` patch [4], IRMOS real-time framework [5], RESCH [6], etc. Instead of modifying the system directly, RT-Xen [7] tries to apply real-time mechanisms in the hypervisor level(Xen).

Each work has its emphasis. Initially, our work is motivated by IRMOS. The IRMOS can meet three of our golden rules except for the compatibility one. IRMOS scheduler can only reserve CPU bandwidth for RT tasks in Linux.

RTAI adapts a dual kernel approach. Besides the Linux kernel, there is a real-time kernel for serving real-time tasks. It aims to reduce the worst case latency to run a real-time application. As for the latency, the performance of RTAI is a benchmark for other work.

In AQuoSA, the CBS rules are used to reserve CPU bandwidths. As a CPU time distribution mechanism, it is based on RT scheduling with SCHED_RR policy and fails the multi-processor support rule. However, inside the AQuoSA architecture, there is a feedback-based mechanism to dynamically configure the reservation parameter. Such an adaptive resource allocation is meaningful since workloads in a system will fluctuate with time.

The `sched_deadline` work is very promising to be merged into the mainline kernel as the deadline based scheduler. It should be expected that the `oxc` framework can also reserve bandwidths for tasks scheduled by this deadline based scheduler.

RESCH implements several real-time algorithms in modules that can be loaded into the kernel dynamically. This reminds us if we can realize the `oxc` framework as modules.

RT-Xen tries to bridge the gap between real-time scheduling theory and Xen hypervisor and instantiates a suite of real-time scheduling algorithms. To provide real-time guarantee in the cross platform hypervisor layer is very attractive. And very likely the real-time support from the operating system is necessary for such a purpose. In this case, a real-time CPU bandwidth control mechanism like our work will be helpful.

Our framework is purely interested in CPU bandwidth distribution. To the author's knowledge, there does not exist other work that can achieve all golden rules.

0.6 Organization of the thesis

Chapter 1 (**Background**) gives a brief overview of the knowledge that is needed to understand the `oxc` framework. At first, the CBS algorithm for resource reservation is explained. Then the architecture of Linux scheduling is described. By architecture it means the emphasis is on how different scheduling components are organized. In the end of this chapter, related work is generally discussed, including the RT throttling and CFS bandwidth control in mainline Linux and a more extensive introduction on some work listed above.

Chapter 2 (**Design**) shows the definition of an ox container and its attributes. The theory of ox container scheduling, which is the soul of our work, is explored in this chapter.

Chapter 3 (**Implementation**) presents details of the oxc framework. It first explains new data structures added in the kernel and extensions on original kernel structures. The next part concerns how to implement the oxc scheduling theory in the kernel. Then, important functions, including how to reserve CPU bandwidths, in the oxc framework are studied. A complete introduction on interfaces of the framework and how to use them is also in this chapter. At last, the relationship between oxc framework and Linux CPU bandwidth control methods is investigated.

In Chapter 4 (**Experiment**), two experiments are conducted to evaluate the overhead and performance of the oxc framework (or oxc control). The experiment set-up, kernel recording tool, benchmark tool, etc, are introduced first. Then at each experiment, there is the design, data collection and result analysis.

Finally, the Chapter 5 (**Conclusion**) sums up the current status of the oxc framework and discusses future development of the work.

Chapter 1

Background

1.1 The Constant Bandwidth Server theory

The original CBS algorithm [1] defines rules to reserve bandwidth on a single processor, and every server can manage a single task. The CBS algorithm was later extended to multi-processor global scheduling [8], and to hierarchical scheduling [9, 10, 11] (where multiple tasks can coexist in the same server).

A Constant bandwidth server can be used to provide temporal isolation for a task or a set of tasks. In the terminology of CBS theory, the waking up or creation of a served task is also called the arrival of a request for the server. The following are the classic CBS rules, focusing on single processor.

- A Constant Bandwidth Server (CBS) is characterized by an ordered pair (Q_s, T_s) where Q_s is the maximum budget and T_s is the period of the server. The ratio $U_s = Q_s/T_s$ is denoted as the server bandwidth.
- A CBS server manages two internal variables that define its state: c_s (initialized as C_s) is the current budget at time t and d_s is the current deadline assigned by the server to a request (zero-initialized). Requests served by different servers are scheduled with earliest deadline first. The c_s will decrease with the same value as the time a served task runs.
- A CBS is said to be active at time t if there are pending requests. If a new request arrives while the server is still active, then it is queued in

a server queue or it can preempt the current request and the current request will be queued; this is not part of CBS rules and can be managed with an arbitrary discipline, for example FIFO.

- When a new request arrives at instant t and the server is idle, there are two cases. If $c_s \geq (d_s - t)U_s$, then the server generates a new deadline ($d_s = d_s + T_s$) and c_s is recharged to the maximum value Q_s . Otherwise, the request is served with the current deadline and budget value.
- When a request is completed, the server picks the next (if it exists) pending request from the internal queue and schedule it with the current budget value and deadline.
- When a budget is exhausted ($c_s = 0$), it is recharged at the maximum value ($c_s = Q_s$) and the current deadline is postponed by one period ($d_s = d_s + T_s$).

In our work, we use a hard version of CBS (HCBS). The HCBS rules differ in the situation when a server's budget is exhausted, where the budget is not recharged immediately as the soft resource reservation:

- When a budget is exhausted ($c_s = 0$), the server and the request it serves are suspended until the current deadline (d_s), when the budget is recharged to maximum value ($c_s = Q_s$) and the deadline is updated by one period ($d_s = d_s + T_s$).
- When a request arrives and the server is suspended, the request is put in the server's internal queue.

By this hard variant of CBS, a server reserves exactly Q_s units of time every T_s time units for its requests given the condition that the sum of all servers' utilization on a CPU is not greater than 1.

A complete description of the CBS and of all its variant, using a state machine formalism is available in [12].

1.2 The Linux Scheduler

A scheduler is responsible for distributing CPU cycles to tasks in the system according to some scheduling algorithm. In Linux, tasks refer to a process or a thread and correspond to the data structure `struct task_struct`. The emphasis in this section is to clarify the relationships and connections among different scheduling components. To understand the Linux scheduling architecture is the first step to explore the oxc framework. For details about how linux schedulers work, people can read corresponding chapters in [13] [14].

1.2.1 Scheduling classes

Linux scheduling system adapts a modular design, and the basic building block is a scheduling class, which is an instance of `struct sched_class`¹. Scheduling algorithms are implemented as scheduling classes and a scheduling class is a scheduler component (or simply called a scheduler). The set of all schedulers compose the generic scheduler in Linux. The `struct sched_class` defines a set of interfaces that a scheduler module must implement. Each scheduler fulfils details behind the interface and carries out its specific scheduling behaviour.

There are three scheduling classes in mainline Linux kernel: `rt_sched_class`, `cfs_sched_class` and `idle_sched_class`, defined in `rt.c`, `fair.c`, and `idle.c`, respectively, in directory `kernel/sched`. Each scheduling class is responsible for scheduling a type of tasks. Tasks scheduled with `cfs_sched_class` are called *normal tasks*, and tasks scheduled by `rt_sched_class` are called *rt tasks*. `idle_sched_class` deals with special idle tasks which do nothing and occupy the CPU when no RT or normal tasks need a CPU.

Now, it's time to see the semantics of scheduling operations for a scheduler. The following listing contains main fields of the data structure `struct sched_class`.

Listing 1.1: Scheduling operations for a scheduler

¹Defined in `include/linux/sched.h`

```

struct sched_class {
    const struct sched_class *next;
    void (*enqueue_task) (struct rq *rq, struct task_struct *p,
        int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p,
        int flags);
    void (*check_preempt_curr) (struct rq *rq, struct task_struct
        *p, int flags);
    struct task_struct * (*pick_next_task) (struct rq *rq);
    void (*put_prev_task) (struct rq *rq, struct task_struct *p);
    void (*set_curr_task) (struct rq *rq);
    void (*task_tick) (struct rq *rq, struct task_struct *p, int
        queued);
    ...
};

```

- **next**: Scheduling classes are linked in a chain, as shown in 1.1 on the following page. Whenever a task is needed, the scheduler from the beginning to the end of the chain is checked and corresponding scheduling methods are called until a task is found. So, schedulers in front have higher priority to execute their tasks.
- **enqueue_task**: Called when a task enters a runnable state. The task is then enqueued into a runqueue, which is an instance of `struct rq`.
- **dequeue_task**: When a task is no longer runnable, this function is called to move corresponding task from a runqueue.
- **check_preempt_curr**: This function checks if a task that entered the runnable state should preempt the currently running task.
- **pick_next_task**: This function chooses the task to run next. The newly picked up one can be the one currently occupying the CPU; in this case, no context switches are needed.
- **put_prev_task**: This is the last scheduling activity for a task before it gives up the executing opportunity on a CPU. In fact, it can happen

that after this operation, the same task still occupies the CPU, as it is picked up again through `pick_next_task`.

- `set_curr_task`: This function is called when a task changes its scheduling class or changes its task group.
- `task_tick`: This function is the most frequently called scheduling function. It is a good point to update the scheduling information, and it might lead to task switch.

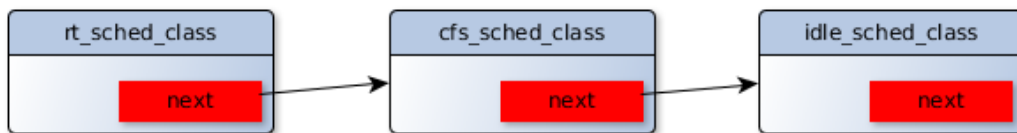


Figure 1.1: Scheduling classes in Linux

The basic scheduling unit in Linux is the *scheduling entity*, which represents both tasks and task groups. There are two kinds of scheduling entities²: CFS (scheduling) entities and RT (scheduling) entities. They are separately defined by `struct sched_entity` and `struct sched_rt_entity`. When we say a task is enqueued in a runqueue, more precisely, it is the task's (CFS or RT) scheduling entity that is enqueued.

Listing 1.2: A task embeds scheduling entities

```

struct task_struct {
    ...
    struct sched_entity se;
    struct sched_rt_entity rt;
    ...
};
  
```

Both CFS and RT entities are embedded in `struct task_struct`. For any task, its status can switch between a CFS task and a RT task through the system call `sched_setscheduler`.

²Both are defined in `include/linux/sched.h`

When `CONFIG_FAIR_GROUP_SCHED` is set, CFS task grouping is enabled. And `CONFIG_RT_GROUP_SCHED` is the kernel configuration for RT task group scheduling. A task group can contains both RT tasks and normal tasks, as shown in listing 1.3. For each CPU, a task group uses a RT entity and a CFS entity to represent its RT tasks and normal tasks on a CPU. Each type of tasks inside a task group is scheduled independently by its own scheduling class.

Listing 1.3: A task group contains both RT and CFS tasks

```
struct task_group {
#ifdef CONFIG_FAIR_GROUP_SCHED
    /* sched_entity of this group on each cpu */
    struct sched_entity **se;
    ...
#endif
#ifdef CONFIG_RT_GROUP_SCHED
    /* sched_rt_entity of this group on each cpu */
    struct sched_entity **rt_se;
    ...
#endif
    ...
};
```

1.2.2 Runqueue centered scheduling

Every hook in `struct sched_class` deals with the data structure `struct rq`³, which is called runqueue in Linux. We say that Linux scheduling is runqueue centered. In Linux, the `struct rq` is a per CPU data structure; each cpu is associated with a dedicated runqueue. Despite its name, `struct rq` is not a queue. The `struct rq` contains a large amount of information. Its partial contents that are necessary for understanding the remaining of the thesis are listed below.

Listing 1.4: The runqueue structure

³Defined in `kernel/sched/sched.h`

```

struct rq {
    ...
    unsigned long nr_running;
    struct cfs_rq cfs;
    struct rt_rq rt;
    struct task_struct *curr, *idle;
    u64 clock;
    u64 clock_task;
#ifdef CONFIG_SMP
    int cpu;
#endif
    ...
};

```

- `nr_running` specifies the number of runnable tasks having been enqueued in the runqueue.
- `cfs` and `rt` are two specific runqueues for `cfs_sched_class` and `rt_sched_class` respectively. In order to handle specific type of tasks, different schedulers define new type of runqueue data structures. When we say a task is enqueued into a runqueue, it is finally into its corresponding specific runqueue. Each task group has one CFS runqueue and RT runqueue per CPU to enqueue the tasks and sub task groups it contains in that CPU. Listing 1.5 on the following page adds this new information to the knowledge just introduced for a task group, as in listing 1.3 on the previous page. The default task group in the system points their CFS runqueue and RT runqueue to fields contained in the per CPU runqueue directly.
- `curr` points to the task currently running in this runqueue.
- `idle` points to a special idle task. This is the task occupying the CPU when no other tasks are runnable.
- `clock` and `clock_task` are time information kept by the runqueue. They updated by `update_rq_clock` method and some scheduling operation can rely them as time source.

- `cpu` tells the CPU of this runqueue is devoted to.

Listing 1.5: Specific runqueue information within a task group

```

struct task_group {
#ifdef CONFIG_FAIR_GROUP_SCHED
    /* sched_entity of this group on each cpu */
    struct sched_entity **se;
    /* runqueue "owned" by this group on each cpu */
    struct cfs_rq **cfs_rq;
    ...
#endif
#ifdef CONFIG_RT_GROUP_SCHED
    /* sched_rt_entity of this group on each cpu */
    struct sched_entity **rt_se;
    struct rt_rq **rt_rq;
    ...
#endif
    ...
};

```

1.2.3 Completely Fair scheduler

The Completely Fair Scheduler (CFS) is implemented in `fair_sched_class`. Most tasks inside Linux are scheduled by completely fair scheduling class and are normal tasks, which can be further divided into three sub types given scheduling policies (`SCHED_NORMAL`, `SCHED_BATCH` and `SCHED_IDLE`⁴).

CFS tries to distribute CPU cycles fairly to tasks and task groups according to their *weight*. A specific runqueue structure `struct cfs_rq` is provided to deal with normal tasks. Recall that an instance of such cfs runqueue is embedded in the per-CPU runqueue and each task group holds a pointer to cfs runqueue on each CPU to store CFS tasks belonging to it. A little more details on CFS runqueue and CFS scheduling entity follow.

⁴This `SCHED_IDLE` policy is not related to `idle_sched_class` which aims to handle a special idle task.

Listing 1.6: The CFS runqueue

```

struct cfs_rq {
    unsigned long nr_running;
    struct rb_root tasks_timeline;
#ifdef CONFIG_FAIR_GROUP_SCHED
    struct rq *rq;
    struct task_group *tg;
#endif
    ...
};

```

- `nr_running` is the number of CFS tasks(entities) in this CFS runqueue.
- `tasks_timeline` is the root of the red-black tree [15] where all CFS entities enqueued into this CFS runqueue are stored. This article will not go into details of the red-black tree mechanism, people only need to know it is an efficient way to sort and access data elements.
- `rq` is the per CPU runqueue that the task group `tg` is finally enqueued.
- `tg` is the task group that owns this CFS runqueue.

Listing 1.7: The CFS scheduling entity

```

struct sched_entity {
    ...
    struct cfs_rq *cfs_rq;
#ifdef CONFIG_FAIR_GROUP_SCHED
    struct cfs_rq *my_rq;
#endif
    ...
};

```

- `cfs_rq` is where this entity is to be queued.
- `my_rq` is the CFS runqueue owned by this entity(group). Remember that a scheduling entity can also represent a task group.

Now there is enough information to show how different scheduling components (`sched_entity`, `task_struct`, `task_group` and `struct cfs_rq`) are related in completely fair scheduler.

In this case that CFS task group scheduling is enabled. The CFS scheduling scheme is shown in figure1.2. This is not a complete scheme:

1. Under a task group there could be sub groups, which behave as the task in the figure
2. In the system, there is a top group, which includes all tasks in the system by default; tasks in this group are enqueued in the CFS runqueue embedded in the per CPU runqueue directly.

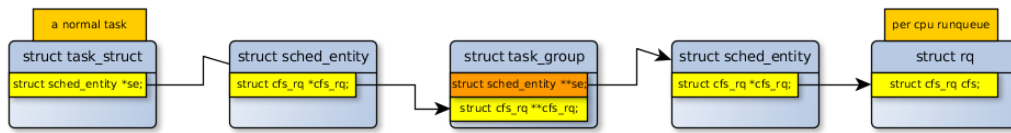


Figure 1.2: CFS scheduling when CFS group scheduling is enabled.

If CFS task group scheduling is not enabled, a task is directed to its per CPU runqueue by a `task_rq` marco. `task_rq` also works for RT scheduling when RT task group scheduling is not enabled. In fact, this `task_rq` can be used even in case RT and CFS task group scheduling are enabled. It just that when such task group scheduling is set, normally the information in each element between a task and the destination runqueue is more important than simply returning a runqueue.

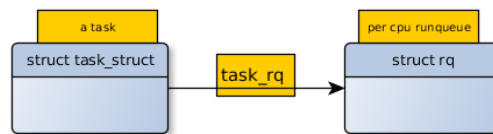


Figure 1.3: Scheduling scheme without group scheduling

We call the scheme in Figure 1.2 and 1.3 *scheduling routes* in CFS scheduling. The route source is a task and the destination is a runqueue. The feature

of a scheduling route is that if one component in the route is known, then other scheduling components in the direction towards the destination can be tracked. The concept of scheduling route is first invented in our work. Later you will see the theory behind oxc framework explores scheduling routes in Linux extensively. We believe a new concept is worthy in formalizing the work for oxc framework.

1.2.4 Real time scheduler

Tasks with POSIX real time policies `SCHED_FIFO` and `SCHED_RR` are scheduled by the real time scheduling class `rt_sched_class` and are called RT tasks. Given figure 1.1 on page 5, RT tasks are always scheduled over normal tasks.

`SCHED_FIFO` implements a simple first-in, first-out scheduling algorithm. A running `SCHED_FIFO` task can only be preempted by a higher priority RT task. `SCHED_RR` is `SCHED_FIFO` with timeslices — it is a round robin algorithm. When a `SCHED_RR` task exhausts its timeslice, another `SCHED_RR` task of the same priority is picked to run a timeslice, and so on. In either case, a RT task cannot be preempted by a lower priority task.

The RT scheduling class provides with a sub runqueue structure `struct rt_rq` to deal with RT tasks.

Listing 1.8: The RT runqueue

```
struct rt_rq {
    struct rt_prio_array active;
    unsigned long rt_nr_running;
#ifdef CONFIG_RT_GROUP_SCHED
    struct rq *rq;
    struct task_group *tg;
#endif
    ...
};

struct rt_prio_array {
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1);
    struct list_head queue[MAX_RT_PRIO];
};
```

All RT tasks with the same priority, let's say *prio*, are kept in a linked list headed by *active.queue[prio]*. If there is a task in the list, the corresponding bit in *active.bitmap* is set. All other fields have the same meaning as in CFS runqueue. Analogous with the CFS scheduling entity, the following `struct sched_rt_entity` is self explanatory enough.

Listing 1.9: The RT scheduling entity

```
struct sched_rt_entity {
    ...
    struct rt_rq *rt_rq;
#ifdef CONFIG_RT_GROUP_SCHED
    struct rt_rq *my_rq;
#endif
    ...
};
```

When `CONFIG_RT_GROUP_SCHED` is set, figure 1.4 shows the scheduling route for RT scheduling. If RT task group scheduling is not enabled, still *task_rq* marco will be used.

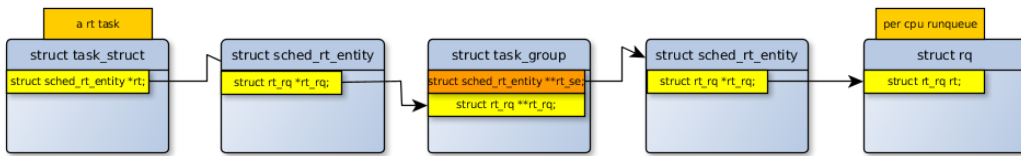


Figure 1.4: RT scheduling when RT group scheduling is enabled.

1.3 Related work

1.3.1 RT throttling

Enabling `CONFIG_RT_GROUP_SCHED` lets users explicitly allocate CPU bandwidth to RT tasks in task groups [16]. It uses the *control group* (cgroup) [17] virtual file system. Each cgroup associates a set of tasks with a set of resources, called *subsystems*. For example *cpuset* subsystem [18] is responsible for assigning a set of CPUs and Memory Nodes to tasks in a cgroup. Such

tasks and resources can be further distributed in sub cgroups. Each cgroup is represented by a directory in the cgroup file system and a hierarchy of cgroups maps to a hierarchy of directories. In the directory, each mounted subsystem provides a list of files that are used as interfaces to control the allocation of a resource. Through mounting the *cpu* subsystem, two interfaces *cpu.rt_period_us* and *cpu.rt_runtime_us* are used to control the CPU bandwidth for RT tasks in each cgroup. That is, the total execution time of RT tasks in a cgroup on each CPU in time length *rt_period_us* cannot exceed *rt_runtime_us*. If this constraint is met, RT tasks would not be chosen to run on that CPU until a new period; we call such tasks be throttled.

No matter `CONFIG_RT_GROUP_SCHED` is set or not, in order to avoid RT tasks forever occupy the CPU, there is a system wide setting that constraints RT tasks' execution through the `/proc` virtual file system :

```
/proc/sys/kernel/sched_rt_period_us
/proc/sys/kernel/sched_rt_runtime_us
```

This applies to all RT tasks in a system. By default, The default values for *sched_rt_period_us* and *sched_rt_runtime_us* are 1000000 (1s) and 950000 (0.95s) respectively. This means that RT tasks are limited to 95% of the total system processor time, enforced on a per-second period. The remainder of that time, 5%, is left for CFS tasks.

1.3.2 CFS bandwidth control

Basically, CFS bandwidth control [19] applies the same technique in RT throttling on normal tasks. It is a CFS task group scheduling extension which allows the specification of the maximum CPU bandwidth available to normal tasks in a cgroup or cgroup hierarchy. The kernel configuration item `CONFIG_CFS_BANDWIDTH` allows the user to use this function. The bandwidth assigned to a cgroup is specified using a *quota(cpu.cfs_quota_us)* and a *period(cpu.cfs_period_us)*. By specifying them, normal tasks in a cgroup will be limited to *cfs_quota_us* units of CPU time within the period of *cfs_period_us*. Recall that in RT throttling in last paragraph, the reserved bandwidth through cgroup interfaces are applied in each CPU

individually.

1.3.3 AQuoSA

The Adaptive Quality of Service Architecture composes two parts: a resource reservation scheduler and a feedback-based control mechanism. The scheduler uses CBS rules to reserve CPU bandwidth for a task, which is a RT task with SCHED_RR policy. Given the error between the reserved computation and the amount of CPU cycles really consumed, the feedback controller adapts CBS reservation parameters to provide quality of service CPU allocation in the system. The control mechanism depends on CBS performance, not the scheduling details. That is, such a control mechanism can be applied to general CBS based scheduling. AQuoSA lacks considerations on multi-processor platform.

1.3.4 Sched_deadline patch

The sched_deadline patch for Linux kernel is being developed to extend current mainline Linux with a deadline-based scheduling method. In the work of sched_deadline, a new scheduling class (scheduler) `dl_sched_class` is implemented and has the highest priority among all scheduling classes. Tasks from this scheduling class are assigned deadlines according to CBS rules and scheduled in "earliest deadline first (EDF)" way. So, when creating a deadline task or changing a task's scheduler to `dl_sched_class`, not only you need to specify the scheduling policy (SCHED_DEADLINE in this case), you should also clearly set the CPU bandwidth assigned to this task, corresponding to the *maximum budget* and *period* parameters of a CBS. However, these two parameters are not supported by mainline Linux system calls. This problem solved by importing a new system call interface `sched_setscheduler_ex`, which only allows for creating a reservation attached to a task. On multi-processor platforms, the deadline tasks are scheduled in a global EDF way [20].

1.3.5 IRMOS real-time framework

The name IRMOS comes from the European project "Interactive Real-time Multimedia Applications on Service Oriented Infrastructures". The IRMOS framework replaces RT throttling mechanism in mainline Linux with real time CPU reservation (still CBS). It modifies the RT scheduling class directly and reuses the existing cgroup interfaces for RT throttling. Still, users can configure the cgroup interface as what we saw in RT throttling, the difference is that this time the CPU bandwidth is allocated in a guaranteed way. To control the distribution of reserved CPU bandwidths in the cgroup hierarchy, two new cgroup interfaces are added inside each cgroup directory to specify the CPU bandwidth that is allowed to be further reserved by descendant cgroups, which of course should not exceed the reserved bandwidth in this parent cgroup.

Chapter 2

Design of OXC Scheduling

2.1 Open-Extension Container Structure

Linux scheduling is runqueue, `struct rq`, centered and each scheduling class implements a set of interfaces to deal with the runqueue structure. In mainline Linux, `struct rq` is a per CPU structure. Each scheduling class defines its scheduling operations (`enqueue`, `dequeue`, etc.) with this per CPU runqueue. On Multiple processor platforms, tasks can migrate among different runqueues, also depending on behaviours defined in a specific scheduler. In other words, on each CPU, a scheduling system is built up based on the associated runqueue. Different per CPU scheduling systems cooperate with each other by task migrating operations defined by specific scheduling classes and construct the system level scheduling. One question raised here could be what if there are extra runqueues and how they can be utilized. Ideally, suppose there is one extra runqueue, each scheduler can still use it as the scheduling parameter and a scheduling system can be built around it. If there are more than one extra runqueues, they can form a pseudo system level scheduling system.

Extended from the above idea, a data structure named Open-Extension Container(OXC) is proposed in Linux kernel, shown in figure 2.1. The ox container is designed as an abstract data structure; that is, any data structure contains a `struct rq` runqueue inside can be called the ox container. After

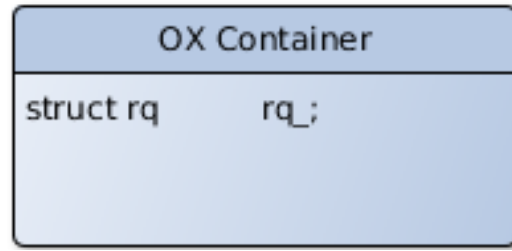


Figure 2.1: Open-Extension Container

bringing this new data structure into the kernel, there is not only per CPU runqueues in the system now, but also per oxc runqueues.

2.2 The oxc scheduling

As there are extra per oxc runqueues besides per CPU ones in a system, they can be candidates passed as parameters to scheduling operations. From the standpoint of a scheduler, it manages a task over the runqueue according to implementation details in the scheduling class, and as long as a runqueue parameter is provided for its scheduling operations, the scheduler does not care whether it is associated with a CPU or from an ox container. So, as a oxc local runqueue is passed to hooks of scheduling classes, **tasks would enqueue, operate and dequeue on a per ox container runqueue**. This is called **the oxc scheduling**. The task enqueued in an ox container's local runqueue is called an oxc task. Of course, an oxc task can be a normal task or a RT task. For tasks or schedulers, the ox container behaves as a virtual CPU equipped with its own runqueue.

2.2.1 Scheduling routes in oxc enabled Linux

For tasks and scheduling classes, there is no difference between a per CPU and per container runqueue. This will be clearly shown when we see the scheduling route under the oxc scheduling. Previously, we saw the path which relates each task with the per CPU runqueue it runs. Figure 1.2 on page 10 is the scheduling route for a task under CFS scheduling when

CONFIG_FAIR_GROUP_SCHED is set; figure 1.4 on page 12 is the scheduling route under RT scheduling when RT group scheduling is enabled; figure 1.3 on page 10 is the scheduling route for a task without task group scheduling. Now we are going to extend the scheduling routes in Linux for the oxc scheduling. As for how to connect different components and build such routes in Linux kernel, it is the content in chapter 3.

When the oxc scheduling joins the system, original scheduling routes will be extended. Nevertheless, it's very natural to adapt Linux scheduling routes to the oxc scheduling. Figure 2.2(a) shows the extended scheduling route for CFS scheduling with CONFIG_FAIR_GROUP_SCHED enabled. The only change in this route is that for an oxc task, the terminal of the path will be a per ox container runqueue instead of its per CPU counterpart. Figure 2.2(b) shows the scheduling route for RT scheduling with CONFIG_RT_GROUP_SCHED set; the situation is similar to CFS scheduling case we just saw. In fact, the same path is used to lead a task to its per CPU or ox container runqueue. This means that the same codes can still be used to find both kinds of runqueues.

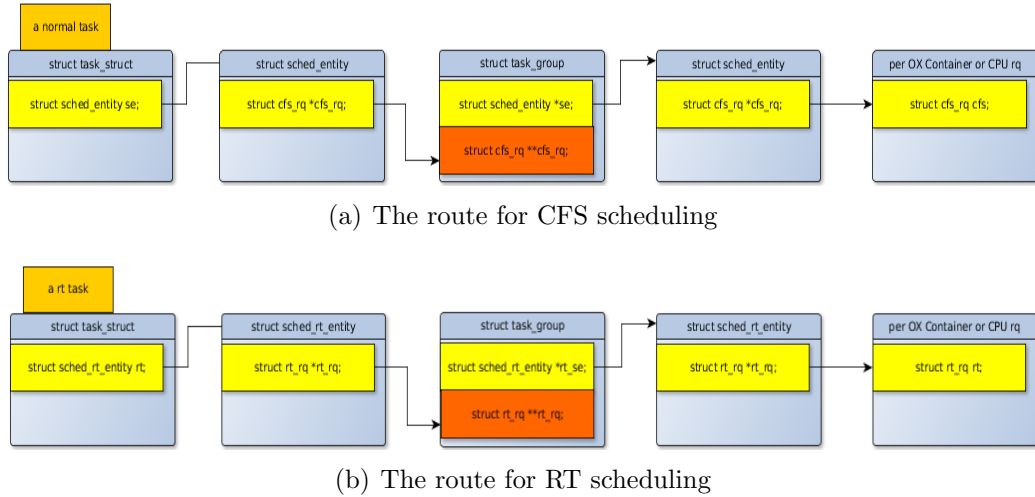


Figure 2.2: Extended scheduling routes with task group scheduling

In section 1.2.3, we introduce that when task group scheduling is not enabled, a macro `task_rq` is used to associate a task to its runqueue. The

`task_rq` is defined as follows:

```
#define task_rq(p)                cpu_rq(task_cpu(p))
```

This macro can only return the dedicated rq for the CPU where the task is currently running on. So, when task group scheduling is not enabled, in order to merge oxc scheduling in the system, a new path leading a task to a runqueue is needed, shown in figure 2.3. This route has two branches. When both RT and CFS group scheduling are not enabled, these two paths coexist for a task since the `struct task_struct` (listing 1.2) embeds both RT and CFS entities. Also for this reason, when both `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED` are set, the two routes in figure 2.2 originate from a task at the same time too.

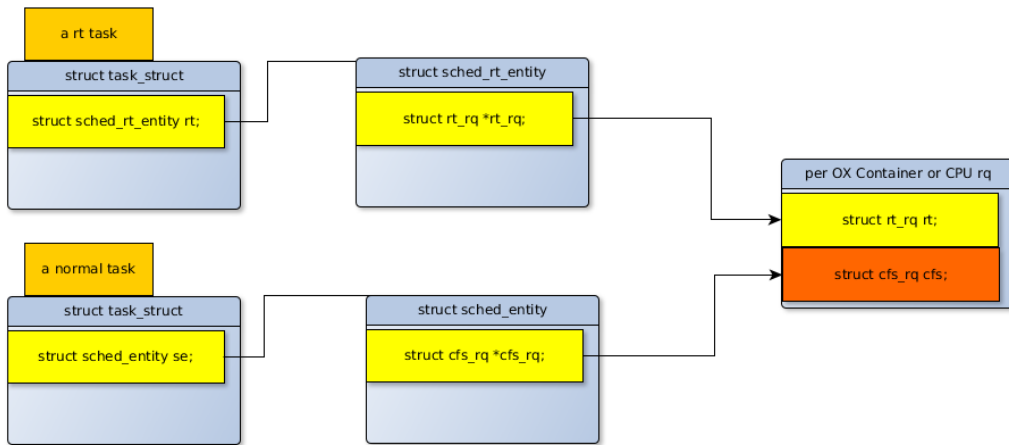


Figure 2.3: Extended scheduling route without task group scheduling

2.2.2 Features of the oxc scheduling

The first feature of oxc scheduling is that it is compatible with Linux's modular scheduling design. Tasks dealt by the RT scheduler or Completely Fair Scheduler can naturally work under the oxc scheduling system. When there is new scheduling algorithm implemented in Linux kernel, like the case in *sched_deadline* patch we mentioned before, the new scheduler has to fulfil details behind scheduling interfaces in `struct sched_class`. Again, for each scheduling class, they do not care whether a per CPU or per container

runqueue is passed to its interfaces as the input, and the new scheduling class can also work under oxc scheduling. So, the ox container structure is **open to extension**; this is where its name comes from.

When there is only per CPU runqueues, each scheduling operation defined in `struct sched_class` will affect all tasks in the CPU. The oxc scheduling provides another opportunity to restrict the influence of a scheduling class in a fine grained scale. Now, the span to apply a scheduler can be controlled in the unit of an ox container.

2.3 Motivation for oxc scheduling framework

In section 1.3, we see that, based on Linux, CPU bandwidth control can be applied in the level of single tasks or task groups which are scheduled by some policies. Such controls can be real-time or non real-time. One observation is that there does not exist a mechanism that can control CPU bandwidth for all kinds of tasks as a whole without requirement of a task's scheduling details.

Suppose there is an ox container, different schedulers can use it to enqueue, operate, and dequeue their tasks. If a fraction of CPU bandwidth can be allocated to this ox container, all kinds of tasks inside the container will use it as if running on a less powerful machine. This is the oxc solution to the CPU bandwidth control for general types of tasks.

Based on this idea, we develop an oxc scheduling framework in Linux that can realize hierarchical multi-processor reservation for tasks without requiring scheduling policies. In the following chapter, details on how to implement this oxc scheduling framework in Linux kernel are described.

Chapter 3

Development of OXC Framework

The oxc framework is still ongoing. Latest codes can be found in github¹. The oxc framework is not a scheduler. Although it cooperates with different scheduling classes, its pure responsibility is managing the distribution of CPU power, which depends on modular schedulers to use it for scheduling tasks. Under oxc framework, we also call oxc scheduling as oxc control since it is utilized to control CPU bandwidth reservation.

3.1 Implementation of ox container structure

An ox container `struct oxc_rq`, listing 3.1, is defined in Linux kernel. In `struct oxc_rq` there are fields for reserving bandwidth from a CPU using CBS rules. In the following context, an `struct oxc_rq` instance is also called an ox container, container, or oxc runqueue for the same meaning. An oxc runqueue also corresponds to a constant bandwidth server in CBS theory. The CPU reservation for oxc runqueues follows the implementation of CBS reservation for a RT runqueue in IRMOS framework.

Listing 3.1: The ox container : `struct oxc_rq`

```
struct oxc_rq {
```

¹<https://github.com/YTYAYTYAYOUCHENG/linux>

```

unsigned long oxc_nr_running;
inx oxc_throttled;
u64 oxc_deadline;
u64 oxc_time;
u64 oxc_runtime;
ktime_t oxc_period;
struct hrtimer oxc_period_timer;
raw_spin_lock oxc_runtime_lock;
struct rq rq_;
struct rq *rq;
struct rb_node rb_node;
};

```

- `oxc_nr_running` is the number of oxc tasks enqueued in the container's local runqueue. We say these tasks work in the ox container.
- `oxc_throttled` is set when an ox container runs out of its budget in a period. Here "throttled" is a convention inherited from RT throttling and maps to the suspended state of a CBS.
- `oxc_deadline` is current deadline of this ox container, which is a server in CBS theory.
- `oxc_time` is currently consumed budget in a period.
- `oxc_runtime` and `oxc_period` are CBS parameters: `oxc_runtime` is maximum budget and `oxc_period` is the period.
- `oxc_period_timer` is timer which will activate at recharging time. If at some point, the value of `oxc_time` is larger than the value of `oxc_runtime`, then the ox runqueue will be throttled and its timer will be set to fire, at the current deadline, `oxc_deadline`, to recharge the container.
- `oxc_runtime_lock` guarantees that the update of an ox container's reservation parameters (`oxc_runtime` and `oxc_period`) and internal state (`oxc_time` and `oxc_deadline`) happen in a consistent way.

- `rq_` is the local runqueue of the ox container.
- `rq` points to a per CPU runqueue and its CPU is where the ox container reserves bandwidth from.
- `rb_node` is used to put an ox runqueue in a red black tree. All ox runqueues reserve bandwidth from the same CPU are sorted in a red-black tree. In this tree, an ox container's `oxc_deadline` value is used to order nodes.

For each CPU, there is a red-black tree which stores all ox runqueues that reserve bandwidths in this CPU and orders them with their current deadline. The ox runqueue with earliest deadline is stored in the leftmost node. This tree is called the edf tree and is defined in `struct oxc_edf_tree`.

Listing 3.2: The EDF tree

```
struct oxc_edf_tree {
    struct rb_root rb_root;
    struct rb_node *rb_leftmost;
};
```

The pointer `rb_leftmost` helps fast access to the earliest deadline ox container in a CPU.

An ox container is responsible for reserving bandwidth from a CPU. Another data structure `struct hyper_oxc_rq` is defined to reserve bandwidth from multiple CPUs. A `struct hyper_oxc_rq` instance is called a hyper ox container.

Listing 3.3: The hyper ox container

```
struct hyper_oxc_rq {
    cpumask_var_t cpus_allowed;
    struct oxc_rq ** oxc_rq;
};
```

- `cpus_allowed` specifies the CPUs that are used to reserve bandwidth.
- `oxc_rq` is an array of (pointers to) ox containers to reserve bandwidth from CPUs identified in `cpus_allowed`.

3.2 Extensions on original data structures

Several new data structures have been imported in the kernel that will involve with Linux schedulers, yet the interfaces defined in `struct sched_class` are kept unmodified. Extensions are added in some original data structures in order to merge newly defined data structures in the system. Such extensions are not complex.

Listing 3.4: Extensions on `struct rq`

```
struct rq {
    ...
    int in_oxc;
    struct oxc_edf_tree oxc_edf_tree;
};
```

Two fields are added in `runqueue` structure. The `in_oxc` is used to distinguish per CPU and ox container `runqueue`. As the name says, for an ox container's local `runqueue`, its `in_oxc` field is set. And `oxc_edf_tree` in a per CPU `runqueue` is the edf tree for a CPU to keep and sort ox containers.

The kernel configuration option `CONFIG_CGROUP_SCHED` is required by the oxc framework. This option allows to create arbitrary task groups using the "cgroup" pseudo filesystem. In current implementation of oxc framework, reservation is made for tasks in a control group. Tasks in a cgroup is represented by the `struct task_group` structure. And extensions also happen inside it.

Listing 3.5: Extensions on `struct task_group`

```
struct task_group {
    ...
    struct hyper_oxc_rq *hyper_oxc_rq;
    int oxc_label;
};
```

Because tasks in a cgroup can span multiple CPUs, `struct task_group` is a good place to put the hyper ox container. If a task group has been allocated CPU bandwidths through oxc control, then this task group runs inside a hyper ox container, and its `hyper_oxc_rq` points to that hyper container;

otherwise, this field is NULL. As before, a task group inside a hyper ox is called an oxc task group. There are three types of oxc task groups: an oxc group whose father is not an oxc task group; an oxc task group whose father is an oxc group with a different hyper ox container; an oxc task group with the same hyper ox container as its father. The `oxc_label` field is used to differ them. For non oxc task group, this field is not utilized.

When oxc scheduling is enabled in the kernel, there are two kinds of tasks in the system: oxc tasks and non oxc tasks. To difference between them is that oxc tasks work is (or will be) enqueued in an ox container's local runqueue and non oxc tasks work with a per CPU runqueue. So, from a task, its associated runqueue can be tracked according to the scheduling route in figure 2.2 or 2.3. As long as the runqueue is found, given its `in_oxc` field, the status of this runqueue and the task can both be fixed. Consider that the question "is that an oxc task?" happens frequently in the framework, a `is_oxc_task` field is added in `struct task_struct` for efficient recognition of an oxc task.

Listing 3.6: `is_oxc_task` field in `struct task_struct`

```
struct task_struct {
    int is_oxc_task;
    ...
};
```

When a task runs in an ox container, this new field is set.

3.3 To direct a task to a per ox container runqueue

In section 2.2, we show the scheduling routes when there exists oxc scheduling in a system. This section will introduce the details on how to build these scheduling routes.

3.3.1 To build the scheduling route in mainline Linux

In order to schedule a task in a per oxc runqueue, the first thing is to associate this task with the local runqueue of an oxc. To understand this, let's first see how the system associates a task to a runqueue in mainline Linux, where there is only per CPU runqueues. This is done through the method `set_task_rq`.

Listing 3.7: To associate tasks with a per CPU runqueue in mainline Linux

```
void set_task_rq(struct task_struct *p, unsigned int cpu)
{
#ifdef CONFIG_FAIR_GROUP_SCHED
    p->se.cfs_rq = task_group(p)->cfs_rq[cpu];
    p->se.parent = task_group(p)->se[cpu];
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    p->rt.rt_rq = task_group(p)->rt_rq[cpu];
#endif
}
```

As demonstrated in list 3.7, codes inside `set_task_rq` build up the front part of the scheduling route when `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED` are set. When RT or CFS task group scheduling is enabled, each task is then directed to its task group. In mainline Linux, the second part of a scheduling route only directs a task group to the per CPU runqueues. Such paths are connected when the task group is created. Figure 3.1 shows the hint how a task group joins the scheduling route during its creation. In addition, now we know that a scheduling route is built backwards from an runqueue end.

In case that task group scheduling is not enabled, recall the scheduling path where the `task_rq` leads a task to its per CPU runqueue directly.

3.3.2 To build the scheduling route in oxc enabled Linux

Now, we have seen the point when a task or task group joins the scheduling route in mainline Linux. These points are still time to fill elements in

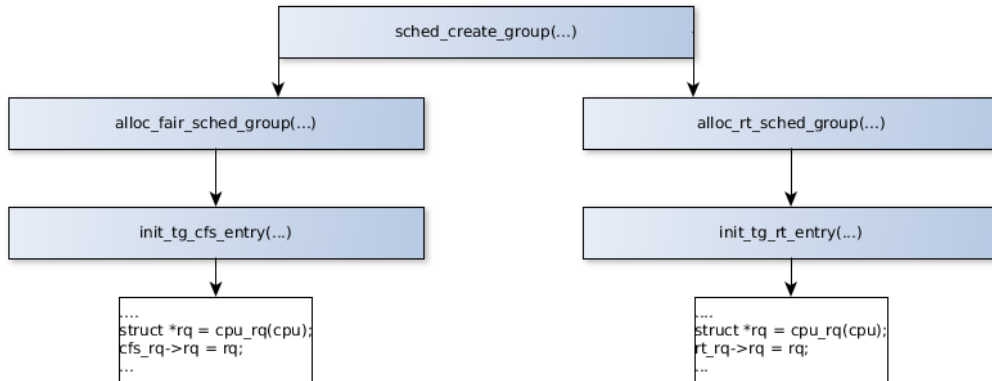


Figure 3.1: The creation of a task group in original Linux

scheduling routes, figure 2.2 and 2.3, after oxc runqueues are imported in Linux.

Previously, the `set_task_rq` method does not deal with the task that is not in a RT or CFS task group. This is because for tasks without group scheduling, the scheduling route `task_rq` is utilized. However, `task_rq` does not work for an oxc task to locate the per container runqueue. The functionality of `set_task_rq` is then extended to care about tasks without group scheduling.

Listing 3.8: The extended `set_task_rq`

```

void set_task_rq(struct task_struct *p, unsigned int cpu)
{
    struct task_group *tg = task_group(p);

#ifdef CONFIG_FAIR_GROUP_SCHED
    p->se.cfs_rq = tg->cfs_rq[cpu];
    p->se.parent = tg->se[cpu];
#else
    if(!tg->hyper_oxc_rq)
        p->se.cfs_rq = &cpu_rq(cpu)->cfs;
    else
        p->se.cfs_rq = &tg->hyper_oxc_rq->oxc_rq[cpu]->
            rq_.cfs;
#endif
}

```

```

#ifdef CONFIG_RT_GROUP_SCHED
    p->rt.rt_rq = tg->rt_rq[cpu];
    p->rt.parent = tg->rt_se[cpu];
#else
    if(!tg->hyper_oxc_rq)
        p->rt.rt_rq = &cpu_rq(cpu)->rt;
    else
        p->rt.rt_rq = &tg->hyper_oxc_rq->oxc_rq[cpu]->
            rq_.rt;
#endif
    if (rq_of_task(p)->in_oxc == 1)
        p->is_oxc_task = 100;
    else
        p->is_oxc_task = 0;
}

```

When task group scheduling is enabled, there is no difference for setting a task's runqueue in both the mainline Linux and oxc enabled Linux. The interesting part happens when task group scheduling is not set. This time, given that the task group is associated with a hyper oxc or not, the task is directed to a per CPU runqueue or per an oxc runqueue (the container is supplied by the `hyper_oxc_rq`). This corresponds to the first part of the scheduling route in figure 2.3. In the end of the method, `is_oxc_task` label of a task is configured. When `task_rq_oxc` is invoked, the scheduling route for a task has already been set. It tracks the scheduling route to find the runqueue that a task is just associated. After this extended `set_task_rq` call, both routes with or without group scheduling are built up.

In the mainline Linux, the end part of a scheduling route is built up when a task group is created. Within the oxc enabled kernel, things will be a little more complex. In the oxc applied Linux, a task group can be associated with a hyper oxc and the contained runqueues in three cases: 1) If its parent is associated with a hyper oxc, then when it is created it will inherit its parent's hyper ox container. 2) The task group is explicitly attached to a hyper oxc. 3) When the group's one ascendant task group is attached to a hyper oxc, its `hyper_oxc_rq` field will point to that hyper oxc too.

Corresponding to case 1, now when a task group is created, there will

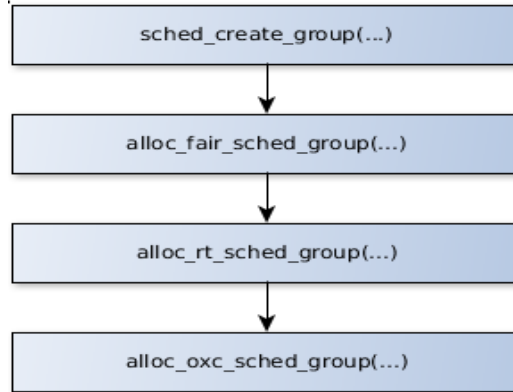


Figure 3.2: The creation of a task group in oxc enabled Linux

be a initialization routine for oxc scheduling. A sketch is shown in figure 3.2. The details of this routine `alloc_oxc_sched_group` is shown below.

Listing 3.9: OXC scheduling related initilization during task group creation

```

int alloc_oxc_sched_group(struct task_group *tg, struct
    task_group *parent)
{
    int i;

    tg->hyper_oxc_rq = parent->hyper_oxc_rq;
    if( parent->hyper_oxc_rq) {
        for_each_possible_cpu(i) {
#ifdef CONFIG_FAIR_GROUP_SCHED
            tg->cfs_rq[i]->rq =
                &tg->hyper_oxc_rq->oxc_rq[i]->
                    rq_;
            if( !parent->se[i] && tg->se[i])
                tg->se[i]->cfs_rq =
                    &tg->hyper_oxc_rq->oxc_rq[
                        i]->rq_.cfs;
#endif
#ifdef CONFIG_RT_GROUP_SCHED
            tg->rt_rq[i]->rq =
                &tg->hyper_oxc_rq->oxc_rq[i]->
                    rq_;
            if( !parent->rt_se[i] && tg->rt_se[i])

```

```

                                tg->rt_se[i]->rt_rq =
                                    &tg->hyper_oxc_rq->oxc_rq
                                        [i]->rq_.rt;
#endif
                                }
                                tg->oxc_label = 100;
                                }
                                else
                                    tg->oxc_label = 0;

                                return 1;
}

```

The `alloc_oxc_sched_group` handles oxc related initialization when a new task group is created. At first, the a newly created task group will inherit its parent task group's hyper container. If the parent is an oxc task group, this child task group will be directed to per oxc runqueues contained in the hyper oxc; the result corresponds to the end part of a scheduling route. And the `oxc_label` for such a child oxc task group is 100.

Case 2 and 3 actually happen at the same time. When explicitly putting the parent group inside a hyper ox container, the descendant task groups will also be directed to this hyper oxc. Two methods `init_tg_cfs_entry_oxc` and `init_tg_rt_entry_oxc` will be involved in this procedure to associate task groups with the hyper ox container. The structure in the two is similar. As an example, Here we only study how the CFS part of a task group is handled through `init_tg_cfs_entry_oxc`.

Listing 3.10: To explicitly direct a task group (CFS part) to an OXC local runqueue

```

static void init_tg_cfs_entry_oxc(struct task_group *tg,
                                struct cfs_rq *cfs_rq,
                                struct sched_entity *se, int cpu,
                                struct sched_entity *parent,
                                struct oxc_rq *oxc_rq)
{
    struct rq *rq = rq_of_oxc_rq(oxc_rq);
    init_tg_cfs_entry(tg, cfs_rq, se, cpu, parent);
}

```

```

tg->cfs_rq[cpu]->rq = rq;
if( !parent && se)
    se->cfs_rq = &rq->cfs;
}

```

A brief explanation on the parameters:

- `tg` is the task group to be dealt with.
- `cfs_rq` is the CFS runqueue that this task group owns.
- `se` is the CFS entity that represents `tg`.
- `cpu` specifies the `cfs_rq` pointer inside `tg` that will be redirected. The function `init_tg_cfs_entry_oxc` redirects one CFS runqueue inside a `tg` to an `oxc_rq`, in each call. A hyper ox container can have more than one `oxc_rq` inside and `tg` has multiple CFS runqueues too. To relate a task group with a hyper ox container requires the `init_tg_cfs_entry_oxc` be called multiple times.
- `parent` points to the parent CFS scheduling entity.
- `oxc_rq` contains the destined runqueue.

As for codes, first `rq_of_oxc_rq` returns the `oxc` local runqueue. The mainline kernel method `init_tg_cfs_entry` is then called to re-initialize CFS related work for `tg`. The real bridging point is in the following line where `tg` connects its local `cfs_rq` on `cpu` to the per `oxc` runqueue just got. After `init_tg_cfs_entry_oxc` is invoked over every CPU for the task group, CFS tasks and task groups under `tg` will work in the scheduling route leading them to an `oxc` local runqueue. When a task group is explicitly directed to a hyper ox container, the whole family of task groups under it will also be associated with this hyper ox container. This task group will be the top of this hierarchy, and it will be enqueued in the per `oxc` runqueue's embedded CFS runqueue directly. The last `if` condition returns true when `tg` is top in the `oxc` group hierarchy.

3.4 Run tasks under OXC scheduling framework

As long as per oxc runqueue joins the scheduling route, modular schedulers can handle tasks in an ox container without differentiating oxc tasks from non oxc tasks. In case to schedule an oxc task, just pass the task itself and its oxc local runqueue, instead of per CPU runqueue, to its corresponding scheduling class. The local runqueue of an oxc task can be easily tracked along the scheduling route. And the scheduler will behave as usual. That is, this oxc scheduling framework is transparent to both schedulers and tasks.

Because we consider reserving CPU bandwidth for an ox container, there are scheduling operations that before or after passing parameters to them, the reservation information should be updated. For these kinds of scheduling operations, we adapt a relaying mechanism. The parameter is first passed to another function and after necessary actions, the scheduling operation is called inside this function. We say that such scheduling operations are encapsulated in oxc (scheduling) functions. Still scheduling details for a task are not the framework's work.

In order to fulfil real time guarantee, oxc tasks are always privileged to non oxc tasks. Among oxc tasks inside a container, the priority relation is the same as in Linux. The ox container itself can be considered as a virtual Linux system.

For each scheduling operation defined in `struct sched_class`, there are three fates for them under oxc scheduling framework: some are encapsulated inside oxc functions so as to work with oxc tasks; some can work under the framework directly; and others are not supported. The table 3.1 displays the three classes of scheduling operations. The naming convention for oxc functions which encapsulate a scheduling operation inside is appending the original name with `_oxc`. For example, the scheduling operation `task_tick` is called inside in `task_tick_oxc`. The enqueue and dequeue are two exceptions, they are enclosed in `enqueue_task_oxc` and `dequeue_task_oxc`.

The following will be descriptions on oxc functions with emphasis on ones that encapsulate a scheduling operation inside.

Work inside an oxc function	Work without encapsulation	Unsupported
check_preempt_curr pick_next_task put_prev_task set_curr_task task_tick enqueue_task_rq dequeue_task_rq	yield_task yield_to_task task_waking task_woken set_cpus_allowed task_fork switched_from switched_to prio_changed get_rt_interval task_move_group	Others

Table 3.1: The way to handle a scheduling operation under the oxc framework

3.4.1 To obtain the runqueue of a task

A method `rq_of_task`, listing 3.11, is used to obtain the runqueue of a task. The runqueue returned can be an oxc local runqueue or per CPU one depending that whether the task is inside a container. This function can be used to replace the `task_rq` macro under oxc framework. For any task, it has both CFS scheduling entity and RT entity. And the RT and CFS scheduling routes both exists for a task. Here, we exploit the CFS scheduling route. Given `CONFIG_FAIR_GROUP_SCHED` is set or not, the corresponding scheduling route is explored to track the runqueue.

Listing 3.11: Codes to obtain the runqueue of a task

```

struct rq* rq_of_task(struct task_struct *p)
{
    struct rq *rq;

#ifdef CONFIG_FAIR_GROUP_SCHED
    rq = p->se.cfs_rq->rq;
#else
    rq = task_rq_fair_oxc(p);
#endif
    return rq;
}

```

One important use of this function is in `oxc_rq_of_task`, which takes a task as the input, and returns the ox container the task is inside. In case the input is not an oxc task, NULL will be returned. The `oxc_rq_of_task` is one most often invoked method under oxc framework. It first gets a task's runqueue using the `rq_of_task`, then obtains the ox container the runqueue is in, in case the runqueue is in an container.

3.4.2 To enqueue an oxc task

When an oxc task arrives, besides enqueue it in the oxc local runqueue, the ox container's reservation information may be updated if necessary. The `enqueue_task_oxc` shows the typical scheme for oxc functions to enclose a scheduling operation.

Listing 3.12: To enqueue an task into the oxc local runqueue

```
void enqueue_task_oxc(struct rq *rq, struct task_struct *p, int
    flags)
{
    struct oxc_rq *oxc_rq = oxc_rq_of_task(p);
    struct rq *rq_ = rq_of_oxc_rq(oxc_rq);

    /* Update the local runqueue' clock. */
    update_rq_clock(rq_);

    /*
     * Enqueue the task into the local runqueue
     * by its scheduling class.
     */
    p->sched_class->enqueue_task(rq_, p, flags);

    inc_oxc_tasks(p, oxc_rq);
    enqueue_oxc_rq(oxc_rq);
}
```

The method `oxc_rq_of_task` tracks the scheduling route and returns the ox runqueue of an oxc task. Then the ox container's local runqueue's time information is updated. Although the ox container does not care about

scheduling details of tasks inside it, tasks are indeed enqueued in its local runqueue and may rely the runqueue's time information. We can see that all scheduling details are carried out by a task's scheduling class as the `enqueue_task` operation of the scheduling class is called with the task and local runqueue as inputs. The `inc_oxc_tasks` method simply increases the number of oxc tasks in the oxc runqueue by one. The reason that such a simple function is still kept is that as the framework grows more complex in the future, extra operations can be put in this method.

Listing 3.13: To update the number of tasks inside a container

```
static inline void
inc_oxc_tasks(struct task_struct *p, struct oxc_rq *oxc_rq)
{
    oxc_rq->oxc_nr_running ++;
}
```

Until now, the oxc task has been put in the local runqueue. If before the arrival of this task the ox container is empty and not throttled, it is time to put this container in its EDF tree. This is the work of `enqueue_oxc_rq` method.

Listing 3.14: To put an ox container in the EDF tree if necessary

```
static void enqueue_oxc_rq(struct oxc_rq *oxc_rq)
{
    int on_rq;

    on_rq = oxc_rq_on_rq(oxc_rq);

    BUG_ON(!oxc_rq->oxc_nr_running);
    BUG_ON(on_rq && oxc_rq_throttled(oxc_rq));

    if( on_rq) {
        /* Already queued properly. */
        return;
    }
    /* We do not put a throttled oxc_rq in the edf tree. */
    if( oxc_rq_throttled(oxc_rq))
        return;
```

```

    oxc_rq_update_deadline(oxc_rq);
    __enqueue_oxc_rq(oxc_rq);
}

```

`on_rq` tells if the container `oxc_rq` is in a edf tree. `BUG_ON` is a Linux kernel macro. If the condition it checks is `true`, then the kernel will crash! Because we just put a task in the local runqueue, so the first `BUG_ON` should be passed. When an ox container runs out of its budget, it should be moved from the edf tree, this is what the second `BUG_ON` checks. Now we pass the two `BUG_ON`s. If the `oxc_rq` is already on edf tree or moved from the tree because of exhausting budget, nothing needs to be done. There are two conditions for an ox container to be outside an edf tree: it is throttled or it is empty. If the last `if` condition is passed, this means a task just joins an empty container. Recall the CBS rules "when a task arrives and the server is idle, update the deadline if necessary". This is exactly what `oxc_rq_update_deadline` does. When it comes to CPU reservation, an ox container corresponds to a constant bandwidth server in CBS theory. `_enqueue_oxc_rq` is a quite mechanical procedure to put an oxc runqueue in an edf tree.

3.4.3 To dequeue an oxc task

`dequeue_task_oxc` is the opposite method of `enqueue_oxc_rq`.

Listing 3.15: To remove a task from the oxc local runqueue

```

static void
dequeue_task_oxc(struct rq *rq, struct task_struct*p, int flags)
{
    struct rq *rq_ = rq_of_task(p);
    struct oxc_rq *oxc_rq = container_of(rq_, struct oxc_rq,
        rq_);

    /* Update the local runqueue. */
    update_rq_clock(rq_);
    /*
     * Dequeue the task from the local runqueue

```

```

        * by its scheduling class.
        */
    p->sched_class->dequeue_task(rq_, p, flags);

    dec_oxc_tasks(p, oxc_rq);
    dequeue_oxc_rq(oxc_rq);
}

```

The layout inside `dequeue_task_oxc` is the same as what we see in `enqueue_task_oxc`: local runqueue's clock information is updated, local runqueue and task are relayed to the corresponding scheduler, the task number and EDF tree are updated. When an oxc task leaves the container, it is the time to check if the oxc is empty or not, which is done in `dequeue_oxc_rq`.

Listing 3.16: To remove an ox container from the EDF tree if necessary

```

static void dequeue_oxc_rq(struct oxc_rq *oxc_rq)
{
    int on_rq;

    on_rq = oxc_rq_on_rq(oxc_rq);
    /*
     * Here we do not expect throttled oxc_rq to be in the
     * edf tree. Note that when an oxc_rq exceeds its
     * maximum budget, it is dequeued via
     * sched_oxc_rq_dequeue().
     */
    BUG_ON(on_rq && oxc_rq_throttled(oxc_rq));
    /*
     * If an oxc_rq is not in the edf tree, it should
     * be throttled or have no tasks enqueued.
     */
    BUG_ON(!on_rq && !oxc_rq_throttled(oxc_rq) && !oxc_rq->
        oxc_nr_running);

    if( on_rq && !oxc_rq->oxc_nr_running) {
        /* Dequeue the oxc_rq if it has no tasks. */
        __dequeue_oxc_rq(oxc_rq);
        return;
    }
}

```

```
    }
}
```

The comments are explainable enough. `__dequeue_oxc_rq` is the counterpart of `__enqueue_oxc_rq` to remove an oxc runqueue from the EDF tree.

3.4.4 To check the preemption condition

When a task wakes up from sleeping or is created, the scheduler will check if it can preempt temporarily running task in the same CPU. If the current task is an oxc task and the waking task is not an oxc task, the later cannot preempt the current one. If both are non oxc tasks, Linux already has methods to check. So here we only pay attention to the situation when the waking task is an oxc task.

Listing 3.17: The preemption check for an oxc task

```
static inline int
check_preempt_oxc_rq(struct task_struct *curr, struct
    task_struct *p, int flags)
{
    struct oxc_rq *oxc_rq = oxc_rq_of_task(p);
    struct oxc_rq *oxc_rq_curr = oxc_rq_of_task(curr);
    const struct sched_class *class;

    if(oxc_rq_throttled(oxc_rq)
        return 0;
    /*
     * Tasks from a unthrottled oxc_rq always has a higher
     * priority than non oxc tasks.
     */
    if( !oxc_rq_curr)
        return 1;

    /* Both p and current task are in the same oxc_rq. */
    if( oxc_rq_curr == oxc_rq) {
        if( p->sched_class == curr->sched_class)
            curr->sched_class->check_preempt_curr(
```

```

                                &oxc_rq->rq_, p, flags);

    else {
        for_each_class(class) {
            if( class == curr->sched_class)
                break;
            if( class == p->sched_class) {
                resched_task(curr);
                break;
            }
        }

        return 0;
    }

    /*
     * p and current tasks are oxc tasks from
     * different ox containers.
     */
    return oxc_rq_before(oxc_rq, oxc_rq_curr);
}

```

`curr` and `p` are the current task and waking task respectively. If task `p`'s container is throttled, it cannot preempt currently running task. Otherwise, if `curr` is not an oxc task, the privilege is given to oxc task `p`. When both `p` and `curr` are oxc tasks and are contained in the same oxc runqueue: if they are even in the same scheduling class, it's the modular scheduler's responsibility to decide if a preemption can happen or not; otherwise, the task whose scheduling class has higher priority in the scheduler chain is chosen to run. In the last case, they two are from different ox containers. Now, `oxc_rq_before` checks if a container's deadline is before another's deadline. The one with earlier deadline will run.

3.4.5 To pick up an oxc task

When to pick a most eligible task to run in a system, oxc tasks should be checked first. If there is no eligible oxc tasks, then non oxc tasks are con-

sidered. `pick_next_task_oxc` is responsible for choosing the most eligible oxc task in a CPU.

Listing 3.18: To pick up the most eligible oxc task

```
static struct task_struct* pick_next_task_oxc(struct rq *rq)
{
    struct oxc_rq *oxc_rq;
    struct rq *rq_;
    struct task_struct *p, *curr;
    const struct sched_class *class;
    /* This clock update is necessary! */
    update_rq_clock(rq);
    update_curr_oxc_rq(rq);
    oxc_rq = pick_next_oxc_rq(rq);
    if( !oxc_rq)
        return NULL;

    rq_ = rq_of_oxc_rq(oxc_rq);

    update_rq_clock(rq_);

    for_each_class(class) {
        if( class != &idle_sched_class) {
            p = class->pick_next_task(rq_);
            if( p) {
                rq_->curr = p;
                return p;
            }
        }
    }

    return NULL;
}
```

Inside the oxc function `pick_next_task_oxc`, there is one thing to note: not only local runqueue's clock is updated, but also the per CPU runqueue's clock is updated here. This is because the reservation time of an ox container is counted using the per CPU runqueue's clock and to keep the clock on time for the container's use, here it is updated. `pick_next_oxc_rq` is called to

pick the ox runqueue with the earliest deadline in a CPU. Then along the scheduling class chain, each scheduler uses its scheduling operation trying to find the most eligible task in the ox container's local runqueue. A very important method here is `update_curr_oxc_rq`. The budget consumption of an ox container actually happens in this method.

Listing 3.19: To update an ox container's runtime information

```
static void update_curr_oxc_rq(struct rq *rq)
{
    struct task_struct *curr = rq->curr;
    struct oxc_rq *oxc_rq = oxc_rq_of_task(curr);
    u64 delta_exec;
    /*
     * If current task is not oxc task, simply return.
     */
    if( !oxc_rq)
        return;

    delta_exec = rq->clock - oxc_rq->oxc_start_time;
    oxc_rq->oxc_start_time = rq->clock;
    if( unlikely((s64)delta_exec < 0))
        delta_exec = 0;

    raw_spin_lock(&oxc_rq->oxc_runtime_lock);

    oxc_rq->oxc_time += delta_exec;
    if( sched_oxc_rq_runtime_exceeded(oxc_rq)) {
        resched_task(curr);
    }

    raw_spin_unlock(&oxc_rq->oxc_runtime_lock);
}
```

`update_curr_oxc_rq` updates the runtime value of an oxc runqueue. If the budget in current period is exhausted, the current task needs to be rescheduled. the local spinlock `oxc_runtime_lock` protects the update of runtime from interleave. The `sched_oxc_rq_runtime_exceeded` is used to check if the ox container has exceeded its budget in a period.

Listing 3.20: To check if an ox container should be throttled

```

static int sched_oxc_rq_runtime_exceeded(struct oxc_rq *oxc_rq)
{
    u64 runtime = sched_oxc_rq_runtime(oxc_rq);
    u64 period = sched_oxc_rq_period(oxc_rq);

    /*
     * If the runtime is set as 'RUNTIME_INF',
     * the ox container can run without throttling.
     */
    if( runtime == RUNTIME_INF)
        return 0;

    /*
     * If the runtime to be larger the the period,
     * the ox container can run without throttling.
     */
    if( runtime >=period)
        return 0;

    /* There is still budget left. */
    if( oxc_rq->oxc_time < runtime)
        return 0;

    /*
     * The reservation in a period has been exhausted,
     * to set the throttling label, remove the oxc_rq
     * from the edf tree and start the recharging timer.
     */
    else {
        oxc_rq->oxc_throttled = 1;
        sched_oxc_rq_dequeue(oxc_rq);
        start_oxc_period_timer(oxc_rq);

        return 1;
    }
}

```

Inside `sched_oxc_rq_runtime_exceeded`, at first a series of non exceeded conditions is checked, which is easy to understand. The last *else*

statement deals with the case that the container should be throttled: the `oxc.throttled` label is set, the `oxc` runqueue is removed from the `edf` tree and the timer is set to fire at the current deadline, `oxc_deadline`, to recharge the budget.

3.4.6 `put_prev_task_oxc`

The scheduling operation `put_prev_task` is called when the currently running task is possible to be replaced. It performs some conclusion work for the task. Although the currently running task may keep running without giving up the CPU. If currently running task is an `oxc` task, when `put_prev_task` is called, this is also a point to update the `ox` container runtime information.

Listing 3.21: Conclusion work before an `oxc` task is switched out of a CPU

```
static void
put_prev_task_oxc(struct rq* rq, struct task_struct *p)
{
    struct rq *rq_ = rq_of_task(p);

    update_rq_clock(rq_);
    update_curr_oxc_rq(rq);

    p->sched_class->put_prev_task(rq_, p);
}
```

Now, when a `put_prev_task` operation is needed and the currently running task is an `oxc` task, instead of calling the `put_prev_task` defined in a scheduling class directly, our `put_prev_task_oxc` encapsulation will be called first then `ox` container local runqueue and current task will be relayed to the corresponding scheduler.

3.4.7 `set_curr_task_oxc`

The `put_prev_task/set_curr_task` is used whenever the current task is changing policies or groups. Similiar as what happens to `put_prev_task`, there is an `oxc` function `set_curr_task_oxc` that performs the operation

`set_curr_task_oxc` inside and start an ox container's budget counting.

Listing 3.22: Another point to update reservation information

```
static void set_curr_task_oxc(struct rq *rq)
{
    struct task_struct *curr = rq->curr;
    struct rq *rq_ = rq_of_task(curr);
    struct oxc_rq *oxc_rq;

    oxc_rq = container_of(rq_, struct oxc_rq, rq_);

    oxc_rq->oxc_start_time = oxc_rq->rq->clock;

    update_rq_clock(rq_);
    curr->sched_class->set_curr_task(rq);
}
```

One thing to note in listing 3.22 is that this time the per CPU runqueue parameter is passed to the task's corresponding scheduling class. First of all, this is feasible because `set_curr_task` operation updates the information in the scheduling route excluding the runqueue (at least in scheduling classes in mainline Linux) and the current task in the per CPU runqueue and current task in the earliest deadline oxc runqueue are the same one. So, there is no difference to pass which runqueue. The real reason is that there is a possible inconsistent state under current oxc framework. Initially, the current task in an ox container's local runqueue is NULL², which is different from the per CPU runqueue's behaviour that the default current task would be the special idle task from scheduling class `sched_idle`. At this time, if the current task is moved into this ox container, the `set_curr_task` operation is called even before the task is really enqueued into the container's local runqueue. That is, the current task in the local runqueue is still NULL when `set_curr_task` is called and this will cause problems. Thus, a per CPU runqueue is used here temporarily.

²This will be improved in future work

3.4.8 task_tick_oxc

The `task_tick` operation is the most frequently called scheduling operation and is used to update the task's timing information. The oxc control also utilizes this opportunity to update the current ox container's runtime information and local runqueue when the CPU is occupied by an oxc task.

Listing 3.23: The most frequently called entry to update a container's runtime

```
static void
task_tick_oxc(struct rq *rq, struct task_struct *p, int queued)
{
    struct rq *rq_ = rq_of_task(p);

    update_curr_oxc_rq(rq);
    update_rq_clock(rq_);

    p->sched_class->task_tick(rq_, p, queued);
}
```

3.5 SMP support in oxc framework

An ox container based scheduling system acts with no difference as the per CPU scheduling system. Inside different ox containers, the scheduling is performed independently and a group of ox containers can be arranged in one hyper container to realize multi-processor bandwidth reservation. For a hyper ox container, tasks are partitioned to each ox container and task migration or load balancing between different ox containers in a hyper container are missing now.

3.6 User interfaces provided by OXC framework

Currently, the user interfaces, mainly for test purposes, of oxc framework are based on cgroup virtual filesystem. The CPU reservation functionality is realized through *cpu* cgroup subsystem. This *cpu* cgroup subsystem is for CPU bandwidth control in Linux. RT throttling, CFS group scheduling and CFS bandwidth control are all realized through it. The following demonstrates how to use the oxc framework. The hardware platform is assumed to have dual processors.

To mount the *cpu* cgroup subsystem(in directory /cgroup):

```
#mount -t cgroup -ocpu none /cgroup
```

To create a cgroup for CPU reservation :

```
#mkdir -p /cgroup/cg
```

Observe the files inside /cgroup/cg directory, there is one new file named `cpu.oxc_control` and it is the interface to control CPU bandwidth in oxc framework. However, if one tries to see the content of this file:

```
#cat /cgroup/cg/cpu.oxc_control
```

It will display nothing. This is because by default the oxc reservation is disabled until the reservation is triggered for the first time. The reservation is triggered by setting reservation parameters. To reserve bandwidth in a CPU, three parameters should be specified: CPU number, maximum budget and period. For example:

```
#echo 0 100000/1000000 1 20000/500000 > cg/cpu.oxc_control
```

This command reserve 100ms every 1s on CPU 0 and 20ms every 500ms on CPU 1 to cgroup cg. 0 and 1 are CPU numbers. 100000 and 20000 are maximum budgets and 1000000 and 500000 are periods. In fact, behind this command, a hyper ox container with two ox containers inside is created. Tasks and task groups inside cgroup cg and its descendant will be work inside these two ox containers since now. The two containers' reservation parameters are specified by the command inputs. The unit for budget and period

value in the command is micro second. This follows the convention in *cpu* cgroup subsystem, since when you use other bandwidth control mechanisms the value is also considered in micro seconds. Tasks inside this cgroup and its further sub cgroups will run using the above reserved CPU bandwidth. Now we can say *cg* is contained in a hyper *ox* container.

Now to cat the content of the `cpu.oxc_control` interface under directory `/cgroup/cg`:

```
#cat /cgroup/cg/cpu.oxc_control
```

The reservation parameters we just set will be displayed:

```
0 100000/1000000 1 20000/500000
```

So the file `oxc_control` is an interface used for both setting up and displaying reservation parameters. Reservation parameters can be configured in the same way as they are initially set. Furthermore, there is no need to set up or configure reservation parameters for two CPUs at the same time. Suppose in some point, users decide to decrease the reservation from CPU 1, they can simply use the following command to operate on CPU 1 only.

```
#echo 1 20000/1000000 > cg/cpu.oxc_control
```

The reservation on CPU 1 is decreased to 20ms every 1s and the reservation on another CPU is not interfered.

To create a sub cgroup for cgroup *cg*:

```
#mkdir -p /cgroup/cg/cg_0
```

The cgroup *cg_0* is contained in the same hyper container as its parent. Try to cat the `cpu.oxc_control` file in this sub cgroup:

```
#cat /cgroup/cg/cg_0/cpu.oxc_control
```

An error message will be returned. This is because for a cgroup family contained in a hyper container, only the top cgroup is allowed for people to browse and modify reservation parameters.

People can move tasks to cgroup *cg* and *cg_0*. For example

```
#echo 1982 > cg/tasks
```

```
#echo 1983 > cg_0/tasks
```

This moves task with pid 1982 and 1983 to cgroup `cg` and `cg_0` respectively. Tasks can be RT tasks or normal tasks. All tasks inside an ox container behave as working on a virtual Linux system and utilize the reserved bandwidth.

The oxc tasks can move between different cgroups contained in the same hyper container. They can move between ox containers and hyper containers. They can also leave an ox container and return to be a non oxc task.

Until now, how to reserve CPU bandwidth under oxc framework is introduced. Now let's see how to distribute the reserved CPU power. Although users cannot browse reservation parameters in cgroup `cg_0`, they can indeed set reservation parameters for `cg_0`, which will trigger reserved power redistribution.

```
#echo 0 100000/2000000 1 20000/1000000 > cg/cpu.oxc_control
```

After this, a new hyper container with reservation parameters 1000000/1500000 on CPU 0 and 20000/1000000 on CPU 1 will be created and `cg_0` and its descendant cgroups will be associated with it. Now, although `cg` and `cg_0` are still in the same hierarchy through the cgroup directory observation, they are indeed contained in two different hyper containers. The ideal semantics of the above command should also include that the reserved bandwidth by `cg` need to be decreased the same value as distributed to `cg_0`. This behaviour is still missing in current prototype implementation of oxc framework. Yet this is indeed implementable. Also, the total reserved bandwidth in the system should not be more one in each CPU; this condition test is not realized either.

3.7 Cooperation with scheduling mechanisms inside Linux

First let's discuss a simple case is when both `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED` are not set; that is, task grouping is not enabled. In this case, oxc tasks share the reserved CPU bandwidth directly according to their scheduling policies. In fact, in this situation, the oxc

control can be used to repeat RT throttling and CFS bandwidth control. However, at this time, the reservation is in a real time way and is more flexible since CPU bandwidth from different CPUs can be different under oxc framework. The result of IRMOS real time scheduler can also be achieved by our framework. Consider future scheduling classes that are possible to be merged in Linux kernel. For example the `schedule_deadline` patch, which does not have task grouping scheme by itself. Not only that it can work with oxc scheduling naturally because of the ox container's "open-to-extension" feature, but also the oxc framework can utilize cgroup virtual filesystem as a way to group deadline tasks.

When the kernel compilation option `CONFIG_FAIR_GROUP_SCHED` is set, the CFS task group scheduling is enabled. Task groups inside the same hyper ox container follow the rules of CFS grouping and share the reserved CPU power. CFS group scheduling is applied in different areas independently: each hyper ox container is an area and outside all ox containers there is an area.

When `CONFIG_RT_GROUP_SCHED` is set, RT throttling is enabled. It's necessary to first analyze the possible result when RT throttling is applied in a hyper ox container. Suppose inside an container, Q/T is the bandwidth reserved from the CPU and RT throttling sets parameters as Q'/T' and $Q'/T' \leq Q/T$. The ideal behaviour for such an ox container would be : the container distributes reserved CPU power to tasks inside it; and RT tasks inside a container will be throttled when Q' units of CPU cycles are exhausted in a period T' , then non RT tasks in this container can run. However, in real world a coarse combination could cause unreliable results . For example, $Q'/T' = 1/10$ and $Q/T = 10/50$. It can happen that there are other higher priority containers in the same CPU and in one period the example container get the right to use the CPU on the last 10 units of CPU cycles in its period. RT tasks inside this container immediately take over the CPU; yet after one unit, they are throttled. So, RT tasks only run 1 unit over 50 unit of CPU cycles although 10 units are reserved. The RT throttling result inside a hyper ox container is not stable. Even if we set the period parameter in RT throttling the same as its container's, because that RT throttling and oxc

control use different timers, there is a unsynchronization between them and can lead to still complex situations.

In a statement, to apply RT throttling naively inside an ox container is not efficient. One possible solution is to count the time consumption in RT throttling using the ox container's timer, this basically means to implement a copy of RT throttling in oxc framework itself. In such a case, if some constraints are put, like the RT throttling period should equal to the container's period, we can expect predictable behaviour. Another solution is simply disable RT throttling inside an ox container, because oxc scheduling framework itself can perform the same result in a real time way as RT throttling. In current implementation, there is no communication between RT throttling and oxc control. And when `CONFIG_RT_GROUP_SCHED` is set, the result is not satisfiable.

The test between CFS bandwidth and oxc control is not conducted yet. If `CONFIG_CFS_BANDWIDTH` is set, we expect that the situation could be similar to what we see in RT throttling case.

Among cgroup subsystems, there is one cpuset subsystem also affecting scheduling behaviour in a cgroup. After cpuset cgroup subsystem is mounted in a cgroup, there is an interface `cpuset.cpus` appearing in the dorectory. This file can control which CPU the tasks inside this cgroup can use. For example,

```
#echo 1 > cpuset.cpus
```

This will result tasks inside this cgroup can only run on CPU 1. In oxc scheduling framework, we have the concept of hyper ox container, which control the CPUs that tasks inside it can run in. So, this idea is compatible with cpuset cgroup subsystem. However, until now the two mechanisms work independently; future work to bridge the two will make the system more consistent.

Chapter 4

Experiments

The overhead introduced by oxc framework includes three parts:

- The time required to execute codes brought by oxc functions.
- The context switches introduced by the oxc control.
- The degradation of modular schedulers' performance under oxc framework.

The third item is caused by the implementation limitation can be minimized or removed by improving implementation details. For example, to access the per CPU runqueue in Linux is an optimized operation. However, the access to a per ox container runqueue is not so efficient, this will give a penalty when scheduler works inside an ox container.

There are two experiments carried out to evaluate the overhead in the oxc framework. In experiment A, the code execution time of frequently invoked oxc functions is measured. In experiment B, the overall overhead of oxc control is estimated through comparisons with RT throttling and CFS bandwidth control.

The main interest in experiment A is the cost of these oxc functions that incorporate a scheduling operation and codes to regulate the bandwidth reservation. Current oxc framework implementation is still a prototype. Some kernel features are not considered under the framework yet. For instance,

the `priority inheritance` [21], which is important for the kernel’s real-time performance and will influence number of context switches. So, instead of counting and analyzing context switches directly, in experiment B the overhead of scheduling inside an `ox` container is approximately evaluated by a relative way. As for the context switches caused by importing CBS based scheduling in the kernel, people can refer to [3] for more information; yet these results do not straightly apply to `oxc` work.

The hardware and software used in the experiment are shown in table 4.1.

<i>Hardware platform</i>	
Processor	Intel(R) Core(TM) Duo E8500
Frequency	3.16GHz
RAM	4Gb
<i>Software platform</i>	
Linux distribution	Ubuntu 11.10
Compiler version	gcc 4.6.1
Kernel version	3.4.0-rc+
Benchmark tool	tbench (dbench 4.00)

Table 4.1: Hardware-Software platform

The chapter is organized like this: the tracing tool (Ftrace) and synthetic benchmark tool (tbench) we use in the experiment are described; then it’s the design and result analysis of each experiment; finally, what we learn from the experiment is concluded.

4.1 Ftrace in Linux kernel

Ftrace [22] is an internal tracer designed to help out developers of systems to find out what is going on inside the kernel. The name ftrace comes from “function tracer”, which is its original purpose. Now there are various kinds of tracers incorporated in Ftrace. You can use it to trace context switces, hong long interrupts are disabled, and so on.

Ftrace uses *debugfs* file system to hold control files as well as file to display output. Typically, ftrace is mounted at `/sys/kernel/debug`.

```
#mount -t debugfs nodev /sys/kernel/debug
```

After this command, a directory `/sys/kernel/debug/tracing` will be created containing interfaces to configure ftrace and display results.

```
#cd /sys/kernel/debug/tracing
```

The following commands will be assumed to be called under `tracing` directory. There are several kinds of tracers available in ftrace, simply cat the `available_tracers` file in the `tracing` directory. The output could vary with enabling or disabling kernel configuration options concerned with ftrace functionality in compilation time.

```
#cat available_tracers
blk function_graph mmiotrace wakeup_rt wakeup function
sched_switch nop
```

The `function` is the function tracer. It uses the `-pg` option of `gcc` to have every function in the kernel call a special function `mcount()` for tracing all kernel functions. The `function_graph` is similar to the function tracer except that the function tracer probes functions on their entry whereas the function graph tracer traces on both entry and exit of a function. It is called function graph tracer because it provides the ability to draw a graph of function calls similar to C code as tracing results. This `function_graph` is what we use in experiments. To enable the function graph tracer, just echo `function_graph` into the `current_tracer` file.

```
#echo function_graph > current_tracer
```

A trace can be started and stopped through configuring `tracing_on` file. Echo 0 into this file to disable the tracer or 1 to enable it. Cat the file will display whether the tracer is enabled or not.

The output of the trace is held in file `trace` in a human readable format. The ftrace will by default trace all functions in the kernel. In most cases, people only care about particular functions. To dynamically configure which function to trace, the `CONFIG_DYNAMIC_FTRACE` kernel option should be set in compilation time to enable dynamic ftrace. Actually, `CONFIG_DYNAMIC_FTRACE` is highly recommended and defaultly set because of its performance enhancement. To filter which function to trace

or not, two files are used: `set_ftrace_filter` for enabling the tracing of a specific function and `set_ftrace_notrace` to disable the tracing of some function. A list of available functions that you can add to these files is listed in `available_filter_functions`. In the later experiment, the `oxc` function `task_tick_oxc` is traced by setting up like this:

```
#echo task_tick_oxc > set_ftrace_filter
```

4.2 Tbench

The `tbench` [23] benchmark is a tool that measures disk throughput for simulated netbench runs. `Tbench` reads a load description file called `client.txt` that was derived from a network sniffer dump of a real netbench run and it produces only the TCP and process load and no filesystem calls. In the debian platform, the `tbench` tool is part of the `dbench` package. One example to run a `tbench` test:

```
$tbench_srv  
$tbench 2 -t 100
```

The `tbench_srv` should be invoked before running `tbench`. The second command starts two `tbench` connections with one client thread and one server thread in each connection. The two connections will run simultaneously and the runtime of the benchmark will be 100 seconds.

4.3 Experiment A

4.3.1 The experiment design

In this experiment, the execution time of `oxc` functions are measured. In a Linux system, even if the `oxc` patch is applied in the kernel, when there is no `oxc` tasks, the system performs as a plain Linux system. In such a case, the possible `oxc` overheads include the code execution time in function `is_oxc_task` and the `oxc` related initialization when a scheduling group is created; both are negligible.

During the experiment, the execution time of following oxc functions are measured using Ftrace:

- `check_preempt_curr_oxc`
- `pick_next_task_oxc`
- `put_prev_task_oxc`
- `task_tick_oxc`

They are chosen because that the most frequently invoked scheduling operations are enclosed inside them. The oxc functions `enqueue_task_rq_oxc` and `dequeue_task_rq_oxc` happen when a task enters or leaves an ox container. And they are not often called in the following experiment scheme.

There will be six individual tests differing in the number of hyper ox containers in the system. In the first test, there is only one hyper ox container; then, in each later test one more hyper ox container would be added. All hyper ox containers in the experiment are identical. Each hyper ox-container has two ox containers with the same CPU reservation parameter $0.1ms/1ms$. Each ox container has one dummy task within it. The dummy task simply runs a forever while loop and it is a RT task with policy `SCHED_FIFO`. This while loop task will exhaust the reservation in its ox container. The scheduling policy is arbitrarily chosen without special thoughts. On the contrary, the situation for modular scheduling inside an ox container is intentionally arranged as simple as possible to clarify the effect of oxc framework.

The measured execution time of above oxc functions comprises the time consumed by codes involving with the oxc control and operations defined in modular scheduler which are encapsulated inside the these functions.

4.3.2 Experiment results

The results of six tests are listed in table 4.2. The index of a test indicates the number of hyper ox containers in that test. The two fields in the pair are average value and standard deviation of the measured function execution time in micro seconds.

	test1	test2	test3	test4	test5	test6
pick_next_task_oxc	(0.168, 0.083)	(0.155, 0.070)	(0.206, 0.178)	(0.230, 0.215)	(0.211, 0.216)	(0.246, 0.251)
put_prev_task_oxc	(0.827, 0.049)	(0.834, 0.096)	(0.820, 0.103)	(0.801, 0.111)	(0.829, 0.146)	(0.852, 0.251)
task_tick_oxc	(0.272, 0.192)	(0.275, 0.182)	(0.263, 0.155)	(0.261, 0.15)	(0.245, 0.158)	(0.249, 0.146)
check_preempt_curr_oxc	-	-	-	-	-	-

Table 4.2: Measured execution time, in micro seconds, of oxc functions in the format of (*mean, standard deviation*)

The first surprise is from the row for `check_preempt_curr_oxc`. That is, no tracing result for `check_preempt_curr_oxc` is recorded. An analysis of overhead in this function is needed. The details of this function is in section 3.4.4. There are three cases when checking if an oxc task can preempt the currently running task. When the current task is not an oxc task or both two are oxc tasks and not in the same ox containers, the comparison cost is just several `if-else` instructions. Otherwise they are two oxc tasks in the same container; however, in our experiment set-up, there is only one task inside an ox container. In short words, overheads brought by this function is trivial. Although this could be the reason to explain that the `ftrace` fails to measure the execution time of `check_preempt_cur_oxc` during tests, it reminds us that it may be attractive to develop a new recording tool, given the feature of hierarchical scheduling, so as to evaluate the oxc framework more accurately and conveniently.

The test results of other three oxc functions are illustrated in figure 4.1, 4.2 and 4.3. The variable parameter in each test is the number of ox containers. And experiment results show that at least in current oxc framework, the codes execution time is influenced by the number of ox containers in the system.

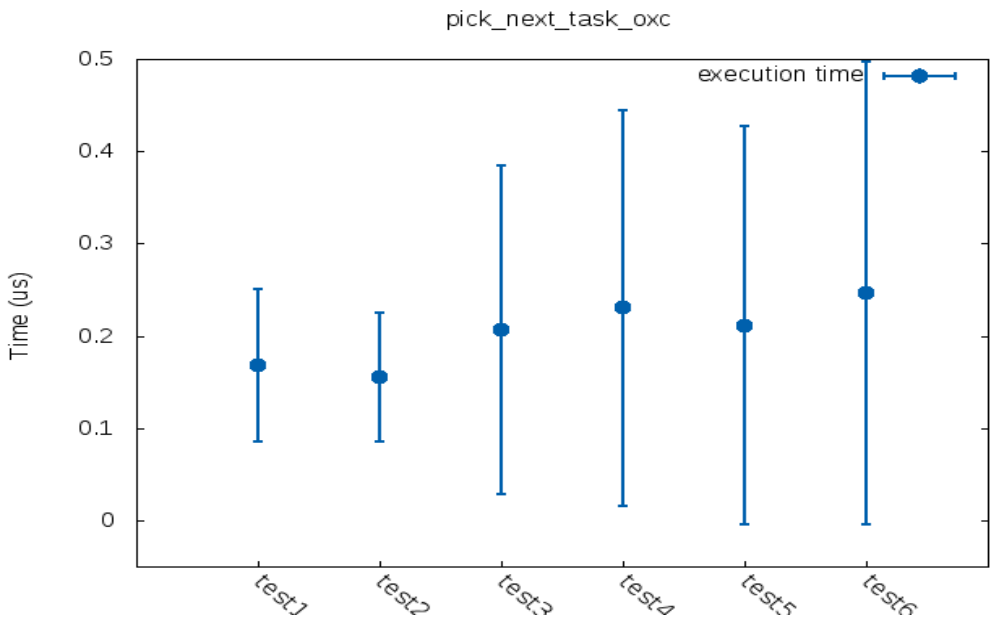


Figure 4.1: Measured execution time for pick_next_task_oxc

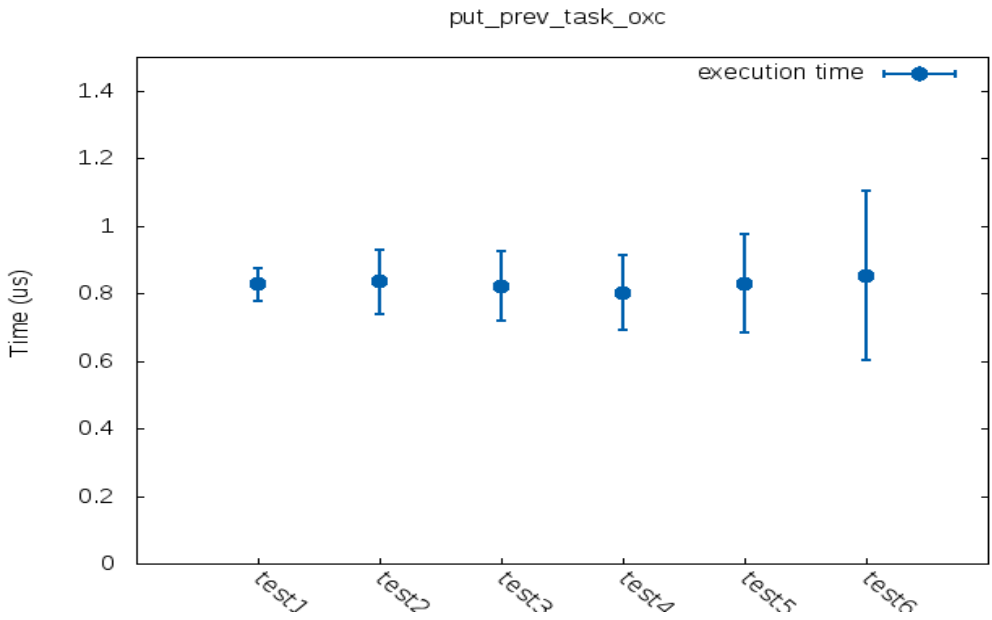


Figure 4.2: Measured execution time for put_prev_task_oxc

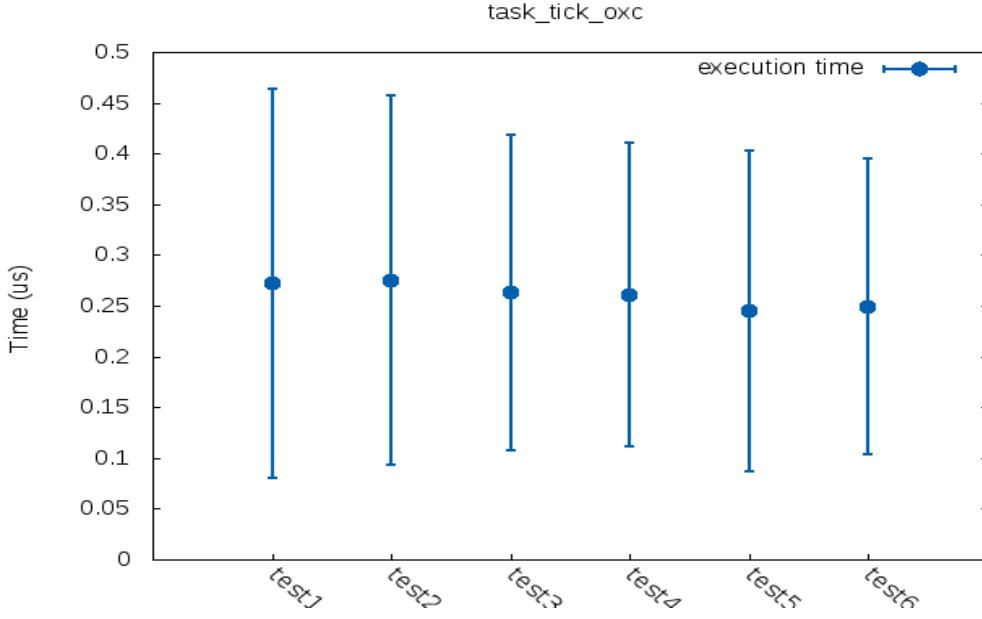


Figure 4.3: Measured execution time for `task_tick_oxc`

Figure 4.1 shows the statistical result of `pick_next_task_oxc` in each test. One observation from the figure is that with more ox containers joining the system, the time spent on executing the function codes fluctuates more. This trend also reflects in the results for `put_prev_task_oxc`, as in figure 4.2. However, figure 4.3 for `task_tick_oxc` does not show this pattern.

Now we are going to probe why the execution time of `task_tick_oxc` is more stable than the result of the other two. A look at the body of the three functions in listing 3.18, 3.21 and 3.23, we can find that except for the enclosed scheduling operations, other codes in the three functions are actually the same: to update the runqueue and the current ox container's runtime information. So, the different variance behaviour in measured execution time for oxc functions is very likely to be caused by the performance fluctuation of scheduling operations when they happen inside the oxc container.

The three encapsulated scheduling operations `pick_next_task_rt`, `put_prev_task_rt` and `task_tick_rt` are defined in RT scheduling class. Specifically, the codes of `task_tick_rt`, which is called inside oxc function `task_tick_oxc`, is listed below. This is a very simple function. If peo-

ple read the other two scheduling operations' codes in *linux/sched/rt.c*, this function is still less complex in dealing with runqueues and tasks. When scheduling operations are called inside an ox container, there will be extra cost because the raw implementation details, and such an influence may be smaller when the scheduling operation itself is simple. This explains why the result in 4.3 is stable in both mean value and standard variance.

Listing 4.1: The simple body of `task_tick_rt`

```
static void
task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
    update_curr_rt(rq);

    watchdog(rq, p);

    /*
     * RR tasks need a special form of timeslice management.
     * FIFO tasks have no timeslices.
     */
    if (p->policy != SCHED_RR)
        return;

    /*
     * The left part has no effects in our tests.
     */
    ...
}
```

4.4 Experiment B

4.4.1 Experiment design

The aim of this experiment is to estimate the overall overhead in oxc framework through comparison with non real-time CPU bandwidth control mechanisms that are already in Linux kernel. During tests the synthetic load is generated using the *tbench* benchmark tool. The CPU bandwidth allocated

to tbench connections are allocated by oxc control, RT throttling and CFS bandwidth control individually. The tbench throughput results of oxc framework are then compared with RT throttling and CFS bandwidth control. By such comparisons, the overhead introduced by oxc control is then evaluated in a relative way.

Two tbench connections will be set up in the system. Each connection will be dedicated to one CPU. Without constraints, they will consume all CPU time. In the experiment, the CPU bandwidth allocated to tbench traffic is restricted. The per CPU bandwidth parameter used in tests includes $0.05s/1s$, $0.1s/1s$, $0.2s/1s$, $0.4s/1s$, $0.6s/1s$ and $0.8s/1s$. Note that these are the per CPU bandwidth that is planned to assign to tbench tasks. Each CPU bandwidth control mechanism will restrict the tbench execution not exceed the configured value. And the throughput results will be inversly proportional to the overhead in each bandwidth control mechanism.

When RT throttling is tested, tbench clients and servers will be scheduled as RT tasks with policy `SCHED_RR`. Correspondingly, to compare with RT throttling, tbench threads inside the ox container will be set as RT tasks with `SCHED_RR` policy too. When comparing the oxc control with CFS bandwidth control, the tbench threads inside ox containers will run as normal tasks.

4.4.2 Experiment results

The throughput results are shown in table 4.3 and 4.4.

per CPU bandwidth	RT throttling	oxc control + RT scheduling
0.05s/1s	21.9335	18.5313
0.1s/1s	43.5794	36.89324
0.2s/1s	92.7356	73.5099
0.4s/1s	172.582	147.806
0.6s/1s	233	225.72
0.8s/1s	319.297	297.0607

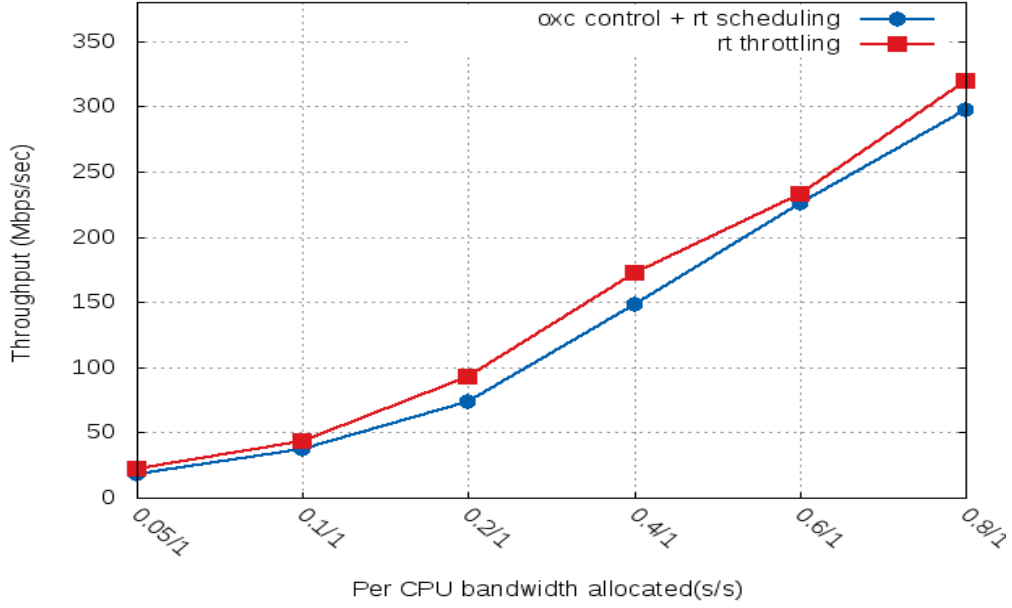
Table 4.3: Throughputs, in Mbps/sec, under RT throttling and oxc control

per CPU bandwidth	CFS bandwidth control	oxc control + CFS scheduling
0.05s/1s	24.8825	19.2151
0.1s/1s	52.2106	39.4268
0.2s/1s	106.226	77.4225
0.4s/1s	215.071	151.465
0.6s/1s	323.628	234.369
0.8s/1s	433.025	305.1

Table 4.4: Throughputs, in Mbps/sec, under CFS bandwidth and oxc control

After a glance on the two tables, it's apparent to note that the throughput results under CFS bandwidth control outperform the other two. There are two reasons for this. Firstly, the overhead brought by CFS bandwidth control is indeed lower than the other two mechanisms. Secondly, both oxc tasks and RT tasks are more greedy than CFS tasks when they occupy a CPU. For oxc tasks, both RT and normal tasks are lower priority tasks. Between RT tasks and CFS tasks, the latter one is the lower priority task. The oxc task or RT task will not give up a CPU to a lower priority task until the CPU reservation is totally consumed. So lower priority tasks in the system are stressed more under oxc control or RT throttling, especially when high reservation parameters are configured. However, in CFS scheduling, normal tasks, with or without CPU reservation, can share the CPU evenly.

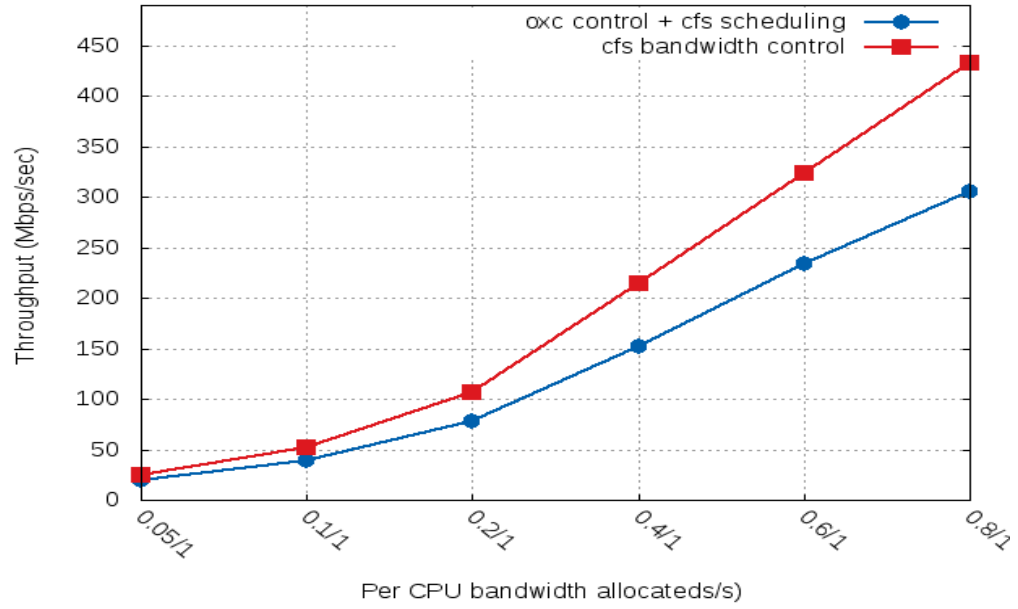
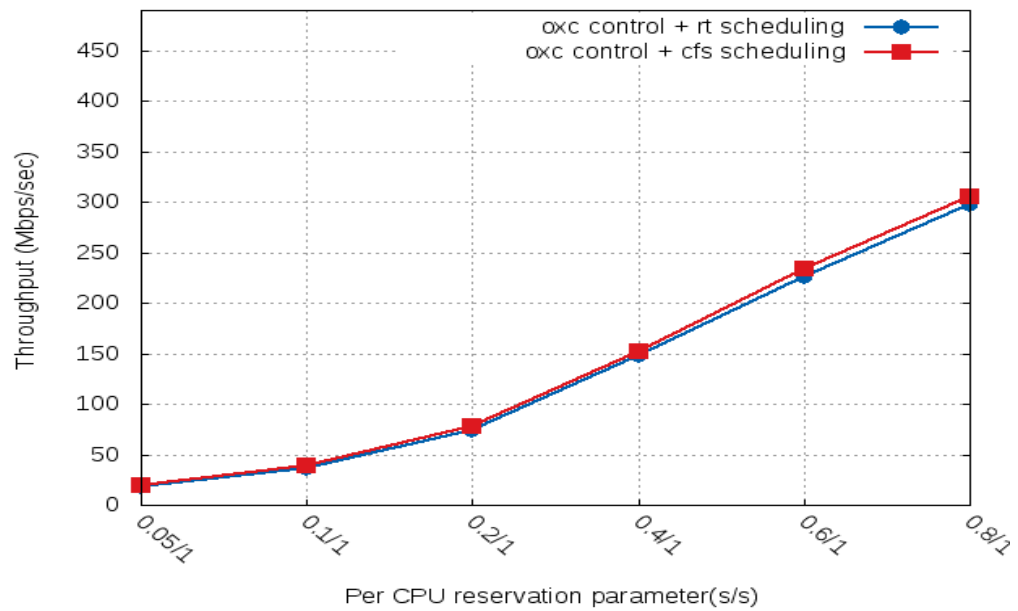
Having solved the above question, let's first study the performance of tbench tasks in oxc framework when they are scheduled with SCHED_RR policy. The comparison between oxc control and RT throttling in table 4.3 is visualized in figure 4.4. At first, the throughput result under RT throttling is higher and growing faster than the result in oxc control. However, as more CPU bandwidths are reserved to tbench tasks, the throughput results of the two are converging. In fact, the throughput growing trend in oxc control are consistent; yet in RT throttling, it's not like that. When a relatively small fraction of CPU is assigned through RT throttling and oxc control, RT throttling shows a better performance. However, with increasing the reserved CPU bandwidth, the stress of RT throttling on the whole system is rising too, which slows growth of the throughput. Under oxc control, the throughput result is almost linear with the given reserved bandwidth. The

Figure 4.4: *oxc control* vs. *RT throttling*

overhead in oxc control behaves as a constant factor. This comparison gives us another meaningful implication. When oxc control is used, allocations of bandwidth in the system should be cautioned so as to achieve an optimal system performance.

The comparison between oxc control and CFS bandwidth control is shown in figure 4.5. As we just analyzed, CFS bandwidth control has much better throughput results. One observation is that although with less raising speed, the throughput increasing trend in oxc control has the similar shape as in CFS bandwidth control.

At last, figure 4.6 mixes the statistics in table 4.3 and 4.4 and draws the throughput results under oxc control with RT throttling and CFS scheduling in the same graph. The results are quite close, and the difference can be regarded as the scheduling cost between RT and CFS scheduling in an ox container. Still, CFS scheduling shows better results than RT scheduling even under oxc framework. Assume, in the future, our oxc framework is fully developed and the degradation of modular schedulers is negligible or deterministic. And in such ideal conditions we get the result shown in figure 4.6.

Figure 4.5: *oxc control* vs. *CFS bandwidth control*Figure 4.6: *oxc control + RT throttling* vs. *oxc control + CFS scheduling*

Such a result may demonstrate that CFS scheduling introduces less overhead in the system than RT scheduler. This is indeed one possible application of oxc framework. In some cases, when comparing two schedulers, we can set up the environment inside an ox container instead of a real machine. We can even prepare a certain number of ox containers to simulate a lightweight networked testbed.

4.5 Experiment feedbacks

The experiment does not show that the performance of oxc control is better than existing bandwidth control methods. This is also not the experiment objective (the comparison result with RT throttling is a small surprise). The experiment outcome has an unstated meaning for future development of oxc framework. Experiment A gives precise measurement of oxc function execution time and confirms the necessity to improve implementation quality of the oxc framework. In experiment B, the overall performance of oxc framework is compared with RT throttling and CFS bandwidth control. Its experiment analysis shows us that how to distribute CPU bandwidth will affect both the work inside an ox container and the whole system behaviour; it also raises one possibility for oxc framework's future use. These feedbacks would be considered in the evolvement of the oxc framework.

Chapter 5

Conclusions and Future Work

In this thesis we present the design and development of the oxc scheduling framework. The oxc framework is a hierarchical multi-processor reservation mechanism that targets to manage CPU power distribution to tasks in Linux without constraints on the scheduling details.

Alongside the design and development of the framework, we bring some new ideas into the world. We conclude the four golden rules that may be useful in developing and evaluating a CPU bandwidth management mechanism for a GPOS. We formalize the relationship among different components in Linux scheduling by inventing the concept scheduling route, which will assist people for studying and probing the Linux kernel scheduling. We propose the ox container structure in Linux and the oxc scheduling theory.

The theoretical background of the oxc framework is the oxc scheduling, which may have a more profound influence besides our framework. In the world of oxc scheduling, tasks are managed by a scheduler around the ox container (a virtual CPU). An independent scheduling system can be constructed above the virtual CPU and its behaviour is the same as a Linux per CPU scheduling system. And the theory of oxc scheduling can find its use in any case when an independent scheduling is needed. A big feature of the oxc scheduling is that it is naturally compatible with Linux schedulers.

Although current implementation of the oxc framework is still a prototype, it shows great potential for an opportunity to be seriously applied in real

world. For the first time, there is work in Linux that can manage CPU power distribution for all types of tasks. It can reserve bandwidths for RT tasks and CFS tasks in Linux, which demonstrates the theory of oxc scheduling. Under the framework, CFS task group scheduling is support. Naively work on multi-processor support and hierarchical reservation distributing are included in the framework and demonstrate that under the oxc framework such functionalities are achievable.

During experiments, the overhead of oxc functions is measured and confirms that there is the necessity to refine the implementation of the oxc scheduling control. In another experiment, the performance of the oxc framework is not as good as existing work; however, it also shows the feasibility of the framework in Linux.

In order for the oxc framework to really attract interests from both academic and industrial fields. There are four issues that must be sloved first.

- The CPU bandwidth allocation in the oxc framework now is made in a uncontrolled way depending on the wise decision from users. A admission control for bandwidth distribution should be added.
- Strategies for task migrations among ox containers using the same reservation, which means ox containers in the same hyper ox container, should be added.
- Temporarily, the hierarchical distribution of reserved CPU bandwidth is implemented in a faked way. That is, the bandwidth distributed to sub groups is a new reservation from the system instead of inheriting from the parent's reservation. The connection between parent's and children's reservations needs to be added.
- Until now, the oxc framework does not have a consideration for resource sharing among ox containers. If two tasks accessing the same critical section are in the same ox container, the Linux has its way to schedule the two tasks. However, if the two are in different ox containers or one is oxc tasks and the other one is not, it may happen that an earliest deadline ox container is blocked. Now the oxc framework has no work

on this problem. For the similiar reason, synchronization between tasks from different ox containers is a dangerous operation.

Besides the above four items related to the survival of the oxc work, there are some less critical issues that are better to be done.

Under oxc framework, the performance of RT throttling is not satisfiable. The relationship between oxc framework and `cpuset` cgroup subsystem is not well studied. According to the oxc scheduling theory, a scheduler's influential scale can be flexibly controlled. Now in oxc framework, a scheduling class works without restraints. It is greatly meaningful to explore the oxc control together with other work like AQuoSA, RT-Xen, RESCH, etc.

Acknowledgments

After six months' efforts, I am concluding this thesis, which is the most devoted work I have ever done. Great appreciation to my thesis supervisor, Giuseppe Lipari. It's amazing to work with you. Your involvement keeps delicate balance that the problems I meet (in thesis and life) can be solved and my potential for independent and creative thinking is highly motivated. There are too many stupid, unrelated and weird opinions that I have told you. However, I could always get positive and beneficial feedbacks. Thanks to my supervisor Luigi Palopoli in Trento. At the early days after I came to Italy, I was confused. You helped me be confident. I enjoyed the last two years' time. I met these professors, colleagues and friends that make my life special. I am not going to list every name. I sincerely thank you all. Dear Liang, because of you, I am the luckiest person in the world. Baba mama, I love you!

Bibliography

- [1] L. Abeni and G. C. Buttazzo, “Integrating multimedia applications in hard real-time systems,” in *IEEE Real-Time Systems Symposium*, pp. 4–13, 1998.
- [2] “Rtai - real time application interface.” <https://www.rtai.org/>.
- [3] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, “Aquosa - adaptive quality of service architecture,” *Softw., Pract. Exper.*, vol. 39, no. 1, pp. 1–31, 2009.
- [4] D. Faggioli, M. Trimarchi, F. Checconi, and S. Claudio, “An edf scheduling class for the linux kernel,” in *Proceedings of the 11th Real-Time Workshop (RTLW)*, October 2009.
- [5] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, “Hierarchical multiprocessor cpu reservations for linux kernel,” in *5th OSPERT Workshop*, July 2009.
- [6] Y. I. S. Kato, R. Rajkumar, “A loadable real-time scheduler suite for multicore platforms,” December 2009.
- [7] S. Xi, J. Wilson, C. Lu, and C. D. Gill, “Rt-xen: towards real-time hypervisor scheduling in xen,” in *EMSOFT*, pp. 39–48, 2011.
- [8] S. K. Baruah, J. Goossens, and G. Lipari, “Implementing constant-bandwidth servers upon multiprocessor platforms,” in *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pp. 154–163, IEEE, 2002.

- [9] G. Lipari and S. Baruah, “A hierarchical extension to the constant bandwidth server framework,” in *rtas*, p. 0026, Published by the IEEE Computer Society, 2001.
- [10] G. Lipari and E. Bini, “A methodology for designing hierarchical scheduling systems,” *Journal of Embedded Computing*, vol. 1, no. 2, pp. 257–269, 2005.
- [11] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti, “A hierarchical framework for component-based real-time systems,” *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 116, pp. 253–266, 2005.
- [12] A. Mancina, D. Faggioli, G. Lipari, J. N. Herder, B. Gras, and A. S. Tanenbaum, “Enhancing a dependable multiserver operating system with temporal protection via resource reservations,” *Real-Time Systems*, vol. 43, no. 2, pp. 177–210, 2009.
- [13] W. Mauerer, *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., 2008.
- [14] R. Love, *Linux Kernel Development (3rd Edition)*. Addison-Wesley, 2010.
- [15] “Red-black trees (rbtree) in linux.” Documentation/scheduler/rb-tree.txt.
- [16] “Real-time group scheduling.” Documentation/scheduler/sched-rt-group.txt.
- [17] “Cgroups.” Documentation/cgroups/cgroups.txt.
- [18] “Cpusets.” Documentation/cgroups/cpusets.txt.
- [19] “Cfs bandwidth control.” Documentation/scheduler/sched-bwc.txt.
- [20] J. Lelli, G. Lipari, and G. Anastasi, “Design and development of deadline based scheduling mechanisms for multiprocessor systems,” Master’s thesis, University of Pisa, 2010.

- [21] “Rt-mutex subsystem with pi support.” Documentation/rt-mutex.txt.
- [22] “ftrace - function tracer.” Documentation/trace/ftrace.txt.
- [23] “tbench(1) - linux man page.” <http://linux.die.net/man/1/tbench>.