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

Youcheng Sun

August 22, 2012

Abstract

Attentions have been being paid to extend general purpose operating systems with real time functionalities. In this thesis, a scheduling framework, which can control Central Processing Unit (CPU) bandwidth distribution on multi-processor platforms in a real-time way, is proposed in Linux. 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 in Linux, 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. Experimental results are not available now ...

Contents

Introduction

Linux is the most widely deployed open source general purpose operating system. Traditionally, the aim of Linux scheduling is to distribute cpu cycles fairly among tasks and task groups according to their relative importances. Because of the popularity and compatibility, Linux based systems are used to serve various kinds of purposes. In some cases, fairness is not enough or necessary.

There is the class of *real time* applications that have soft or hard requirements on the cpu cycles they receive, normally in some time length, in order to work well. Such timing guarantee (or real time guarantee) cannot be provided just by fairness. Multi-media applications belong to this class. Indeed, simply from cpu bandwidth management's point of view, a real time gurantee is also useful. In many situations, people prefer to distributing cpu 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 from the 10% and assign it to a subset of the tasks.

Nowadays, multi-core architecture is successfully utilized to boost computing capability for computing devices. It's no surprise to see an embedded device with more than one CPU inside. Computing systems with multiple processors are becoming mainstream. Yet high speed processors and more cores do bring more challenges instead of solving the CPU timing guarantee problem. Also, no matter how powerful the platform is, there is always time when people need more. So, to fully utilize multi-core platform is also a good reason to manage computing power in a real time way.

In mainline Linux kernel, there are two so called real time 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 gurantee 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.

There is work that extends Linux with real time capabilities to fullfill the timing guarantee requirement: RTAI[1], AQuoSA[2], `sched-deadline` patch[3], IRMOS real time framework, and RESCH and so on. Instead of modifying the system directly, RT-Xen tries to apply real time mechanisms in the hypervisor level(Xen). Each work has its emphasis. Initially, our work is motivated by IRMOS. Our framework has two features:

- It can predictively distribute cpu cycles above 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.

In Linux, there is a scheduling system on each CPU. Different such per CPU scheduling systems construct the system level scheduling by task migration mechanisms among different CPUs. For the first time in Linux, our framework provides with the opportunity to build extra scheduling systems besides these destinated with each CPU. We call the framework Open-Extension Container(OXC) scheduling framework. Open-Extension (OX) container is a new data structure in Linux that is raised by our work. It is the fundamental element in the framework. Based on ox container structure, the concept "per oxc scheduling system", whose behaviour is the same as "per CPU scheduling system" in Linux, is given. Several per oxc scheduling systems coorperate and work as the "pseudo (Linux) system level scheduling".

In oxc scheduling framework, each ox-container can reserve an amount of bandwidth from a CPU through CBS rules [4]. The per oxc scheduling

system based on it utilize this computing power to shcedule tasks as if working on a less powerful cpu. This is how oxc framework distribute reserved CPU cycles to tasks under it. Because the per oxc scheduling has the same behaviour as per CPU scheduling in Linux, general types of tasks can run using the reserved bandwidth. On multiple processor platforms, different OXCs can inpdependently reserve bandwidths from a subset of total CPUs and scheduling systems above them work together to imitate the behaviour of the Linux system level scheduling. The basic unit to apply a scheduling policy under OXC framework is an OX container.

Chapter 1

Background

1.1 The Constant Bandwidth Server theory

A Constant Bandwidth Server(CBS) is characterized by a budget c_s and an pair (Q_s, T_s), where Q_s is the maximum budget and T_s is the period of the server. $U_s = Q_s/T_s$ is called the server bandwidth. Such a server can be utilized to serve a set of tasks, which can be tasks with soft, hard or non real time guarantee requirements. CBS theory defines rules to reserve bandwidth on a single processor. In our work, we use a hard version of CBS.

For a specific server S, at any instant, a fixed deadline $d_{s,k}$ is associated with the server. A CBS is said to be active at time t if there are pending tasks and c_s is not 0; otherwise it is called idle. At any time, among all active servers, the one with earliest deadline is chosen. Then a served task of this server is picked to execute. CBS does not restrict the rule to pick up a particular task. For example, first in first out (fifo), rate monotonic scheduling and any user defined rule can be used. As the picked task executes, the server budget c_s is decreased by the same amount. When budget c_s reaches 0, the server become inactive. At each deadline point, the c_s will be recharged to Q_s and a new server deadline will be generated as $d_{s,k+1} = d_{s,k} + T_k$. Initially, $c_s = Q_s$ and $d_{s,0} = 0$. When a task arrives at time t and the server is idle, if $c_s \geq (d_{s,k} - t)U_s$, the server updates its deadline as $d_{s,k+1} = t + T_s$ and c_s is recharged to maximum value Q_s.

Given a set of servers $\{S_0, S_1, ..., S_n\}$, if

$$\sum_{i=0}^{n} U_i \leq 1$$

then, every T_i time units, a server S_i can obtain Q_s time units to serve its tasks. In other words, U_i is the bandwidth a server S_i reserves from a cpu.

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. As for how each scheduling algorithm in Linux is implemented, it's neither the interest of this section or oxc framework. 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 "wolfgang""R.Love".

1.2.1 Scheduling classes

Linux scheduling system adapts a modular design, and the basic modularity is a scheduling class, which is an instance of `struct sched_class`. Scheduling algorithms are implemented as scheduling classes and a scheduling class is a modular scheduler(or simply called a scheduler). All modular schedulers composes the generic scheduler in Linux. The `struct sched_class` defines a set of interfaces which need to be realized in order to implement a scheduler in Linux. These methods are all scheduling operations a scheduler in Linux can perform. Each scheduler fullfills details behind the interface and carries out its specific scheduling behaviour. There are three scheduling classes in mainline Linux: rt_sched_class, cfs_sched_class and idle_sched_class. Each scheduling class is responsible for scheduling a type of tasks. Tasks scheduled `cfs_sched_class` are called normal tasks. Tasks scheduled by

rt_sched_class are called rt tasks. idle_sched_class deals with special idles tasks which does 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.

Listing 1.1: Shceduling operations for a scheduler

```
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. 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 occupting 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 is the first scheduling operation for a task after a task is chosen to occupy the CPU.
- `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.

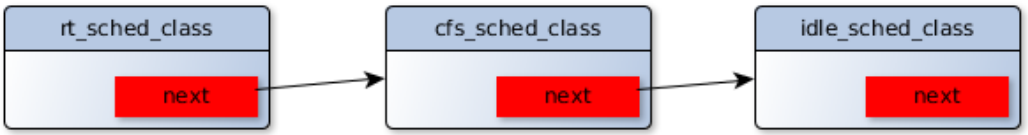


Figure 1.1: Scheduling classes in Linux

The basic scheduling unit in Linux is scheduling entity, which can represent both tasks and task groups. There are two kinds of scheduling entities: `struct sched_entity` for cfs scheduling class and `struct sched_rt_entity` for rt scheduling class. To differentiate, here we call them cfs (scheduling) entity and rt (scheduling) entity respectively.

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 methods like `sched_setscheduler`.

When `CONFIG_FAIR_GROUP_SCHED` is set, cfs task grouping is enabled. And `CONFIG_RT_GROUP_SCHEED` is the kernel configuration for rt task group scheduling.

```
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`, which is called run queue in Linux. We can say that Linux scheduling is runqueue centered. In Linux, the `struct rq` is a per CPU data structure; each cpu is associated with a runqueue. Although the name indicates, `struct rq` is not a queue. Let's have a look at the inside of a runqueue structure:

```
struct rq {
  ...
  unsigned long nr_running;
  struct cfs_rq cfs;
  struct rt_rq rt;
  struct task_struct *curr, *idle;
  u64 clock;
```

```
#ifdef CONFIG_SMP
  int cpu;
#endif
  ...
};
```

- `nr_running` specifies the number of runnable tasks 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. For example, `struct cfs rq` and `struct rt_rq`.
- `curr` points to the task currently running.
- `idle` points to a special idle task when no other tasks are runnable.
- `clock` is updated by `update_rq_clock` method.
- `cpu` tells the CPU of this runqueue.

1.2.3 Completely Fair scheduler

Completely fair scheduler 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`[1]).

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 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 tasks belonging to it. The scheduling entity handled by cfs scheduling class is `struct sched_entity`. Here instead of studying the details of CFS, we are going to see how different

[1]This SCHED_IDLE policy is not related to idle_sched_class which aims to handle a special idle task.

scheduling components (sched_entity, task_struct, task_group and struct cfs_rq) are related.

```
struct cfs_rq {
        unsigned long nr_running;
        u64 min_vruntime;
        struct rb_root tasks_timeline;
#ifdef CONFIG_FAIR_GROUP_SCHED
        struct rq *rq;
  struct task_group *tg;
#endif
  ...
};
```

- nr_running is the number of tasks in this cfs runqueue.
- min_vruntime tracks the minimum virtual runtime of all tasks associated with this cfs runqueue.
- tasks_timeline is the root of the red-black tree where all tasks associated with this cfs runqueue is stored. Tasks are ordered by their virtual run time value.
- rq is the per CPU runqueue this cfs runqueue is embedded in.
- tg is the task group that owns this cfs runqueue.

```
struct sched_entity {
  ...
  struct cfs_rq *cfs_rq;
#ifdef CONFIG_FAIR_GROUP_SCHED
  struct cfs_rq *my_q;
#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 schedulign entity can also represent a task group.

When cfs task group scheduling is enabled. In this case, 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 defaultly; tasks in this group are enqueued in the cfs runqueue embedded in the per CPU runqueue directly.

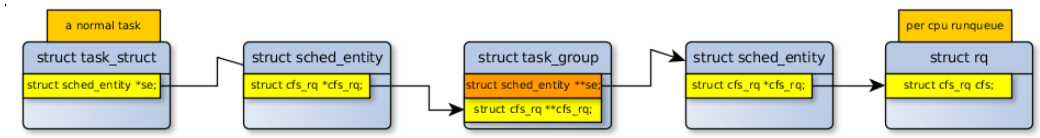


Figure 1.2: CFS scheduling when cfs group scheduling is enabled.

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 tasks when rt task group scheduling is not enabled. 1.3.

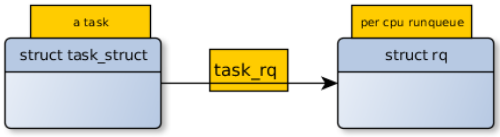


Figure 1.3: Scheduling scheme without group scheduling

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 figure1.1, rt tasks are always schedueld 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 algrithm. When a `SCHED_RR` task exhausts its timeslice, another `SCHED_RR` task of the same priprity 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.

```
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 are kept in a linked list headed by *active.queue*[*prio*]. If there is a task in the liset, the corresponding bit in *active.bitmap* is set.

The connections among `struct sched_rt_entity` and other scheduling components are similar to the struct sched_entity case.

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

When `CONFIG_RT_GROUP_SCHED` is set, figure1.4 shows the scheduling scheme of rt tasks. If rt task group scheduling is not enabled, still *task_rq* marco will be used.

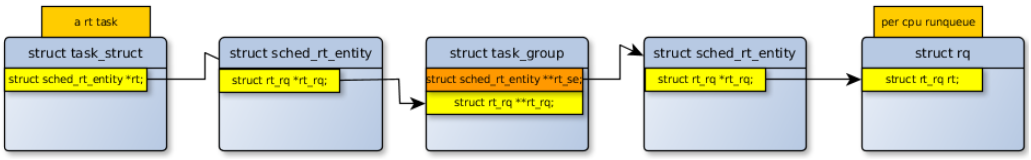


Figure 1.4: RT scheduling when rt group scheduling is enabled.

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. It uses the *control group* (cgroup) virtual file system. Each cgroup associates a set of tasks with a set of resources, called *subsystems*. For example *cpuset* subsystem 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 choosen 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 thc 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.

1.3.2 CFS bandwidth control

Basically, CFS bandwidth control is the same technique as RT throttling applying on normal tasks. It is a `CONFIG_FAIR_GROUP_SCHED` extension which allows the specification of the maximum CPU bandwidth available to normal tasks in a cgroup or cgroup hierarchy. The bandwidth allowed to a cgroup is specified using a quota(`cpu.cfs_quota_us`) and a period(`cpu.cfs_period_us`). By specifying this, 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 1.3.1, 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 an a feedback-based control mechanism. The scheduler uses CBS rules to rserve CPU bandwidth for a task, which is a rt task with SCHED_RR policy in its Linux implementation. Given the error between the reserved computation and the amount of CPU cycles really consumed, the feedback controller adapts CBS reservation paramters 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 Schedule-deadline patch

The schedule-deadline patch for Linux kernel is being developed to extend current mainline Linux with a deadline-based scheduling method. In schedule-deadline, a new scheduling class (scheduler) is implemented and has highest priority among all scheduling classes. Tasks scheduled by this scheduling class are called sched tasks. A sched task is assigned deadlines according to CBS rules and scheduled in "earliest deadline first (EDF)" way.

1.3.5 IRMOS real-time framework

The ragued name IRMOS comes from the European project "Interactive Real-time Multimedia Applications on Service Oriented Infrastructures". The IRMOS framework replace rt throttling mechanism in mainline Linux with real time CPU reservation(still CBS), and reuses the existing interfaces. So, users configure the cgroup interface as what we saw in rt throttling(1.3.1), the difference is that this time the CPU bandwidth is allocated in a guaranteed way. Also, new cgroup interfaces are added to assist reserved CPU power distribution in the cgroup hierarchy.

Chapter 2

Design and development

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 platform, tasks can migrate between different runqueues, also depending on behaviours defined in specific scheduler. In other words, on each CPU, a scheduling system is built up bsed on the associated runqueue. Different per CPU scheduling systems cooperate with each other by task migrations defiend by specific scheduling class and construct the system level scheduling. The idea is that if there is one extra runqueue, each scheduler can still use it as scheduling parameter and a scheduling system can be built around it. If there are more than one extra runqueues, they can produce a pseudo system level scheduling system.

Here a data structure named Open-Extension Container(OXC) is proposed, shown in figure2.1. The ox container is designed as an abstract data structure; that is, any data structure contains a `struct rq` runqueue inside can be called an ox container. Now, in the system there is not only per CPU runqueues, but also per oxc runqueues.

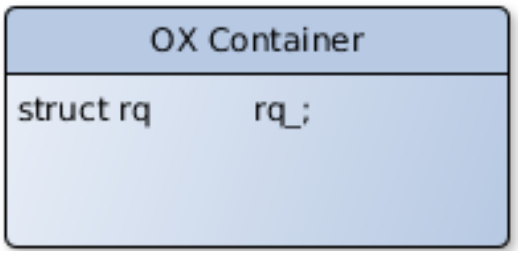


Figure 2.1: Open-Extension Container

2.2 The oxc scheduling

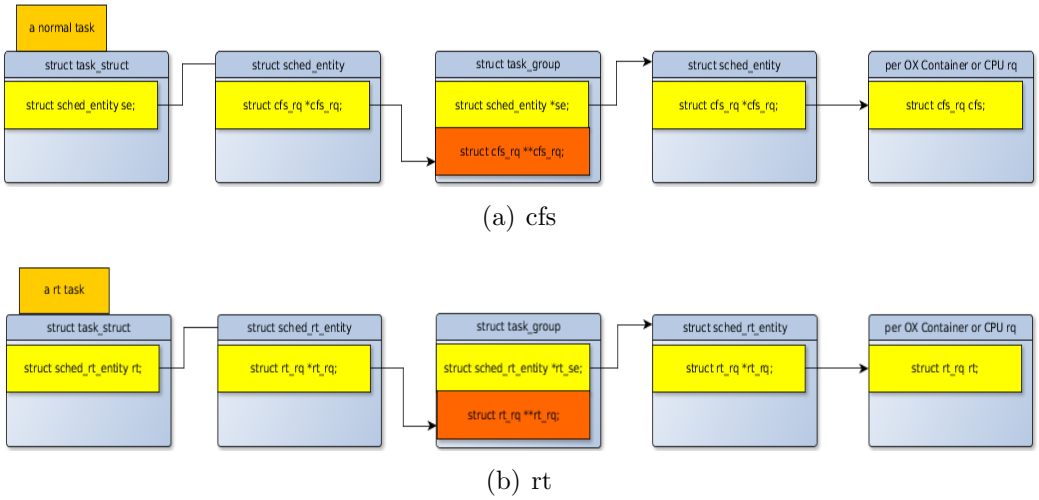
As ox containers are defined, there are extra runqueues besides per CPU ones in a system. For each scheduler, they manage tasks above runqueues according to implementation details in their scheduling class, and as long as a runqueue parameter is provided for their schedulinf operations, they have do not care it is from a CPU or 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 container runqueue. As for tasks and scheduling classes, there is no difference between a per CPU and per container runqueue. This can be clearly shown when we see the scheduling scheme in the system with oxc scheduling. Recall the path which relates each task with the per CPU runqueue it runs above in section 1.2.3 and 1.2.4. Now let's see the scheduling scheme after the ox container joins the system with its local runqueue.

From figure 2.2(a) and 2.2(b) we can see when task group scheduling is enabled, the scheduling route we saw before can also lead a task to its per container runqueue. The same codes can still be used to find both kinds of runqueues.

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))
```

The macro returns the associated 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 route to lead a task to a runqueue



(a) cfs

(b) rt

Figure 2.2: Scheduling route with task group scheduling in OXC enabled Linux

is needed, shown in figure 2.3. Actually, this path already exists in Linux kernel. Just because in current Linux, there is no runqueue other than per CPU ones and people ignore to exploit it.

The first feature of oxc scheduling is that it is compatibe with Linux original scheduling. Tasks dealt by rt scheduler or cfs can naturally work under the oxc scheduling system. When there is new scheduling algorithm implemented in Linux kernel, like the *sched deadline* patch we mentioned before, the new scheduler has to fullfill details behind scheduling interfaces in `struct sched_class`. Again, for each shceduling class, they do not care the interface is passed a per CPU or per container runqueue as the parameter, and the new scheduling class can also work under oxc scheduling. So, the ox container structure is open to extension; this is where the name is from.

Based on per CPU scheduling, each scheduling operation defined in `struct sched_class` will affect all tasks in the CPU. The oxc scheduling provides another opportunity to apply a scheduling class in a fine grained scale. Now, the scale to apply a scheduler can be controlled in the unit of an ox container.

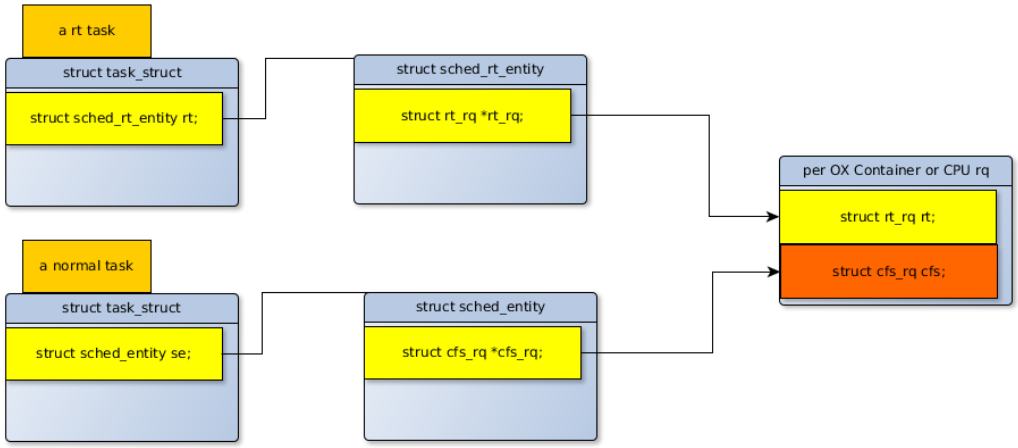


Figure 2.3: New scheduling route for tasks when task group scheduling is not enabled

2.3 Movtivation 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 shceudled by some policies. Such control can be real-time or non real-time. One obervation here is that there does not exist a mechanism that can control CPU bandwidth for all kinds of tasks as a whole. Such a general control mechanism is the initial consideration of OXC framework.

Suppose there is an ox container, different schedulers can use it to enqueue, operate, and dequeue its tasks. If a fraction of CPU bandwidth can be assigned to this oxc, all kinds of tasks will use it as running on a less powerful machine. This is the oxc solution to realize CPU bandwidth control as a whole.

2.4 Development of OXC scheduling framework

The oxc framework is still ongoing. Stable codes can be found in github[1]. The oxc framework is not a scheduler, it only interests in CPU bandwidth control

[1]https://github.com/YIYAYIYAYOUCHENG/linux

to an oxc, which depends on modular schedulers to use it for scheduling tasks.

2.4.1 Implementation of ox container structure

An ox container `struct oxc_rq` is implemented in Linux kernel. An `struct oxc_rq` can reserve bandwidth from a CPU using CBS rules. The CPU reservation for oxc runqueue follows the implementation of CBS reservation for a rt runqueue in IRMOS framework. In the following context, an `struct oxc_rq` is also called ox container, container, and oxc runqueue for the same meaning. An oxc runqueue also correspond to a constant bandwidth server in CBS theory.

Listing 2.1: `struct oxc_rq`

```
struct oxc_rq {
  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 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.
- `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 points. If at some point, the value of `oxc_time` is larger than the value of `oxc_runtime`, then `oxc_throttled` should be set until `oxc_period_timer` fires to recharge the container.
- `oxc_runtime_lock` guarantee that the timing information of the oxc is updated in a consistant way.
- `rq_` is the local runqueue of the ox container.
- `rq` points to a per CPU runqueue and this CPU is where the ox container reserves bandwidth from.
- `rb_node` is used to put an ox quneueu in a red black tree. All oxc runqueues reserve bandwidth from the same CPU is put in a red black tree associated with this CPU. In this tree, an ox container's `oxc_deadline` value is used to order nodes. runqueue

All oxc runqueues that reserve bandwidth from the same CPU are put in the same red black tree, which orders its nodes according to the ox container's `oxc_deadline` value. The one with latest deadline is stored in the leftmost node.

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

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

An ox container is responsible for reserving bandwidth from a CPU. Th `struct hyper_oxc_rq` is defined to reserve bandwidth from multiple CPUs.

```
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 ox containers to reserve bandwidth from CPUs specified in `cpus_allowed`.

2.4.2 Extensions on original data structures

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 2.2: Extensions in `struct rq`

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

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 help locate the ox container's edf tree in a CPU.

In current implementation of oxc framework, reservation is made for tasks in a control group. Tasks in a cgroup is represented in the `struct task_group` structure. So extensions also happen inside it.

Listing 2.3: Extensions in `struct task_group`

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

Because tasks in a group can span multiple CPUs, the hyper container is used to reserve bandwidth for it. If a task group runs above a hyper oxc, its `hyper_oxc_rq` points to the hyper container; otherwise, this field is NULL.

When you associate a hyper container to a task group, its non oxc decendant task groups will also be associated with this hyper container, the `oxc_label` field is used to specify this kind of sub task groups.

If a task runs in a ox container, it is called an oxc task. In section 2.2, we show how to direct a task to the runqueue it runs above, which may be a per CPU or container one. A further check of the runqueue's `in_oxc` field can tell us whether a task is an oxc task or not. Consider that such "is that an oxc task?" is often used in the framework, a `is_oxc_task` field is added in `struct task_struct` for efficient reason.

Listing 2.4: `is_oxc_task` field in `struct task_struct`

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

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

2.4.3 The point to direct a task to a per ox container runqueue

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 associate a task to a runqueue in mainline Linux, where there is only per CPU runqueue concepts.

Listing 2.5: To associate a task with a runqueue in original 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
}
```

When task group scheduling is enabled, each task is directed to its task group. This correspond to the first part of scheduling route. The second part of the scheduling route from task group to per CPU runqueue is done when the task group is created. Figure **??** shows the hint how a task group joins the scheduling route during its creation. In case that task group scheduling is

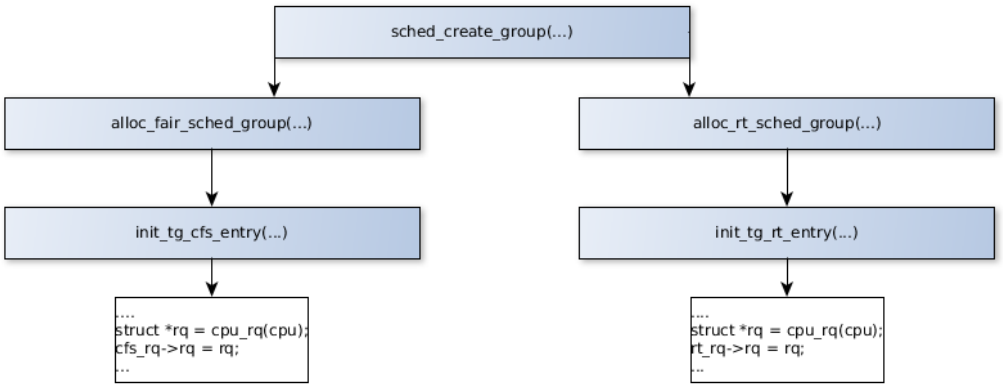


Figure 2.4: The creation of a task group in original Linux

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

Now, we have seen the point a task joins the scheduling path in original Linux. It's time to see the scenario after per oxc runqueue is imported in Linux. Still, `set_task_rq` is the point to associate a task with a runqueue, per CPU or per oxc now.

Listing 2.6: To associate a task with a runqueue under oxc framework

```
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 (task_rq_oxc(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 in cases of original Linux and oxc enabled Linux. The interesting part happens when task group scheduling is not set. This time, given the task group is associated with a hyper oxc or not, the task is drected to a per CPU or oxc runqueue. This corresponds the scheduling route in figure 2.3. In the end of the method, `is_oxc_task` is configured. `task_rq_oxc` tracks the shceduling route to find the runqueue.

In oxc enabled Linux, a task group can be associated with a hyper oxc and the runqueues it contains in three cases:1. when it is created, its parent is associated with a hyper oxc 2. the task group is explicitly attached to a hyper oxc 3. one its acendant task group is attached to a hyper oxc.

Listing 2.7: OXC scheduling related initilization during task group creation

```
int alloc_oxc_sched_group(struct task_group *tg, struct
    task_group *parent)
```

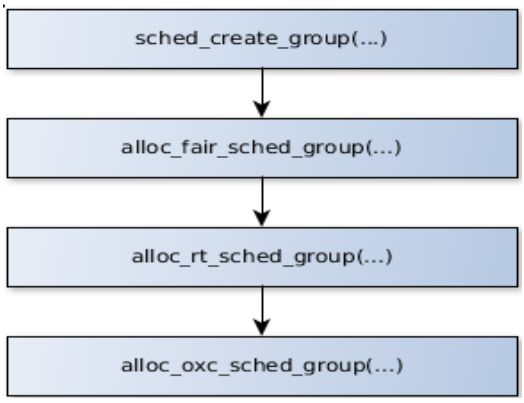


Figure 2.5: The creation of a task group in oxc enabled Linux

```
{
        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;
}
```

`alloc_oxc_sched_group` deals with 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, it directs its scheduling path to the per oxc runqueue. And the `oxc_label` for such child oxc task group is 100.

A task group can direct to a OXC local runqueue explicitly through `init_tg_cfs_entry_oxc` and `init_tg_rt_entry_oxc`. Let's have a look at `init_tg_cfs_entry_oxc` as an example.

Listing 2.8: To explicitlt direct a task group 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;
}
```

Brief explaination on the parameters:

- `tg` is the task group to be dealt with.
- `cfs_rq` is the cfs runqueue the cfs entity of this task group is enqueued.
- `se` is the cfs entity that represents `tg`.
- `cpu` specifies the cfs_rq pointer inside `tg` that will be redirected.
- `parent` points to the parent cfs scheduling entity.

- `oxc_rq` contains the destinated runqueue.

As for codes, `rq_of_oxc_rq` first returns the OXC local runqueue. `init_tg_cfs_entry` initialize CFS related work for `tg`. Then, `tg` directs its local `cfs_rq` for containing tasks on `cpu` to the per OXC runqueue just got. From now on, cfs tasks and task groups under `tg` work in the scheduling route which will lead them to an OXC local runqueue. By directing a hierarchy of task group to an OXC, there is a top group, which will be enqueud in the per OXC runqueue's embedded cfs runqueue directly.

Now we see how to direct a task group to a per OXC runqueue when task group scheduling is enabeld. In the whole procedure, tasks under this group is untouched.

2.4.4 Run tasks under OXC scheduling framework

As long as per OXC runqueue joins the scheduling route, the scheduling of tasks is compatible with modular schedulers in Linux. For an OXC task, just pass the task itself and its OXC local runqueue, instead of per CPU runqueue, to its corresponding scheduling class. And the scheduler will behave as usual.

Because we consider reserving CPU bandwidth for an OXC runqueue, necessary operations are performed before relaying a task to its modular scheduler. Still scheduling details are not the framework's interest.

In order to fullfill real time guarantee, oxc tasks are always privileged to non oxc tasks. Amg 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.

To obtain the runqueue of a task

A method `tas_rq_oxc` is used to obtain the runqueue of a task. The runqueue retuned can be oxc local runqueue or per CPU one depending that whether the task is inside a container.

```
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;
}
```

For any task, it has both cfs scheduling entity and rt entity. So the rt and cfs scheduling routes both exists for a task. Here, we utilize the cfs scheduling route. Given `CONFIG_FAIR_GROUP_SCHED` is set or not, the corresponding scheduling routes are explored to track the runqueue.

To enqueue an oxc task

When an oxc task arrives, besides enqueue it in the oxc local runqueue, the ox container information may be updated if necesary.

```
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);
}
```

`oxc_rq_of_task` tracks the scheduling route and returns the ox container of an oxc task. Then local runqueue's time information is updated. Although

the ox container does not care about scheduling details of tasks inside it, tasks are enqueued in its local runqueue and may rely the runqueue's time information. Then we see all scheduling details are dealt by a task's scheduling class as the `enqueue_task` of the scheduling class is called with the task and local runqueue. The `inc_oxc_tasks` method is simple.

```
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. The coming of an oxc task in a container may change the relation among ox containers in the same edf tree. This is the work of `enqueue_oxc_rq` method.

```
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_ONs. If the oxc_rq is already on edf tree or moved from the tree because of exhausting budget, nothing to be done. There are two conditions for an ox container outside an edf tree: it is throttled or it is empty. 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 what oxc_rq_update_deadline does. When it comes to CPU reservation, an ox container corresponds to a constant bandwidth server in CBS theory. texttt_enqueue_oxc_rq is quite a mechanical procesure to put an oxc runqueue in an edf tree.

To dequeue an oxc task

dequeue_task_oxc is the opposite method of enqueue_oxc_rq.

```
static void dequeue_task_oxc(struct rq *rq, struct task_struct *
    p, int flags)
{
        struct rq *rq_ = task_rq_oxc(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 structure of dequeue_task_oxc are the same as enqueue_task_oxc: local runqueue's time information is updated, local runqueue and task is relayed to the corresponding scheduler, and the task number is decreased.

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.

```
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 more tasks.
                    */
                __dequeue_oxc_rq(oxc_rq);
                return;
        }
}
```

The comments are explainable enough. __dequeue_oxc_rq removes a oxc runqueue from the edf tree.

To check the preemption

When a task wakes up from sleeping or is created, the scheduler will check if it can preempt current 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 interest in the case when the waking task is 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 wakeing task respectively. If task `p`'s container is throttled, it cannot preempt currently running task. Otherwise, if `curr` is not an oxc task, `p` has higher priprity and preempt `curr`. When two tasks 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 given; 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 containers. Now, `oxc_rq_before` checks if a contaienr's deadline is before another's deadline. The one with smaller deadline will run.

To pick up an oxc task

When to pick a most eligible task to run, oxc tasks should be checked first. If there is no eligible oxc tasks, then non oxc tasks are considered. `pick_next_task_oxc` is responsible for choosing the most eligible oxc task in a CPU.

```
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;
}
```

There are one thing to note here: 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` picks the ox runqueue with latest deadline in a CPU. And along the scheduling class chain, each scheduler uses its own way trying to find the most eligible task in the local runqueue. Another import method here is `update_curr_oxc_rq`. The budget comsumption really happens here.

```
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 information of an oxc runqueue. If the budget in current period is exhausted, the current task needs to be resched. the local spinlock oxc_runtime_lock protects the update of runtime from interleave.

```
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 are checked, which is easy to understand. The last *else* statement deals with the case tha tthe container is throttled: the `oxc_throttled` label is set, the oxc runqueue is removed from the edf tree and the timer is set and will fire at the next deadline to recharge the budget.

put_prev_task operation

The scheduling operation 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 being preempted. If currently running task is an oxc task, when `put_prev_task` is called, this is also a point to update the ox container information.

```
static void put_prev_task_oxc(struct rq* rq, struct task_struct
    *p)
{
           struct rq *rq_ = task_rq_oxc(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.

set_curr_task operation

The above put_prev_task is the last scheduling operation before a task gives up a CPU (of course, if it is chosen again immediately, it can still occupy the CPU). The set_curr_task is the first scheduling operation a task performs when it is chosen to use the CPU. If the newly chosen task is an oxc task, it is also the point to prepare the ox container for time updating.

```
static void set_curr_task_oxc(struct rq *rq)
{
        struct task_struct *curr = rq->curr;
        struct rq *rq_ = task_rq_oxc(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 is that inside this encapsulation, the per CPU runqueue parameter is passed to the task's corresponding scheduling class. This is because set_curr_task operation updates the infomation in the scheduling route except for the runqueue itself. So, there is no difference to pass which runqueue. The real reason is that, there is a possible insconsistent state. Initially, the current task in ox container local runqueue is NULL[2], which is not a special idle task from scheduling class sched_idle. Because set_curr_task is called even before the task is enqueued in the runqueue(

[2]This will be fixed in future work

maybe oxc local one). And this will cause problems.

task_tick operation

The `task_tick` operation is the most frequently called scheduling operation and is used to update the task's timing information. So, if the task parameter in this method is oxc type, this is the point to update the confainer's information and local runqueue's clock.

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

        update_curr_oxc_rq(rq);
        update_rq_clock(rq_);

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

2.4.5 SMP support in oxc framework

An ox container based scheduling system on a CPU has no difference with the per CPU based scheduling system. Each ox container works independently and they are arranged in one hyper container. For a hyper container, tasks are partitioned to each ox container and task migration or load balancing between different ox contaienrs in a hyper container are not realized now.

2.4.6 User interfaces provided with OXC framework

Currently, oxc framework provides users with cgroup interfaces. The CPU reservation functionality is realized through *cpu* cgroup subsystem. This *cpu* cgroup subsystem is for CPU bandwidth control in Linux. RT throttling, fairness group scheduling and CFS bandwidth control are all realized through it. The following describes how to use the OXC framework and the CPU number is assumed to be 2.

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 are one new file `cpu.oxc_control` which is the interface to control CPU bandwidth in oxcframework. 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 initially until the reservation is triggered for the first time. The reservation is triggerred 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. The unit for budget and period value is millisecond. This follows the convention in *cpu* cgroup subsystem. So 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.

To "cat" the content of the `cpu.oxc_control` interface under directory `/cgroup/cg`:

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

The above parameters set will be displayed:

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

Reservation parameters can be configured in the same way. Furthermore, there is no need to 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.

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

The reservation on CPU 1 is dereased 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 /cgroup/cg/cg_0/cpu.oxc_control:

```
#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 has the right to browse and modify reservation parameters.

People can move tasks to cgroup cg and textttcg_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. Note here we use "ox container", not "hyper ox container", because in temporary iplementation, tasks inside a hyper ox container are partitioned into each CPU.

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 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 decendant cgroups will be associated with it. Now, although cg and cg_0

are still in the same hierarchy in the cgroup directory observation, they are indeeded contained in two different hyper containers. The ideal effct of the above command should also include that the reserved bandwidth by `cg` need to decrease the same vaue as distributed to `tg_0`. This behaviour is still missed in current prorotype 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 realised either.

2.4.7 Cooperation with scheduling mechanisms inside Linux

When `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED` are not set; that is, task grouping is not enabled. Tasks work in an ox container just as the oxc scheduling theory explains. In fact, in this case, oxc framework can work as RT throttling and CFS bandwidth control in a real time behaviour. 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 deadline tasks in `schedule_deadline` patch, they do not have task group scheme. The oxc framework provide them a real time way to group their tasks and because of the "open to extension" feature for ox container, to merge them with oxc framework is natural.

When fairness task group shceduling is enabled, task groups under the same hyper ox container follow the rules of fairness task grouping and share the reserved CPU power. Fairness task group scheduling is applied in different areas independently: each hyper ox container is an area, outside ox containers there is the other area. Inside one hyper ox container, bandwidths are reserved and task groups inside this hyper ox container share the reserved computation power arrording to the fairness task group scheduling rules.

When `CONFIG_RT_GROUP_SCHED` is set, RT throttling is enabled. Let's 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 group will be throttled when $Q^{'}$ units of CPU cycles are exhausted in period $T^{'}$, then non rt tasks can run. However, there are also other possibilities. For example, $Q^{'}/T^{'} = 1/10$ and $Q/T = 10/50$. Suppose 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 thsi container immediately, yet after one unit, they are throttled and has to give up. So, during the whole period, rt tasks only run 1 unit over 50 unit of CPU cycles. The RT throttling result inside a hyper ox container is not stable. Even we set the period parameter in RT throttling the same as its container's, because of RT throttling using different *hrtimer* from oxc reservation, the unsyhcronization between can also cause even more complex situation.

In a word, to merge RT throttling directly inside container is not efficient. One possible solution is to count the time consumption in rt throttling using the same timer, this basically means to implement a copy of RT throttling in oxc framework itself. In such case, if some constraints are put, like the RT throttling period should equal to its container's period, we can expect predictable behaviour. Another solution is simply disable RT throttling inside container, because oxc scheduling framework itself can perform the same result in a real time way as RT throttling. In current implementation, solution one is used when `CONFIG_RT_GROUP_SCHED` is set, and result is not satisfiable.

The behaviour when CFS bandwidth control is enabled. We predict the behaviour should be similiar to what we see in RT throttling case.

Among cgroup subsystems, there is one `cpuset` also effecting scheduling behaviour in a cgroup. After `cpuset` cgroup subsystem is mounted in a cgroup, there is an interface `cpuset.cpus` appearring in the dorectory. Which 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 which CPUs tasks inside hyper container can run. So, this idea is compatible with `cpuset` cgroup subsystem. However, until now the two work independently; future work to bridge the two will make the system more efficient.

Chapter 3

Experimental Results

The overhead introduced by oxc framework composes two parts:

- The time required to execute the code of framework functions.
- The context switches introduced by the oxc framework.

Current oxc framework implementation is still a prototype one. Some kernel features are not considered under the framework yet. For example, the `priority inheritance`, which is important for the kernel's real time performance and will influence number of context switches. However, the execution time of oxc function codes is fixed and can be measured. As for the context switches caused by importng CBS based scheduling in the kernel, there is previous work .... This may help readers understand what happens under the oxc framework. In the rest of this chapter, our experiment will concentrate on measuring execution time of oxc functions.

3.1 Ftrace in Linux kernel

Ftrace 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 and the reason it is used here. 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 firectory `/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` dorectory.

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

The `function` is 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 and measure execution time of them. This is what we need. To enable the function tracer, just *echo* `function` into the `current_tracer` file.

```
#echo function > current_tracer
```

The 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 displat whether the tracer is enabled or not.

The output of the trace in 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 recommanded and defaultly set because of its performance enhancement. To filter which function to trace or not, two files are used, one for enabling and one for disabling the tracing of specific functions. They are `set_ftrace_filter` and `set_ftrace_notrace`. A list of available functions that you can add to these files is listed in `available_filter_functions`.

3.2 Experiment set up

Hardware platform	
Processor	AMD
Frequency	1666MHz
RAM size	512 Mb

Table 3.1: Hardware-Software platform

Chapter 4

Conclusions and Future Work

Bibliography

[1] "Rtai - real time application interface." https://www.rtai.org/.

[2] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "Aquosa - adaptive quality of service architecture," 2009.

[3] D. Faggioli, M. Trimarchi, F. Checconi, and S. Claudio, "An edf scheduling class for the linux kernel," October 2009.

[4] L. Abeni and G. C. Buttazzo, "Integrating multimedia applications in hard real-time systems," 1998.