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

Youcheng Sun

Graduate Program in Computer Science and Engineering

September 14, 2012

Outline

Outline

Four golden rules

- Hierarchical CPU bandwidth distribution Tasks in a Linux system can be organized in a flat or hierarchy. In modern systems, the hierarchyical way is preferred. A good mechanism should be able to distribute the CPU bandwidths hierarchically.
- Serve all kinds of tasks In Linux, there are different types of tasks that scheduled by different schedulers. A good mechanism should be able to serve all kinds of tasks.
- Support multi-processor platforms
- Provide real-time guaruntee Linux is used to serve various scenarios, among which some have temporal requirements. A good mechanism should be able to provide real-time guarantee.

Before our work, no existing mechanisms can satisfy all four golden rules

- Previous work includes RT throttling, CFS bandwdith control, AQuoSA, IRMOS real-time framework, etc.
- One common failure is rule 2.

Before our work, no existing mechanisms can satisfy all four golden rules

- Previous work includes RT throttling, CFS bandwdith control, AQuoSA, IRMOS real-time framework, etc.
- One common failure is rule 2. **Each previous mechanism can only work for a specific type of tasks in Linux.**

Before our work, no existing mechanisms can satisfy all four golden rules

- Previous work includes RT throttling, CFS bandwdith control, AQuoSA, IRMOS real-time framework, etc.
- One common failure is rule 2. **Each previous mechanism can only work for a specific type of tasks in Linux.**

In our work, a framework that satisfies all golden rules is proposed.

Outline

Resource reservation

- The Constant Bandwidth Server (CBS) algorithm is used to reserve CPU bandwidth in the work.
- A CBS is characterized by an ordered pair (Q, P), where Q is called maximum budget and P is period.
- A CBS can reserve Q every P time units from a CPU.

Linux scheduling

Linux adapts modular scheduling design. Each scheduler is an instance of the `struct sched_class`.

```
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 (*yield_task) (struct rq *rq);
        bool (*yield_to_task) (struct rq *rq, struct task_struct *p, bool preempt);

        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);
        ...
};
```

- These hooks are all scheduling operations for a scheduler.

Linux scheduling

Linux adapts modular scheduling design. Each scheduler is an instance of the `struct sched_class`.

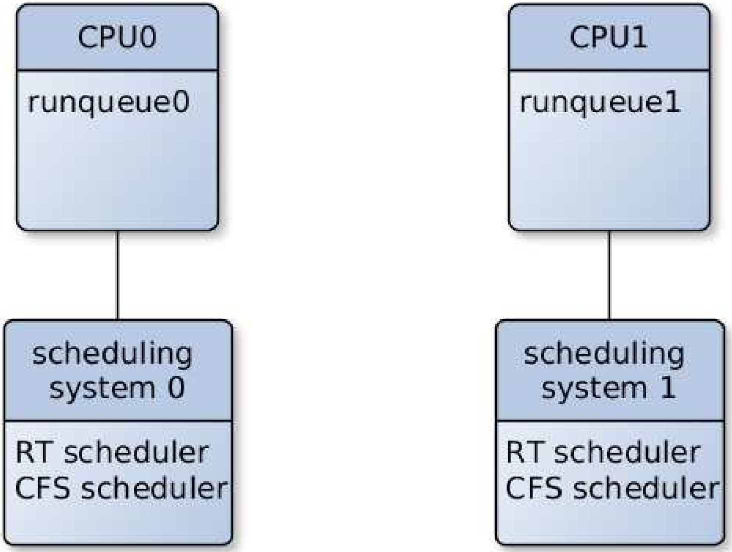
```
struct sched_class {
        const struct sched_class *next;

        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 (*yield_task) ( struct rq *rq );
        bool (*yield_to_task) ( struct rq *rq , struct task_struct *p, bool preempt);

        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);
        ...
};
```

- These hooks are all scheduling operations for a scheduler.

The Linux scheduling is runqueue centered

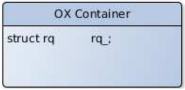
- Every hook in a scheduling class deals with the `struct rq`. A `struct rq` instance is called runqueue in Linux.
- The `struct rq` is a per CPU structure. That is, there is a runqueue per CPU in Linux.
- A scheduling system is built around the runqueue in each CPU. Different per CPU scheduling systems cooperate with each other through task migration strategy defined by schedulers.

The scheme of per CPU scheduling in Linux



Outline

The Open Extension Container (OXC) structure

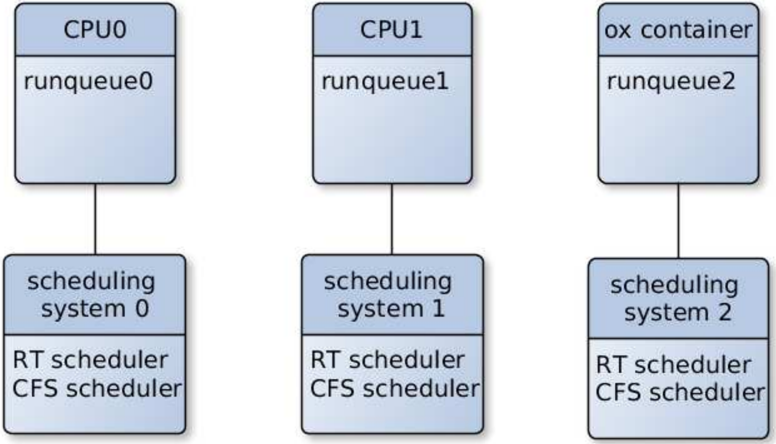


- The idea of ox container is simple: **any data structure that contains a runqueue inside is regarded as the ox container**

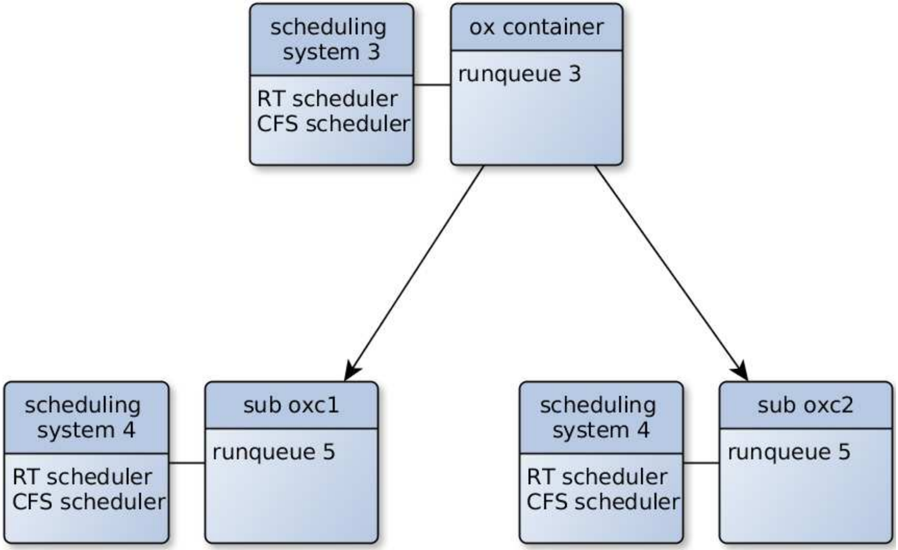
Pass an ox container's local runqueue to a scheduling operation

- For a scheduler, it needs a runqueue so as to deal with its tasks. This runqueue can be the per CPU runqueue or an ox container's local runqueue.
- When an ox container's local runqueue is used by a scheduler, tasks will run in this local runqueue.
- Now, besides per CPU scheduling system, there is the **per ox container scheduling system** in Linux.

Per oxc scheduling + per CPU scheduling coexist in Linux



The oxc scheduling can be structured in a hierarchy



To allocate CPU bandwidth for an ox container

- Different schedulers can use an ox container to enqueue, operate and dequeue their tasks.
- Suppose a fraction of CPU bandiwdth is allocated to an ox container, all kinds of tasks inside the container will use it as if running on a less powerful CPU.

To allocate CPU bandwidth for an ox container

- Different schedulers can use an ox container to enqueue, operate and dequeue their tasks.
- Suppose a fraction of CPU bandiwdth is allocated to an ox container, all kinds of tasks inside the container will use it as if running on a less powerful CPU.

Based on this idea, the **oxc (scheduling) framework** is developed to control CPU bandwidth allocated to tasks, **which can be any type**, in a real-time way.

A concrete ox container that can reserve CPU bandwidth

- This an ox container that can reserve bandwidth from a CPU according to CBS rules.

```
struct oxc_rq {
        unsigned long oxc_nr_running;
        int oxc_throttled;
        u64 oxc_runtime;
        ktime_t oxc_period;
        u64 oxc_deadline;
        u64 oxc_time;
        u64 oxc_start_time;

        struct hrtimer oxc_period_timer;
        raw_spinlock_t oxc_runtime_lock;
        bool oxc_needs_resync;

        struct rq rq_;
        struct rq *rq;
        struct rb_node rb_node;
};
```

`oxc_runtime` and `oxc_period` are the maximum budget and period parameters in CBS.

A concrete ox container that can reserve CPU bandwidth

- This an ox container that can reserve bandwidth from a CPU according to CBS rules.

```
struct oxc_rq {
        unsigned long oxc_nr_running;
        int oxc_throttled;
        u64 oxc_runtime;
        ktime_t oxc_period;
        u64 oxc_deadline;
        u64 oxc_time;
        u64 oxc_start_time;

        struct hrtimer oxc_period_timer;
        raw_spinlock_t oxc_runtime_lock;
        bool oxc_needs_resync;

        struct rq rq_;
        struct rq *rq;
        struct rb_node rb_node;
};
```

`oxc_runtime` and `oxc_period` are the maximum budget and period parameters in CBS.

the local runqueue of an ox container

SMP support

- An ox container can only reserve CPU bandwidth from a CPU.
- The hyper ox container structure is defined for reserving bandwidths from multiple processors.

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

`cpus_allowed` specifies the CPU that bandwidths are reserved from.

SMP support

- An ox container can only reserve CPU bandwidth from a CPU.
- The hyper ox container structure is defined for reserving bandwidths from multiple processors.

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

`cpus_allowed` specifies the CPU that bandwidths are reserved from.

The ox containers to reserve bandwidths from corresponding CPUs

Outline

Experiment set-ups

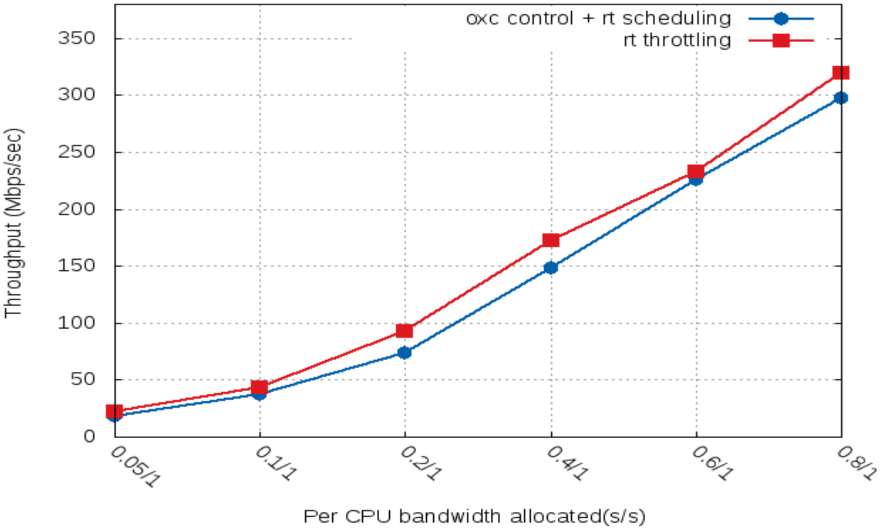
Candidates :

- RT throttling : CPU bandwidth control work, without real-time guarantee, for RT tasks.
- CFS bandwidth control : CPU bandwidth control work, without real-time guarantee, for CFS tasks.
- oxc control : Our CPU bandwidth control work, with real-time guarantee, for any type of tasks.

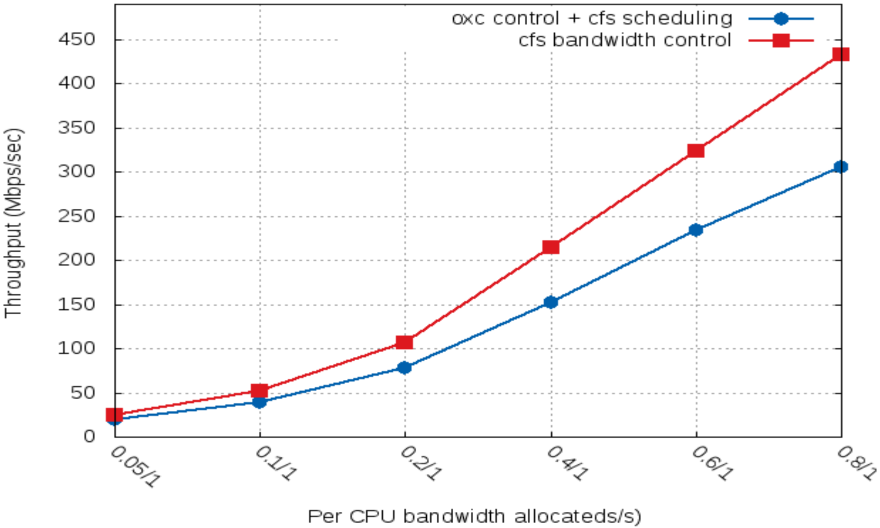
Experiment rules:

- Two tbench connections will be set up in the system; each connection is dedicated to one CPU in a dual processor platform.
- The CPU bandwidth allocated to tbench traffic is restricted using the above work individually.

Results: oxc control vs. RT throttling



Results: oxc control vs. CFS bandwidth control



Results

- Given that our work is still a prototype, such results are very encouraging.

Results

- Given that our work is still a prototype, such results are very encouraging.
- In addition: we can provide real-time gurantee ;-)

Results

- Given that our work is still a prototype, such results are very encouraging.
- In addition: we can provide real-time gurantee ;-)

What I am really trying to sell :
Our one-for-all solution for CPU bandwidth control in Linux is feasible!!!

Outline

Conclusion

- In mainline Linux there are two schedulers.
- Numerous schedulers have been or will be implemented based on Linux for various reasons.
- There exists CPU bandwidth control work dealing with some of these schedulers. But each such mechanism can only work with for one scheduler (one specific type of tasks).

Conclusion

- In mainline Linux there are two schedulers.
- Numerous schedulers have been or will be implemented based on Linux for various reasons.
- There exists CPU bandwidth control work dealing with some of these schedulers. But each such mechanism can only work with for one scheduler (one specific type of tasks).
- **The oxc framework can provide real-time CPU bandwidth control for tasks of all these schedulers!**
- The project is just started; more work is coming.