Completely Fair Scheduler in Coffer

Seungho Jang

Computer Architecture and Systems Lab, 2023 Summer

1 Red Black Tree

1.1 Red Black Tree

A red black tree is a binary search tree in which each node has a color (red or black) associated with it (in addition to its key and left and right children) and the following 3 properties hold:

1. (root property) The root of the red black tree is black
2. (red property) The children of a red node are black
3. (black property) For each node with at least one null child, the number of black nodes on the path from the root to the null child is the same

1.2 Time Complexity

Function	Amortized	Worst Case
Search	O(logn)	O(logn)
Insert	O(1)	O(logn)
Delete	O(1)	O(logn)

2 Completely Fair Scheduler

2.1 Completely Fair Scheduler

Completely fair scheduler is a process scheduler implemented based on per CPU run queues, whose nodes are time ordered schedulable entities that are kept sorted by red black trees.

2.2 Algorithm

Each per CPU run queue sorts schedulable entity in a time ordered fashion into a red black tree, where the leftmost node is occupied by the entity that has received the least slice of execution time. The nodes are indexed by processor execution time in nanoseconds.

A maximum execution time is also calculated for each process to represent the time the process would have expected to run on an ideal processor. This is the time the process has been waiting to run, divided by the total number of process.

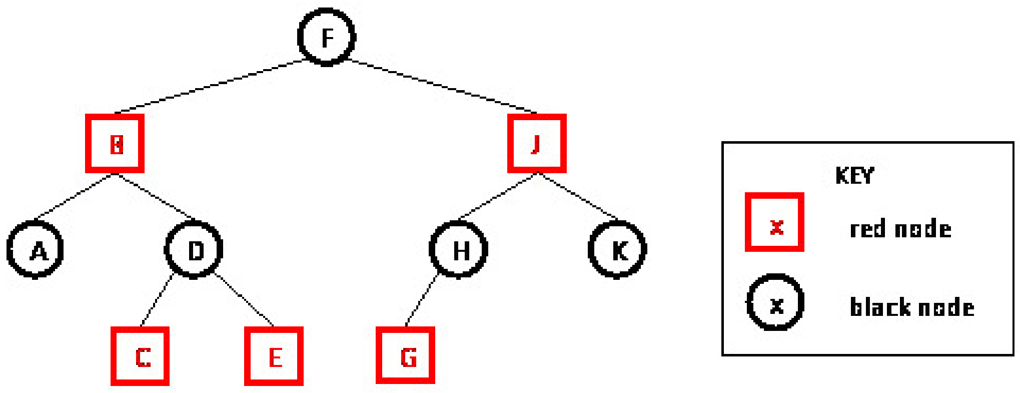


Figure 1: An example of a red-black tree

When the scheduler is invoked to run a new process:

1. The leftmost node of the scheduling tree is chosen (as it will have the lowest spent execution time), and sent for execution.
2. If the process simply completes execution, it is removed from the system and scheduling tree.
3. If the process reaches its maximum execution time or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its newly spent execution time.
4. The new leftmost node will then be selected from the tree, repeating the iteration.

3 Linux

3.1 An overview of CFS

The main idea behind the CFS is to maintain balance (fairness) in providing processor time to tasks. To determine the balance, the CFS maintains the amount of time provided to a given task in what's called the virtual runtime.

Rather than maintain the tasks in a run queue, as has been done in prior Linux schedulers, the CFS maintains a time ordered red black tree.

With tasks (represented by sched_entity objects) stored in the time ordered red black tree, tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the left side of the tree, and tasks with the least need of the processor (highest virtual runtimes) are stored toward the right side of the tree.

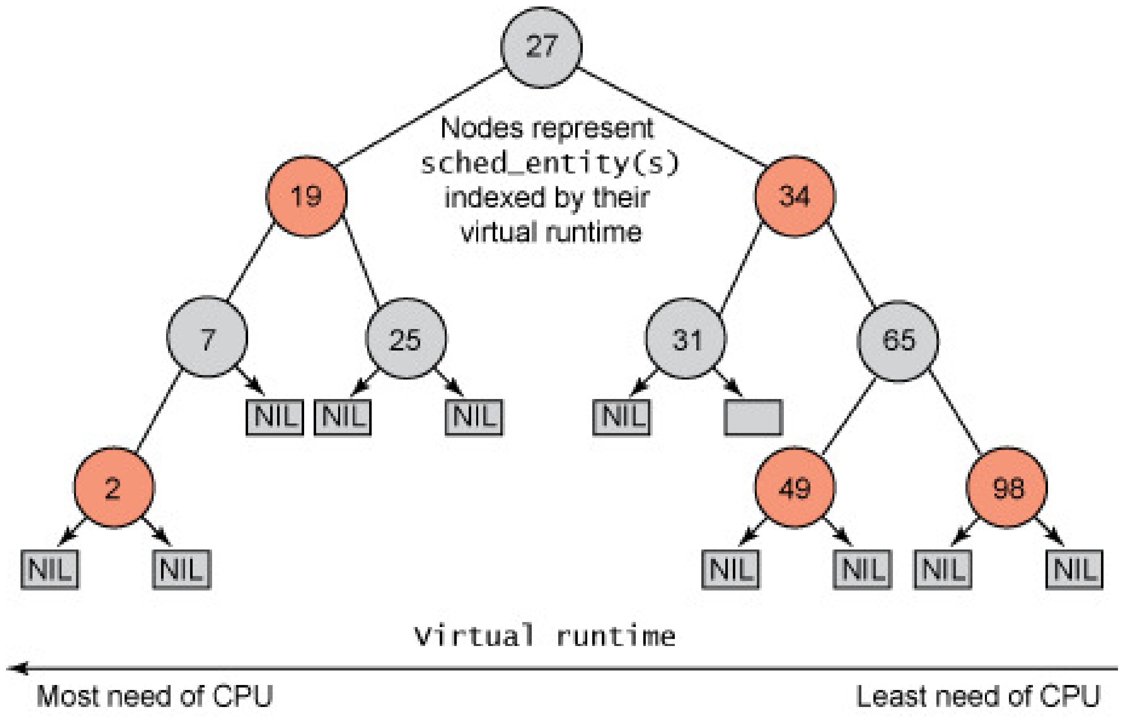
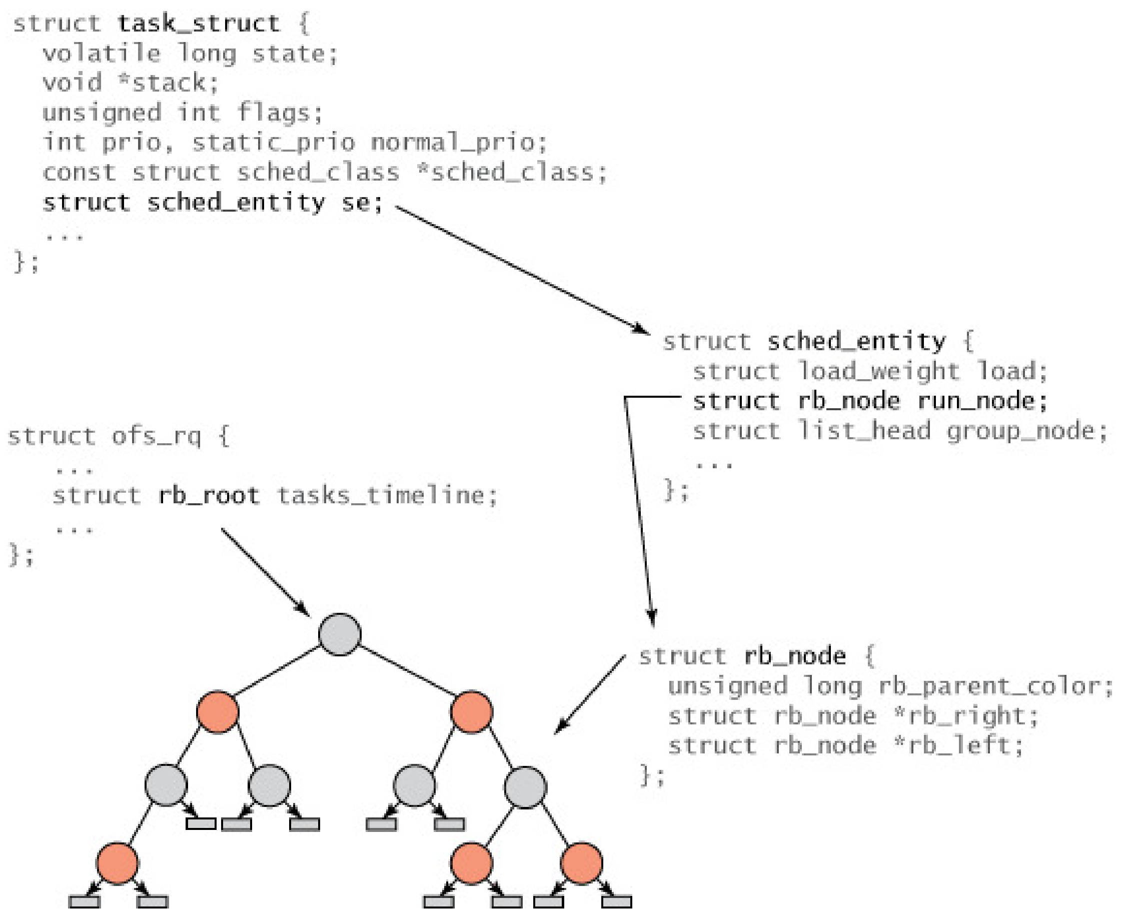


Figure 2: Example of a red-black tree

The scheduler then, to be fair, picks te left most node of the red black tree to schedule next to maintain fairness. The task accounts for its time with the CPU by adding its execution time with the CPU by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable.

3.2 CFS internals

All tasks within Linux are represented by a task structure called task_struct. This structure (along with others associated with it) fully describes the task and includes the task's current state, its stack, process flags, priority (both static and dynamic), and much more. But because not all tasks are runnable, you won't find any CFS related fields in task_struct. Instead, a new structure called sched_entity was created to track scheduling information.



```
struct task_struct {
  volatile long state;
  void *stack;
  unsigned int flags;
  int prio, static_prio normal_prio;
  const struct sched_class *sched_class;
  struct sched_entity se;
  ...
};
```

```
struct sched_entity {
  struct load_weight load;
  struct rb_node run_node;
  struct list_head group_node;
  ...
};
```

```
struct ofs_rq {
   ...
   struct rb_root tasks_timeline;
   ...
};
```

```
struct rb_node {
  unsigned long rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};
```

Figure 3: Structure hierarchy for tasks and the red-black tree

The root of the tree is referenced via the rb_root element from the cfs_rq structrue. Each node in the red black tree is represented by an rb_node, which ontains nothing more than the child references and the color of the parent. The rb_node is contained within the sched_entity structure, which includes the vruntime, which indicates the amount of time the task has run and seres as the index for the red black tree. Finally the task_struct sits at the top, which fully describes the task and includes the sched_entity structure.

schedule() function in ./kernel/sched.c preempts the currently running task (unless it preempts itself with yield()). Note that CFS has no real notion of time slices for preemption, becuase the preemption time is variable. put_prev_task() function returns the currently running task (now preempted) to the red black tree. pick_next_task() function simply picks the left most task from the red black tree and returns the associated sched_entity. With this reference, a simple call to task_of() identifies the task_struct reference returned.

3.3 CFS Scheduler

Ideal multi tasking CPU is a CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at 1/nr_running speed. On real hardware, we can run only a single task at once, so we have to introduce the concept of virtual runtime. The virtual runtime of a task is actual runtime normalized to the total number of running tasks.

In CFS the virtual runtime is expressed and tracked via the per task vruntime (nanosec unit) value. This way, it's possible to accurately timestamp and measure the expected CPU time a task should have gotten. On ideal hardware, at any time all tasks would have the same vruntime value.

CFS's task picking logic is based on this vruntime value and it thus very simple: it always tries to run the task with the smallest vruntime value (the task which executed least so far). CFS always tries to split up CPU time between runnable tasks as close to ideal multitasking hardware as possible.

CFS uses a time ordered rbtree to build a timeline of future task execution. CFS also maintains the rq-¿cfs.min_vruntime value, which is a monotonic increasing value tracking the smallest vruntime among all tasks in the runqueue. The total amount of work done by the system is tracked using min_vruntime; that value is used to place newly activated entities on the left side of the tree as much as possible.

The total number of running tasks in the runqueue is accounted through the rq-¿cfs.load value, which is the sum of the weights of the tasks queued on the runqueue. CFS maintains a time ordered rbtree, where all runnable tasks are sorted by the vruntime key. CFS picks the leftmost task from this tree and sticks to it. As the system progresses forwards, the executed tasks are put into the tree more and more to thre right slowly but surely giving a chance for every task to become the leftmost task and thus get on the CPU within a deterministic amount of time.

3.4 Schedulers: the plot thickens

CFS works with a single red black tree to track all processes which are in a runnable state. The process which pops up at the leftmost node of the tree is the one which is most entitled to run at any given time.

So the key to understanding this scheduler is to get a sense for how it caculates the key value used to insert a process into the tree. When a task goes into the run queue, the current time is noted. As the process waits for the CPU, the scheduler tracks the amount of processor time it would have been entitled to; this entitlement is simply the wait time divided by the number of running processes (with a correction for different priority values).

The CFS scheduler offers a single tunable: a granularity value which describes how quickly the scheduler will switch processes in order to maintain fairness. A low granularity gives more frequent switching; this setting translates to lower latency for interactive responses but can lower throughput slightly.

There is a relatively small set of methods implemented by each scheduler module, starting with the queueing functions:

```
void (*enqueue_task) (struct rq *rq, struct task_struct *p);

void (*dequeue_task) (struct rq *rq, struct task_struct *p);

void (*requeue_task) (struct rq *rq, struct task_struct *p);
```

When a task enters the runnable state, the core scheduler will hand it to the appropriate scheduler module with enqueue_task(); a task which is no longer runnable is taken out with dequeue_task(). The requeue_task() function puts the process behind all others at the same priority; it is used to implement sched_yield().

A few functions exists for helping the scheduler track processes:

```
void (*task_new) (struct rq *rq, struct task_struct *p);

void (*task_init) (struct rq *rq, struct task_struct *p);

void (*task_tick) (struct rq *rq, struct task_struct *p);
```

The core scheduler will call task_new() when processes are created. task_init() initializes any needed priority calculations and such; it can be called when a process is reniced, for example. The task_tick() function is called from the timer tick to update accounting and possibly switch to a different process.

When it's time for the core scheduler to choose a process to ru, it will use these methods:

```
struct task_struct * (*pick_next_task) (struct rq *rq);

void (*put_prev_task) (struct rq *rq, struct task_struct *p);
```

The call to pick_next_task() asks a scehduler module to decide which process (among those in the class managed by that module) should be running currently. When a task is switched out of the CPU, the module will be informed with a call to put_prev_task().

4 Scheduler

CFS in Coffer Presentation (naive)

1. initialize() initializes the scheduler.

2. next_thread() picks next thread to run.

3. push_thread() push thread into the scheduler queue.
4. tick() handles periodic event on kernel tick.

5 cfs-complex

CFS in Coffer Presentation (cfs-complex)

5.1 struct ThreadFields

```
pub struct ThreadFields {
    pub se: SchedulerEntity,
}

pub struct SchedulerEntity {
    pub max_exectime: AtomicUsize,
    pub vruntime: AtomicUsize,
    pub waittime: AtomicUsize,
}

impl SchedulerEntity {
    pub const fn new() -> Self {
        SchedulerEntity {
            max_exectime: AtomicUsize::new(0),
            vruntime: AtomicUsize::new(0),
            waittime: AtomicUsize::new(0),
        }
    }
}
```

5.2 struct Cfs

```
pub struct Cfs {
    cfs_rq: [SpinLock<BTreeMap<usize, Thread>>; MAX_CPU],
    running: [SpinLock<AtomicCell<Thread>>; MAX_CPU],
    waiting: [SpinLock<VecDequeue<Thread>>; MAX_CPU],
}
```

5.3 initialize()

5.4 next_thread()

```
impl Scheduler for CFS {
    fn next_thread(&self) -> Option<Thread> {
        core_id = id of current core;

        if cfs_rq[core_id] is empty {
            return None;
        } else {
            t = pop first element from cfs_rq[core_id];
            t.se.max_exectime =
                (now - t.se.waittime) / length of cfs_rq[core_id];
            running[core_id] = t.clone();

            return Some(t);
        }
    }
}
```

5.5 tick()

```
impl Scheduler for CFS {
    fn tick(&'static self) -> EventHandleResult {
        core_id = id of current core;

        t = running[core_id];
        t.se.vruntime += 1;
        t.se.max_exectime -= 1;

        if t.se.max_exectime == 0 {
            return EventHandleResult::YieldThread;
        } else {
            pop t from running[core_id] and push t to waiting[core_id];
            return EventHandleResult::Ok;
        }
    }
}
```

5.6 push_thread()

```
impl Scheduler for CFS {
    fn push_thread(&self, t: Thread, _hint: PushHint) {
        core_id = id of t's bounded core;

        for waiting_thread in waiting[core_id] {
            if t.tid == waiting_thread.tid {
                t.se.vruntime = waiting_thread.se.vruntime;
                remove waiting_thread from waiting[core_id];
            }
        }

        if t is new thread {
            t.se.vruntime = vruntime of cfs_rq[core_id]'s first element;
        }

        t.se.waittime = now;
        insert t to cfs_rq[core_id];
    }
}
```

5.7 evaluation

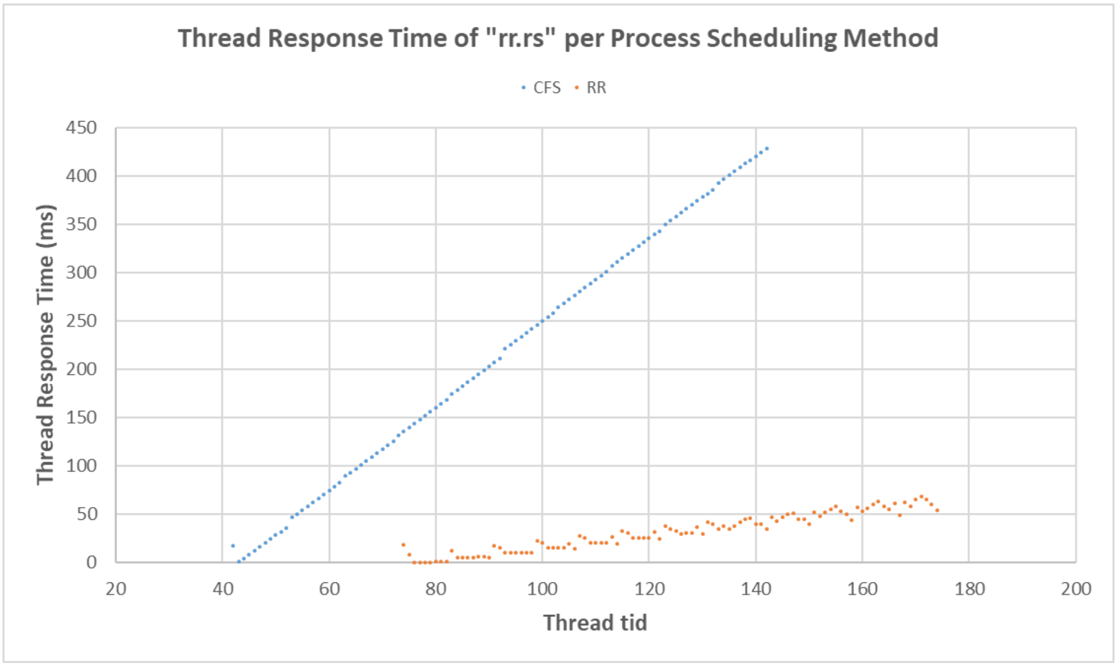


Figure 4: Thread Response Time (cfs-complex vs round robin)

Thread response time of cfs-complex is about 8 times longer than thread response time of round robin. I estimated three reasons for slow thread response time.

The first is the Worst case of the binary search tree. The rr.rs script used to measure thread response time is a script in which 100 threads add 1 to a specific atomic variable once. Due to its nature, 100 threads are generated sequentially, so in the current cfs-complex implemented as a binary search tree rather than a red black tree, a Worst case with a run queue biased to one side is formed.

The second is initializing time. Currently, there are several unnecessary data structures in the cfs-complex that do not require ideal cfs schedulers such as running and waiting. It is thought that it will take a significant time to initialize this.

The third is the non-implementation of load balancing. Due to the nature of CofferOS, where eight cores run in parallel, the presence or absence of load balancing has a significant impact on system performance. Currently, load balancing is implemented in the round robin, and not in the cfs-complex. Considering that the difference in read response time between the two process scheduling methods is about 8 times, it is presumed that the non-implementation of load balancing had the greatest impact.

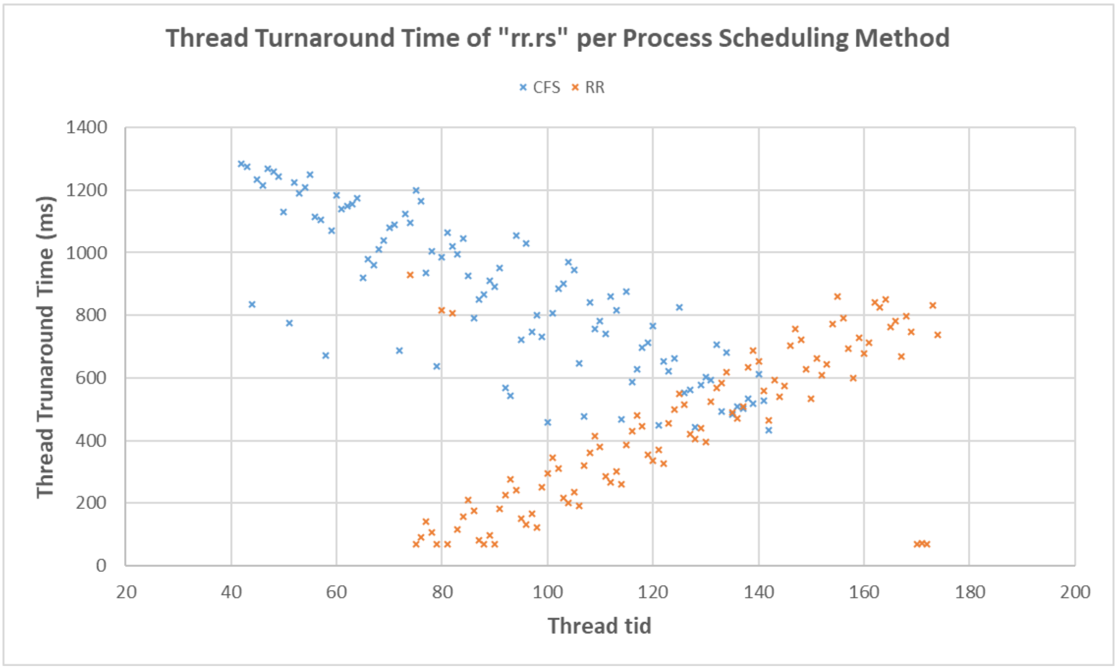


Figure 5: Thread Turnaround Time (cfs-complex vs round robin)

The turnaround time for cfs-complex is listed in reverse order, unlike the turnaround time for round robin. This is believed to be a phenomenon due to the difference in the max execution time allocation method between the two process scheduling methods. For round robin, the max execution time is fixed at 5 ticks, so any thread on rr.rs is sufficient to add a value to the atomic variable. However, in the case of cfs-complex, since the max execution time is obtained as (now-waittime) / len (cfs), the thread that waited longer in the run queue is assigned a longer max execution time, which is believed to terminate earlier than the first thread executed.

When the three response time and the three turnaround time were combined, the difference of maximum response time between the two process scheduling methods was 360 ms, and the difference of maximum turnaround time was 355 ms. Considering that the definition of response time is first run time-arrival time and the definition of turnaround time is completion time-arrival time, there was little difference between completion time and first run time, while there was a big difference between first run and recovery times. Therefore, it is necessary to set a goal in the future in the direction of reducing the read response time.

At this point, I have set three main objectives as improvements to further develop the cfs-complex.

First, information related to process scheduling, which was previously managed by Scheduler Entity, will be wrapped in Arc. Previously, due to Rust's Ownership, several owners could not access Schduler Entity, so it was solved by implementing unnecessary data structures such as running and waiting. However, it will reduce unnecessary data structures and reduce the initialization time of the process scheduler by wrapping the Scheduler Entity with Arc and

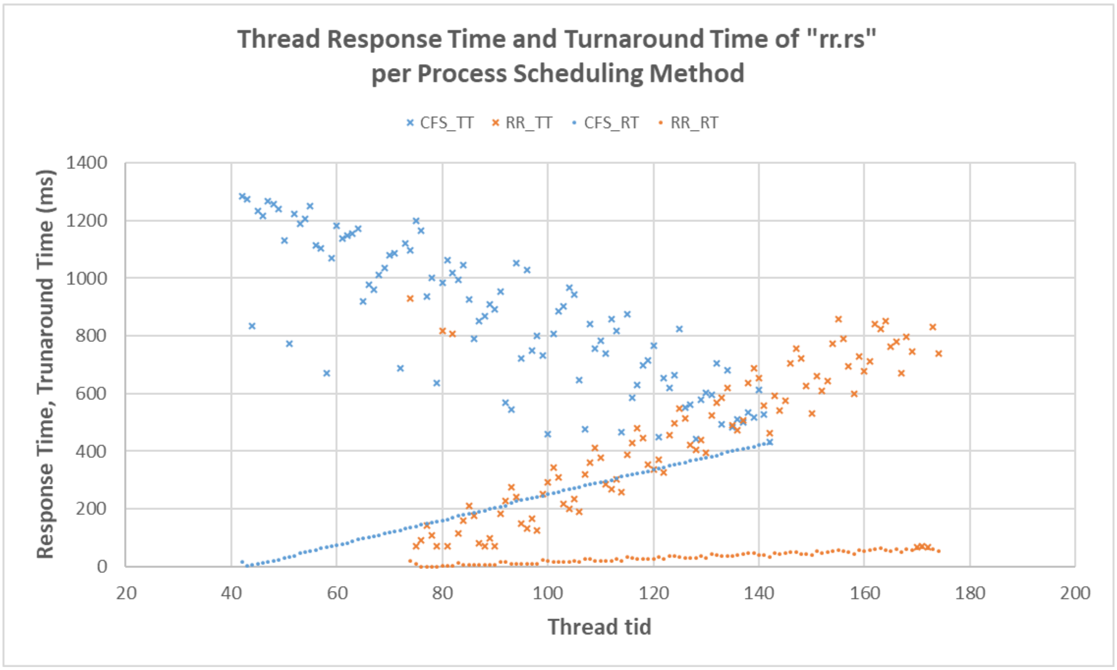


Figure 6: Thread Response/Turnaround Time (cfs-complex vs round robin)

allowing multiple owners to access it.

Second, max_exectime, vruntime, waittime, etc., which were previously managed in tick units, will be changed to nanosecond units to enable more precise process scheduling.

Third, load balancing will be implemented to eliminate the inefficiency caused by using only one existing core and increase the simultaneous utilization of the operating system.

6 cfs-simple

CFS in Coffer Presentation (cfs-simple)

6.1 struct ThreadFields

```
pub struct ThreadFields {
    pub se: Option<Arc<SchedulerEntity>>,
}

pub struct SchedulerEntity {
    pub max_exectime: AtomicCell<Duration>,
    pub vruntime: AtomicCell<Duration>,
    pub exectime: AtomicCell<Instant>,
    pub waittime: AtomicCell<Instant>,
}

impl SchedulerEntity {
    pub const fn new() -> Self {
        SchedulerEntity {
            max_exectime: AtomicCell::new(Duration::ZERO),
            vruntime: AtomicCell::new(Duration::ZERO),
            exectime: AtomicCell::new(Instant::ZERO),
            waittime: AtomicCell::new(Instant::ZERO),
        }
    }
}
```

6.2 struct Cfs

```
pub struct Cfs {
    cfs_rq: [SpinLock<BTreeMap<Duration, Thread>>; MAX_CPU],
    running: [SpinLock<RefCell<Arc<SchedulerEntity>>>; MAX_CPU],
}
```

6.3 move_threads()

```
impl CFS {
    fn move_threads(&self, from: usize, to: usize) {
        from_min_vruntime = vruntime of cfs_rq[from]'s first element;
        to_min_vruntime = vruntime of cfs_rq[to]'s first element;
        relative_vruntime = to_min_vruntime - from_min_vruntime;

        from_num_threads = threads number of cfs_rq[from];
        to_num_threads = threads number of cfs_rq[to];
        num_move_threads =
            (from_num_threads + to_num_threads) / 2 - to_num_threads;
        loop num_move_threads times {
            t = pop last element from cfs_rq[from];
            t.se.vruntime += relative_vruntime;
            push t to cfs_rq[to];
        }
    }
}
```

6.4 do_load_balance()

CFS in Coffer Presentation (load balancing)

```
impl CFS {
    fn do_load_balance(&self, core_id: usize) {
        loop {
            num_threads = threads number of cfs_rq[core_id];
            avg_num_threads = average threads number of every cfs_rqs;
            load_balance_range = plus minus 10% of avg_num_threads;

            if load_balance_range not contains num_threads {
                if num_threads > avg_num_threads {
                    move_threads(core_id, id of idleist core);
                } else {
                    move_threads(id of busiest core, core_id);
                }
            }

            sleep for load_balance_interval milliseconds; (default: 50ms)
        }
    }
}
```

6.5 initialize()

```
impl Scheduler for CFS {
    fn initialize(&'static self) {
        for core_id in active cores {
            spawn thread bounded to core core_id,
                executes do_load_balance(core_id);
        }
    }
}
```

6.6 next_thread()

```
impl Scheduler for CFS {
    fn next_thread(&self) -> Option<Thread> {
        core_id = id of current core;
        if cfs_rq[core_id] is empty {
            return None;
        } else {
            t = pop first element from cfs_rq[core_id];
            t.se.exectime = now;
            t.se.max_exectime =
                (now - t.se.waittime) / length of cfs_rq[core_id];
            running[core_id] = t.se.clone();

            return Some(t);
        }
    }
}
```

6.7 tick()

```
impl Scheduler for CFS {
    fn tick(&'static self) -> EventHandleResult {
        core_id = id of current core;

        se = running[core_id];
        executed = now - se.exectime;
        to_be_executed = se.max_exectime - executed;

        if to_be_executed <= 0 {
            se.vruntime += executed;

            return EventHandleResult::YieldThread;
        } else {
            return EventHandleResult::Ok;
        }
    }
}
```

6.8 push_thread()

```
impl Scheduler for CFS {
    fn push_thread(&self, t: Thread, _hint: PushHint) {
        core_id = id of t's bounded core;

        if t is new thread {
            t.se.vruntime = vruntime of cfs_rq[core_id]'s first element;
        }

        t.se.waittime = now;
        insert t to cfs_rq[core_id];
    }
}
```

6.9 evaluation

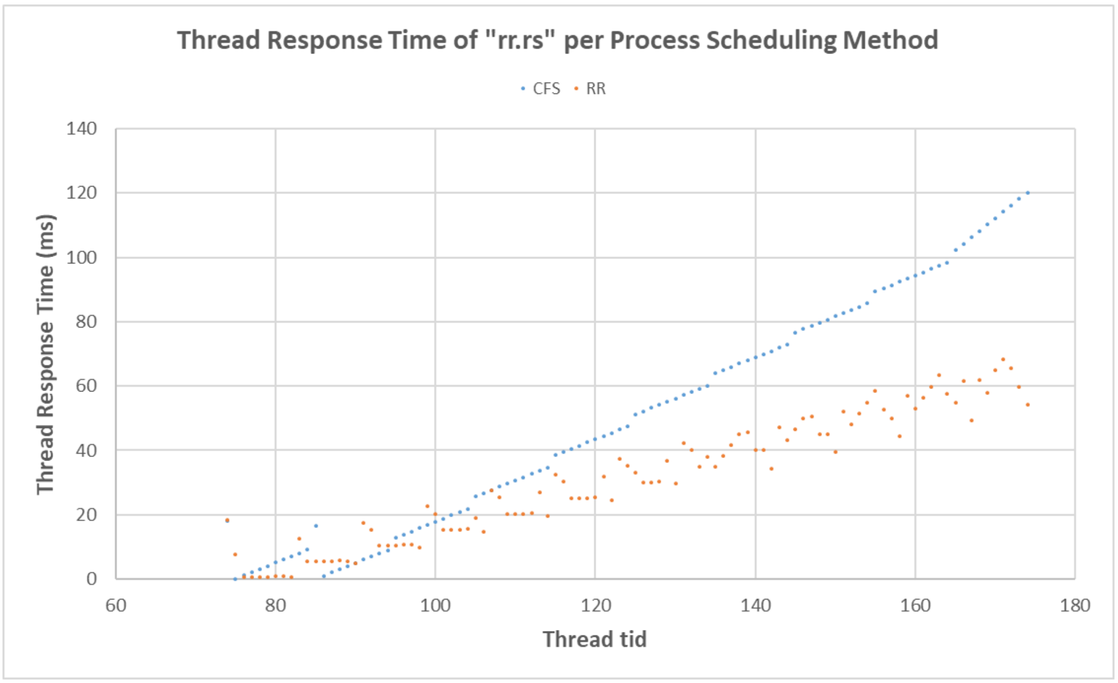


Figure 7: Thread Response Time (cfs-simple vs round robin)

Thread response time of cfs-simple is about 1.5 times longer than thread response time of round robin. After solving the initialize time and load balancing non-implementation problems, it was confirmed that the difference was significantly reduced from 8 to 1.5 times. The worst case problem of the binary search tree remains a challenge to be solved.

There was no significant difference in the thread turnaround time. However, unlike the thread turnaround time distribution of round robin, the thread turnaround time distribution of cfs-simple has a lot of randomness, which is believed to be caused by differences in load balancing implementation methods.

When the three response time and the three turnaround time were combined, the difference of maximum response time was reduced from 360 ms to 52 ms, and the difference of maximum turnaround time was reduced from 355 ms to 59 ms.

Compared to round robin, it has made such a big improvement that no significant difference occurs, but improvements still remain. It is believed that a process of more sophisticated and in-depth testing of the current cfs-simple, such as bug resolution, test case fabrication, and red black tree implementation, will be needed.

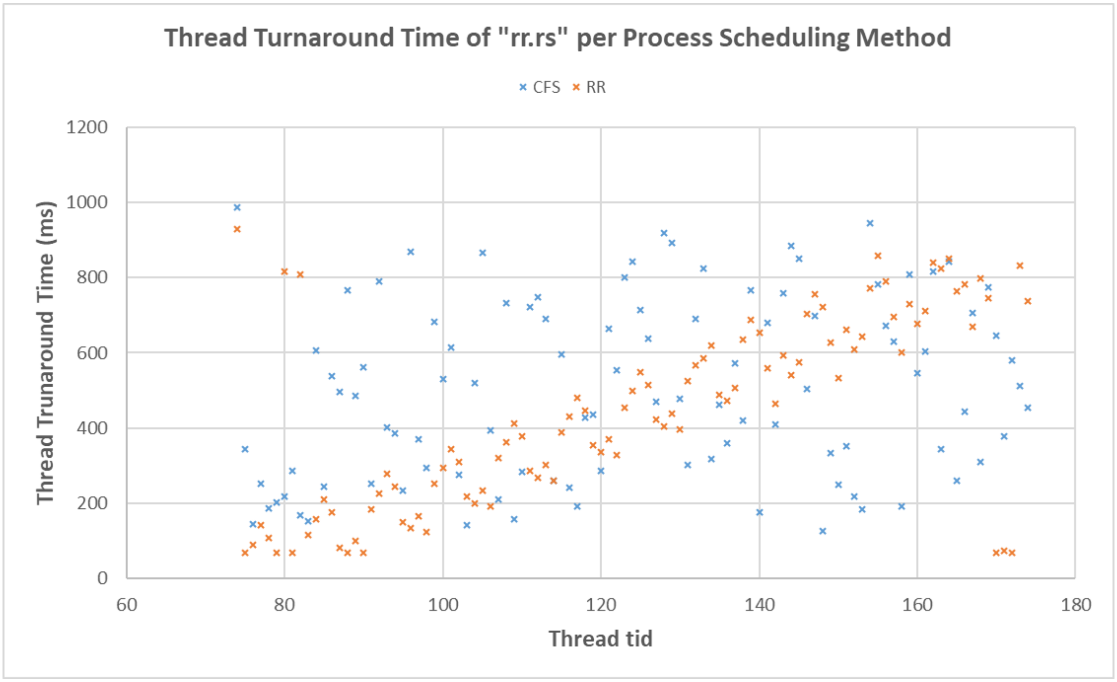


Figure 8: Thread Turnaround Time (cfs-simple vs round robin)

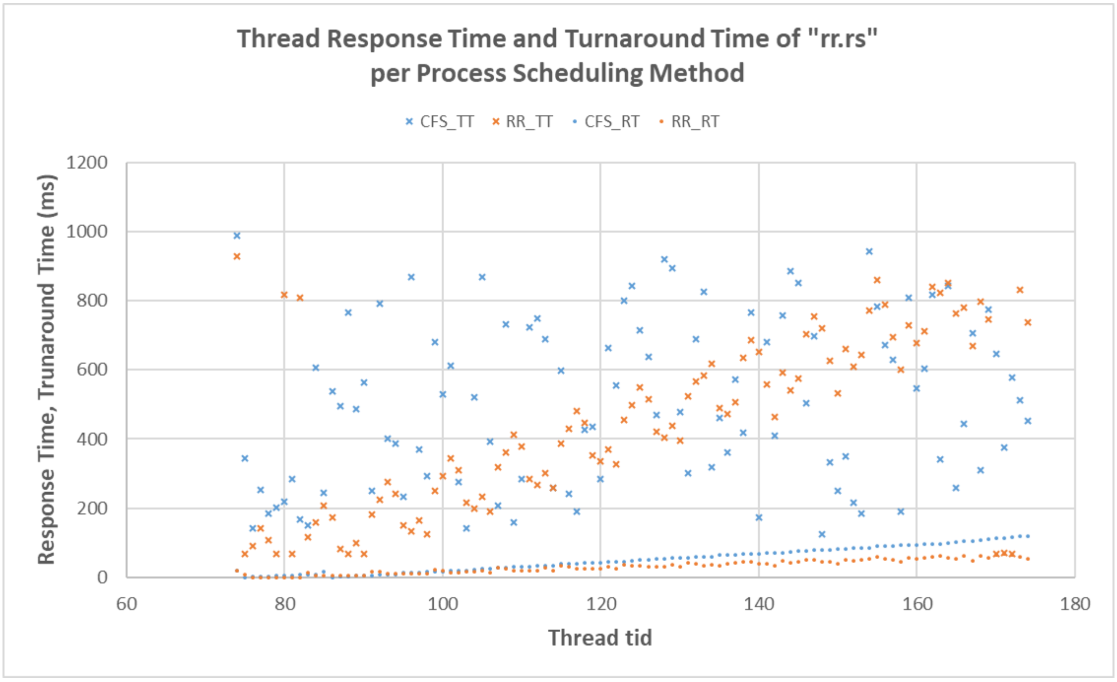


Figure 9: Thread Response/Turnaround Time (cfs-simple vs round robin)

7 Reference

1. Wisconsin Madison: Red-Black Trees
2. Wikipedia: Red–black tree
3. Wikipedia: Completely Fair Scheduler
4. IBM Developer: Inside the Linux 2.6 Completely Fair Scheduler
5. The Linux Kernel: CFS Scheduler
6. LWN.net: Kernel development
7. Github: linux
8. Github: libCoffer.code
9. Github: CofferOS.Tests.code
10. The Rust Programming Language: Rc
11. The Rust Programming Language: Arc
12. Stackoverflow: Completely Fair Scheduler vs Round Robin
13. Figma: CFS in Coffer Presentation