Process Scheduling

Dongyoon Lee

Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
  - syscall, module, kernel data structures
- Process management

Today’s agenda

- What is processing scheduling?
- History of Linux CPU scheduler
- Scheduling policy
- Scheduler class in Linux

Processor scheduler

- Decides which process runs next, when, and for how long
- Responsible for making the best use of processor (CPU)
  - *E.g., Do not waste CPU cycles for waiting process*
  - *E.g., Give higher priority to higher-priority processes*
  - *E.g., Do not starve low-priority processes*

Multitasking

- Simultaneously interleave execution of more than one process
- Single core
  - The processor scheduler gives illusion of multiple processes running concurrently
- Multi-core
  - The processor scheduler enables true parallelism

Types of multitasking OS

- **Cooperative multitasking**: old OSes (e.g., Windows 3.1) and few language runtimes (e.g., Go runtime)
  - A process does not stop running until it decides to *yield CPU*
  - The operating system cannot enforce fair scheduling
- **Preemptive multitasking**: almost all modern OSes
  - The OS can interrupt the execution of a process (i.e., *preemption*)
  - after the process expires its *timeslice*,
  - which is decided by *process priority*

Cooperative multitasking vs. Preemptive multitasking

Process #100

```
long count = 0;
void foo(void) {
 while(1) {
 count++;
 }
}
```

Process #200

```
long val = 2;
void bar(void) {
 while(1) {
 val *= 3;
 }
}
```

Process #300

```
void baz(void) {
 while(1) {
 printf("hi");
 }
}
```

Operating system: scheduler

CPU0

- **Q: how can the preemptive scheduler take the control of infinite loop?**

Scheduling policy: IO vs. CPU-bound tasks

- A set of rules determining *what runs when*
- **I/O-bound processes**
  - Spend most of their time waiting for I/O: disk, network, keyboard, mouse, etc.
  - Runs for only short duration
  - Response time is important
- **CPU-bound processes**
  - Heavy use of the CPU: MATLAB, scientific computations, etc.
  - Caches stay hot when they run for a long time

Scheduling policy: process priority

- **Priority-based scheduling**
  - Rank processes based on their worth and need for processor time
  - Processes with a higher priority run before those with a lower priority

Scheduling policy: Linux process priority

- **Linux has two priority ranges**
  - Nice value: ranges from -20 to +19 (default is 0)
    - High values of nice means lower priority
  - Real-time priority: ranges from 0 to 99
    - Higher values mean higher priority
    - Real-time processes always executes before standard (nice) processes
  - `ps ax -eo pid,ni,rtprio,cmd`

Scheduling policy: timeslice

- How much time a process should execute before being preempted
- Defining the default timeslice in an absolute way is tricky:
    - Too long → bad interactive performance
    - Too short → high context switching overhead

Scheduling policy: timeslice in Linux CFS

- **Linux CFS does not use an absolute timeslice**
  - The timeslice a process receives is function of the load of the system
  - In addition, that timeslice is weighted by the process priority
- When a process `P` becomes runnable:
  - `P` will preempt the currently running process `C` if `P` consumed a smaller proportion of the CPU than `C`

Scheduling policy: example

- Two tasks in the system:
    - Text editor: I/O-bound, latency sensitive (interactive)
    - Video encoder: CPU-bound, background job
- Scheduling goal
    - Text editor: when ready to run, need to preempt the video encoder for good *interactive performance*
    - Video encoder: run as long as possible for better CPU cache usage

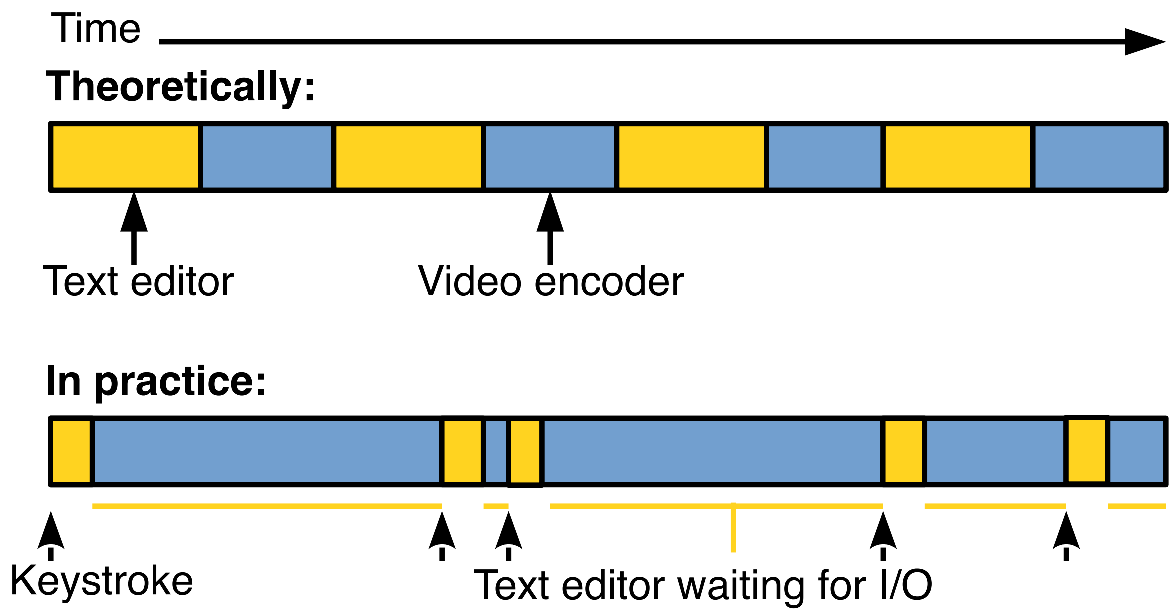
Scheduling policy: example in UNIX systems

- Gives higher priority to the text editor
- Not because it needs a lot of processor but because we want it to always have processor time available when it needs

Scheduling policy: example in Linux CFS

- CFS attempts to offer a fair proportion of CPU time
- CFS keeps track of the actual CPU time used by each program
- E.g., text editor : video encoder = 50% : 50%
    - The text editor mostly sleeps for waiting for user's input and the video encoder keeps running until preempted
    - When the text editor wakes up
        - CFS sees that text editor actually used less CPU time than the video encoder
        - The text editor preempts the video encoder

Scheduling policy: example in Linux CFS



- Good interactive performance
- Good background, CPU-bound performance

Linux CFS design

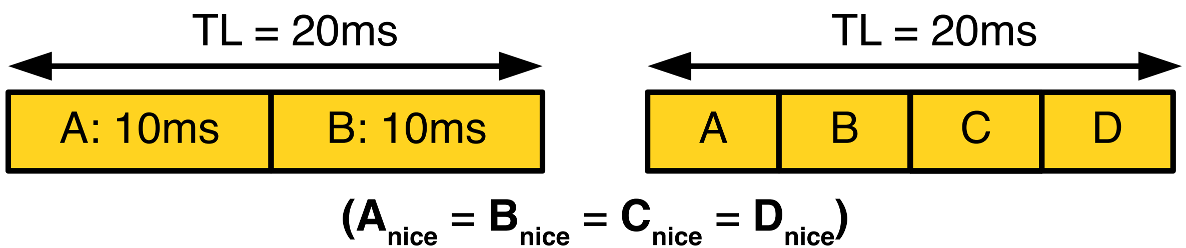
- Completely Fair Scheduler (CFS)
- Evolution of rotating staircase deadline scheduler (RSDL)
- At each moment, each process of the same priority has received an exact same amount of the CPU time
- If we could run `n` tasks in parallel on the CPU, give each `1/n` of the CPU processing power
- CFS runs a process for some times, then swaps it for the runnable process that has run the least

Linux CFS design

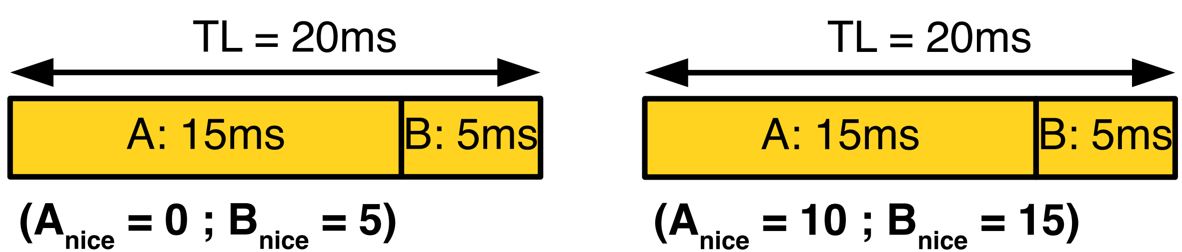
- No default timeslice, CFS calculates how long a process should run according to the number of runnable processes
  - That dynamic timeslice is weighted by the process priority (nice)
  - timeslice = weight of a task / total weight of runnable tasks
- To calculate the actual timeslice, CFS sets a **targeted latency**
  - Targeted latency: period during which all runnable processes should be scheduled at least once
  - Minimum granularity: floor at 1 ms (default)

Linux CFS design

- Example: processes with the same priority



- Example: processes with different priorities



Scheduler class design

- The Linux scheduler is *modular* and provides a *pluggable interface* for scheduling algorithms
  - Enables different scheduling algorithms co-exist, scheduling their own types of processes
- **Scheduler class** is a scheduling algorithm
  - Each scheduler class has a priority.
  - E.g., `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER`
- The base scheduler code iterates over each scheduler in order of priority
  - linux/kernel/sched/core.c: `scheduler_tick()`, `schedule()`

Scheduler class design

- Time-sharing scheduling: `SCHED_OTHER`
    - `SCHED_NORMAL` in kernel code
    - Completely Fair Scheduler (CFS)
    - linux/kernel/sched/fair.c
- Real-time scheduling
    - `SCHED_FIFO`: First in-first out scheduling
    - `SCHED_RR`: Round-robin scheduling
    - `SCHED_DEADLINE`: Sporadic task model deadline scheduling

Scheduler class implementation

- `sched_class`: an abstract base class for all scheduler classes

```
/* linux/kernel/sched/sched.h */
struct sched_class {
    /* Called when a task enters a runnable state */
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Called when a task becomes unrunnable */
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    /* Yield the processor (dequeue then enqueue back immediately) */
    void (*yield_task) (struct rq *rq);
    /* Preempt the current task with a newly woken task if needed */
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
    /* Choose a next task to run */
    struct task_struct * (*pick_next_task) (struct rq *rq,
                                            struct task_struct *prev,
                                            struct rq_flags *rf);
    /* Called periodically (e.g., 10 msec) by a system timer tick handler */
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    /* Update the current task's runtime statistics */
    void (*update_curr) (struct rq *rq);
};
```

Scheduler class implementation

- Each scheduler class implements its own functions

```
/* linux/kernel/sched/fair.c */
const struct sched_class fair_sched_class = {
    .enqueue_task       = enqueue_task_fair,
    .dequeue_task       = dequeue_task_fair,
    .yield_task         = yield_task_fair,
    .check_preempt_curr = check_preempt_wakeup,
    .pick_next_task     = pick_next_task_fair,
    .task_tick          = task_tick_fair,
    .update_curr        = update_curr_fair, /* ... */
};
/* scheduler tick hitting a task of our scheduling class: */
static void task_tick_fair(struct rq *rq, struct task_struct *curr, int queued)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &curr->se;
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        entity_tick(cfs_rq, se, queued);
    } /* ... */
}
```

Scheduler class implementation

- `task_struct` has scheduler-related fields.

```
/* linux/include/linux/sched.h */

struct task_struct {
    struct thread_info       thread_info;
    /* ... */
    const struct sched_class *sched_class; /* sched_class of this task */
    struct sched_entity      se; /* for time-sharing scheduling */
    struct sched_rt_entity   rt; /* for real-time scheduling */
    /* ... */
};
```

Scheduler class implementation

```
/* linux/include/linux/sched.h */

struct sched_entity {
    /* ... */
    struct rb_node      run_node;

    u64                 exec_start;
    u64                 sum_exec_runtime;
    u64                 vruntime; /* how much time a process
                                   * has been executed (ns) */
    struct cfs_rq       *cfs_rq; /* CFS run queue */
    /* ... */
};

struct cfs_rq {
    /* ... */
    struct rb_root_cached   tasks_timeline; /* rb tree */
    /* ... */
};
```

Two scheduler entry points

- The base scheduler code triggers scheduling operations in two cases
  - when processing a timer interrupt (`scheduler_tick()`)
  - when the kernel calls `schedule()`

A timer interrupt calls *scheduler_tick()*

```
/* linux/kernel/sched/core.c */
/* This function gets called by the timer code, with HZ frequency. */
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;
    struct rq_flags rf;

    /* call task_tick handler for the current process */
    sched_clock_tick();
    rq_lock(rq, &rf);
    update_rq_clock(rq);
    curr->sched_class->task_tick(rq, curr, 0); /* e.g., task_tick_fair in CFS */
    cpu_load_update_active(rq);
    calc_global_load_tick(rq);
    rq_unlock(rq, &rf);

    /* load balancing among CPUs */
    rq->idle_balance = idle_cpu(cpu);
    trigger_load_balance(rq);
    rq_last_tick_reset(rq);
}
```

The kernel calls *schedule()*

```
/* linux/kernel/sched/core.c */
/* __schedule() is the main scheduler function. */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    /* pick up the highest-prio task */
    next = pick_next_task(rq, prev, &rf);

    if (likely(prev != next)) {
        /* switch to the new MM and the new thread's register state */
        rq->curr = next;
        rq = context_switch(rq, prev, next, &rf);
    }
    /* ... */
}
```

Scheduler class implementation

```
/* linux/kernel/sched/core.c */
/* Pick up the highest-prio task: */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* ... */
again:
    for_each_class(class) {
        /* In CFS, pick_next_task_fair() will be called */
        p = class->pick_next_task(rq, prev, rf);
        if (p) {
            if (unlikely(p == RETRY_TASK))
                goto again;
            return p;
        }
    }

    /* The idle class should always have a runnable task: */
    BUG();
}
```

Next lecture

- Processing Scheduling II