Process Scheduling

Xiaoguang Wang

Summary of past lectures

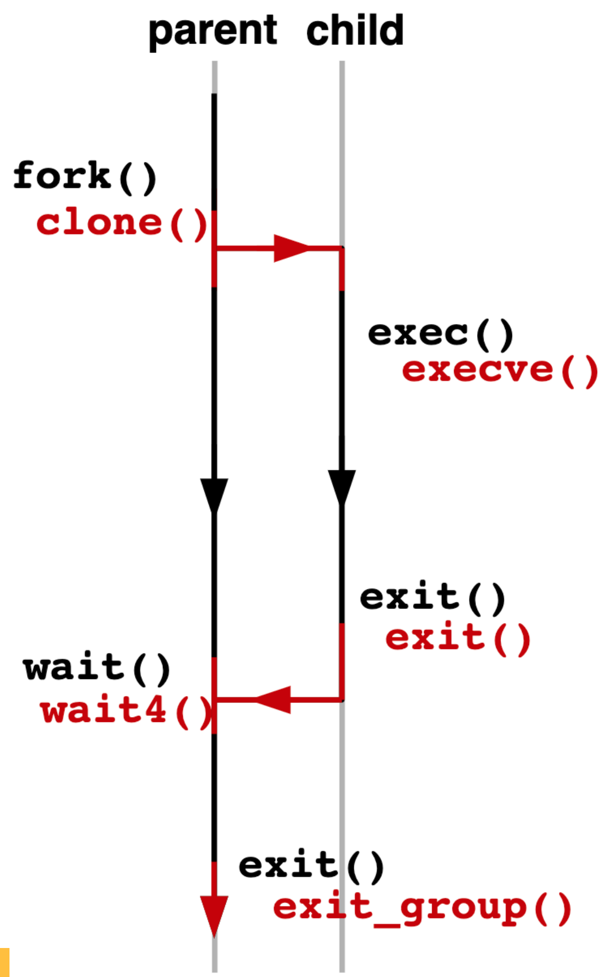
Tools: building, exploring, and debugging Linux kernel

Core kernel infrastructure

- syscalls, module, kernel data structures

Process management

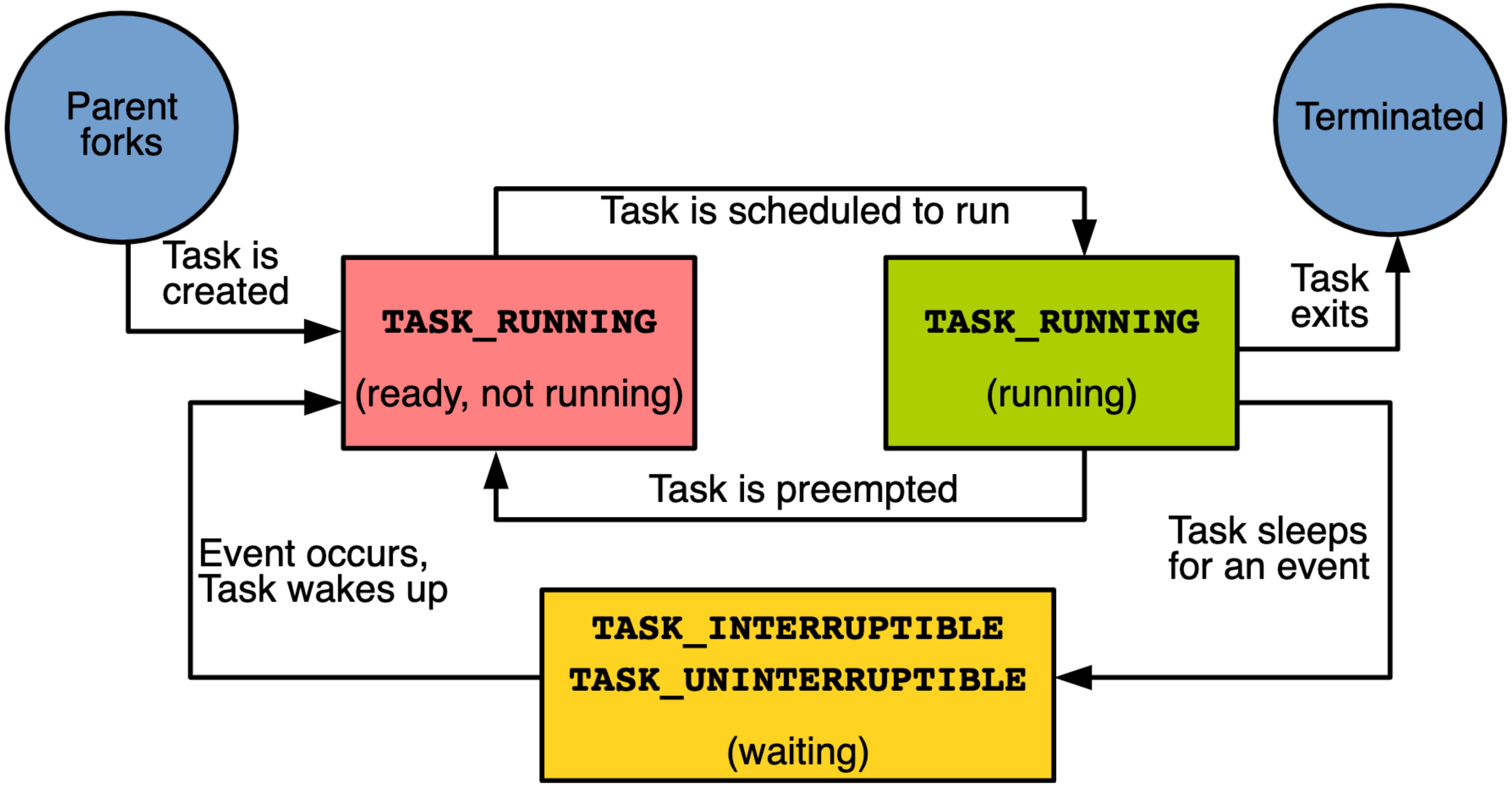
Recap: process from the user-space view



Recap: process descriptor: `task_struct`

```c
/* include/linux/sched.h */
struct task_struct {
    struct thread_info   thread_info;   /* thread information */
    volatile long        __state;       /* task status: TASK_RUNNING, etc */
    void                 *stack;        /* stack of this task */
    int                  prio;          /* task priority */
    struct sched_entity  se;            /* information for processor scheduler */
    cpumask_t            cpus_mask;     /* bitmask of CPUs allowed to execute */
    struct list_head     tasks;         /* a global task list */
    struct mm_struct     *mm;           /* memory mapping of this task */
    struct task_struct   *parent;       /* parent task */
    struct list_head     children;      /* a list of child tasks */
    struct list_head     sibling;       /* siblings of the same parent */
    struct files_struct  *files;        /* open file information */
    struct signal_struct *signal;       /* signal handlers */
    /* ... */
    /* NOTE: In Linux kernel, process and task are interchangably used. */
}; /* TODO: Let's check `pstree` output. */
```

Recap: process status: `task->_state`



Recap: process creation and termination

`fork()` and `exec()`

- `fork()` creates a child, copy of the parent process
  - Copy-on-Write (CoW) -- memory (page tables)
  - Through `clone()` system call (flags indicating sharing -- thread)
- `exec()` loads into a process address space a new executable

Termination on invoking the `exit()` system call

- Can be implicitly inserted by the compiler on `return` from `main()`
- `sys_exit()` calls `do_exit() ...`

Recap: kernel thread

Used to perform background operations in the kernel

- Very similar to user space threads

To create a kernel thread, use `kthread_create()`

When created through `kthread_create()`, the thread is not in a runnable state

Need to call `wake_up_process()` or use `kthread_run()`

Other threads can ask a kernel thread to stop using `kthread_stop()`

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 interleaved 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 time-slice,
- which is decided by process priority

Types of multitasking OS

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**?

I/O- vs. CPU-bound tasks

Scheduling policy: 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

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

Linux process priority

Linux has two priority ranges

- Nice value: ranges from -20 to +19 (default is 0)
  - Higher values of nice means lower priority
  - e.g., `nice -n 5 vim; sudo renice -n -5 -p $(pidof vim)`
- Real-time priority: ranges from 0 to 99
  - Higher values mean higher priority
  - Real-time processes always execute before standard (nice) processes

Try it out: `ps ax -eo pid,ni,rtprio,cmd`

Scheduling policy: time slice

Time slice: the period for which a process is allowed to run uninterrupted in a preemptive multitasking operating system.

Defining the default time-slice in an absolute way is tricky:

- Too long → bad interactive performance
- Too short → high context switching overhead

Example: Real-time Round Robin Scheduling (SCHED_RR)

```
$ cat /proc/sys/kernel/sched_rr_timeslice_ms
100
```

Scheduling policy example

Two tasks in the system:

- Text editor: I/O-bound, latency sensitive (interactive)
- Video encoder: CPU-bound, background job

Scheduling goals:

- Text editor:
- Video encoder:

Scheduling policy example

Gives higher priority to the text editor

- Not because it needs a lot of processor cycles but because we want it to always have processor time available when it needs

Policy: time slice in Linux CFS

CFS: Completely Fair Scheduler

Linux **CFS does not use an absolute time-slice**

- The time slice a process receives is a function of the load of the system (i.e., a proportion of the CPU)
- In addition, that time slice is weighted by the process priority
- When a process P becomes runnable:
  - P will preempt the currently running process C if P consumes a smaller proportion of the CPU than C

Policy: example in Linux CFS

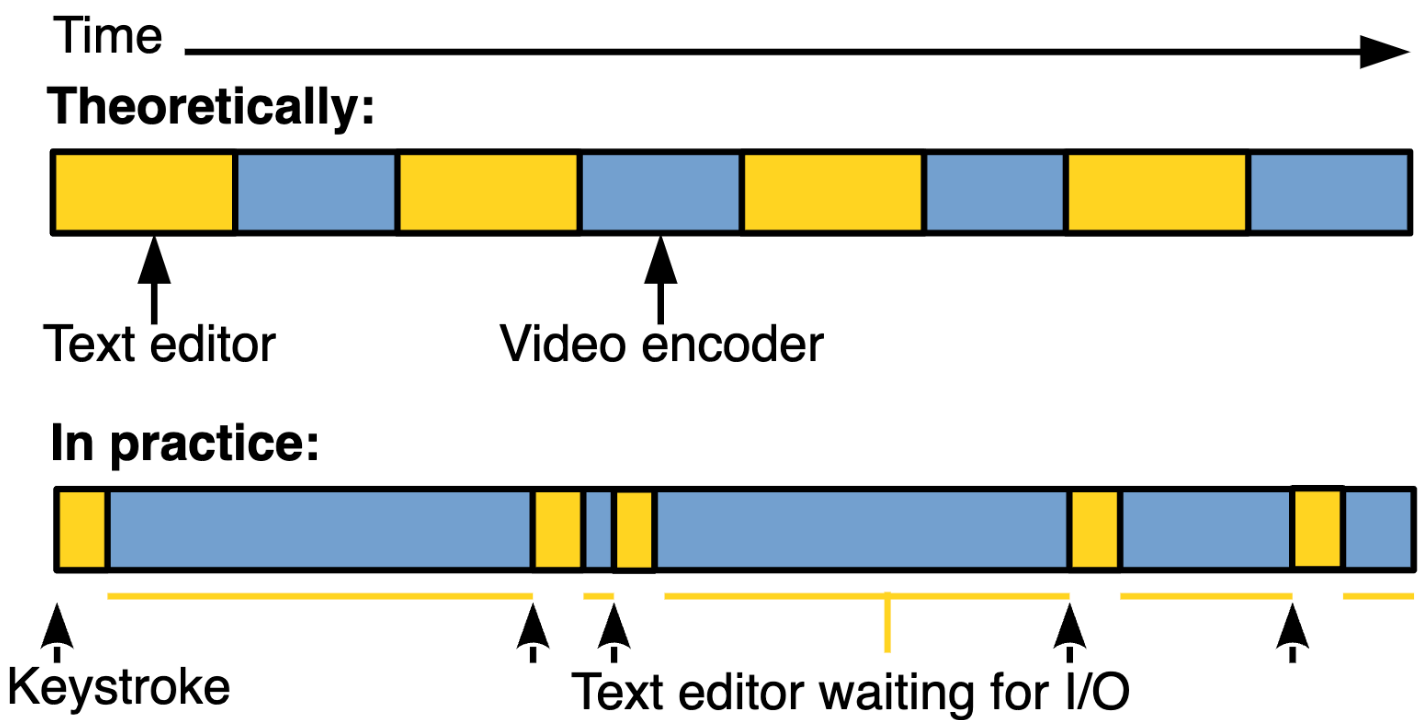
CFS guarantees the text editor a specific proportion of CPU time

- 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 waiting for the user's input, and the video encoder keeps running until preempted
- When the text editor wakes up
  - CFS sees that text editor used less CPU time than the video encoder
  - The text editor preempts the video encoder

Policy: example in Linux CFS



Good interactive performance

Good background, CPU-bound performance

Linux CFS design

At each moment, each process of the same priority has received the exact same amount of CPU time

If we could run `n` tasks in parallel on a CPU, give each `1/n` of the CPU processing power

CFS runs a process for some time, then swaps it for the runnable process that has run the least

Linux CFS design

No default time slice, CFS calculates how long a process should run according to the number of runnable processes

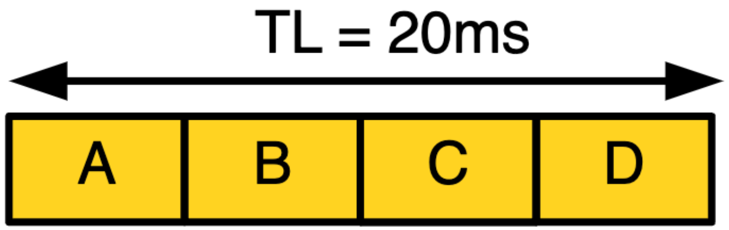
- That dynamic time slice is weighted by the process priority (`nice`)
- time slice = weight of a task / total weight of runnable tasks

To calculate the actual time slice, 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)

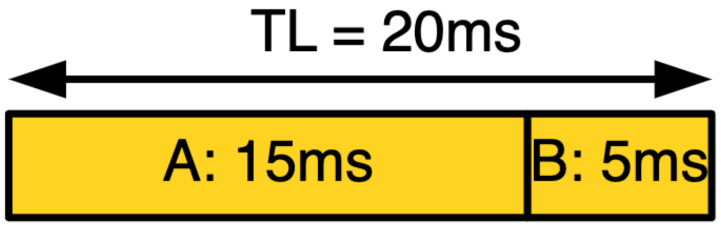
Linux CFS design

Example: processes with the same priority



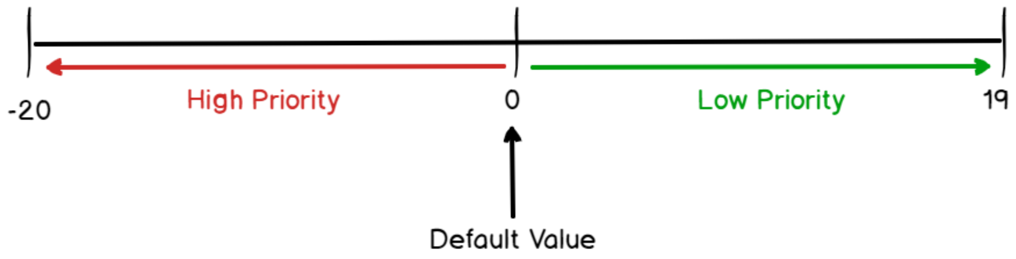
$(A_{nice} = B_{nice} = C_{nice} = D_{nice})$

Example: processes with different priorities



$(A_{nice} = 0 \; ; \; B_{nice} = 5)$

Niceness Scale on Linux



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_BATCH`

The base scheduler code iterates over each scheduler in priority order

- `kernel/sched/core.c`: `scheduler_tick()`, `schedule()`

Scheduler class design

Time-sharing scheduling (non-real-time)

- `SCHED_NORMAL` default scheduling policy
- `SCHED_BATCH` is a scheduling policy for batch processing tasks
- Completely Fair Scheduler (CFS)
- `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

```c
/* 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 */
DEFINE_SCHED_CLASS(fair) = {
    /* const struct sched_class fair_sched_class = { */
    .enqueue_task        = enqueue_task_fair,
    .dequeue_task        = dequeue_task_fair
    .yield_task
    .check_preem
    .pick_next_t
    .task_tick
    .update_cur
};
```

```
/* 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 {
    /* ... */
    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 */
    /* ... */
};
struct sched_entity {
    /* For load-balancing: */
    struct load_weight  load;
    struct rb_node      run_node;
    struct list_head    group_node;
    unsigned int        on_rq;

    u64                 exec_start;
    u64                 sum_exec_runtime;
    u64                 vruntime; /* how much time a process
                                   * has been executed (ns) */
```

Scheduler class implementation

The base scheduler code triggers scheduling operations in two cases

- when processing a timer interrupt (`scheduler_tick()`)
- when the kernel calls `schedule()`

Scheduler class implementation

```c
/* 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);
}
```

Scheduler class implementation

```c
/* 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

```c
/* 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 step: hw 6 is out

hw6 - Design and Implement a CPU Profiler (Part 1)

- write a kernel module to hook the `pick_next_task_fair()` function using `Kprobes`, and count the PID to be scheduled in
- print the statistic results to `/proc/perftop`
- due: Feb 16^{th}

hw4 - linked list and kernel module

- due: tomorrow

hw5 - hash table, rbtree and xarray

- due: next Friday (Feb 9^{th})

Next week

More about the Linux scheduler

Interrupt handling