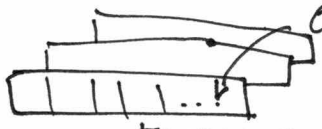
Scheduler (cont.)

CORE scheduler functionality

Schedule()
→ has a run queue
→ selects a task to run
→ context switch to that new task



enqueue to run queue (ex. sth. becomes runnable because woken up, new process, moved from another runqueue)

ADDITIONAL functionality (outside of schedule())

Example: /kernel/fork.c

(3.10 kernel)
p = copy_process(...)
....
wake_up_new_task(p) → set_task_cpu(p,...)
....
activate_task(p) → ... enqueue_task(rq, p, ...) ...

calls enqueue_task on the p→sched_class

class-specific versions of enqueue_task:
- idle_task has no implementation
- fair has enqueue_task_fair

calls enqueue_entity on the CFS runqueue (cfs_rq) of the sched_entity (se) related to the task we are enqueuing (&p→se)

NOTE: there is a list of sched/ing classes, there is a .next member pointer in the sched_class struct

struct cfs_rq has ~~CF~~ on sched/sched.h has "CFR related fields on a runqueue"

I.E. there is a runqueue that is specific to the scheduling class that this enqueue_task_fair is going to put this task on (i.e. on a class-specific rung)

NOTE: In Linux, the generic rung is composed of class-specific rungs and the class-specific functions stick what they want to enqueue and dequeue to those specific rungs.

RQ:
- CFS_RQ
- RT_RQ

In sched/sched.h:
```
struct cfs_rq;
struct rt_rq;
```

NOTE: to add your own scheduler, add its class-specific rung here →

In sched/sched.h: "the main, per-CPU runqueue data structure"
```
struct rq {
...
  struct cfs_rq;   ← the class-specific rungs are members of the main rung
  struct rt_rq;
...
  struct task_struct * idle;   ← the idle scheduler rung, which consists of one task
...
}
```

NOTE: These (cfs, rt, and idle) are all the schedulers implemented in 3.10 kernel!

▶ sched_class struct in sched/sched.h is the generic scheduling class structure; it has pointers to the following functions:

COMSW4118-14-2 10/19/2017

- enqueue_task = put something on the rung
- dequeue_task = take sth. off of the rung
- pick_next_task = the key schedule func. to select what runs next

(!) THESE are called from the core scheduler on sched/core.c:

- task_tick = called (among others) from scheduler_tick, which gets called by the timer code, w/ HZ frequency w/ interrupts disabled. Checks to see if another task should run. (~~once a second~~) More or less, the interrupt handler for the timer interrupt
  - gets the rung for the current CPU (where the timer interrupt went off),
  - gets the current job running on that CPU (rq->curr)
  - calls the scheduling class' task_tick for the sched_class of the currently running task
    - ex: task_tick_idle: does nothing
    - ex: task_tick_rt: ... if (--p->rt.time_slice) return; ...
      - as long as the time-slice is positive, just keep going
      - The RT scheduler is giving each process some time to run (aka time-slice or time-quantum)
      - Otherwise "requeue to the end of queue" and set the reschedule flag on the current task: ... set_tsk_need_resched(p); ...

NOTE: The timer interrupt goes off whenever the timer goes off, the timer goes off at some fixed interval (for most architectures). ← goes off regardless of what else is going on. The OS will check, when returning from the timer interrupt, if it needs to reschedule! Most modern hardware works this way. Some hardware allows us to schedule the timer irregularly, if we know we will be running the same task.

- Expensive to call the timer interrupt? Say we call each millisecond on a 1GHz CPU and it costs us 1000 instructions (out of the 1 million we execute each millisecond). Not really expensive.

▶ Scheduling Algorithms:

(1) **FIFO:**

and run till it's done ← no scheduling decision on timer interrupt

MINUS: while(1) will lock up the CPU

BUT: if current job blocks, it still goes on a wait q and the next one from the rung is scheduled. What happens when the former job wakes up?
- ▶ put it at the end - simple, keeps the FIFO principle, typical implementation
- ▶ Preempt the task that is running
- ▶ insert at the front at the queue

(2) **Round Robin:**

goes around, giving each task an equal shot at running (the easiest fair algo)

But only run for some time quantum say 10ms. Then, if not done, return to back of rung

- Example: Linux RT scheduler task_tick_rt:
  RT scheduler supports two different types of scheduling:
  - SCHED_RR = Round Robin
  - SCHED_FIFO = FIFO

  task_tick_rt has ... if (p->policy != SCHED_RR) return; ...
  I.e. it runs FIFO tasks till completion!
  Otherwise (SCHED_RR), decrease timeslice and, if zero, return to back of rung and reschedule (via. reschedule flag).
- The Linux RT scheduler is a little more complex than the above idea of having a rung with either FIFO or Round Robin for all tasks in it.
- The rt class in Linux has priorities (numbers) associated with each task: You run the task w/ the highest priority.
- The caveat w/ priority scheduling is how to break ties... This is either FIFO or Round Robin as described above.
- Ways to implement a priority scheduler (NOTE: under each implementation, a low priority task will not run if higher one is ready)
  1. One rung w/ tasks w/ diff priorities - scheduler searches the rung to find the job w/ the highest priority and run it (search may be lengthy).
  2. Multiple rungs, one for each priority level - scheduler finds first non-empty queue. NOTE: also, while(1) at high priority will hose up the CPU.
     Linux uses this and w/in each queue the jobs are marked SCHED_RR or SCHED_FIFO. Priority scheduling is ~~default~~ not the default in Linux → you need root privileges to change task priorities.

NOTE: If you CTRL-C in a shell to stop an errand high-priority process, you may not be able to since the shell process needs to actually run and it may never do so while the higher priority process runs (the CTRL-C interrupt will still happen and get executed but the shell process won't have a chance to send the associate signal to the errand high-priority process). You can certainly reboot the computer to stop the high-priority process.

NOTE: fork() copies the scheduling parameters (scheduler, priority, etc.) from parent, so these are inherited.

By default, Linux uses the (completely) Fair Scheduler (CFS) → the first process that gets created is CFS scheduled and that gets inherited.

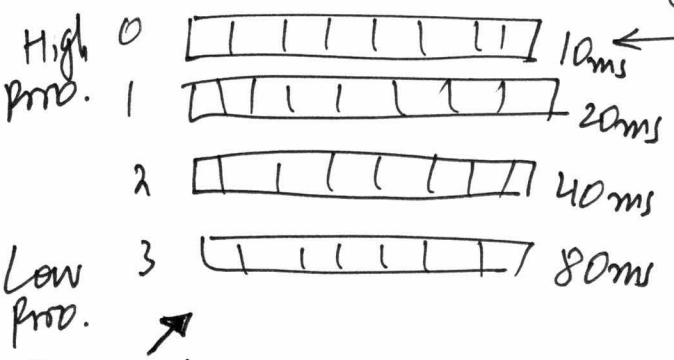
To run in a custom (different) scheduler, you need to find that initial task and, before it creates the other processes, change it or them to use that custom scheduler.

COMSW4118-14-4 10/19/2017

- When you implement a scheduler and need to test it w/ some jobs, make sure that the jobs are actually using this scheduler class!
- Even when you inherit the scheduling parameters of your parent, that does not mean you have to do everything your parent does: ← you adjust some!!!
  - ▶ say parent has 10 ms quantum to run, do we use same for child?
    - Many forks from same parent may usurp the system if each child has 10ms too.
    - You can change so fork() gives ½ time quantum to the child (ex. 5ms left to parent and 5ms to child) - overall quantum stays constant
    - An in-between approach where we reduce the time quantum w/ each fork().

(3) **Shortest Job First:** If you run the shortest job first, the wait time is minimized and the job throughput is maximized. (OVERALL!)

- PROBLEM: How do you know how long the job will run?
- An approximation is used w/ many rungs in a prioritized list:



all jobs start here, each gets 10ms to run, if job is not done in 10ms it gets bumped to next (lower) rung and time-quantum is increased (say 20ms).

EFFECT: short jobs run at high priority for all of their life.

- This is also a common way to assign priorities AKA Multi-Level (Queue) Scheduling and is commonly used (ex. interactive commands are short & high priority)

But, this has fallen out of favor because there are some problems w/ it:

- Long-running tasks may still need to respond quickly (ex. web browser, audio player, etc.) but will get buried to a low priority queue.
- You can improve the above a bit by splitting those long-running processes into continuous runs, for example, between sleeping - so, each time your process sleeps, put it back in the High prio que.
- But such base on sleep/run behavior is also not optimal.