COMSW4118-13-1 10/17/2017

- **schedule() function** (from last time)

This does the CPU scheduling. The essential problem is:
- You have a queue of tasks
- You need to pick a task to run & determine for how long it will run

▶ **When do you schedule?**

1. When the running task voluntarily gives up the CPU (ex. when the task knows it will have to wait, so it puts itself in non-runnable state and calls schedule()).
2. When the running task involuntarily gives up the CPU (ex. timer goes off and the OS determines the current task has run long enough and another needs to be run instead).
3. A task becomes runnable and the scheduler needs to decide if it needs to be run (ex. you fork a new task).
4. When the scheduling parameters change (ex. a high-priority task is demoted)

▶ **Types of scheduling** that have to happen:
- **Non-preemptive** (meaning job is not preempted, it is run till it voluntarily gives up the CPU).
  Only 1 above applies, this is a form of cooperative-multitasking
- **Preemptive** (more complex, but more accurate, can forcefully swap tasks, requires interrupts (timer)
  Cases 1-4 above apply

▲ **NOTE:** cooperative multitasking involves tasks voluntarily giving up the CPU. This is usually implemented inside libraries that apps use by calling yield(). Obviously, a while(1); application will usurp a CPU since it does not call a library function and does not voluntarily yield.

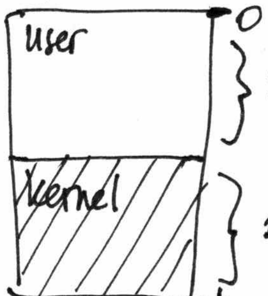
NOTE: The term "preemptive" is used in different contexts:
- scheduling - above discussion
- kernel - different meaning covering only processes running in kernel and if they run till completion of kernel part or can be preempted.
- deadlock - meaning forcefully giving up a lock.

▶ **Calling the scheduler**
- Just call schedule() (ex. semaphore not available, or any time a process needs to block and is no longer runnable)
- Indirectly call schedule() (ex. you are somewhere in the OS where it is not convenient to directly call schedule(), which is a complex and time-consuming function.

/kernel/sched/core.c → schedule() calls __schedule()

For example, in the context of an interrupt, which we want to be fast and cheap, we may not want to call schedule() directly. You can call a function called scheduler_tick() instead, to count time since an interrupt went off and, if it is determined that the interrupt took longer than some time, set a scheduler flag, so the scheduler is called once we are done w/ the interrupt handling and before we return to the process that was running.

- **Process** (address space) →



3GB for user

Sometimes 2 and 2 in some OSs

1GB for kernel (SHARED!)

4GB total process space (virtual)

When process enters kernel code, it switches to the kernel portion of its address space: →

Separate portion of process's address space devoted to running kernel code. → In Linux, though, all processes share the same kernel code and structures, so the kernel portion of the address space is mapped to the same kernel stuff (i.e. code is the same and data values are the same for all processes' kernel space).

NOTE: Processes are separate entities BUT their kernel address space is the same! BUT: Processes get their own kernel stacks!

← But it points to the same space, i.e. code/data is the same for all processes and data changes are seen by all.

NOTE: copy_from_user() and copy_to_user() copy to/from user/kernel address space.

- When an interrupt goes off, we save the user space context (registers, IP, etc.) and jump to the kernel space to run the interrupt handler in the context of the kernel portion of the process we were running (which is in essence the same thing for all processes). When done, we restore the user process state and switch to user space. (this is done in kernel code).
- On this "way out of the kernel", the kernel checks some things:
  - check sched_flag to see if it was set so we don't want to resume the user process, then we call schedule(). Two cases for sched_flag to be set:
    1. Sometimes (ex. an interrupt), it is not convenient to call schedule() directly
    2. Sometimes, you want to batch up changes and only call schedule() once when they are all done.
  - check for signals pending - the signal() syscall sets a flag on the process that received the signal - when it runs, the received signal would be handled by the process, or if no handler or signal can't be handled by process, by kernel (ex. kill signal will cause the process to be exited instead of restarted.

COMSW4118-13-3 10/17/2017

NOTE: check the comments to __schedule() for a description of what the scheduler looks like:

1. get the run queue (curr points to the task currently running)
2. Locking and other stuff
3. Calls pick_next_task() ~~from~~ to ~~get~~ choose a task from the runqueue to run.
4. If prev task ≠ task picked to run:
   - context_switch() from prev to new task
   - now we are running the new task...

...

NOTE: need_resched() is the thing that checks the scheduler flag to see if scheduling is needed.

scheduler_tick() gets called by the timer code with HZ frequency to call task_tick and determine if (an interrupt?) operation has taken too long and scheduling needs to occur.

- A little more about the **structure of the Linux scheduler & how it works:**

▸ Linux allows a variety of scheduling policies (thing that determines what runs when) but has a common implementation mechanism to support ~~it~~ them (i.e. same mechanism for run queues, how ~~you~~ take/add stuff from/to them, etc.)

- **kernel/sched/core.c** is this core underlying scheduler mechanism (includes schedule())
- On top of that, there are a bunch of policies:
  - ▸ **fair.c** for fair scheduler
  - ▸ **rt.c** for the real-time scheduler

  ...
- If you want to add a new scheduler/scheduling policy, you need to add a new scheduling class, but you will keep the core part the same, it will just have to make calls to the ~~new~~ scheduler class to make scheduling decisions. (Ex:) pick_next_task() in core.c goes over all scheduling classes and runs the class-specific version of pick_next_task() and will return the task picked by the first class that returns non-null (i.e. has something to run). But, the code that switches the context to this (likely new) task is back in the core underlying scheduler mechanism.

NOTE: Each task is allowed to belong to one and only one scheduling class at a point in time.

rt is usually first ← order is static! in the list and we get to fair only if rt has nothing to run.

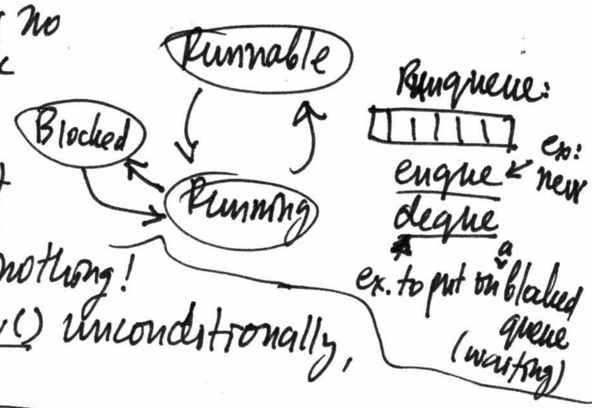
BUT: there is a check "if (likely(rq->nr_running == rq->cfs.h_nr_running))..." for the likely scenario that all tasks use the completely fair scheduler when we will short-circuit to calling it directly, instead of going through all classes.

▶ kernel/sched/sched.h defines the ~~sched~~ class struct with pointer to next, so this is a .. list (static) of sched_class scheduler classes.

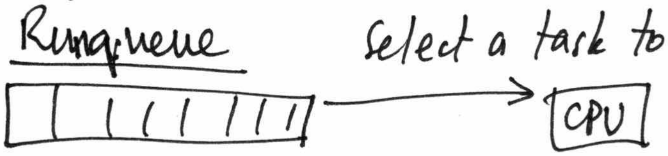
- There are a variety of function pointers in sched_class that a new class has to implement and set.

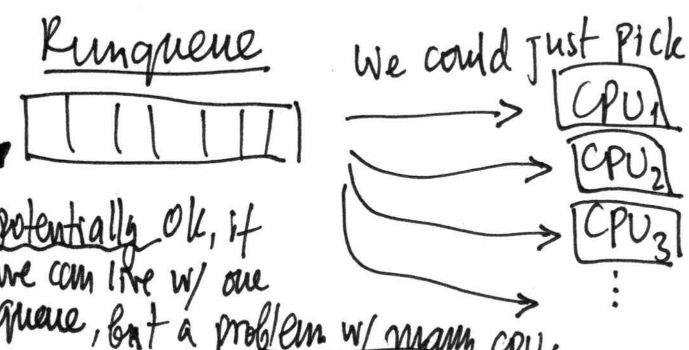
Ex: of a scheduling class (short) is idle_task.c for the idle task (when the CPU has nothing to do, it runs an idle task that does almost nothing but maybe power-saving, etc.). The idle_task scheduler class is used specifically to run the idle task:

- → **NOTE**: The class sets 12 or so function pointers out of 23 function pointers in sched.h – so, not all defined in sched.h are required. (kernel/sched/sched.h, where the internal scheduler defines are, public are in ~~kernel~~ linux/include/sched.h)
- → pick_next_task → pick_next_task_idle to return the idle task from the run queue (rq → idle is always there)
- → dequeue_task → dequeue_task_idle
  just prints err_msg since it makes no sense to dequeue the idle task (never blocks, never sleeps)
- → enqueue_task = not implemented! So, it must not get called for idle scheduler.
- → put_prev_task → put_prev_task_idle = does nothing!
  this gets called from schedule() via put_prev() unconditionally, so we need to implement it



- Traditionally, a scheduler only worried about running a task on one CPU.
- Now, scheduler is expected to make decisions about multiple CPUs – variety of ways to do this:





We could just pick a task to run, whenever a CPU becomes available..

We could do it w/ one queue

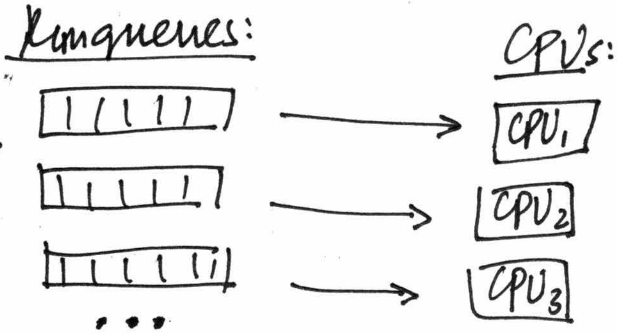
potentially OK, if we can live w/ one queue, but a problem w/ many CPUs.

BUT: Lock contention on the runqueue would be a huge problem with many CPUs (say 64) since timer interrupt usually goes off every 10 milliseconds usually. ❗

COMS W4118-13-5 10/17/2017

To avoid the lock contention, we use separate runqueues per CPU:

Modern Linux uses many runqueues ⟹



When a CPU needs to make a scheduling decision, it only looks at its own runqueue.

Jobs that become runnable can be put on:

- a random CPU runqueue (simple, but open to problems)
- CPU runqueue w/ least tasks (no lock if you don't need to be precise)
-

- /kernel/sched/core.c → one of the first things we do is pick the runqueue associated w/ the CPU: `rq = cpu_rq(cpu);`