Review notes

Start: Let's review the *whole thing*, from the ground up.

We turn on a computer. The operating system needs to get loaded up. What happens now?

- Answer: fault handlers get installed (we need this *very* early)
- Answer: the OS's page table gets initialized! (yup, the OS gets one too!)
- Answer: we map (into physical memory) certain critical pages, e.g. the OS code pages and [mapped] page table entries.

The order of these is specific to the architecture, they're the two *most-basic* things we need to have happen.

At some point, we read the disk and learn about the filesystem, so that we can find things on it. Filesystems are *not covered in this exam,* and will be reviewed later in the class.

We jump to the start of the OS's first code page! And... then, the virtual memory system takes over and runs the rest :)

Ok! The OS is ready to launch a process. What happens now?

- Answer: the OS sets up the *process control block*.
- Answer: the OS (often in the process control block, sometimes in other dedicated memory) sets up the *process page table*
- Answer: the OS maps the process code pages from disk directly into page table; page faults will bring them into memory when needed.
- Answer: the OS makes sure to page in [to physical memory] the first few (mapped) entries of the process's page table (including the page table itself!)
- Answer: the OS sets the status bit in the process-control-block to "Ready"
- Answer: The OS adds the pcb to the scheduler
- Answer: the OS runs the scheduler

The scheduler!

What does the scheduler do? Figure out which process to run next.

How does the scheduler run the process? Just... run it! **Limited Direct Execution**, as much as possible just jump to the first instruction of the process and you're off to the races.

Most important scheduler: the multi-level feedback queue

What is a multi-level feedback queue?

- Answer: a list of "round-robin" queues, in priority order

Follow-up: what is a "round-robin" queue?

- Answer: A queue that runs each process for a fixed (maximum) quantum; all processes in queue will run for first quantum before any process runs for second quantum

Follow-up: how do we ensure a process runs for a fixed maximum quantum?

- Answer: the process can just call yield
- Answer: we have a timer that causes the hardware to interrupt; the interrupt handler forces the process to yield

Back to MLFQ: The rules of the MLFQ

- 1: Take from the highest-priority queue
- 2: Use the highest-priority queue as a round-robin queue
- 3: Put new jobs at the highest-priority queue
- 4: Track the amount of time a process is *actually running*; if it yields before a fixed "quantum" then keep it at the same priority level. *the quantum is tracking is cumulative*. When a quantum has been exhausted (potentially after multiple runs through the RR queue), demote the process
- 5: Every now and then (another hardware timer), append all queues to end of high-priority queue. There's now just one queue.

Sidebar: How many timers are there in the hardware????

- Answer: we can only assume one.

Sidebar: how are we doing all these different timers then???

- Answer: one approach? Just have our *single timer* tick at fixed rate; when timer fires, OS checks if anyone needed to be interrupted / if MLFQ needed to be flattened. If not? just jump back to process

Sidebar: how to set timer?

- Answer: Set it to the *quantum* for which you want processes to (maximally) run without interruption. Make everything else a multiple of that.

Sidebar: is this how things really work?

- Answer: Sometimes! In Linux, *yes* until recently-enough that it was a problem for android phones. But a lot of modern systems are "tickless", and timers are just used as-needed. Gotta be a bit careful about how you set them!

We've launched a process. What now?

Process needs access to hardware / OS-managed resources (like... the ability to launch more processes). How do we provide that?

- Answer: *system calls*.

What is a system call?

- Answer: fancy library function
- Answer: changes *privilege mode*
- Answer: we call doing this a "trap", as in "trapping into the kernel"

What is the privilege mode?

- Answer: controls access to important registers (e.g the page-table base register, so controls all the memory you can access)
- Answer: controls access to important instructions (e.g. in RISC, setting / flushing the TLB)

What if we want to spawn more processes?

- Answer: the *fork* system call makes copy of your process
  - trace through how this works, copying the PCB, on the blackboard
- Answer: the *exec* system call replaces your process
  - trace through how this works, mucking up the page table, on the blackboard
- Answer: the *spawn* system call just does it all at once.

What if we want to swap what process is running?

- Answer: the scheduler decides this
- Answer: *flush the TLB*, store the registers (including the PBTR and PC) to the pcb, load registers from next proc's pcb. (This is slightly simplified).

...And that's about it!