4/8/24 Scheduling Wrap-up & Threads

Round Robin: FIFO w/ time quantum, preemptive

→ unfair to tasks that didn't use the full time quantum

blocks after 1 ms, other tasks run

10ms time slice — I/O bound / interactive ⇒ have to wait a long time compared to its run-time on CPU

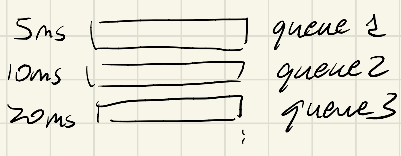
10ms time slice — CPU bound

→ Option 1: jobs w/ less time on the CPU are strictly prioritized.

→ CFS: linux default scheduler, time-ordered red black tree, schedules the task that spent least time on CPU, helps it catch up to its fair share of CPU time

→ Option 2: reduce wait time for I/O bound tasks

→ MLFQ: a number of RR queues w/ different time quantams

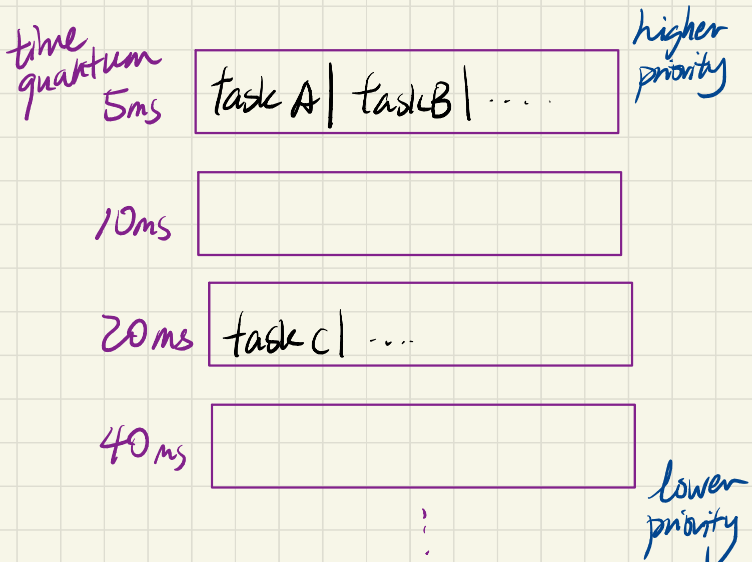


keep the I/O tasks in the top queue, shorter wait time

Multilevel Feedback Queue (MLFQ)

→ shorter time slice ⇒ shorter wait time, good for interactive jobs

→ longer time slice ⇒ less context switch for longer tasks, good for CPU bound tasks



- Scheduler runs tasks from the higher priority queue, if empty, goes to the next queue ...
- current task will be preempted if there are new tasks in higher priority queues.
- RR within a queue

How do we know which tasks go to which Q?

→ assumes all new tasks are short (top Q)

→ if blocks before time slice ⇒ same Q

→ if uses up the full time ⇒ down a Q

→ Starvation for long tasks?

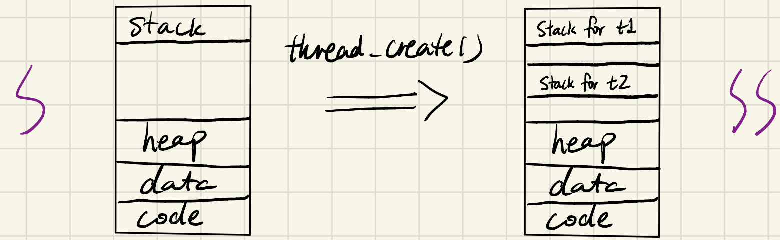
→ priority boost (periodically move all tasks to the top Q)

→ can this be gamed by injecting sleep?

→ set max time for a task per queue

Threads

- → unit of execution / task
  - → execution states : PC, SP, registers
- → multithreaded program (concurrent)
  - → divide program into tasks (threads)
- → Concurrency vs. parallelism → execute simultaneously
  - ↳ structured into tasks, tasks take turn making progress (concurrently)
  - * concurrency can happen on single core, kernel is always concurrent



Single threaded process — multithreaded (2) process

* each has its own user stack & kernel stack

Process = Address Space + OS resources + 1^+ threads

- → threads share code, heap, data, but have their own stack & execution state (PC, regs.)
- → managed & scheduled by the kernel → Thread Control Block (TCB)

→ Switching btwn threads = context switch

save current thread's context onto its kernel stack
switch to the next thread's kernel stack & pop the saved context
if the next thread is from a different process, load new address space & flush the TLB

xk: current thread → scheduler → next thread
(pick new thread to run)

Pthreads API

→ pthread_create(thread_func, args) ↗ PC

→ pthread_join (tid) wait for tid to exit, any thread can join another

→ pthread_exit(exit_status) terminate the calling thread.

→ pthread_detach upon exit, clean up resources (stack) automatically.
(does not require join)

Threads Execution

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	....	Thread is resumed.
		y = y + x;	....
		z = x + 5y;	z = x + 5y;

value of x may change in a multi-threaded process

timer Interrupt