Scheduling

key concepts: round robin, shortest job first, MLFQ, multi-core scheduling, cache affinity, load balancing

Lesley Istead

David R. Cheriton School of Computer Science
University of Waterloo

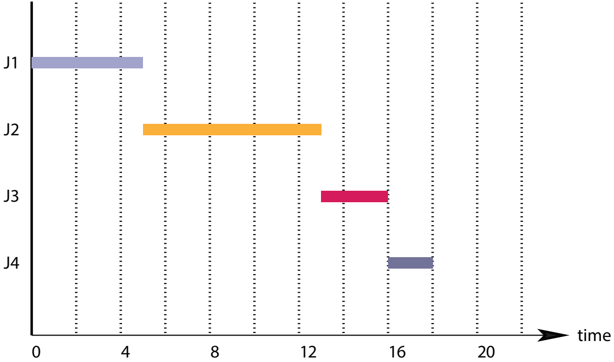
Spring 2021

Simple Scheduling Model

- We are given a set of **jobs** to schedule.
- Only one job can run at a time.
- For each job, we are given
  - job arrival time (a_i)
  - job run time (r_i)
- For each job, we define
  - **response time**: time between the job's arrival and when the job starts to run
  - **turnaround time**: time between the job's arrival and when the job finishes running.
- We must decide when each job should run, to achieve some goal, e.g., minimize average turnaround time, or minimize average response time.

First Come, First Served

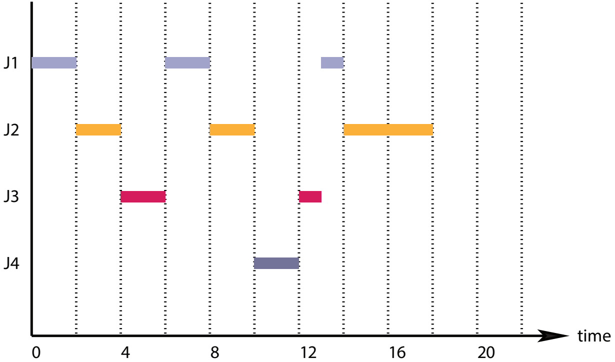
- jobs run in order of arrival
- simple, avoids starvation



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

Round Robin

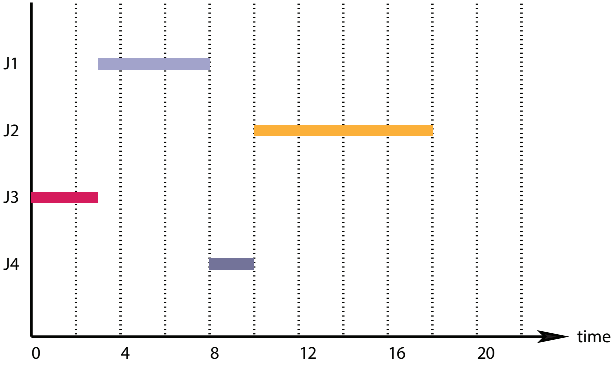
- preemptive FCFC
- OS/161's scheduler



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

Shortest Job First

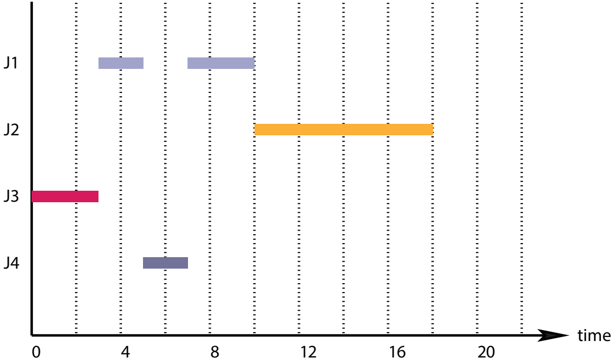
- run jobs in increasing order of runtime
- minimizes average **turnaround** time
- **starvation is possible**



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

Shortest Remaining Time First

- preemptive variant of SJF; arriving jobs preempt running job
- select one with shortest remaining time
- **starvation still possible**



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

CPU Scheduling

- In CPU scheduling, the "jobs" to be scheduled are the **threads**.
- CPU scheduling typically differs from the simple scheduling model:
  - the run times of threads are normally not known
  - threads are sometimes not runnable: when they are blocked
  - threads may have different priorities
- The objective of the scheduler is normally to achieve a balance between
  - responsiveness (ensure that threads get to run regularly),
  - fairness,
  - efficiency

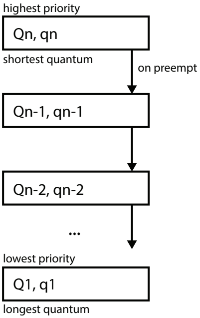
How would FCFS, Round Robin, SJF, and SRTF handle blocked threads? Priorities?

Multi-level Feedback Queues

- **the most commonly used scheduling algorithm in modern times**
- **objective**: good responsiveness for **interactive** threads, non-interactive threads make as much progress as possible
  - **key idea**: interactive threads are frequently blocked, waiting for user input, packets, etc.
- **approach**: given higher priority to interactive threads, so that they run whenever they are ready.
- **problem**: how to determine which threads are interactive and which are not?

MLFQ is used in Microsoft Windows, Apple macOS, Sun Solaris, and many more. It was used in Linux, but no longer is.

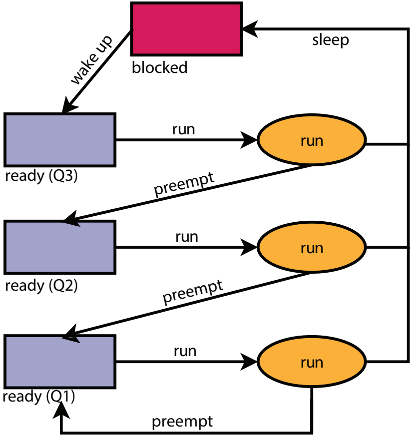
MLFQ Algorithm



- n round-robin ready queues where the priority of $Q_i > Q_j$ if $i > j$
- threads in Q_i use quantum q_i and $q_i \leq q_j$ if $i > j$
- scheduler selects a thread from the highest priority queue to run
    - threads in Q_{n-1} are only selected if Q_n is empty
- preempted threads are put onto the back of the next lower-priority queue
    - a thread from Q_n is preempted, it is pushed onto Q_{n-1}
- when a thread wakes after blocking, it is put onto the highest-priority queue

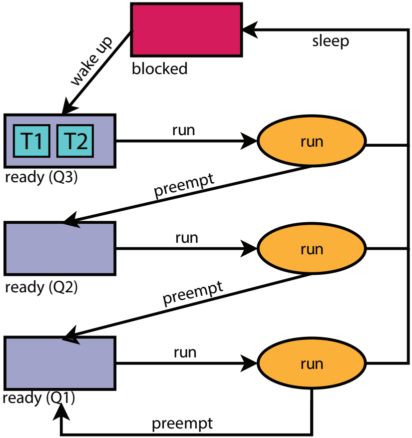
> Since interactive threads tend to block frequently, they will "live" in higher-priority queues while non-interactive threads sift down to the bottom.

3-Queue MLFQ Example



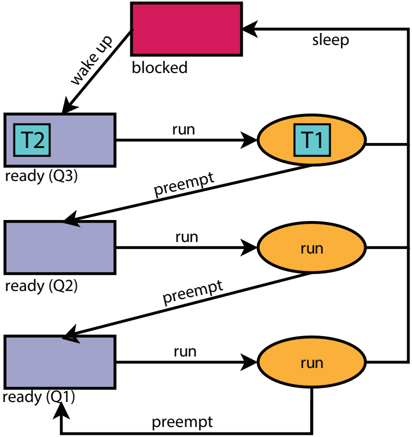
When do threads in $Q1$ run if $Q3$ is never empty?
To prevent starvation, all threads are periodically placed in the highest-priority queue.

3-Queue MLFQ Example



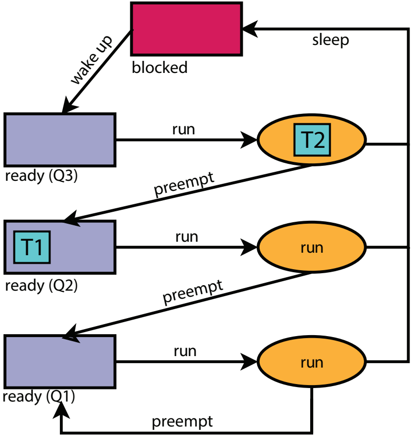
Two threads, T1 and T2, start in $Q3$.

3-Queue MLFQ Example



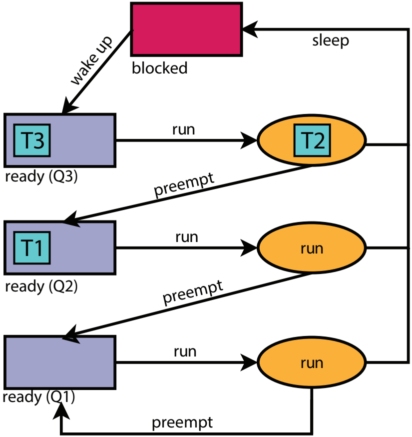
T1 is selected to run.

3-Queue MLFQ Example



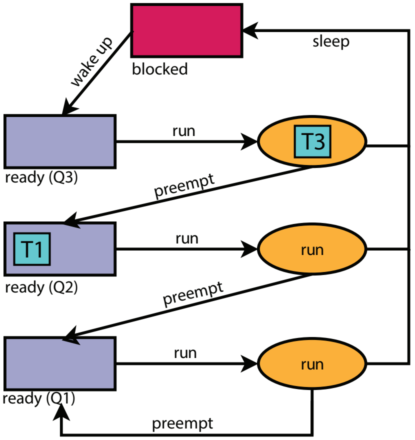
T1 is preempted and pushed onto the back of Q2. T2 is selected to run.

3-Queue MLFQ Example



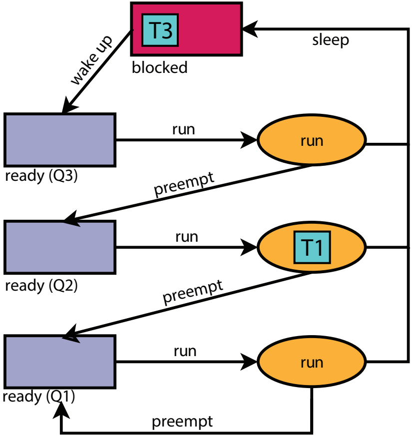
While T2 is running a new thread, T3, arrives.

3-Queue MLFQ Example



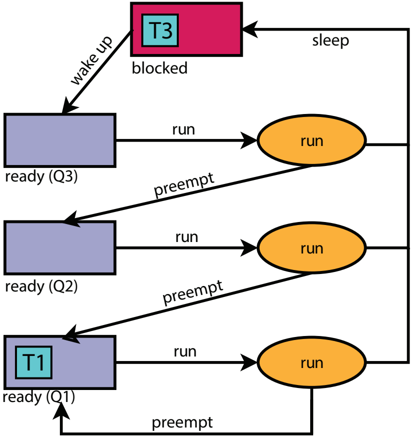
T2 terminates. T3 is selected.

3-Queue MLFQ Example



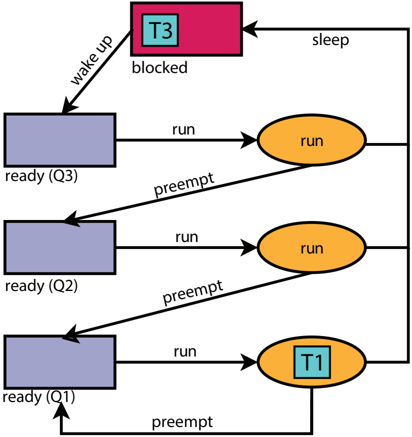
T3 blocks. T1 is selected.

3-Queue MLFQ Example



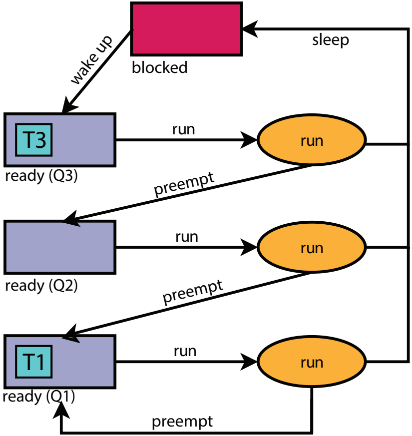
T1 is preempted, it is pushed onto Q1.

3-Queue MLFQ Example



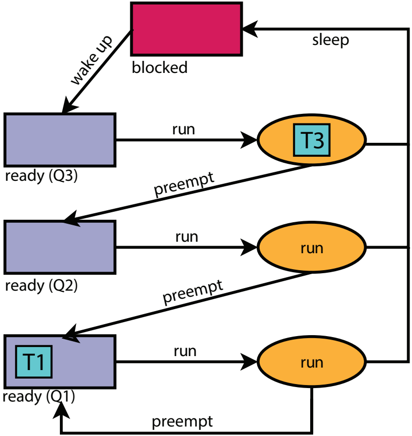
T1 is selected.

3-Queue MLFQ Example



T3 is woken by T1 causing T1 to be preempted. Many variants of MLFQ will preempt low-priority threads when a thread wakes to ensure a fast response to an event.

3-Queue MLFQ Example



T3 is selected.

Linux Completely Fair Scheduler (CFS) - Main Ideas

- each thread can be assigned a **weight**
- the goal of the scheduler is to ensure that each thread gets a "share" of the processor in proportion to its weight
- basic operation
  - track the "virtual" runtime of each runnable thread
  - always run the thread with the lowest virtual runtime
- virtual runtime is actual runtime adjusted by the thread weights
  - suppose w_i is the weight of the ith thread
  - actual runtime of ith thread is multiplied by $\frac{\sum_j w_j}{w_i}$
  - virtual runtime advances slowly for threads with high weights, quickly for threads with low weights

In MLFQ the quantum depended on the thread priority. In CFS, the quantum is the same for all threads and priorities.

CFS Example

Suppose the total weight of all threads in the system is 50 and the quantum is 5.

Time	Thread	Weight	Actual Runtime	Virtual Runtime
t	1	25	5	
	2	20	5	
	3	5	5	
$t+5$	1	25		
	2	20		
	3	5		

Which thread is selected at t? Which thread at $t+5$?

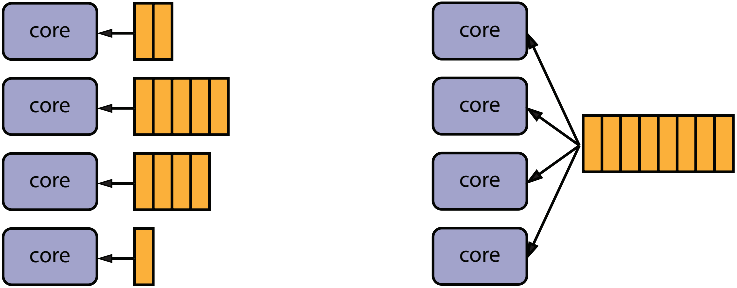
CFS Example

Suppose the total weight of all threads in the system is 50 and the quantum is 5.

Time	Thread	Weight	Actual Runtime	Virtual Runtime
$t+5$	1	25	5	$5 * 50/25 = 10$
	2	20	5	$5 * 50/20 = 12.5$
	3	5	5	$5 * 50/5 = 50$
				T1 is selected
$t+5$	1	25	10	$10 * 50/25 = 20$
	2	20	5	12.5
	3	5	5	50
				T2 is selected

Which thread is selected at t? Which thread at $t+5$?

Scheduling on Multi-Core Processors



per core ready queue vs. shared ready queue

Which offers better performance? Which one scales better?

Scalability and Cache Affinity

- Contention and Scalability
  - access to shared ready queue is a critical section, mutual exclusion needed
  - as number of cores grows, contention for ready queue becomes a problem
- per core design **scales** to a larger number of cores
- CPU cache affinity
  - as thread runs, data it accesses is loaded into CPU cache(s)
  - moving the thread to another core means data must be reloaded into that core's caches
  - as thread runs, it acquires an **affinity** for one core because of the cached data
  - per core design benefits from affinity by keeping threads on the same core
  - shared queue design does not

Load Balancing

- in per-core design, queues may have different lengths
- this results in **load imbalance** across the cores
  - cores may be idle while others are busy
  - threads on lightly loaded cores get more CPU time than threads on heavily loaded cores
- not an issue in shared queue design
- per-core designs typically need some mechanism for *thread migration* to address load imbalances
  - migration means moving threads from heavily loaded cores to lightly loaded cores