CSC 369

Operating Systems

Lecture 5: Scheduling

Today

- Scheduling
  - Goals
  - Types of schedulers
  - Scheduling algorithms

Review

- The OS manages resources for processes
- A process is an instance of a program in execution
  - Includes execution state and resources
- Processes may contain multiple threads
  - Threads within a process share resources
  - Synchronization is needed to coordinate sharing
- Now: how do processes/threads get scheduled?

What is processor scheduling?

- The allocation of processors to threads over time
  - This is the key to *multiprogramming*
    - We want to increase CPU utilization and job throughput by overlapping I/O and computation
    - Mechanisms:
      - thread states, thread queues
    - *Policies:*
      - Given more than one runnable thread, how do we choose which to run next?
      - When do we make this decision?

Let's start with a simple policy

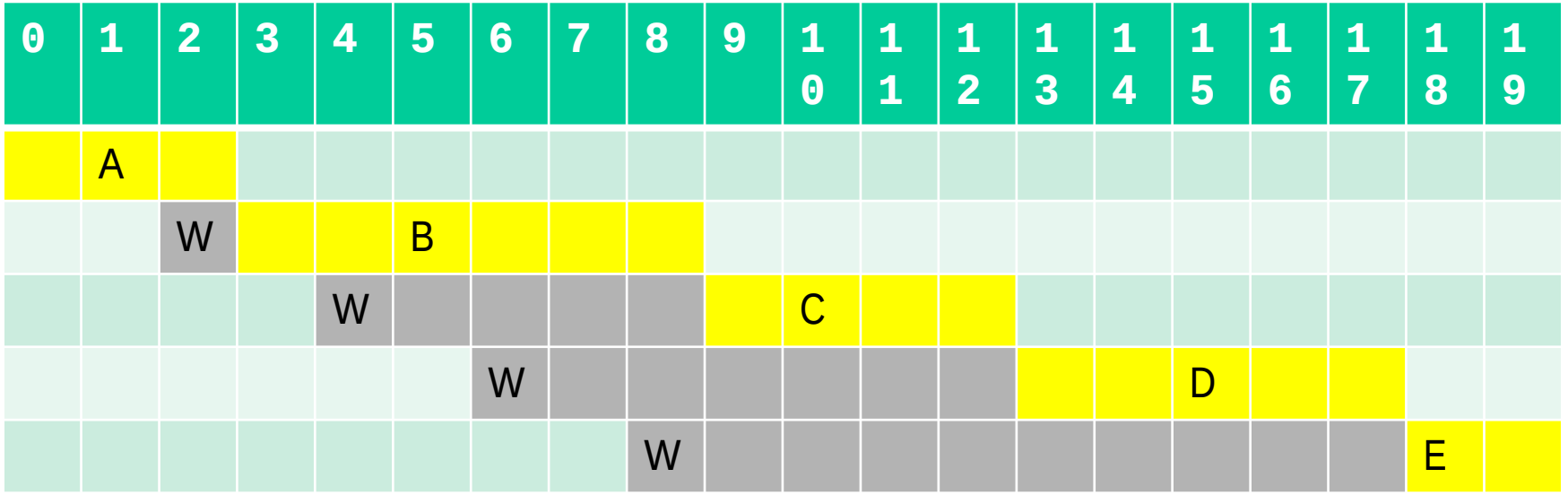
- Assume that processes run to completion when they are first scheduled

Do the first page of the exercise…

FCFS Example

Thread	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Note *E* waits five times as long as it runs!
  - Total run time is 20, total wait time is 23, average wait is 4.6
  - *How do we minimize average wait time?*



Shortest-Job-First Example

Thread	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Pick job that causes least wait time for others
  - Total run time is 20, total wait time is 18, average wait is 3.6

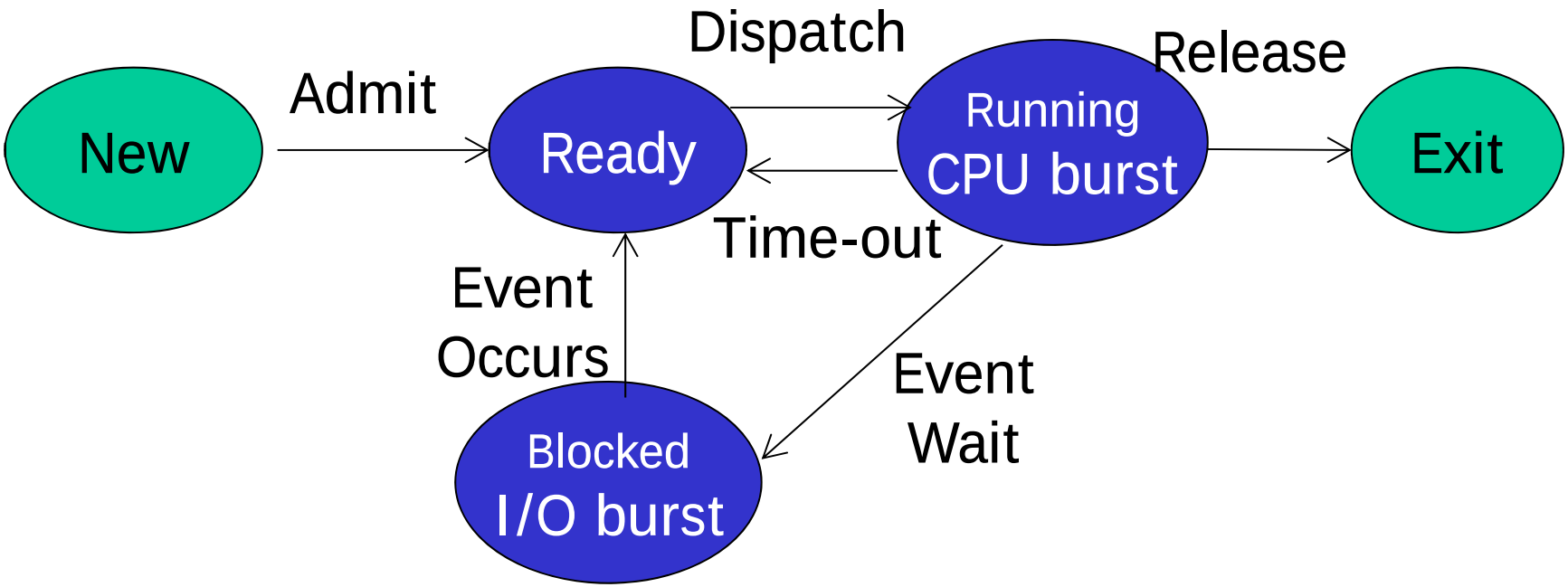
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	A																		
		W			B														
				W								C							
						W									D				
								W	E										

Thread Life Cycle

- Threads repeatedly alternate between computation and I/O
  - Called CPU bursts and I/O bursts
  - Last CPU burst ends with a call to terminate the thread (thread_exit() or equivalent)
  - CPU-bound: very long CPU bursts, infrequent I/O bursts
  - I/O-bound: short CPU bursts, frequent (long) I/O bursts
- During I/O bursts, CPU is not needed
  - Opportunity to execute another process!

Recall State Diagram of Lifecycle

- Thread/Process is blocked during I/O burst
  - Does not use CPU



Scheduling Goals

- All systems
  - Fairness - each thread receives fair share of CPU
  - Avoid starvation
  - Policy enforcement - usage policies should be met
  - Balance - all parts of the system should be busy
- Batch systems
  - Throughput - maximize jobs completed per hour
  - Turnaround time - minimize time between submission and completion
  - CPU utilization - keep the CPU busy all the time

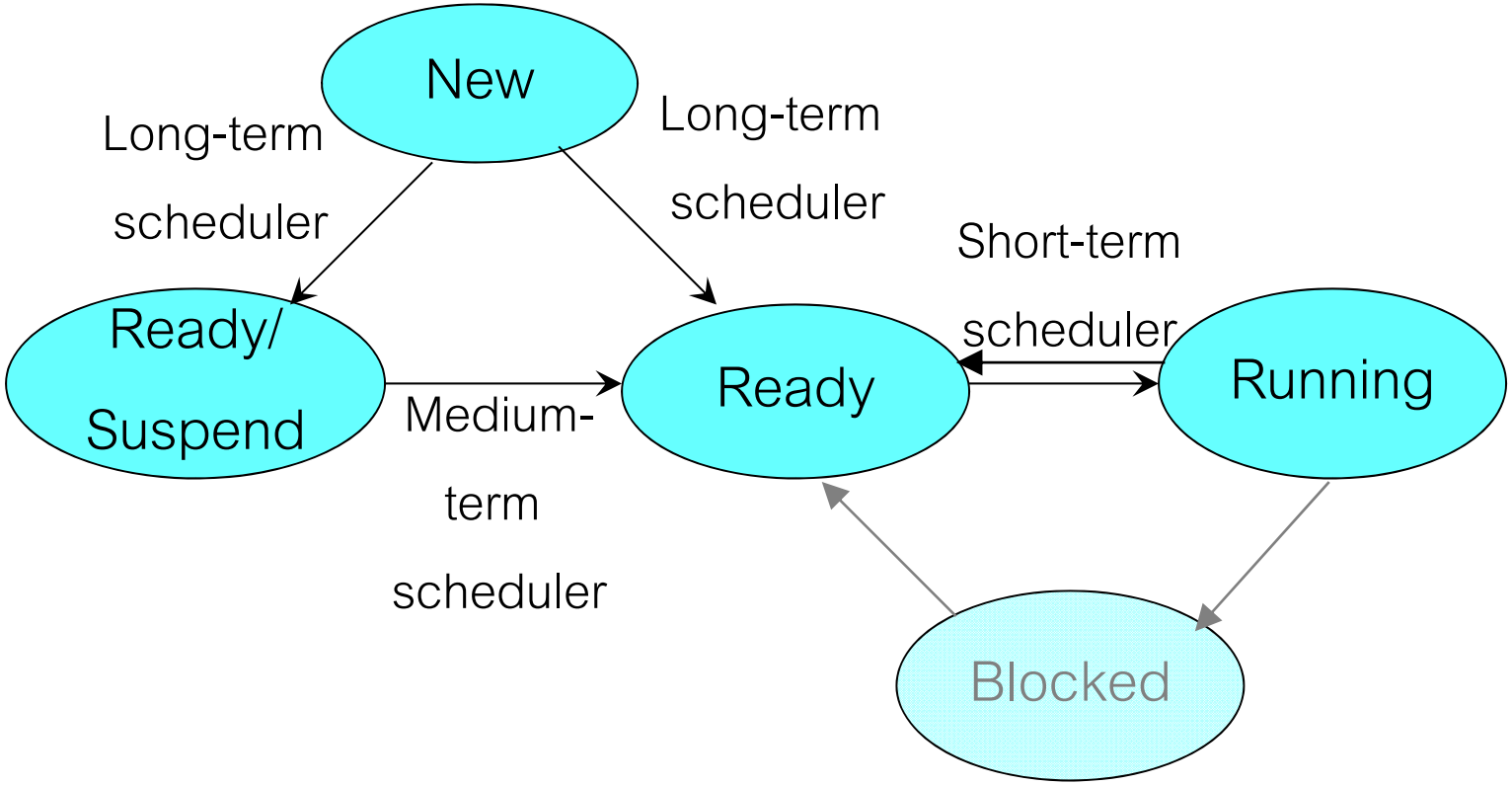
More Goals

- Interactive Systems
  - Response time - minimize time between receiving request and *starting* to produce output
  - Proportionality - “simple” tasks complete quickly
- Real-time systems
  - Meet deadlines
  - Predictability
- Goals sometimes conflict with each other!

Types of CPU Scheduling

- Long-term scheduling
  - aka *admission scheduling*, *admission control*
  - Used in batch systems, not common today
- Medium-term scheduling – happens infrequently
  - aka *memory scheduling*
  - Decides which processes are swapped out to disk
  - We'll talk about this later with memory management
  - Sometimes called "long-term" with admission control ignored
- Short-term scheduler – happens frequently
  - aka *dispatching*
  - Needs to be efficient (fast context switches, fast queue manipulation)

Thread/Process State Diagram



- Dispatching a process from the ready queue is often called *context switching*

What happens on *dispatch*?

- Select next thread from ready queue
- Save currently running thread state
  - Unless the current thread is exiting
- Restore state of next thread
  - Restore registers
  - Restore OS control structures
  - Switch to user mode (if required)
  - Set PC to next instruction in this thread

When to Schedule

- When a thread enters *Ready* state
  - thread creation (or *admission* in batch systems)
  - I/O interrupts (e.g. “I/O done”)
  - Signals (e.g. SIGCONT)
- When the running thread blocks (or exits)
  - Operating system calls (e.g., I/O)
  - Signals (e.g. SIGSTOP)
- At fixed intervals
  - Clock interrupts

Types of Scheduling

- Non-preemptive scheduling
  - once the CPU has been allocated to a thread, it keeps the CPU until it terminates or blocks
  - Suitable for batch scheduling
  - Common with user-level threads packages
- Preemptive scheduling
  - CPU can be taken from a running thread and allocated to another
  - Needed in interactive or real-time systems

Scheduling Algorithms: FCFS

- "First come, first served"
- Non-preemptive
- Choose the thread at the head of the ready queue
  - Queue is maintained in FIFO order
- Average waiting time with FCFS often quite long
  - convoy effect: all other threads wait for the one big thread to release the CPU

Algorithm: Shortest-Job-First

- aka Shortest Process Next, SJF or SPN
  - Pre-emptive version is “shortest remaining time”
- Choose the thread with the shortest expected processing time
  - Programmer estimate
  - History statistics
  - Can be shortest-next-CPU-burst for interactive jobs
- Provably optimal w.r.t. average wait time.

Algorithm: Round Robin

- Designed for time-sharing systems
- Pre-emptive
- Ready queue is circular
  - Each thread is allowed to run for time quantum q before being preempted and put back on queue
- Choice of quantum (aka time slice) is critical
  - as $q \rightarrow \infty$, RR $\rightarrow$ FCFS; as $q \rightarrow 0$, RR $\rightarrow$ processor sharing (PS)
  - we want q to be large w.r.t. the context switch time

Round Robin Example

Thread	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

When a process arrives, it gets placed in the ready queue at the end. A running process that finishes its time slice gets placed in the ready queue at the end too.

When a process finishes its time slice at the moment of arrival of another process – scheduling decision! Which one gets enqueued first? For this exercise, consider that the arriving process gets enqueued first over a process that just finished its slice.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	*Turn-around time*
A	A	W		A																5
		B	B				B	B							B	B				15
				W	C	C					C	C								9
						W			D	D							D	D	D	14
								W					E	E						7

Priority Scheduling

- A priority, *p*, is associated with each thread
- Highest priority job is selected from Ready queue
  - Can be pre-emptive or non-preemptive
- Enforcing this policy is tricky
  - A low priority task may prevent a high priority task from making progress by holding a resource (*priority inversion*)
  - A low priority task may never get to run (*starvation*)

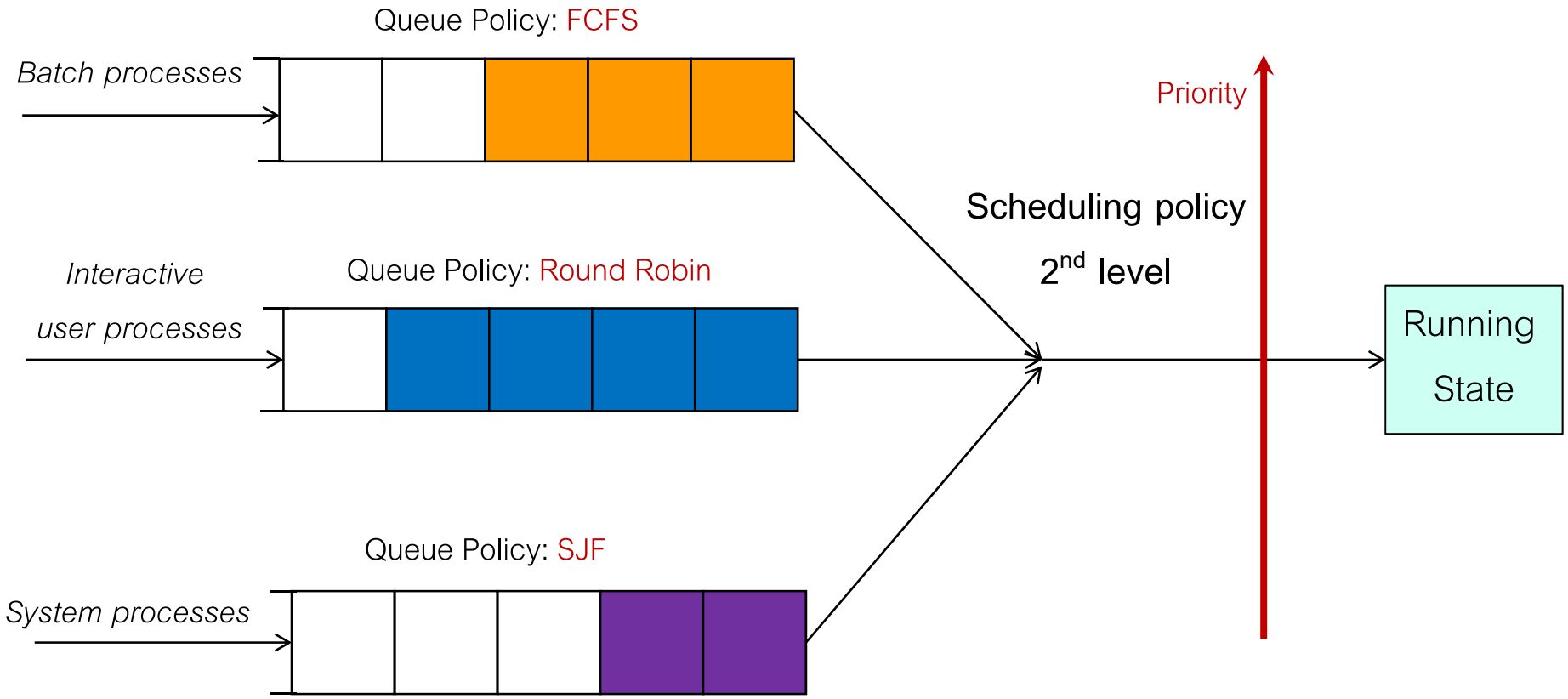
Priority Inversion Example

- Mars Rover *Pathfinder* bug
  - Shared "information bus" – essentially shared memory area
  - Mutual exclusion provided by lock on info bus
  - High priority "bus management" task moves data in/out of information bus, needs to run frequently
    - Watchdog timer resets system if bus mgmt task is not run
  - Low priority "data gathering" task writes data to the information bus, runs infrequently
  - Medium priority, compute-bound "communications" task that does not use the information bus
  - See the problem?
    - Data gathering task locks bus and is preempted by higher priority bus management task, which blocks on the lock. If communications task becomes runnable, data gathering task can't complete and release the lock so high priority task stays blocked.

Multi-Level Queue Scheduling

- Have multiple ready queues
  - Each runnable thread is on only one queue
- Threads are permanently assigned to a queue
  - Criteria include job class, priority, etc.
- Each queue has its own scheduling algorithm
  - Usually just RR
- Another level of scheduling decides which queue to choose next
  - Usually priority-based

Example


Queue Policy: FCFS
Batch processes
Priority
Scheduling policy
2nd level
Interactive
user processes
Queue Policy: Round Robin
Running
State
Queue Policy: SJF
System processes

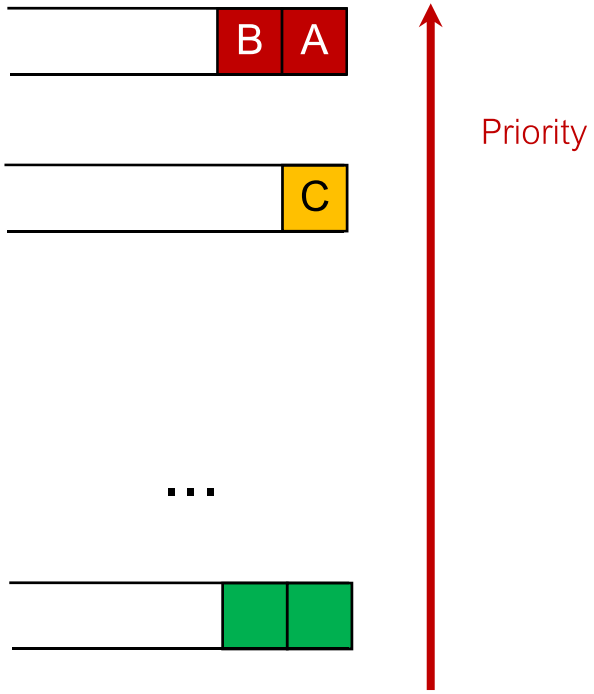
Feedback Scheduling

- Adjust criteria for choosing a particular thread based on past history
  - Can boost priority of threads that have waited a long time (*aging*)
  - Can prefer threads that do not use full quantum
  - Can boost priority following a user-input event
  - Can adjust expected next-CPU-burst
- Use a multi-level queue (MLQ) to move threads between queues
  - Use "feedback" to decide when to move to other queues
  - This is called *multi-level feedback queue (MLFQ)* – *ACM Turing Award!*

- *More examples/exercises in tutorial!*

Multi-level Feedback Queue (in a nutshell)

- A number of distinct queues, each one assigned a different priority level
- Each queue has multiple ready-to-run jobs
- The scheduler always chooses to run the jobs in the queue with the highest priority
- Jobs start in the highest priority queue
- FEEDBACK:
  - If a job uses an entire time slice, its priority gets reduced (gets moved to a lower queue)
  - If the job gives up the CPU before the timeslice is up, it stays at the same priority level
- What can happen?



- Remember to do the readings!

Fair Share Scheduling

- Group threads by user or department
- Ensure that each group receives a proportional share of the CPU
  - Shares do not have to be equal
- Priority of a thread depends on its own priority and past history of entire group
- Lottery scheduling variant - each group is assigned “tickets” according to its share
  - Hold a lottery to find next process to run

Unix CPU Scheduling

- interactive threads are favoured
  - small CPU time slices are given to threads by a priority algorithm that reduces to RR for CPU-bound jobs
- the more CPU time a thread accumulates, the lower its priority becomes (negative feedback)
- aging prevents starvation
- newer Unixes reschedule threads every 0.1 seconds and recompute priorities every second

More Unix Scheduling

- MLFQ with RR within each priority queue
- priority is based on process type and execution history

$$P_j(i) = base_j + [CPU_j(i-1)]/2 + nice_j$$

$$CPU_j(i) = U_j(i)/2 + CPU_j(i-1)/2$$

  - $P_j(i)$: priority of process j at beginning of interval I
    - lower values equal higher priorities
  - $base_j$: base priority of process j
  - $U_j(i)$: processor utilization of process j in interval i
  - $CPU_j(i)$: exponentially weighted average processor utilization by process j through interval i
  - $nice_j$: user-controllable adjustment factor (typically ranges from -20 to 19)

Linux (2.4) CPU Scheduling

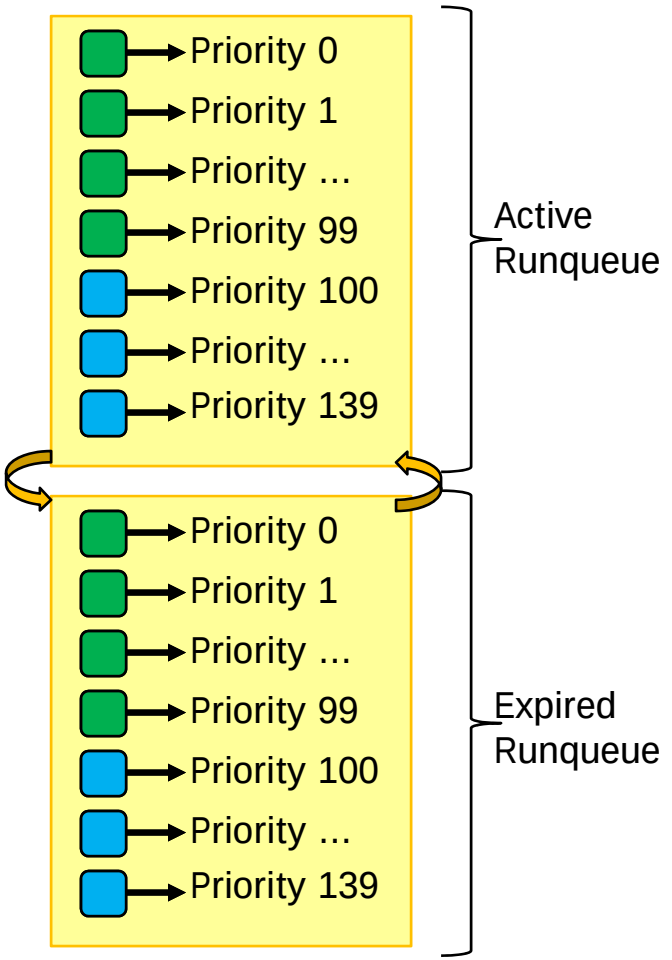
- 2 separate task scheduling algorithms
  - time-sharing and real-time tasks
- Time-sharing: use a prioritized, credit-based algorithm
  - the scheduler chooses task with the most credits
  - at every timer interrupt, the running task loses one credit
  - when its credits reach 0, it is suspended
  - if no runnable tasks have any credits, Linux performs a re-crediting, adding credits to *every* task:
    - credits = credits/2 + priority
- Problem: time for re-credit step proportional to number of processes, O(N)
  - In large scale systems, too much time spent making scheduling decisions

Improving the 2.4 Scheduler

- Re-credit step takes time proportional to the number of processes – O(N)
  - For large scale systems, spend too much time making scheduling decisions
- 2.5 kernel introduced O(1) scheduler
  - Each process gets a time quantum, based on its priority
  - Two arrays of runnable processes, *active* and *expired*
    - Processes are selected to run from the active array
    - When quantum exhausted, process goes on expired
    - When active is empty, swap the two arrays

Linux 2.6 Scheduler

- Based on O(1) scheduler introduced in 2.5
  - Maintains *active* and *expired* lists
- Multi-level feedback queue (MLFQ) scheme
  - 140 queues (and hence 140 priority levels)
    - 0-99 for "realtime"
    - 100-139 for "timesharing"
- Each priority level has its own timeslice
  - Higher priorities get longer timeslices
- A thread may run until it uses up its timeslice
  - After running, priority and timeslice are recalculated and thread is moved to the expired queues
  - When all active queues are empty, swap with expired.



Issues with O(1) Scheduler

- Process “nice” level has odd interaction with scheduler.
  - Niceness determines length of timeslice
    - 2 threads, normal niceness of 0 => 100ms slice
    - 2 threads, niceness of 19 (lowest pri) => 5ms slice
  - CPU share depends on absolute niceness, not relative values
    - T1, niceness 0 has 100ms, T2 niceness 1 has 95ms
    - T1, niceness 18 has 10ms, T2 niceness 19 has 5ms
- Adding waking threads to active queue can delay the switch to the expired queue

Linux CFS Scheduler

- CFS == Completely Fair Scheduling
  - Not really completely fair, but bounded unfairness
- Each thread accumulates “virtual runtime”
  - vruntime is actual runtime weighted by niceness
- Scheduler picks next thread as one with smallest vruntime
- Threads are organized in red-black tree
  - Thread with smallest vruntime is always left-most node
- Scheduler is now O(logN)

Windows CPU Scheduling

- dispatcher uses a 32-level MLFQ priority scheme
  - real-time class has threads with priorities 16 – 31
  - variable class has threads with priorities 0 - 15
- dispatcher traverses the queues from highest to lowest until it finds a ready thread
- when a variable class thread's quantum expires, its priority is lowered; when it is released from a wait, its priority is raised

More on Win NT Scheduling

- preemptive scheduler
- real-time class:
  - all threads have a fixed priority and at a given priority are in a RR queue
- variable class:
  - a thread's priority begins at some initial assigned value and then may change; there is a FIFO queue at each priority level
  - if it has used up its current time quantum, NT lowers its priority; if it is waiting on an I/O event, NT raises its priority (more for interactive waits than for other types of I/O waits)