PROCESS SCHEDULING

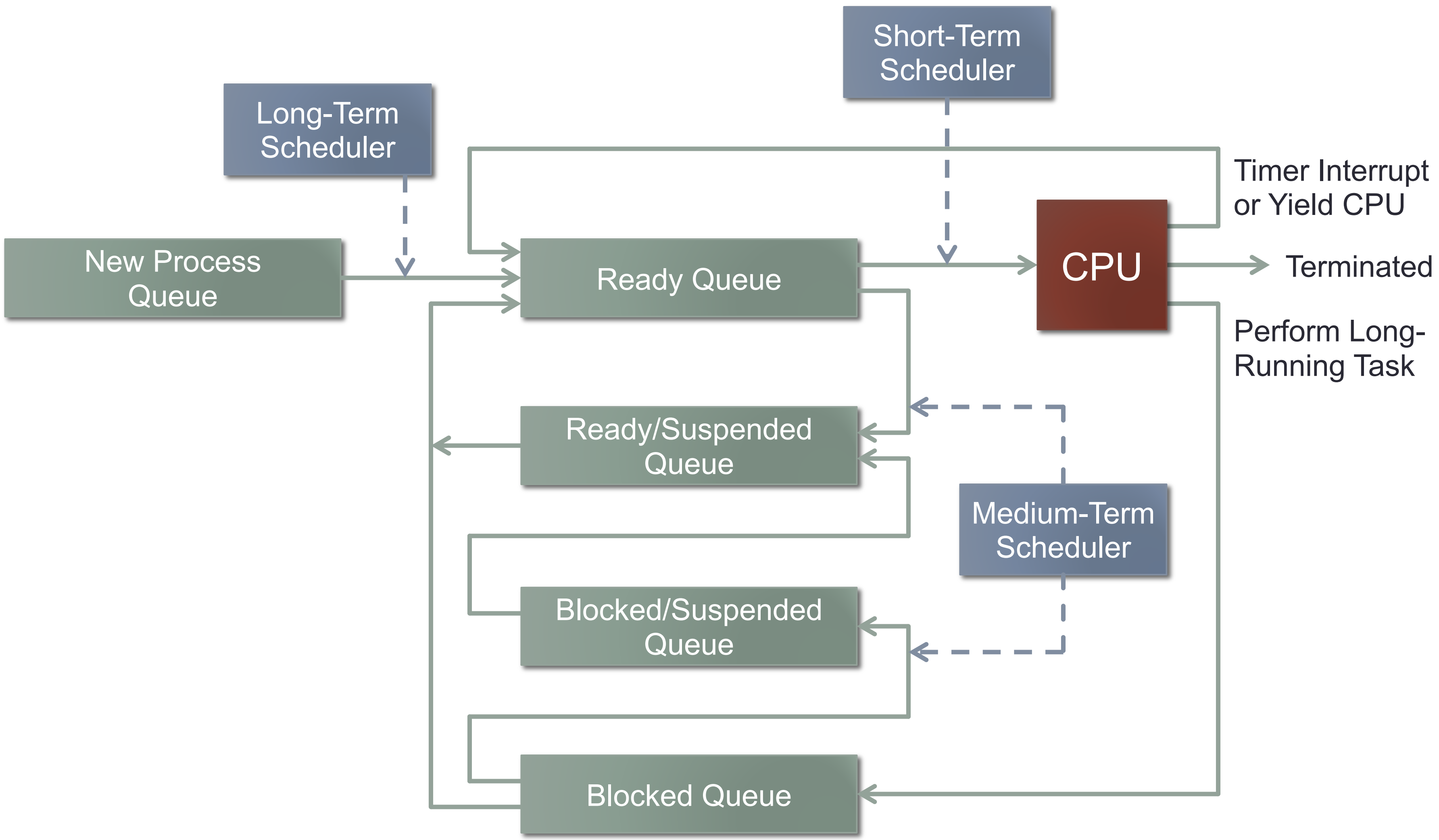
CS124 – Operating Systems

Spring 2024, Lecture 11

Process Scheduling

- Operating systems must manage the allocation and sharing of hardware resources to applications that use them
- Most important resource for multitasking OSes is the CPU
- We want to have multiple concurrently executing processes
  - While some processes are waiting for I/O, other processes can use CPU(s) in the system
- Processes fall into various categories based on their state
  - “Running” processes are on a CPU
  - “Ready” processes don’t have a CPU, but could run if they did (i.e. not blocked on I/O)
- How to allocate CPU time to the processes that can run?
  - Other process states couldn’t run even with the CPU; ignore them!

Process Scheduling: The Big Picture



Process Scheduling: Details

- Mainly focus on short-term scheduler, since this is what all OSes have
- The kernel schedules kernel threads, not processes
  - Scheduling occurs within the kernel, in kernel mode
  - The process' user context has already been saved at this point
- Scheduling and context-switching is always performed at a single point in the operating system kernel
  - e.g. a **`schedule()`** function always performs this task
- Kernel threads always see themselves as entering and exiting this **`schedule()`** function…
- In reality, this function is called by one kernel thread, then (usually) returns on a different kernel thread

Process Scheduling: Details (2)

- The `schedule()` function performs two important tasks:
  1. Choose the next kernel thread to run on the CPU
  2. Switch from the current kernel thread to the new kernel thread
     (if new kernel thread is same as old one, this is mostly a no-op)
- Second part is handled by the **dispatcher**:
  - Changes the CPU context to the new kernel thread
  - If new kernel thread has a user process/thread associated with it:
    - Sets up the user process' memory mapping, changes to user mode, and jumps to the appropriate point in the user process
- Dispatcher must execute as quickly as possible
  - This is pure overhead for the context-switch, and unavoidable

Switching Kernel Threads

- Switching between kernel threads involves <u>three</u> threads!
- Example: Pintos thread-switch function:

  `thread * switch_threads(thread *cur, thread *next)`

  - **`cur`** = thread we are switching away from
  - **`next`** = thread we are switching to
  - Function also returns a **`thread *`** – *why?*
- Example: switch from thread A to thread B
  - Kernel scheduler calls **`switch_threads(A, B)`**
  - This function switches from thread A's CPU context to thread B's CPU context (i.e. thread B's stack, registers, etc.)
- When thread B resumes running, what arguments does it see?
  - When thread B invoked **`switch_threads()`**, it was switching away from B to some <u>other</u> thread C: **`switch_threads(B, C)`**
- The context of thread A gets lost in the switch!

Switching Kernel Threads (2)

- Pintos thread-switch function:

  `thread * switch_threads(thread *cur, thread *next)`

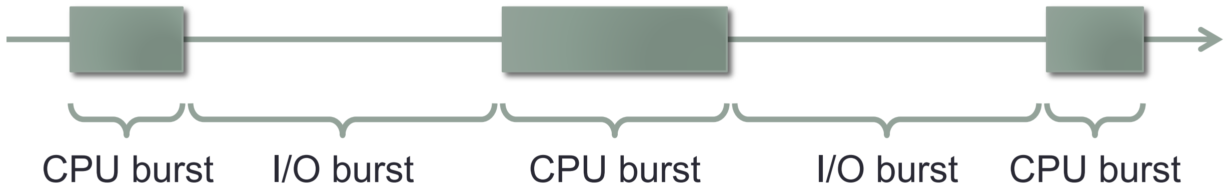
  - **`cur`** = thread we are switching away from
  - **`next`** = thread we are switching to
  - Function also returns a **`thread *`**
- When **`switch_threads()`** switches thread contexts, the current (old) context will be lost (i.e. **`cur`** is forgotten)
- <u>Very important</u> to retain the previous thread context:
  - If the old thread was terminating, need to release the thread's memory, remove it from other structures, etc.
- Before **`switch_threads()`** actually switches thread contexts, it ensures that the old context will be returned to the caller
  - e.g. Pintos saves old thread context into **`%eax`** to ensure it is returned, even though arguments will appear to change during context-switch

Scheduling Algorithms: Measurements

- Many different scheduling algorithms to choose from…
- Many measures to evaluate scheduling algorithms with
- **CPU utilization**: how busy are we keeping the CPU?
- **Throughput**: how many processes are completed in a given unit of time
- **Turnaround time**: how long to finish a given process?
    - This is wall-clock time: includes waiting on I/O, kernel overhead, …
- **Waiting time**: total time a process spends in ready state
    - i.e. the process <u>could</u> run, but it doesn't have an available CPU
- **Response time**: how quickly the process begins producing output
- Scheduling algorithms can optimize for different measures

Scheduling Characteristics

- Processes tend to be bursty in their behavior:



- Most CPU bursts are short, relatively few are long
  - Research usually characterizes the distribution as exponential
- Some schedulers are **nonpreemptive** or **cooperative**:
  - Only perform scheduling operation when the current process blocks, yields or terminates
  - Processes with long CPU bursts aren't preempted
- Other schedulers are **preemptive**:
  - Processes with long CPU bursts will be interrupted, to give other processes time to execute

First-Come First-Served Scheduling

- Simplest algorithm is **first-come first-served** (FCFS)
- Process ready-queue is a simple FIFO
  - (Sometimes called **FIFO scheduling**)
  - New processes are added to the end of the FIFO
  - Process at the front of the FIFO gets the CPU next
  - A process holds the CPU until it blocks, yields, or terminates
  - When it yields or is blocked, it goes to the end of the FIFO
- FCFS scheduling is non-preemptive!
- Generally an uninteresting scheduler
  - Sometimes appears in batch scheduling (needs a long-term scheduler to achieve a good process mix; even then, it's still bad)
  - Lack of preemption makes it undesirable in situations where processes may not terminate (i.e. the real world)
  - Terrible for time-sharing systems requiring high responsiveness

Round-Robin Scheduling

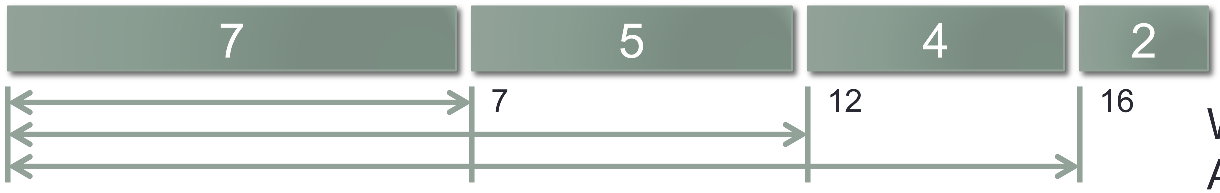
- Adding time-based preemption to FCFS scheduling produces **round-robin** (RR) scheduling
  - Processes get a fixed-size **time slice** or **time quantum** on CPU
- Again, process ready-queue is a simple FIFO
  - Current process runs until it blocks, yields or terminates, or it has used its entire time slice
  - When a process is moved off the CPU, it is put at end of run queue
  - Next process to receive the CPU is taken from front of the queue
- System responsiveness is directly affected by how large the time slice is chosen to be
  - Larger time slices are good for processes with large CPU bursts, but reduce system responsiveness
  - Interactive processes frequently have small CPU bursts, and won't get the CPU until compute-intensive processes are preempted

Shortest-Job-First Scheduling

- **Shortest-job-first** (SJF) scheduling orders processes based on how long their next CPU burst is expected to be
  - More accurate to call it "shortest-next-CPU-burst" scheduling…
- Minimizes the average waiting time of processes
- Example: 4 processes with varying CPU-burst times:
  - 2 units, 4 units, 5 units, 7 units
- Gantt Chart of shortest-job-first ordering:

Wait times: 0, 2, 6, 11
Average wait time: 4.75

- Longest job first (for comparison):



Wait times: 0, 7, 12, 16
Average wait time: 8.75

Shortest-Job-First Scheduling (2)

- Biggest challenge with shortest-job-first scheduling: Predicting the length of processes' next CPU burst!
- Usually the next CPU burst length is predicted using historical data
- Common: use **exponential average** of previous bursts
  - t_n = actual length of CPU burst n
  - τ_{n+1} = predicted burst length of burst $n+1 = \alpha\, t_n + (1 - \alpha)\, \tau_n$
  - τ_n encapsulates history of previous CPU burst lengths
  - α $(0 \le \alpha \le 1)$ weights contributions of recent history and past history
- Expanding:
  - $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n = \alpha\, t_n + (1 - \alpha)\, (\alpha\, t_{n-1} + (1 - \alpha)\, \tau_{n-1}) = \ldots$
  - $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + (1 - \alpha)^2\, \alpha\, t_{n-2} + \ldots + (1 - \alpha)^{n+1}\, \tau_0$
  - τ_0 is initial guess of first CPU burst length

Shortest-Job-First Scheduling (3)

- Shortest-job-first scheduling can be preemptive or non-preemptive
- If preemptive, called **shortest-remaining-time-first** scheduling
  - If a new job is added to the ready queue with a shorter time, it preempts the current job on the processor
- Shortest-job-first scheduling can have **starvation** issues
  - Some ready processes may <u>never</u> receive the CPU
- Scenario:
  - Ready queue contains short jobs and long jobs
- If new short jobs are continually added to the queue:
  - Will continually receive the CPU before longer running jobs

Priority Scheduling

- Shortest-job-first is an example of **priority scheduling**
  - In SJF, the shortest job has the highest priority
- Can also assign processes fixed priorities
- Process priority is usually represented as a number
  - Varies whether higher or lower numbers correspond to high priority
- Priority scheduling can be preemptive or non-preemptive
  - If non-preemptive, a new higher-priority process added to ready queue won't take the CPU from a lower-priority running process
  - If preemptive, a new higher-priority process added to ready queue immediately takes the CPU from a lower-priority running process
- Usually, <u>no</u> time limit is enforced on processes
  - Process holds the CPU until it blocks, yields or terminates.
  - (Or, if preemptive priority scheduling, a higher priority process is added to the ready queue)

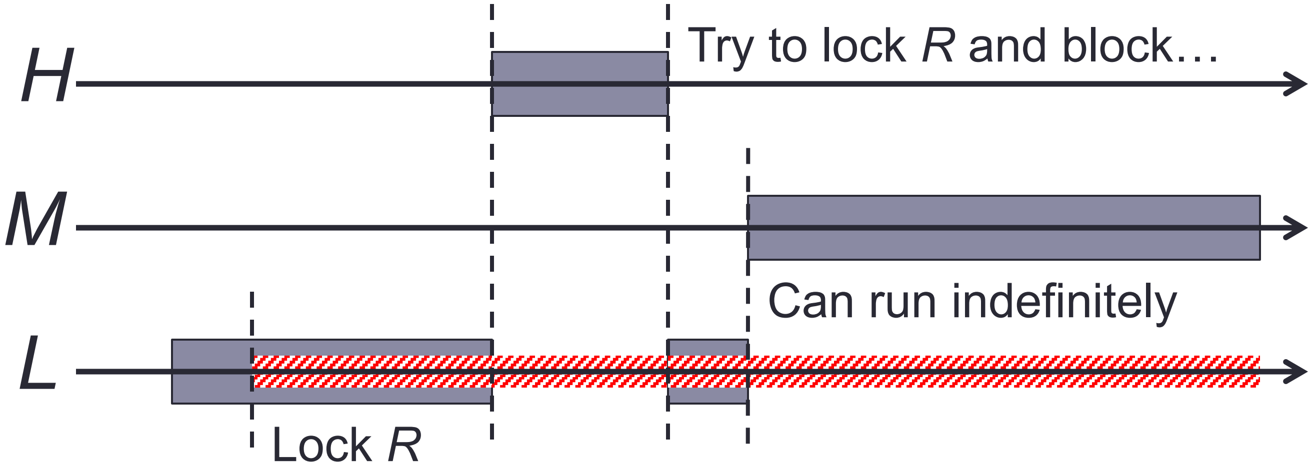
Priority Scheduling (2)

- Priority scheduling is also vulnerable to starvation
  - If high-priority processes are always able to run, lower-priority ready processes will never receive the CPU ☹
- Can solve this problem with **aging**:
  - Slowly increase priority of waiting processes until they finally receive the CPU
  - (Aging is sometimes used in other scheduling algorithms as well)

- Priority scheduling can also suffer from **priority inversion**
  - Higher-priority processes are supposed to preempt lower-priority process…
  - Sometimes, in the context of resource locking, a lower-priority process can preempt a higher-priority process

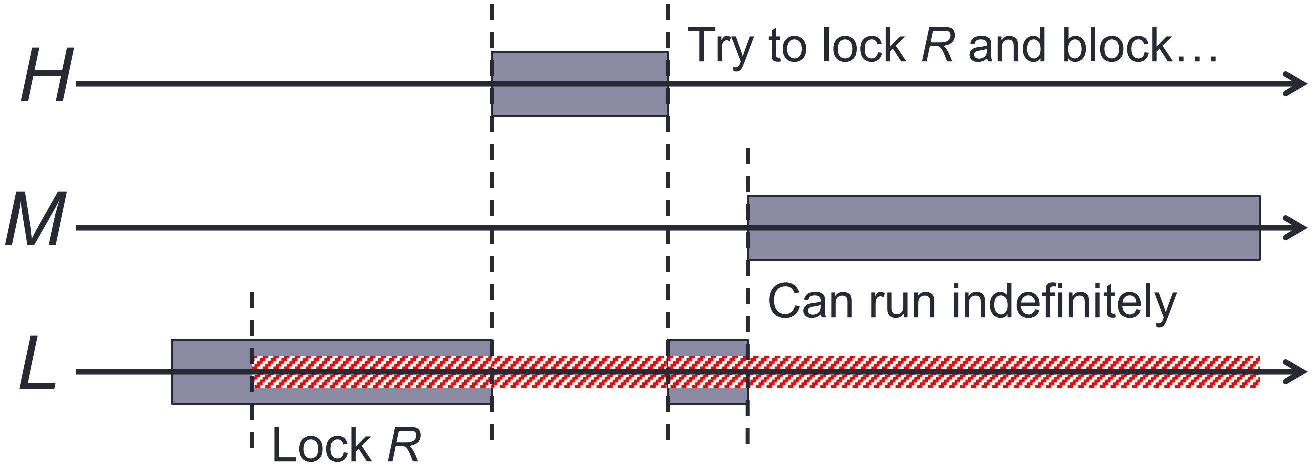
Priority Inversion

- A simple scenario:
  - Low-priority process *L* starts running, and locks shared resource *R*.
  - High-priority process *H* starts running, preempting *L*. (But *L* still holds resource *R*.)
  - *H* needs resource *R*, and attempts to lock it. *H* blocks; *L* resumes.
  - Medium-priority process *M* starts running, preempting *L*. *M* doesn't need *R*, and it continues to run as long as it likes.



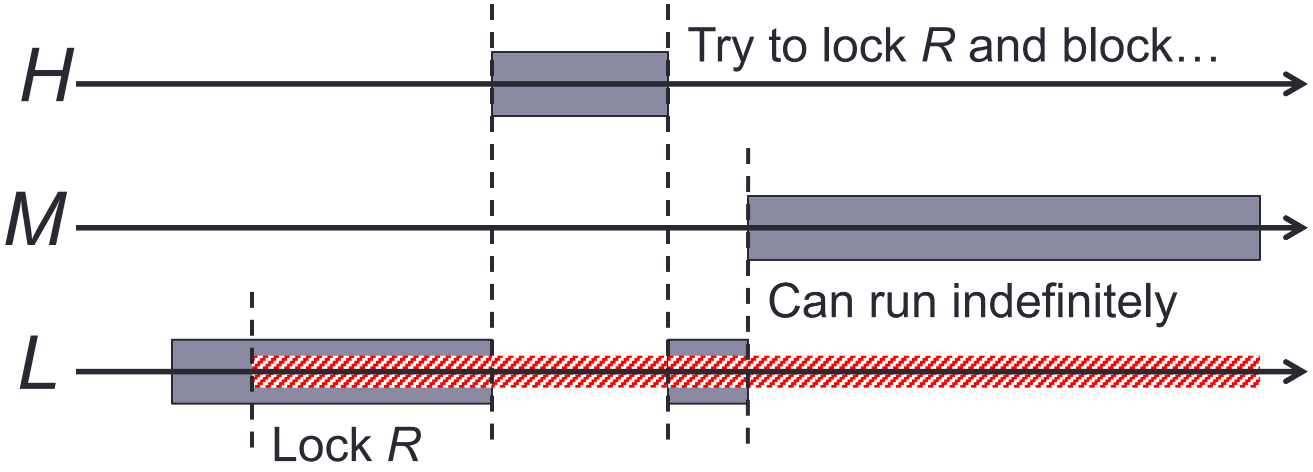
Priority Inversion (2)

- Because *L* is preempted by *M*, it can never finish and release *R* so that *H* can resume its execution.
- Because high-priority processes often carry out system-critical tasks, frequently has very serious consequences



Priority Inversion (3)

- A widely known example: Mars Pathfinder (1997)
  - High-priority process responsible for resetting a watchdog timer
  - High- and low-priority processes shared a lockable resource
  - Medium-priority processes prevented high-priority task from running, causing the spacecraft to reset frequently



Priority Inversion: Solutions

- Several solutions to priority inversion issue
- **Random boosting** (Microsoft Windows)
  - The scheduler randomly boosts the priority of waiting processes to “nudge” the system out of priority inversion
- **Priority ceiling** protocols
  - Every lockable resource is assigned a priority ceiling: the highest priority of any process allowed to lock it
  - When a process acquires the resource, its priority is raised to the resource’s priority ceiling until it unlocks the resource
- **Priority inheritance** (aka **priority donation**) protocols
  - If a high-priority process *H* is blocked waiting for a resource held by a low-priority process *L*, *H* temporarily donates its priority to *L*
  - A process’ priority is the maximum of its own priority, and the priorities of all processes it is currently blocking

Priority Donation

- Priority donation has its own issues
- Frequently, blocked processes can form a chain
  - Process 1 locks resource R1.
  - Process 2 locks R2, then attempts to lock R1, and blocks.
  - Process 3 locks R3, then attempts to lock R2, and blocks.
  - Process 4 locks R4, then attempts to lock R3, and blocks.
  - Process 5 attempts to lock R4, and blocks.
- Each process must donate its priority to all processes it is blocked on
  - Significantly increases the overhead of the priority scheduler
  - (This is why the Mars Pathfinder was sent to Mars with priority donation turned off…)

Priority Donation (2)

- Priority donation also fails in the context of deadlock
  - Process 1 locks R1.
  - Process 2 locks R2.
  - Process 1 attempts to lock R2, and blocks. Process 1 donates its priority to Process 2.
  - Process 2 attempts to lock R1, and blocks. Process 2 donates its priority to … ?
- Now the graph of waiting processes has a cycle in it
  - If the kernel naively follows edges in this graph, it will loop forever
  - Can make priority donation mechanism detect deadlocks in various ways, but (again) increases the overhead of donation

Multilevel Queue Scheduling

- Processes can often be categorized based on their purpose and behavior, e.g.
  - System processes
  - Interactive processes
  - Interactive editing processes
  - Batch processes
- Additionally, divide processes into two main categories: foreground processes and background processes
  - Foreground processes need responsiveness, and generally have small CPU bursts
  - Background processes have large CPU bursts, and aren't interactive
- **Multilevel queue scheduling** maintains a queue for each category of process
  - Queues have a decreasing priority – e.g. system processes are highest priority, batch processes are lowest priority
  - Processes are permanently assigned to a specific queue when they are started, and are not moved between different queues

Multilevel Queue Scheduling (2)

- Process categories and priorities
  - System processes (highest)
  - Interactive processes (high)
  - Interactive editing processes (medium)
  - Batch processes (low)
- Each queue has its own fixed priority
- Usually, high-priority queues <u>always</u> preempt low-priority
  - As long as there are system processes ready to run, they run first!
  - Interactive processes only run when no system processes can run
  - etc.
  - Batch processes only run if <u>no</u> other processes are ready to run
- Also possible to divide CPU time across subset of queues
  - e.g. spend 80% of CPU time running interactive processes, 20% running batch processes

Multilevel Queue Scheduling (3)

- Process categories and priorities
  - System processes (highest)
  - Interactive processes (high)
  - Interactive editing processes (medium)
  - Batch processes (low)
- Each queue can also have its own scheduling algorithm and parameters (e.g. time-slice size)
  - Batch processes can be run with first-come first-served scheduling, or round-robin with a very large time-slice (for runaway processes)
  - Other processes typically run with round-robin scheduling
  - Might also have real-time processes in a high-priority queue, using real-time scheduling algorithms for that queue

Multilevel Feedback Queue Scheduling

- Multilevel queue scheduling isn't very flexible
  - A process' behavior can easily change from foreground to background, or vice versa
  - Examples: MATLAB, Photoshop, media transcoding interface
    - Programs have user interfaces for interactive editing, etc.
    - Also run large compute-intensive tasks with long CPU bursts
- **Multilevel feedback queue** scheduling allows processes to move between the different priority queues
- Goals:
  - Favor short jobs (i.e. processes with short CPU bursts)
    - Premise: approximate shortest-jobs-first scheduling
  - Favor processes that frequently block on I/O
    - Premise: these processes may be interactive, and therefore require increased responsiveness
  - Separate processes based on their observed runtime behavior

Multilevel Feedback Queues (2)

- As before, multiple FIFOs are maintained for processes
  - Each FIFO has its own priority
  - Processes in higher priority queues preempt lower priority queues
  - Frequently, all queues are scheduled using round-robin scheduling, with shorter time-slices for higher-priority queues
- New processes are added to end of highest priority queue
  - Eventually reach the front of the queue and are scheduled
- If a process is preempted by the system, it is sent to the next lower queue
  - If process yields or blocks then it goes to end of the same queue
- Lower-priority processes can also be promoted for good behavior ☺
  - i.e. frequently yields or blocks within time-slice of next higher queue

Multilevel Feedback Queues (3)

- A lot of flexibility in design of multilevel feedback queues:
  - How many queues to manage in the scheduler
  - Scheduling algorithm to use for each queue, or groups of queues
    - (including configuration details such as time-slice size)
  - How to assign a process to an initial queue
  - How to decide when to demote a process to the next lower queue
  - How to decide when to promote a process to the next higher queue

- Because of this flexibility, multilevel feedback queues are widely used in modern operating systems

Multilevel Feedback Queues (4)

- Windows NT/Vista/7 has 32 queues in the scheduler
  - Levels 0-15 are "normal" priorities
  - Levels 16-31 are "soft real-time" priorities
- Mac OS X has multiple queues for threads, falling into four priority bands:
  - Normal (lowest priority), system high priority, kernel mode only, real-time threads (highest priority)
  - Threads cannot move outside their priority bands
- FreeBSD and NetBSD both maintain >200 queues, divided into different categories
- Solaris uses 170 queues, divided into various categories
- Linux used a multilevel feedback queue up to 2.4 kernel…

Next Time

- Continue coverage of process scheduling:

- Real-time scheduling
- More recent Linux schedulers