Announcements

- Program #1
  - Due 2/15 at 5:00 pm
- Reading
  - Finish scheduling
  - Process Synchronization:
    - Chapter 6 (8^{th} Ed) or Chapter 7 (6^{th} Ed)

Scheduling criteria

- Per processor, or system oriented
  - CPU utilization
    - maximize, to keep as busy as possible
  - throughput
    - maximize, number of processes completed per time unit
- Per process, or user oriented
  - turnaround time
    - minimize, time of submission to time of completion.
  - waiting time
    - minimize, time spent in ready queue - affected solely by scheduling policy
  - response time
    - minimize, time to produce first output
    - most important for interactive OS

Short-term scheduling algorithms

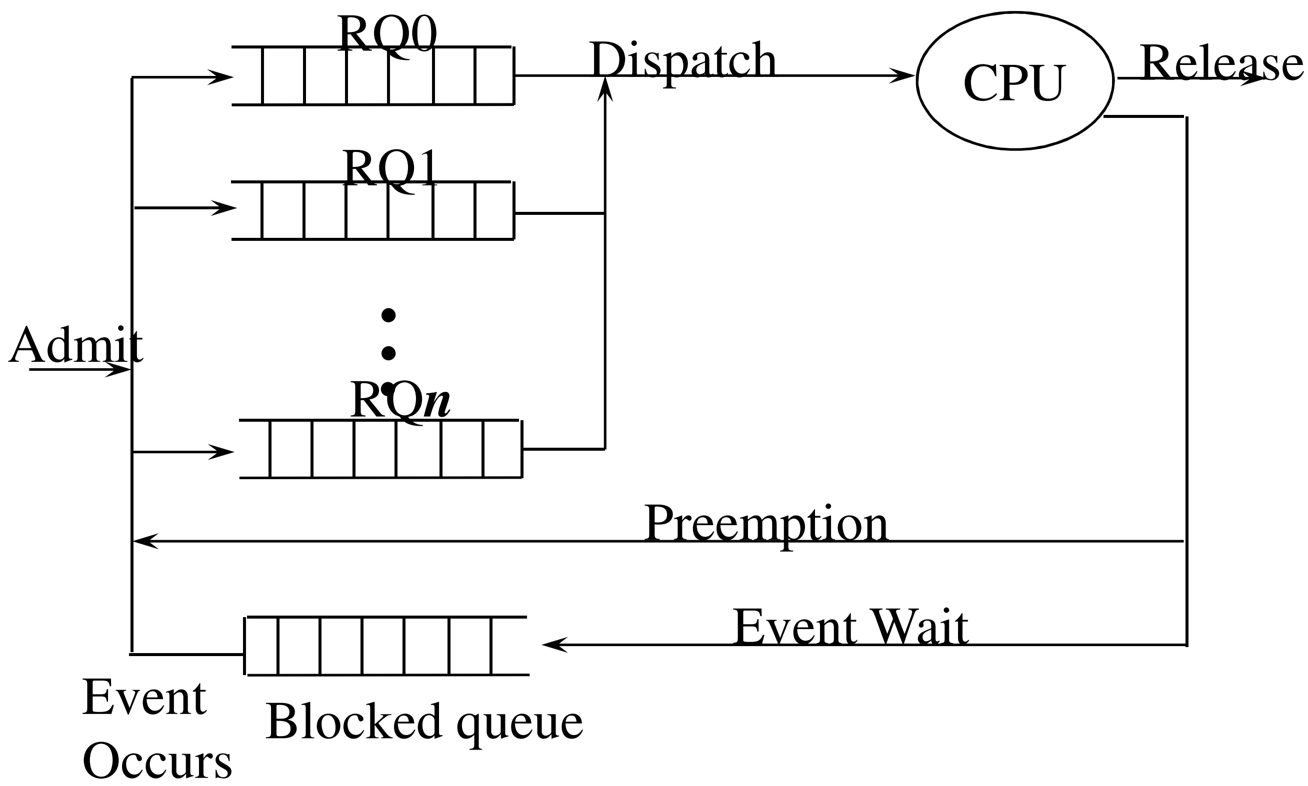
- First-Come, First-Served (FCFS, or FIFO)
  - as process becomes ready, join Ready queue, scheduler always selects process that has been in queue longest
  - better for long processes than short ones
  - favors CPU-bound over I/O-bound processes
  - need priorities, on uniprocessor, to make it effective

Algorithms (cont.)

- Round-Robin (RR)
  - use preemption, based on clock - time slicing
    - generate interrupt at periodic intervals
  - when interrupt occurs, place running process in Ready queue, select next process to run using FCFS
  - what's the length of a time slice
    - short means short processes move through quickly, but high overhead to deal with clock interrupts and scheduling
    - guideline is time slice should be slightly greater than time of "typical job" CPU burst
  - problem dealing with CPU and I/O bound processes

Priority Based Scheduling

- Priorities
  - assign each process a priority, and scheduler always chooses process of higher priority over one of lower priority
- More than one ready queue, ordered by priorities



Priority Algorithms

- Fixed Queues
  - processes are statically assigned to a queue
  - sample queues: system, foreground, background
- Multilevel Feedback
  - processes are dynamically assigned to queues
  - penalize jobs that have been running longer
  - preemptive, with dynamic priority
  - have ***N*** ready queues (RQ0-RQ***N***),
    - start process in RQ0
    - if quantum expires, moved to i + 1 queue

Feedback scheduling (cont.)

- problem: turnaround time for longer processes
  - can increase greatly, even starve them, if new short jobs regularly enter system
- solution1: vary preemption times according to queue
  - processes in lower priority queues have longer time slices
- solution2: promote a process to higher priority queue
  - after it spends a certain amount of time waiting for service in its current queue, it moves up
- solution3: allocate fixed share of CPU time to jobs
  - if a process doesn't use its share, give it to other processes
  - variation on this idea: lottery scheduling
    - assign a process "tickets" (# of tickets is share)
    - pick random number and run the process with the winning ticket.

UNIX System V

- Multilevel feedback, with
  - RR within each priority queue
  - 10ms second preemption
  - priority based on process type and execution history, lower value is higher priority
- priority recomputed once per second, and scheduler selects new process to run
- For process j, P(i) = Base + CPU(i-1)/2 + nice
  - P(*i*) is priority of process *j* at interval *i*
  - Base is base priority of process *j*
  - CPU(*i*) = U(*i*)/2 + CPU(*i*-1)/2
    - U(*i*) is CPU use of process *j* in interval *i*
    - exponentially weighted average CPU use of process *j* through interval *i*
  - nice is user-controllable adjustment factor

UNIX (cont.)

- Base priority divides all processes into (non-overlapping) fixed bands of decreasing priority levels
  - swapper, block I/O device control, file manipulation, character I/O device control, user processes
- bands optimize access to block devices (disk), allow OS to respond quickly to system calls
- penalizes CPU-bound processes w.r.t. I/O bound
- targets general-purpose time sharing environment

Example: Windows NT/XP

- Target:
  - single user, in highly interactive environment
  - a server
- preemptive scheduler with multiple priority levels
- flexible system of priorities, RR within each, plus dynamic variation on basis of current thread activity for ***some*** levels
- 2 priority bands, real-time and variable, each with 16 levels
  - real-time ones have higher priority, since require immediate attention(e.g. communication, real-time task)

Windows NT/XP (cont.)

- In real-time class, all threads have fixed priority that never changes
- In variable class, priority begins at an initial value, and can change, up or down
  - FIFO queue at each level, but thread can switch queues
- Dynamic priority for a thread can be from 2 to 15
  - if thread interrupted because time slice is up, priority lowered
  - if interrupted to wait on I/O event, priority raised
  - favors I/O-bound over CPU-bound threads
  - for I/O bound threads, priority raised more for interactive waits (e.g. keyboard, display) than for other I/O (e.g. disk)

Multi-Processor Scheduling

- Multiple processes need to be scheduled together
  - Called gang-scheduling
  - Allowing communicating processes to interact w/o/ waiting
- Try to schedule processes back to same processor
  - Called affinity scheduling
    - Maintain a small ready queue per processor
    - Go to global queue if nothing local is ready

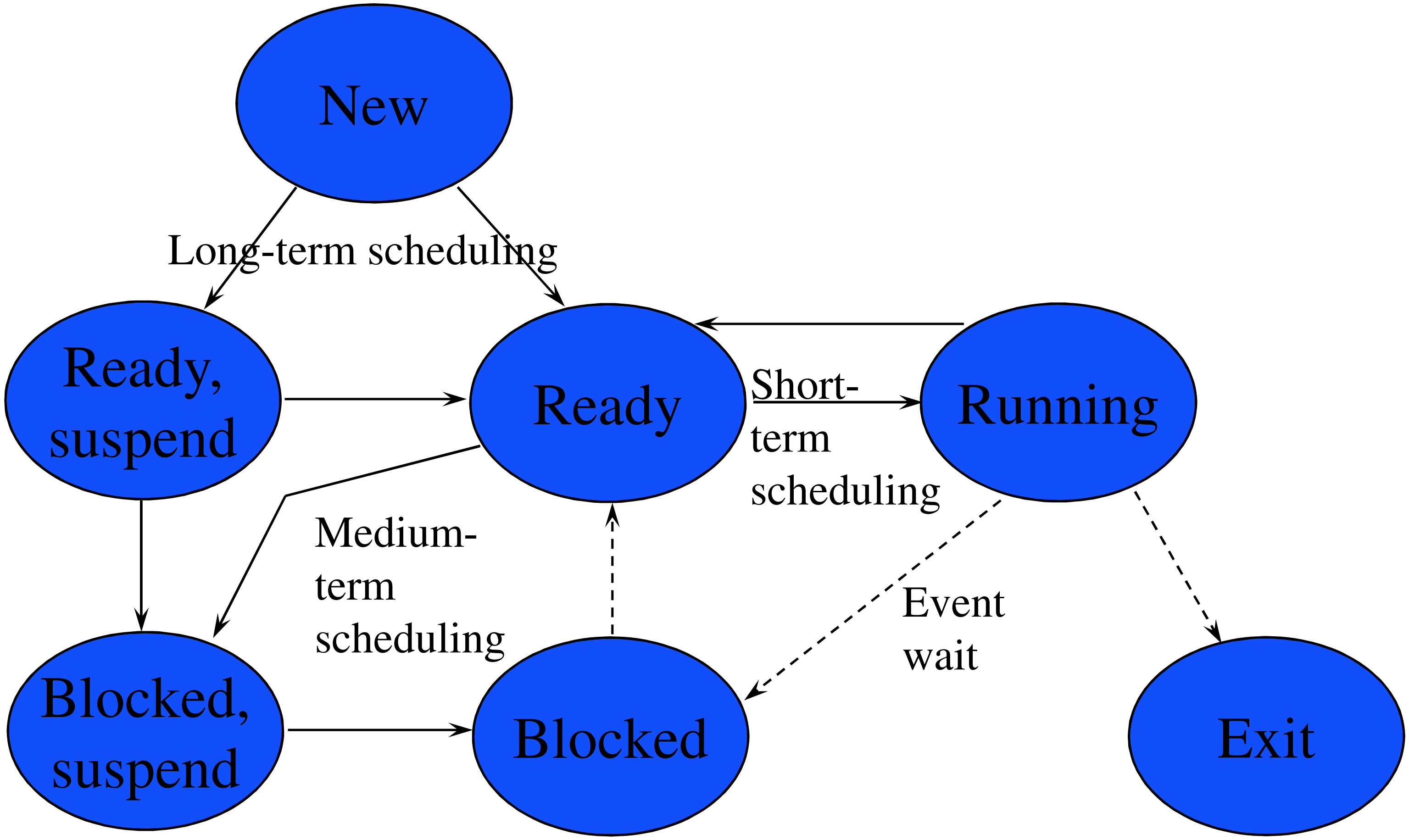
Medium vs. Short Term Scheduling

- Medium-term scheduling
  - Part of swapping function between main memory and disk
    - based on how many processes the OS wants available at any one time
    - must consider memory management if no virtual memory (VM), so look at memory requirements of swapped out processes
- Short-term scheduling (dispatcher)
  - Executes most frequently, to decide which process to execute next
  - Invoked whenever event occurs that interrupts current process or provides an opportunity to preempt current one in favor of another
  - Events: clock interrupt, I/O interrupt, OS call, signal

Long-term scheduling

- Determine which programs admitted to system for processing - controls degree of multiprogramming
- Once admitted, program becomes a process, either:
  - added to queue for short-term scheduler
  - swapped out (to disk), so added to queue for medium-term scheduler
- Batch Jobs
  - Can system take a new process?
    - more processes implies less time for each existing one
    - add job(s) when a process terminates, or if percentage of processor idle time is greater than some threshold
  - Which job to turn into a process
    - first-come, first-serve (FCFS), or to manage overall system performance (e.g. based on priority, expected execution time, I/O requirements, etc.)

Process State Transitions



Cooperating Processes

- Often need to share information between processes
  - information: a shared file
  - computational speedup:
    - break the problem into several tasks that can be run on different processors
    - requires several processors to actually get speedup
  - modularity: separate processes for different functions
    - compiler driver, compiler, assembler, linker
  - convenience:
    - editing, printing, and compiling all at once

Interprocess Communication

- Communicating processes establish a link
  - can more than two processes use a link?
  - are links one way or two way?
  - how to establish a link
    - how do processes name other processes to talk to
      - use the process id (signals work this way)
      - use a name in the filesystem (UNIX domain sockets)
      - indirectly via mailboxes (a separate object)
- Use send/receive functions to communicate
  - send(dest, message)
  - receive(dest, message)

Producer-consumer pair

- producer creates data and sends it to the consumer
- consumer read the data and uses it
- examples: compiler and assembler can be used as a producer consumer pair
- Buffering
  – processes may not produce and consume items one by one
  – need a place to store produced items for the consumer
    • called a buffer
  – could be fixed size (bounded buffer) or unlimited (un-bounded buffer)

Message Passing

- What happens when a message is sent?
  - sender blocks waiting for receiver to receive
  - sender blocks until the message is on the wire
  - sender blocks until the OS has a copy of the message
  - sender blocks until the receiver responds to the message
    - sort of like a procedure call
    - could be expanded into a remote procedure call (RPC) system
- Error cases
  - a process terminates:
    - receiver could wait forever
    - sender could wait or continue (depending on semantics)
  - a message is lost in transit
    - who detects this? could be OS or the applications
- Special case: if 2 messages are buffered, drop the older one
  - useful for real-time info systems