Announcements

- Program #2
  - Due 3/3 at 5:00 pm

- Reading
  - Finish scheduling
  - Process Synchronization:
    - Chapter 6 (8^{th} Ed) or Chapter 7 (6^{th} Ed)

In Class Exercise

- Give each group 15 minutes
  - to finish up their scheduling algorithm.
  - The algorithm should take a list of runnable processes and pick **one** to run next
  - Any criteria can be used
  - May keep data about processes, but need to describe what it is
- Have each group describe their algorithm
  - Ask the others if it does what they claim it does
  - Offer your own critiques of the algorithm
  - If one of the groups repeats another, still have them describe it
    - Look for any differences in how it achieves its goal

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

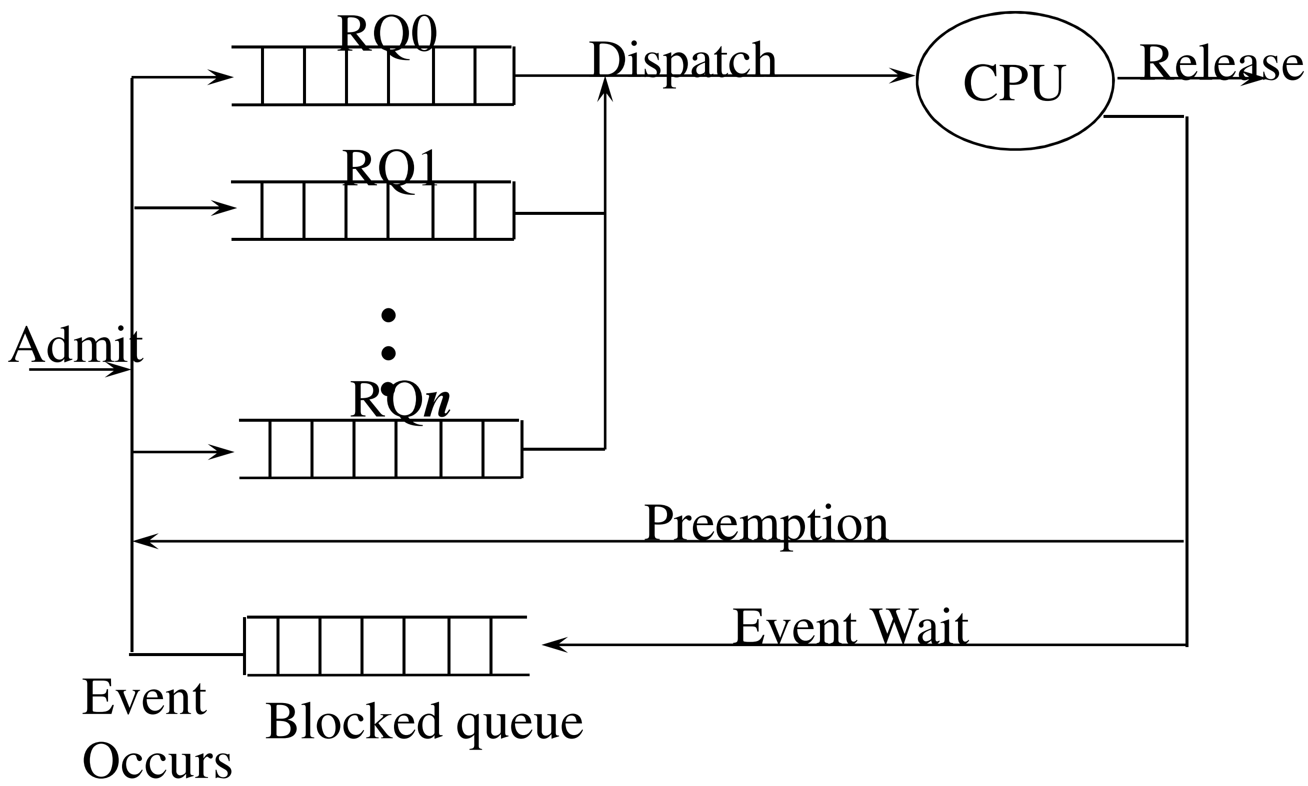
- First-Come, First-Served (FCFS, or FIFO)
  - as process becomes ready, join Ready queue, scheduler always selects process that has been in queue longest
  - better for long processes than short ones
  - favors CPU-bound over I/O-bound processes
  - need priorities, on uniprocessor, to make it effective

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

Readers/Writers Problem

- Data area shared by processors
- Some processes read data, others write data
  - Any number of readers my simultaneously read the data
  - Only one writer at a time may write
  - If a writer is writing to the file, no reader may read it
- Two of the possible approaches
  - readers have priority or writers have priority

Readers have Priority

```
Semaphore wsem = 1, x = 1;
reader()
{
 repeat
     P(x);
          readcount = readcount + 1;
          if readcount = 1 then P (wsem);
     V(x);
     READUNIT;
     P(x);
          readcount = readcount - 1;
          if readcount = 0 V(wsem);
     V(x);
 forever
};

writer()
{
    repeat
         P(wsem);
         WRITEUNIT;
         V(wsem)
    forever
}
```

Comments on Reader Priority

- semaphores x,wsem are initialized to 1
- note that readers have priority - a writer can gain access to the data only if there are no readers (i.e. when readcount is zero, signal(wsem) executes)
- possibility of starvation - writers may never gain access to data