Introduction to OS

Scheduling

MOS 2.4

Mahmoud El-Gayyar

elgayyar@ci.suez.edu.eg

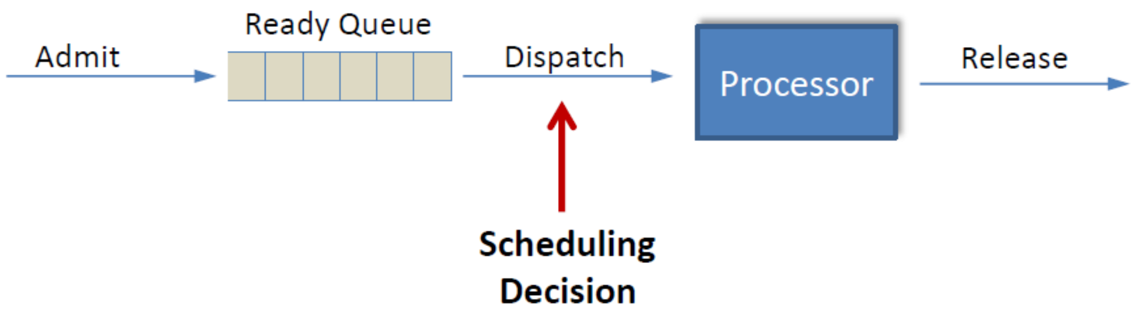
Why Scheduling?

- We know *how* to switch the CPU among processes or threads, but …

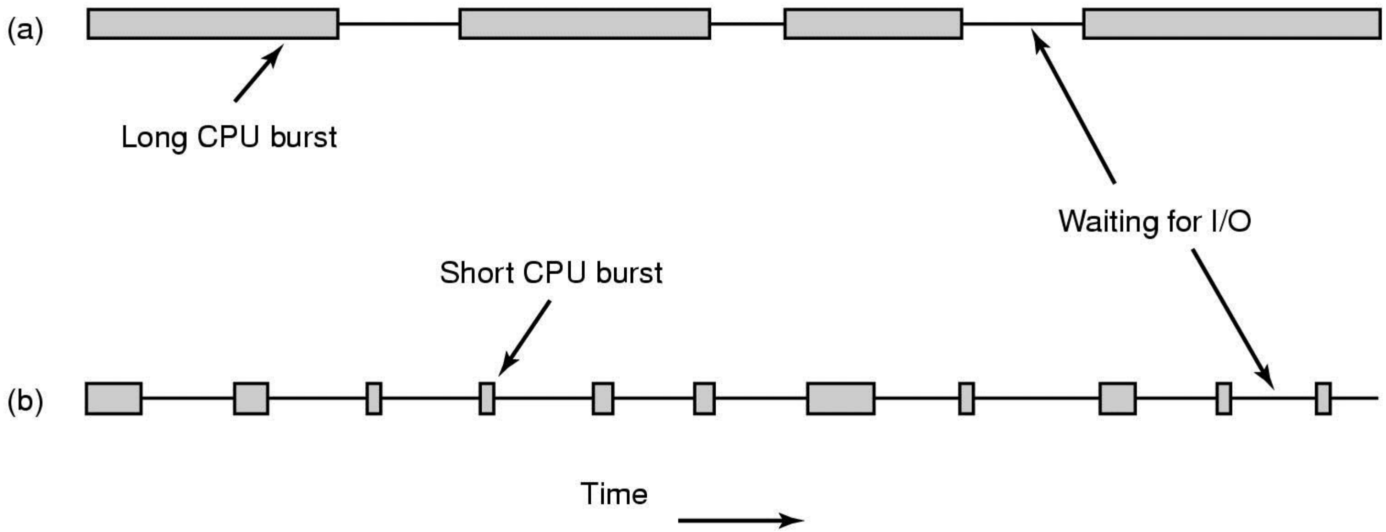
- How do we decide ***which to choose next***?

Role of Dispatcher vs. Scheduler

- **Dispatcher**
  - Low-level mechanism
  - Responsibility: ***context switch***
- **Scheduler**
  - High-level policy
  - Responsibility: ***deciding which process to run***
- Could have an **allocator** for CPU as well
  - Parallel and distributed systems



Process Categorization



- Bursts of CPU usage alternate with periods of I/O wait
  - a **CPU-bound (Compute-bound)** process (a)
  - an **I/O bound** process (b)
- Which process/thread should have preferred access to CPU? Which one should have preferred access to I/O or disk?

Scheduling Performance Criteria

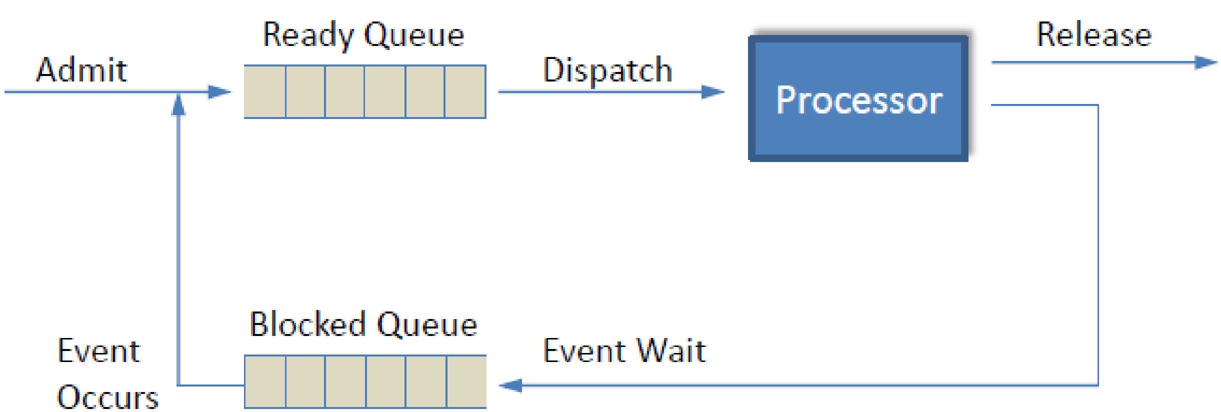
- ***Throughput*** – # of tasks that complete their execution per time unit
- ***Turnaround time*** – Total amount of time to execute one process to its completion
- ***Waiting time*** – amount of time task has been waiting in the ready queue
- ***Response time*** – amount of time from request submission until first response is produced

Scheduling – Policies

- Issues
  - *Fairness* – don't starve task
  - *Priorities* – most important first
  - *Deadlines* – task X must be done by time *t*
  - *Optimization* – e.g. throughput, response time
- Reality — No universal scheduling policy
  - Many models
  - Determine what to optimize - metrics
  - Select an appropriate one and adjust based on experience

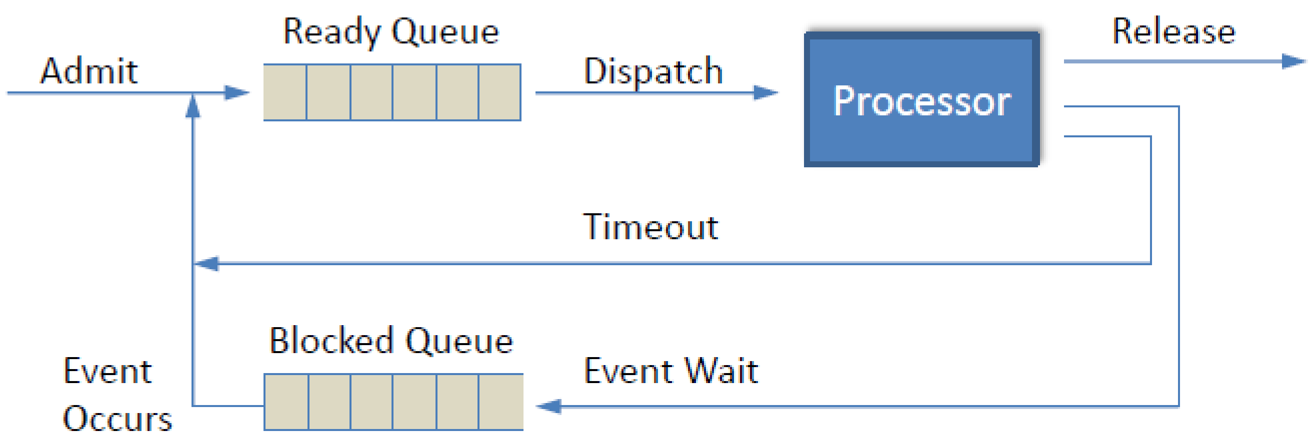
Non-preemptive Scheduling

- Once a process is scheduled, it continues to execute on the CPU, until
  - *it is finished (terminates)*
  - *It releases the CPU voluntarily*
  - *It blocks due to an event:*
    - *I/O interrupts, waits for another process*



Preemptive Scheduling

- The operating system interrupts processes
  - A scheduled process executes, until its time slice is used up, clock interrupt returns control of CPU back to scheduler at end of time slice
    - Current process is suspended and placed in the Ready queue
    - New process is selected from Ready queue and executed on CPU
  - When a process with higher priority becomes ready



Some task Scheduling Strategies

- First-Come, First-Served (FCFS)
- Shortest Job First (SJF)
  - Variation: Shortest Remaining Time First (SRTF)
- Round Robin (RR)
- Multilevel Queue scheduling

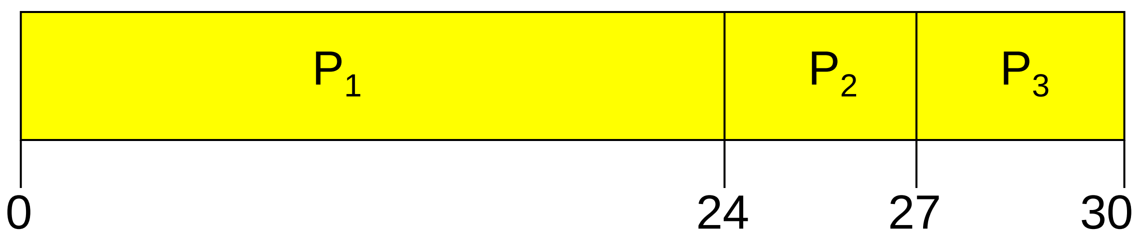
Scheduling Policies
First Come, First Served (FCFS)

- Easy to implement
- Non-preemptive
  - I.e., no task is moved from *running* to *ready* state in favor of another one
- Minimizes context switch overhead

Example: FCFS Scheduling

Task	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that tasks arrive in the order: P_1 , P_2 , P_3
- The time line for the schedule is:–



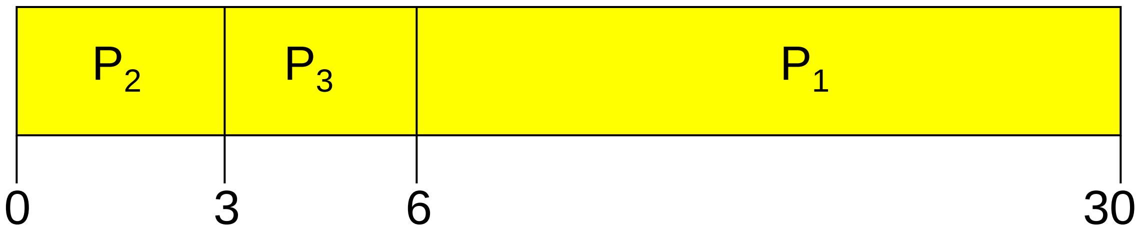
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- *Average waiting time:* $(0 + 24 + 27)/3 = 17$

Example FCFS: Different Order

Suppose instead that the tasks arrive in the order

$$P_2 , P_3 , P_1$$

- The time line for the schedule becomes:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- Previous case exhibits the ***convoy effect:***
  - short tasks stuck behind long tasks

FCFS Scheduling (summary)

- Short tasks penalized
  - I.e., once a longer task gets the CPU, it stays in the way of a bunch of shorter task
- Appearance of random or unpredictable behavior to users
- Does not help in real situations

Shortest-Job-First (SJF) Scheduling

- For each task, identify duration (i.e., length) of its next CPU burst.
- Use these lengths to schedule task with shortest burst

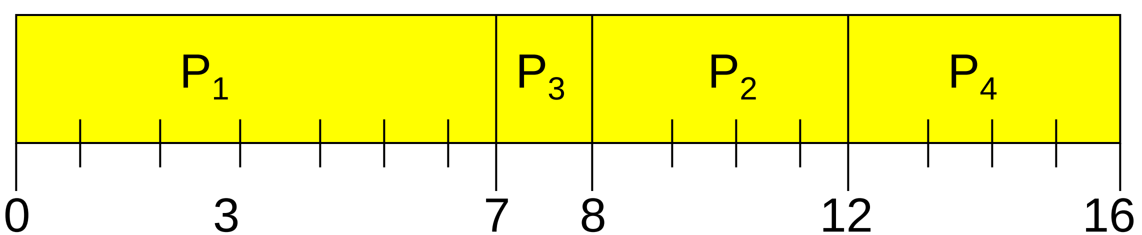
- Two schemes:–
  - ***Non-preemptive*** **–** once CPU given to the task, it is not preempted until it completes its CPU burst
  - ***Preemptive*** – if a new task arrives with CPU burst length less than remaining time of current executing task, preempt.
    - This scheme is known as the ***Shortest-Remaining-Time-First (SRTF)***

- SJF is provably optimal – gives minimum average waiting time for a given set of task bursts
  - Moving a short burst ahead of a long one reduces wait time of short task more than it lengthens wait time of long one

Example of Non-Preemptive SJF

Task	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

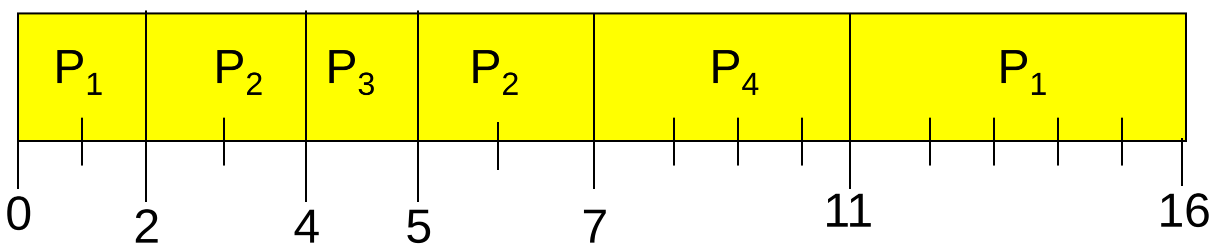


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Task	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Applications of SJF Scheduling

- Multiple desktop windows active at once
    - Document editing
    - Background computation (e.g., Photoshop)
    - Print spooling & background printing
    - Sending & fetching e-mail
    - Calendar and appointment tracking
- Desktop word processing (at thread level)
    - Keystroke input
    - Display output
    - Spell checker

Scheduling Policies – Round Robin

- Round Robin (RR)
  - *FCFS with preemption* based on ***time limits***
  - Ready tasks given a ***quantum*** of time when scheduled
  - Task runs until quantum expires or until it blocks (whichever comes first)
  - Suitable for **interactive** (timesharing) systems
  - *Setting quantum is critical for efficiency*

Round Robin (continued)

- Each task gets small unit of CPU time (*quantum*), usually 20-50 milliseconds.
  - After quantum has elapsed, task is preempted and added to end of ready queue.
- If n tasks in ready queue and quantum = q, then each task gets $1/n$ of CPU time in chunks of $\leq q$ time units.
  - No task waits more than $(n-1)q$ time units.
- Performance
  - q large $\Rightarrow$ equivalent to FCFS
  - q small $\Rightarrow$ may be overwhelmed by context switches

Example of RR with Time Quantum = 20

Task	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The time line is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better ***response***

Comparison of RR and FCFS

Assume: 10 jobs each take 100 seconds – look at when jobs complete

- FCFS – job 1: 100s, job 2: 200s, … job 10:1000s
- RR
  - 1 sec quantum
  - Job 1: 991s, job 2 : 992s …
- RR good for short jobs – worse for long jobs

Application of Round Robin

- Time-sharing systems
- *Fair* sharing of limited resource
  - Each user gets *1/n* of CPU
- Useful where each user has *one* process to schedule
  - Very popular in 1970s, 1980s, and 1990s
- Not appropriate for desktop systems!
  - *One* user, many processes and threads with very different characteristics

Priority Scheduling

- A priority number (integer) is associated with each task
- CPU is allocated to the task with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non-preemptive

Priority Scheduling

- (Usually) preemptive
- Tasks are given *priorities* and ranked
  - Highest priority runs next
  - May be done with multiple queues – *multilevel*
- *SJF* ≡ priority scheduling where priority is next predicted CPU burst time
- Recalculate priority – many algorithms
  - E.g. increase priority of I/O intensive jobs
  - E.g. favor tasks in memory

Priority Scheduling Issue #1

- Problem: ***Starvation*** – low priority tasks may never execute
- Solution: ***Aging*** – as time progresses, increase priority of waiting tasks

Priority Scheduling Issue #2

- ***Priority inversion***
  - *A* has high priority, *B* has medium priority, *C* has lowest priority
  - *C* acquires a resource that *A* needs to progress
  - *A* attempts to get resource, fails and busy waits
    - *C* never runs to release resource!

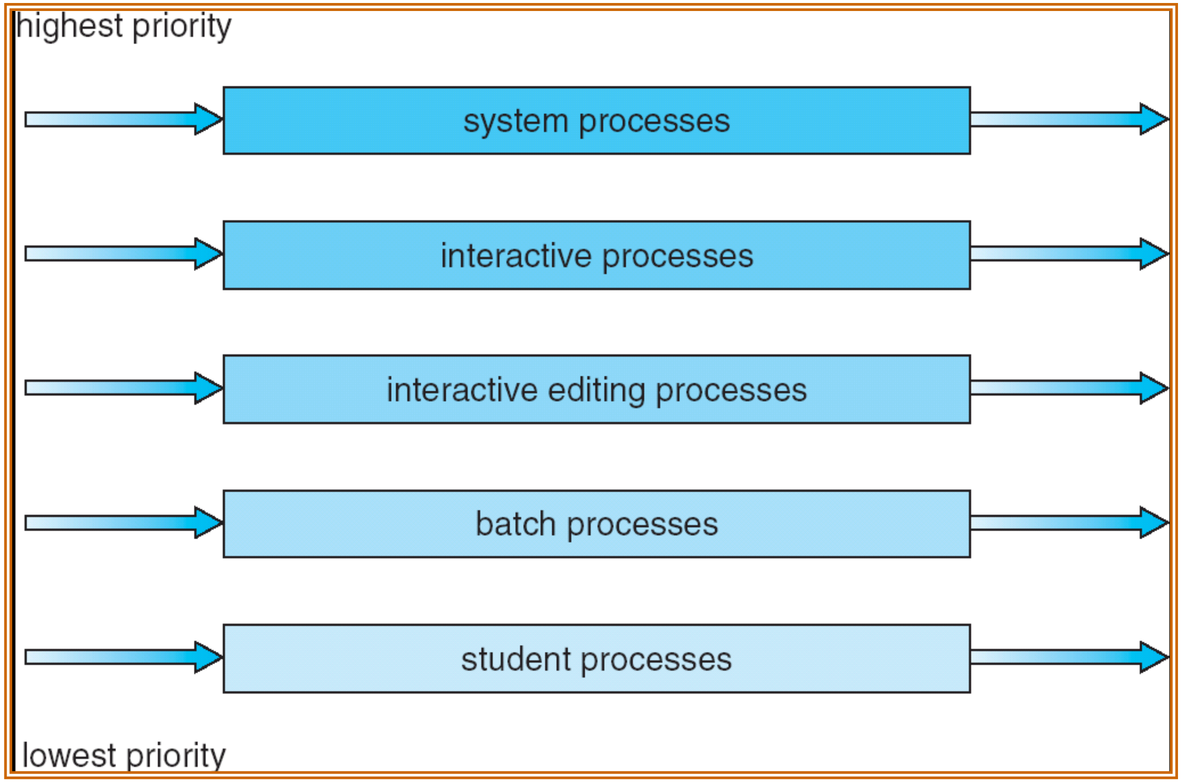
  or

  - *A* attempts to get resources, fails and blocks
    - *B* (medium priority) enters system & hogs CPU
    - *C* never runs!
- **Solution:** Some systems increase the priority of a process/task/job to match level of waiting task

Multilevel Queue

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - *Fixed priority scheduling:* (i.e., serve all from foreground then from background). Possibility of starvation.
  - *Time slice* – each queue gets a certain amount of CPU time which it can schedule amongst its tasks; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling



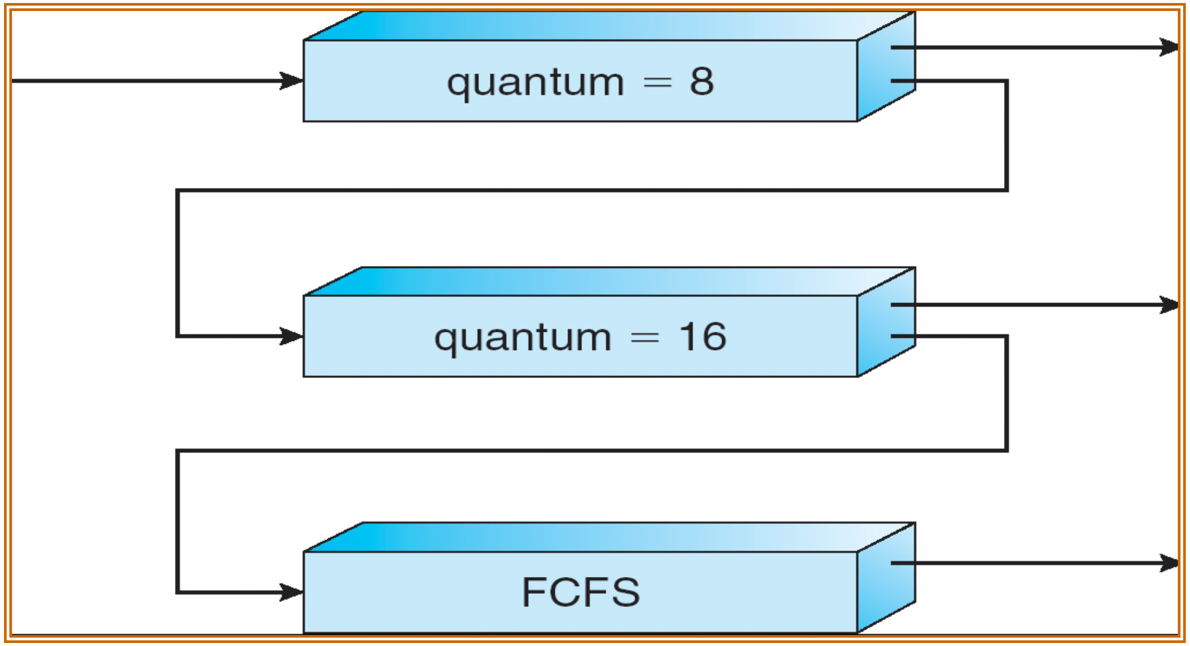
Multilevel Feedback Queue

- A task can move between the various queues
  - **Aging** can be implemented this way
  - "Penalize processes that have been running longer"
  - A process is downgraded according to CPU time consumed so far
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a task
  - method used to determine when to demote a task
  - method used to determine which queue a task will enter when that task needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - New job enters queue Q_0 (FCFS). When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

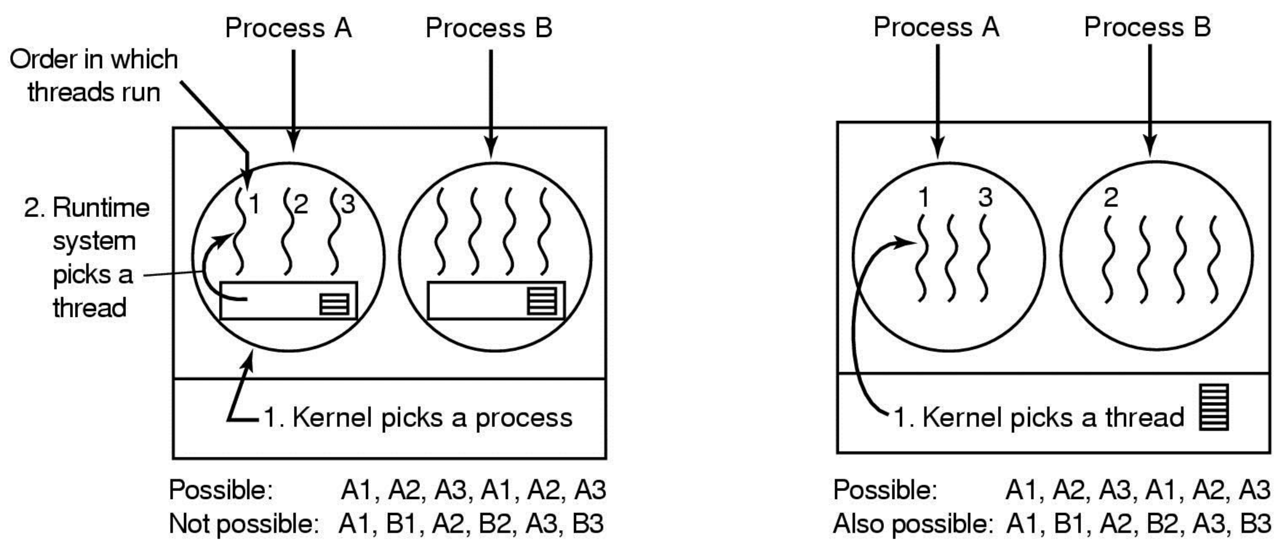
Multilevel Feedback Queues



- Effect:
  - Processes trickle down the priority queues
  - Short processes stop earlier in this descent
  - Long processes will end in the lowest-priority queue

Thread Scheduling

- *Local Scheduling* – How the threads library decides which user thread to run next within the process
- *Global Scheduling* – How the kernel decides which kernel thread to run next



Kernel picks process

Kernel picks Thread

Scheduling – Examples

- Unix – multilevel - many policies and many policy changes over time
- Linux – multilevel with 3 major levels
  - Real-time FIFO
  - Real-time round robin
  - Timesharing
- Windows Vista – two-dimensional priority policy
  - *Process class* priorities
    - Real-time, high, above normal, normal, below normal, idle
  - *Thread* priorities relative to class priorities.
    - Time-critical, highest, …, idle

Scheduling – Summary

- General theme – what is the “best way” to run n tasks on k resources? ($k < n$)
- Conflicting Objectives – no one “best way”
  - Speed vs. fairness
- Incomplete knowledge
  - E.g. – does user know how long a job will take
- Real world limitations
  - E.g. context switching takes CPU time
  - Job loads are unpredictable
- Bottom line – scheduling is hard!
  - Know the models
  - Adjust based upon system experience
  - Dynamically adjust based on execution patterns

Review

- Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?

- Can a measure of whether a process is likely to be CPU bound or I/O bound be determined by analyzing source code? How can this be determined at run time?