CPU Scheduling (cont’d)

– scheduling algorithms
  – FCFS (review)
  – Shortest Job First (SJF)
  – Priority
  – Round-robin
– real-time schedulers

[Silberschatz/Galvin/Gagne: Sections 6.3, 6.5]

(Review)
Performance of FCFS

• 3 processes P1, P2, and P3 with CPU requirements 24, 3, and 3 msecs
  – Arrive at the same time in that order
   
   \[
   \frac{(0+24+27)/3}{3} = 17
   \]
   
   \[
   \frac{(24+27+30)/3}{3} = 27
   \]
   
   \[
   \frac{(30)/3}{3} = 10
   \]

– Can do better if the processes are scheduled differently

\[
\begin{array}{l|l|l|l}
\text{P1} & \text{P2} & \text{P3} \\
\end{array}
\]

– Average waiting time = (0+3+6)/3 = 3 !!!
– Average turnaround time = (3+6+30)/3 = 13 !!!
– Average throughput = (30)/3 = 10

Scheduling Algorithms (2)
Shortest Job First (SJF)

• The next process to be assigned the CPU is one that is ready and with smallest next CPU burst; FCFS is used to break ties
  – From the previous example,
    – P1, P2, P3 arrive at the same time in that order, needing CPU times 24, 3, 3
    – FCFS yielded an average waiting time of 17 units
    – SJF yields order P2, P3, P1, with average waiting time of 3 units
  – Another example
    – P1, P2, P3, P4 requiring bursts of 8, 9, 4, and 5 arrive 1 time unit apart

\[
\begin{array}{l|l|l|l}
\text{P1} & \text{P2} & \text{P3} & \text{P4} \\
\end{array}
\]

FCFS: Average waiting time = \((0 + (8 - 1) + (17 - 2) + (21 - 3))/4 = 10\) units

SJF: Average waiting time = \((0 + (17 - 1) + (8 - 2) + (12 - 3))/4 = 7.75\) units
Evaluation of SJF

- **Pro:** If times are accurate, SJF gives *minimum* average waiting time

**Estimating the burst times**

- For long-term scheduling, user can be “encouraged” to give estimate
  - part of the job submission requirements

- For short-term scheduling, scheduler attempts to predict value
  - the approach assumes some *locality* in process CPU burst times
    - Use exponential averaging
      - \( \tau_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot \tau_n \)
      - where,
        - \( \tau_n \) is the estimated value for the n’th CPU burst
        - \( T_n \) is the actual most recent burst value
      - \( \alpha = 0 \) implies fixed estimate; \( \alpha = 1 \); \( \alpha = 0.5 \)
    - the estimate lags the (potentially) sharper transitions of the CPU bursts

**Modifications to SJF**

- Preemptive SJF (also called *shortest remaining time first*)
  - if the shortest estimated CPU burst among all processes in the ready queue is less than the remaining time for the one running,
    - preempt running process; add it to ready queue w/ remaining time
    - give CPU to process with the shortest CPU burst
  - policy prioritizes jobs with short CPU bursts

- Example: A, B, C, D with bursts 8, 9, 4, 5 arrive 1 time unit apart

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>

SJF: Average waiting time = \((0 + (17 - 1) + (8 - 2) + (12 - 3))/4\) = 7.75 units

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
</table>

Preemptive SJF: Average waiting time = \(( (0 - 0 + 9) + (17 - 1 + 0) + (2 - 2 + 0) + (6 - 3 + 0) )/4 \) = 7 units

**Scheduling Algorithms (3)**

**Priorities: A More General Scheduling Notion**

- Elements of a *priority-based scheduler*
  - Process priorities (for example 0..100)
    - convention: a smaller number means higher priority
  - Tie-breaker mechanism
    - Example: FCFS
  - Map priority to considerations we have in mind
    - Internal
      - memory and other needs of the job
      - ratio of CPU to I/O burst times
      - number of open files etc.
    - External
      - the amount of money paid by the process owner
      - the importance of the user group running the process
  - *Priority-based scheduling*
    - assign the CPU to the process with highest priority
    - may be used with or without preemption
Priority-based Scheduling: Example

- Consider five processes A, B, C, D, and E
  - With burst times: 10, 1, 2, 1, 5
  - With priorities: 3, 1, 3, 4, 2 (lower is better)
  - Arriving at times: 0, 0, 2, 2, 3

Without preemption:

```
B     A     E     C     D
```

Average waiting time: \( \frac{(1 - 0) + (0 - 0) + (16 - 2) + (18 - 2) + (11 - 3))/5 = 7.8} \)

With preemption:

```
B     A     E     A     C     D
```

Average waiting time: \( \frac{(1 - 0 + 7) + (0 - 0) + (16 - 2) + (18 - 2) + (3 - 3))/5 = 7.6} \)

Problems with Priority Schemes

- Process can be overtaken by higher priority processes arriving later
  - can happen continuously: leads to starvation
  - leads to better overall performance perhaps
    - but not from the point of view of the process in question

- Common solution: A process' priority goes up with its age
  - FCFS is used to break ties between processes with equal priorities
  - For a process in ready queue, its priority will eventually be the highest

- A low-priority process holds resources required by a high-priority process? (priority inversion)

- Common solution: Priority inheritance
  - process with lock inherits priorities of processes waiting for the lock
  - priority reverts to original values when lock is released

Example of Priority Ageing: Unix

- Priority goes up with lack of CPU usage
  - process accumulates CPU usage
  - every time unit (~ 1 second)
    - recalculates priority
      \( \text{priority} = \text{CPUusage} + \text{basepriority} \)
    - halves CPU usage carried forward
      \( \text{CPUusage} = (\text{CPUusage})/2 \)
    - recall that smaller number implies a higher priority
  - basepriority is settable by user
    - within limits
    - using “nice”

- Assuming all processes have the same base priority:
  - Are new processes prioritized over existing ones?
  - How does the priority of a process change over its lifetime?

Scheduling Algorithms (4): Round Robin (RR)

- A strictly preemptive policy

- At a general level
  - choose a fixed time unit, called a quantum
  - allocate CPU time in quanta
  - preempt the process when it has used its quantum
    - Unless the process yields the CPU because of blocking
      - typically, FCFS is used as a sequencing policy
        - each new process is added at the end of the ready queue
        - when a process blocks or is preempted, it goes to the end of the ready queue
    - very common choice for scheduling interactive systems
Round-robin Scheduling: Example

- Consider five processes A, B, C, and D
  - With burst times: 4, 1, 2, 5
  - Arriving at times: 0, 0, 2, 3
- Round-robin system with quantum size 1 unit
  - Overhead of context switching a process: 0.2 units
    - Incurred only when a process is preempted or needs to block

Waiting time: \( \frac{(0 - 0 + 6.2) + (1.2 - 0 + 2.6) + (3.4 - 2 + 3.6)}{4} = 4.15 \) units

Response time: \( \frac{(0 + 1.2 - 0 + 3.4 - 2 + 4.6 - 3)}{4} = 1.05 \) units

CPU utilization?

Choice of Quantum Size

- Quantum size \( q \) is critical
- Affects waiting and turnaround times
  - if \( q \) is the quantum size and there are \( n \) processes in the ready queue,
    - the maximum wait is \( (n-1) \cdot q \) units of time
  - as \( q \) increases, we approach FCFS scheduling
  - as \( q \) decreases
    - the rate of context switches goes up, and the overhead for doing them
    - the average wait time goes down, and the system approaches one with \( 1/n \) the speed of the original system

Hybrid Schemes: Multilevel Queue Scheduling

- Processes are partitioned into groups based on static criteria
  - background (batch)
  - foreground (interactive)
- All the processes in a fixed group of the partition share the same scheduling strategy and a distinct family of queues
  - different scheduling algorithm can be used across different groups
    - foreground: Round Robin
    - background: FCFS
- Need to schedule the CPU between the groups as well; for example,
  - fixed-priority: e.g., serve all from foreground, then from background
    - possibility of starvation
  - time slice: each group gets a certain fraction of the CPU
    - e.g., 80% to foreground in RR, 20% to background in FCFS

Generalization: Multilevel Feedback Queues

- Provide a mechanism for jobs to move between queues
  - ageing can be implemented this way
- Complete specification
  - queues: number, scheduling algorithms (within and across queues)
  - promotion and demotion policies
  - which queue should a process enter when it needs service?
- Example: 3 queues: \( Q_0 \) (FCFS, 8ms), \( Q_1 \) (FCFS, 16ms), \( Q_2 \) (FCFS)
Choosing a Scheduling Approach

- Identify metrics for evaluation
  - we have already seen a variety of metrics
    - throughput, wait time, turnaround time, ...
  - the goal is to start with an expectation or specification of what the scheduler should do well
    - for example, we might wish to have a system in which
      - the CPU utilization is maximized, subject to a bound on the response time

- Evaluate how different scheduling algorithms perform
  - deterministic modeling
    - requires accurate knowledge of job and system characteristics
    - practical only for real-time and embedded systems
  - more detailed performance evaluation
    - queueing models, simulation, measurement

- See Section 6.6 for details

Real-Time Scheduling: Concepts

- Processes have real-time requirements (deadlines)
  - e.g., a video-frame must be processed within certain time
  - growing in importance
    - media-processing on the desktop
    - large-scale use of computers in embedded settings
      - sensors produce data that must be processed and sent to actuators

- Real-time tasks typically considered along two dimensions
  - aperiodic (only one instance) versus periodic (once per period T)
  - hard real-time (strict deadlines) versus soft real-time
    - hard real-time tasks require resource reservation, and
      - typically specialized hardware and scheduling algorithms
      - earliest-deadline first
      - rate-monotonic scheduling
      - details are beyond the scope of this class
    - our focus is on supporting soft real-time tasks in a general environment

Soft Real-Time Scheduling

- Most contemporary, general-purpose OSes deal with soft real-time tasks by being as responsive as possible
  - ensure that when a deadline approaches, the task is quickly scheduled
    - minimize latency from arrival of interrupt to start of process execution

Soft Real-Time Scheduling: OS Requirements

- Minimize interrupt processing costs
  - minimization of intervals during which interrupts are disabled

- Minimize dispatch latency
  - preemptive priority scheduling
    - real-time processes have higher priority than non-real-time processes
    - priority of real-time processes does not degrade over time
    - current activity must be preemptible
      - Unacceptable options
        - traditional UNIX approach (waiting for system call completion)
        - preemption at safe points
      - Acceptable: entire kernel must be preemptible (e.g., Solaris 2)
        - kernel data structures protected by synchronization mechanisms
      - Must cope with the priority inversion problem
        - A lower-priority process holds a resource required by the higher-priority process
        - See Review Question 13
Windows NT/2000 Scheduler

- Preemptive, priority based
- 32 priority levels
  - higher priority numbers imply higher priority
    - 0-15 are variable priority classes
      - normal processes start off at this level
      - process has a base priority (can take values from 0-15)
      - threads in the process can start at priority = \( (\text{base priority} \pm 2) \)
        - NT Executive raises priorities of I/O-bound threads (max value is 15)
        - NT Executive lowers priorities of CPU-bound threads (min value is \( \text{base priority} - 2 \))
    - 16-31 are real-time priority classes
      - real-time threads have a fixed priority
      - threads within a particular level processed according to RR

Advanced Topic: Fair-Share Scheduling

- Problems with priority-based systems
  - priorities are absolute: no guarantees when multiple jobs with same priority
  - no encapsulation and modularity
    - behavior of a system module is unpredictable: a function of absolute priorities assigned to tasks in other modules
- Solution: Fair-share scheduling
  - each job has a share: some measure of its relative importance
    - denotes user’s share of system resources as a fraction of the total usage of those resources
    - e.g., if user A’s share is twice that of user B
      - then, in the long term, A will receive twice as many resources as B
- Traditional implementations
  - keep track of per-process CPU utilization (a running average)
  - reprioritize processes to ensure that everyone is getting their share
  - are slow!

Example Fair-Share Policy: Lottery Scheduling

- A randomized mechanism for efficient proportional-share resource management
  - each process has certain number of lottery tickets (its share)
    - Processes reside in a conventional ready queue structure
    - each allocation is determined by holding a lottery
      - Pick a random ticket number
      - Grant resource to process holding the winning ticket

Why Does Lottery Scheduling Work?

- Expected allocation of resources to processes is proportional to the number of tickets that they hold
- Number of lotteries won by a process has a binomial distribution
  - probability \( p \) of winning = \( t/T \)
  - after \( n \) lotteries, \( E[w] = np \) and variance = \( np(1-p) \)
- Number of lotteries to first win has a geometric distribution
  - \( E[n] = 1/p \), and variance = \((1-p)/p^2 \)