(Review)
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

Outline

- Announcements
  - Lab 3 now due March 24th
    - Demos on March 24th (Monday) and 25th (Tuesday)
  - Midterm exam on March 12th
    - Review session on March 10th

- CPU Scheduling (cont’d)
  - real-time schedulers

- Deadlocks
  - system model
  - deadlock characterization
  - deadlock prevention

[Silberschatz/Galvin/Gagne: Sections 6.5, 8.1–8.4]

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

- Processes have real-time requirements (deadlines)
- Announcements
- CPU Scheduling (cont’d)
- Deadlocks
- Announcements
- CPU Scheduling (cont’d)
- Deadlocks

Outline

- Announcements
  - Lab 3 now due March 24th
    - Demos on March 24th (Monday) and 25th (Tuesday)
  - Midterm exam on March 12th
    - Review session on March 10th

- CPU Scheduling (cont’d)
  - real-time schedulers

- Deadlocks
  - system model
  - deadlock characterization
  - deadlock prevention

[Silberschatz/Galvin/Gagne: Sections 6.5, 8.1–8.4]

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

- Processes have real-time requirements (deadlines)
- Announcements
- CPU Scheduling (cont’d)
- Deadlocks
- Announcements
- CPU Scheduling (cont’d)
- Deadlocks

Outline

- Announcements
  - Lab 3 now due March 24th
    - Demos on March 24th (Monday) and 25th (Tuesday)
  - Midterm exam on March 12th
    - Review session on March 10th

- CPU Scheduling (cont’d)
  - real-time schedulers

- Deadlocks
  - system model
  - deadlock characterization
  - deadlock prevention

[Silberschatz/Galvin/Gagne: Sections 6.5, 8.1–8.4]
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)
    - NT Executive raises priorities of I/O-bound threads (max value is 15)
    - NT Executive lowers priorities of CPU-bound threads (min value is 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 = \((1-p)/p^2\)

Outline

- Announcements
  - Lab 3 due March 10th
  - Demos on March 10th (Monday) and 11th (Tuesday)
  - Midterm exam on March 12th
  - Review session on March 10th

- CPU Scheduling (cont’d)
  - real-time schedulers

- Deadlocks
  - system model
  - deadlock characterization

[Silberschatz/Galvin/Gagne: Sections 6.5, 8.1–8.2]
System Model for Deadlocks

- **Resources**
  - different types of resources (e.g., memory space, CPU cycles, file handles)
  - processes request a resource type, not a particular resource
    - any of the resources in that type can be used to satisfy the request
- **Processes**
  - use resources
    - request resource type i
    - use resource type i
    - release resource type i
  - a process can request multiple instances of a resource
  - OS intervenes on request and release
- **Deadlocks:** Caused by processes waiting for events that never happen
  - events of interest: request and release
  - events can be for different resource types

Conditions for Deadlock

- Deadlocks involve a set of processes contending for a set of resources
- All of the following conditions must hold for deadlock to occur
  - Mutual Exclusion
    - at least one resource can only be used by one process at any one time
  - Hold and Wait
    - there must exist at least one process that is holding at least one resource, and is waiting to acquire additional resources currently held by other processes
  - No Preemption
    - processes cannot be forced to give up resources
  - Circular Wait
    - there is a sequence of processes \( p_1, p_2, ..., p_n, p_1 \)
    - such that \( p_i \) is waiting for a resource held by \( p_{i+1} \)

Conditions for Deadlock: Dining Philosophers

- Conditions
  - Mutual exclusion
    - each chopstick can only be used by one philosopher
  - Hold and wait
    - philosophers hold on to a chopstick while requesting another
  - No preemption
    - not possible to force a philosopher to give up a chopstick
  - Circular wait
    - philosopher waits on philosopher \( i+1 \) ...

Philosopher\(_i\)

```c
P( chopstick[i] );
P( chopstick[1+i mod 5] );
EAT
V( chopstick[i] );
V( chopstick[1+i mod 5] );
THINK
```

Graph Representations of Deadlocks

A Resource Allocation Graph (RAG)

- Two types of nodes
  - Processes and
  - Resources
- Three types of directed edges between Processes and Resources
  - request edge: a solid edge from \( P \) to \( r \), indicating that \( P \) has requested \( r \)
  - assignment edge: a solid edge from \( r \) to \( P \), indicating that the OS has already allotted resource \( r \) to process \( P \)
  - claim edge: a dotted edge from a process node \( P \) to a resource node \( r \), indicating that \( P \) may request \( r \) at some point in the future
- We shall focus only on requests for exclusive access to a resource
  - handling of mixed access types is slightly complicated
Resource Allocation Graphs: Example

- Assignment edges originate from a resource instance
- A request edge is *instantaneously* transformed to an assignment edge if resources are available

Deadlocks in RAGs with Single Resource Instances

- A cycle in the graph is a necessary and sufficient condition for the existence of a deadlock

Deadlocks in RAGs with Multiple Resource Instances

- A cycle in the graph is a necessary but not sufficient condition for the existence of a deadlock
  - if a cycle does not exist: no deadlock
  - if a cycle exists: there may or may not be a deadlock

Methods for Handling Deadlocks

Three approaches with different cost-performance tradeoffs

- **Prevention**
  - deadlock cannot possibly occur

- **Avoidance**
  - deadlock can occur, but there are algorithms to avoid it
  - relies on the OS having an *advance model* of possible resource requests from processes

- **Detection and Recovery**
  - deadlock may occur, but there are ways of detecting it and recovering
  - this method is preferable when deadlocks happen rarely
Deadlock Prevention

- Approach: Ensure that the necessary conditions for deadlocks are never satisfied

- Prevent one of the following from becoming true
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait

Deadlock Prevention (1): Mutual Exclusion

- Mutual exclusion is not a problem for sharable resources
  - an example is a “read-only” file which is a resource that can be accessed simultaneously

- Problem: Some resources are inherently not sharable
  - so, denying the mutual exclusion condition cannot be enforced in general

Deadlock Prevention (2): Hold-and-Wait

- Approach: Guarantee that when a process requests a resource it does not hold any other resources
  - Choice 1: A process requests and is allocated all of its resources before it begins execution
    - require system calls requesting resources to precede all other system calls
    - Need all resources to be requested with a single call
  - Choice 2: A process releases any resources it is holding before it requests for new ones
  - Choice 3: A process that is “holding” a resource immediately releases it if another of its requests cannot be satisfied currently

- Limitations
  - inefficient
  - lowered resource utilization
  - starvation

Deadlock Prevention (3): No Preemption

- Approach: Take away resources from a process (preemption) and give them to another waiting process
  - some resources are preemptible
    - e.g., memory space, disk space (on a particular disk)
    - these can be taken away from a process
  - examples of non-preemptible resources?

- Choices
  - protocol 1: if a process is holding some resources and requests other resources that cannot be granted to it, all of its resources are taken away
  - protocol 2: when a process requests additional resources, see if these resources are being held by a process that is itself waiting for new resources. In this situation, preempt the second process

- Limitation: Cost of preemption
  - a process may get preempted even when there is no deadlock
Deadlock Prevention (4): Circular Wait

- Example:
  - Processes need three resources: memory, disk, printer
  - Consider two cases:
    - Case 1: processes pick own order in which to ask for resources
    - Case 2: each process asks first for memory, then disk, then the printer
  - Which of these cases can result in deadlock?

- Approach: Impose an order on resource acquisition
  - all the N types of resources in the system are linearly ordered
  - each is given a number, called rank, in the range \( 1, 2, \ldots, N \)
  - the resources of the same type all have the same rank
  - different types of resources get distinct ranks
  - processes are required to sequence their resource requests in strictly increasing order of rank
    - i.e., they ask for all the “smaller” rank resources first

- In our example
  - rank(memory) = 1, rank(disk) = 2, rank(printer) = 3

- Why does this work?

Deadlock Prevention: Circular Wait (cont’d)

- Why it works
  - suppose the circular wait consists of processes P1, P2, ..., Pn, P1
  - suppose P(i) is waiting on a resource held by P(i+1) of rank R(j)
    - P(i+1) must have been granted all resources it needs of rank R(j)
    - it must therefore be waiting for a resource of rank \( R(j) + 1 \)
  - since a cycle of strictly increasing ranks cannot exist, there can exist no such cycle.

- Two related points
  - an equivalent strategy is one where a process, when it requests a resource of a particular rank, releases all those with a higher rank
  - typical rank orders are based on natural usage
    - e.g., since storage devices are used “before” printers, they get smaller ranks

Deadlock Prevention: Summary

- Main idea: Prevent one of the four necessary conditions
  - mutual exclusion
  - hold-and-wait: ask for all resources at start
  - no preemption
  - circular wait: resource ranking scheme

- Limitations
  - inefficient
    - static allocation of resources reduces concurrency
    - a process may need to be preempted even when there is no deadlock
  - restrictive
    - requires allocation of future resource requirements before it starts executing

- Alternative approaches?