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

Outline

- Announcements
  - Mid-term exam on March 6th
  - Lab 3 due back March 18th (Monday after Spring Break)
- CPU Scheduling
  - Soft real-time scheduling
  - Example: Windows 2000
- Deadlocks
  - system model
  - deadlock characterization

[ Silberschatz/Galvin/Gagne: Section 6.5, 6.7, 8.1 – 8.2]
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
      - 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 \( \text{base} \) = \( \text{base priority} \pm 2 \)
      - 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

Process Deadlock

- Example:
  - 2 processes, each holding a different resource in exclusive mode, and each requesting access to the resource held by the other process
    - e.g., processes requiring access to disk and printer
      - one process acquires the disk and waits for the printer
      - the other acquires the printer and waits for the disk
    - neither will make progress!
- Definition
  A deadlock occurs when a set of processes in a system is blocked waiting on requirements that can never be satisfied.
  These processes, while holding some resources, are requesting accesses to resources held by other processes in the same set.
  In other words, the processes are involved in a circular wait.
- Resolving the deadlock requires the intervention of some process outside those involved in the deadlock

Deadlock versus Starvation

- Deadlock: A process waits for a resource that is currently assigned to another process, which is in turn waiting for another resource ...
- Starvation: A process waits for a resource that continually becomes available but is never assigned to the waiting process
- Two major differences between deadlock and starvation
  - in starvation, it is not certain that a process will never get the requested resource (i.e., there is a chance it might), while a deadlocked process is permanently blocked
  - in starvation, the resource under contention is continuously available, whereas this is not true in a deadlock
- Starvation is typically easier to fix than deadlock
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 $p_1, p_2, ..., p_d, p_1$
    - 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** $i$
  - $P(\text{chopstick}[i])$
  - $P(\text{chopstick}[i+1 \mod 5])$
  - $EAT$
  - $V(\text{chopstick}[i])$
  - $V(\text{chopstick}[i+1 \mod 5])$
  - $THINK$

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