Announcements

• HW #3 is due Friday Feb. 25
  – extra office hours Thursday 1 pm - post this
• PA #2 assigned, due Thursday March 17
• Midterm is Thursday March 10
  – Midterm review is Tuesday March 8
• Read chapter 10
From last time...

• We discussed:
  – Dining Philosophers Problem - deadlock!
  – monitors
    • high-level synchronization primitives
    • don’t have to explicitly P() and V()
    • augmented with condition variables
  – monitor-based solution to Dining Philosophers problem
Deadlock

- saw earlier that semaphores provide mutual exclusion, but can introduce deadlock
  - 2 process, each desires a resource locked by the other process
  - can occur easily due to programming errors, e.g. by switching order of P and V, etc.
- Even if no obvious programming errors, deadlock can occur
  - deadlock is difficult to anticipate by a single thread, because the code looks fine, and deadlock is a higher-level concept that involves the distributed behavior of multiple processes/threads
  - deadlock is difficult to anticipate, detect, reproduce, prevent, avoid, and recover from
Deadlock

- A set of processes is in a deadlock state when every process in the set is waiting for an event (e.g. release of a resource) that can only be caused by another process in the set

- Multithreaded programs are good candidates for deadlock
  - thread-thread deadlock within a process
  - thread-thread deadlock across processes
Deadlock Detection

• Modeling deadlock:
  - to use a resource, a process must
    1. request() a resource -- must wait until it’s available
    2. use() or hold() a resource
    3. release() a resource
  - thus, we have resources and processes
  - Most of the following discussion will focus on reusable resources

Processes

Resources

Buffer1    ...    BufferN    File F    Disk D

P1

P2

P3

P1 holds Buffer 1 and File F
P2 holds Buffer N
P3 holds Disk D
Deadlock Detection

• a resource allocation graph can be used to model deadlock
  – try to represent deadlock by a directed graph $D(V,E)$, consisting of
    • vertices $V$: namely processes and resources
    • and edges $E$:
      – a request() for a resource $R_j$ by a process $P_i$ is signified by a directed arrow from process $P_i \rightarrow R_j$
      – a process $P_i$ will hold() a resource $R_j$ via a directed arrow $R_j \rightarrow P_i$
Deadlock Detection

- **Example 1:**
  - P1 holds an *instance* of resource R2, and is requesting resource R1
  - P2 holds R1 and an instance of R2, and requests R3
  - P3 holds R3
  - There is no deadlock
    - if the graph contains no cycles or loops, then there is no deadlock
Deadlock Detection

• Example 2:
  – same graph as before, except now P3 requests an instance of R2
  – Deadlock occurs!
    • P3 requests R2, which is held by P2, which requests R3, which is held by P3 - this is a loop
      – P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3
      – If P1 could somehow release an instance of R2, then we could break the deadlock
    • But P1 is part of a second loop:
      – P_3 \rightarrow R_2 \rightarrow P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3
      – So P1 can't release its instance of R2
  – if the graph contains cycles or loops, then there may be the possibility of deadlock
    • but does a loop guarantee that there is deadlock?
Deadlock Detection

**Example 3:**
- there is a loop:
  - $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- In this case, there is no deadlock
  - either $P_2$ can release an instance of $R_1$, or $P_4$ can release an instance of $R_2$
    - this breaks any possible deadlock cycle
- if the graph contains cycles or loops, then there *may be the possibility* of deadlock, but this is not a guarantee of deadlock
Necessary Conditions for Deadlock

- Deadlock can arise if the following 4 conditions hold simultaneously:
  1. Mutual exclusion  
     - at least 1 resource is held in a non-sharable mode. Other requesting processes must wait until the resource is released
  2. Hold and wait  
     - a process may hold a resource while request (and waiting for) another one
  3. No preemption  
     - resources cannot be preempted and can only be released by the process holding it, after the process is finished. No OS intervention is allowed. A process cannot withdraw its request.
  4. Circular wait  
     - A set of waiting processes \{P_0, ..., P_{n-1}\} must exist such that \(P_i\) waits for a resource held by \(P_{(i+1)\%n}\)
Approaches to Handling Deadlocks

- **Prevention**
  - provide methods to guarantee that at least 1 of the 4 necessary conditions for deadlock does not hold

- **Avoidance**
  - the OS is given advanced information about which process requests which resource when
    - this is used to determine whether the OS can satisfy the resource requests without causing deadlock

- **Detection and Recovery**

- **Ignore and Pretend**
  - this is the most common approach, based on the assumption that deadlock is relatively infrequent
  - UNIX and Windows use this approach
Deadlock Prevention

• Prevent the mutual exclusion condition from coming true
  – This is opposite of our original goal, which was to provide mutual exclusion.
  – Also, many resources are non-sharable and must be accessed in a mutually exclusive way
    • example: a printer should print a file X to completion before printing a file Y. A printer should not print half of file X, and then print the first half of file Y on the same paper
  – thus, it is unrealistic to prevent mutual exclusion
Deadlock Prevention

• Prevent the hold and wait condition from coming true
  – prevent a process from holding resources and requesting others
  – Solution I: request all resources at process creation
  – Solution II: release all held resources before requesting a set of new ones simultaneously
  – Example: a process reads from the DVD drive and writes the file to disk, sorts the file, then sends the file to the printer
    • Solution I: request the DVD drive, disk, and printer at process creation
    • Solution II: break the task down into wholly contained non-dependent pieces
      – obtain the DVD and disk together for the file transfer, then release both together
      – next obtain the disk and printer together for the printing operation, then release both together
Deadlock Prevention

• Disadvantages of Hold-and-wait solutions
  – don’t know in advance all resources needed
  – poor resource utilization
    • a process that is holding multiple resources for a long time may only need each resource for a short time during execution
  – possible starvation
    • a process that needs several popular resources simultaneously may have to wait a very long time
Deadlock Prevention

• Prevent the “No Preemption” condition from coming true
  – allow resources to be preempted
  – Policy I: If a Process X requests a held resource, then all resources currently held by X are released. X is restarted only when it can regain all needed resources
  – Policy II: If a process X requests a resource held by process Y, then preempt the resource from process Y, but only if Y is waiting on another resource. Otherwise, X must wait.
    • the idea is if Y is holding some resources but is waiting on another resource, then Y has no need to keep holding its resources since Y is suspended
  – Disadvantages:
    • these policies don’t apply to all resources, e.g. printers should not be preempted while in the middle of printing, disks should not be preempted while in the middle of writing a block of data
    • can result in unexpected behavior of processes, since an application developer may not know a priori which policy is being used
Deadlock Prevention

• Prevent the circular wait condition from coming true
  – Solution I: a process can only hold 1 resource at a time
    • disadvantage: in some cases, a process needs to hold multiple resources to accomplish a task
  – Solution II: impose a total ordering of all resource types and require each process to request resources in increasing order
    • this prevents a circular wait - see next slide
Deadlock Prevention

• Example of preventing circular waits using ordering:
  – Order all resources into a list: R1, R2, ..., Rm, where R1 < R2 < ... < Rm
    • tape drive = R1
    • disk drive = R2
    • printer = R10
  – Impose the rule that a process holding R_i can only request R_j if R_j > R_i
  – If a process P holds some R_k and requests R_j such that R_j < R_k, then the process must release all such R_k, acquire R_j, then reacquire R_k
    • can lead to poor performance
Deadlock Prevention

• Applying ordering of resources to break circular waiting in the Dining Philosophers Problem
  - R1 < R2 < R3 < R4 < R5
  - Process P1 requests R1, then R2, proceeding Right to Left
  - Process P2 requests R2, then R3, proceeding Right to Left
  - ...
  - Process P5 requests R1, then R5, proceeding *Left then Right due to ordering*
    • thus, P5 blocks on R1, not R5, which breaks any possibility of a circular deadlock - why?