

Deadlock

- Consider this example:

Process A

```
printer->wait( );
disk->wait( );
print file
printer->signal( );
disk->signal( );
```

Process B

```
disk->wait( );
printer->wait( );
print file
disk->signal( );
printer->signal( );
```

- *Deadlock* occurs when two or more processes are each waiting for an event that will never occur, since it can only be generated by another process in that set
- Deadlock is one of the more difficult problems that OS designers face
 - As we examine various approaches to dealing with deadlock, notice the tradeoffs between how well the approach solves the problem, and its performance /OS overhead

1

Fall 2001, Lecture 20

Deadlock (cont.)

- OS must distribute system *resources* among competing processes:
 - CPU cycles preemptable
 - Memory space preemptable
 - Files non-preemptable
 - I/O devices (printer) non-preemptable
- A request for a type of resource can be satisfied by any resource of that type
 - Use any 100 bytes in memory
 - Use either one of two identical printers
- Process *requests* resource(s), *uses* it/them, then *releases* it/them
 - We will assume here that the resource is *re-usable*; it is not *consumed*
 - Waits if resource is not currently available

2

Fall 2001, Lecture 20

Deadlock Conditions

- These 4 conditions are **necessary** and **sufficient** for deadlock to occur:
 - **Mutual exclusion** — if one process holds a resource, other processes requesting that resource must wait until the process releases it (only one can use it at a time)
 - **No preemption** — resources are released voluntarily; neither another process nor the OS can force a process to release a resource
 - **Hold and wait** — processes are allowed to *hold* one (or more) resource and be *waiting* to acquire additional resources that are being held by other processes
 - **Circular wait** — there must exist a set of waiting processes such that P₀ is waiting for a resource held by P₁, P₁ is waiting for a resource held by P₂, ... P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held P₀

3

Fall 2001, Lecture 20

Deadlock Prevention

- Basic idea: ensure that one of the 4 conditions for deadlock can not hold
- **Mutual exclusion** — if one process holds a resource, other processes requesting that resource must wait until the process releases it
 - Hard to avoid mutual exclusion for non-sharable resources
 - Printers
 - Files
 - I/O devices or network connections
 - For printer, avoid mutual exclusion through spooling — then process won't have to wait on physical printer
 - However, many resources are sharable, so deadlock can be avoided for them
 - Read-only files (binaries, perhaps)
 - Most files in your account

4

Fall 2001, Lecture 20

Deadlock Prevention (cont.)

- **No preemption** — resources are released voluntarily; neither another process nor the OS can force a process to release a resource
 - To avoid, allow preemption
 - If process A requests resources that aren't available, see who holds those resources
 - If the holder (process B) is waiting on additional resources, preempt the resource requested by process A
 - Otherwise, process A has to wait
 - » While waiting, some of its current resources may be preempted
 - » Can only wake up when it acquires the new resources plus any preempted resources
 - If a process requests a resource that can not be allocated to it, **all** resources held by that process are preempted
 - Can only wake up when it can acquire all the requested resources
 - Only works for resources whose state can be saved/restored (memory, not printer)

5

Fall 2001, Lecture 20

Deadlock Prevention (cont.)

- **Hold and wait** — processes are allowed to *hold* one (or more) resource and be *waiting* to acquire additional resources that are being held by other processes
 - To avoid, ensure that whenever a process requests a resource, it doesn't hold any other resources
 - Request all resources (at once) at beginning of process execution
 - Process which loops forever?
 - Request all resources (at once) at any point in the program
 - To get a new resource, release all current resources, then try to acquire new one plus old ones all at once
 - Difficult to know what to request in advance
 - Wasteful; ties up resources and reduces resource utilization
 - Starvation is possible

6

Fall 2001, Lecture 20

Deadlock Prevention (cont.)

- **Circular wait** — there must exist a set of waiting processes such that P₀ is waiting for a resource held by P₁, P₁ is waiting for a resource held by P₂, ... P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held P₀
 - To avoid, impose a total order on all resources, and require process to request resource in that order
 - Order: disk drive, printer, CDROM
 - Process A requests disk drive, then printer
 - Process B requests disk drive, then printer
 - Process B does not request printer, then disk drive, which could lead to deadlock
 - Order should be in the logical sequence that the resources are usually acquired
 - Allow process to release all resources, and start request sequence over
 - Or force process to request total number of each resource in a single request

7

Fall 2001, Lecture 20

Dealing with Deadlock

- *The Ostrich Approach* — stick your head in the sand and ignore the problem
- *Deadlock prevention* — prevent deadlock from occurring by eliminating one of the 4 deadlock conditions
- *Deadlock detection* algorithms — detect when deadlock has occurred
 - *Deadlock recovery* algorithms — break the deadlock
- *Deadlock avoidance* algorithms — consider resources currently available, resources allocated to each process, and possible future requests, and only fulfill requests that will not lead to deadlock

8

Fall 2001, Lecture 20

Resource-Allocation Graph

- The deadlock conditions can be modeled using a directed graph called a *resource-allocation graph* (RAG)

- 2 kinds of nodes:

- *Boxes* — represent resources
 - Instances of the resource are represented as dots within the box
- *Circles* — represent processes

- 2 kinds of (directed) edges:

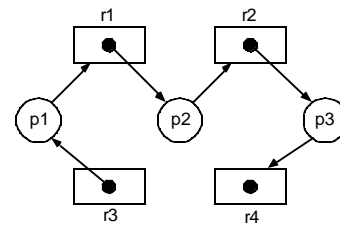
- *Request edge* — from process to resource — indicates the process has requested the resource, and is waiting to acquire it
- *Assignment edge* — from resource instance to process — indicates the process is holding the resource instance

- When a request is made, a request edge is added

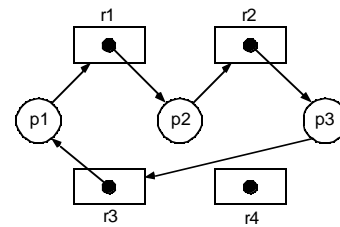
- When request is fulfilled, the request edge is transformed into an assignment edge
- When process releases the resource, the assignment edge is deleted

Interpreting a RAG With Single Resource Instances

- If the graph does **not** contain a cycle, then **no** deadlock exists



- If the graph **does** contain a cycle, then a deadlock **does** exist



- With single resource instances, a cycle is a necessary and sufficient condition for deadlock