Deadlock

- Consider this example:

Process A	Process B
printer–>wait( );	disk–>wait( );
disk->wait( );	printer->wait( );
print file	*print file*
printer->signal( );	disk->signal( );
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

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

Deadlock Conditions

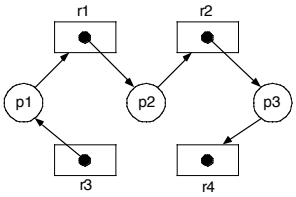
- These 4 conditions are **necessary** and **sufficient** for deadlock to occur:
  - **Mutual exclusion** — if one process holds a resource, other processes requesting that resource must wait until the process releases it (only one can use it at a time)
  - **No preemption** — resources are released 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, … Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held P0

Resource-Allocation Graph

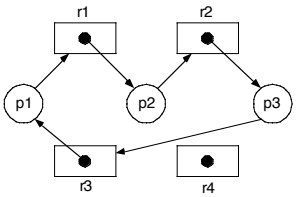
- The deadlock conditions can be modeled using a directed graph called a *resource-allocation graph* (RAG)
  - 2 kinds of nodes:
    - *Boxes* — represent resources
      - Instances of the resource are represented as dots within the box
    - *Circles* — represent processes
  - 2 kinds of (directed) 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

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 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
- *Deadlock detection* algorithms — detect when deadlock has occurred
  - *Deadlock recovery* algorithms — break the deadlock

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

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)

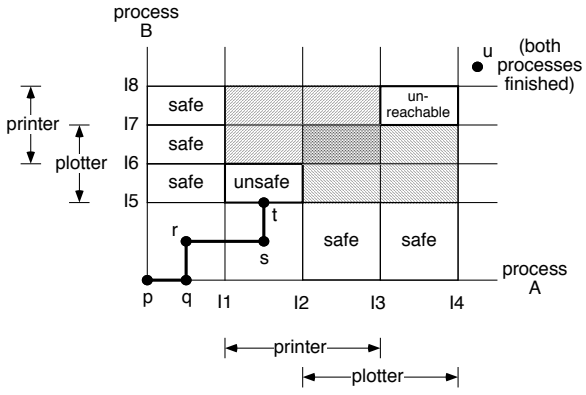
Deadlock Prevention (cont.)

- **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

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, … Pn-1 is waiting for a resource held by Pn, and Pn 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

Deadlock Avoidance — Motivation



- Example to motivate a D.A. algorithm:
  - state p — neither process running
  - state q — scheduler ran A
  - state r — scheduler ran B
  - state s — scheduler ran A, A requested and received printer
  - state t — schedule ran B

Deadlock Avoidance — Motivation (cont.)

- Look at shaded areas:
  - The one shaded "\\\" represents both processes using printer at same time — this is not allowed by mutual exclusion
  - Other ("///") is similar, involving plotter
- Look at box marked "unsafe"
  - If OS enters this box, it will eventually deadlock because it will have to enter a shaded (illegal mutual exclusion) region
    - All paths must proceed up or right (why?)
  - Box is unsafe — should not be entered!
    - From state t, must avoid the unsafe area by going to the right (up to I4) (blocking B)
- At state t, the OS must decide whether or not to grant B's request
  - A good choice will avoid deadlock!
  - Need to know resource needs in advance

Deadlock Avoidance — Safe and Unsafe States

	Has	Max
A	3	9
B	2	4
C	2	7

free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

free: 1
(b)

	Has	Max
A	3	9
B	0	—
C	2	7

free: 5
(c)

	Has	Max
A	3	9
B	0	—
C	7	7

free: 0
(d)

	Has	Max
A	3	9
B	0	—
C	0	—

free: 7
(e)

	Has	Max
A	9	9
B	0	—
C	0	—

free: 1
(f)

	Has	Max
A	0	—
B	0	—
C	0	—

free: 10
(g)

- State (a) is *safe*, meaning there **exists** a sequence of allocations that allows **all** processes to complete:
  - B runs, asks for 2 more resources, 1 free
    - B finishes, releases its resources, 5 free
  - C runs, asks for 5 more resources, 0 free
    - C finishes, releases its resources, 7 free
  - A runs, gets 6 more, everyone done…

Deadlock Avoidance — Safe and Unsafe States (cont.)

	Has	Max
A	3	9
B	2	4
C	2	7

free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

free: 0
(c)

	Has	Max
A	4	9
B	0	—
C	2	7

free: 4
(d)

- Suppose we start in state (a), and reach state (b) by giving A another resource
  - B runs, asks for 2 more resources, 0 free
    - B finishes, releases its resources, 4 free
  - C can't run — might want 5 resources
    - Same for A
- State (b) is unsafe, meaning that from there, deadlock **may** eventually occur
  - State (b) is not a deadlocked state — the system can still run for a bit
  - Deadlock may not occur — A might release one of its resources before asking for more, which allows C to complete

The Banker's Algorithm for Single Resources (Dijkstra, 1965)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

free: 10
(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

free: 2
(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

free: 1
(c)

- A banker has granted lines of credit to customers A, B, C, and D (unit is $1000)
  - She knows it's not likely they will all need their maximum credit at the same time, so she keeps only 10 units of cash on hand
  - At some point in time, the bank is in state (b) above, which is *safe*
    - Can let C finish, have 4 units available
    - Then let B or D finish, etc.
  - But… if banker gives B one more unit (state (c) above), state would be *unsafe* — if everyone asks for maximum credit, **no** requests can be fulfilled

The Banker's Algorithm for Single Resources (cont.)

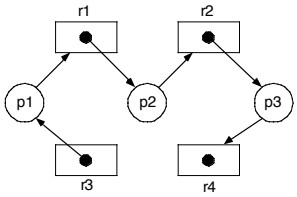
- Resource-request algorithm:
  - The banker considers each request as it occurs, determining whether or not fulfilling it leads to a safe state
    - If it does, the request is granted
    - Otherwise, it is postponed until later
- Safety algorithm:
  - To determine if a state is safe, the banker checks to see if she has enough resources to satisfy some customer
    - If so, she assumes those loans will be repaid (i.e., the process will use those resources, finish, and release all of its resources), and she checks to see if she has enough resources to satisfy another customer, etc.
  - If all loans can eventually be repaid, the state is safe and the initial request can be granted

Evaluation of Deadlock Avoidance Using the Banker's Algorithm

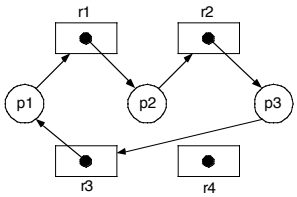
- Advantages:
  - No need to preempt resources and rollback state (as in deadlock detection & recovery)
  - Less restrictive than deadlock prevention
- Disadvantages:
  - Maximum resource requirement for each process must be stated in advance
  - Processes being considered must be independent (i.e., unconstrained by synchronization requirements)
  - There must be a fixed number of resources (i.e., can't add resources, resources can't break) and processes (i.e., can't add or delete processes)
  - Huge overhead — must use the algorithm every time a resource is requested

Interpreting a RAG With Single Resource Instances (Review)

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



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



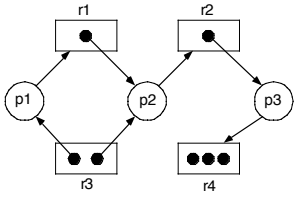
- With **single** resource instances, a **cycle** is a **necessary** and **sufficient** condition for deadlock

Deadlock Detection (Single Resource of Each Type)

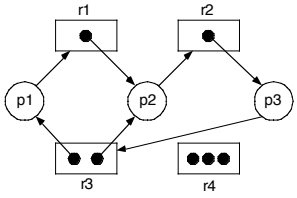
- If all resources have only a single instance, deadlock can be detected by searching the resource-allocation graph for cycles
  - Silberschatz defines a simpler graph, called the *wait-for* graph, and searches that graph instead
    - The wait-for graph is the resource-allocation graph, minus the resources
    - An edge from p1 to p2 means p1 is waiting for a resource that p2 holds (here we don't care which resource is involved)
- One simple algorithm:
  - Start at each node, and do a depth-first search from there
  - If a search ever comes back to a node it's already found, then it has found a cycle

Interpreting a RAG With Multiple Resource Instances

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



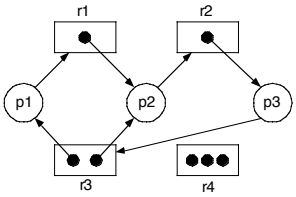
- If the graph **does** contain a cycle, then a deadlock **may** exist



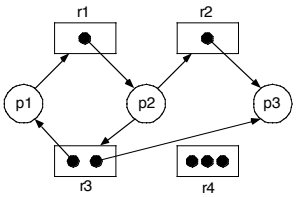
- With **multiple** resource instances, a **cycle** is a **necessary** (but not **sufficient**) condition for deadlock

Interpreting a RAG With Multiple Resource Instances (cont.)

- If the graph **does** contain a knot (and a cycle), then a deadlock **does** exist



- If the graph **does not** contain a knot, then a deadlock **does not** exist



- With **multiple** resource instances, a **knot** is a **sufficient** condition for deadlock

Deadlock Detection (Multiple Resources of Each Type)

- This algorithm (Coffman, 1971) uses the following data structures:

Existing Resources (E1, E2, E3, …, Em)

Available Resources (A1, A2, A3, …, Am)

Current Allocation

$$\begin{bmatrix} C11 & C12 & C13 & \cdots & C1m \\ C21 & C22 & C23 & \cdots & C2m \\ \vdots & \vdots & \vdots & & \vdots \\ Cn1 & Cn2 & Cn3 & \cdots & Cnm \end{bmatrix}$$

Request

$$\begin{bmatrix} R11 & R12 & R13 & \cdots & R1m \\ R21 & R22 & R23 & \cdots & R2m \\ \vdots & \vdots & \vdots & & \vdots \\ Rn1 & Rn2 & Rn3 & \cdots & Rnm \end{bmatrix}$$

- n processes, m types of resources
  - **Existing Resources** vector tells number of resources of each type that exist
  - **Available Resources** vector tells number of resources of each type that are available (unassigned to any process)
  - i-th row of **Current Allocation** matrix tells number of resources of each type allocated (assigned) to process i

Deadlock Detection (Multiple Resources of Each Type) (cont.)

- Every resource is either allocated or available
  - Number of resources of type j that have been allocated to all processes, plus number of resources of type j that are available, should equal number of resources of type j in existence
- Processes may have unfulfilled requests
  - i-th row of **Request** matrix tells number of resources of each type process i has requested, but not yet received
- Notation: comparing vectors
  - If A and B are vectors, the relation $A \leq B$ means that each element of A is less than or equal to the corresponding element of B (i.e., $A \leq B$ iff $A_i \leq B_i$ for $0 \leq i \leq m$)
  - Furthermore, $A < B$ iff $A \leq B$ and $A \neq B$

Deadlock Detection Algorithm (Multiple Resources of Each Type)

- Operation:
  - Every process is initially unmarked
  - As algorithm progresses, processes will be marked, which indicates they are able to complete, and thus are not deadlocked
  - When algorithm terminates, any unmarked processes are deadlocked
- Algorithm:
  1. Look for an unmarked process Pi for which the i-th row of the **Request** matrix is less than or equal to the **Available** vector
  2. If such a process is found, add the i-th row of the **Current** matrix to the **Available** vector, mark the process, and go back to step 1
  3. If no such process exists, the algorithm terminates

Deadlock Detection Example (Multiple Resources of Each Type)

Existing Resources (4 2 3 1)

Available Resources (2 1 0 0)

Current Allocation

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

resources = (tape drive plotter printer CDROM)

- Whose request can be fulfilled?
  - Process 1 — no — no CDROM available
  - Process 2 — no — no printer available
  - Process 3 — yes — give it the requested resources, and after it completes and releases those resources, A = (2 2 2 0)
  - Process 1 still can't run (no CDROM), but process 2 can run, giving A = (4 2 2 1)
  - Process 1 can run, giving A = (4 2 3 1)

After Deadlock Detection: Deadlock Recovery

- How often does deadlock detection run?
  - After every resource request?
  - Less often (e.g., every hour or so, or whenever resource utilization gets low)?
- What if OS detects a deadlock?
  - Terminate a process
    - All deadlocked processes
    - One process at a time until no deadlock
      - Which one?
      - One with most resources?
      - One with less cost?
        - » CPU time used, needed in future
        - » Resources used, needed
      - That's a choice similar to CPU scheduling
    - Is it acceptable to terminate process(es)?
      - May have performed a long computation
        - » Not ideal, but OK to terminate it
      - Maybe have updated a file or done I/O
        - » Can't just start it over again!

After Deadlock Detection: Deadlock Recovery (cont.)

- Any less drastic alternatives?
  - Preempt resources
    - One at a time until no deadlock
    - Which "victim"?
      - Again, based on cost, similar to CPU scheduling
    - Is rollback possible?
      - *Preempt* resources — take them away
      - *Rollback* — "roll" the process back to some safe state, and restart it from there
        - » OS must *checkpoint* the process frequently — write its state to a file
      - Could roll back to beginning, or just enough to break the deadlock
        - » This second time through, it has to wait for the resource
        - » Has to keep multiple checkpoint files, which adds a lot of overhead
    - Avoid starvation
      - May happen if decision is based on same cost factors each time
      - Don't keep preempting same process (i.e., set some limit)

Evaluating the Approaches to Dealing with Deadlock

- *The Ostrich Approach* — ignoring the problem
  - Good solution if deadlock isn't frequent
- *Deadlock prevention* — eliminating one of the 4 deadlock conditions
  - May be overly restrictive
- *Deadlock avoidance* — only fulfilling requests that will not lead to deadlock
  - Need too much a priori information, not very dynamic (can't add processes or resources), huge overhead
- *Deadlock detection and recovery* — detect when deadlock has occurred, then break the deadlock
  - Tradeoff between frequency of detection and performance / overhead added