EECS 111:

System Software

Lecture: Deadlocks

Prof. Mohammad Al Faruque

The Henry Samueli School of Engineering
Electrical Engineering & Computer Science
University of California Irvine (UCI)

Outline

- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

The Deadlock Problem

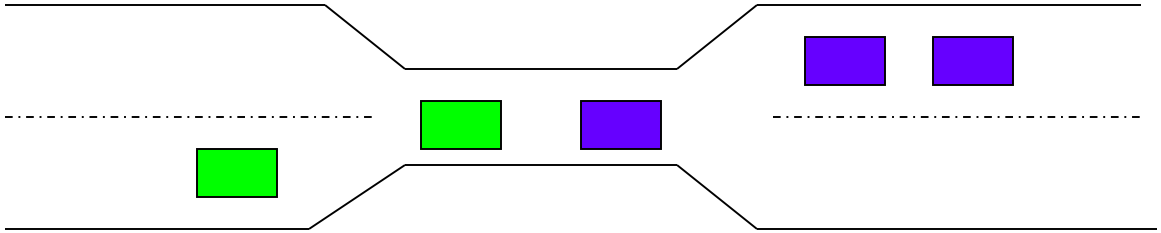
- **A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.**
  - **Example 1**
    - System has 2 tape drives. P1 and P2 each hold one tape drive and each needs the other one.
  - **Example 2**
    - Semaphores A and B each initialized to 1

P0	*P1*
wait(A)	*wait(B)*
wait(B)	*wait(A)*

Definitions

- **A process is *deadlocked* if it is waiting for an event that will never occur.**
  - Typically, more than one process will be involved in a deadlock (the deadly embrace).
- **A process is *indefinitely postponed* if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes,**
  - i.e. the process is ready to proceed but never gets the CPU.

Example - Bridge Crossing



- Assume traffic in one direction.
  - Each section of the bridge is viewed as a resource.
- If a deadlock occurs, it can be resolved only if one car backs up (preempt resources and rollback).
  - Several cars may have to be backed up if a deadlock occurs.
  - Starvation is possible

Resources

- **Resource**
  - commodity required by a process to execute
- **Resources can be of several types**
  - Serially Reusable Resources
    - CPU cycles, memory space, I/O devices, files
    - acquire -> use -> release
  - Consumable Resources
    - Produced by a process, needed by a process - e.g. Messages, buffers of information, interrupts
    - create →acquire →use
    - Resource ceases to exist after it has been used

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request → use → release

Deadlock Characterization

- **Deadlock can arise if four conditions hold simultaneously.**
  - **Mutual exclusion: only one process at a time can use a resource**
  - **Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes**
  - **No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task**
  - **Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.**

Resource-Allocation Graph

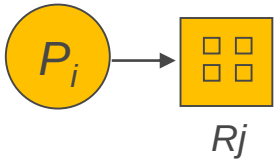
- A set of vertices *V* and a set of edges *E*.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

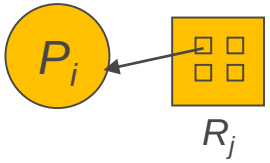
- Process

- Resource Type with 4 instances

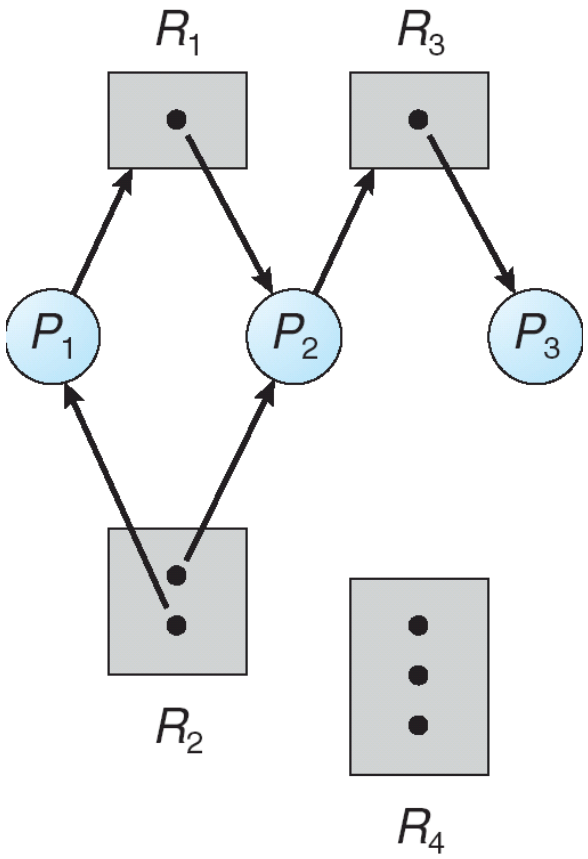
- P_i requests instance of R_j



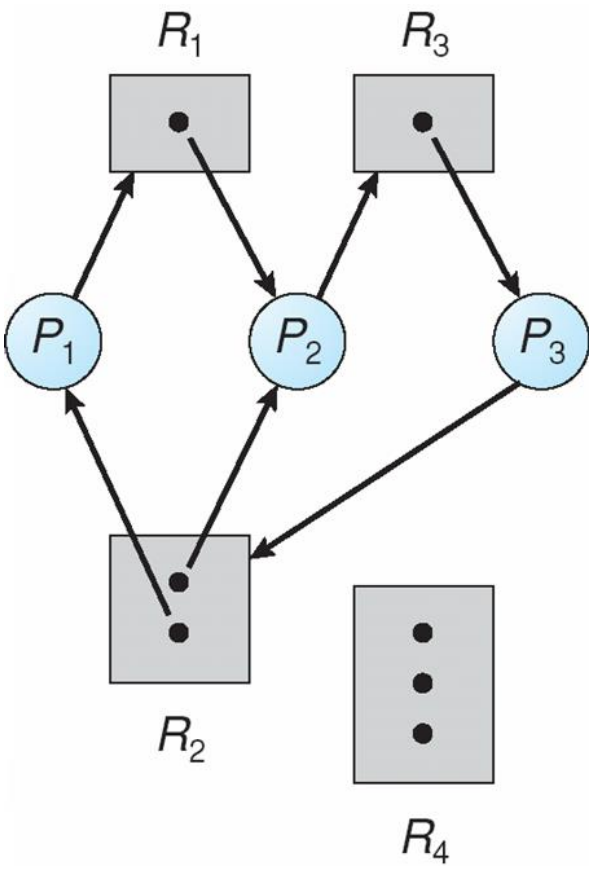
- P_i is holding an instance of R_j



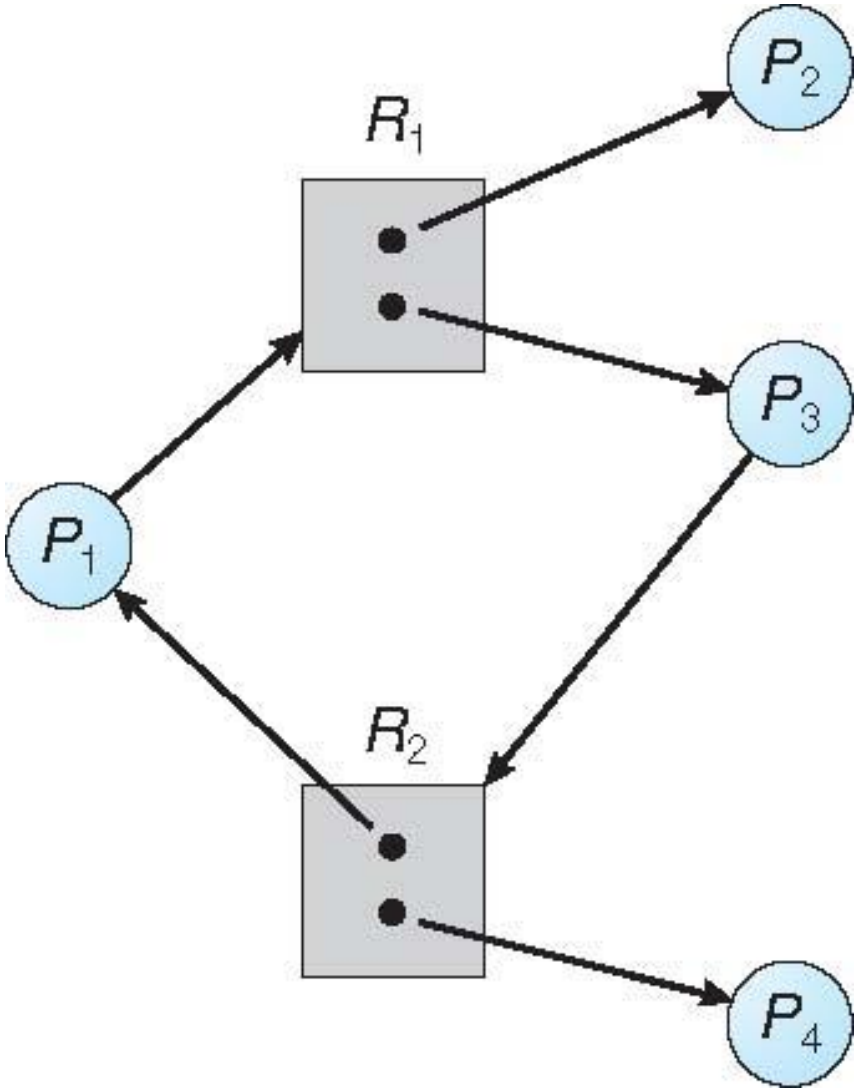
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  1. if only one instance per resource type, then deadlock
  2. if several instances per resource type, possibility of deadlock

Outline

- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

Methods for Handling Deadlocks

1. **Ensure that the system will *never* enter a deadlock state**

2. **Allow the system to enter a deadlock state and then recover**

3. **Ignore the problem and pretend that deadlocks never occur in the system;**
   - **used by most operating systems, including UNIX**

Deadlock Management

1. **Prevention**
   - Design the system in such a way that deadlocks can never occur
2. **Avoidance**
   - Impose less stringent conditions than for prevention, allowing the possibility of deadlock but sidestepping it as it occurs.
3. **Detection**
   - Allow possibility of deadlock, determine if deadlock has occurred and which processes and resources are involved.
4. **Recovery**
   - After detection, clear the problem, allow processes to complete and resources to be reused. May involve destroying and restarting processes.

Outline

- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

Deadlock Prevention

- **If any one of the conditions for deadlock (with reusable resources) is denied, deadlock is impossible.**
- **Restrain ways in which requests can be made**
  - **Mutual Exclusion**
    - non-issue for sharable resources
    - cannot deny this for non-sharable resources (important)
- **Hold and Wait - guarantee that when a process requests a resource, it does not hold other resources.**
  - Force each process to acquire all the required resources at once. Process cannot proceed until all resources have been acquired.
    1. Low resource utilization,
    2. starvation possible

Deadlock Prevention (Cont.)

- **No Preemption –**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- **Circular Wait –**
  - impose a total ordering of all resource types, and
  - Require that processes request resources in increasing order of enumeration; if a resource of type N is held, process can only request resources of types > N.

Self study → proof

Outline

- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

Deadlock Avoidance

- **Requires that the system has some additional *a priori* information available**
  - **Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need**
  - **The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition**
  - **Resource-allocation *state* is defined by**
    - the number of available and allocated resources, and
    - the maximum demands of the processes

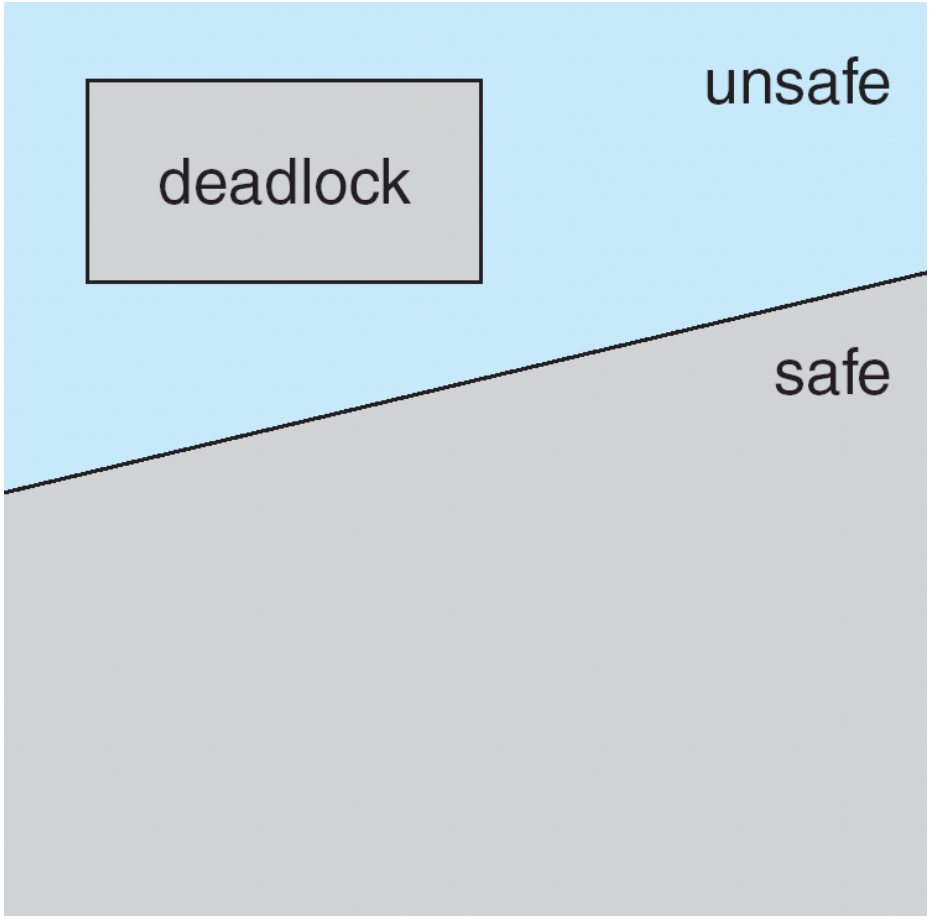
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



Avoidance algorithms

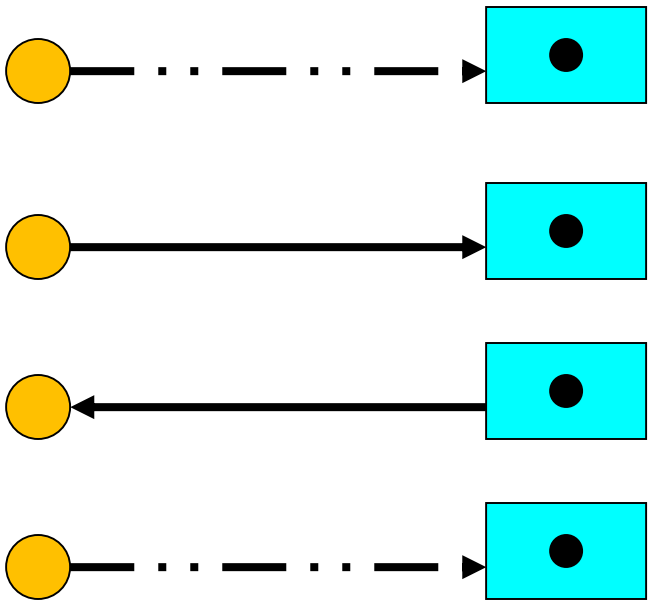
1. **Single instance of a resource type**
   - **Use a resource-allocation graph**

2. **Multiple instances of a resource type**
   - **Use the banker's algorithm**

Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system

Claim Graph



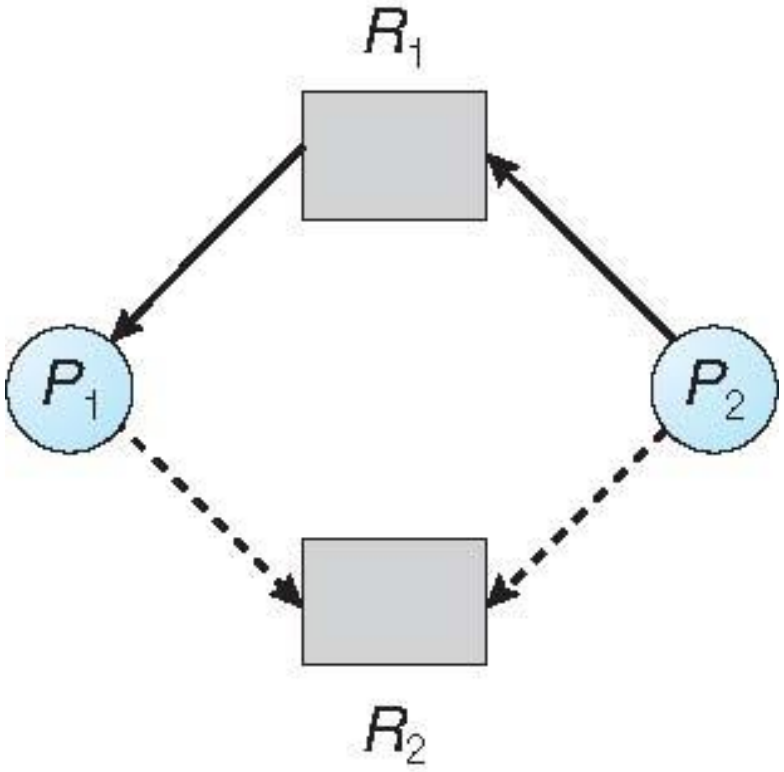
Process claims resource

Process requests resource

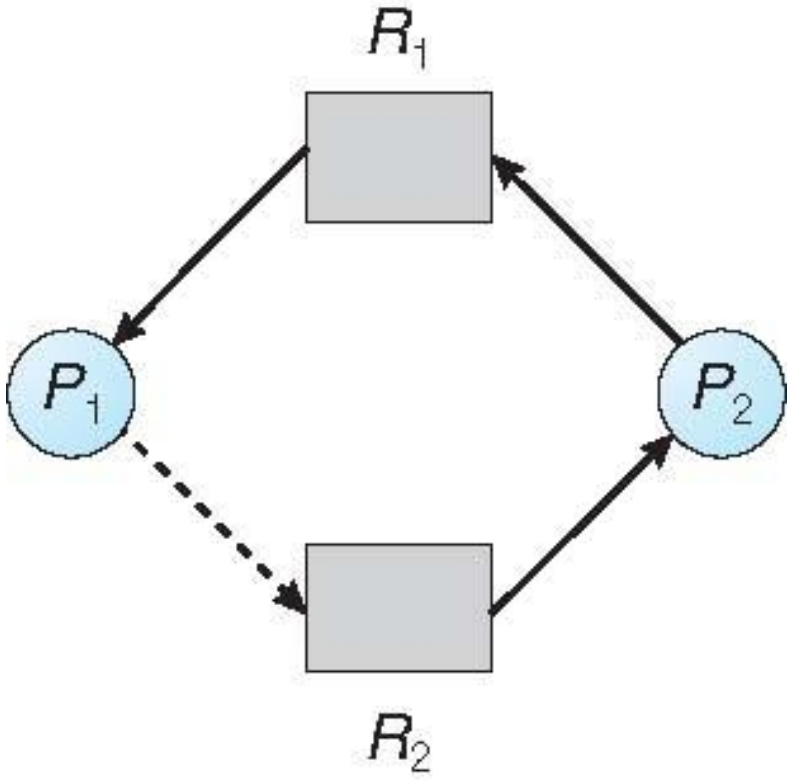
Process is assigned resource

Process releases resource

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Used for multiple instances of each resource type.
- Each process must a priori claim maximum use of each resource type.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

- **Let *n* = number of processes and**
- ***m* = number of resource types.**
  - *Available*: Vector of length *m*. If *Available*[*j*] = *k*, there are *k* instances of resource type *Rj* available.
  - *Max*: $n \times m$ matrix. If *Max*[*i*,*j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj*.
  - *Allocation*: $n \times m$ matrix. If *Allocation*[*i*,*j*] = *k*, then process *Pi* is currently allocated *k* instances of resource type *Rj*.
  - *Need*: $n \times m$ matrix. If *Need*[*i*,*j*] = *k*, then process *Pi* may need *k* more instances of resource type *Rj* to complete its task.

Need[*i*,*j*] = *Max*[*i*,*j*] - *Allocation*[*i*,*j*]

Safety Algorithm

1. **STEP 1: Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize**
   - *Work* := *Available*
   - *Finish*[*i*] := *false* for *i* = 1,2,…,n.
2. **STEP 2: Find an *i* (i.e. process *Pi*) such that both:**
   - *Finish*[*i*] = *false*
   - *Need_i* <= *Work*
   - If no such *i* exists, go to step 4.
3. **STEP 3: *Work* := *Work* + Allocation_*i***
   - *Finish*[*i*] := *true*
   - go to step 2
4. **STEP 4: If *Finish*[*i*] = *true* for all *i*, then the system is in a safe state.**

Resource-Request Algorithm for Process *Pi*

- **Request_i = request vector for process Pi. If Request_i[j] = k, then process Pi wants k instances of resource type Rj.**
  1. STEP 1: If *Request*(*i*) ≤ *Need*(*i*), go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
  2. STEP 2: If *Request*(*i*) ≤ *Available*, go to step 3. Otherwise, Pi must wait since resources are not available.
  3. STEP 3: Pretend to allocate requested resources to Pi by modifying the state as follows:

     Available := *Available* - *Request* (i);

     Allocation (*i*) := *Allocation* (*i*) + *Request* (*i*);

     Need (*i*) := *Need* (*i*) - *Request* (*i*);

     - If safe ⇒ resources are allocated to *Pi*.
     - If unsafe ⇒ *Pi* must wait and the old resource-allocation state is restored.

Example of Banker's Algorithm

- **5 processes**
  - P0 - P4;
- **3 resource types**
  - A(10 instances), B (5 instances), C (7 instances)
- **Snapshot at time T0**

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Example (cont.)

- The content of the matrix *Need* is defined to be *Max* - *Allocation*.
- The system is in a safe state since the sequence <*P*1,*P*3,*P*4,*P*2,*P*0> satisfies safety criteria.

	Need		
	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Example: P1 requests (1,0,2)

- **Check to see that Request ≤ Available**
  - ((1,0,2) ≤ (3,3,2)) ⇒ true.

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Example (cont.)

- Executing the safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Outline

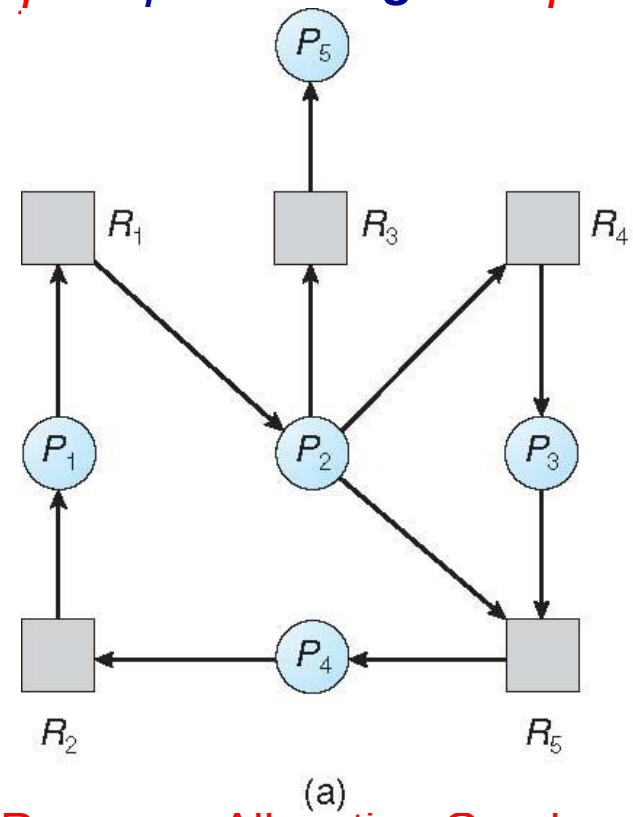
- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

Deadlock Detection

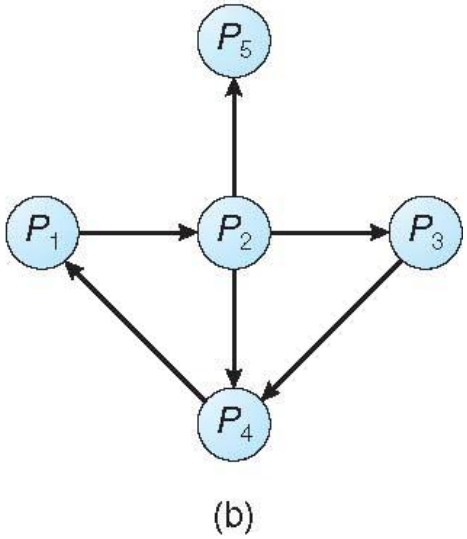
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Different from Resource-Allocation Graph
- Maintain wait-for graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j



Resource-Allocation Graph



Corresponding wait-for graph

Single Instance of Each Resource Type

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Several Instances of a Resource Type

- **Data Structures**
  - *Available*: Vector of length *m*. If *Available*[*j*] = *k*, there are *k* instances of resource type *Rj* available.
  - *Allocation*: *n* × *m* matrix. If *Allocation*[*i*,*j*] = *k*, then process *Pi* is currently allocated *k* instances of resource type *Rj*.
  - *Request* : An *n* × *m* matrix indicates the current request of each process. If *Request* [*i*,*j*] = *k*, then process *Pi* is requesting *k* more instances of resource type *Rj* .

Deadlock Detection Algorithm

1. **Step 1:** Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively. Initialize
   - ***Work*** := ***Available***
   - For $i = 1,2,\ldots,n$, if $Allocation(i) \neq 0$, then $Finish[i] := false$, otherwise $Finish[i] := true$.
2. **Step 2:** Find an index ***i*** such that both:
   - $Finish[i] = false$
   - $Request\ (i) \leq Work$
   - If no such ***i*** exists, go to step 4.

Deadlock Detection Algorithm

3. Step 3: *Work* := *Work* + Allocation(*i*)

- ***Finish*[*i*] := *true***
- **go to step 2**

4. Step 4: If *Finish*[*i*] = *false* for some *i*, $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if *Finish*[*i*] = *false*, then *Pi* is deadlocked.

Algorithm requires an order of m × (n^2) operations to detect whether the system is in a deadlocked state.

Example of Detection Algorithm

- 5 processes - *P*0 - *P*4; 3 resource types - *A*(7 instances), *B*(2 instances), *C*(6 instances)
- Snapshot at time *T* 0: <*P*0,*P*2,*P*3,*P*1,*P*4> will result in *Finish*[*i*] = true for all *i*.

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Example (cont.)

- **P2 requests an additional instance of type C.**
- **State of system**
  - Can reclaim resources held by process P0, but insufficient resources to fulfill other processes' requests.
  - Deadlock exists, consisting of *P* 1,*P* 2,*P* 3 and *P* 4.

	Request		
	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

Detection-Algorithm Use

- **When, and how often to invoke depends on:**
  - **How often a deadlock is likely to occur?**
  - **How many processes will need to be rolled back?**
    - One for each disjoint cycle
- **How often --**
  - **Every time a request for allocation cannot be granted immediately**
    - Allows us to detect set of deadlocked processes and process that "caused" deadlock. Extra overhead.
    - Every hour or whenever CPU utilization drops.
  - **With arbitrary invocation there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes "caused" the deadlock.**

Outline

- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

Recovery from Deadlock: 1) Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?
- This is a policy decision!!!

Recovery from Deadlock: 2) Resource Preemption

1. **Selecting a victim – minimize cost**
2. **Rollback – return to some safe state, restart process for that state**
3. **Starvation – same process may always be picked as victim, include number of rollback in cost factor**

Outline

- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

Combined approach to deadlock handling

- **Combine the three basic approaches**
  - Prevention
  - Avoidance
  - Detection

  → allowing the use of the optimal approach for each class of resources in the system.
- **Partition resources into hierarchically ordered classes.**
  - **Use most appropriate technique for handling deadlocks within each class.**

References

Part of the contents of this lecture has been adapted from the book Abraham Silberschatz, Peter B. Galvin, Greg Gagne: "Operating System Concept ", Publisher : Wiley; 9 edition (December 17, 2012), ISBN-13: 978-1118063330

Slides also contain lecture materials from John Kubiatowicz (Berkeley), John Ousterhout (Stanford), Nalini (UCI), Rainer (UCI), and others

Some slides adapted from http://www-inst.eecs.berkeley.edu/~cs162/ Copyright © 2010 UCB

Thank you for your attention