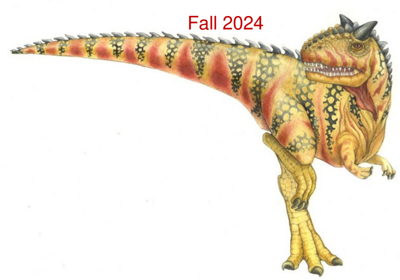
Lecture 9: Deadlocks

Hossein Asadi (asadi@sharif.edu)

Rasool Jalili (jalili@sharif.edu)

Fall 2024



Lecture 9: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Lecture Objectives

- To Develop a Description of Deadlocks, which Prevent sets of Concurrent Processes from Completing their Tasks
- To Present a number of Different Methods for Preventing or Avoiding Deadlocks in a Computer System

System Model

- System Consists of Resources
  - To be distributed among a number of competing processes
- Resource Types R_1, R_2, . . ., R_m

  E.g,: 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**
    - Accomplished using system calls such as request() and release() device or open() and close() file, OR allocate() and free() memory

System Model (cont.)

- Physical Resources
  - Printers, tape drives, memory space, or CPU cores/cycles
- Logical Resources
  - Files, semaphores, and monitors
- Example: Consider a system with 3 CD RW drives and 3 running processes
  - Each process holds one CD RW drive and now requests another drive ➔ Deadlock
- Multithreaded Programs:
  - Good candidate for deadlock
  - Multiple threads compete for shared resources

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 process holding it, after that process has completed its task

Deadlock Characterization (cont.)

- **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, …, 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.

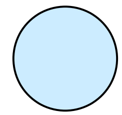
Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

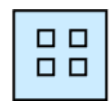
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, set consisting of all processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, set consisting of all resource types in the system
- **Request Edge** – directed edge $P_i \rightarrow R_j$
- **Assignment Edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (cont.)

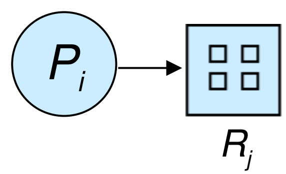
- Process



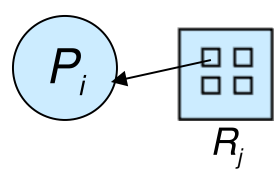
- Resource Type with 4 instances



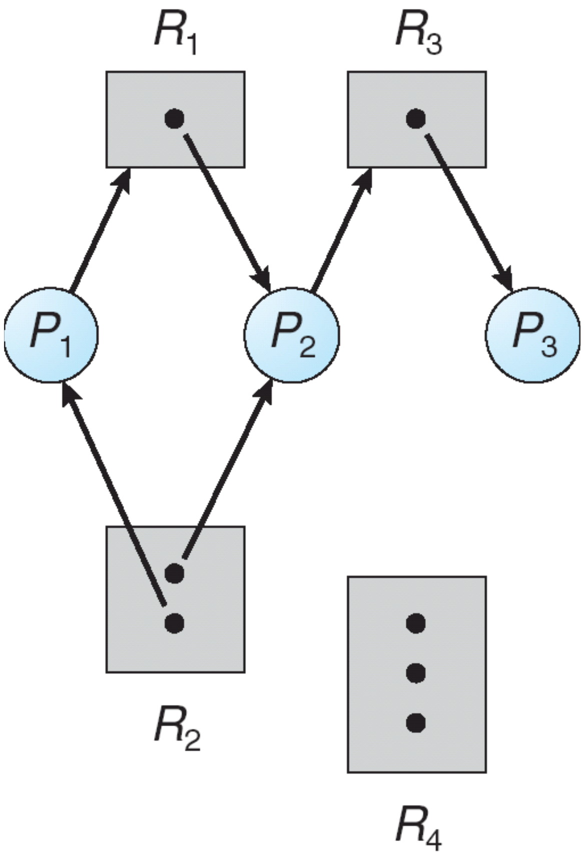
- P_i requests instance of R_j



- P_i is holding an instance of R_j

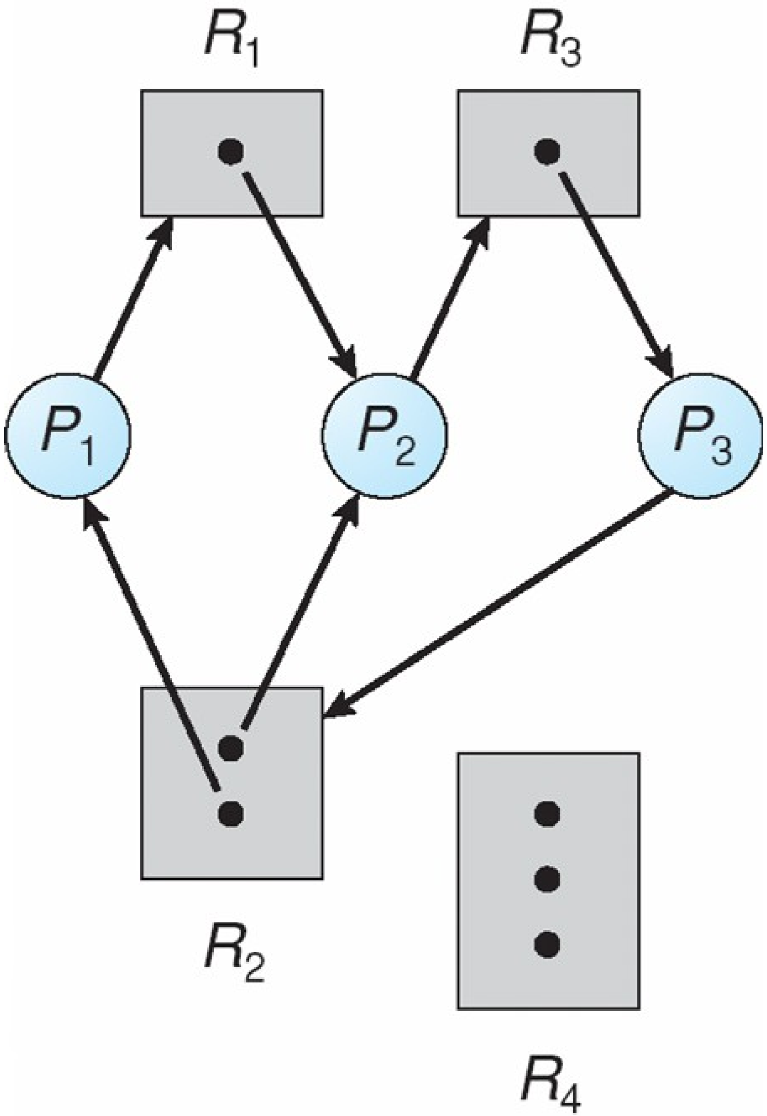


Example of a Resource Allocation Graph

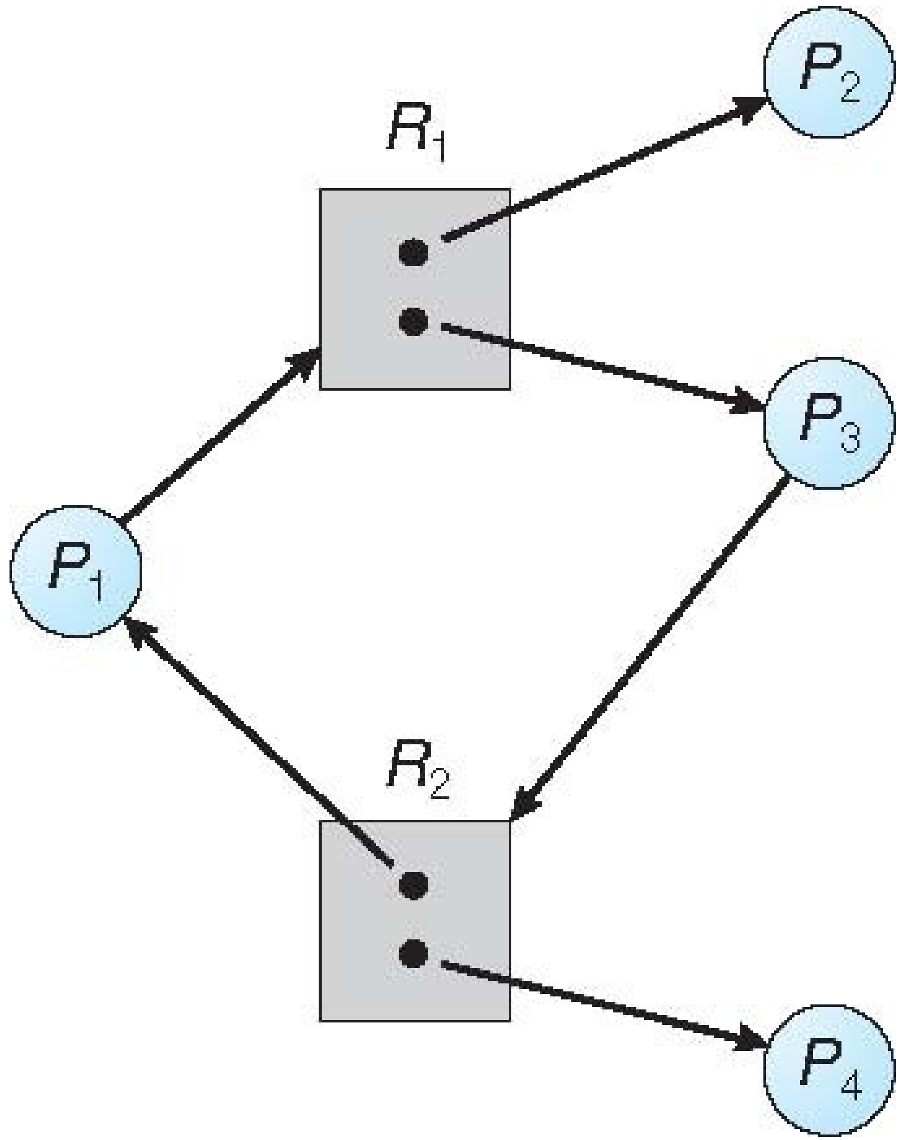


- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
  - R1: one instance, R2 two instances, R3 one instance, and R4 three instances
- E = {P1➔R1, P2➔R3, …., R3➔P3}

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 ⇒
  - Only one instance per resource type ➔ deadlock
  - Cycle involves only a set of resource types, each of which has only a single instance ➔ deadlock
    - In the above two cases, cycle is necessary & sufficient condition for existence of deadlock
  - Several instances per resource type ➔ possibility of deadlock
    - In this case, cycle is necessary but not sufficient

Methods for Handling Deadlocks

- Ensure that System will ***Never*** Enter a Deadlock State
  - Deadlock prevention: try to violate one of necessary conditions for deadlock
  - Deadlock avoidance: try to regulate how/when requests can be made to acquire resources
    - More conservative approach than deadlock prevention
- Allow System to enter a Deadlock State and then recover

Methods for Handling Deadlocks (cont.)

- Ignore Problem and Pretend that Deadlocks Never occur in system
  - Used by most OSes, including UNIX
  - Up to application developer to detect and handle deadlocks
- What if Deadlocks are not Resolved?
  - Deterioration of system performance
    - Eventually need a manual restart
  - Deadlock occur very infrequent ➔ cheaper approach in mainstream applications
    - Instead of employing prevention, avoidance, or detection and recovery methods

Deadlock Prevention

- **Mutual Exclusion**
  - Not required for sharable resources
  - A process never needs to wait for a sharable resources
  - Must hold for non-sharable resources
  - Example
    - Read-only files

Deadlock Prevention (cont.)

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Solution 1: Require process to request and be allocated all its resources before it begins execution
  - Solution 2: Or allow process to request resources only when process has none allocated to it
  - Cons
    - Low resource utilization ☹
    - 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 list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as new ones
  - This protocol applicable only to resources whose state can be easily saved and restored later
    - CPU registers and memory space: applicable ☺
    - Printers and tape drives: not (easily) applicable ☹

Deadlock Prevention (cont.)

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - A process which holds R(i), can request instance of R(j) if F(Rj) > F(Ri)
  - Ensuring order by application developer
    - Can use lock-order verifier (e.g., witness in FreeBSD)
- Example
  - F(tape)=1, F(disk drive)=5, and F(printer)=12

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/*************************************/
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

- Lock Ordering does not Guarantee Deadlock Prevention if Locks can be acquired Dynamically
  - Ordering is broken with unordered arguments

```
void transaction(Account from, Account to, double amount)
{  mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

- Transactions 1 and 2 execute concurrently.
- Transaction 1 transfers $25 from account A to account B, & Transaction 2 transfers $50 from account B to account A

Deadlock Avoidance

Requires that system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare ***maximum number*** of resources of each type that it may need
- Deadlock-avoidance algorithm dynamically examines resource-allocation state to ensure that there can never be a circular-wait condition
- "Resource-allocation *state"* is defined by number of available and allocated resources, and maximum demands of processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ (aka, safe sequence) of ALL processes in systems such that for each P_i, resources that P_i can still request can be satisfied by currently available resources + resources held by all P_j, with $j<i$

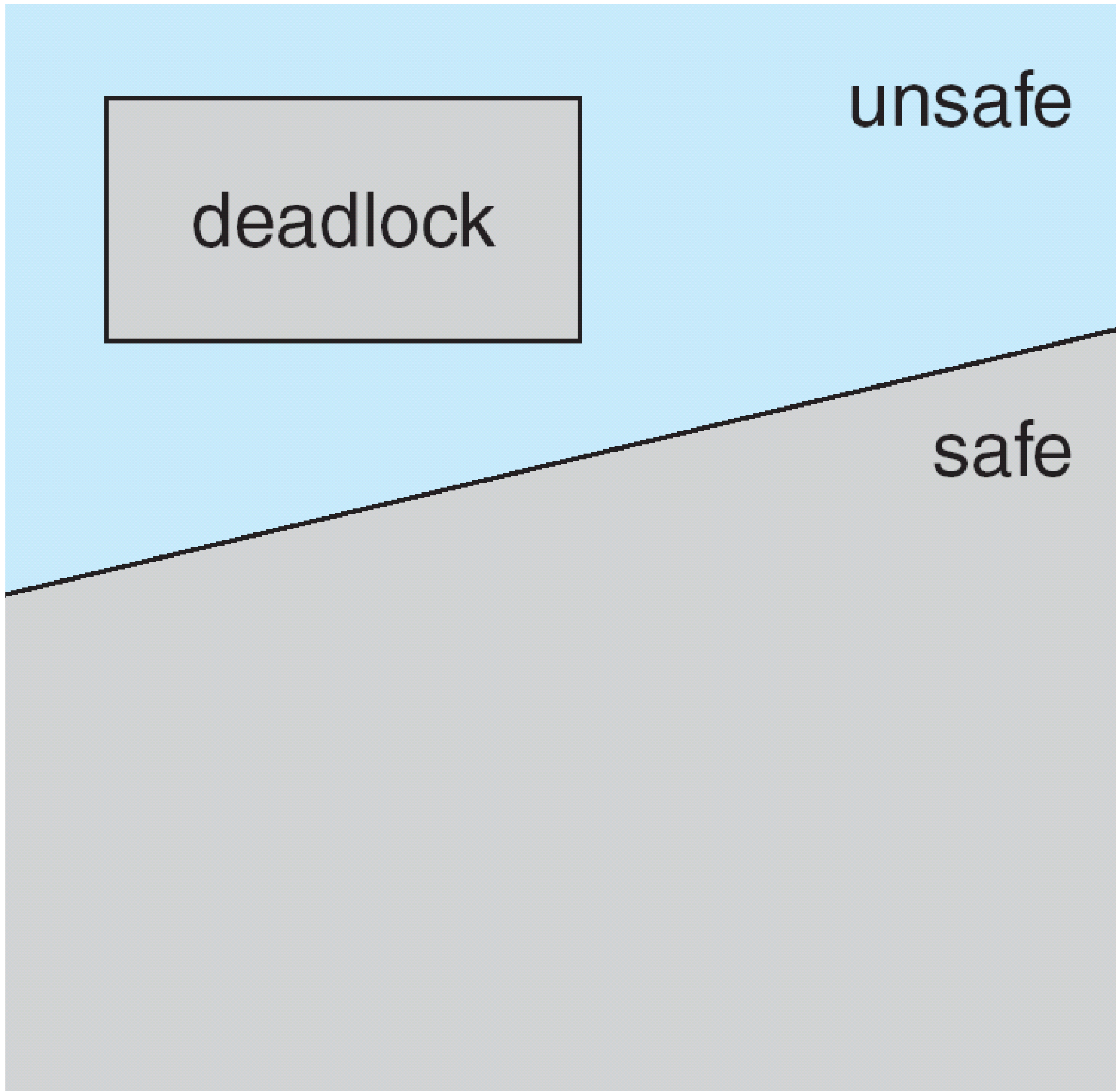
Safe State (cont.)

- 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

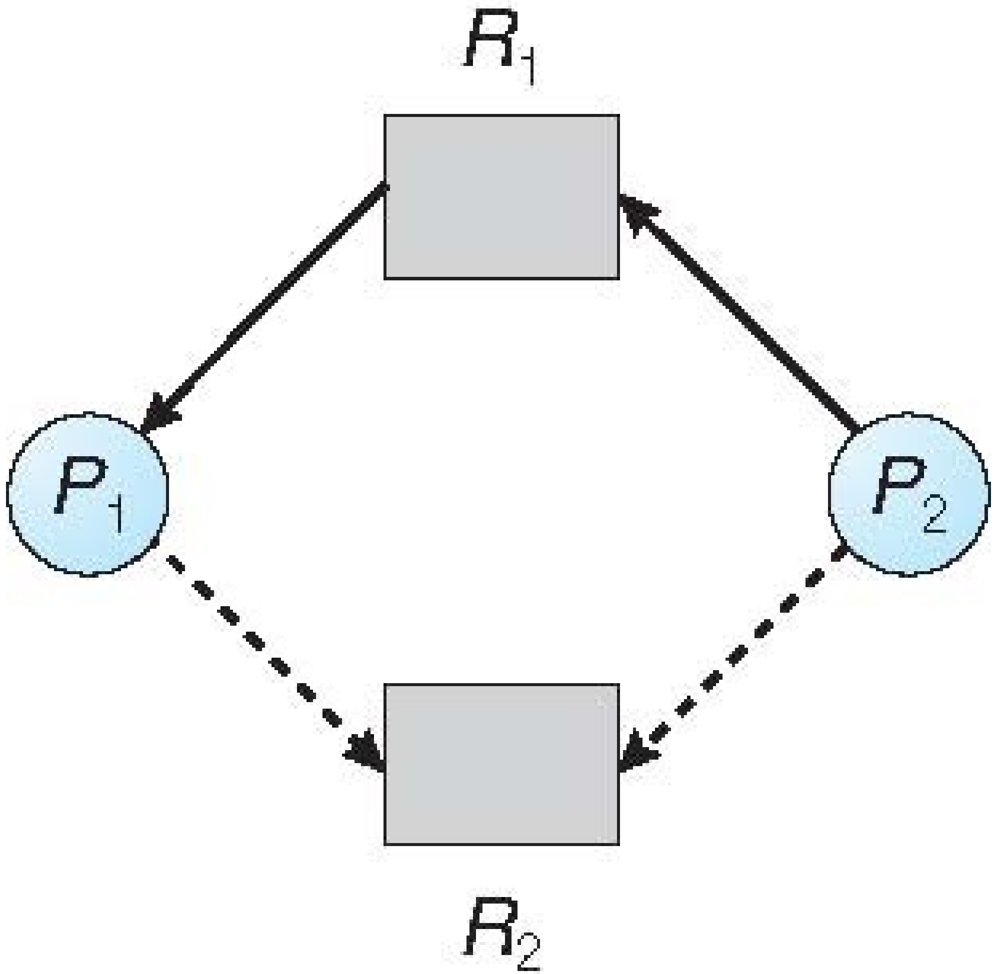
- Single Instance of a Resource Type
  - Use a resource-allocation graph

- Multiple Instances of a Resource Type
  - Use banker's algorithm
    - Reading assignment

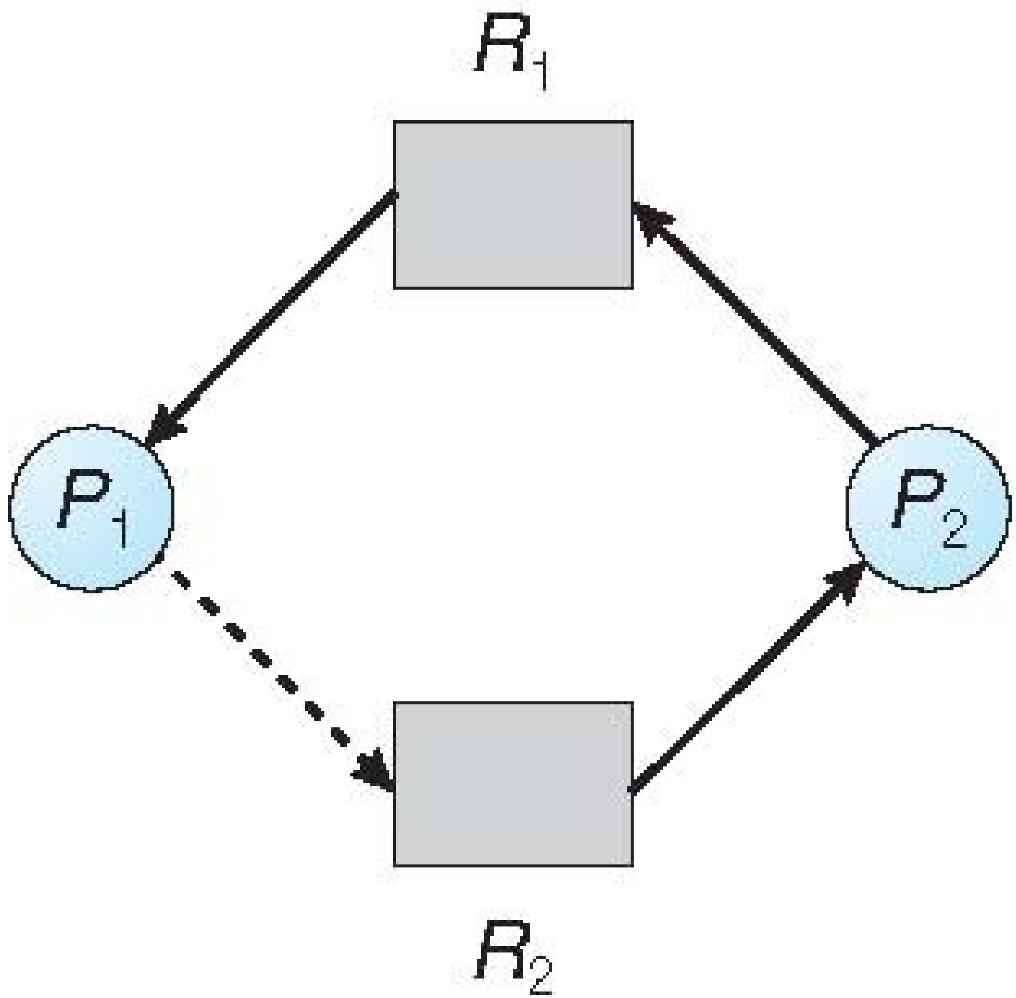
Resource-Allocation Graph Scheme

- **Claim edge**
  - $P_i \rightarrow R_j$ indicates that process P_i 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 resource is allocated to process
- Resource is Released by a process ➔ Assignment Edge reconverts to a claim edge
- Resources must be claimed *a priori* in system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Why not assigning R2 to P2 after making sure that P1 doesn't need R2 in the near future?

Resource-Allocation Graph Algorithm

- Suppose that P_i requests a R_j
- Request can be Granted only if Converting Request Edge to an Assignment Edge does not Result in Formation of a Cycle in Resource Allocation Graph
- Algorithm Complexity $O(n^2)$
- Low Resource Utilization
  - Resources might be available and not be allocated to processes

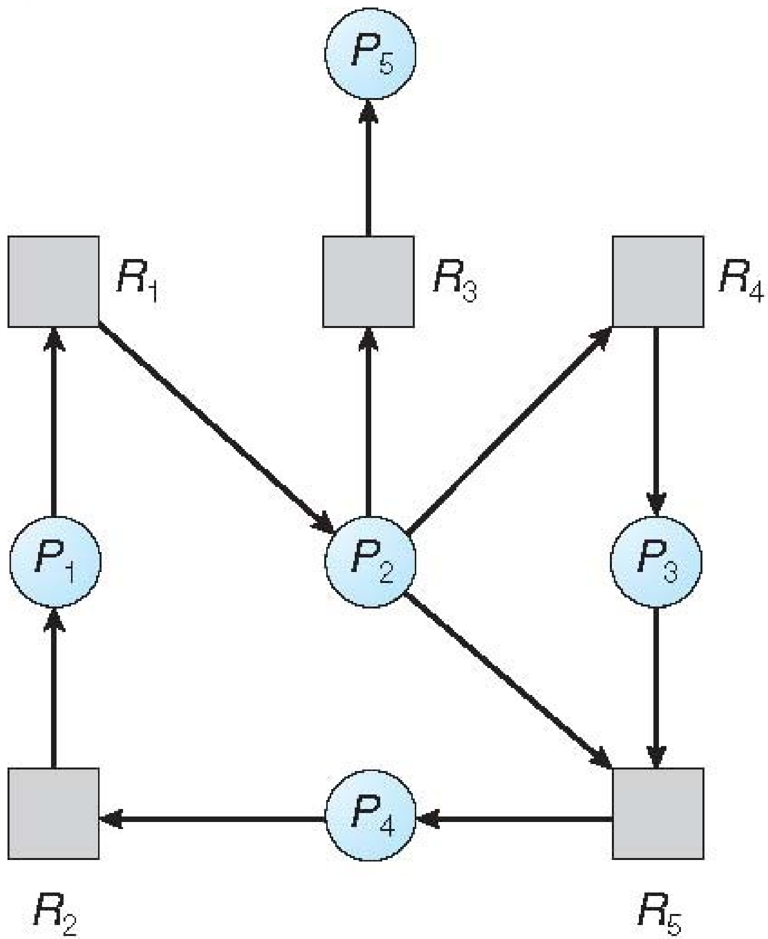
Deadlock Detection

- Allow System to Enter Deadlock State
- Detection Algorithm
  - Single instance of each resource type
  - Multiple instances of a resource type
    - Reading assignment
- Recovery Scheme

Single Instance of Each Resource Type

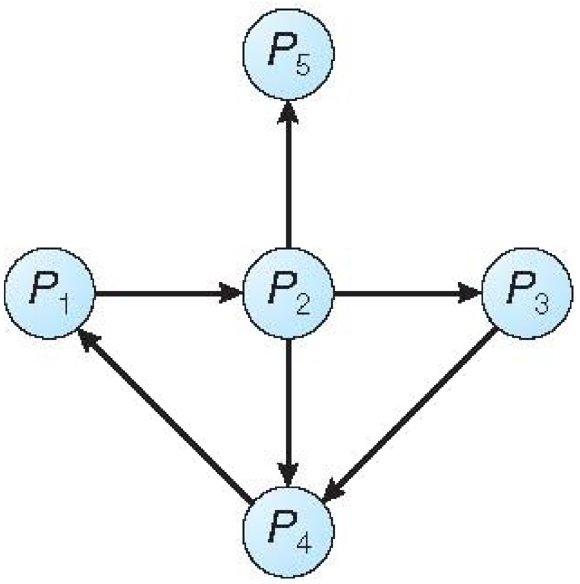
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an Algorithm that Searches for a Cycle in graph
  - If there is a cycle ➔ there exists a deadlock
- An Algorithm to Detect a Cycle in a Graph Requires an Order of n^2 operations
  - Where n is number of vertices in graph

Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for Graph

Detection-Algorithm Usage

- 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
- If Detection Algorithm is Invoked Arbitrarily, there may be many cycles in resource graph and so we would not be able to tell which of many deadlocked processes “caused” deadlock

Recovery from Deadlock: Process Termination

- Abort all Deadlocked Processes
  - Significant expense
  - Most processes need to be restarted for re-computation
- Abort one process at a time until Deadlock Cycle is Eliminated
  - After each process is aborted, a deadlock-detection algorithm must be invoked
    - To see if any process is still deadlocked

Recovery from Deadlock: Process Termination

- Issues with Aborting a Process
  - Updating a file
  - Printing data
- In which Order should we Choose to Abort?
  1. Priority of process
  2. How long process has computed, and how much longer to completion
  3. Resources process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

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

Reading Assignment

- Banker's Algorithm
- Deadlock Detection
  - Multiple instances of a resource type

Banker's Algorithm

- Multiple Instances
- Each Process must a Priori Claim Max Use
- 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 Banker's Algorithm

Let *n* = number of processes, and *m* = number of resources types.

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

  $Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

    Work = Available

    ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

    (a) ***Finish* [*i*] = *false***

    (b) $\boldsymbol{Need_i \le Work}$

    If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$

    Finish*[*i*] = *true

    go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$.

If $\boldsymbol{Request_i[j] = k} \rightarrow \boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition (as process has exceeded its maximum claim)
2. If $\boldsymbol{Request_i \leq Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5 instances), and *C* (7 instances)

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (cont.)

- Content of ***Need*** is defined to be ***Max – Allocation***

Need

	A	*B*	*C*
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- System is in a Safe State
  - Since sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (i.e., (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Several Instances of a Resource Type

- **Available**: A vector of length ***m*** indicates number of available resources of each type
- **Allocation**: An ***n* x *m*** matrix defines number of resources of each type currently allocated to each process
- **Request**: An ***n* x *m*** matrix indicates current request of each process
  - If ***Request* [*i*][*j*] = *k***, ➔ process P_i is requesting ***k*** more instances of resource type R_j

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:

   (a) ***Work = Available***

   (b) For ***i* = 1,2, …, *n***, if ***Allocation*** $_i \neq$ **0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:

   (a) ***Finish*[*i*] == *false***

   (b) ***Request*** $_i \leq$ ***Work***

   If no such ***i*** exists, go to step 4

Detection Algorithm (cont.)

3. ***Work* = *Work* + *Allocation*$_i$**
   Finish*[*i*] = *true
   go to step 2

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

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

Example of Detection Algorithm

- Five Processes P_0 through P_4; three Resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1		2 0 0	2 0 2
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example (cont.)

- P_2 requests an additional instance of type C

<u>*Request*</u>

	A	*B*	*C*
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

End of Lecture 9