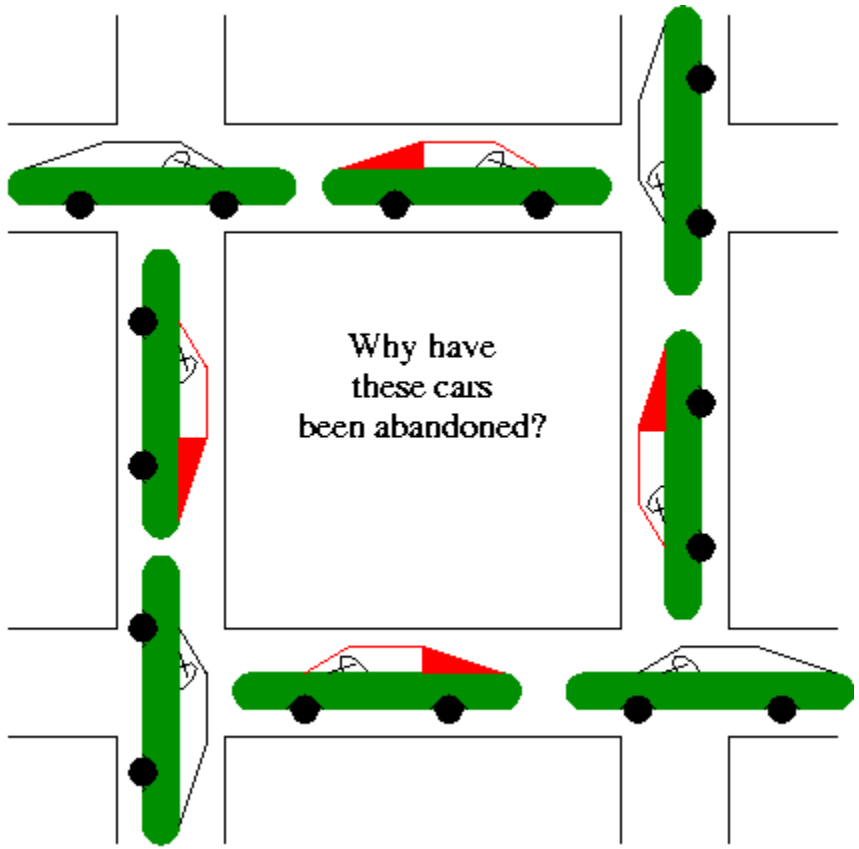
VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)


# SWE3001-Operating Systems

Prepared By
**Dr. L. Mary Shamala**
Assistant Professor
SCOPE/VIT

# Module 4: Deadlocks

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



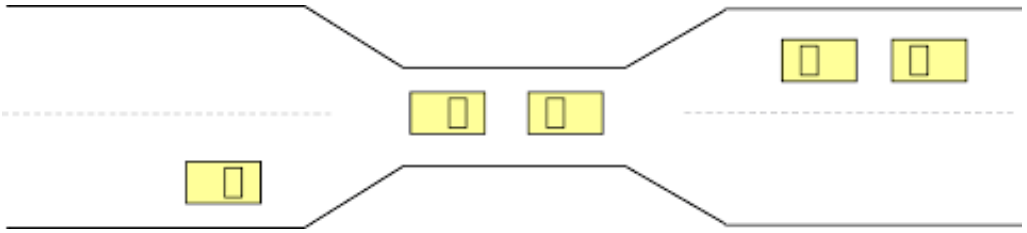
# System Model

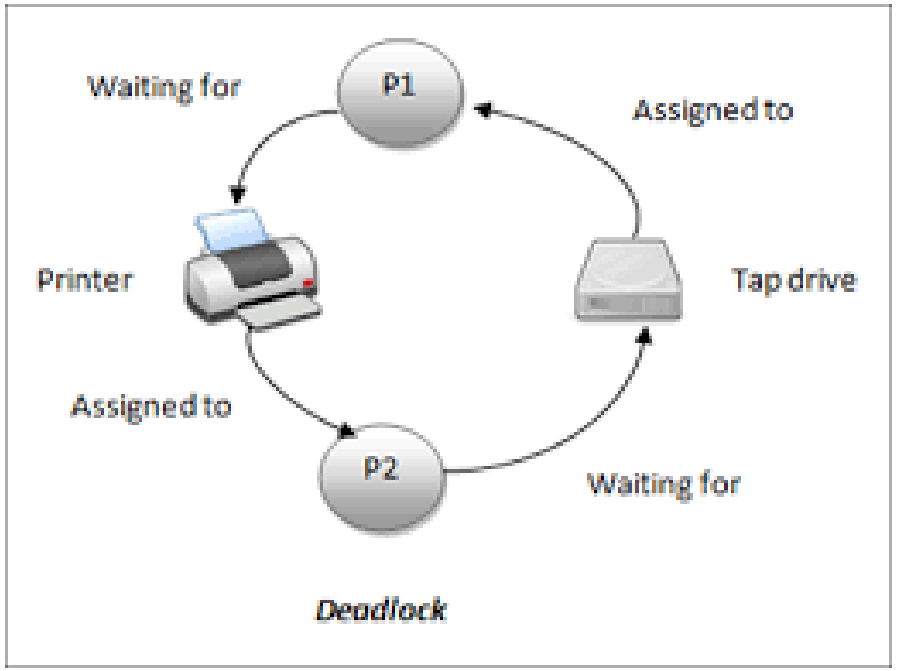
- System consists of finite number of resources
- Resource types $R_1$, $R_2$, . . ., $R_m$
  *CPU cycles, memory space, I/O devices*
- Each resource type $R_i$ has $W_i$ instances.
- A process must request a resource before using it and must release the resource after using it.
- A process may request as many resources as it requires to carry out its designated task.
- The number of resources requested may not exceed the total number of resources available in the system.

# System Model…

- Each process utilizes a resource as follows:
  - Request
  - Use
  - Release
- A set of processes is in a **deadlocked** state when every process in the set is waiting for an event that can be caused only by another process in the set.
- Events are resource acquisition and release.
- The resources may be either physical resources or logical resources
- However, other types of events may result in deadlocks
- Deadlocks may involve same or different resource types.

# Illustration of Deadlock



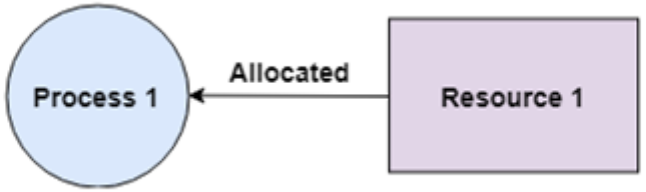


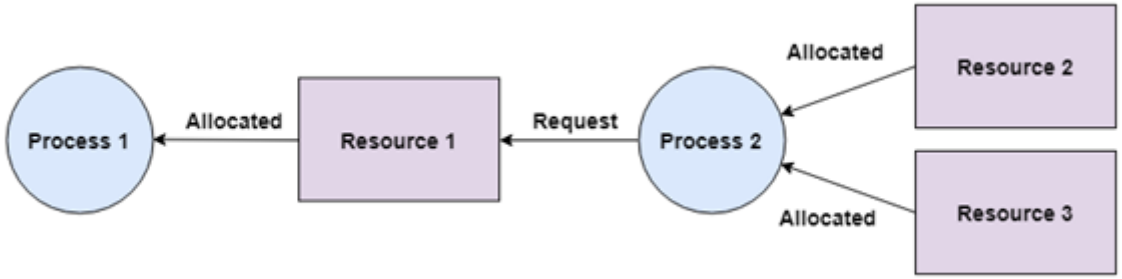
# Deadlock Characterization

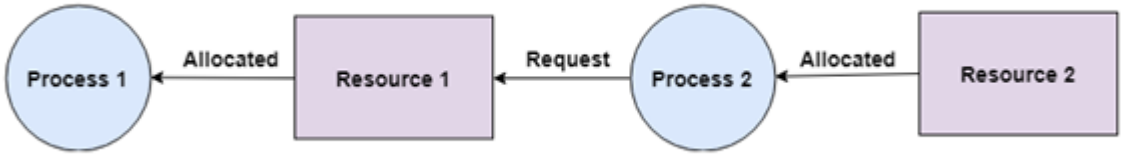
- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.
- **Necessary Conditions**

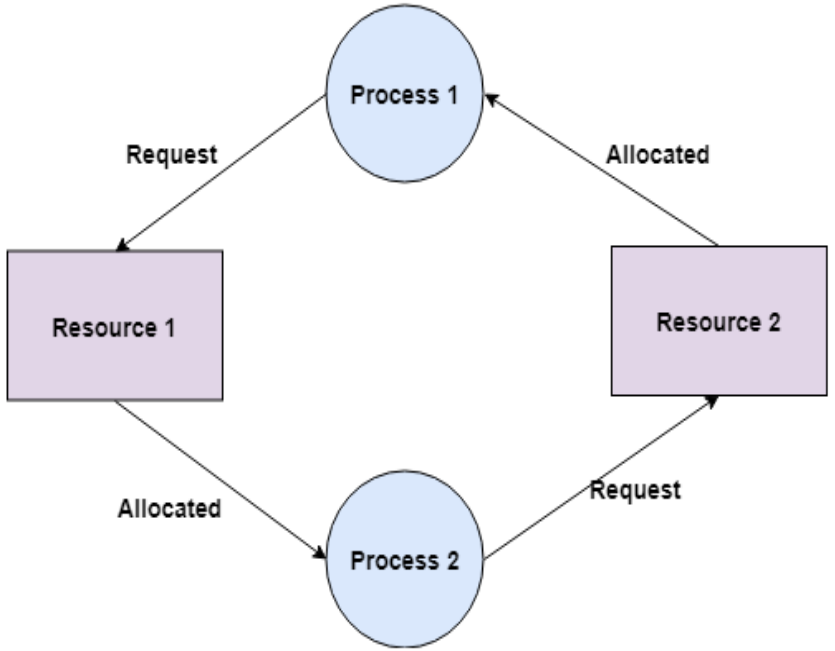
1. **Mutual exclusion**: only one process at a time can use a resource
2. **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. **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 $P_0$.

# Identify the necessary condition to deadlock!!!



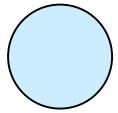



# Resource-Allocation Graph

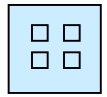
- Deadlocks can be described more precisely
- A system resource-allocation graph is a directed graph consisting of a set of vertices $V$ and a set of edges $E$.
- V is partitioned into two types of nodes:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **Request edge** – directed edge from process $P_i$ to resource type $R_j$ is denoted by $P_i \rightarrow R_j$
- **Assignment edge** – directed edge from resource type $R_j$ to process $P_i$ is denoted by $R_j \rightarrow P_i$

# Resource-Allocation Graph (Cont.)

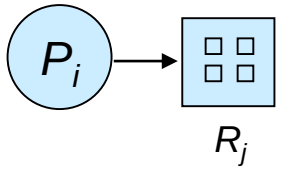
- Process



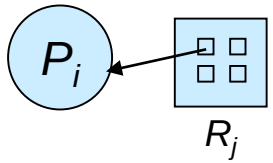
- Resource Type with 4 instances



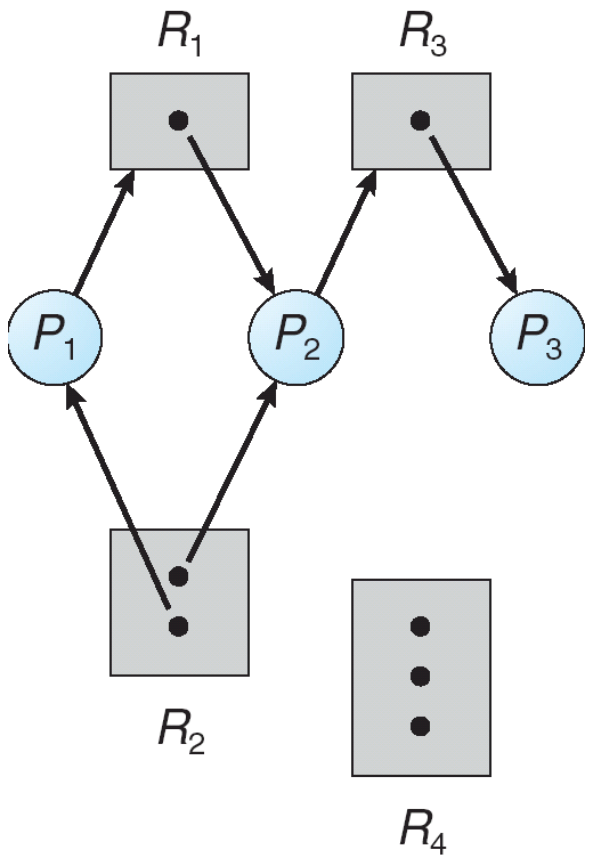
- $P_i$ requests instance of $R_j$




- $P_i$ is holding an instance of $R_j$



# Example of a Resource Allocation Graph

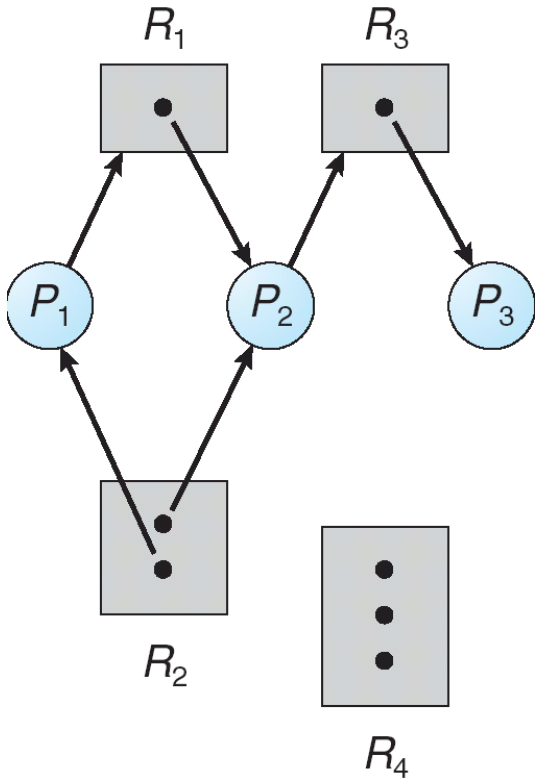


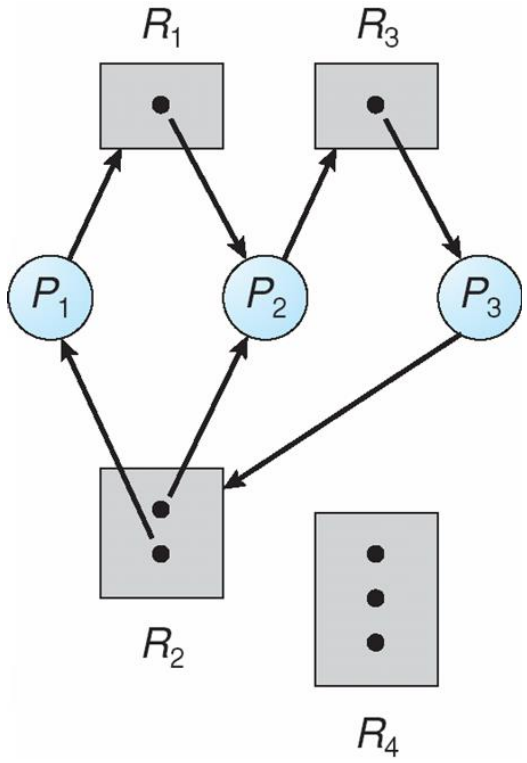

- The sets P, R, and E:
  - P={P1,P2,P3}
  - R={R1,R2,R3,R4}
  - E= $P_1 \rightarrow R_1$, $P_2 \rightarrow R_3$, $R_1 \rightarrow P_2$, $R_2 \rightarrow P_2$, $R_2 \rightarrow P_1$, $R_3 \rightarrow P_3$
- Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type R4
- Process states:
  - Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1 .
  - Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3 .
  - Process P3 is holding an instance of R3 .

If the graph contains no cycles, then no process in the system is deadlocked.

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

# Resource Allocation Graph With A Deadlock

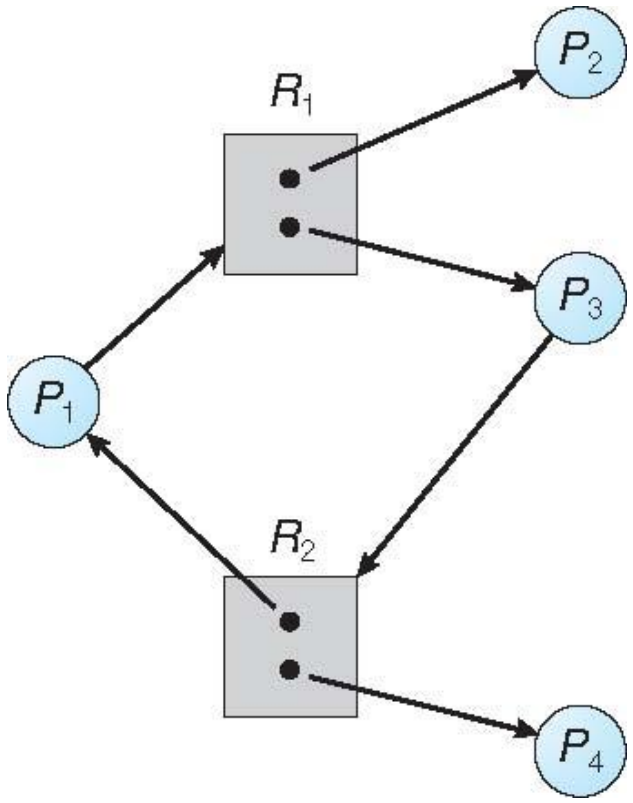



- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred
- a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred
- a cycle in the graph is a necessary not a sufficient condition for the existence of deadlock.

# Graph With A Cycle But No Deadlock




- There is a cycle:

  P1 → R1 → P3 → R2 → P1
- However, there is no deadlock.

**Basic Facts**

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

# Methods for Handling Deadlocks

- Three ways to deal with the deadlock problem

1. Ensure that the system will ***never*** enter a deadlock state:
   - Deadlock prevention
   - Deadlock avoidance
2. Allow the system to enter a deadlock state, detect it and then recover
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

# Deadlock Prevention

- Prevent deadlocks by limiting how requests can be made.
- By ensuring that at least one of the necessary conditions cannot hold, we can prevent the occurrence of a deadlock.
- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - 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 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 each process requests resources in an increasing order of enumeration

# 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

```
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, and Transaction 2 transfers $50 from account B to account A

# Deadlock Avoidance

- Requires that the system has some additional **a priori** information available
- The various algorithms that use this approach differ in the amount and type of information required.
- The 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
- The resource-allocation **state** is defined by the number of available and allocated resources and the maximum demands of the processes

# Safe State

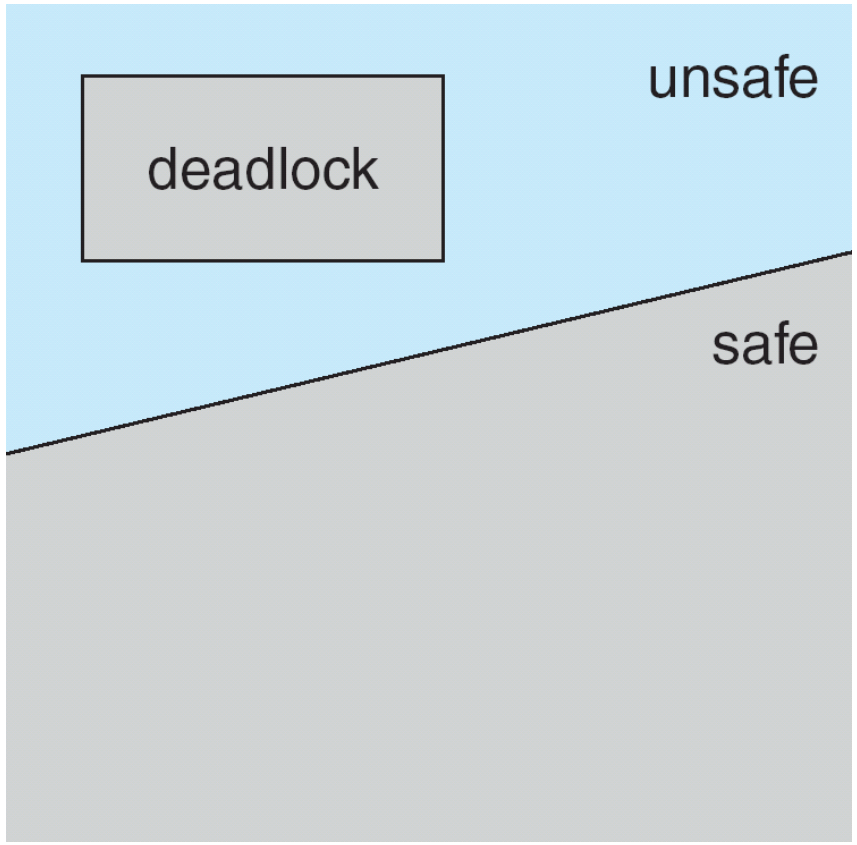
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state
- A state is **safe** if the system can allocate resources to each process in some order and still avoid a deadlock.
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ 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
  - If no such sequence exists, then the system state is said to be unsafe.

# Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Not all unsafe states are deadlocks, however (Figure 7.6).
- An unsafe state may lead to a deadlock

- Avoidance $\Rightarrow$ ensures that a system will never enter an unsafe state.

# Safe, Unsafe, Deadlock State



# Safe, Unsafe, Deadlock State …

- Consider a system with twelve magnetic tape drives and three processes: P0 , P1 , and P2 .

| Process | Maximum Needs | Current Needs |
|---|---|---|
| P0 | 10 | 5 |
| P1 | 4 | 2 |
| P2 | 9 | 2 |

- At time $t_0$ , the system is in a safe state.
- A system can go from a safe state to an unsafe state.
- Suppose that, at time $t_1$ , process P2 requests and is allocated one more tape drive.
- The system is no longer in a safe state.

# Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state.
- Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.

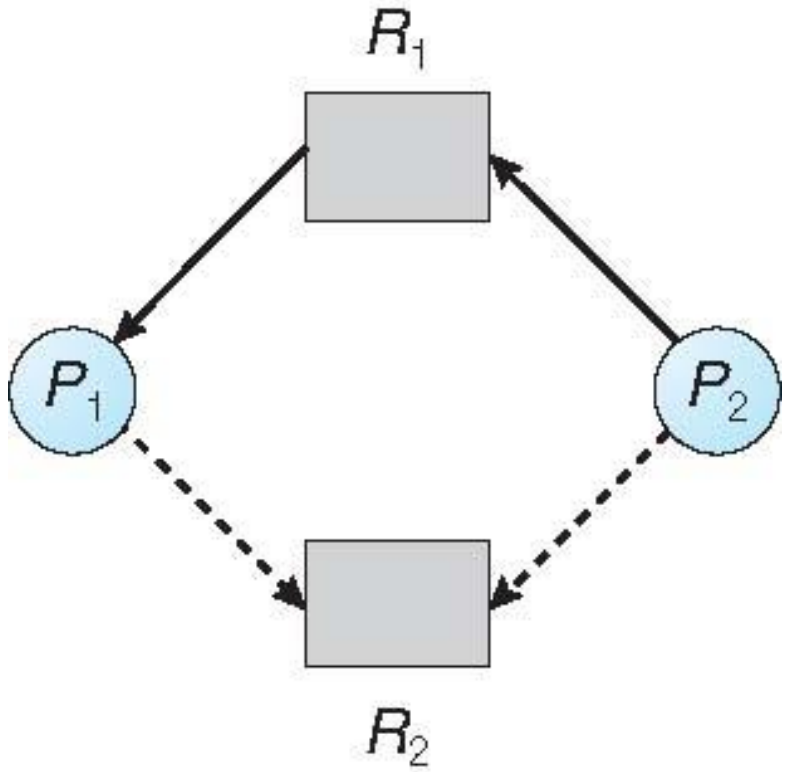
# Resource-Allocation Graph Algorithm

- In addition to the request and assignment edges a new type of edge, called a claim edge is used
- **Claim edge** $P_i \rightarrow R_j$ indicated that process $P_j$ may request resource $R_j$ at some time future; 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

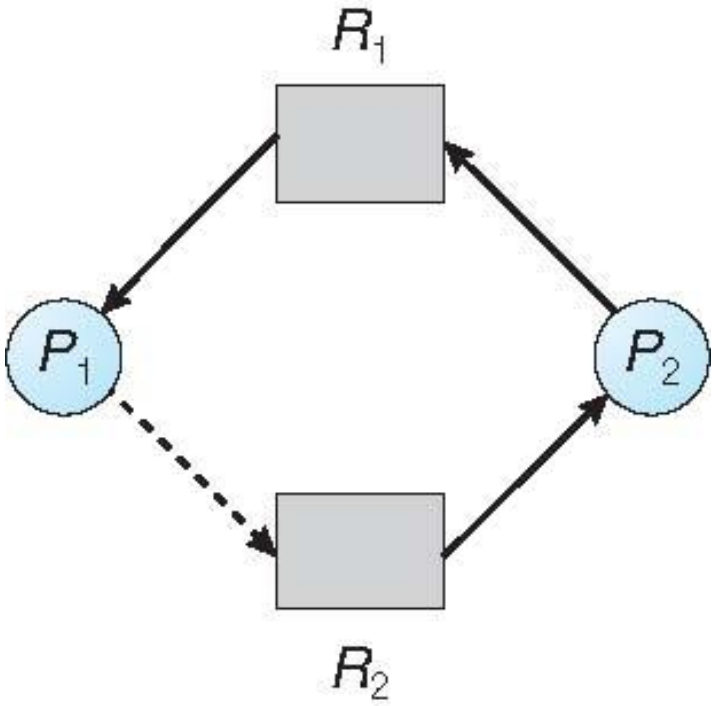
# Resource-Allocation Graph Algorithm…

- 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
- Check for safety by using a cycle-detection algorithm→ $n^2$ operations
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state.
- In that case, process Pi will have to wait for its requests to be satisfied.

# Resource-Allocation Graph for Deadlock Avoidance



# Unsafe State In Resource-Allocation Graph



# Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum 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 the 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**: $n$ x $m$ 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) $Need_i \leq Work$

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

3. $Work = Work + Allocation_i$

   ***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 $P_i$

$Request_i$ = request vector for process $P_i$. If $Request_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise $P_i$ must wait, since resources are not available
3. Pretend to allocate requested resources to $P_i$ 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 $P_i$
- If unsafe $\Rightarrow$ $P_i$ must wait, and the 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* (5instances), 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.)

- The content of the matrix ***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 |

- The system is in a safe state since the sequence < $P_1$, $P_3$, $P_4$, $P_2$, $P_0$> satisfies safety criteria

# Example: $P_1$ Request (1,0,2)

- Check that Request $\leq$ Available (that is, (1,0,2) $\leq$ (3,3,2) $\Rightarrow$ 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?

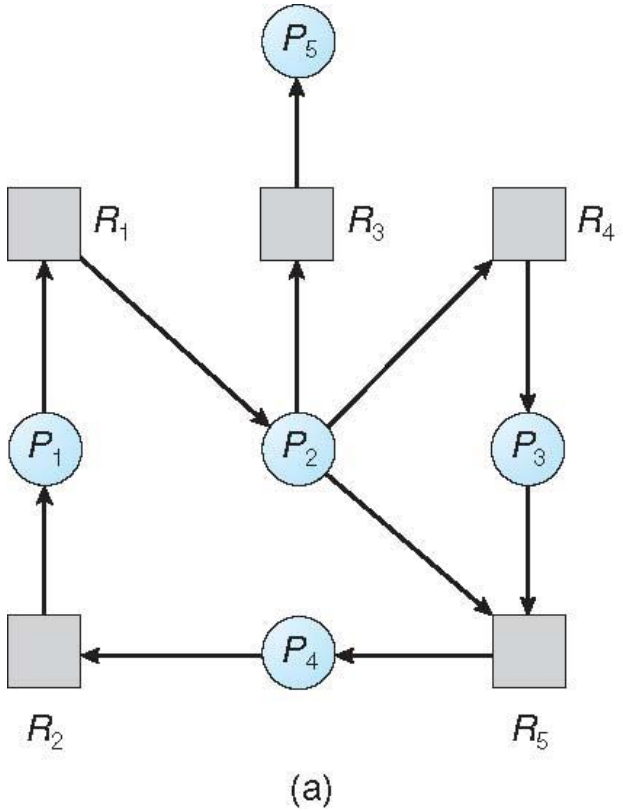
# Deadlock Detection

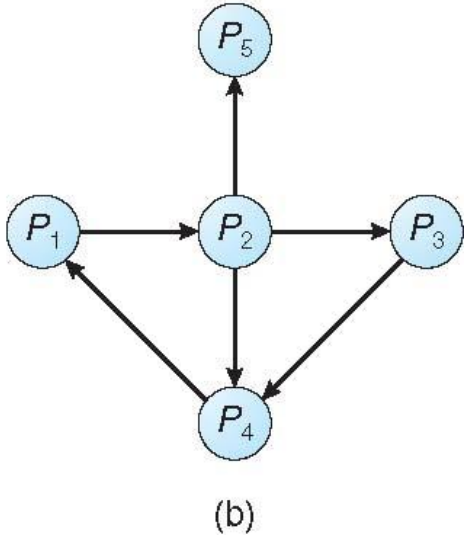
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

# 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 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

# Resource-Allocation Graph and Wait-for Graph




Resource-Allocation Graph

Corresponding wait-for graph

# Several Instances of a Resource Type

- **Available**: A vector of length ***m*** indicates the number of available resources of each type
- **Allocation**: An ***n*** **x** ***m*** matrix defines the number of resources of each type currently allocated to each process
- **Request**: An ***n*** **x** ***m*** matrix indicates the current request of each process. If ***Request*** **[*i*][*j*] = *k***, then 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 $\boldsymbol{Allocation_i \neq 0}$, then
   ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) $\boldsymbol{Request_i \leq Work}$

   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 $C$

| | *Request* |
|---|---|
| | *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 $P_4$

# 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 the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

# Recovery from Deadlock

- Inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  - Process Termination-simply to abort one or more processes to break the circular wait
  - Resource Preemption- preempt some resources from one or more of the deadlocked processes

# Recovery from Deadlock:  Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In both methods, the system reclaims all resources allocated to the terminated processes.
- Abort those processes whose termination will incur the minimum cost.
- Many factors may affect which process is chosen:
  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?

# Recovery from Deadlock: Resource Preemption

- Successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- If preemption is required to deal with deadlocks, then three issues need to be addressed:
- **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

# Practice Problems

1. Consider the following snapshot of a system:

| | Allocation | Max |
|---|---|---|
| | A B C D | A B C D |
| P0 | 3 0 1 4 | 5 1 1 7 |
| P1 | 2 2 1 0 | 3 2 1 1 |
| P2 | 3 1 2 1 | 3 3 2 1 |
| P3 | 0 5 1 0 | 4 6 1 2 |
| P4 | 4 2 1 2 | 6 3 2 5 |

# Practice Problems…

- Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the processes may complete. Otherwise, illustrate why the state is unsafe.

  a. Available = (0, 3, 0, 1)

  b. Available = (1, 0, 0, 2)

# Practice Problems…

2. Consider a system consisting of m resources of the same type being shared by n processes. A process can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

   a. The maximum need of each process is between one resource and m resources.

   b. The sum of all maximum needs is less than m + n.

# Practice Problems…

3. Consider the following snapshot of a system:

| | Allocation | Max | Available |
|---|---|---|---|
| | A B C D | A B C D | A B C D |
| P0 | 0 0 1 2 | 0 0 1 2 | 1 5 2 0 |
| P1 | 1 0 0 0 | 1 7 5 0 | |
| P2 | 1 3 5 4 | 2 3 5 6 | |
| P3 | 0 6 3 2 | 0 6 5 2 | |
| P4 | 0 0 1 4 | 0 6 5 6 | |

- Answer the following questions using the banker's algorithm:
  a. What is the content of the matrix Need?
  b. Is the system in a safe state?
  c. If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

# Summary

- A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- A deadlock can occur only if four necessary conditions hold simultaneously in the system: mutual exclusion, hold and wait, no preemption, and circular wait.
- To prevent deadlocks, we can ensure that at least one of the necessary conditions never holds.
- Avoiding deadlocks requires that the operating system have a priori information about how each process will utilize system resources
- The banker's algorithm, for example, requires a priori information about the maximum number of each resource class that each process may request.
- A deadlock-detection algorithm must be invoked to determine whether a deadlock has occurred.