Operating System
CS 2006
Lecture 9

Ms. Mahzaib Younas

Lecturer Department of Computer Science

FAST NUCES CFD

Outlines

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

System Model

- System consists of resources
- Resource types R1, R2, . . ., Rm
  - CPU cycles, memory space, I/O devices
- Each resource type Ri has Wi instances.

System Model

- Each process utilizes a resource as follows:

1. **Request**

   The thread requests the resource. If the request cannot be granted immediately
   Example: if a mutex lock is currently held by another thread), then the requesting thread must wait until it can acquire the resource.

2. **Use**

   The thread can operate on the resource

   Example: if the resource is a mutex lock, the thread can access its critical section

3. **Release**

   The thread releases the resource.

Deadlock with Semaphore

- Data:
  - A semaphore **S_1** initialized to 1
  - A semaphore **S_2** initialized to 1
- Two threads T_1 and T_2
  - ***T_1*:**

    wait(s_1)

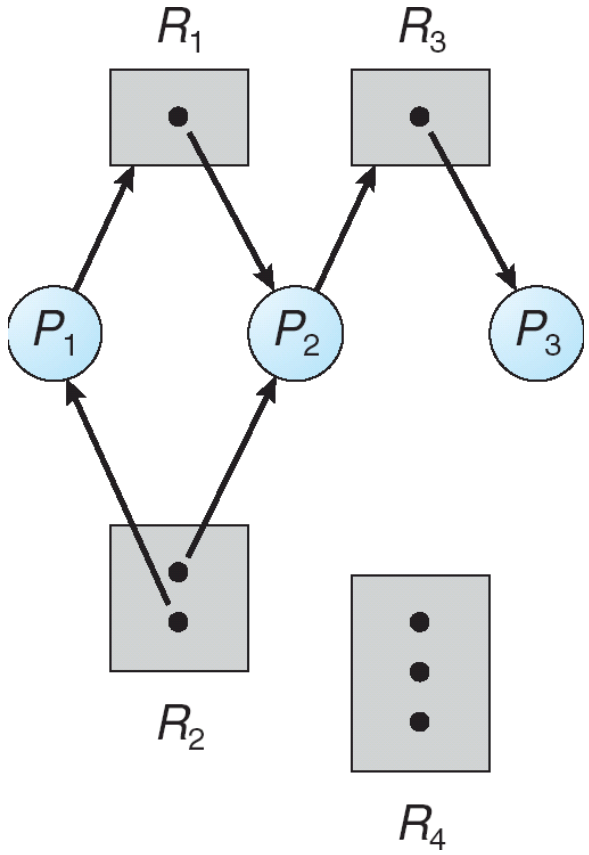
    wait(s_2)
  - ***T_2*:**

    wait(s_2)

    wait(s_1)

Example of Resource Allocation Graph

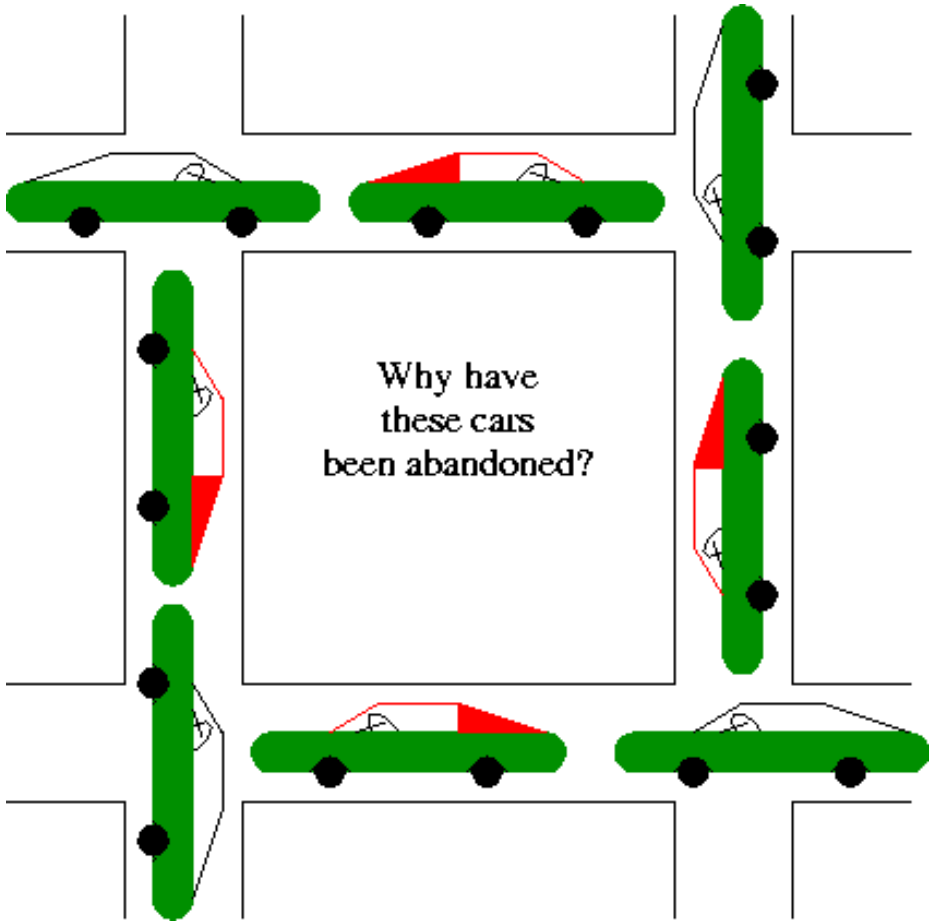
- P1 is holding an instance of R2
- P2 is holding an instance of R1 and R2 respectively.
- P3 is holding an instance of R3



- P1 requests for instance of R1

Deadlock

- A **deadlock** occurs when a every member of a set of processes is waiting for an event that can only be caused by a member of the set.
- Consider the below figure:
  - The processes are the cars.
  - The resources are the spaces occupied by the cars



Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set of processes
- Deadlock prevents sets of concurrent processes from completing their tasks

- Example 1:
  - Example 1 System has 2 disk drives
  - Process P1 and process P2 each hold one disk drive and each needs another one to complete their respective tasks.

Deadlock Characterization: 4 Conditions

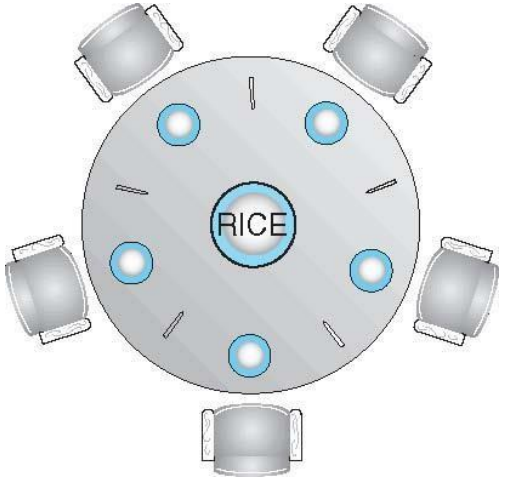
- Deadlock can arise if four conditions hold simultaneously.
  1. Mutual exclusion
  2. Hold and wait
  3. No preemption
  4. Circular wait
- All four conditions must hold for a deadlock to occur

Mutual Exclusion

- Mutual exclusion

  only one process at a time can use a resource ; that is, at least one resource must be held in non-sharable mode.

- Requesting process must wait until resource is released
- Example
  - The Dining philosopher problem, a chopstick is a resource which can be used only by a single philosopher.

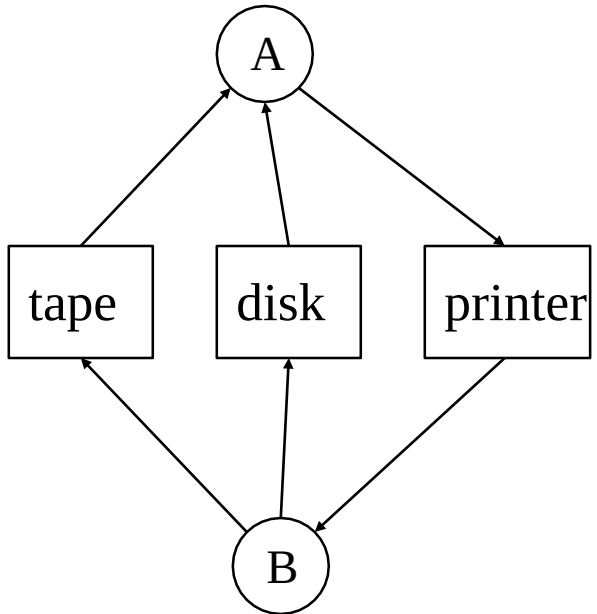


Hold and wait

- Hold and wait

  A process holding at least one resource is waiting to acquire additional resources held by other processes.

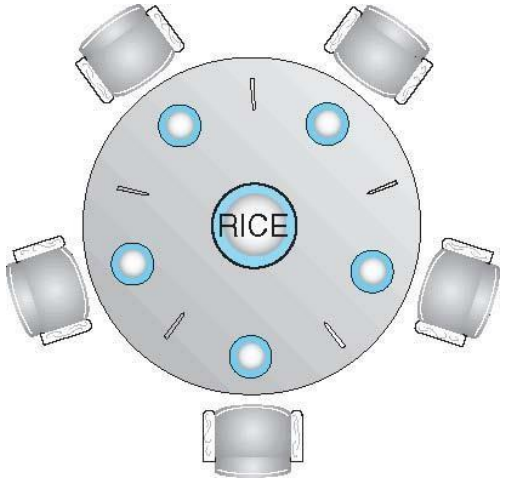
- Example
  - Process A copies data from a tape drive to a file on disk, sorts the file and then prints. However, it has acquired only tape drive and disk drive and is waiting for printer which is held by Process B



No Premmption

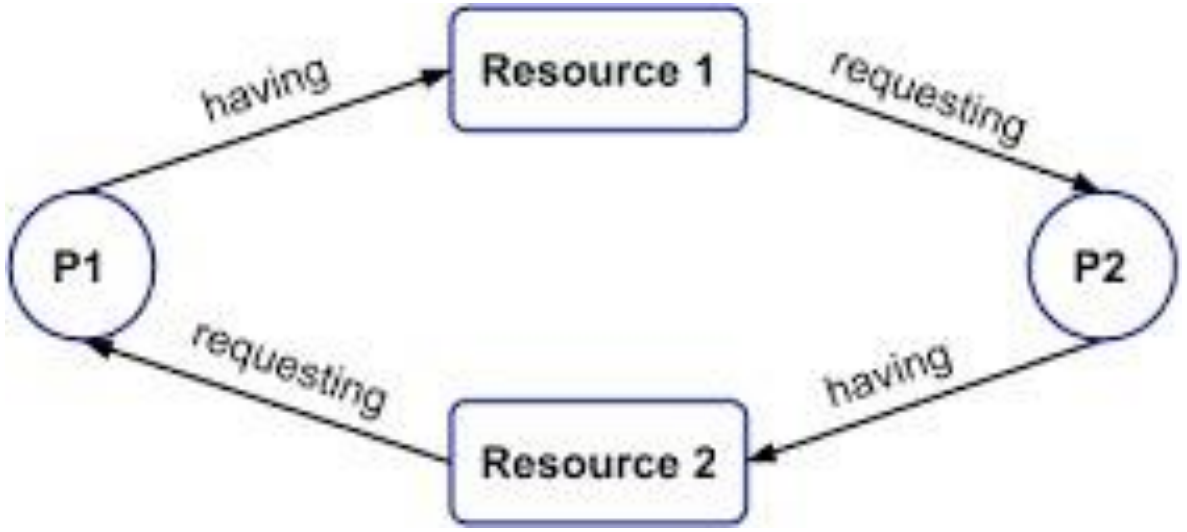
- **No preemption:**

  A resource can be released only voluntarily by the process holding it; that is, resources cannot be preempted. In other words, no resource can be forcibly removed from a process holding it



Circular Wait

- Circular wait:
  - there exists a set of waiting processes {P0, P1, …, Pn} such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.



Resource Allocation Graph

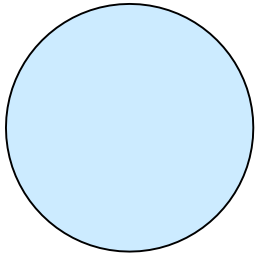
- Deadlock Description:

Resource-Allocation Graph represents the state of a resource allocation system at a given moment in time.

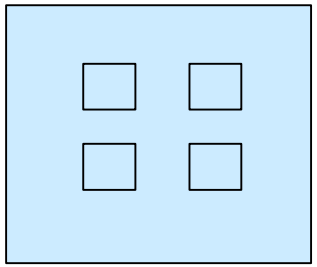
- It is a directed graph consisting of vertices (Points) V and a set of edges (arcs) E.
- V is partitioned into two types:
  - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system
  - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system

Resource Allocation Graph

- Process ; circles



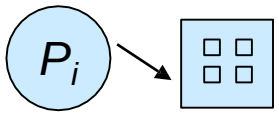
- Resource Type with 4 instances ; rectangles with a dot for each instance



Resource Allocation Graph

- **request edge** – directed edge $P_i \rightarrow R_j$
  - Process P_i has requested an instance of resource type R_j
  - And P_i is currently waiting for that resource

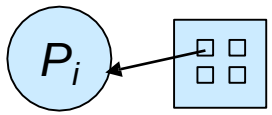
- Pi requests an of instance of Rj



Resource Allocation Graph

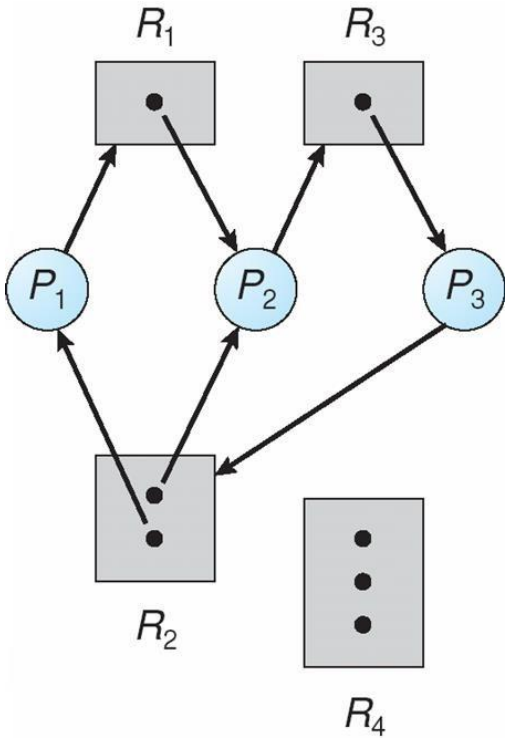
- Assignment edge – directed edge Rj → Pi
- Instance of Rj has been allocated to process Pi

- Pi is holding an instance of Rj



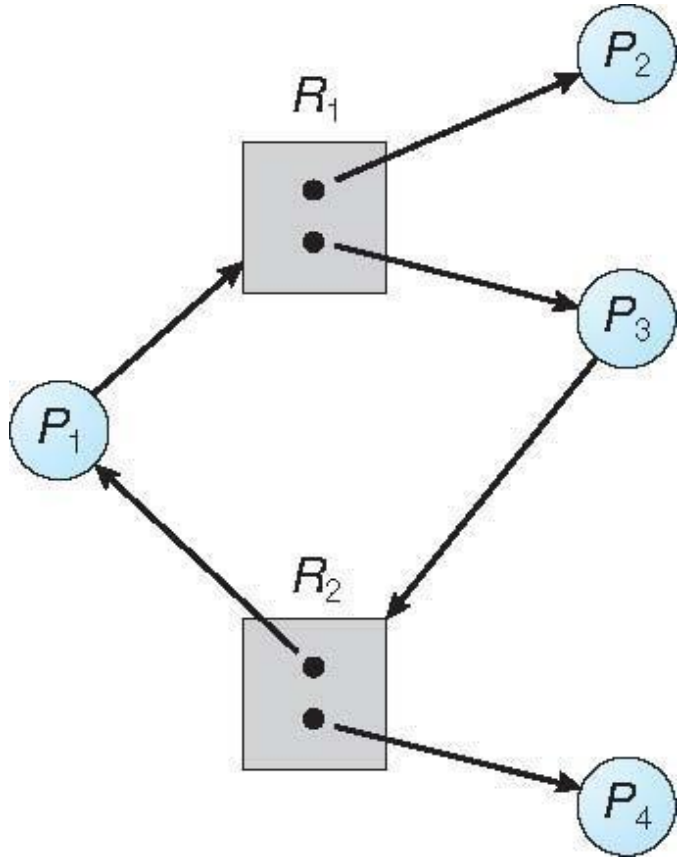
Resource Allocation Graph with deadlock

- Deadlock: P2 is waiting for resource R3 held by P3,
  - P3 is waiting for resource R2 held by P2
  - P1 is waiting for resource R1 held by P2



Graph Cycle with no Deadlock

- No deadlock: since P4 may release its instance of resource type R2
  - which can be allocated to P3; thus breaking the cycle



Basic Facts

- If graph contains no cycles then no deadlock
- If graph contains a cycle then deadlock may exist

- if only one instance per resource type, then deadlock
  - Cycle is necessary and sufficient condition for deadlock
- if several instances per resource type, possibility of deadlock
  - Cycle is necessary but not sufficient condition for deadlock

Methods for Handling Deadlock

1. Devise a protocol ensuring that the system will never enter a deadlock state
2. Allow the system to enter a deadlock state
3. Ignore the problem and pretend that deadlocks never occur in the system;

Method 1:

- Deadlock prevention scheme
  - Ensure that at least one of the four necessary conditions cannot hold
  - Prevent deadlocks by constraining how requests can be made
- Deadlock avoidance
  - OS uses additional knowledge:
  - Which resources a process will request in its lifetime
  - Currently available resources, and, currently allocated resources
  - Future release of each process
- Then can decide If a process should wait, or, if a request can be satisfied

Method 2

System allows algorithm that examines the state of the system to determine if deadlock has occurred and an algorithm to recover from the deadlock

Deadlock Prevention

- Deadlock Prevention means making sure deadlocks never occur.

- By ensuring that at least one of four conditions cannot hold, we can prevent the occurrence of a deadlock.

1. Eliminate Mutual Exclusion
2. Eliminate Hold and wait
3. Eliminate no preemption
4. Eliminate circular wait

Eliminate Mutual Exclusion

- Make resources sharable (e.g., read-only files); But not all resources can be shared.
  - Sharable resources cannot be involved in a deadlock; wait never needed
  - But some resources are inherently non-sharable, e.g., mutex locks cannot be simultaneously shared by several process

Eliminate Hold and Wait

Must guarantee that whenever a process requests a resource, it does not hold any other resources.

- Removing Hold-and-Wait implies every process must acquire all resources it needs all at once.

Protocol 1:

- Require process to request and be allocated all its resources before it begins execution
- Example:
- Process copies data from a
  - DVD drive to a file on disk
  - sorts the file
  - Prints the file.
- According to protocol 1, process must initially request the DVD drive, disk file, and printer.
- It will hold the printer for its entire execution even though it needs printer only at the end

Protocol 2:

- Allow process to request resources only when the process has none allocated to it.

A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Example:

- this method requests only DVD and disk file initially, copies from DVD to disk and then release both. The process must then request the disk file and printer.
- must release both after copying file from disk to printer and terminate

Eliminate No Preemption

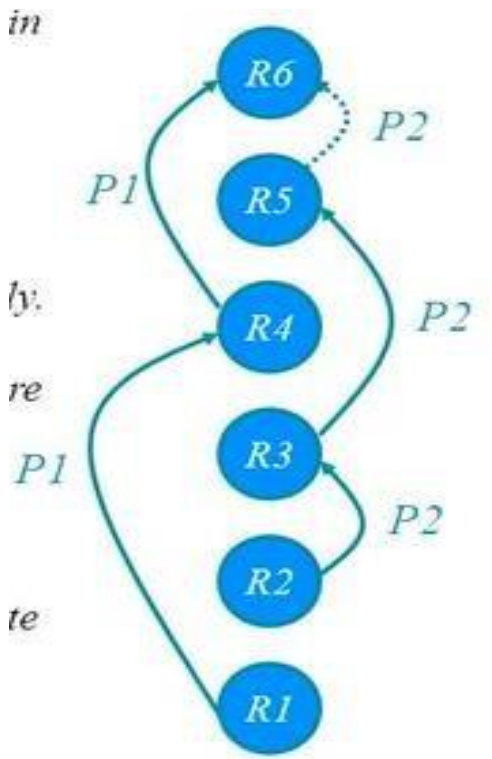
- means being able to forcibly take resources away from a process.
- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released ***(Preempted)*** and are allocated to another requesting process.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Eliminate Circular Wait

- Eliminate Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
- Request resources according to the decided order or request
- Define a linear ordering of resource types
- Let R ={R1,R2,R3,..Rm} be the set of resource types. Assign each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our example:
- For example
  - Tape drive = 1
  - Disk drive = 4
  - Printer = 6

Eliminate Circular Wait

- If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.
- Processes can request resources whenever they want to, but all requests must be made in numerical order.



Deadlock Example

```
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 Avoidance

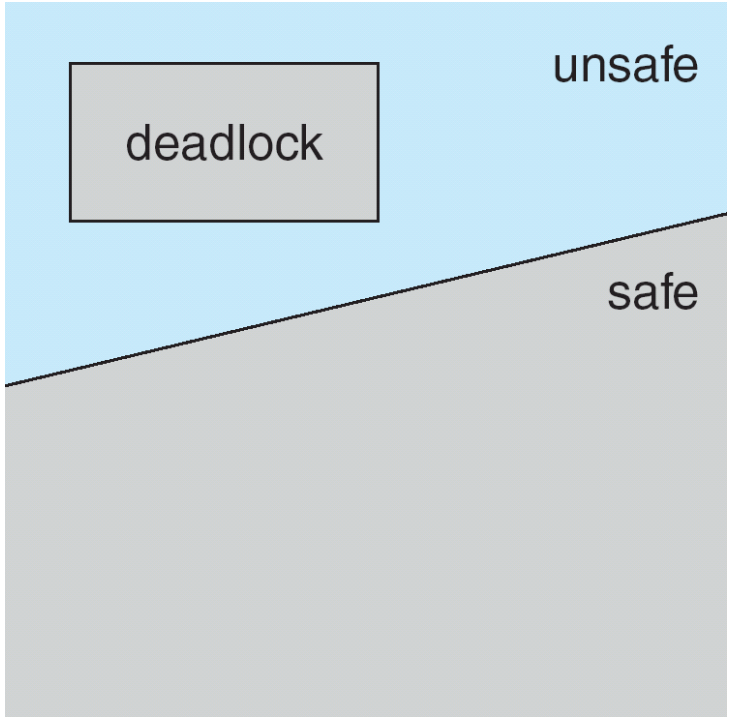
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state 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 <P1, P2, …, Pn> of ALL the processes in the systems such that for each process Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj
- That is:
  - If process Pi resource needs are not immediately available, then Pi can wait until all Pj have finished
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate
  - When Pi terminates, Pi +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
- According to Detail:
  - "An unsafe state does not imply the existence of deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to deadlock.“
- Avoidance
  - ensure that a system will never enter an unsafe state.



First Example: Safe State

- System with 24 tape drives and three processes
- P0 may require 20 tape-drives during execution, P1 may require 8, and P2 may require up to 18.
- Assume, P0 is holding 10 tape drives, P1 holding 5 and P2 holding 4 tape drives.
- The remaining available resources are: 24-19=5.
- P1 needs 3 more tape drives to finish, so it can be allocated 3 drives.
- Once P1 is done, it will return all 8 resources, making the total available resources = 10.
- P0 may proceed further obtaining the available 10 tape drives, once it completes it will return all 20 tape drives available = 20.
- P2 may proceed finally, and obtain 14 tape drives out of 20 available.
- The system is said to be in safe state, since there is a safe sequence that avoids the deadlock. <P1, P0, P2>

Process	Max. Need	Allocated	Need
P_0	20	10	10
P_1	8	5	3
P_2	18	4	14

Example 2: Safe State

- Again, consider system with 12 tape drives and three processes
- The process, however, have different maximum needs and allocated resources.
- Following the same procedure as in the previous slide.
  - The available resources are: 3
  - It will be safe to first allocate resources to P1
  - After P1 completes, 5 resources are allocated to P0 and then finally P2 may obtain the available resources.
  - So safe sequence is: < P1, P0, P2>

Process	Max. Need	Allocated
P_0	10	5
P_1	4	2
P_2	9	2

Unsafe State

- Consider the system with ***12 tape drives*** and three processes from the previous example:
- **What if P_2 requests and is allotted one tape drive?**
- The available resources with system = 2 tape drive.
- The system is no longer in safe state. Only P_1, can be allocated all its tape drives, when it is done it will return them and ***the available*** tapes will be four.
- Both P_0 ($need_0 = 5$) and P_2 ($need_2 = 5$), need more than the available four, if they request , they will have to wait which may result in deadlock.
- It was a ***mistake*** to grant one or more tape drives to P_2
- It would have been safe if we had made P_2 wait.

Process	Max. Need	Allocated	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

Deadlock Avoidance

- Single instance of a resource type
  - Use a resource-allocation graph

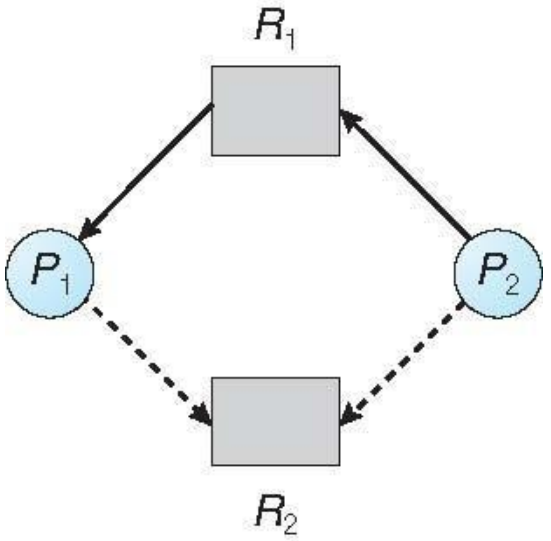
- Multiple instances of a resource type
  - Use the banker’s algorithm

Resource Allocation Graph Scheme

- **Claim edge** P_i - - - > 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, $P_i \rightarrow R_j$
- **Request edge** converted to an **assignment edge** when the resource is allocated to the process $R_j \rightarrow P_i$
- When a resource is released by a process, **assignment edge** reconverts to a **claim edge** P_i - - - > R_j
- Resources must be claimed *a **priori*** in the system

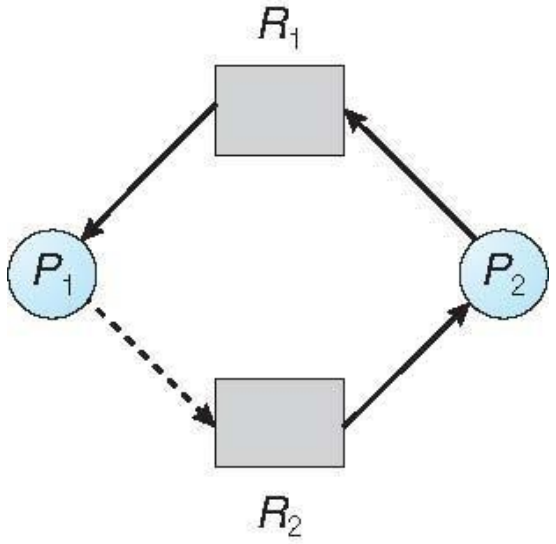
Resource Allocation Graph with Claim Edges

- R1 is held by P1.
- P2 requests R1.
- P1 and P2 may need R2 and therefore have claimed R2. (Made a priori information available to the system)



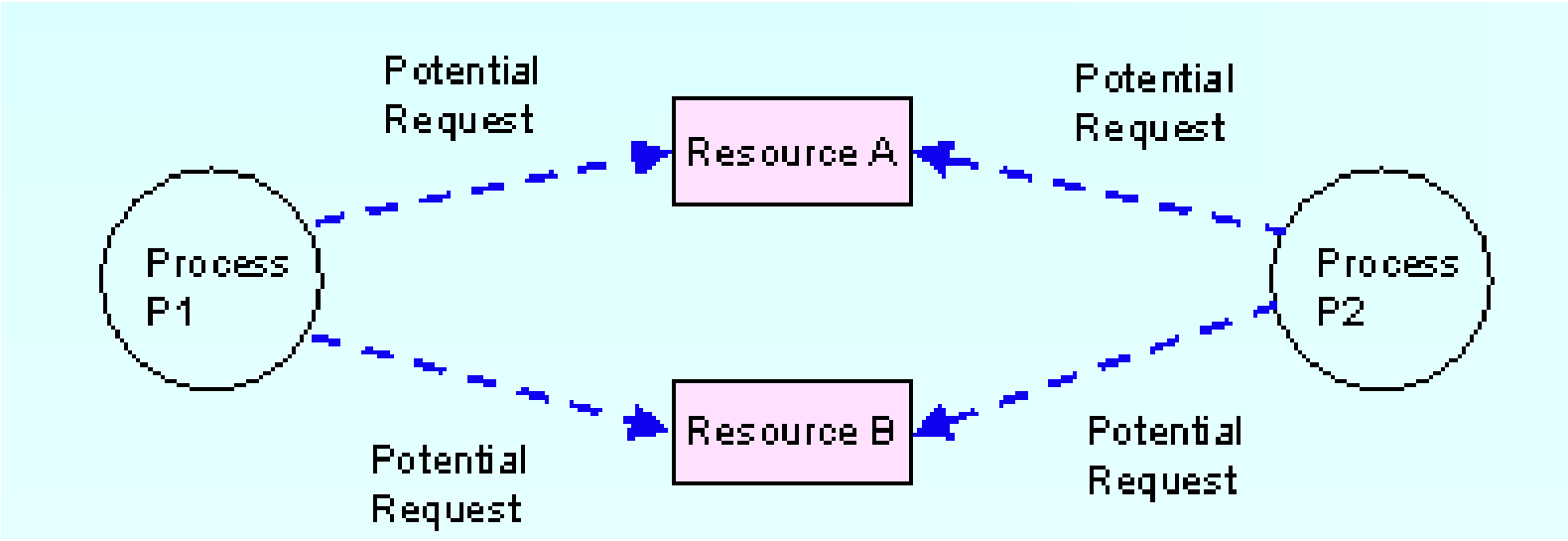
Unsafe State Resource Allocation Graph

- Now suppose that process P2 requests a resource R2
- 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

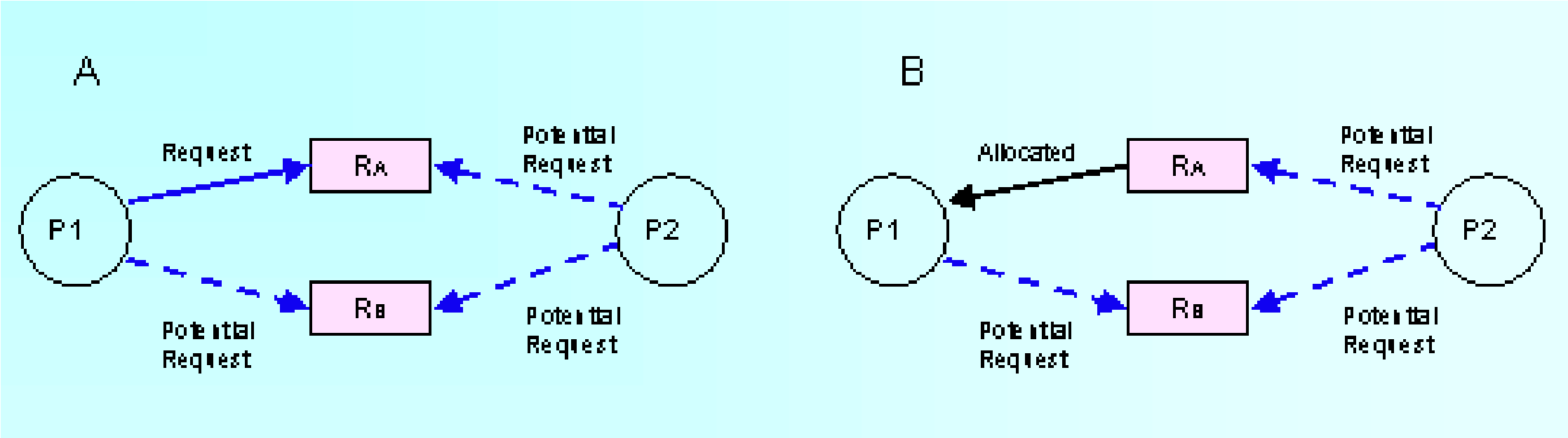


P_2 must wait! A cycle found.

Unsafe State of Resource Allocation Graph



Unsafe State of Resource Allocation Graph



Banker Algorithm

- Banker's algorithm is used to avoid deadlock in a system where the resources have multiple instances.
- Kev idea: Ensure that the system of processes and resources is always in a safe state
- Multiple Instances
- Each process must in advanced 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

Baker’s Algorithm

- The Banker's algorithm allows:
  - mutual exclusion
  - wait and hold
  - no preemption
- Prevents:
  - Circular wait

- User process may only request one resource at a time.
- System grants request only if the request will result in a safe state.

Banker’s Algorithm

- Banker’s algorithm comprises of Data Structures and two algorithms:
  - Safety algorithm
  - Resource request algorithm

Bankers' Algorithm

- **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*]**

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

Safety Algorithm

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

Work = Available *//The number of resources available with the system*

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

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

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

(b) $Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

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

3. $Work = Work + Allocation_i$ *//assume Pi is finished then return resources back to the system*

Finish*[*i*] = *true *//process is assumed to be finished.*

go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state. otherwise, the processes whose index is false may potentially be involved in a deadlock in the future.

Resource-Request Algorithm for Process Pi

Requesti = request vector for process Pi.

If Requesti [j] = k then process Pi wants k instances of resource type Rj

1. ***If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim***
2. ***If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available***
3. ***Pretend to (tentatively) allocate requested resources to Pi by modifying the state as follows:***

Available = Available – Requesti;

Allocationi = Allocationi + Requesti;

Needi = Needi – Requesti;

If safe ⇒ the resources are allocated to Pi

If unsafe ⇒ Pi must wait. and the old resource-allocation state is restored. The requested resources are not allocated to Pi.

Banker Algorithm

- 5 processes P_0 through P_4;

  3 resource types: A (10 instances), B (5 instances), and C (7 instances) :[10 5 7]

- Snapshot at time T_0:

Process	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	

- Allocated = [7 2 5]
- **Find out the Need vector for each process for each of the resource?**

- The content of the matrix ***Need[i,j]*** is defined to be ***Max[i,j] – Allocation[i,j]***

Process	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	

Process	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

- Find out if the System is in safe state using the ***Safety Algorithm***.
  - The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria. [See the next slide for the step by step details of the algorithm]

Step 1: Work = Available → Work = [3 3 2]

Step 2:

$Need_i \leq Work$//resources needed by the process i are less than the available resources

$Work = Work + Allocation_i$

Let's start from P_0, $Need_0$ <= Work? No

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

Safe Sequence: < >

Work = [3 3 2]

How about P_1?

$Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

Safe Sequence: <P_1, >

Work = [5 3 2]

For P_2? No

For P_3?

$Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

Safe Sequence: <P_1, P_3, >

$Work = [\ 7\ 4\ 3]$

$For\ P_4?$

$Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2			

Safe Sequence: <P_1, P_3,P_4, >

$Work = [\ 7\ 4\ 5]$

$Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2	7	5	5

Safe Sequence: < P_1, P_3,P_4,P_0, >

$Work = [\ 7\ 5\ 5]$

$Need_i \leq Work$ *//resources needed by the process i are less than the available resources*

$Work = Work + Allocation_i$

Process	Need			Allocation			Work		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2	7	5	5

Safe Sequence: < P_1, P_3,P_4,P_0, P_2 >

Class Exercise

- Five processes P0 through P4; three resource types
  A (7 instances), B (2 instances), and C (6 instances)

Process	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	0	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

What is the safe sequence, such that all the processes execute without deadlock?

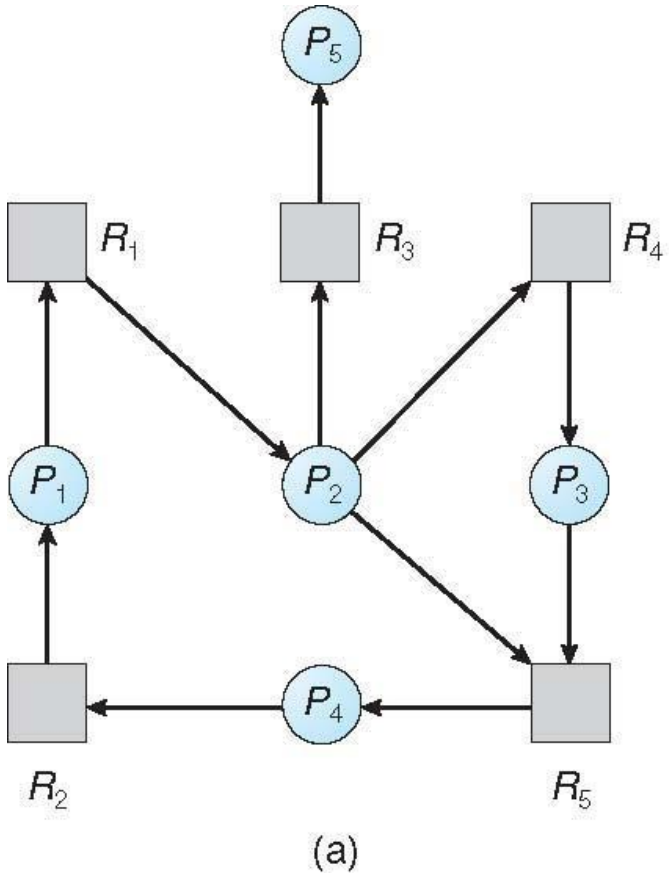
Deadlock Detection

- Most systems do not do anything about deadlock. They just allow the system to enter a deadlock state if it happens.
- Such systems can use a deadlock detection algorithm to discover when deadlock has occurred.
- If deadlock is detected, the system can then use a deadlock recovery algorithm to undo the deadlock.

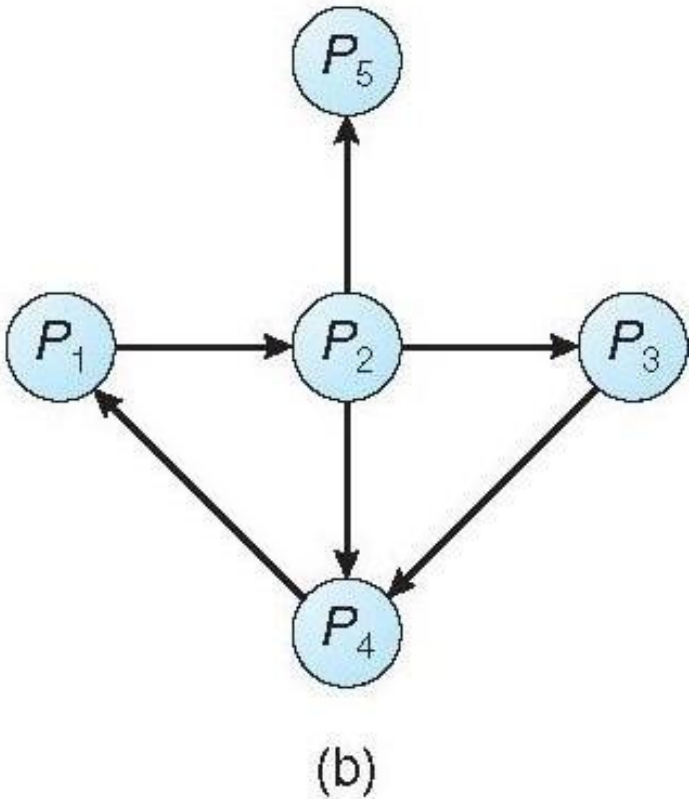
Single Instance for each Resource Type

- If all resources have only a ***single instance***, then for deadlock detection we use a **wait-for** graph.
  - **Wait-for** graph does not have resource nodes.
- 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

Single Instance for each Resource Type



Resource-Allocation Graph



Corresponding wait-for graph

Drawbacks of Resource Allocation Graph

- The wait-for graph is not applicable to a resource-allocation system with multiple instances of each resource type.

Detection of Deadlock for multiple Instance Resources

- Deadlock detection algorithm used for system with multiple instances of resource types uses data structures similar to the banking algorithm.
- 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 Pi is requesting k more instances of resource type Rj.

Detection ALgorithm

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

(a) ***Work = Available*** *//resources currently available with the system*

(b) For ***i* = 1,2, …, *n***, if $\boldsymbol{Allocation_i \neq 0}$, then

Finish*[i] = *false; otherwise, ***Finish*[i] = *true*** *//Process 'i' is assumed finished, no resources have been allocated to it.*

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

3. $\boldsymbol{Work = Work + Allocation_i}$ *//System reclaims resources.*

   Finish*[*i*] = *true *//Process 'i' is finished.*

   go to step 2

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

Detection Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?

Detection Algorithm Usage

- **Option 1:** 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.
- **Option 2:** We can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
  - We cannot only identify the deadlocked set of processes but also the specific process that caused the deadlock.
  - **Downside:** However, invoking the deadlock detection algorithm for every resource request will incur overhead.

Deadlock Recovery

- Two options for breaking the deadlock:
  - Process Termination: Abort one or more processes to break the circular wait.
    - Abort all deadlocked processes.
    - Or abort one process at a time until the deadlock cycle is eliminated
- Resource Preemption
  - Preempt some resources from one or more of the deadlock processes. Using resource preemption, we successively preempt some resource from processes and give these resources to other processes until the deadlock is broken.

Issues from Recovery from deadlock

Some issues:

- Selecting a victim
  - Which resources and which processes are to be preempted?
- 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

Live Lock

Livelock is another form of liveness failure.

- It is similar to deadlock; both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons.
- Whereas deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set.
- livelock occurs when a thread continuously attempts an action that fails.

Live Lock

- **Analogy:**
  - Livelock is similar to what sometimes happens when two people attempt to pass in a hallway:
  - One moves to his right, the other to her left, still obstructing each other's progress. Then he moves to his left, and she moves to her right, and so forth.
  - They aren't blocked, but they aren't making any progress.
- **Example**
  - Livelock can be illustrated with the Pthreads pthread_mutex_trylock() function, which attempts to acquire a mutex lock without blocking.