

CMPS 405: OPERATING SYSTEMS

Deadlock

This material is based on the operating system books by Silberschatz and Stallings and other Linux and system programming books.

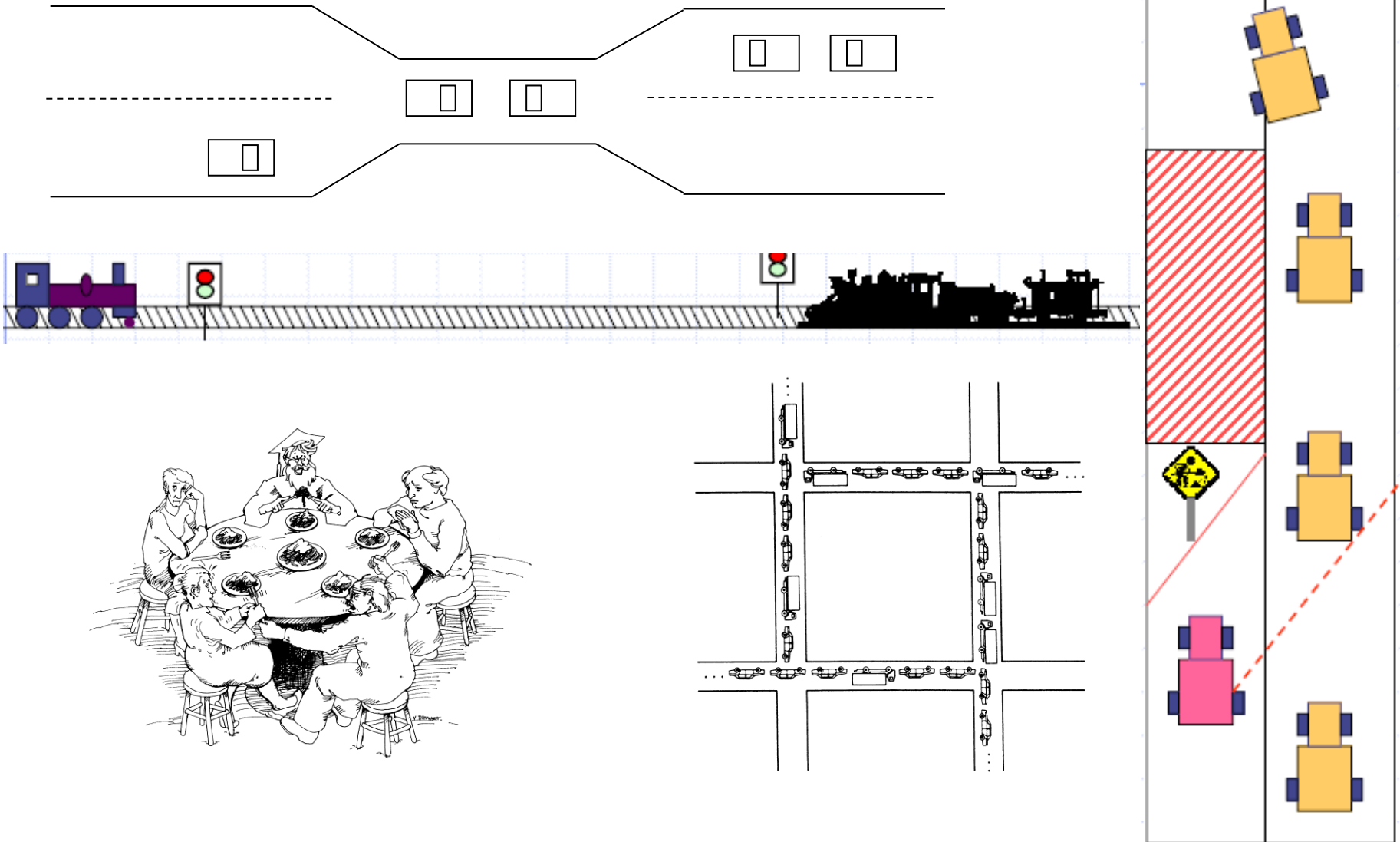
Objectives

- ❖ 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.

Topics

- ❖ Motivations to The Deadlock Problem
- ❖ System Model using a Resource-Allocation Graph (RAG)
- ❖ Deadlock Characterization
- ❖ Methods for Handling Deadlocks
 - Deadlock Prevention
 - Deadlock Avoidance
 - Resource-Allocation Graph Scheme
 - Banker's Algorithm
 - Deadlock Detection and Recovery
 - Recovery from Deadlock

Motivations to The Deadlock Problem



Motivations to The Deadlock Problem

❖ Definition of deadlock

- A situation in which a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the same set of processes.

❖ Example

- System has 2 disk drives.
- P_1 and P_2 each hold one disk drive and each needs the other one to proceed.

❖ Example

- semaphores A and B , initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Demonstrating Deadlock in Java

Thread A

```
class A implements Runnable
{
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            first.lock();
            // do something
            second.lock();
            // do something else
        }
        finally {
            first.unlock();
            second.unlock();
        }
    }
}
```

Thread B

```
class B implements Runnable
{
    private Lock first, second;

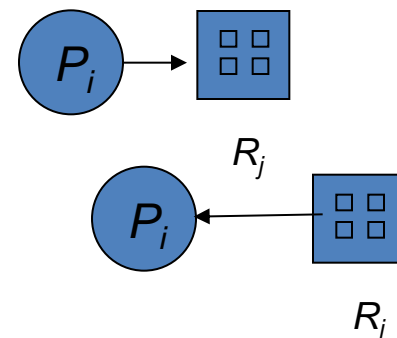
    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            second.lock();
            // do something
            first.lock();
            // do something else
        }
        finally {
            second.unlock();
            first.unlock();
        }
    }
}
```

Deadlock is possible if: threadA -> lockY -> threadB -> lockX -> threadA

System Model Using a Resource-Allocation Graph (RAG)

- ❖ $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
- ❖ $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
 - *Physical and logical resources: CPU cycles, memory space, I/O devices, files, folders, ..*
 - Each resource type R_i has W_i instances (a number of instances).
- ❖ Process utilization of a resource follows this order:
 - Request: request edge – directed edge $P_i \rightarrow R_j$
 - Use: assignment edge – directed edge $R_j \rightarrow P_i$
 - Release
- ❖ RAG: set of vertices V of types P and R and a set of edges E .

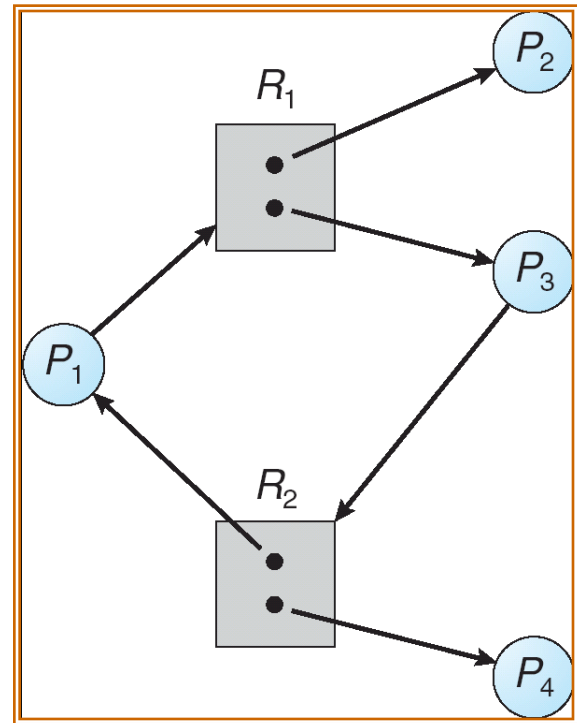
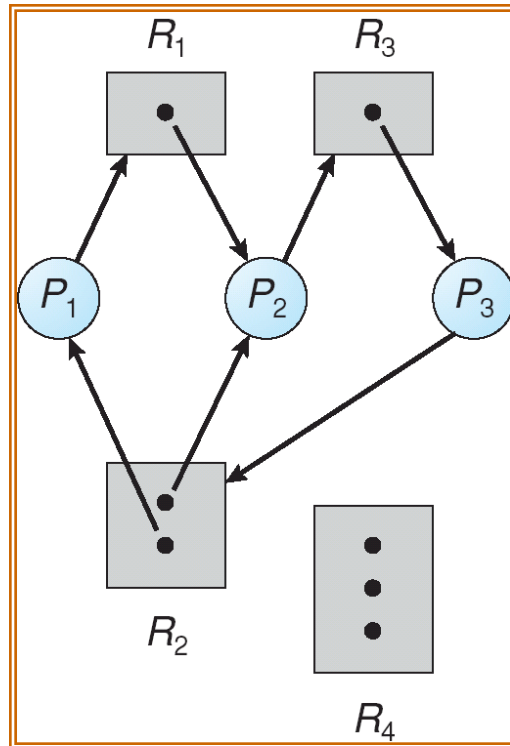
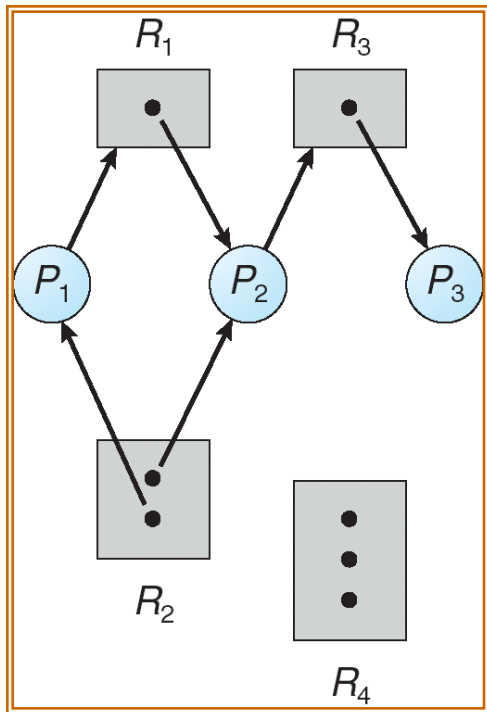


Deadlock Characterization

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 $\{P_0, P_1, \dots, P_n\}$ 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 .

Examples of Resource Allocation Graph



❖ Basic Facts:

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

Methods for Handling Deadlocks


1. Ensure that the system will *never* enter a deadlock state.
 - ❑ **Deadlock prevention:**
 - ❑ a set of methods for ensuring that at least one of the necessary conditions cannot hold.
 - ❑ **These methods constraining how requests for resources can be made.**
 - ❑ **Deadlock avoidance:**
 - ❑ Requires OS be given in advance which resources a process will request and use during its lifetime.
 - ❑ An adopted algorithm will decide for a process requesting a resource to wait or not.
2. Allow the system to enter a deadlock state and then recover. Needs:
 - ❑ Deadlock detection method and
 - ❑ Deadlock recovery strategy
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

❖ Mutual Exclusion –

- must hold for non-sharable resources but not required for sharable resources.
 - Read-only files are sharable resources that should be granted to multiple process access.

❖ Hold and Wait –

- Can use one of these two protocols to guarantee that whenever a process requests a resource, it does not hold any other resources:
 1. Require process to request and be allocated all its resources before it begins execution.
 - Implemented by requiring system calls requesting resources for a process to precede all other system calls.
 2. Allow process to request resources only when the process has none.
-  Have none → request some → use them → release them all →
- Disadvantages: Low resource utilization; starvation possible.
 - Example: Copy data from tape drive to disk file, sort disk file, and then print the result to the printer?

Tape drive – disk file – printer

Deadlock Prevention (Cont.)

- ❖ **No Preemption** – Two protocols can be used:
 1. 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 preempted (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.
 2. If a process requests resources that are currently not available, check if these resources are allocated to another process that is waiting for additional resources
 - If yes, preempt the resources from the waiting process and allocate them to the requesting process.
 - If No, requesting process must wait
 - *** Process can be restarted only when it is allocated the new resources it is requesting and the recovers any resources that have been preempted while it was waiting.
- Both protocols can only be applied to resources whose state can be easily saved and restored later (CPU registers, memory space)

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.
- ❖ The protocol:
 - Each resource type is assigned a unique integer. $F: R \rightarrow N$
 - Each process can request resources only in an increasing order of enumeration:
 - Use a single request for multiple instances of the same resources type.
 - Process should release any held resources $>$ the requested one prior to the request.
- ❖ Prove that Circular Wait cannot hold. *Hint: proof by contradiction.*

Deadlock Avoidance

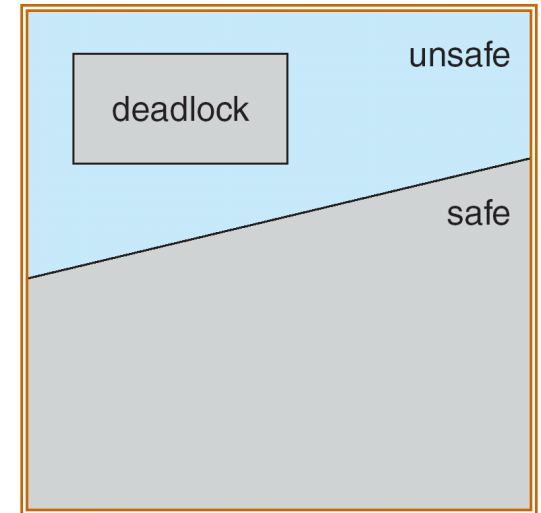
- ❖ Requires that the system has some additional *a priori* information available.
 - Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
 - The deadlock-avoidance algorithm dynamically examines the resource-allocation state 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 $\langle P_1, P_2, \dots, P_n \rangle$ 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 \Rightarrow no deadlocks.
- ❖ If a system is in unsafe state \Rightarrow possibility of deadlock.
- ❖ Avoidance \Rightarrow ensure that a system will never enter an unsafe state.



Avoidance algorithms

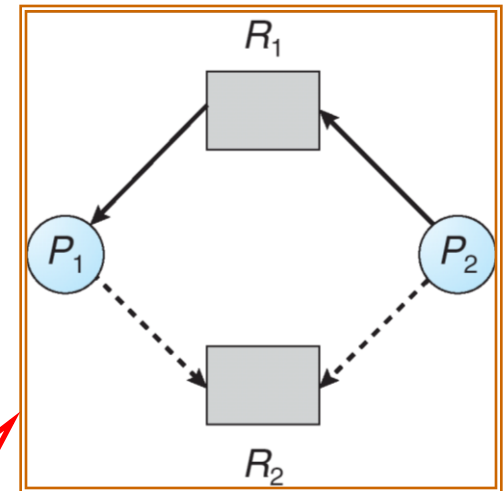
- ❑ Single instance of a resource type. Use a **resource-allocation graph**
- ❑ Multiple instances of a resource type. Use the **banker's algorithm**

Resource-Allocation Graph Scheme

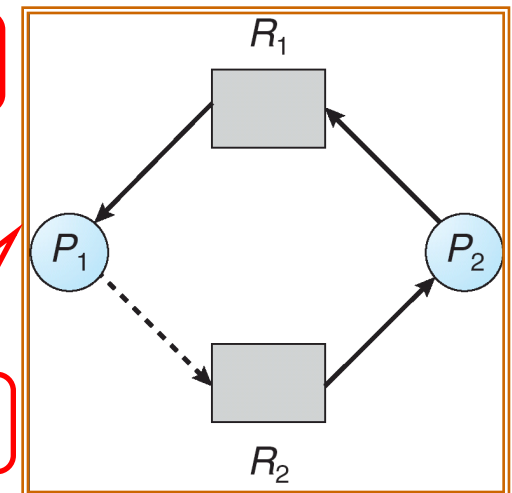
- ❖ *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j ; represented by a dashed line.
- ❖ Claim edge converts to request edge when a process requests a resource.
- ❖ Request edge converted to an assignment edge when the resource is allocated to the process.
- ❖ When a resource is released by a process, assignment edge reconverts to a claim edge.
- ❖ Resources must be claimed *a priori* in the system. Or, a process can claim resources if all of its edges are claim edges.

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



Safe State



Unsafe State

? **Banker's Algorithm**

- ❖ Used for the case of multiple instances of resources.
- ❖ 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**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j .
- ❖ **Allocation**: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j .
- ❖ **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

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

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 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	P_0 7 4 3
P_1	2 0 0	3 2 2		P_1 1 2 2
P_2	3 0 2	9 0 2		P_2 6 0 0
P_3	2 1 1	2 2 2		P_3 0 1 1
P_4	0 0 2	4 3 3		P_4 4 3 1

Safety Algorithm

1. Let **Work** and **Finish** be vectors of length m and n , respectively. Initialize:
 $Work = Available$
 $Finish[i] = false$ for $i = 0, 1, \dots, n-1$.
2. Find and i such that both:
 $Finish[i] = false$
 $Need_i \leq Work$
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.

Simply,
try to find an
execution sequence

***An algorithmic presentation for an easy implementation by programming languages.**

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 :

Max - Allocation

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	P0 7 4 3
P_1	2 0 0	3 2 2	p1 5 3 2	P1 1 2 2
P_2	3 0 2	9 0 2	p3 7 4 3	P2 6 0 0
P_3	2 1 1	2 2 2	p4 7 4 5	P3 0 1 1
P_4	0 0 2	4 3 3	p2 10 4 7	P4 4 3 1
			p0 10 5 7	

□ The system is in a safe state since

□ the sequence **< P1, P3, P4, P2, P0 >** satisfies safety criteria.

Resource-Request Algorithm for Process P_i

Request = 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

***An algorithmic presentation for an easy implementation by programming languages.**

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 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	
	A B C	A B C	A B C	<u>Need</u> A B C
P_0	0 1 0	7 5 3	3 3 2	
P_1	2 0 0	3 2 2		P_0 7 4 3
P_2	3 0 2	9 0 2		P_1 1 2 2
P_3	2 1 1	2 2 2		P_2 6 0 0
P_4	0 0 2	4 3 3		P_3 0 1 1
				P_4 4 3 1

- The system is in a safe state since
 - the sequence **< P1, P3, P4, P2, P0 >** 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.

Old state

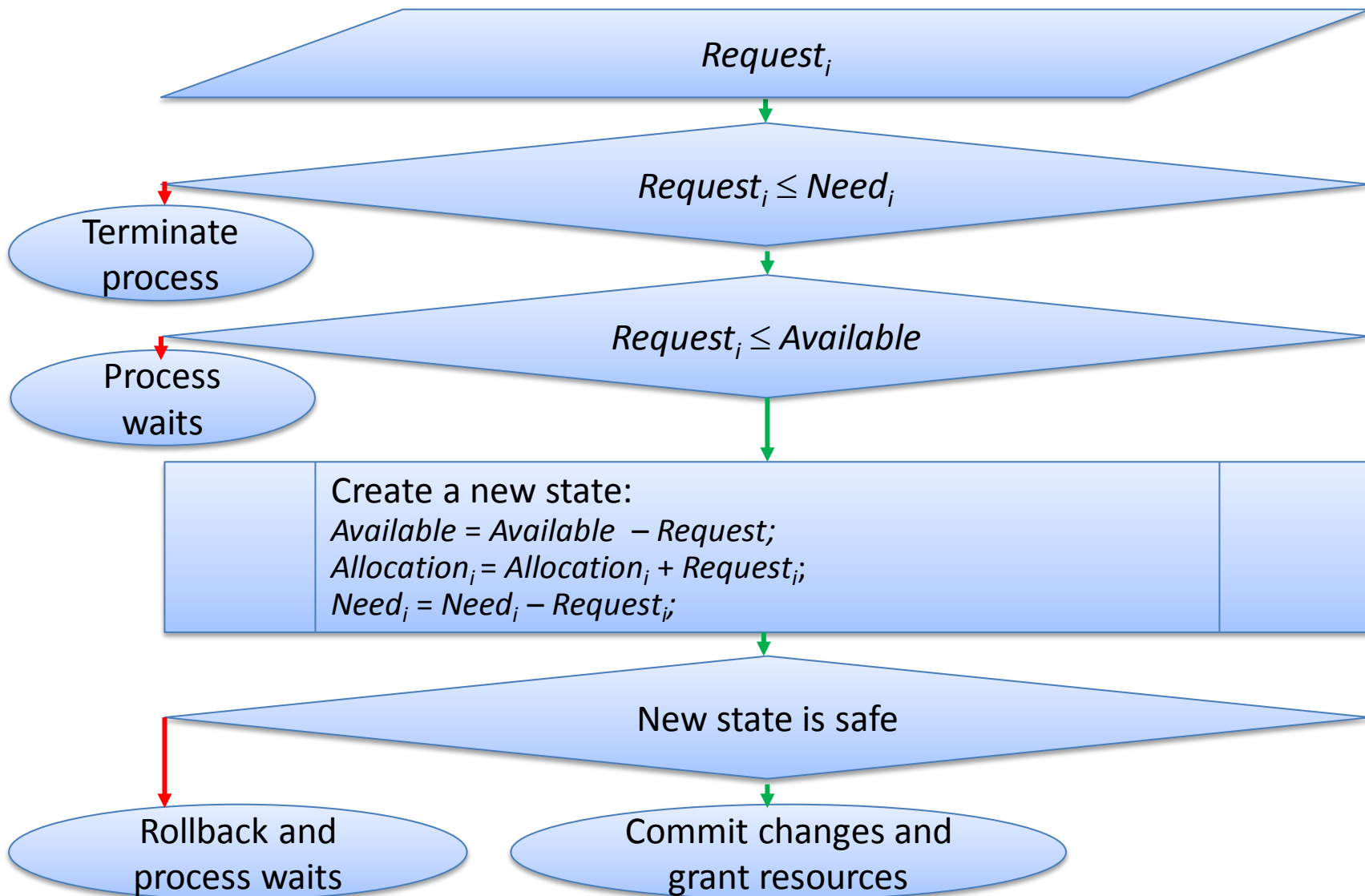
	<u>Allocation</u>	<u>Max</u>	<u>Needed</u>	<u>Available</u>
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

New state

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.
- ❖ Can request for (3,3,0) by P_4 be granted?
- ❖ Can request for (0,2,0) by P_0 be granted?

The Bankers Algorithm



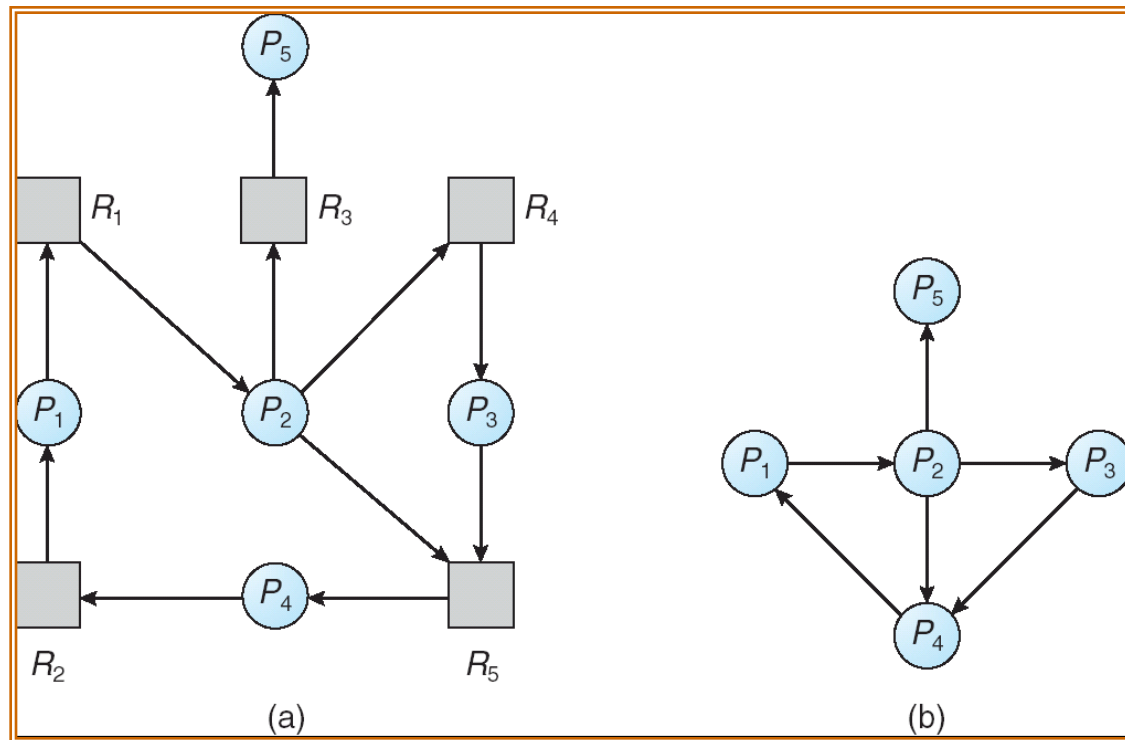
Deadlock Detection

- ❖ 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 $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- ❖ **Request:** An $n \times m$ 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 .

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 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ 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	1
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

1. Let *Work* and *Finish* be vectors of length m and n , respectively Initialize:
 - (a) *Work* = *Available*
 - (b) For $i = 1, 2, \dots, n$, 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.
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.

***An algorithmic presentation for an easy implementation by programming languages.**

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: 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?
 - Priority of the process.
 - How long process has computed, and how much longer to completion.
 - Resources the process has used.
 - Resources process needs to complete.
 - How many processes will need to be terminated.
 - 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.

Practical Exercise

Q. Consider the following snapshot of a system with 7 resources of type A, 6 of type B, 4 of type C, and 8 of type D.

Needed				
	A	B	C	D
P0	0	2	1	2
P1	0	1	0	1
P2	2	4	0	2
P3	0	1	2	0
P4	2	0	2	2

Current Allocation				
	A	B	C	D
P0	2	0	1	1
P1	0	1	2	1
P2	1	0	0	1
P3	1	2	1	0
P4	1	0	0	1

Will the request of process P0 for one instance of resource B and one instance of Resource D be granted? Show your work

Test 1:

Test2:

Test 3:

Available				
	A	B	C	D

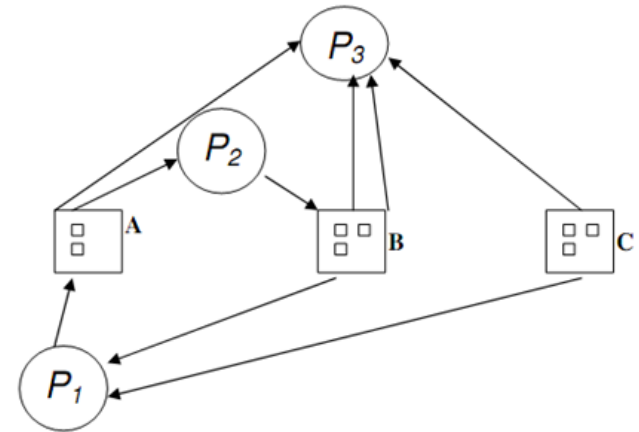
Execution sequence is:

Therefore,

Practical Exercise

Q Answer the following based on the resource allocation graph given below: (an arrow corresponds to one instance request or assignment).

- Write the allocation matrix.
- Find the available vector.
- Draw the wait-for graph
- Is the system in deadlock?



Practical Exercise

Q. Consider the following snapshot of a system with 6 resources of type A, 5 of type B, 4 of type C, and 7 of type D.

Maximum Demand				
	A	B	C	D
P0	2	2	2	3
P1	0	2	2	2
P2	3	4	0	3
P3	1	3	3	0
P4	3	0	2	3

Current Allocation				
	A	B	C	D
P0	2	0	1	1
P1	0	1	2	1
P2	1	0	0	1
P3	1	2	1	0
P4	1	0	0	1

a) Is the system in Safe state or not? Show your work.

Execution sequence is:

Therefore,

Available				
	A	B	C	D

Needs				
	A	B	C	D
P0				
P1				
P2				
P3				
P4				

Practical Exercise

b) Will the request of process P0 for one instance of resource B and one instance of Resource D be granted? Show your work.

Test 1:

Test2:

Test 3:

Available				
A	B	C	D	

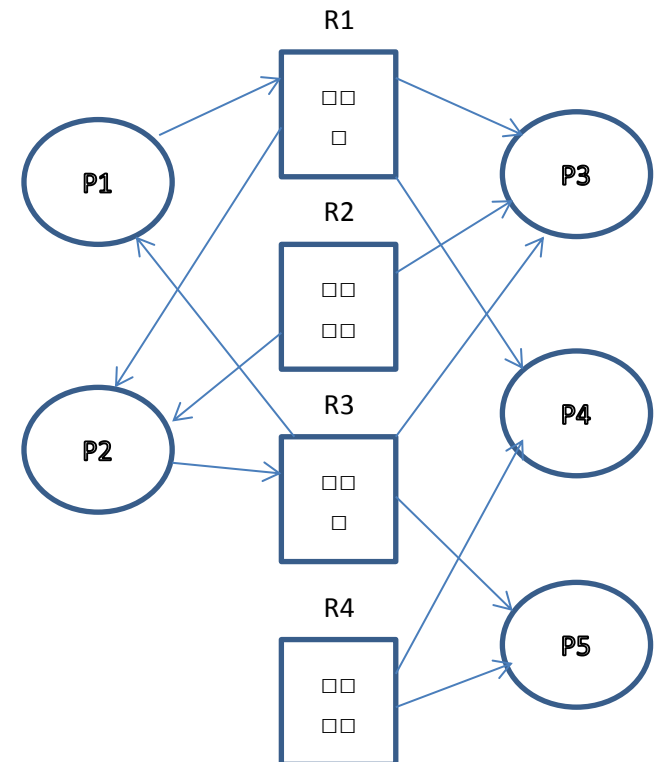
Execution sequence is:

Therefore,

Practical Exercise

Q. Answer the following based on the resource allocation graph given below. (An arrow corresponds to one instance request or assignment).

- a) Find the allocation matrix.
- b) Find the available vector.
- c) Draw the wait-for graph.
- d) Is the system in deadlock? Why?



Practice

- ❖ Check the code on BB.
 - Study the code.
 - Execute the code and try to reason their execution results.
 - Produce your own copy of the code with modifications aiming at deeper understanding of the subject.
- ❖ Implement deadlock prevention with
 - Resource allocation graphs
 - Banker's algorithm