OPERATING SYSTEMS

DEADLOCKS

DEADLOCKS

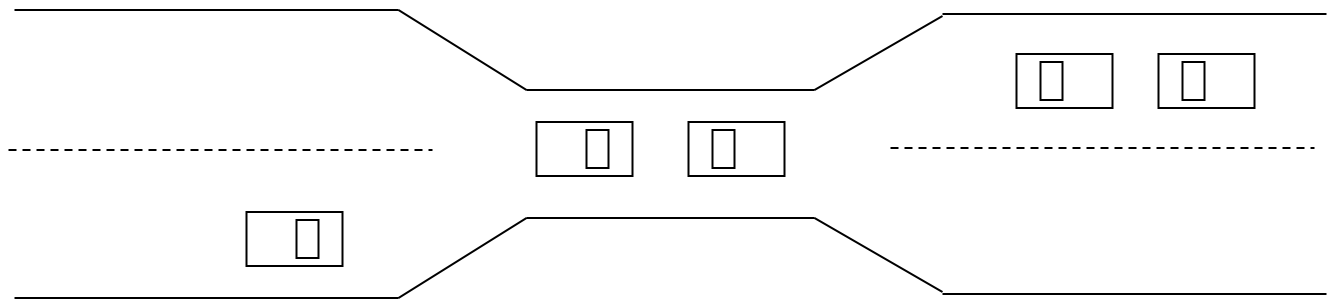
- In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

- This results from sharing resources such as memory, devices, files, links.

- Operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs.

DEADLOCKS

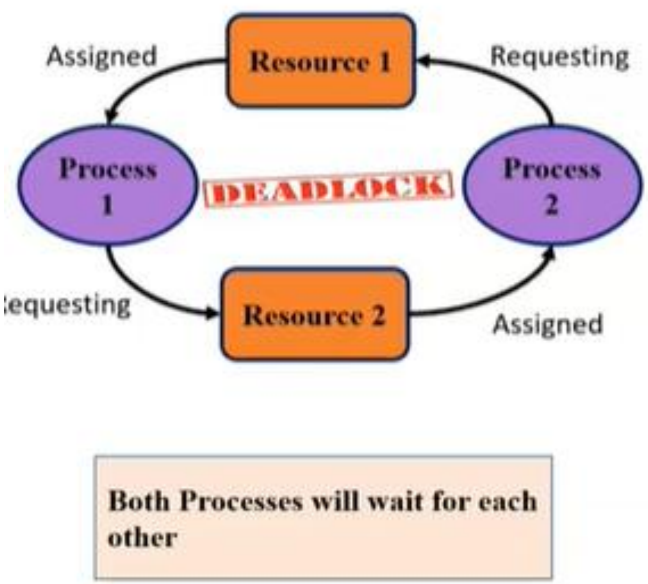
Bridge Crossing Example

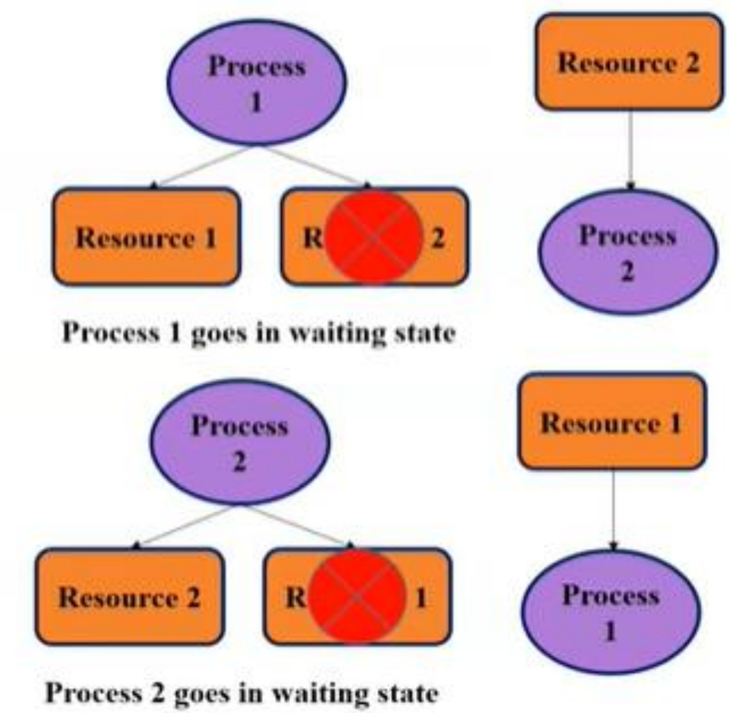


- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

Deadlock Principles

- **<u>A deadlock is a permanent blocking of a set of processes</u>**
  - ✓ a deadlock can happen while threads/processes are competing for system resources or communicating with each other





System Model

- A system consists of a finite number of resources (physical or logical) to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, semaphores and I/0 devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type *CPU* has as two instances. Similarly, the resource type *printer* may have five instances.
- If a process requests an instance of a resource type, the allocation of *any* instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly..
- Each process utilizes a resource as follows:
  - request
  - use
  - Release
- The request and release of resources are system calls. Request and release of resources that are not managed by the operating system can be accomplished through the wait() and signal() operations on semaphores or through acquisition and release of a mutex lock. For each use of a kernel managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource
- To illustrate a deadlocked state, consider a system with one printer and one DVD drive. Suppose that process P; is holding the DVD and process Pi is holding the printer. If P; requests the printer and P1 requests the DVD drive, a deadlock occurs. This example illustrates a deadlock involving the different resource type.
- A programmer who is developing multithreaded applications must pay particular attention to this problem. Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources

DEADLOCKS

DEADLOCK CHARACTERISATION

NECESSARY CONDITIONS

ALL of these four **must** happen simultaneously for a deadlock to occur:

Mutual exclusion

At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and Wait

A process holds a resource while waiting to acquire another resource held by other processes.

No Preemption

There is only voluntary release of a resource - nobody else can make a process give up a resource.

Circular Wait

A set { P0 , Pl, ... , P11 } of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held by P0.

Methods for Handling Deadlocks

- **Prevention**
  - Ensure that the system will *never* enter a <u>deadlock</u> state. Prevent any one of the 4 conditions from happening

- **Avoidance**
  - Ensure that the system will *never* enter an <u>unsafe</u> state. Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations.

- **Detection**
  - Allow the system to enter a deadlock state and then recover

- **Do Nothing**
  - Ignore the problem and let the user or system administrator respond to the problem; used by most operating systems, including Windows and UNIX

To ensure that deadlocks never occur, the system can either use a deadlock-prevention or a deadlock-avoidance scheme

Deadlock Prevention

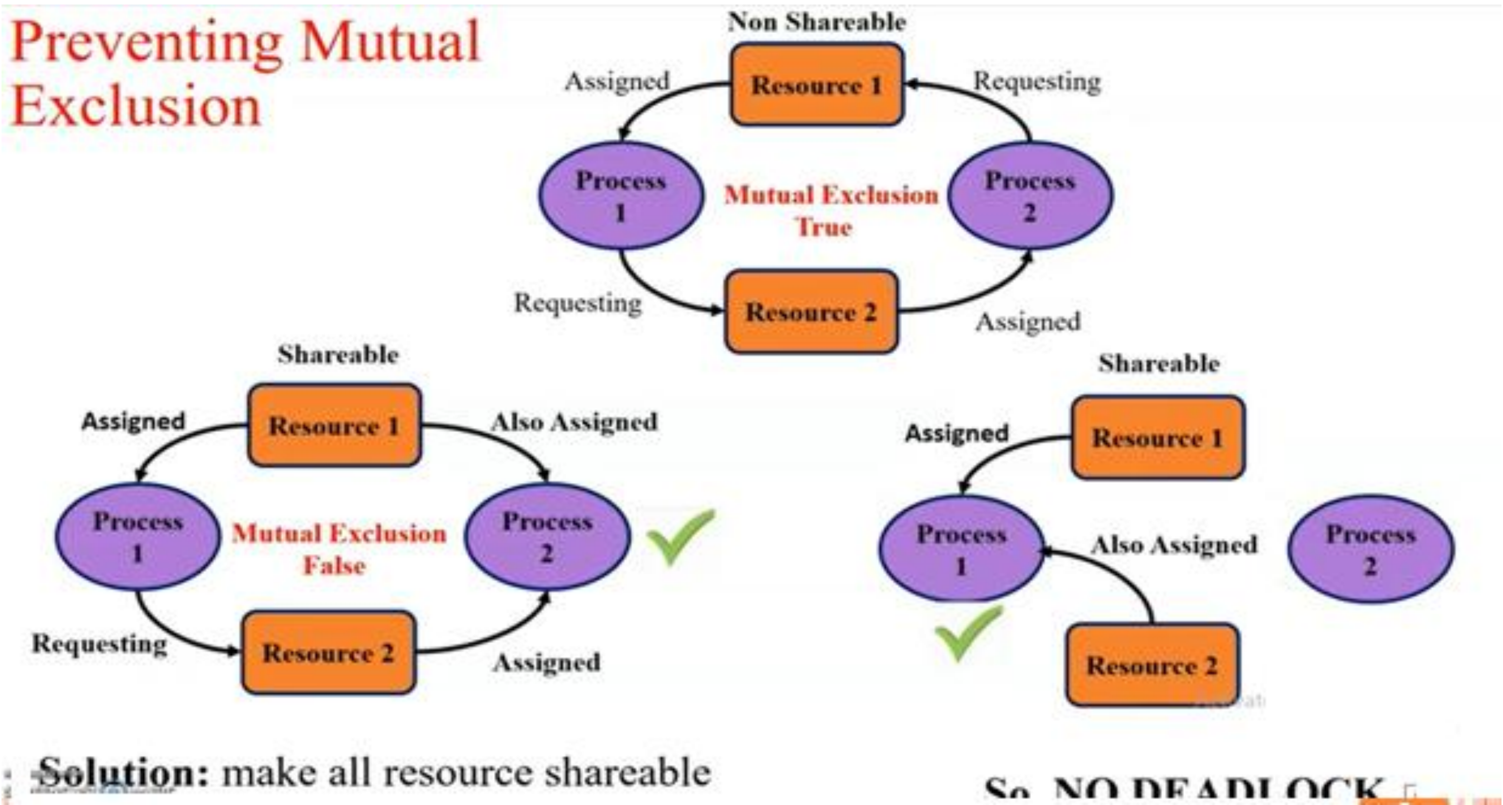
Do not allow one of the four conditions to occur.

Mutual exclusion:

Must holds for printers and other non-sharable resources.

Shared entities (read only files) don't need mutual exclusion (and aren't susceptible to deadlock.). A process never needs to wait for a sharable resource.

In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

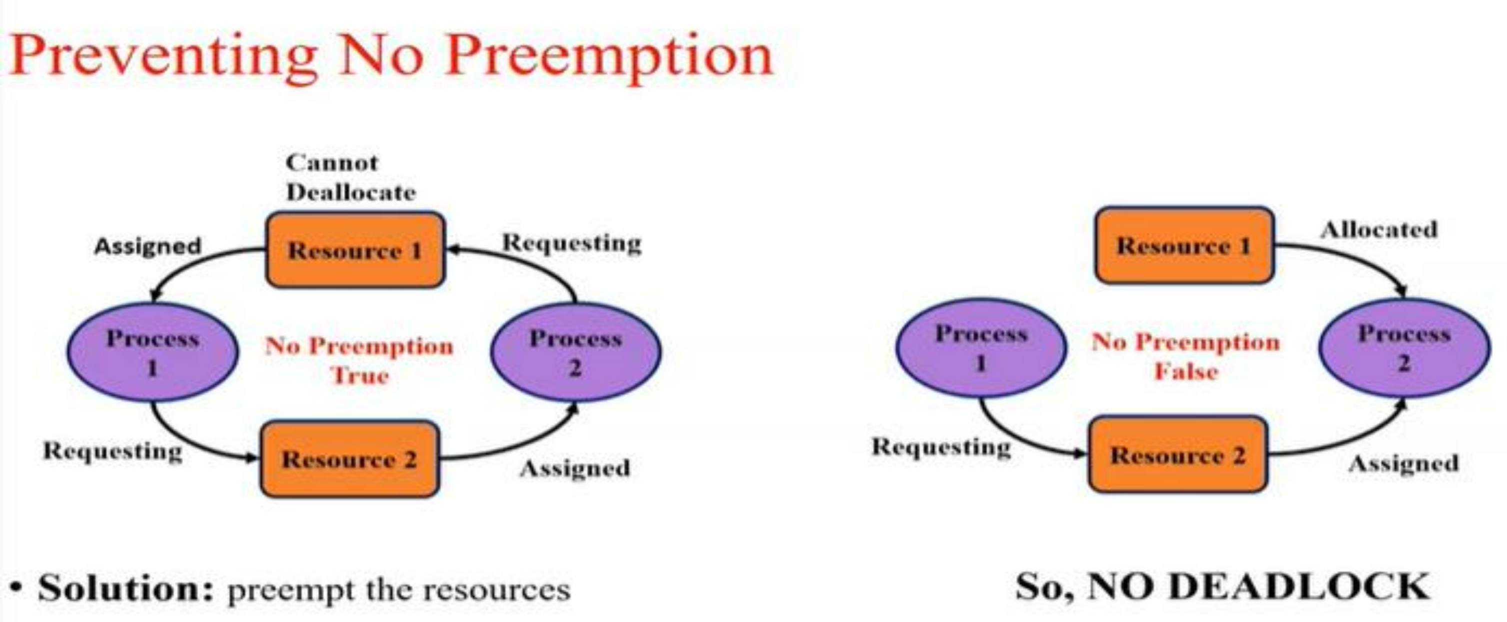


DEADLOCKS

Deadlock Prevention

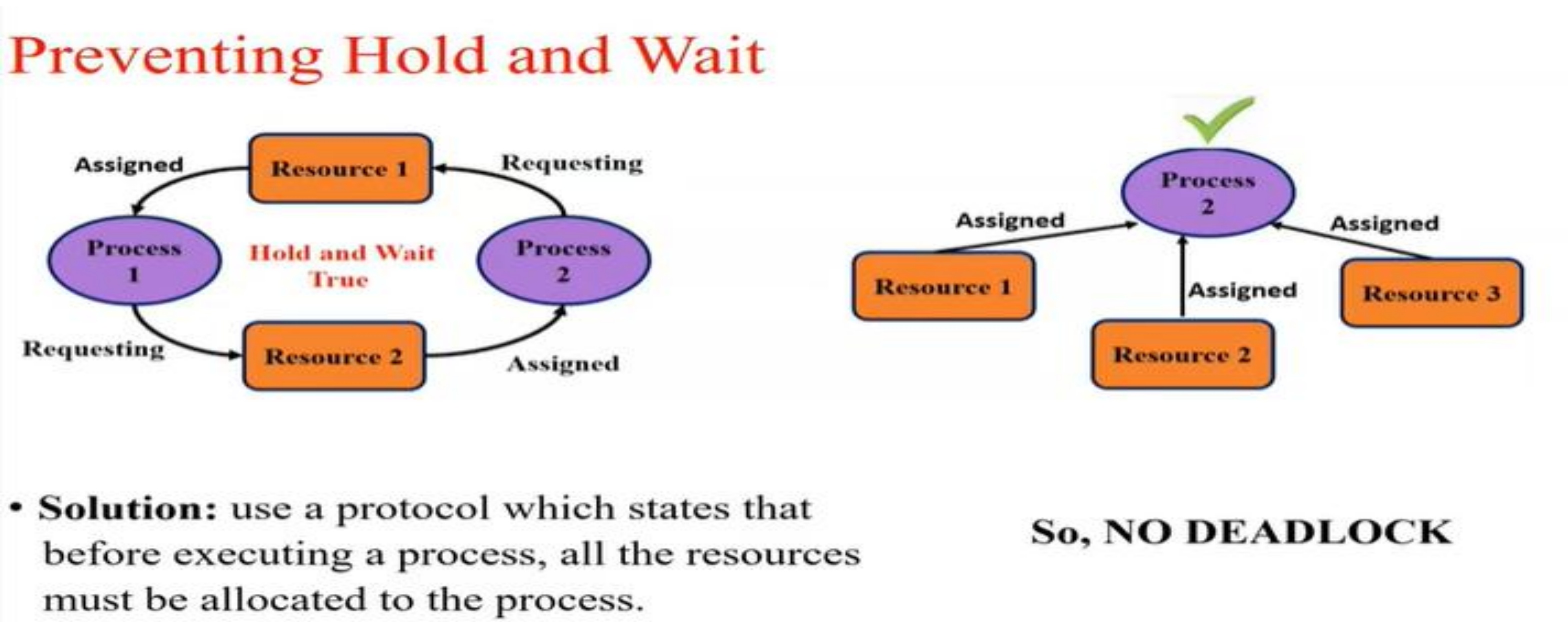
No preemption:

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held by the process are released
- Preempted resources are added to the list of resources for which the process is waiting
- A process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting .



Hold and wait:

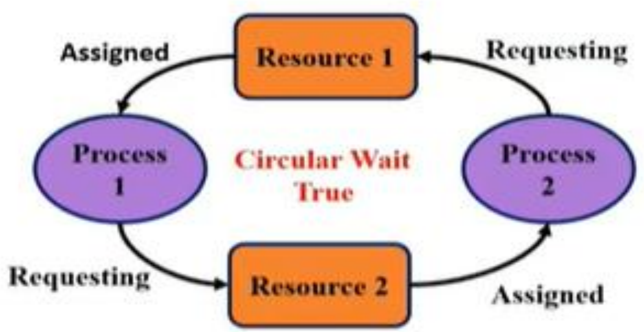
- 1. Conservative approach (Process will wait for all resources), 2. Do not hold, 3. Wait timeouts
- We must guarantee that whenever a process requests a resource, it does not hold any other resources
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
- An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- Require a process to request and be allocated all its resources before it begins execution, or allow a process to request resources only when the process has none
- Result: Low resource utilization; starvation possible



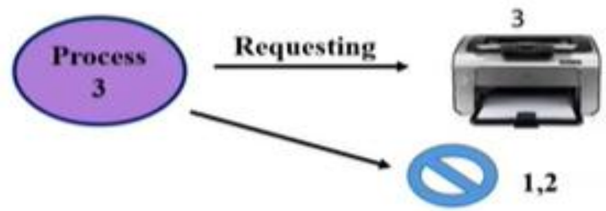
CIRCULAR WAIT

TO MAKE SURE THAT THIS CONDITION NEVER HOLDS, IMPOSE A <u>TOTAL ORDERING</u> OF ALL RESOURCE TYPES, AND REQUIRE THAT EACH PROCESS REQUESTS RESOURCES IN AN INCREASING ORDER OF ENUMERATION.

Preventing Circular Wait



Sr No	Resource	Priority
1	Tape Drive	1
2	Disk Drive	2
3	Printer	3
4	CPU	4



- **Solution:** number all the resources, and if the process needs to access a resource then it needs to send request in increasing order

Tape Drive =1 < 3
Disk Drive =2 < 3
CPU =4 > 3 so can request CPU

So, NO DEADLOCK

DEADLOCKS

Deadlock Avoidance

If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state.

Possible states are:

Deadlock	No forward progress can be made.
Unsafe state	A state that **may** allow deadlock.
Safe state	A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.

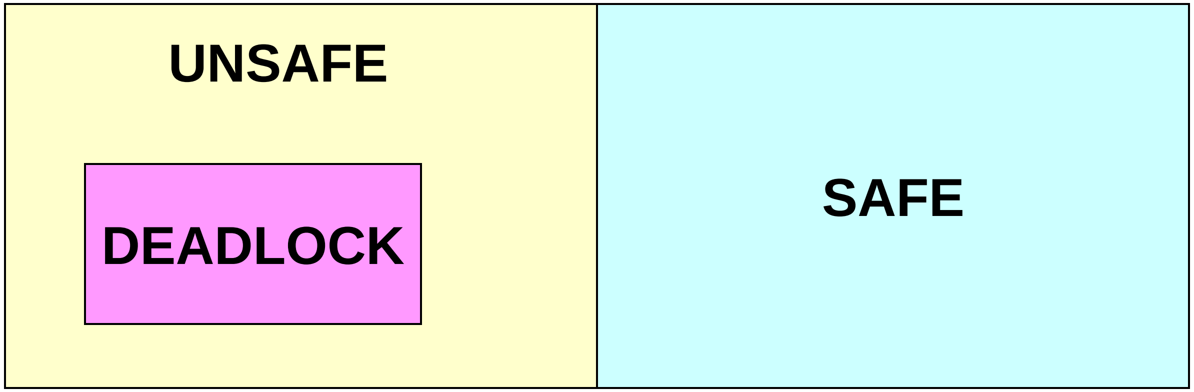
The rule is simple: If a request allocation would cause an unsafe state, do not honor that request.

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.

DEADLOCKS

Deadlock Avoidance

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.



If a system is in <u>safe</u> state $\Rightarrow$ no deadlocks.
A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure below). An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs. The behavior of the processes controls unsafe states.

If a system is in <u>unsafe</u> state $\Rightarrow$ possibility of deadlock

Avoidance $\Rightarrow$ ensure that a system will <u>never</u> enter an <u>unsafe</u> state

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 can be called as safe state if system is capable of providing the resources to each and every process up to its maximum value or as per its requirement. A safe state cant be a deadlock state.
- The deadlock state is known as unsafe state. In unsafe state, system is not able to allocate the different resources required by the processes.
- A system is in a safe state only if there exists a safe sequence
- A sequence of processes <P_1, P_2, …, P_n> is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make, can be satisfied by currently available resources plus resources held by all P_j, with $j < i$.
- That is:
  - If the 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.

Avoidance algorithms

- For a single instance of a resource type, use a resource-allocation graph

- For multiple instances of a resource type, use the banker's algorithm

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
- A resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Banker's Algorithm

- The name was chosen because the algorithm. could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

- Used when there exists **multiple** instances of a resource type

- Each process must **a priori** claim maximum use. This number may not exceed the total number of resources in the system

- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

- When a process gets all its resources, it must return them in a finite amount of time

- Two algorithms are used in bankers algorithm:
- **Safety Algorithm**:-determines whether the given system is in safe state or not.
- **Resource-Request Algorithm**:- Tells whether to assign a new resource to a process or not.

Data Structures for the Banker's Algorithm

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

- ***Available***: A vector of length m indicates the number of available resources of each type. If available $[j] = k$, there are k instances of resource type R_j available.

- ***Max***: *An n X m* matrix defines the maximum demand of each process. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.

- ***Allocation***: *An n X m* matrix defines the number of resources of each type currently allocated to each process. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.

- ***Need***: *An n X m* matrix indicates the remaining resource need of each process. 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

- Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize
  - *Work* := *Available*
  - *Finish*[*i*] := *false* for i = 0,1,2,…,n-1.
- Find an i (i.e. process *Pi*) such that both:
  - *Finish*[*i*] = *false*
  - *Need_i* <= *Work*
  - If no such i exists, go to step 4.
- *Work* := *Work* + Allocation_*i*
  - *Finish*[*i*] := *true*
  - go to step 2
- If *Finish*[*i*] = *true* for all i, then the system is in a safe state.

"BANKER's Algo"

Total A=10, B=5, C=7

Deadlock Avoidance

Deadlock Detection

Process	Allocation			Max Need			Available			Remaining Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3						
P_2	2	0	0	3	2	2						
P_3	3	0	2	9	0	2						
P_4	2	1	1	4	2	2						
P_5	0	0	2	5	3	3						

"BANKER's Algo"

Total A=10, B=5, C=7

Deadlock Avoidance

Deadlock Detection

Process	Allocation A	Allocation B	Allocation C	Max Need A	Max Need B	Max Need C	Available A	Available B	Available C	Remaining Need = Max - Allocation A	Remaining Need B	Remaining Need C	
P_1	0	1	0	7	5	3	3	3	2	7	4	3	P_1
P_2	2	0	0	3	2	2				1	2	2	P_2
P_3	3	0	2	9	0	2				6	0	0	P_3
P_4	2	1	1	4	2	2				2	1	1	P_4
P_5	0	0	2	5	3	3				5	3	1	P_5
	7	2	5										

Safe Sequence

Unsafe.

"BANKER's Algo"

Total A=10, B=5, C=7

Deadlock Avoidance
Deadlock Detection

Total A=10, B=5, C=7

Deadlock Avoidance
Deadlock Detection

Process	Allocation			Max Need			Current Available			Remaining Need = Max - Allocation		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2			
P_2	2	0	0	3	2	2	5	3	2			
P_3	3	0	2	9	0	2	7	4	3			
P_4	2	1	1	4	2	2	7	4	5			
P_5	0	0	2	5	3	3	7	5	5			
	7	2	5				10	5	7			

$$\begin{array}{ccc} 10 & 5 & 7 \\ -7 & -2 & -5 \\ \hline 3 & 3 & 2 \end{array}$$

Safe Sequence.

Unsafe.

$P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

Consider the matrix below. Use Banker's algorithm to find out the following:

a) Need Matrix
b) Is system in safe state? If yes then find out the safe sequence

	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				

	Allocation				Max				Available				Need				
	A	B	C	D	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	0	0	0	0	✓ ✓
P1	1	0	0	0	1	7	5	0					0	7	5	0	X
P2	1	3	5	4	2	3	5	6					1	0	0	2	X ✓
P3	0	6	3	2	0	6	5	2					0	0	2	0	✓ ✓
P4	0	0	1	4	0	6	5	6					0	6	4	2	X ✓
	2	9	10	12													

Total instances	A	B	C	D
	3	14	12	12

	Allocation				Max				Available				Need			
	A	B	C	D	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	0	0	0	0
P1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P3	0	6	3	2	0	6	5	2	2	14	11	8	0	0	2	0
P4	0	0	1	4	0	6	5	6	2	14	12	12	0	6	4	2
	2	9	10	12					3	14	12	12				

Add the resources allocated to P1 to available resources

Safe sequence: <P0, P2, P3, P4, P1>

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

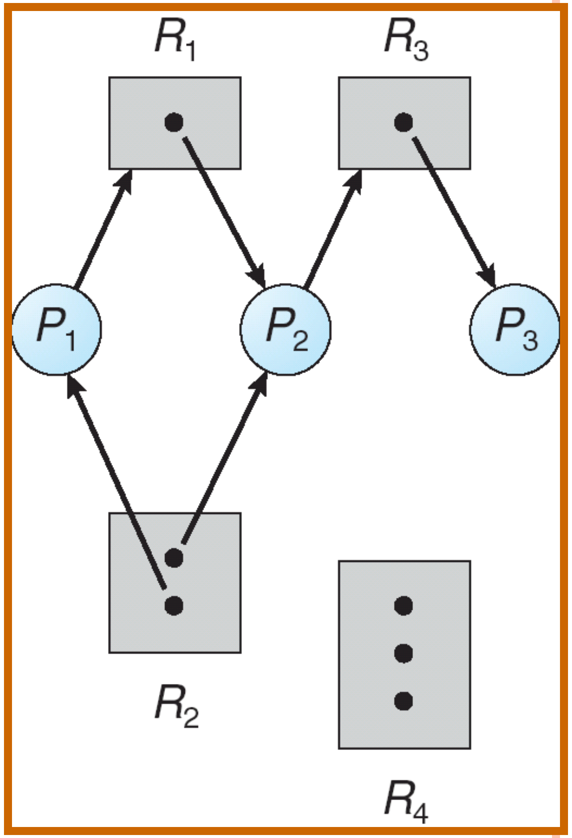
Resource-Allocation Graph Scheme

- Introduce a new kind of edge called a claim edge
- *Claim edge* $P_i \dashrightarrow R_j$ indicates that process P_i may request resource R_j; which is represented by a dashed line
- A claim edge converts to a request edge when a process **requests** a resource
- A request edge converts to an assignment edge when the resource is **allocated** to the process
- When a resource is **released** by a process, an assignment edge reconverts to a claim edge
- Resources must be **claimed *a priori*** in the system

Resource-Allocation Graph

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

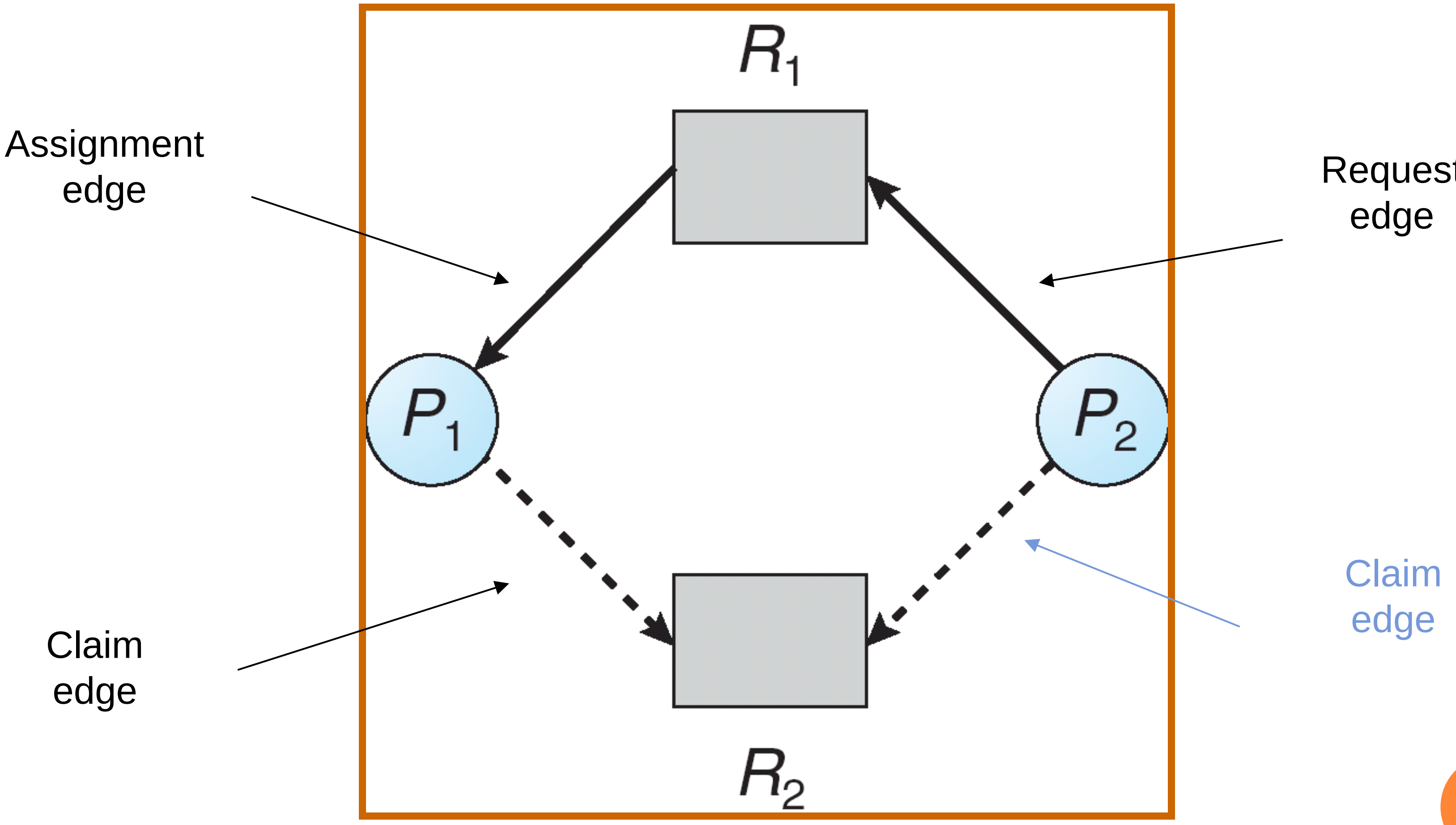
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge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



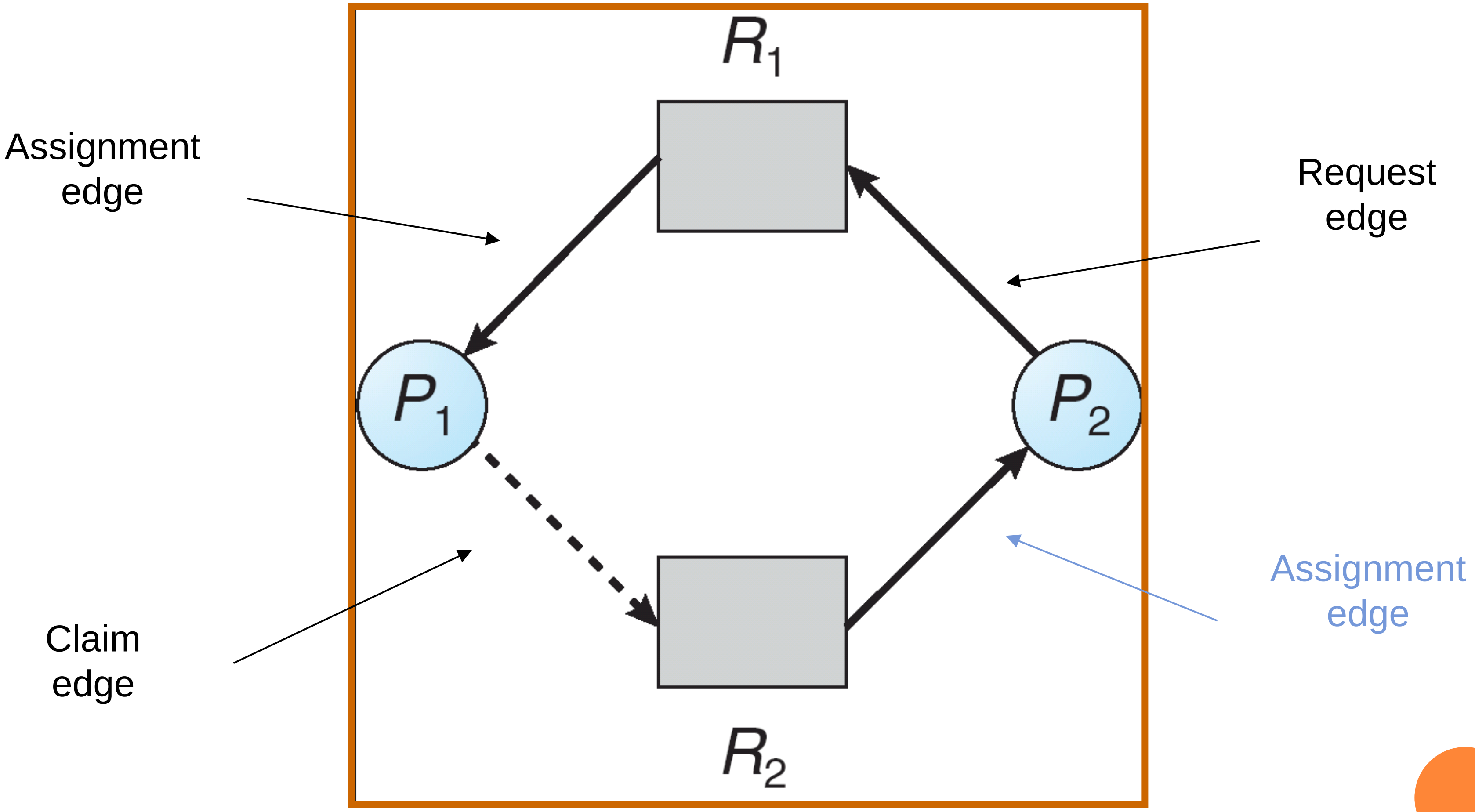
An arrow from the **process** to **resource** indicates the process is **requesting** the resource. An arrow from **resource** to **process** shows an instance of the resource has been **allocated** to the process.

Process is a circle, resource type is square; dots represent number of instance of resource.

Resource-Allocation Graph with Claim Edges



Unsafe State In Resource-Allocation Graph

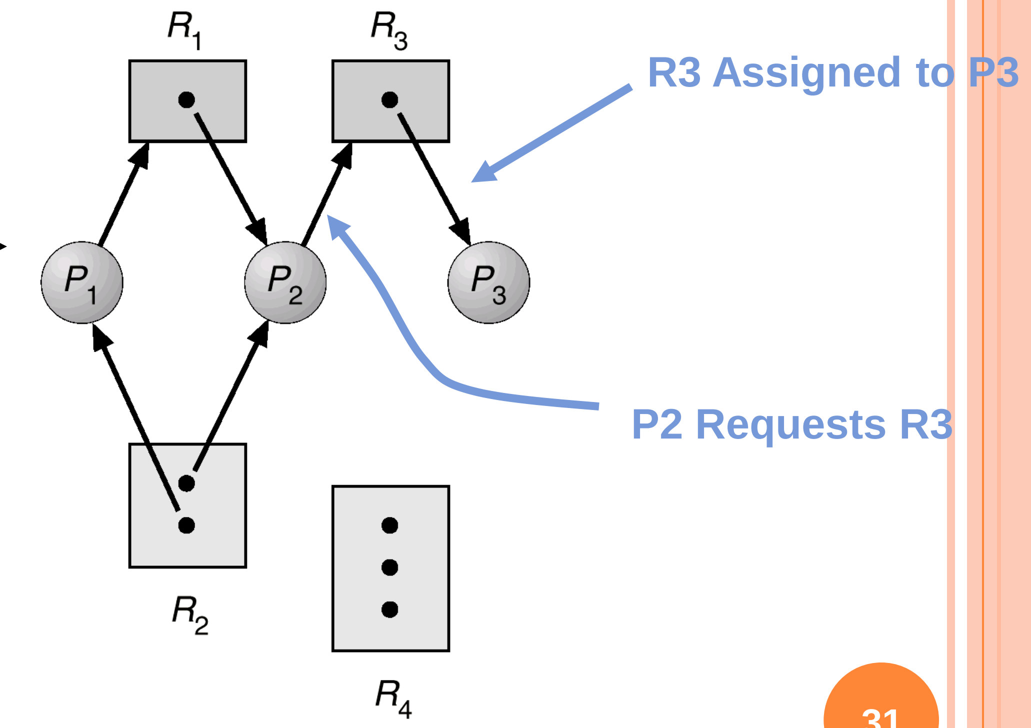


DEADLOCKS

RESOURCE ALLOCATION GRAPH

- If the graph contains no cycles, then no process is deadlocked.
- If there is a cycle, then:
  a) If resource types have multiple instances, then deadlock MAY exist.
  b) If each resource type has 1 instance, then deadlock has occurred.

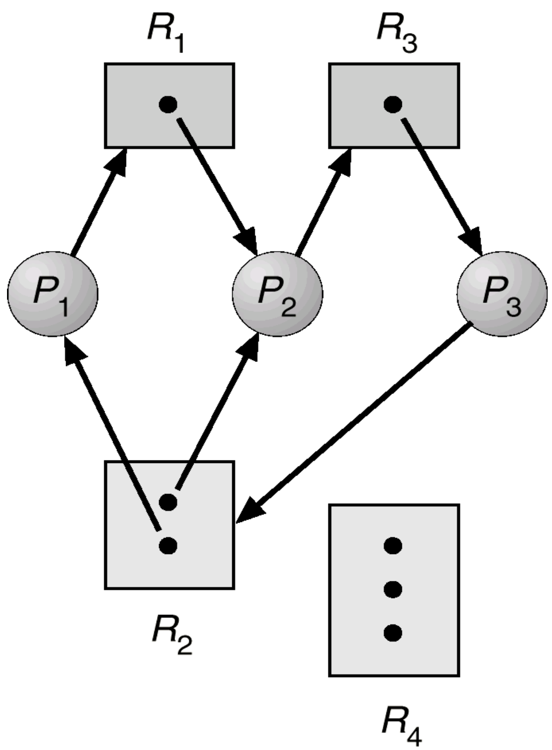
Resource allocation graph



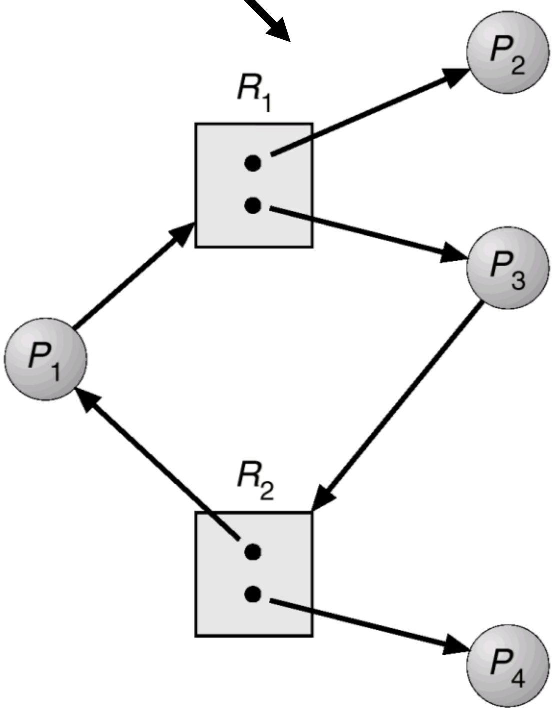
DEADLOCKS

RESOURCE ALLOCATION GRAPH

Resource allocation graph with a deadlock.



Resource allocation graph with a cycle but no deadlock.



Relationship of cycles to deadlocks

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

Deadlock Detection

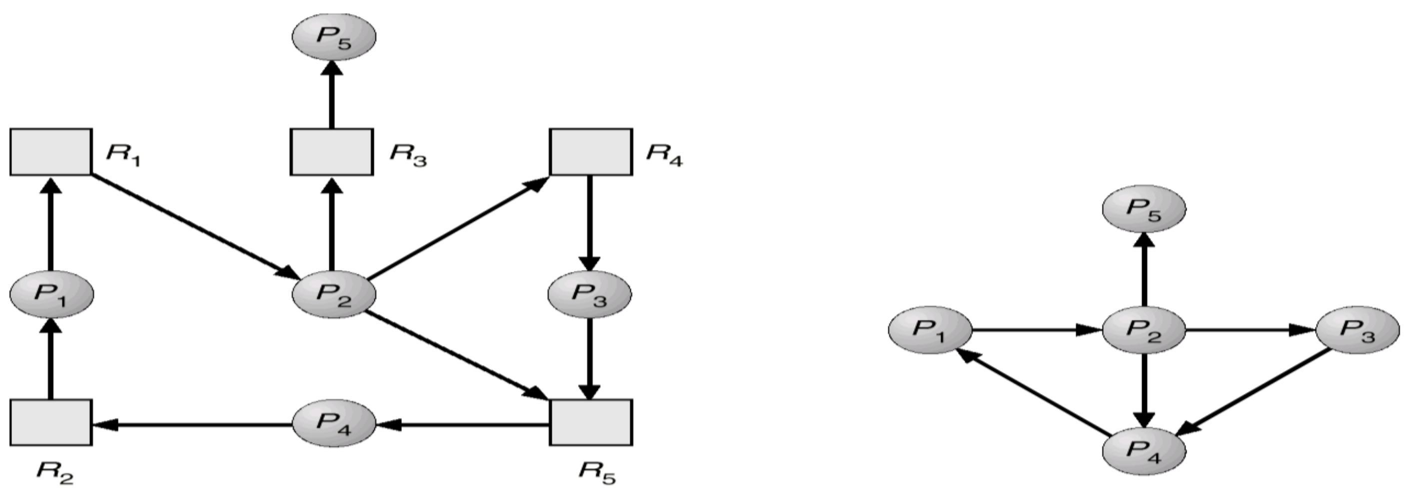
If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

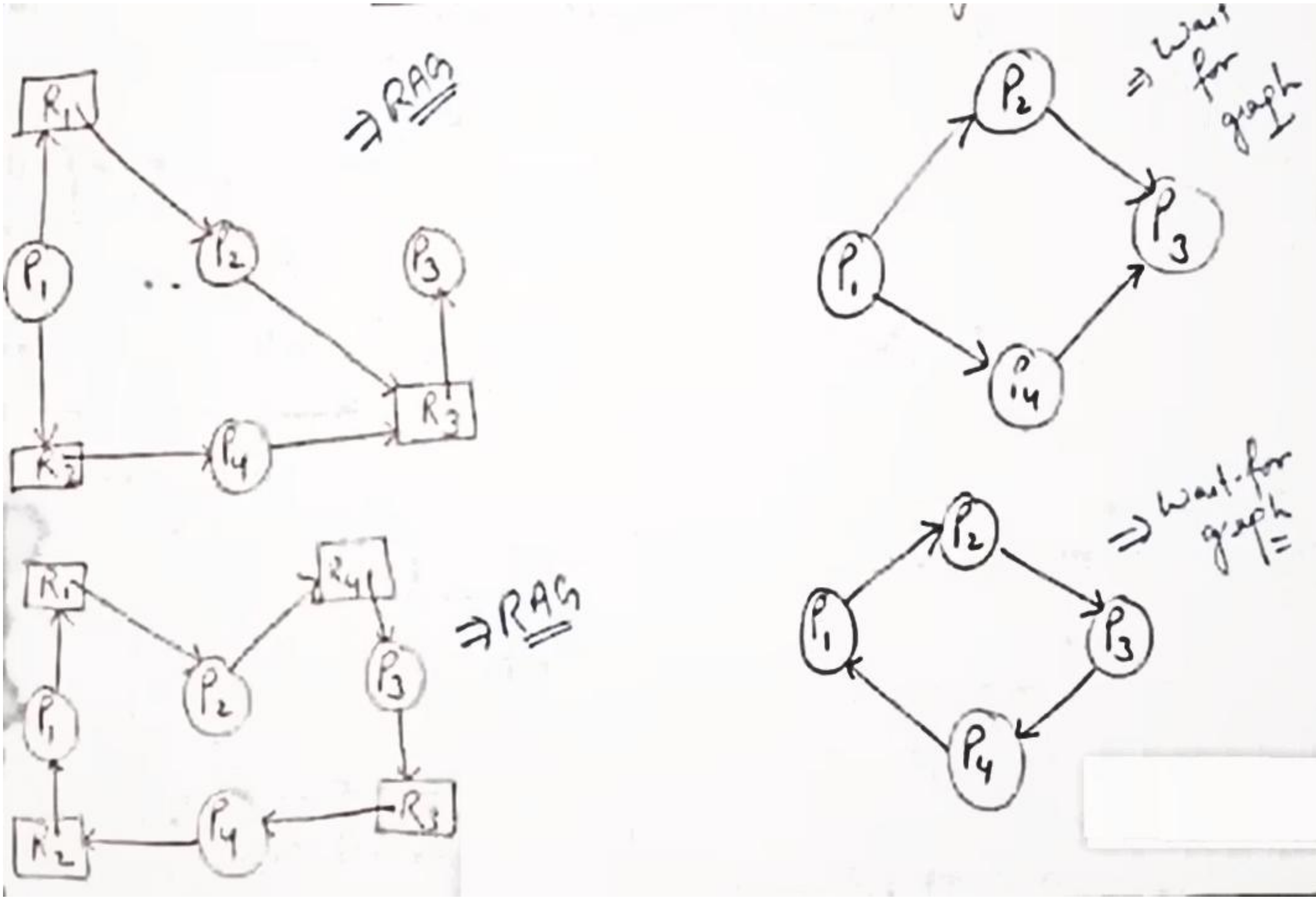
We elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type

SINGLE INSTANCE OF A RESOURCE TYPE

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- Wait-for graph == remove the resource nodes from the usual graph and collapse appropriate edges.
- An edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs. An edge Pi->Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi ->Rq and Rq -> Pj for some resource Rq
- A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of nxn operations, where n is the number of vertices in the graph.



(a) (b)



⇒ RAG
R1
R2
R3
P1
P2
P3
P4
⇒ Wait for graph I
⇒ RAG
R1
R2
R3
R4
P1
P2
P3
P4
⇒ Wait-for graph II

Deadlock Detection

SEVERAL INSTANCES OF A RESOURCE TYPE

If multiple instances of resources are there and there is a cycle in wait-for graph then deadlock may or may not be there

The algorithm employs several time-varying data structures that are similar to those used in the banker's 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] equals k, then process Pi is requesting k more instances of resource type Rj.

Deadlock Detection

1. Let work and finish be vectors of length m and n respectively. Initialize work[] = available[]. For i = 0,1,2,...n-1, if allocation[i] != 0 then // For all n processes

   finish[i] = false; otherwise, finish[i] = true;

2. Find an i process such that:

   finish[i] == false and request[i] <= work

   If no such i exists, go to step 4.

3. work = work + allocation[i]

   finish[i] = true

   goto step 2

4. if finish[i] == false for some i, then the system is in deadlock state.

   If finish[i] == false, then process P[i] is deadlocked.

You may wonder why we reclaim the resources of process P_i (in step 3) as soon as we determine that $Request_i \leq Work$ (in step 2b). We know that P_i is currently *not* involved in a deadlock (since $Request_i \leq Work$). Thus, we take an optimistic attitude and assume that P_i will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes P_0 through P_4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T_0, we have the following resource-allocation state:

	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	

Banker's algo (Safety algo)

	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	0	1	0
P_2	3	0	3	0	0	0	3	1	3
P_3	2	1	1	1	0	0	5	2	4
P_4	0	0	2	0	0	2	7	2	4
							7	2	6

Total

A	B	C
7	2	6

$< P_0\ P_2\ P_3\ P_1\ P_4 >$

Suppose now that process P_2 makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

	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

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P_0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

anker's algo(Safety algo)

	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	0	1	0
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

total :-

A	B	C
7	2	6

P0

P1, P2,P3,P4 are deadlocked processes

7.6.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes. In the extreme, then, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of processes but also the specific process that "caused" the deadlock. (In reality, each of the deadlocked processes is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may create many cycles in the resource graph, each cycle completed by the most recent request and "caused" by the one identifiable process.

Of course, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals—for example, once per hour or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and causes CPU utilization to drop.) If the detection algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles. In this case, we generally cannot tell which of the many deadlocked processes "caused" the deadlock.

DEADLOCKS

Deadlock Recovery

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

PROCESS TERMINATION:

- Abort all deadlocked processes-- this is expensive. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later
- Abort one process at a time until the deadlock cycle is eliminated (time consuming).
- Select which process to terminate based on priority, time executed, time to completion, needs for completion, or depth of rollback
- In general, it's easier to preempt the resource, than to terminate the process.

RESOURCE PREEMPTION:

To eliminate deadlocks using resource preemption, 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**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost.
- **Rollback** the preempted process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.
- **Starvation** -Ensure that a process can be picked as a victim" only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.