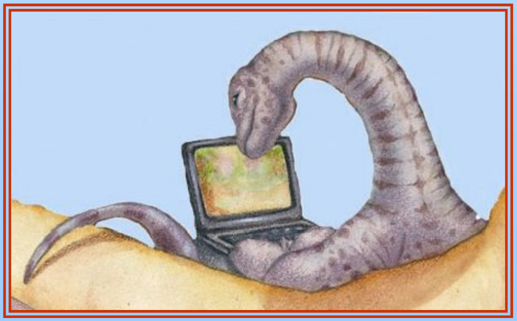
Chapter 7: Deadlocks



Chapter 7: Deadlocks

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

Chapter 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.

Deadlock

- A process requests resources;
- If the resources are not available at that time, the process goes to waiting state.
- Sometimes a waiting process is never again able to change state, because the resources it requested are held by other waiting processes.
- This situation is called deadlock.

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
  (**Eg., Resource type Printer has two instances**)
- Each process utilizes a resource as follows:
  - request (wait if it is used by another process)
  - use
  - Release
- The request and release of resources are system calls (request() and release()).

Deadlock Characterization

Necessary conditions

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** At least one resource must be held in a non-sharable mode; that is 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:** Resources can not be preempted; that is 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 $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

  P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

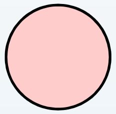
Resource-Allocation Graph

- Deadlocks are described using 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 $P_i \rightarrow R_j$ (P_i has requested an instance of resource type R_j and is currently waiting for that resource).
- **assignment edge** – directed edge $R_j \rightarrow P_i$ (an instance of resource type R_j has been allocated to process P_i)

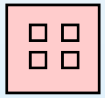
- When process P_i requests an instance of resource type R_j, a **request edge** is inserted in the resource allocation graph.
- When this request is fulfilled, the request edge is transformed into **assignment edge**.
- Assignment edge is deleted when a resource is released.

Resource-Allocation Graph (Cont.)

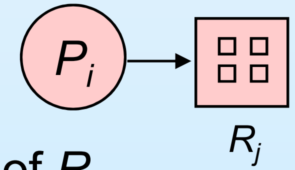
- Process



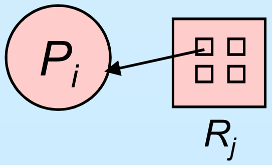
- Resource Type with 4 instances



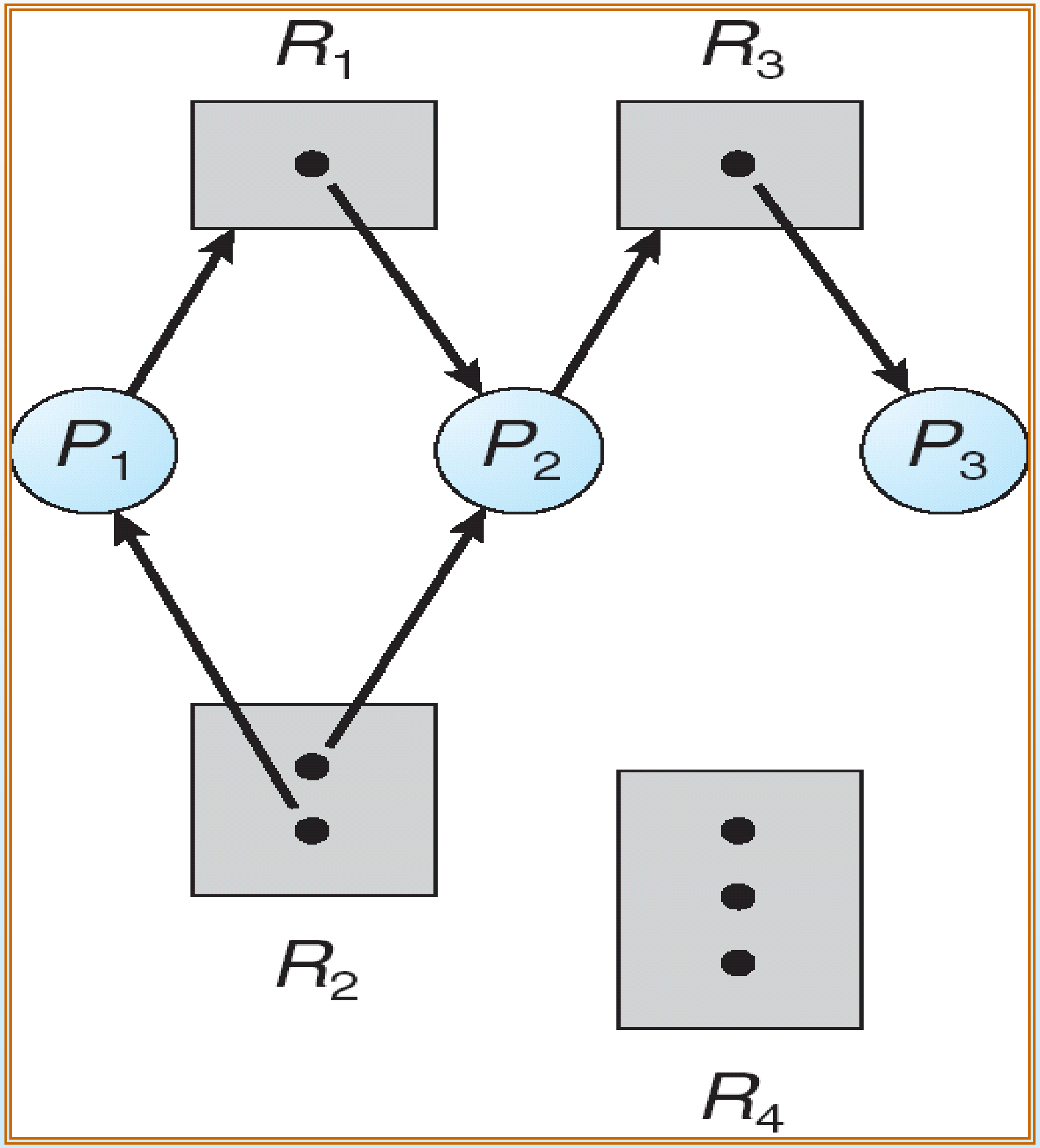
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

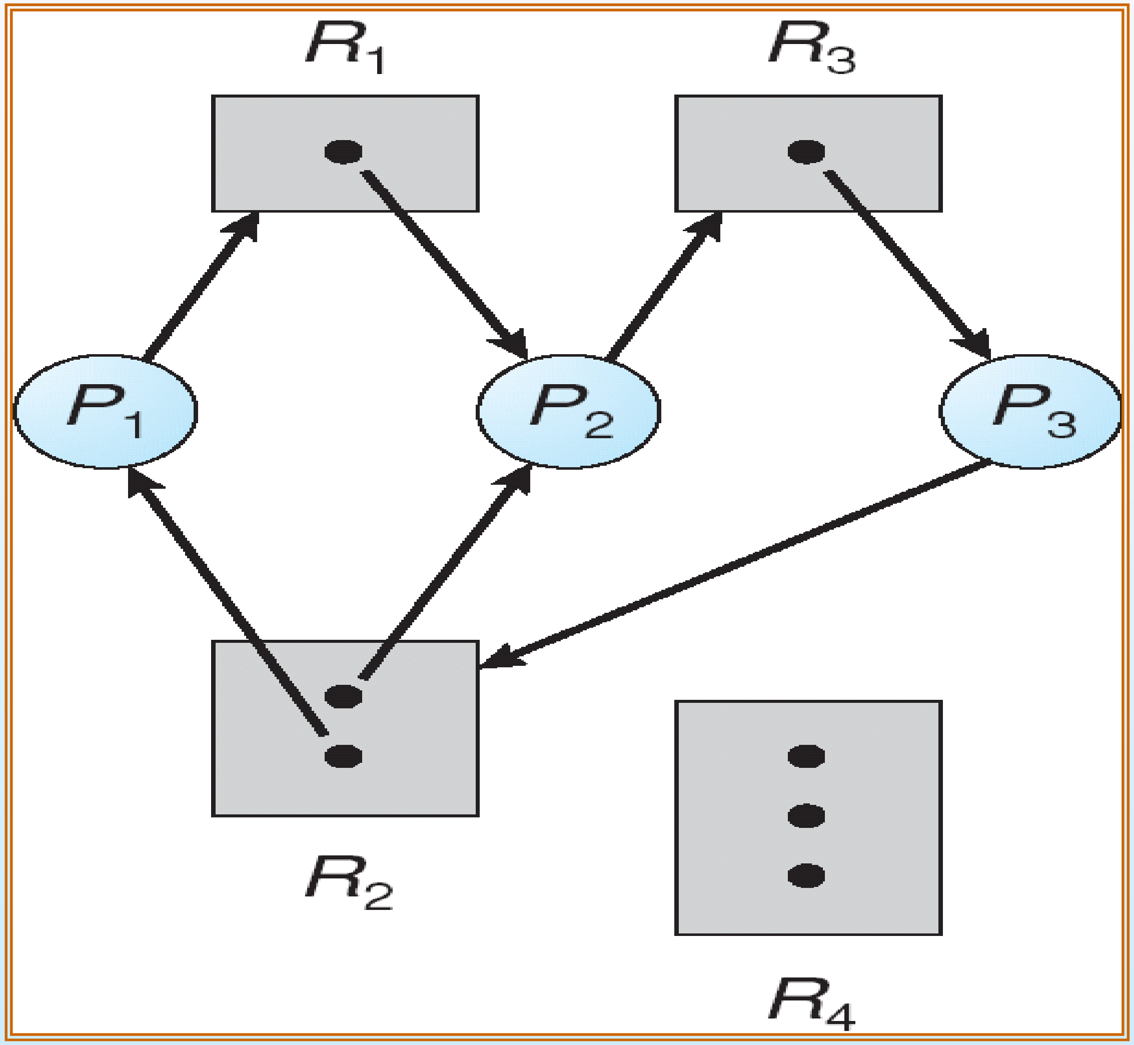


- Resource allocation graph in the figure shows the following situation
- Resource instances:
  - One instance of resource type R_1
  - Two instances of resource type R_2
  - One instance of resource type R_3
  - Three instances of resource type R_4
- The sets P, R and E are
  - $P = \{ P_1, P_2, P_3 \}$
  - $R = \{ R_1, R_2, R_3, R_4 \}$
  - $E = \{ P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3 \}$
- 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 R1and an instance of R2 and is waiting for an instance of R3.
  - Process P3 is holding an instance of R3.

Basic Facts

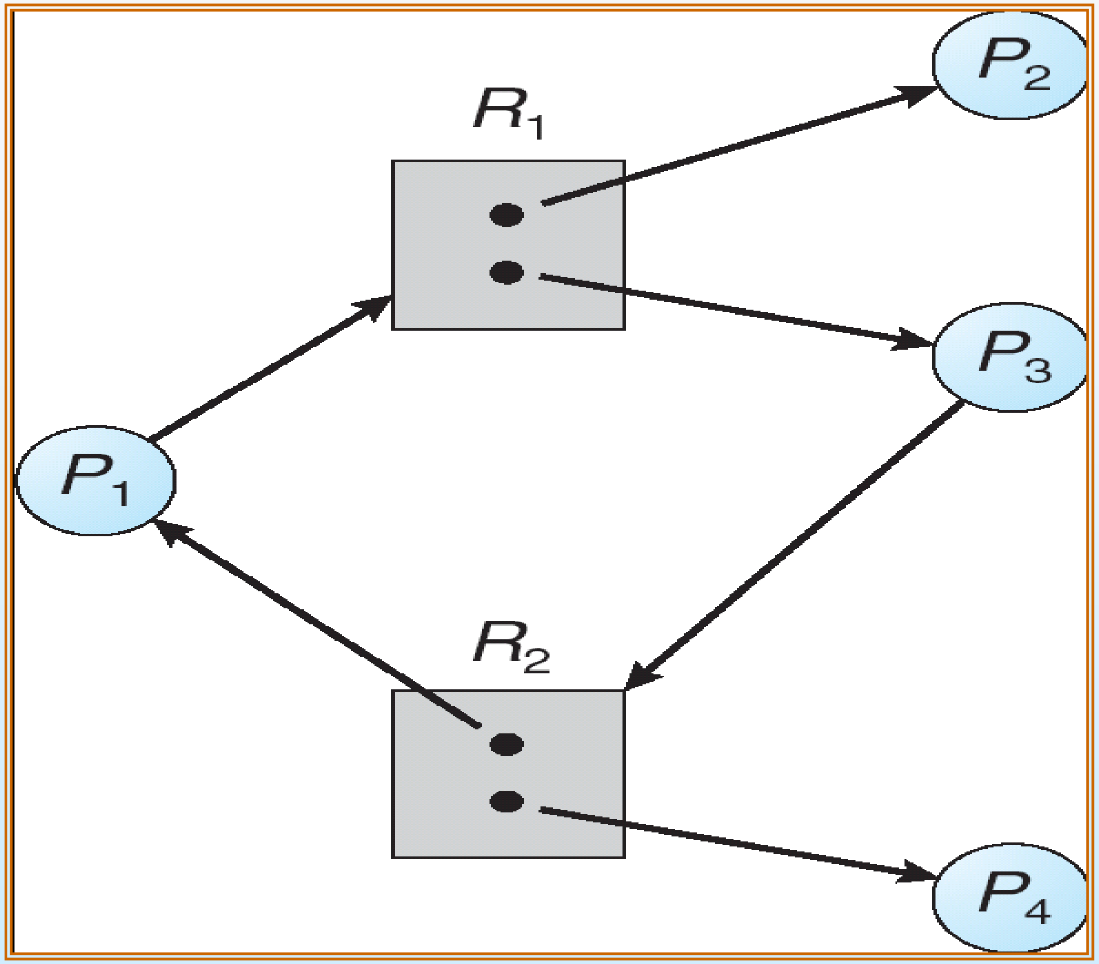
- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$ dead lock may exist.
  - if only one instance per resource type, then deadlock. In this case a cycle in a graph is both a necessary and sufficient condition for the existence of a deadlock.
  - if several instances per resource type, cycle does not necessarily imply that deadlock has occurred.

Resource Allocation Graph With A Deadlock



- Suppose that process P3 requests an instance of R2; so P3 $\rightarrow$ R2 is added to the graph.
- At this point, two minimal cycles exists.
  - $P_1 \rightarrow R_1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$
  - $P_2 \rightarrow R_3 \rightarrow P3 \rightarrow R2 \rightarrow P2$
- Processes P1, P2, P3 are deadlocked.
- P2 is waiting for the resource R3 which is held by process P3. process P3 is waiting for either P1 or P2 to release R2. In addition, P1 is waiting for P2 to release R1.

Resource Allocation Graph With A Cycle But No Deadlock



- In this Figure, we have a cycle.
- P1 $\rightarrow$ R1 $\rightarrow$ P3 $\rightarrow$ R2 $\rightarrow$ P1
- However there is no deadlock.
- If process P4 releases one instance of R2, then that resource can be allocated to P3, breaking the cycle.

Methods for Handling Deadlocks

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

Deadlock Prevention

- For a deadlock to occur, each of the following four necessary conditions must hold.

I. **Mutual Exclusion**
II. **Hold and Wait**
III. **No Preemption**
IV. **Circular Wait**

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

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources (Eg. read only files); must hold for nonsharable resources (Eg. Printer)
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.

  Method I

  - Require process to request and be allocated all its resources before it begins execution,

  Method II

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

Eg., Process that copies data from a DVD drive to a file on a disk, sort the file, and then prints the result.

When Method I is used

- Here all the 3 resources must be requested at the beginning of the process

When Method II is used

- Process initially request only DVD and disk file ;
- it copies from the DVD drive to the disk and then releases both;
- The process then must again request disk file and the printer;
- After copying the disk file to the printer it releases both.

Disadvantages of both methods

- 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 (that is, process must wait), then all resources currently being held are preempted (implicitly released).
  - The 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.
  - Eg. tape drive, disk drive, printer

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.

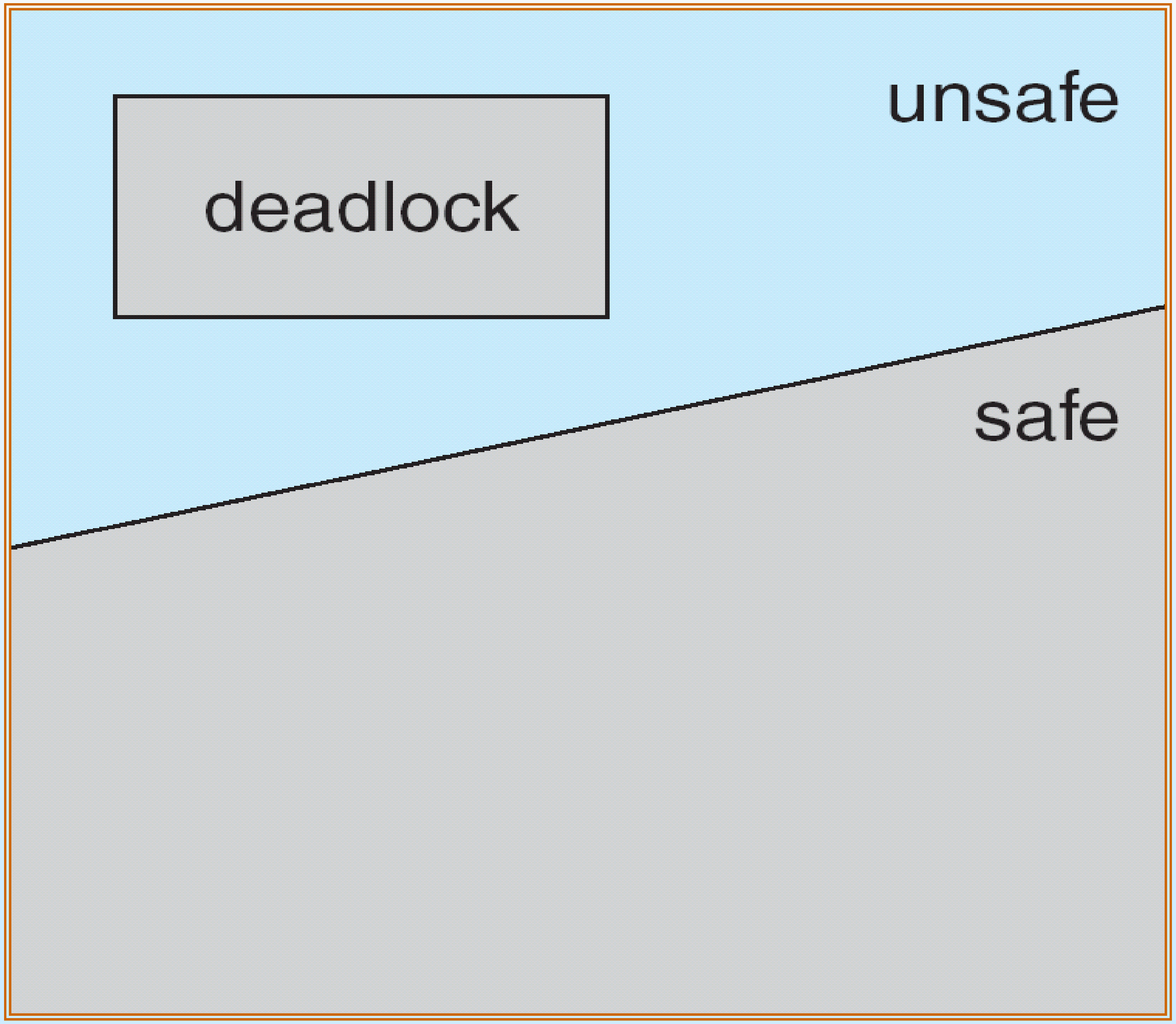
Safe State

- 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 safe sequence of all processes.
- Sequence <P_1, P_2, …, P_n> is safe if 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.
  - 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.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State



Eg., Consider a system with 12 magnetic tape drives. At time t_0 ,

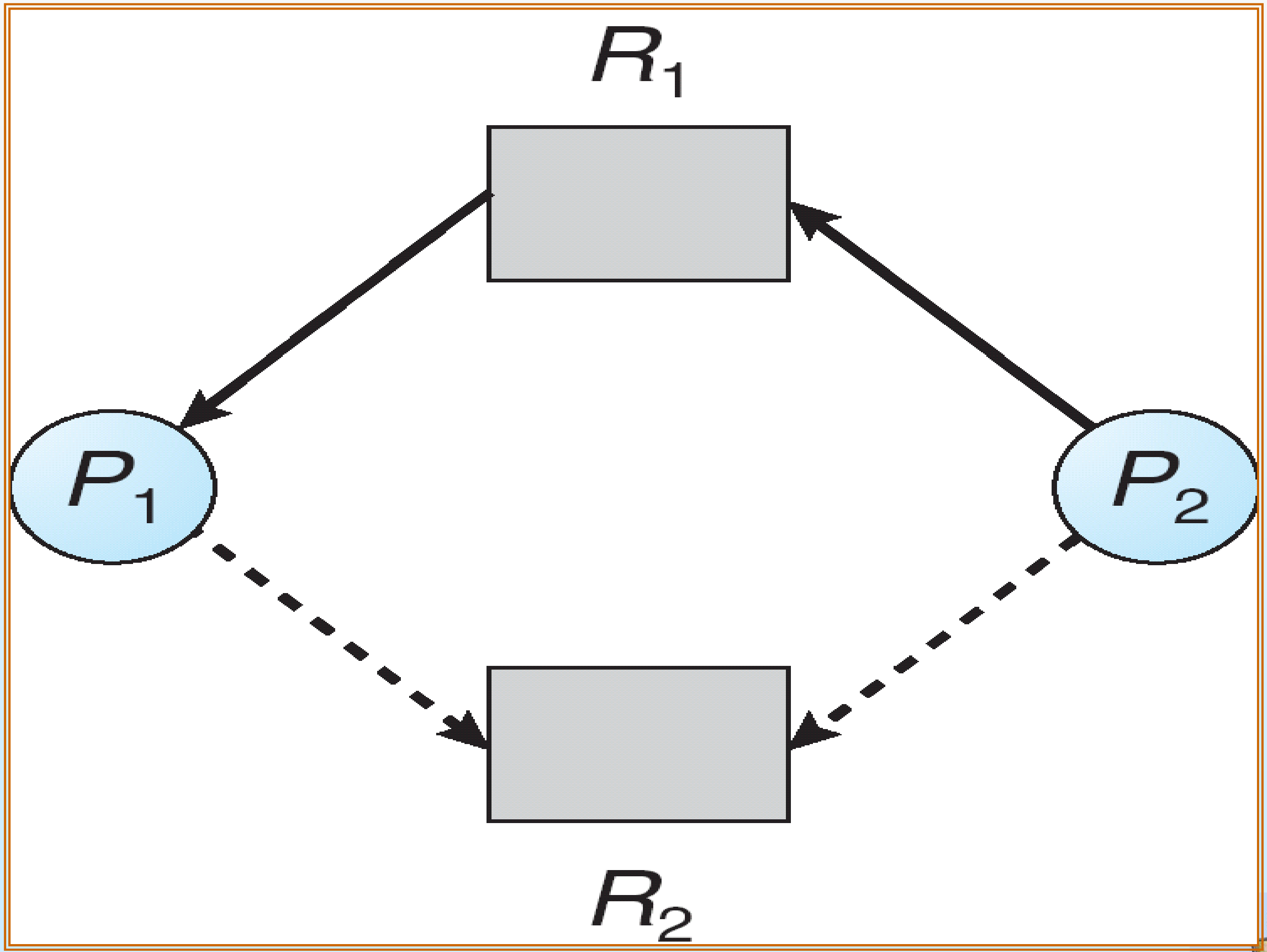
process	Maximum needs	Current needs
P_0	10	5
P_1	4	2
P_2	9	2

- at time t0, the system is in a safe state. The sequence <P1,P0, P2> satisfies the safety condition.
- A system can go from safe to unsafe state.
- At time t1, process p2 requests and is allocated one more tape drive; the system is no longer in a safe state.
- Deadlock results.
- Our mistake was granting the request from process p2 for one more tape drive.
- If we had p2 wait until either of the other processes had finished and released the resources, deadlock is avoided.

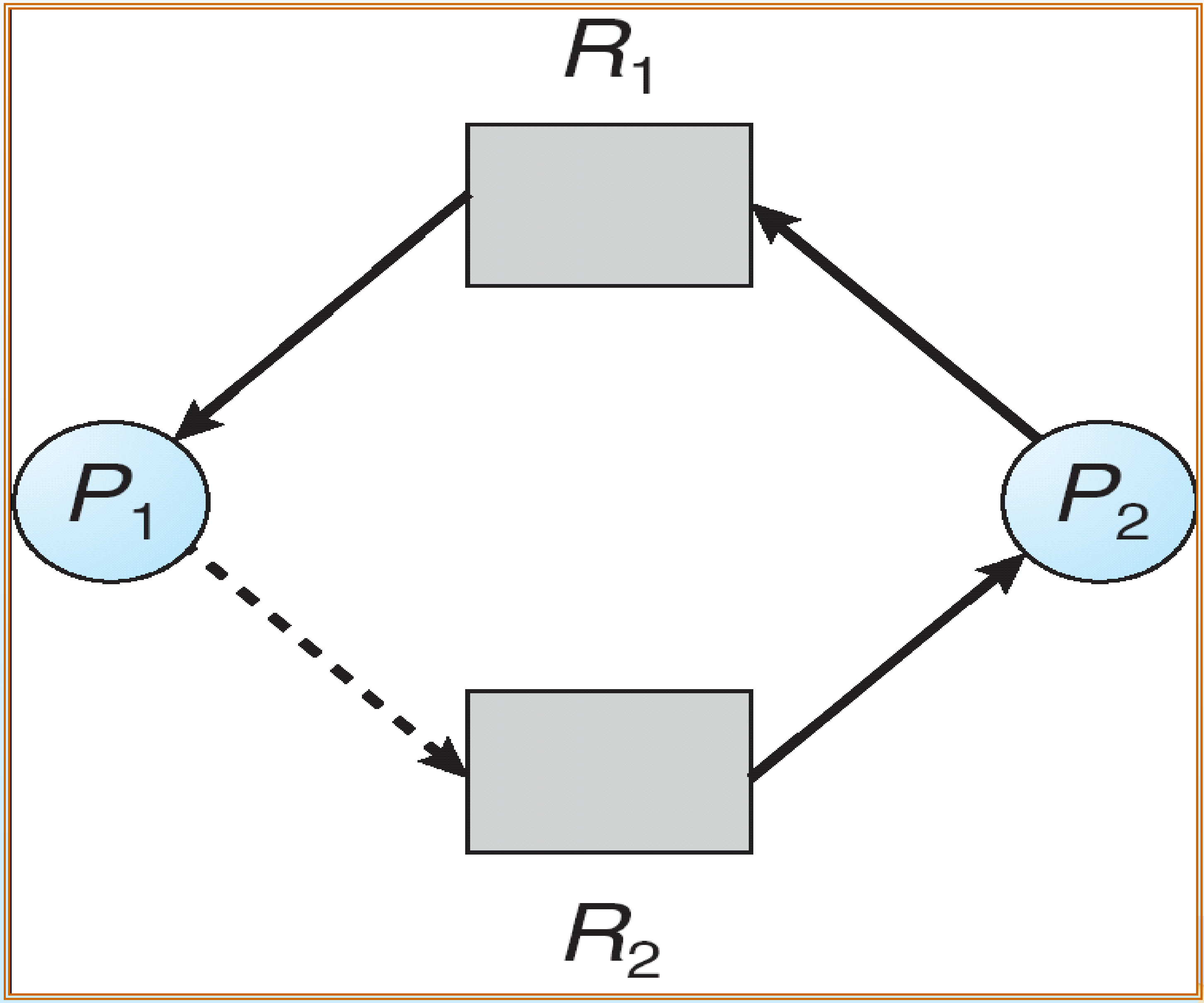
Resource-Allocation Graph Algorithm

- *Applicable when* ***One*** *instance of each resource type are available.*
- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j at some time in future (represented by a dashed line).
- Claim edge converts to request edge when a process requests a resource.
- 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 For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Multiple instances.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- 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.

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

Safety Algorithm

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

   Work = *Available*

   Finish [*i*] = *false* for *i* =0,1,2,3, …, *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 = 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 have allocated the 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 Pi.*
- *If unsafe $\Rightarrow$ Pi 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) (Cont.)

- Check that Request $\le$ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ 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 1	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 <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

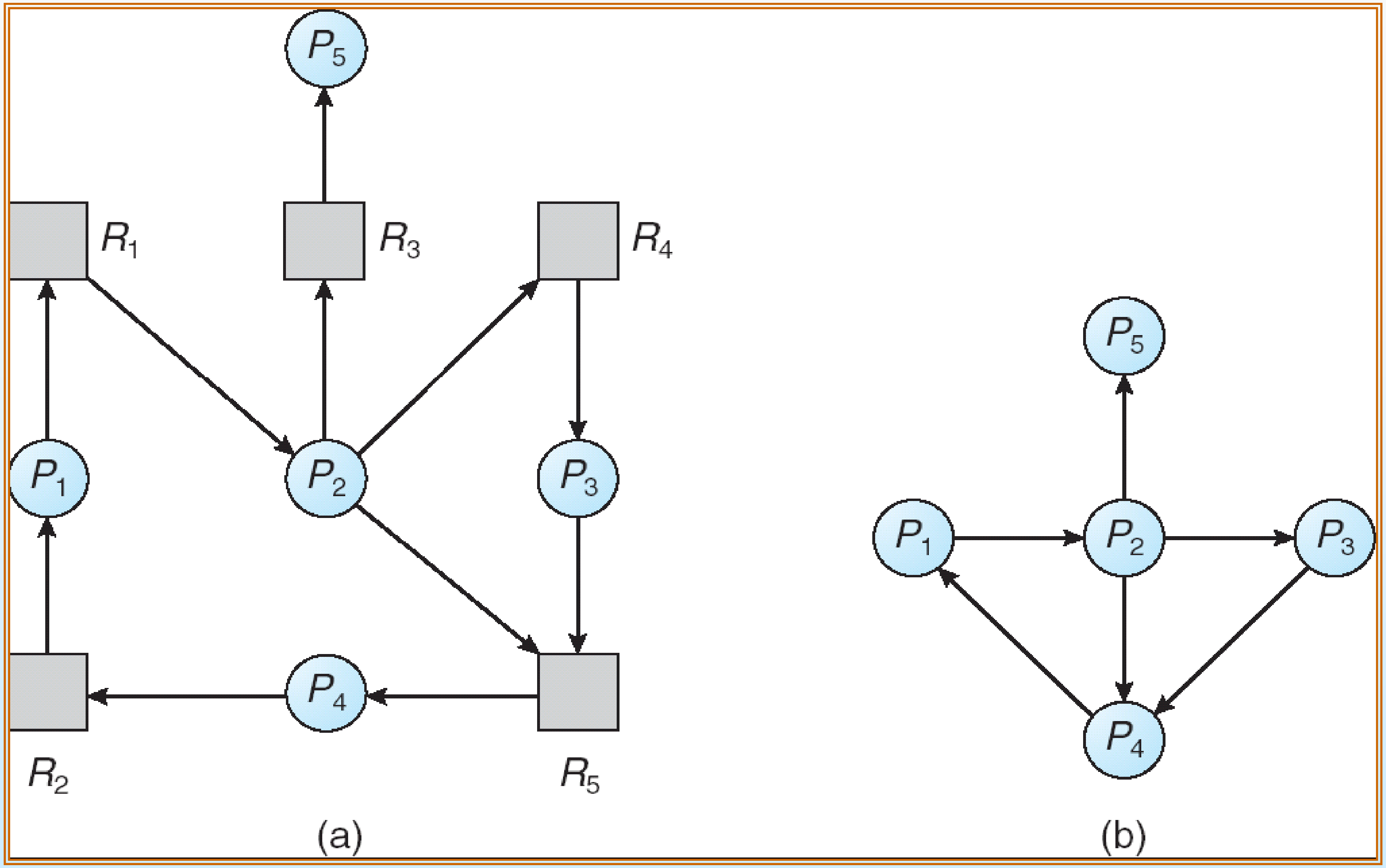
Deadlock Detection

- 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.
- 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 $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.

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

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

   (b) $Request_i \leq Work$

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

Detection Algorithm (Cont.)

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

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

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	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 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.

Program for Bankers Algorithm

- #include<stdio.h>
- #include<conio.h>
- int max[10][10];
- int alloc[10][10];
- int need[10][10];
- int avail[10];
- int n,r;
-
- void inputt()
- {
- int i,j;
- printf("Enter the no of Processes\t");
- scanf("%d",&n);
-

```
printf("Enter the no of resource types\t");
 scanf("%d",&r);
 printf("Enter the Max Matrix\n");
 for(i=0;i<n;i++)
 {
  for(j=0;j<r;j++)
  {
   scanf("%d",&max[i][j]);
  }
 }
 printf("Enter the Allocation Matrix\n");
```

```
for(i=0;i<n;i++)
{
 for(j=0;j<r;j++)
 {
  scanf("%d",&alloc[i][j]);
 }
}
printf("Enter the available Resources\n");
for(j=0;j<r;j++)
{
 scanf("%d",&avail[j]);
}
}
```

```
void show()
{
 int i,j;
 printf("Process\t Allocation\t Max\t Available\t");
 for(i=0;i<n;i++)
 {
  printf("\nP%d\t   ",i);
  for(j=0;j<r;j++)
  {
   printf("%d ",alloc[i][j]);
  }
```

```
 printf("\t");
  for(j=0;j<r;j++)
  {
   printf("%d ",max[i][j]);
  }
  printf("\t");
  if(i==0)
  {
   for(j=0;j<r;j++)
   printf("%d ",avail[j]);
  }
 }
}
```

```
void cal()
{
 int finish[10],temp,need[10][10],count,fcount=0,l;

 int i,j,k;
 for(i=0;i<n;i++)
 {
  finish[i]=0;
 }
 //find need matrix
```

```
for(i=0;i<n;i++)
{
 for(j=0;j<r;j++)
 {
  need[i][j]=max[i][j]-alloc[i][j];
 }
}
printf("\n");
 for(k=0;k<n;k++)
   for(i=0;i<n;i++)
   {
     count=0;
```

```
    for(j=0;j<r;j++)
    {
       if((finish[i]==0)&&(need[i][j]<=avail[j]))
          count++;
    }
    if(count==r)
    {
      for(l=0;l<r;l++)
        avail[l]=avail[l]+alloc[i][l];
      finish[i]=1;
      printf("P%d->",i);
      break;
    }
  }
```

```
for(i=0;i<n;i++)
    if(finish[i])
      fcount++;
 if(fcount==n)
 {
    printf("the system is in safe state");

    /* for(i=1;i<=n;i++)
       printf("%d",pr[i]); */
 }
 else
     printf("the system is not in safe state");
 getch();
}
```

```
void main()
{

 int i,j;
 clrscr();
 printf("Banker's Algorithm\n");
 inputt();
 show();
 cal();
 getch();
}
```

- /* Banker's Algorithm
- Enter the no of Processes 5
- Enter the no of resource types 3
- Enter the Max Matrix
- 7 5 3
- 3 2 2
- 9 0 2
- 2 2 2
- 4 3 3
- Enter the Allocation Matrix
- 0 3 0
- 3 0 2
- 3 0 2
- 2 1 1
- 0 0 2

- Enter the available Resources
- 2 1 0

Process	Allocation	Max	Available
P0	0 3 0	7 5 3	2 1 0
P1	3 0 2	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

- the system is not in safe state
- */

- /*
- Banker's Algorithm
- Enter the no of Processes   5
- Enter the no of resource types 3
- Enter the Max Matrix
- 7 5 3
- 3 2 2
- 9 0 2
- 2 2 2
- 4 3 3

- Enter the Allocation Matrix
- 0 1 0
- 2 0 0
- 3 0 2
- 2 1 1
- 0 0 2
- Enter the available Resources
- 3 3 2

Process	Allocation	Max	Available
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

- P1->P3->P0->P2->P4->the system is in safe state

End of Chapter 7