OPERATING SYSTEMS

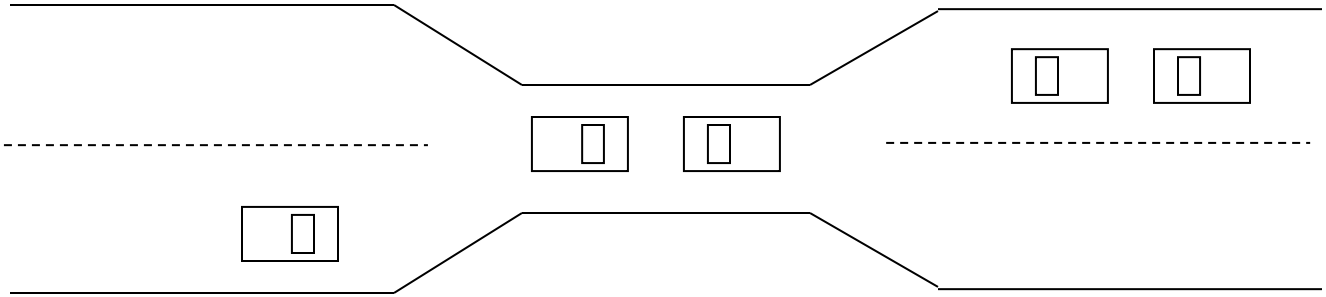
Subject code : CSC 404

Subject In-charge

Nidhi Gaur
Assistant Professor
email: nidhigaur@sfit.ac.in

Module 3: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).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

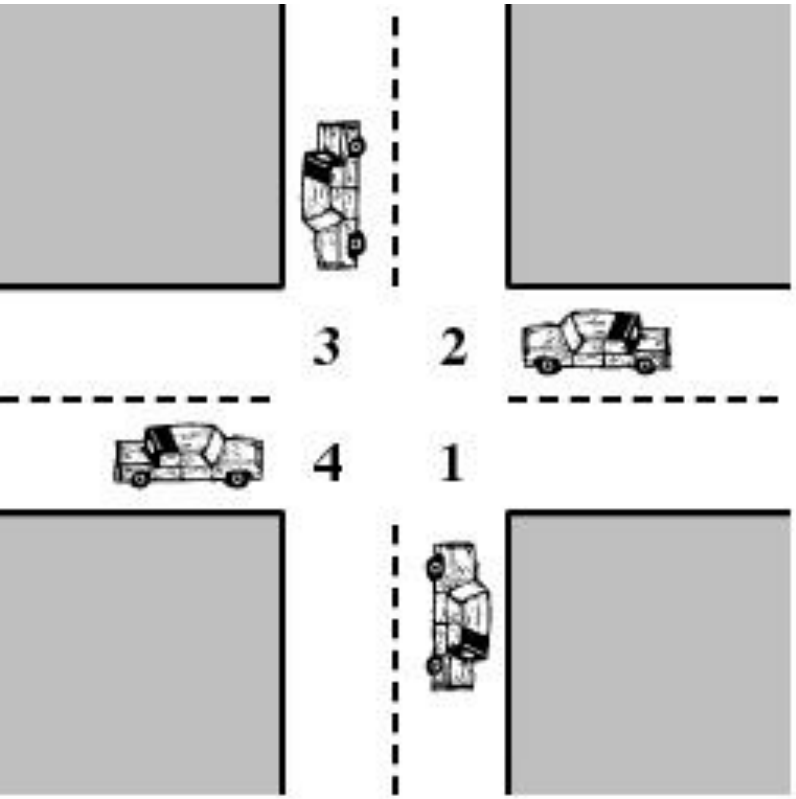
Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes

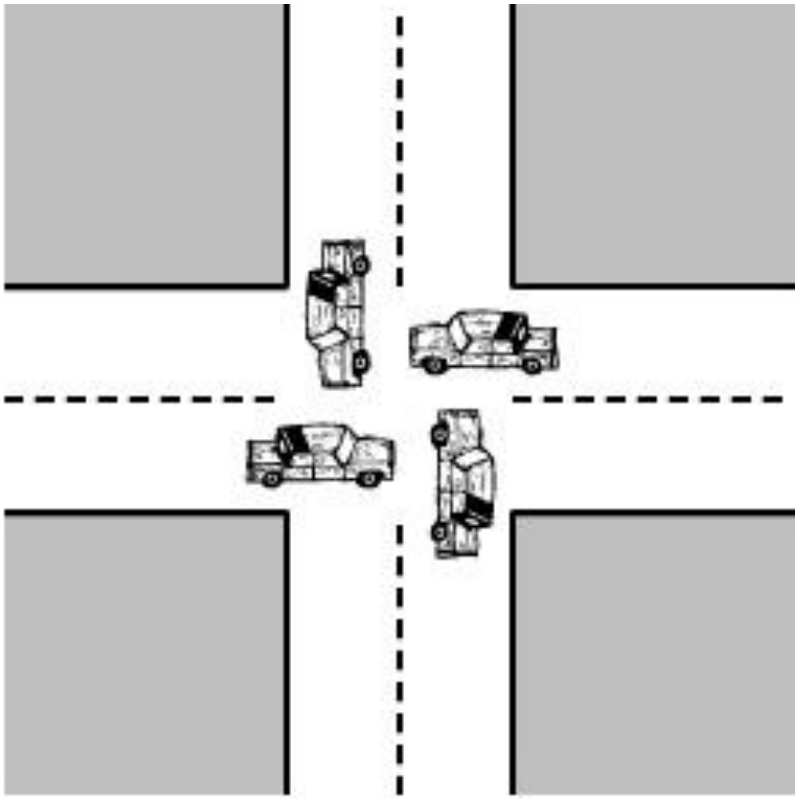
System Model

- Various types of resources in multiprogramming environment.
- 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.
- Each process utilizes a resource as follows:
  - request
  - use
  - release



(a) Deadlock possible



(b) Deadlock

Figure 6.1 Illustration of Deadlock

Deadlock

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

Example

–System has 2 disk drives.

–P_1 and P_2 each hold one disk drive and each needs another one.

Example

–semaphores *A* and *B*, initialized to 1

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

Consumable Resources

- Created (produced) and destroyed (consumed) by a process
- Interrupts, signals, messages, and information in I/O buffers
- May take a rare combination of events to cause deadlock

Reusable Resources

- Used by one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Processors, I/O channels, main and secondary memory, files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

Another Example

- Space is available for allocation of 200K bytes, and the following sequence of events occur
- Deadlock occurs if both processes progress to their second request

P1

```
. . .
Request 80K bytes;
. . .
Request 60K bytes;
```

P2

```
. . .
Request 70K bytes;
. . .
Request 80K bytes;
```

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.

Deadlock Characterization

- 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 $P_2, \ldots, P_{n-1}$ 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.

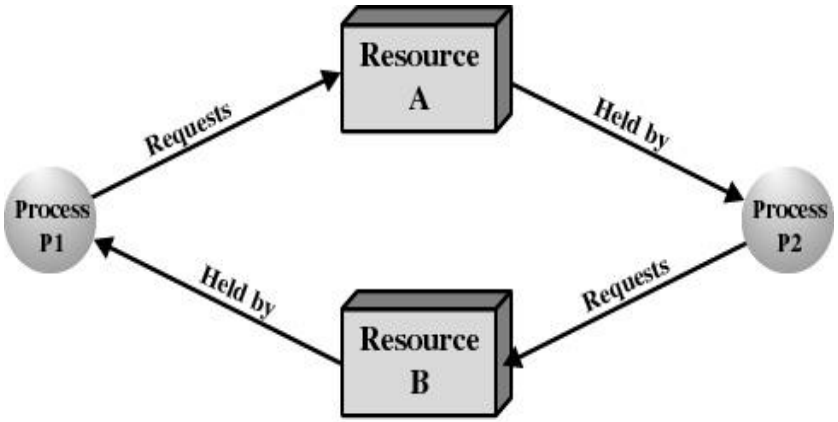


Figure 6.5 Circular Wait

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.

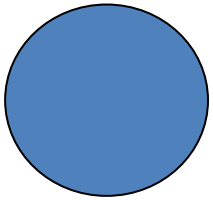
– $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

Request edge – directed edge $P_1 \rightarrow R_j$

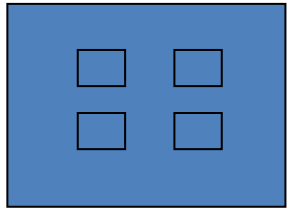
Assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

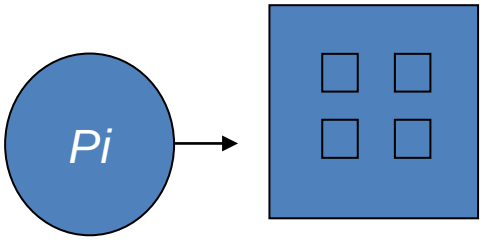
Process



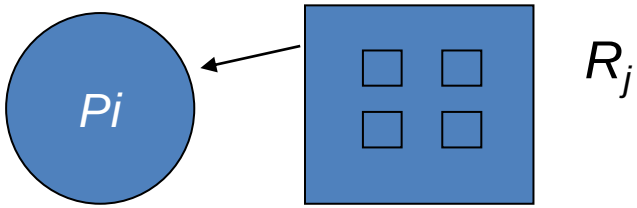
Resource Type with 4 instances



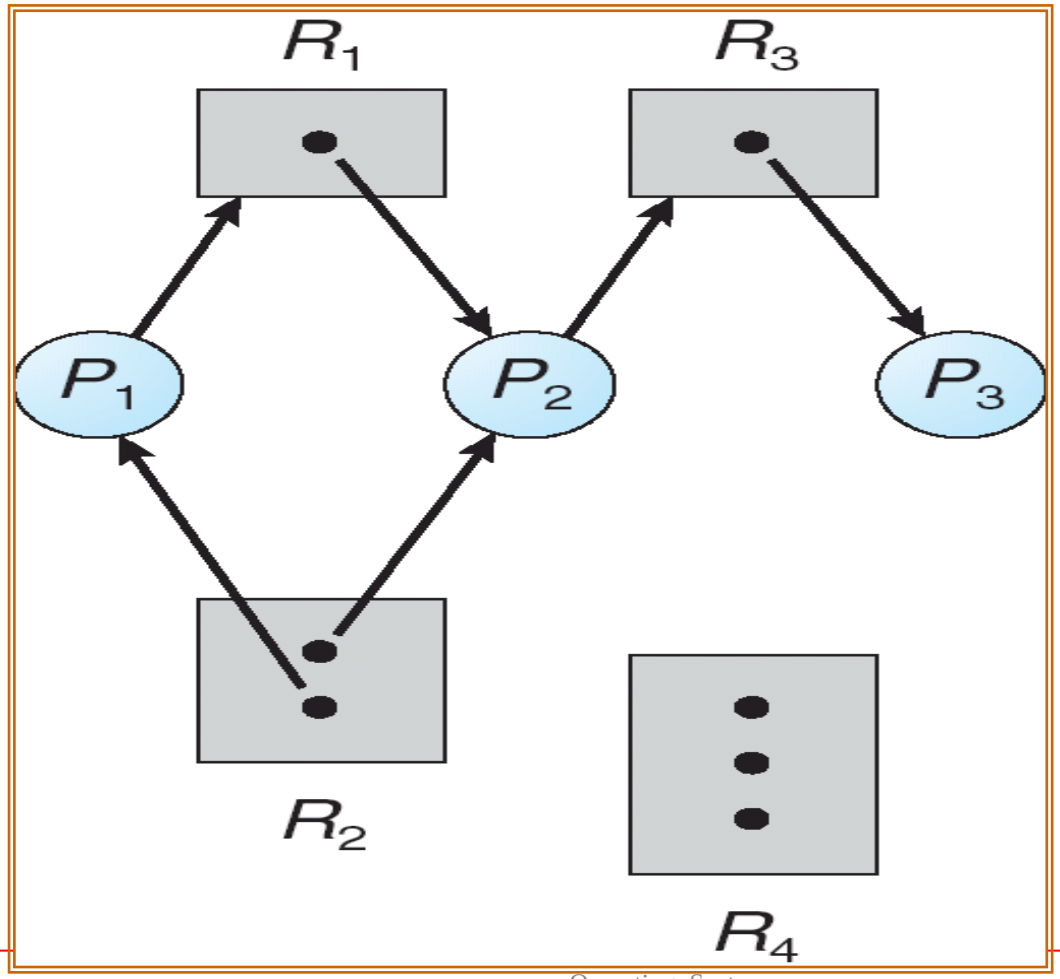
P_i requests instance of R_j



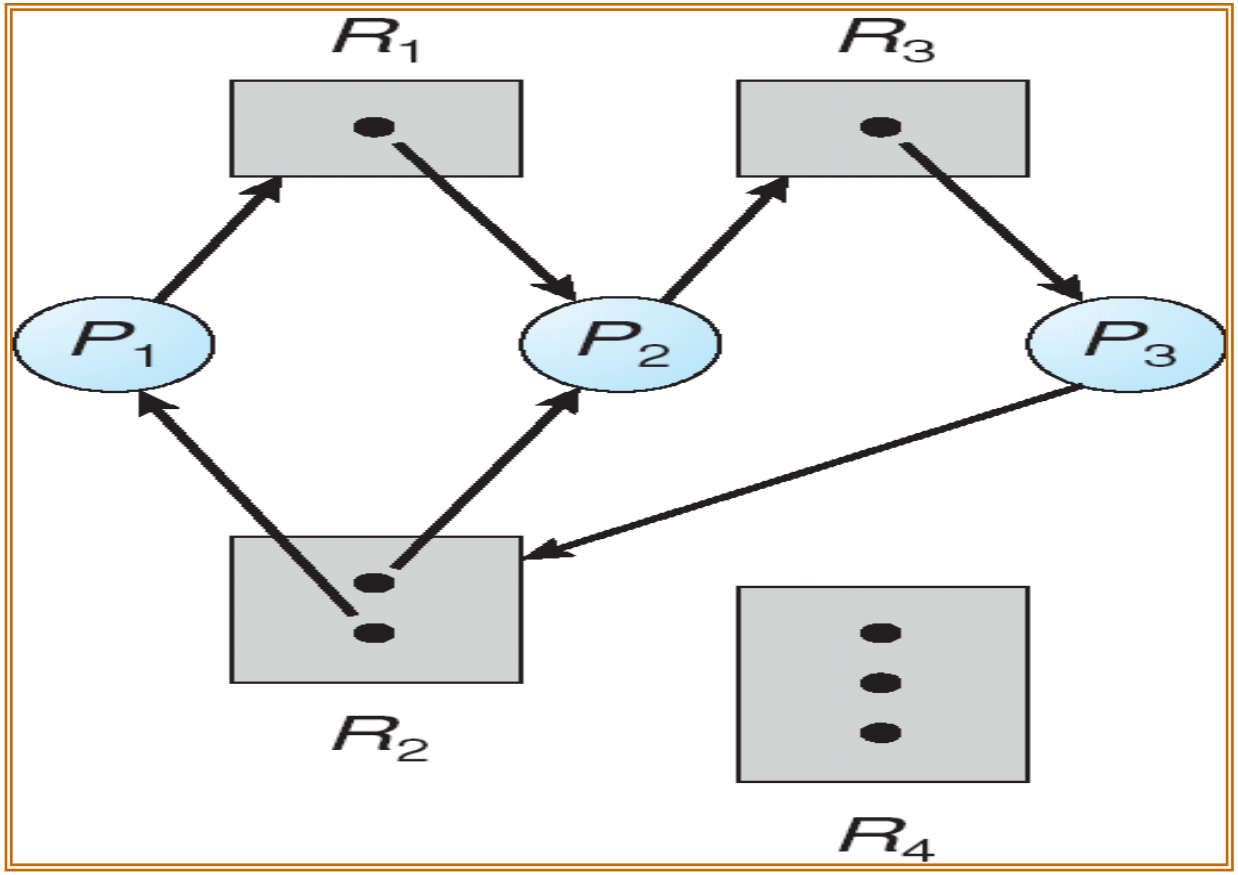
P_i is holding an instance of R_j



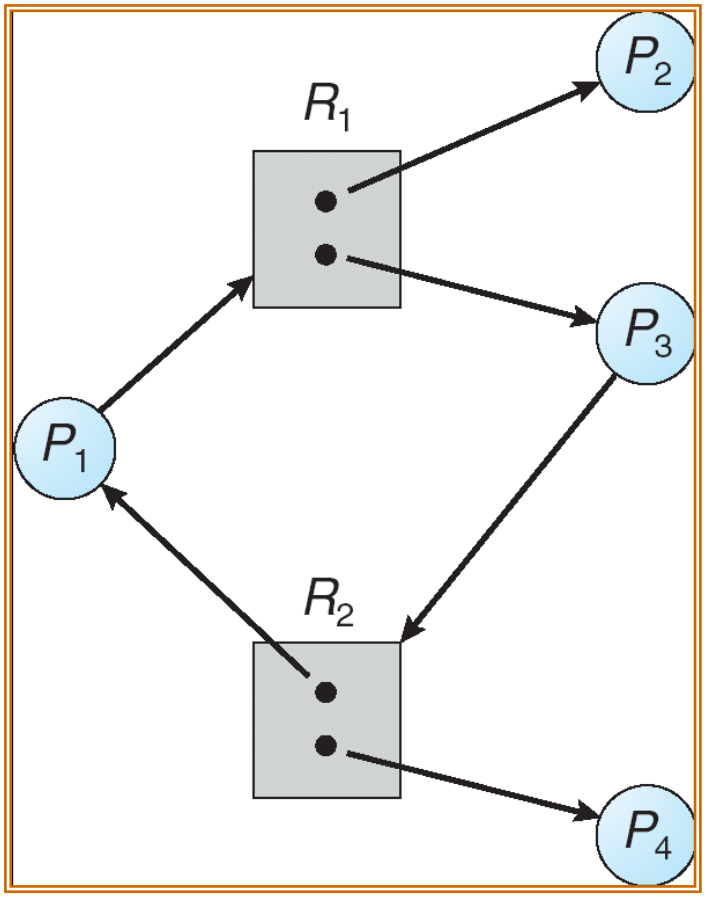
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

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

Deadlock Prevention

Restrain the ways request can be made.

- Mutual Exclusion – must hold for non sharable resources.
- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.

Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.

Deadlock Prevention (Cont.)

- No Preemption –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Circular wait
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process request

Deadlock avoidance

- What is the difference between deadlock prevention and deadlock avoidance?

Deadlock Avoidance

- Deadlock avoidance is concerned with starting with an environment where deadlock is theoretically possible, but by some algo in the OS , it is ensured, before allocating any resource that after allocating it, a deadlock can be avoided.
- Deadlock prevention was concerned with imposing certain restrictions on the environment or processes so that deadlocks can never occur.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

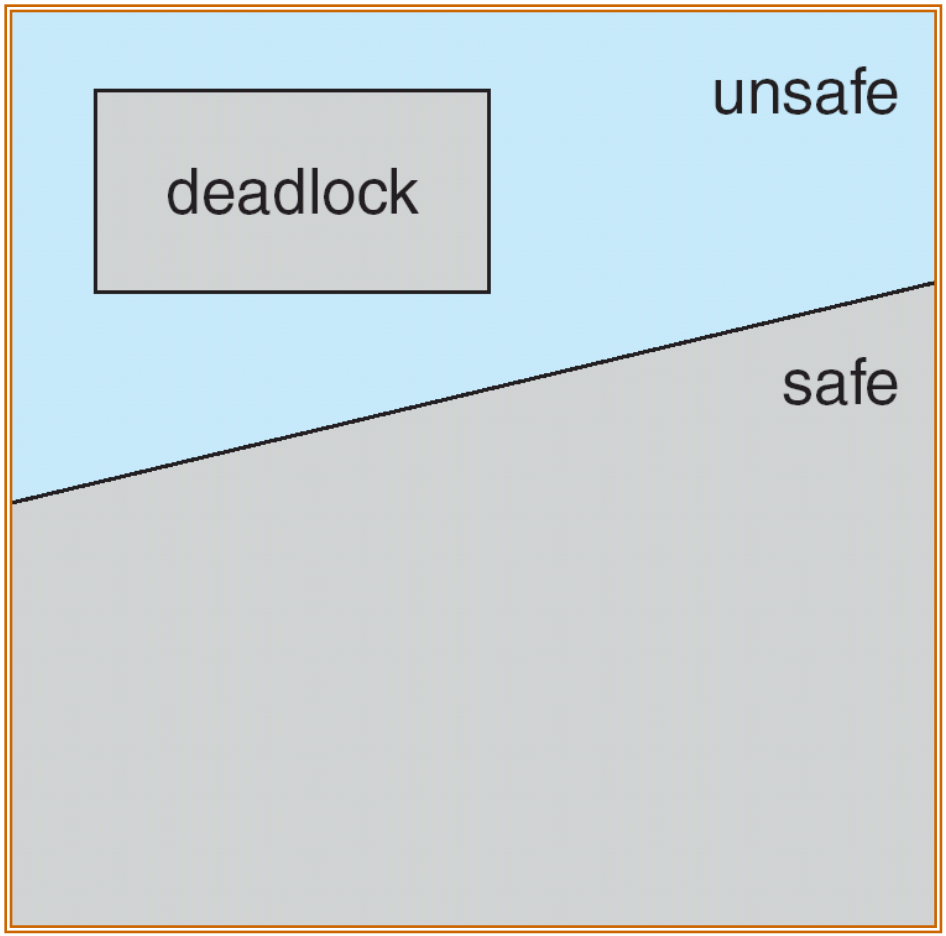
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.

That is:

–If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.

- State of the system is the current allocation of resources to process
- Safe state is where there is at least one sequence that does not result in deadlock
- Unsafe state is a state that is not safe, can lead to deadlock.

Safe, Unsafe , Deadlock State



Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

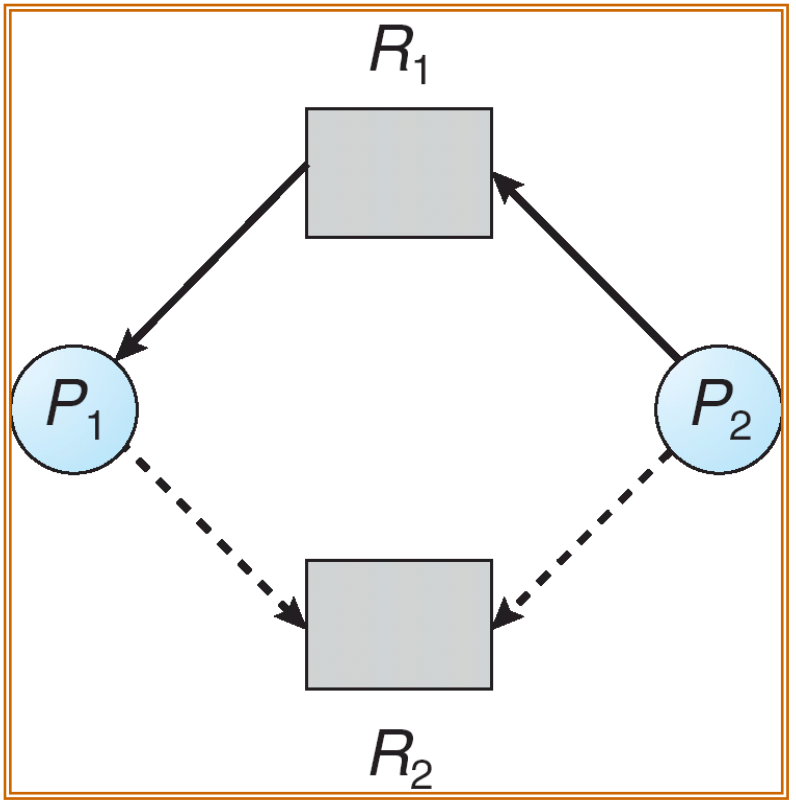
Avoidance algorithms

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

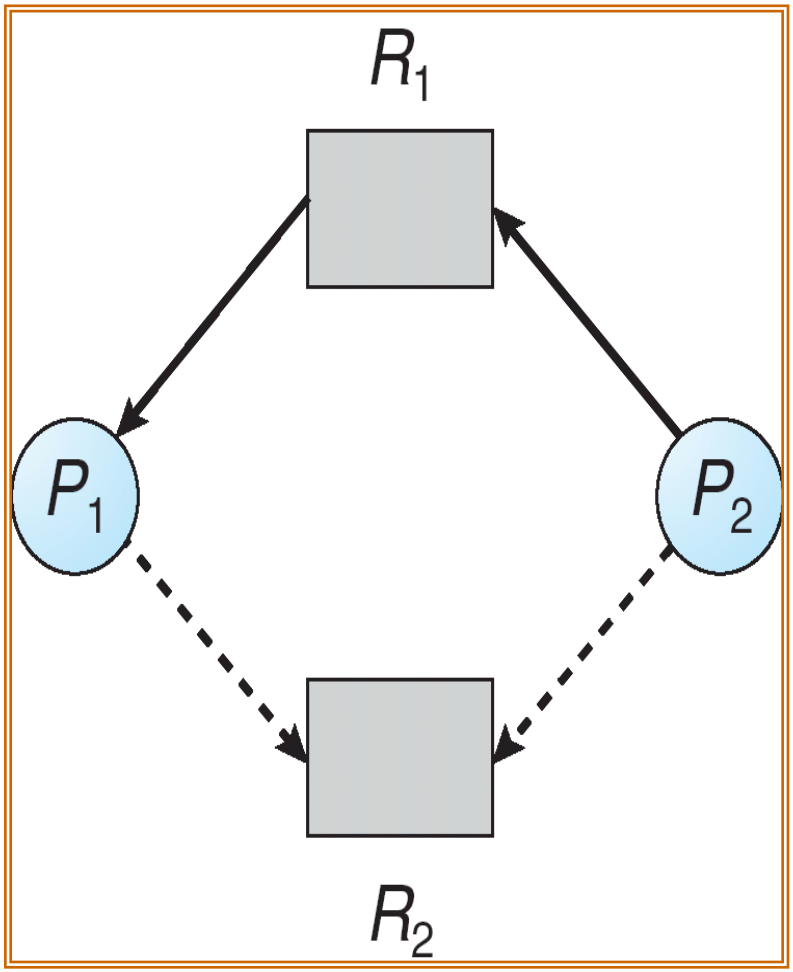
- Multiple instances of a resource type. Use the banker's algorithm

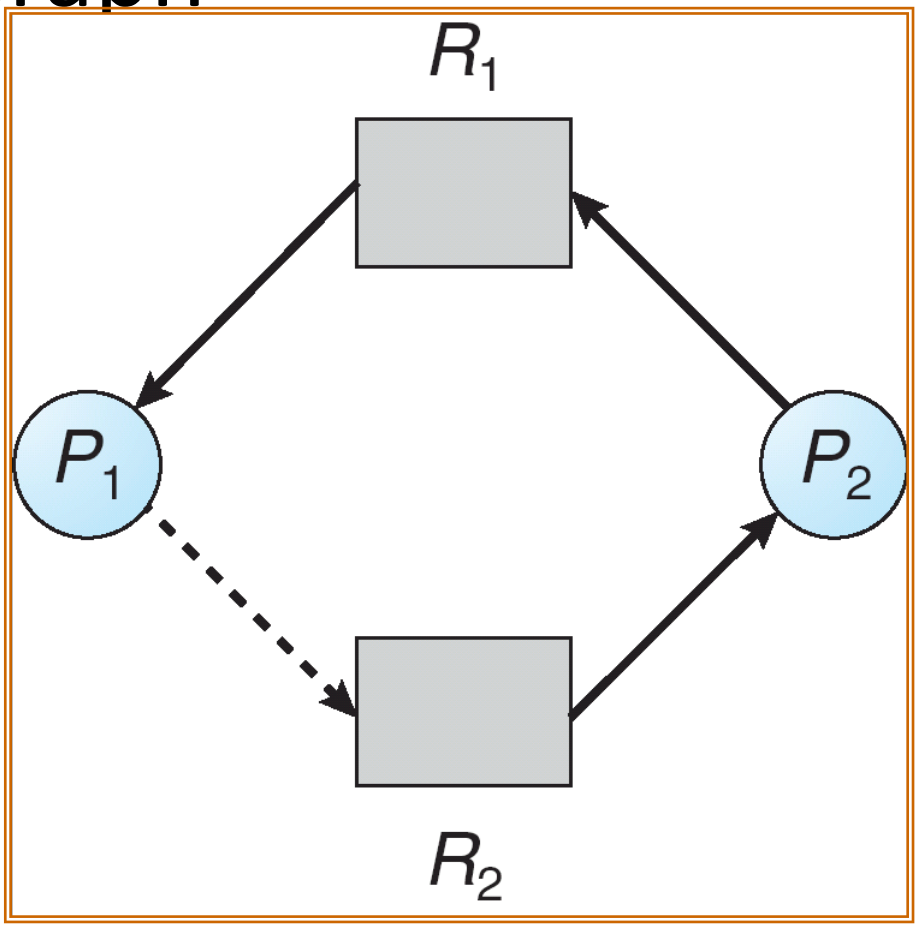
Resource-Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converts 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.



Unsafe State In Resource-Allocation Graph





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

Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

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

Available: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available.

Max: n x m matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.

Allocation: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j.

Need: *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task.

***Need* [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*].**

Safety Algorithm

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

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1.

2. Find an *i* such that both:

(a) *Finish* [*i*] = *false* *// process i not yet done*

(b) $Need_i \leq Work$ *//its need can be satisfied*

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

3. $Work = Work + Allocation_i$ *// run it and reclaim*
   Finish[*i*] = *true* *// process i completes*
   go to step 2.
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Example

- Consider a system with 12 magnetic tape drives. Snapshot at time T0 is:

Process	Maximum needs	Allocated
P0	10	5
P1	4	2
P2	9	2

Available
3

Find need matrix and safe sequence.

NEED Matrix

Process	Maximum needs	Allocated	Need = Max-Allocation
P0	10	5	5
P1	4	2	2
P2	9	2	7

Example

- Consider a system with 12 magnetic tape drives. Snapshot at time T0 is:

Process	Maximum needs	Allocated
P0	10	5
P1	4	2
P2	9	2

Available/ work
3

Need = Max-Allocation
5
2
7

1) Let Work vector be length m
Finish be vector of length n
Initialize Work = Available
Finish[i] = false, for i = 0,1,2,.... ,n-1

2) Find an index i such that

- Finish[i] == false
- Need i <= work

If no such i exists , go to step 4.

3) Work = work + Allocation
Finish[i] = true
Go to step 2.

4) If Finish[i] == true for all i, then system is in safe state.

1 P0 Need< = Work NO
2 P1 Need<= Work YES
Work = Work+ Allocation
= 3+2 =5
3.P2 Need <= Work NO
4. P0 Need<= Work YES
Work = Work + Allocation
= 5+5=10
5. P2 Need<= Work YES
Work = Work + Allocation
=10+2=12

Example

- Safe sequence < P1 P0 P2>

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	

Find the need matrix and safe sequence.

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

Example of Banker's Algorithm

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
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

Total resources = Allocation + Available

	A	B	C
=	10	5	7

Example of Banker's algo(safety)

1. P0 : Need <= Work

   7 4 3 <= 3 3 2 NO

2. P1 : Need <= Work

   1 2 2 <= 3 3 2 YES

   Work = Work + allocation

   3 3 2 + 2 0 0

   = 5 3 2

3 P2: Need <= Work

   6 0 0 <= 5 3 2 NO

3 P3: Need <= Work YES

   0 1 1 <= 5 3 2

   5 3 2+ 2 1 1 = 7 4 3

5. P4: Need<= Work YES

   431 <= 7 4 3

   Work = Work + allocation

   7 4 3 + 0 0 2 = 7 4 5

6. P0: Need <= Work

   7 4 3 <= 7 4 5 YES

   Work = 7 4 5 + 0 1 0

   = 7 5 5

7. P2 : Need <= Work YES

   Work = 7 5 5 + 3 0 2

   = 10 5 7

Available/ WORK

A	B	C
3	3	2

	Need	
	A	***B C***
P0	**7**	**4 3**
P1	**1**	**2 2**
P2	**6**	**0 0**
P3	**0**	**1 1**
P4	**4**	**3 1**

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_0, P2> satisfies safety criteria.

Example

	MAX			ALLOCATION		
PROCESS	**R1**	**R2**	**R3**	**R1**	**R2**	**R3**
P1	5	6	3	2	1	0
P2	8	5	6	3	2	3
P3	4	8	2	3	0	2
P4	7	4	3	3	2	0
P5	4	3	3	1	0	1

TOTAL Res		
R1	**R2**	**R3**
15	8	8

Q1. Find out available resources
Q2. Calculate need matrix.
Q2. Determine whether system is in safe state or no?

example

- Available = total – allocation

  = 3 3 2

Need matrix

R1	R2	R3
3	5	3
5	3	3
1	9	0
4	2	3
3	3	2

SAFE SEQUENCE:
< P5 P4 P1 P2 P3 >

Resource-Request Algorithm for Process P_j

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:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *If safe $\Rightarrow$ the resources are allocated to Pi.*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example: Resource Request

Check that Request $\leq$ Available ; that is say P1 requests

$(1,0,2) \leq (3,3,2) \Rightarrow$ true.

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

What about request (3, 3,0) by P1 ?

Deadlock Detection

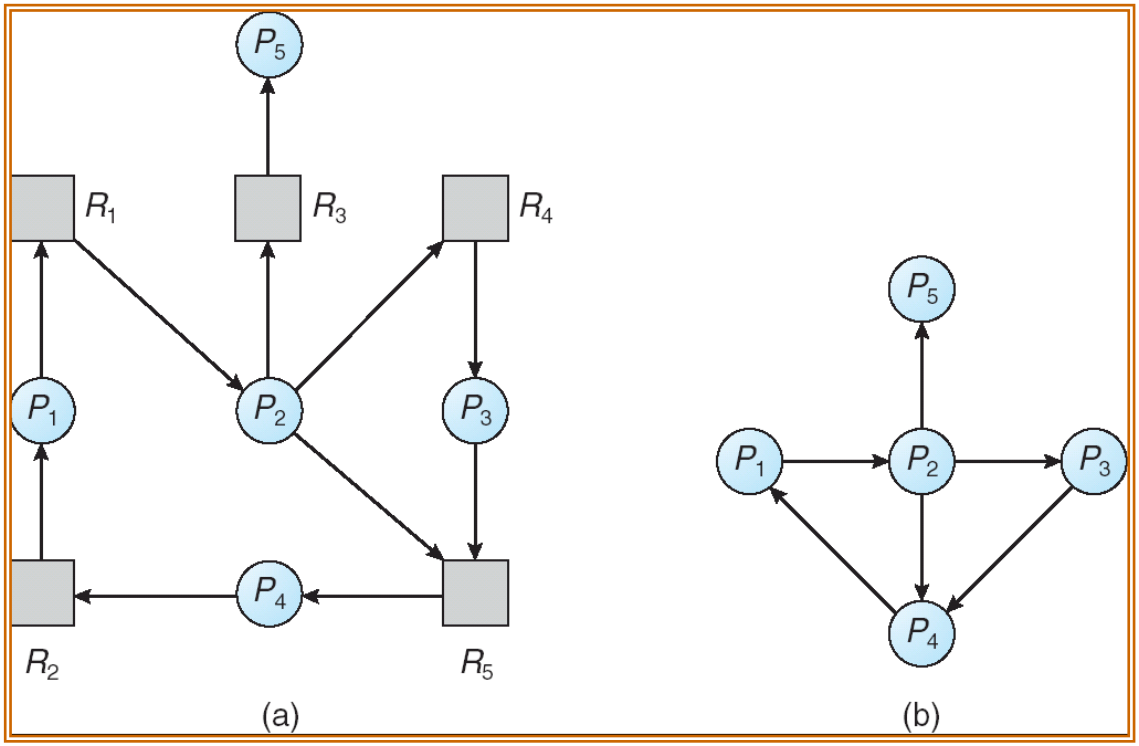
In deadlock detection, requested resources are granted to processes whenever possible. Periodically the OS performs an algorithm that allows it to detect the circular wait condition.

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

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

Available: A vector of length *m* indicates the number of available resources of each type.

Allocation: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

Request: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i,j*] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Deadlock Detection

ALGORITHM

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector W to equal the available vector.
3. Find an index i such that process i is currently unmarked and the ith row of Q(request matrix) is less than or equal to W i.e. $Q_{ik} <= W_k$. If no such row is found terminate.
4. If such a row is found, mark process i and add the row of allocation matrix to W i.e. set $W_k = W_k + A_{ik}$

return to step 3.

Deadlock Detection

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request Matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation Matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource Vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available Vector

ALGORITHM

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector W to equal the available vector.
3. Find an index i such that process i is currently unmarked and the ith row of Q(request matrix) is less than or equal to W i.e. $Q_{ik} <= W_k$. If no such row is found terminate.
4. If such a row is found, mark process i and add the row of allocation matrix to W i.e. set $W_k = W_k + A_{ik}$ return to step 3.

Example deadlock detection

Deadlock Detection

- Mark P4 as P4 has no allocated resources.
- Set W = (00001)
- Requirement of Process P3 is <= W, so mark P3 and set W = W+(00010)
- Hence W = (00011)
- No other unmarked process has a row in Q that is less than or equal to W.

Therefore terminate the algorithm.

P1 and P2 unmarked indicates these are deadlocked.

Strategies once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Selection Criteria Deadlocked Processes

- Least amount of processor time consumed so far
- Least number of lines of output produced so far
- Most estimated time remaining
- Least total resources allocated so far
- Lowest priority

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.

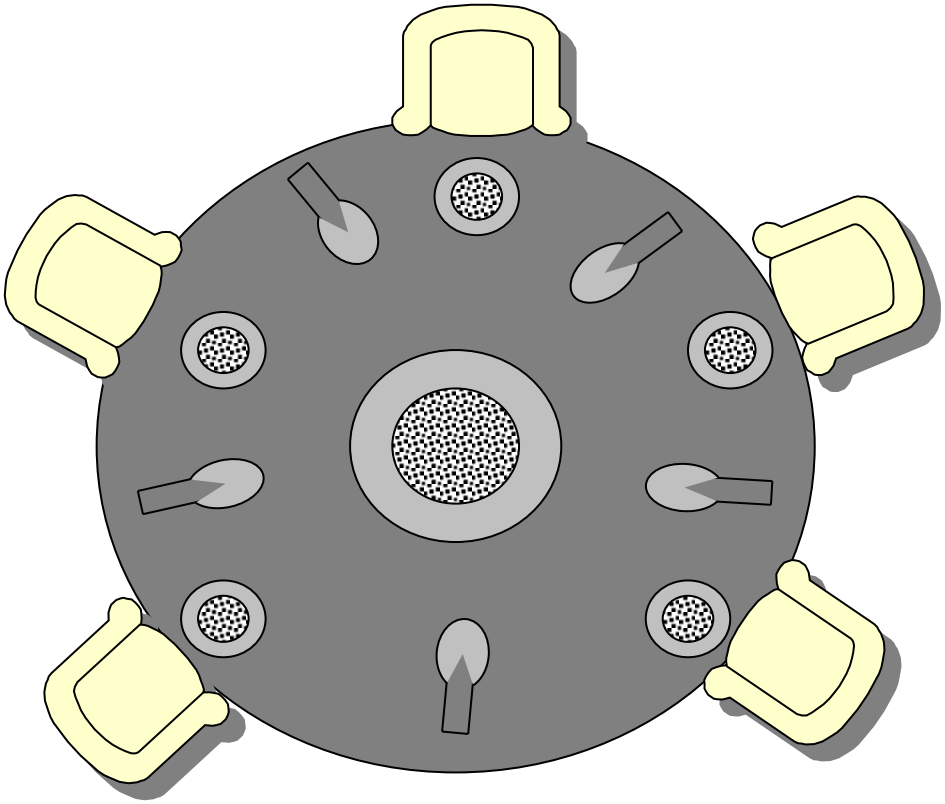
In which order should we choose to abort?

- 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.

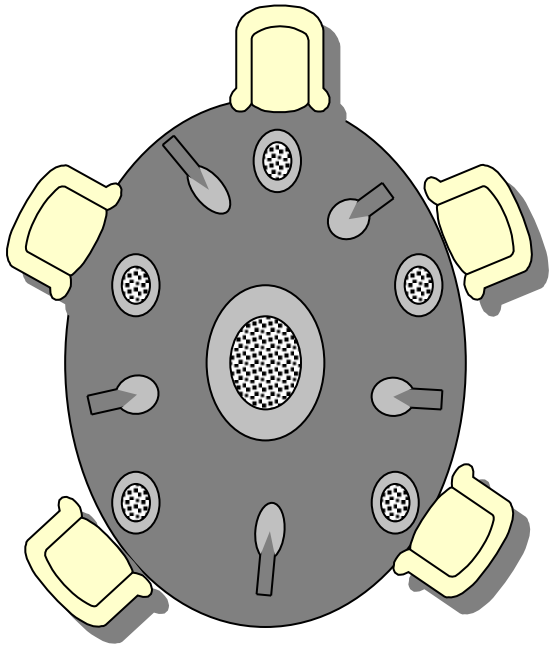
Dining Philosophers Problem



Dining Philosophers Problem

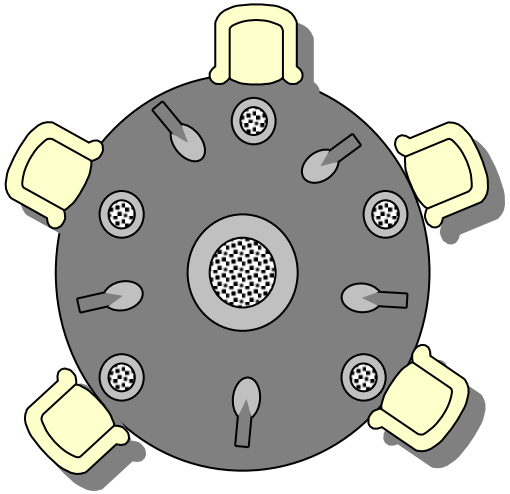
Consider five philosophers who spend their lives thinking and eating.

- Philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of noodles/rice, and the table is laid with 5 plates and 5 chopsticks/spoons
- When philosopher is hungry he picks up two chopsticks that are closest, a philosopher may pick only one chopstick at a time.



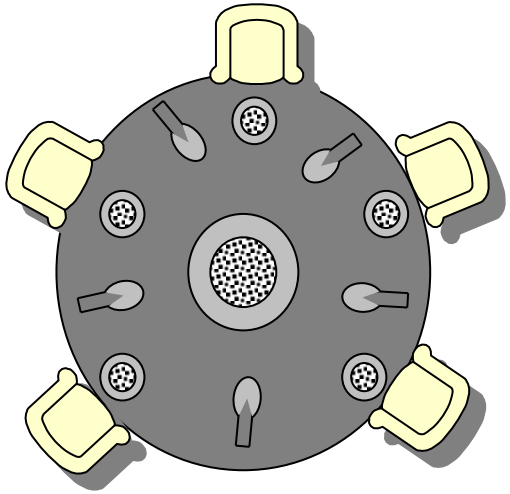
Dining Philosophers Problem

- When a hungry philosopher has both the chopsticks/spoons at the same time, he eats without releasing his chopsticks/spoons.
- When he is finished eating, he puts down both of his chopsticks/spoons and starts thinking again.



Dining Philosophers Problem

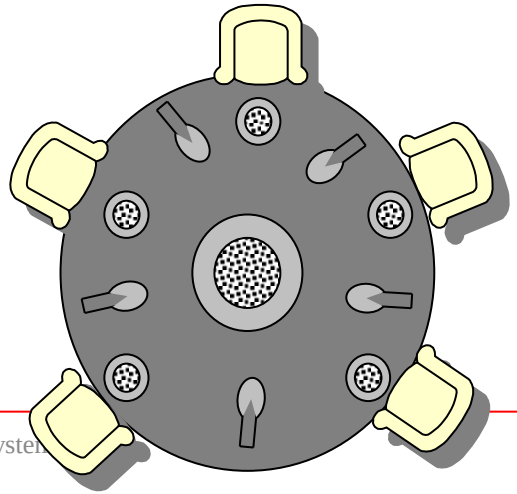
- This is a classic problem to understand the synchronization among processes in an environment where there is scarcity of resources.
- To eat they must use two chopsticks or two spoons but there is scarcity of chopsticks or spoons.



Dining Philosophers Problem

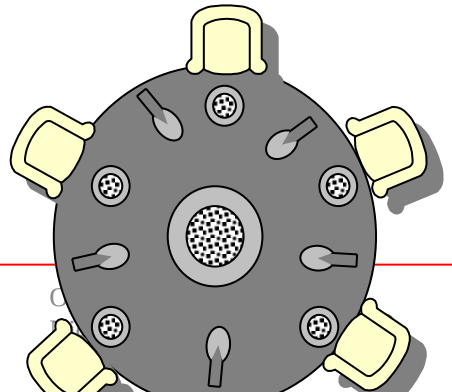
- The two conditions that need to be noted are:

1. There are only five spoons available, each placed between two philosophers
2. Philosophers can start eating only when they have two spoons. They can't eat with one spoon. It means that every philosopher takes one spoon from left and another from right and start eating. After eating put down both the spoons.



Dining Philosophers Problem

- The problem here is obvious that all the philosophers cannot eat together as there are only five spoons whereas there is demand for 10.
- The solution should be such that if one philosopher is eating with one spoon, the adjacent philosopher should wait even if he is able to get one spoon.
- If all philosophers pick up one spoon at the same time, then this leads to deadlock, since every philosopher is waiting for other to release and no one is releasing.
- So a philosopher should not hold a spoon if he is not able to get the other one.

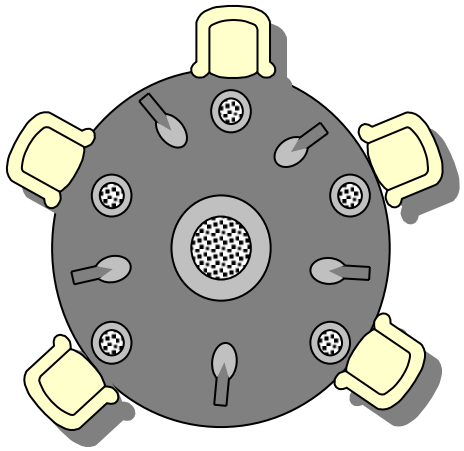


Dining Philosophers Problem

- One simple solution is to represent each spoon by a semaphore, a spoon is mutually exclusive so we consider semaphore to protect each spoon.
- A philosopher tries to grab the spoon by executing a wait operation on that semaphore; he releases his spoon by executing the signal on the appropriate semaphore.
- We take an array of five elements

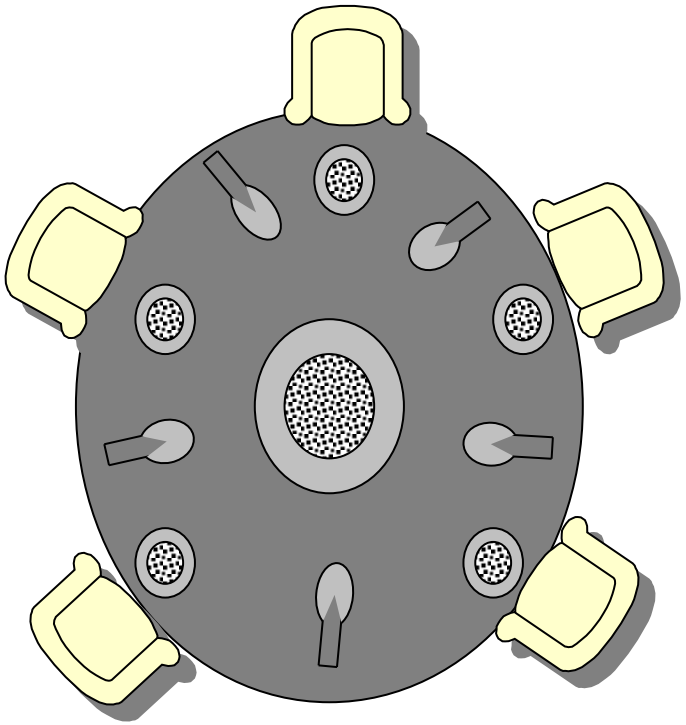
Shared data: semaphore spoon[5];

Where all elements of spoon are initialized to 1.

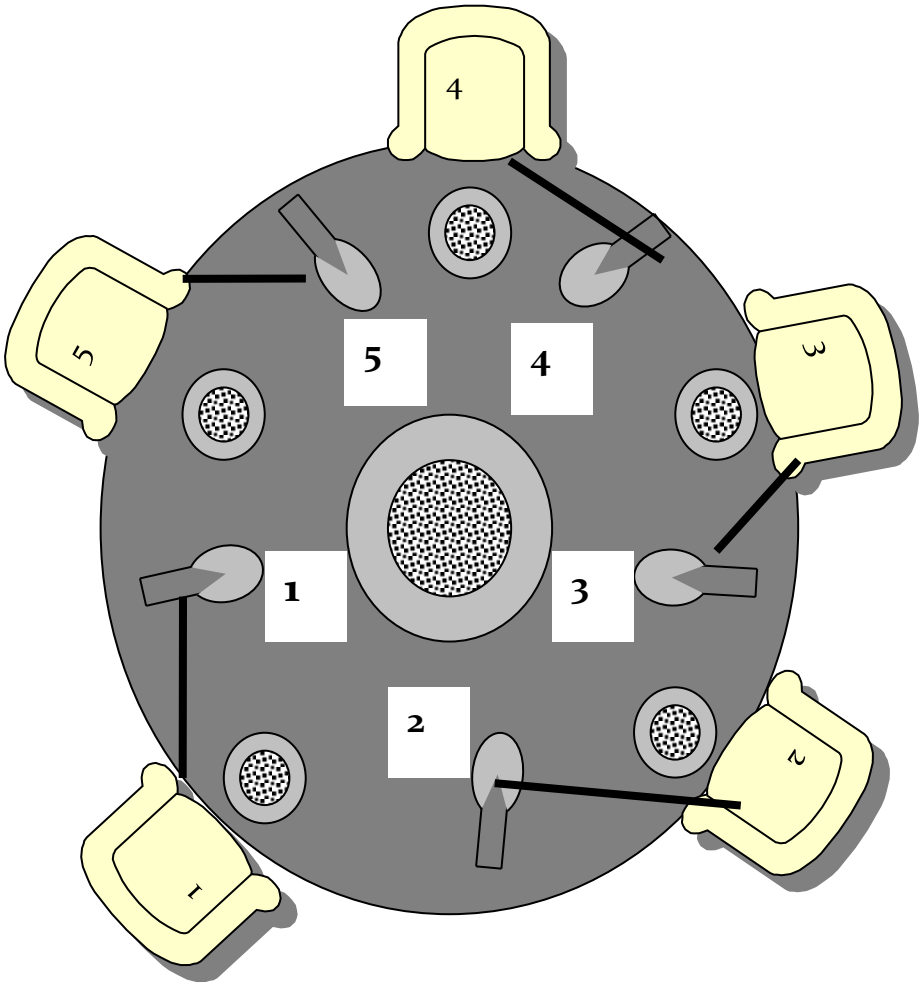


Dining Philosophers Problem

```
semaphore spoon[5]=[1];
int i;
void philosopher (int i)
{      while (true)
{    think();
wait(spoon[i]);
wait(spoon[(i+1) mod 5];
eat();
signal(spoon[(i+1)mod 5];
signal(spoon[i]);
}
}
```



What if all philosophers put wait on their left spoon first then can enter into deadlock



Dining Philosophers Problem

Solution to the problem that is free of deadlocks:-

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up spoons only if both spoons are available(must pick them in critical section.)