OPERATING SYSTEM: CSET209

Question: Consider a system with ***n*** process and 6 tape drives. If each process requires 2 tape-drives to complete their execution then what is the maximum value of ***n*** which ensures deadlock free execution.

Question: Consider a system with ***3*** process and peak demand of each process is 5, 9, 13 respectively. what is the minimum resources required to ensures deadlock free execution.

Question: Consider a system with ***3*** process. If each process requires 2 tape-drives to complete their execution then what is the minimum value of tape-drivers which ensures deadlock free execution.

Question: Consider a system with ***n*** process and 6 tape drives. If each process requires 3 tape-drives to complete their execution then what is the maximum value of ***n*** which ensures deadlock free execution.

DEADLOCK HANDLING STRATEGIES

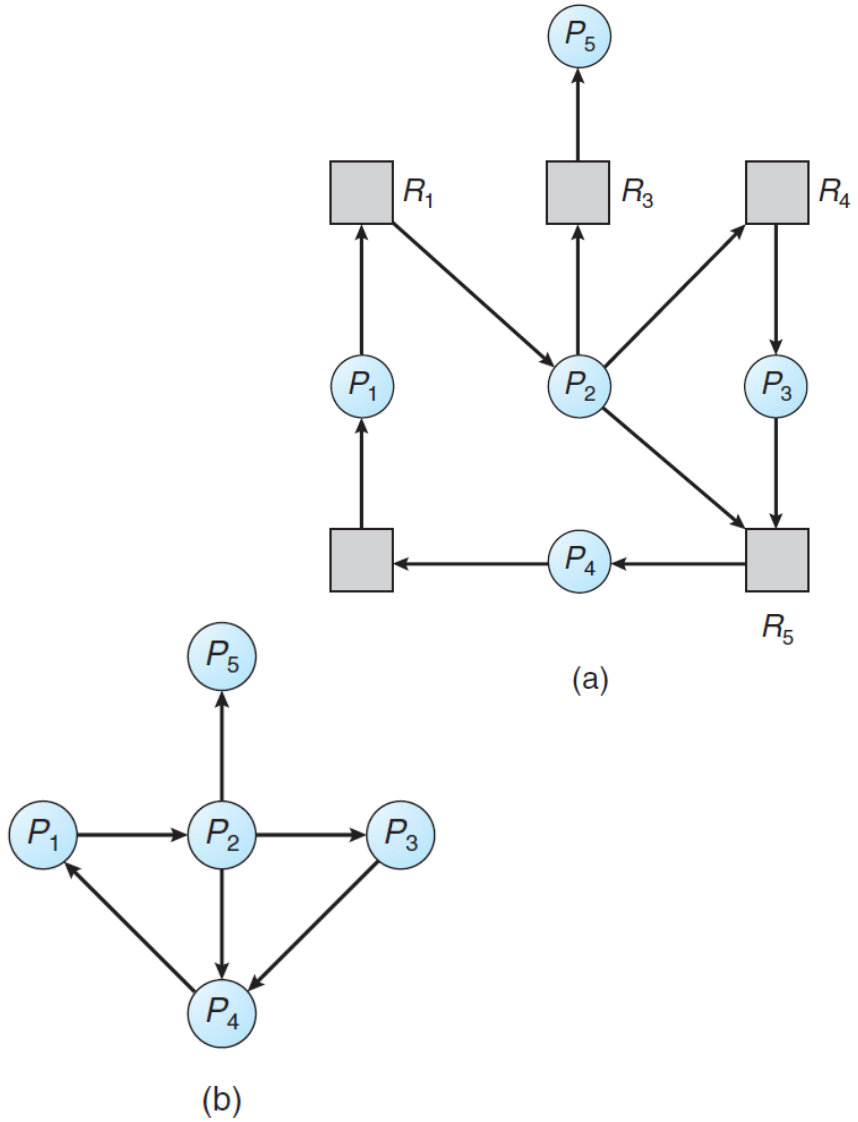
- Detection and Recovery
  - Provides algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock
- Avoidance
  - Given some additional knowledge, the operating system can decide for each request whether or not the process should wait. Thus, ensure that **deadlocks never occur.**
- Prevention
  - Provides a set of methods to ensure that at least one of the necessary conditions cannot hold. Thus, ensure that **deadlocks never occur**.

DEADLOCK DETECTION: SINGLE INSTANCE

- 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.
- Figure (b) shows a wait for graph corresponding to resource allocation graph (Figure (a)).



DEADLOCK DETECTION: MULTIPLE INSTANCES

- Five processes ***P0*** through ***P4***;

  Three resource types

  A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time ***T0***:

	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	

- System is **NOT** in deadlocked state. Sequence <***P0, P2, P3, P1, P4***> will satisfy the requirements for all.

DEADLOCK DETECTION: MULTIPLE INSTANCES

- Suppose, ***P*2** requests an additional instance of type ***C***

	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

- **State of system?**
  - Deadlock exists, consisting of processes ***P*1, *P*2, *P*3**, and ***P*4**

DETECTION ALGORITHM: MULTIPLE INSTANCES

Data Structures

- **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: MULTIPLE INSTANCES

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

   Initialize:

   (a) ***Work = Available***

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

   Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

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

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

   (b) $\boldsymbol{Request_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$

   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 $\boldsymbol{P_i}$ is deadlocked

Algorithm requires an order of O(*m* x *n*2) operations to detect whether the system is in deadlocked state

DETECTION ALGORITHM USAGE

- When should invoke the detection algorithm? The answer is depends on two questions
  - 1. how often is deadlock likely to occure?
  - 2. How many process will be affected by deadlock when it happens?

DEADLOCK RECOVERY
(PROCESS TERMINATION)

What should the OS do when it detects a deadlock?

1. Raise an alarm to tell the users and administrators and let them deal with the deadlock manually.
2. let the system recover from the deadlock automatically.
   - **Abort all deadlocked processes**. will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time,
   - **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
     - Priority
     - Computation time already taken
     - Resources required to be completed
     - Interactive or not

DEADLOCK RECOVERY
(RESOURCE PRE-EMPTION)

Successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. Issues involved in this strategy are as follows.

- **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

DEADLOCK PREVENTION

- The deadlock has the following characteristics:
  - Mutual Exclusion
  - Hold and Wait
  - No preemption
  - Circular wait
- **Deadlock Prevention:** We can prevent a Deadlock by eliminating any of the above four conditions.

DEADLOCK PREVENTION

Eliminate Mutual Exclusion:

- Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource. However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.
- It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

DEADLOCK PREVENTION

Hold and Wait

- Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.
- However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.
- **!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)**

DEADLOCK PREVENTION

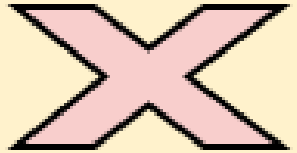
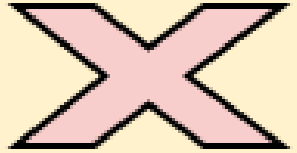
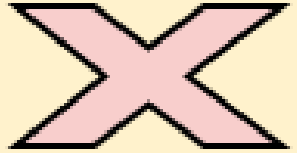
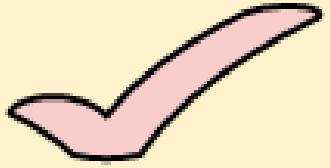
No Preemption

- Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.
- This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.
- Consider a printer is being used by any process. If we take the printer away from that process and assign it to some other process then all the data which has been printed can become inconsistent and ineffective and also the fact that the process can't start printing again from where it has left which causes performance inefficiency.

Circular Wait

- To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

DEADLOCK PREVENTION

Condition	Approach	Is Practically Possible?
Mutual Exclusion	Spooling	
Hold and Wait	Request for all the resources initially	
No Preemption	Snatch all the resources	
Circular Wait	Assign priority to each resources and order resources numerically	

DEADLOCK AVOIDANCE

- **Deadlock avoidance** ensures system will never go into a state which could potentially create a deadlock situation.
- It requires that the operating system be given additional information **in advance** such as:
  - Resources currently allocated to each process.
  - Future requests and release of resources for each process.
- Two Deadlock avoidance algorithms are
  - **Resource-allocation-graph algorithm**: If there is only one instance of each resource type.
  - **Banker's algorithm**: If there are multiple instances each resource type.

SAFE STATE

Consider a system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2.

- At time *t*0, state is as shown in the Figure 1.
- As the sequence <*P*1, *P*0, *P*2> satisfies the safety condition. Hence, at t0, the system is in **safe state**.
- Suppose that, at time *t*1, process *P*2 requests and is allocated one more tape drive. **The system is no longer in a safe state. Why?**
- So, If we had made *P*2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.
- Suppose that, at time *t*1, instead of process *P*2, P1 requests and is allocated one more tape drive. Would the system state be safe then?

Processes	Maximum needs	Current Allocation	Available
P0	10	5	3
P1	4	2	
P2	9	2	

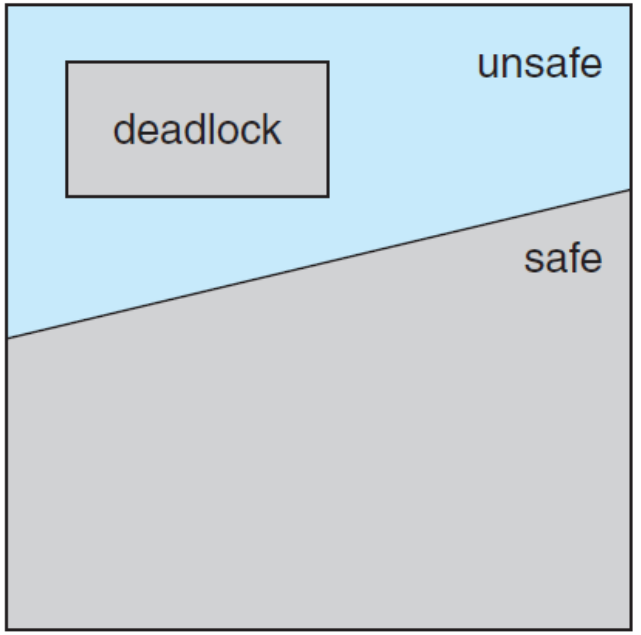
Figure 1. Current state at t0.

Processes	Maximum needs	Current Allocation	Available
P0	10	5	2
P1	4	2	
P2	9	3	

Figure 2. Current state at t1.

SAFE STATE

- The resource allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the processes.
- A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**.
- A sequence of processes <*P*1, *P*2, ..., *Pn*> is a safe sequence for the current allocation state if, for each *Pi*, the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj*, with *j* <*i*.
- If no such sequence exists, then the system state is said to be *unsafe*.
- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.



DEADLOCK AVOIDANCE USING RESOURCE ALLOCATION GRAPH

- Consider the resource-allocation graph of Figure 1.
- In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge**.
- A claim edge $Pi \rightarrow Rj$ indicates that process Pi may request resource Rj at some time in the future.
- Suppose that $P2$ requests $R2$. Although $R2$ is currently free, we cannot allocate it to $P2$, since this action will create a cycle in the graph.
- A cycle, as mentioned, indicates that the system is in an unsafe state.
- If $P1$ requests $R2$, and $P2$ requests $R1$, then a deadlock will occur.

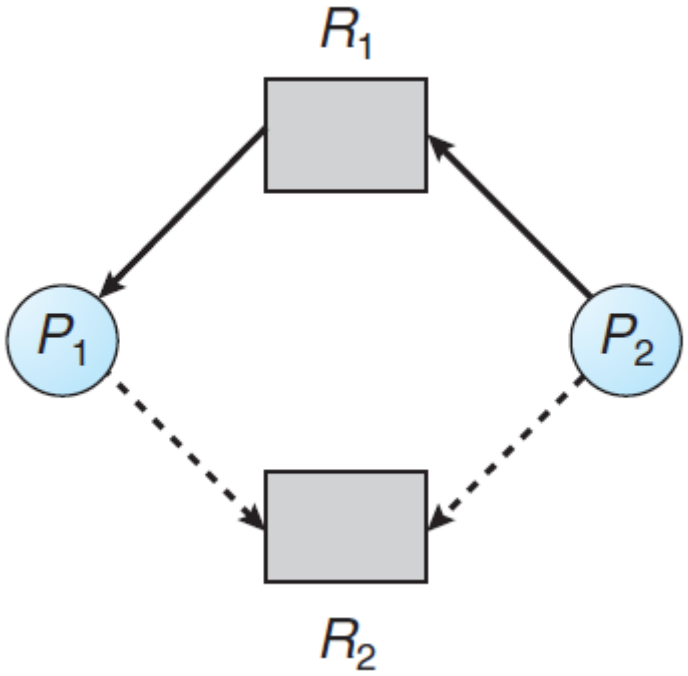
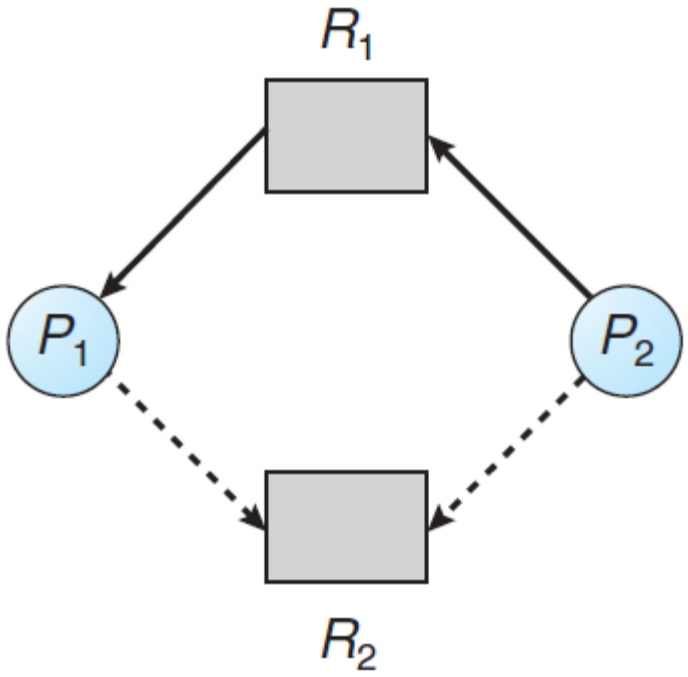


Figure 1

DEADLOCK AVOIDANCE USING RESOURCE ALLOCATION GRAPH

- When process *Pi* requests resource *Rj*, the claim edge *Pi* → *Rj* is converted to a request edge. **The request can be granted only if converting the request edge *Pi* → *Rj* to an assignment edge *Rj* → *Pi* does not result in the formation of a cycle in the resource-allocation graph.**
- If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.



DEADLOCK AVOIDANCE: BANKER'S ALGORITHM

- Consider a system with five processes *P*0 through *P*4 and three resource types *A*, *B*, and *C*. For example, A be a Tape drive, B be Scanner and C be CD Rom.
- Let Resource type *A* has 10 instances, resource type *B* has 5 instances, and resource type *C* has 7 instances.
- Let at T0, snapshot of the system be as shown in Table 1.
- Additionally, the matrix ***Need*** is defined as

  Max- Allocation and shown in Table 2.

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	I	0	7	5	3	3	3	2
PI	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	I	I	2	2	2			
P4	0	0	2	4	3	3			

Table 1

	Need		
	A	B	C
P0	7	4	3
PI	I	2	2
P2	6	0	0
P3	0	I	I
P4	4	3	I

Table 2

DEADLOCK AVOIDANCE: BANKER'S ALGORITHM

- The system is currently in a safe state. How?
- The sequence <*P*1, *P*3, *P*4, *P*2, *P*0> satisfies the safety criteria.

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
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			

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

DEADLOCK AVOIDANCE: BANKER'S ALGORITHM

- Suppose now that process *P*1 requests one additional instance of resource type *A* and two instances of resource type *C*, so *Request*1 = (1,0,2).
- first check that *Request*1 ≤ ***Available***—that is, that (1,0,2) ≤ (3,3,2), which is true.
- Then pretend that this request has been fulfilled, and we arrive at the following new state:
- We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <P_1, P_3, P_4, P_0, P_2> satisfies the safety requirement. Hence, we can immediately grant the request of process P_1.

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	2	3	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			

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

DEADLOCK AVOIDANCE: BANKER'S ALGORITHM

- However, when the system is in this state, a request for (3,3,0) by *P4* cannot be granted, since the resources are not available.
- Furthermore, a request for (0,2,0) by *P0* cannot be granted, even though the resources are available, since the resulting state is unsafe.

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	2	3	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			

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

DEADLOCK AVOIDANCE: BANKER'S ALGORITHM
(RESOURCE-REQUEST ALGORITHM)

The algorithm for determining whether requests can be safely granted can be described as follows.

Let $\boldsymbol{Request}_i$ be the request vector for process *Pi* . If ***Requesti*** [*j*] == *k*, then process *Pi* wants *k* instances of resource type *Rj*. When a request for resources is made by process *Pi*, the following actions are taken:

1. If $\boldsymbol{Request}_i \leq \boldsymbol{Need}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $\boldsymbol{Request}_i \leq \boldsymbol{Available}$, go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

   $\boldsymbol{Available} = \boldsymbol{Available} - \boldsymbol{Request}_i$;

   $\boldsymbol{Allocation}_i = \boldsymbol{Allocation}_i + \boldsymbol{Request}_i$;

   $\boldsymbol{Need}_i = \boldsymbol{Need}_i - \boldsymbol{Request}_i$;

   If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for ***Requesti*** , and the old resource allocation state is restored.

DEADLOCK AVOIDANCE: BANKER'S ALGORITHM
(SAFETY ALGORITHM)

Algorithm for finding out whether or not a system is in a safe state can be described as follows:

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

   Work = ***Available*** and ***Finish***[i] = ***false*** for $i = 0, 1, ..., n - 1$.

2. Find an index 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.