Saye State

A System is in Saye State only if there exists a saye sequence.

A system is in Saye State if there exists a sequence $\langle P_1, P_2, \ldots P_n \rangle$ of all the processes in the systems such that for P_i, the resources that P_i can still request can be satisfied by currently available resources held by all the P_j, where $j < i$.

And,

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

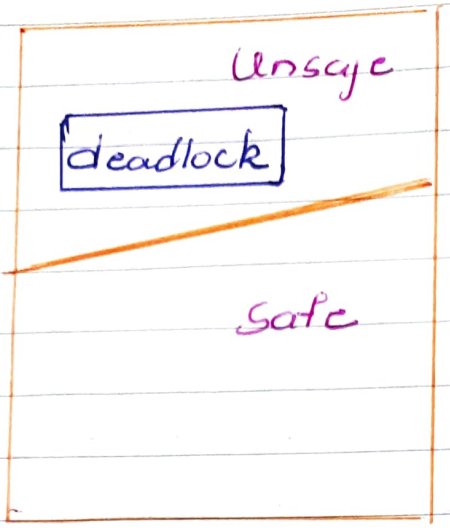
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources & terminate.

- When P_i terminates, P_{i+1} can obtain its needed resources and so on.

↳ A saye state indicates no deadlock

↳ Unsaye State indicates the possibility of deadlock

Avoidance is to ensure System will not enter unsaye state.



A safe state is not a deadlocked state & deadlocked state is an unsafe state. An unsafe state may lead to deadlock.

Example:

A system has 12 tapes. &

	Max needs	Current Need(holding)
P_0	10	5
P_1	4	2
P_2	9	2

The sequence $< P_1, P_0, P_2 >$ satisfies the safety condition.

↳

P_1 needs 4 & it has acquired 2. Total used tapes are 9 & 3 more are available. P_1 uses & release all 4.

and so on.

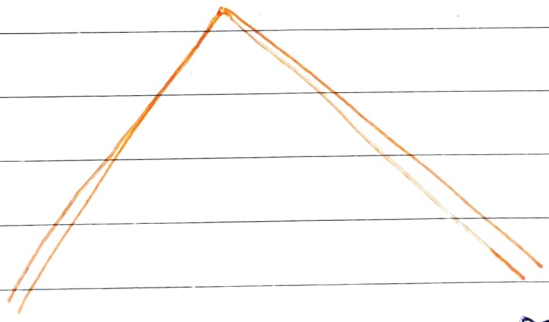
$< P_1, P_0, P_2 >$ is safe state. sequence giving system a safe state.

Now assume P_2 makes a request for one more & gets allocated. We have:

	Max Needs	Current Need (hold)
P_0	10	5
P_1	4	2
P_2	9	3

When P_1 completes and returns, there are 4 available statapes but both P_0 & P_2 need 5. So, we enter a dead lock. Hence the system is no more in saje state.

Avoidance Algorithms



Single instance of resource type

Multiple instances of resource type

Resource Allocation Graph

Banker's Algorithm

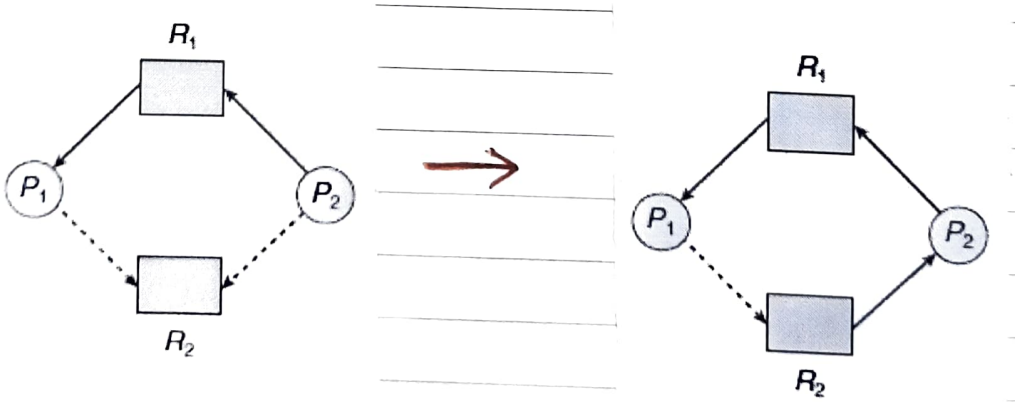
Resource Allocation Graph Algorithm

We introduce a new type of edge called as Claim edge.

- Claim edge, $P_i \rightarrow R_j$ indicates that Process P_i may request resource R_j
- It is represented by a dashed line
- Claim converts to request edge when process requests a resource
- Request edge is converted to an assignment edge when resource is allocated.
- When resource is released, assignment edge reconverts to a claim edge

So,

- Resources must be claimed a priori in the System.



The request can be granted only if converting the request edge to an assignment edge doesnot result in the formation of a cycle in resource allocation graph.

Banker's Algorithm

- Each process must a priori claim the 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

n = number of processes
m = number of resource types

then,

- **Available**. A vector of length m indicates the number of available resources of each type. If $Available[j]$ equals k, then k instances of resource type R_j are available.
- **Max**. An $n \times m$ matrix defines the maximum demand of each process. If $Max[i][j]$ equals k, then process P_i may request at most k instances of resource type R_j.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $Allocation[i][j]$ equals k, then process P_i is currently allocated k instances of resource type R_j.
- **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If $Need[i][j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that $Need[i][j]$ equals $Max[i][j] - Allocation[i][j]$.

As described above:

Need[i][j] = Max[i][j] - Allocation[i][j]

Using above, we define Safety algorithm as,

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.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is saye.

We next decribe the algorithm for determining whether requests can be safely granted. Let Request i be the request vector for process P_i. If Request[j] == k, then process P_i wants k instances of R_j. When a request is made by P_i, following occur:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process 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 the resulting resource-allocation is safe, the transaction is completed & P_i is allocated its resources.

If the state is unsafe, P_i must wait for Request$_i$ & the old resource-allocation state is restored.

Example:

	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	→ Snapshot at T_0

Need = Max - Allocation.

	Need	We have
	A B C	A = 10 instances
P_0	7 4 3	B = 05 instances
P_1	1 2 2	C = 07 instances.
P_2	6 0 0	
P_3	0 1 1	
P_4	4 3 1	

The sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the Safety Criteria.

Let us now say, P_1 requests one additional instance of A & 2 additional of C.

$Request_1 = (1, 0, 2)$

We check if $Request_1 \leq Available$

$(1, 0, 2) \leq (3, 3, 2)$

When this is granted, we arrive at:

	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 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Is this new State Safe?

→ Yes, we can obtain the Safe sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ using Safety algorithm & the request can be granted.

Conside the following new requests:

* $(3, 3, 0)$ by P_4 → Cannot be granted
* $(0, 2, 0)$ by P_0 → Results in unsafe state & hence cannot be granted.

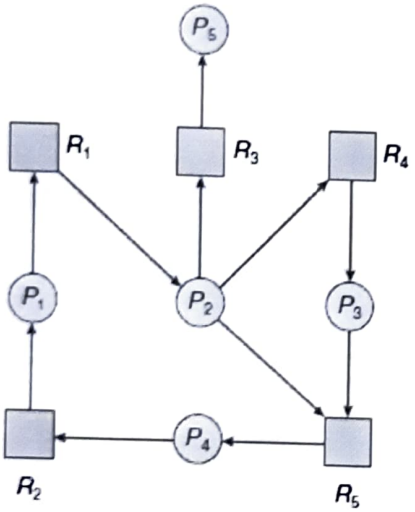
Note: Can you implement the algorithm?

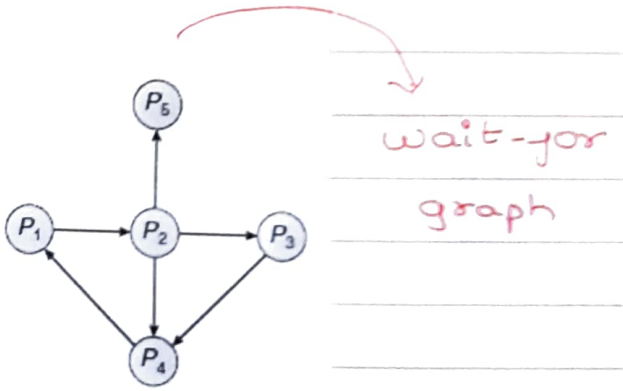
Deadlock Detection

If we don't employ either prevention or avoidance, we need

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

Single Instance of each resource Type





Convert the resource-allocation graph to wait-for graph.

Periodically invoke the algorithm that searches for a cycle in the graph. If there is a cycle, there is a deadlock.

An algorithm to detect a cycle in a graph requires $O(n^2)$.

$n \rightarrow$ no of vertices.

Several Instances of a Resource Type

Data Structures:

- **Available**. A vector of length m indicates the number of available resources of each type.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request**. An $n \times m$ matrix indicates the current request of each process. If $Request[i][j]$ equals 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 $Work = Available$. For $i = 0, 1, ..., n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$. Otherwise, $Finish[i] = true$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == false$ for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

The algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

prakash hegade

Example:

A = 7 instances B = 2 instances C = 6 instances

	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	

We can find that $<P_0, P_2, P_3, P_1, P_4>$ is a Safe Sequence.

Consider a new request of P_2 as (0, 0, 1). Request matrix will be,

	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

Now there is a deadlock. Only finish of P_0 will be true & rest all will be false.
P_1, P_2, P_3 and P_4 are deadlocked.

Detection-Algorithm Usage:

The algorithm invoke depends on two factors:

- How often is a deadlock likely to occur?
- How many processes will be affected by deadlock when it happens?
  - ↳ the cost of roll back

Recovery From Deadlock:

Once a deadlock is detected,
- The informed operator can deal with it manually
- System recovers from it automatically.

We can abort the process or pre-empt the resources from deadlocked processes.

Process Termination

→ Abort all deadlocked processes
- great expense
- will have to do recomputations

→ One process at a time is aborted until its deadlock free.
- after each abort deadlock detection algorithm needs to be invoked.

Abort can consider following parameters

- priority of process
- time completed & remaining
- Resources used
- resources required
- how many processes will need to be terminated
- Is process interactive or batch?

Resource Preemption

If preemption is required to deal with deadlocks, then three issues need to be addressed.

- Selecting a victim such that cost is minimized
- Rollback and return to some safe state, restart process from that state.
- Same process may always be picked as victim causing starvation. So cost factor can include number of rollbacks done to prevent starvation.