DEAD LOCKS

Resource Allocation

Sharable Resources

Resources that can be shared among number of processes

- Memory, Hard Disk etc..

Non Sharable resources

Resources that cannot be shared among number of processes

- Printer, Tape Drive, CDROM Drive etc...

A resource type can have more than one instance

Resource Allocation

Under normal operation, a resource allocations proceed like this::

1. Request a resource (suspend until available if necessary).

2. Use the resource.

3. Release the resource.

Deadlock Characterization

Deadlock can arise if <u>four</u> 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.

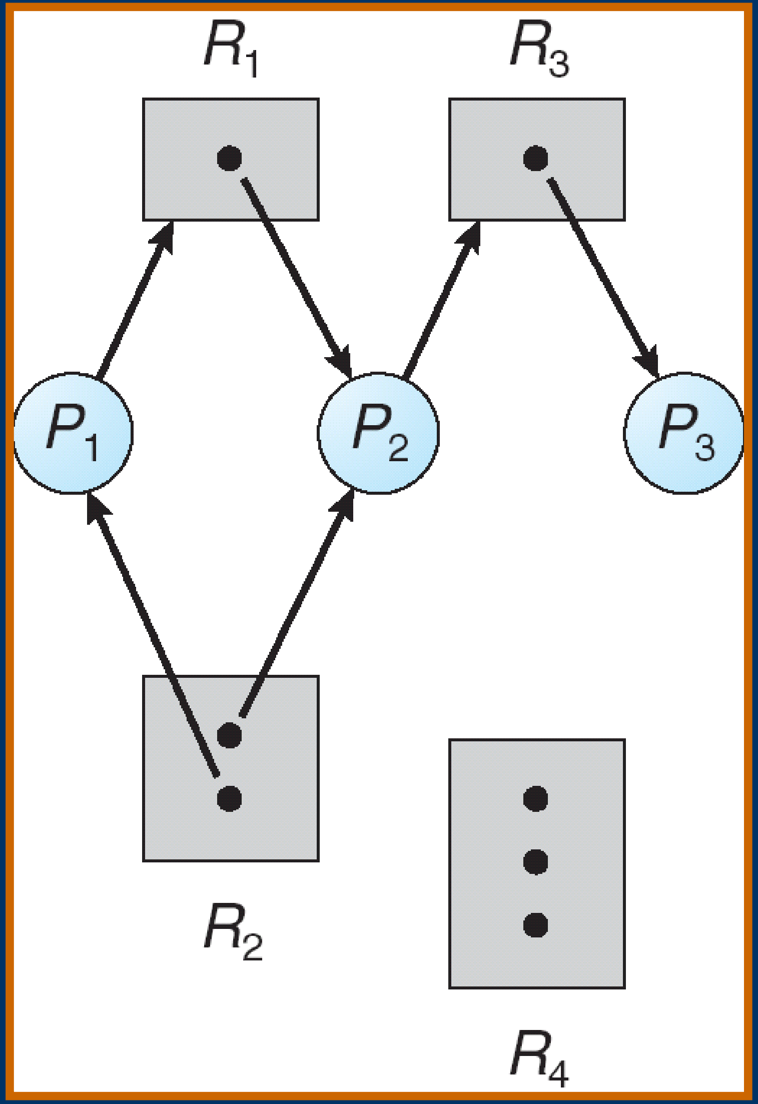
Deadlock Characterization

Deadlock can arise if <u>four</u> conditions hold simultaneously.

- **No Resource Preemption**
  - 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 $\{P_0, P_1, \ldots, Pn\}$ 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.

Directed Resource Allocation Graph

- **DRAG** is a Graphical representation showing the relationship between processes and resources.
- Helps in detecting deadlocks.

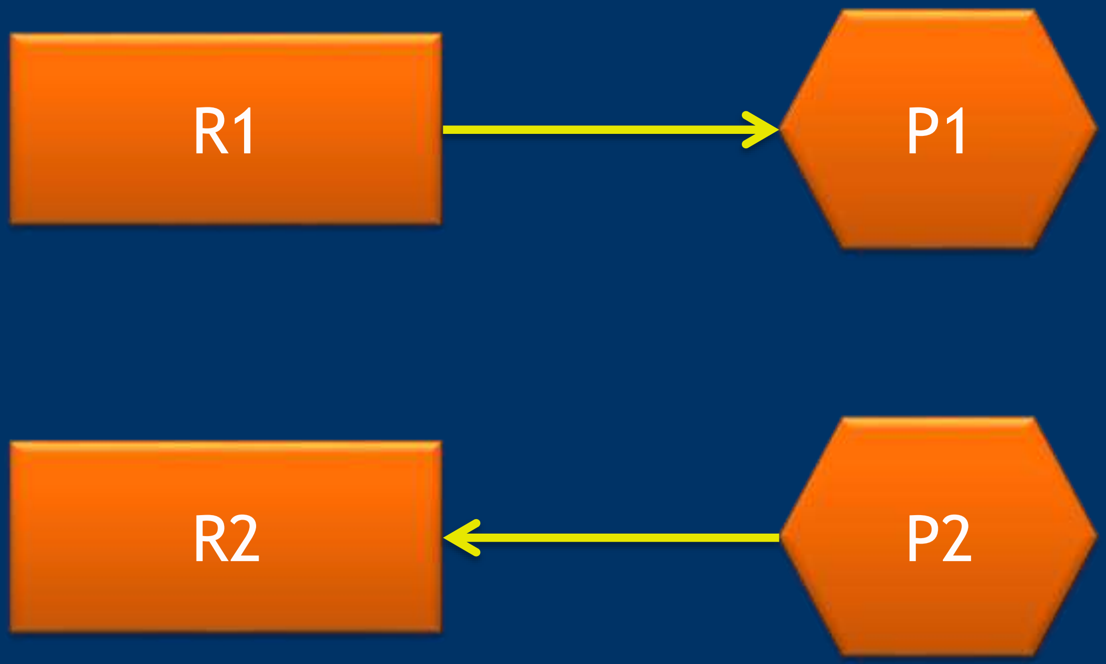


Resource-Allocation Graph

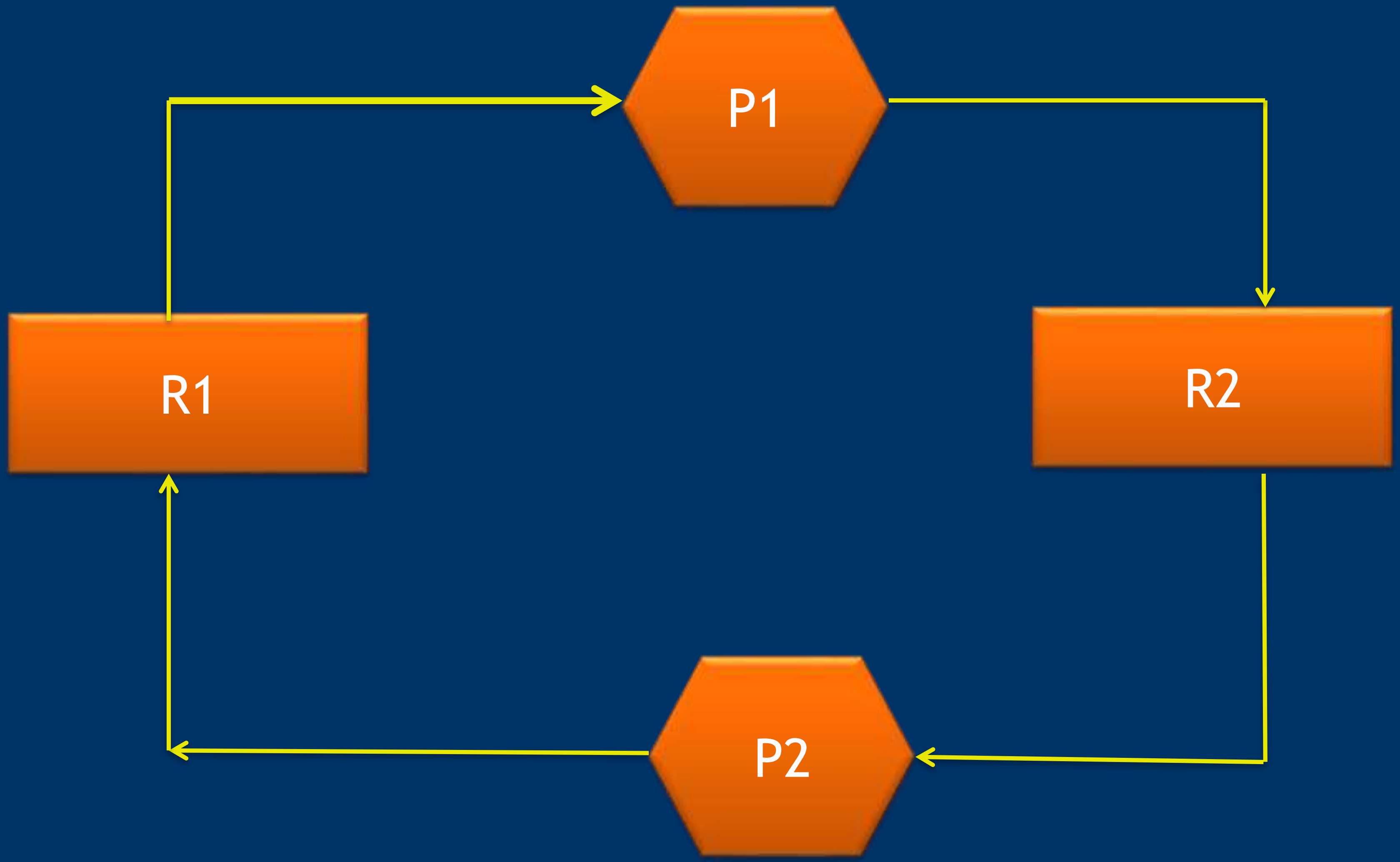
A set of vertices *V* and a set of edges *E*.

- 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 $P_i \rightarrow R_j$
- Assignment edge - directed edge $R_j \rightarrow P_i$

Resource Allocation Graph

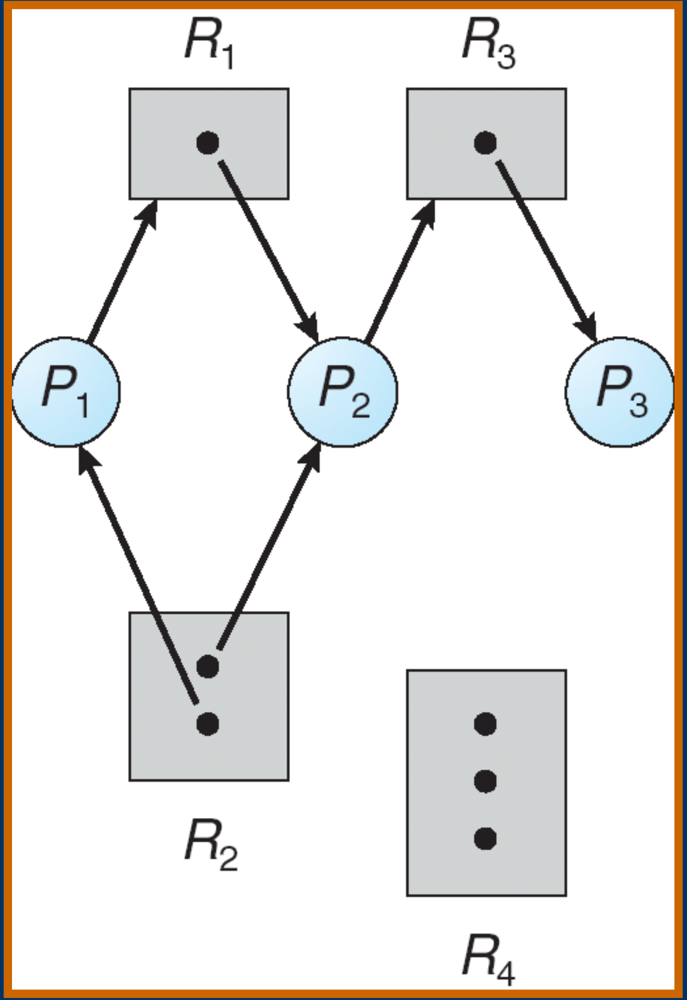


Resource Allocation Graph

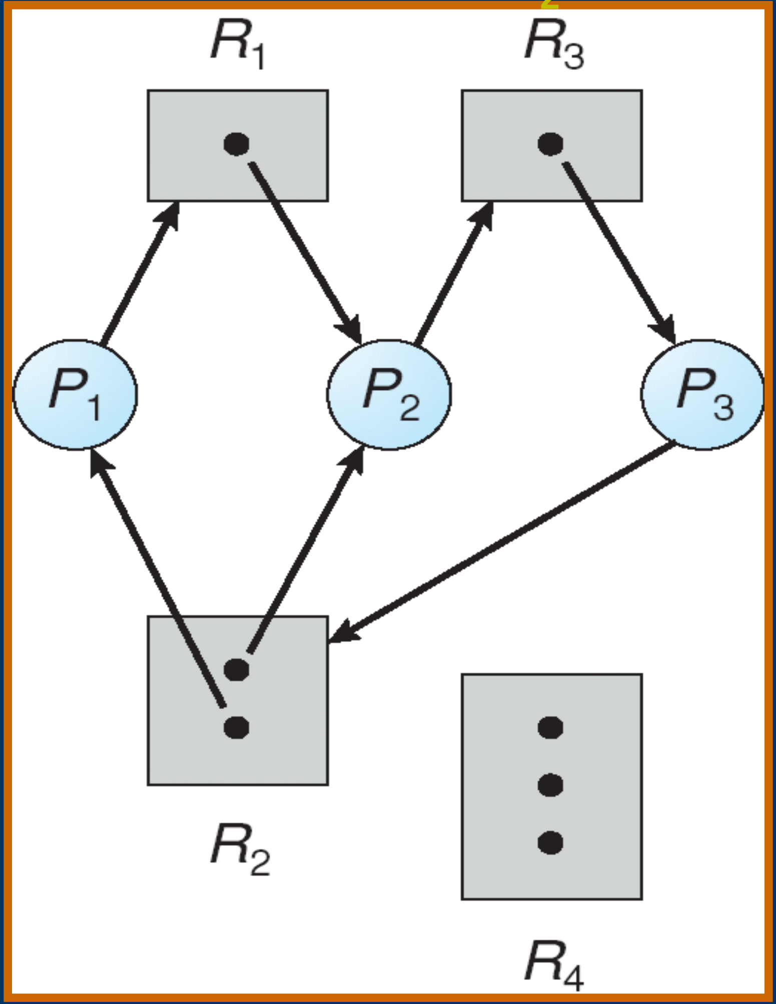


Resource Allocation Graph With A Deadlock

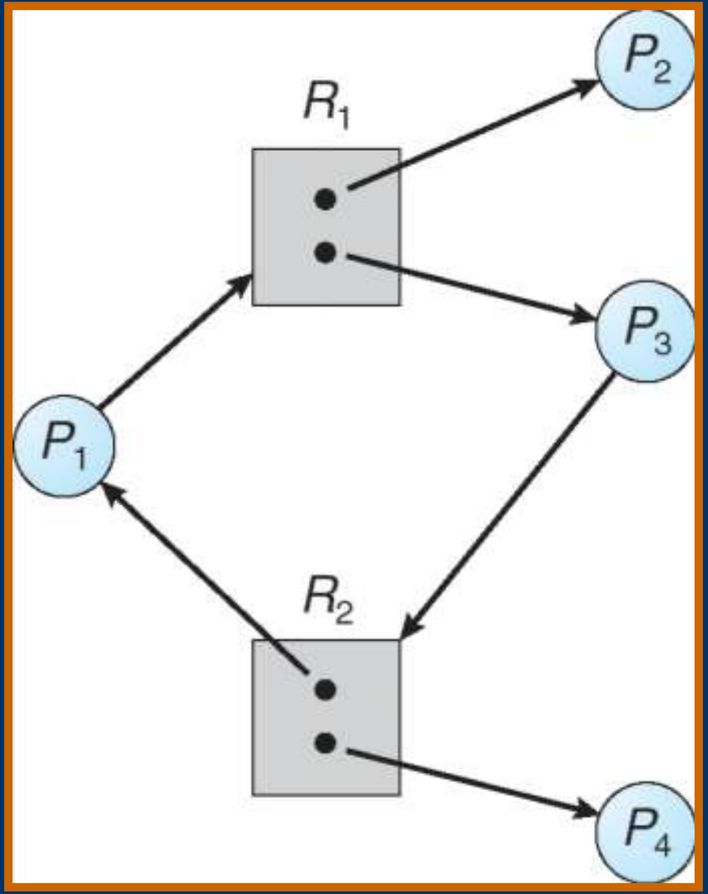
Before P_3 requested an instance of R_2



After P_3 requested an instance of R_2



Graph With A Cycle But No Deadlock



Process P_4 may release its instance of resource type R_2. That resource can then be allocated to P3, thereby breaking the cycle.

Relationship of cycles to deadlocks

- If a resource allocation graph contains no cycles

  $\Rightarrow$ No deadlock

- If a DRAG contains a cycle and if only one instance exists per resource type

  $\Rightarrow$ Deadlock

- If a DRAG contains a cycle and if several instances exists per resource type

  $\Rightarrow$ Possibility of deadlock

Deadlock Strategies

- Ignoring Deadlock
- Detecting Deadlock
- Recovering From Deadlock State
- Deadlock Prevention
- Deadlock Avoidance

Deadlock Avoidance

- **Deadlock prevention**
  - Low device utilization
  - Low system throughput
- **Deadlock Avoidance**
  - Additional information about how resources are to be requested.

Deadlock Avoidance

- 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

Deadlock Avoidance

- **Allows more concurrency than prevention**
- **Deadlock Avoidance Algorithm**
  - Do not start a process if its total demand might lead to deadlock.
  - Do not grant an incremental resource request if this allocation could lead to deadlock.

Deadlock Avoidance

- 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 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 $j < i$

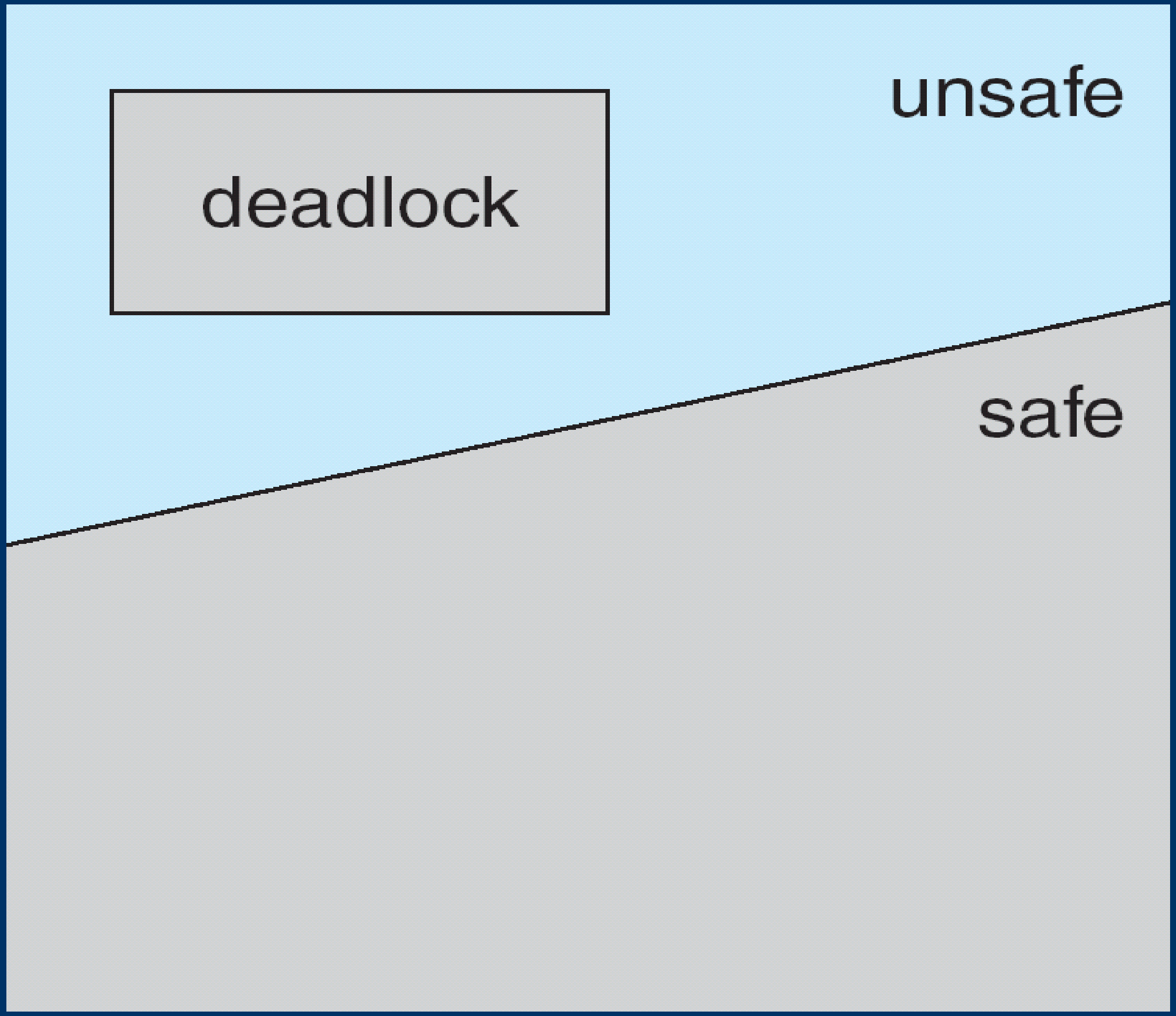
Deadlock Avoidance

- 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.

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



Banker's Algorithm

- Tentatively grant each resource request.
- Analyze resulting system state to see if it is "safe".
- If safe, grant the request.
- If unsafe refuse the request (undo the tentative grant)
- Block the requesting process until it is safe to grant it.

Data Structures for the Banker's Algorithm

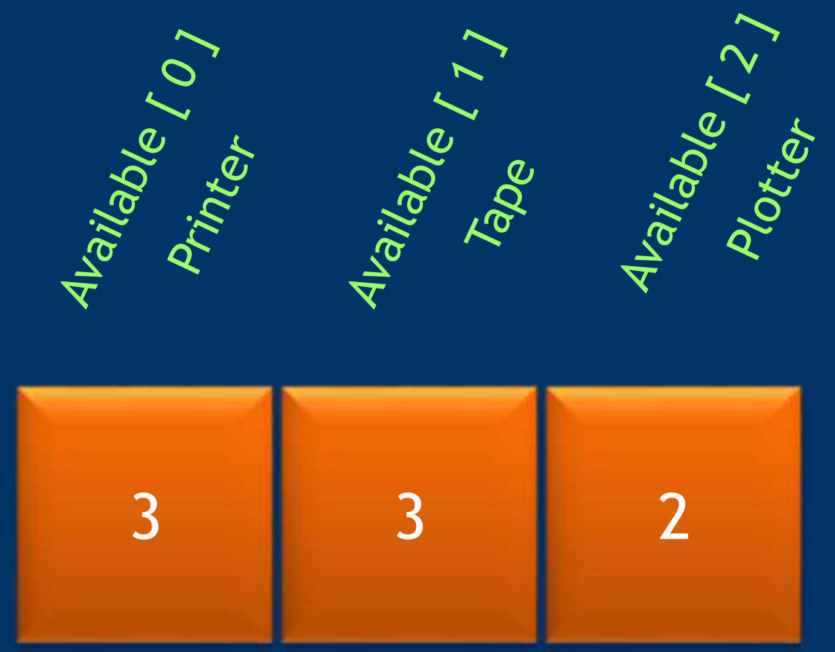
- Let n = number of processes, and m = number of resources types.
- **Available** : Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
- **MAX** : n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j

Data Structures for the Banker's Algorithm

- **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

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

Available [m]



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

MAX [n][m]

	Printer	Tape	Plotter
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

- If *Max* [i, j] = k, then process P_i may request at most k instances of resource type R_j

Allocation [n][m]

	Printer	Tape	Plotter
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

- If Allocation[i, j] = k then P_i is currently allocated k instances of R_j

	Printer	Tape	Plotter
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

	Printer	Tape	Plotter
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

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

Need [n][m]

	Printer	Tape	Plotter
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

- 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:

   (a) *Work = Available*

   (b) For $i = 1, 2, \ldots, n$, if $Allocation_i \neq 0$, then

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

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

   (a) $Finish[i] == false \;\&\&\; Need_i \leq Work$

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

Safety Algorithm

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.

Safety Algorithm

Work = Available, Finish[i]='F'

Work [0] Printer	Work [1] Tape	work [2] Plotter
3	3	2

WORK

P0	P1	P2	P3	P4
F	F	F	F	F

FINISH

5 - Processes

10 - Printers, 5 - Tapes, 7 - Plotter