

# Operating Systems

## Deadlocks - Complete

Dr. P. Sateesh Kumar



# Resources

- Physical resources
  - like prints, tape drives, memory space, and CPU cycles
- Logical resources
  - like files, semaphores and monitors
- Reusable Resources
  - Processes obtain resources that they later release for reuse by other processes
  - Processors, I/O channels, main and secondary memory, files, databases, and semaphores
- Consumable Resources
  - Created and destroyed by a process
  - Interrupts, signals, messages, and information in I/O buffers



# 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.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

# System Model

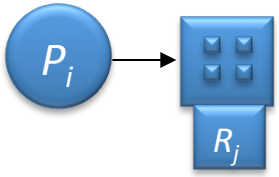
## Resource-Allocation Graph

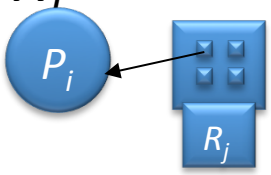
A set of vertices $V$ and a set of edges $E$.

- 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 $R_j \rightarrow P_i$

# System Model

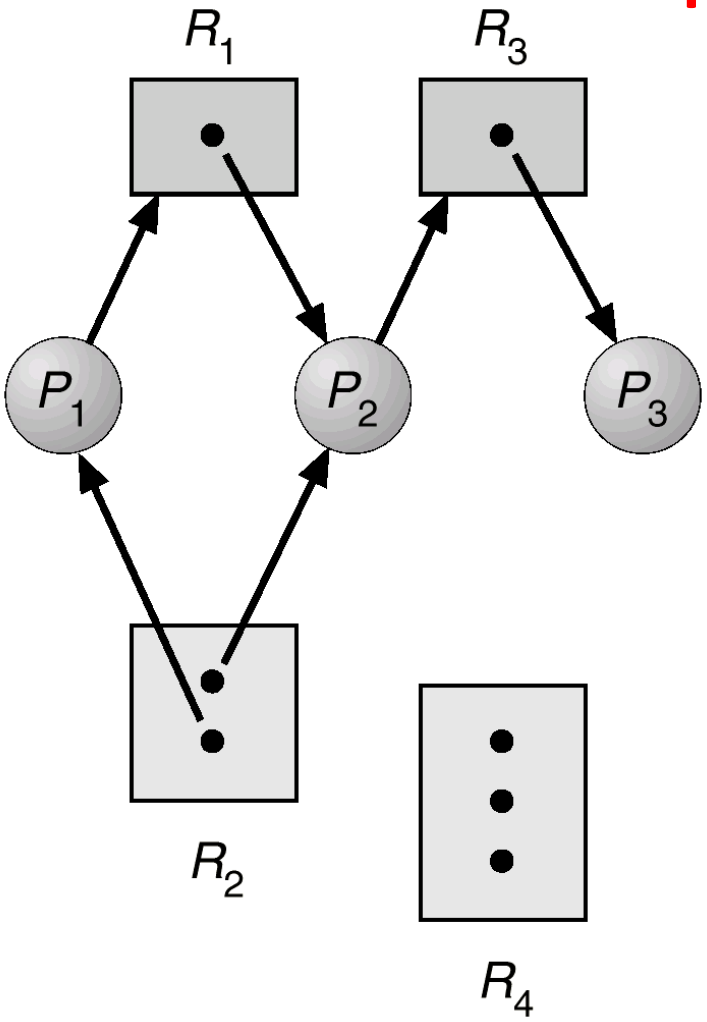
## Resource-Allocation Graph

- Process 
- Resource Type with 4 instances 
- $P_i$ requests instance of $R_j$ 

- $P_i$ is holding an instance of $R_j$ 

# System Model

## Example of a Resource Allocation Graph

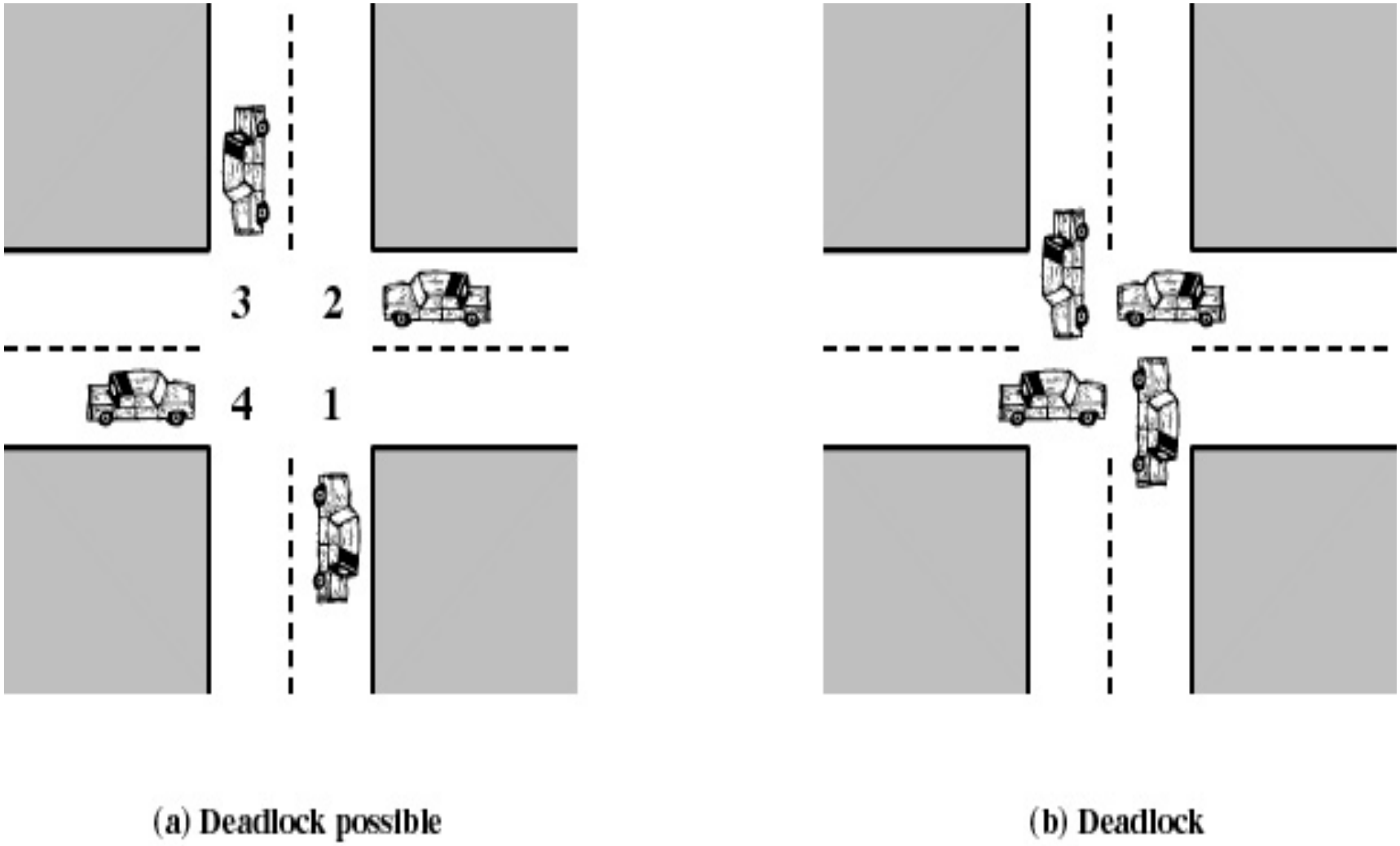




# Deadlocks

- In a multiprogramming environment, several processes may compete for a finite number of resources
- A process requests resources, if the resources are not available at that time, the process enters a wait state
- Some times a waiting process may never again able to change state, because the resources they have requested are held by other waiting processes
- This situation is called "***Deadlock***"

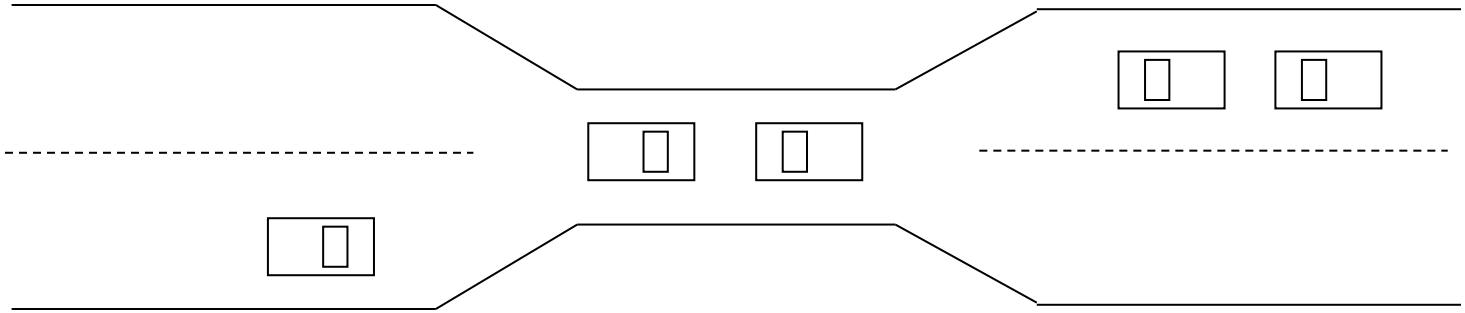
# Illustration of Deadlock




(a) Deadlock possible

(b) Deadlock

# Illustration of Deadlock



# Example

- A system has 2 hard disks
- P holds → 1 hard disk (say, A)
- Q holds → 1 hard disk (say, B)
- Both need another hard disk that is held by other process.
- Hence, Deadlock
  (if indefinite !!)

# Example

## Situation - 1

**Process P**

• • •
Get A
• • •
Get B
• • •
Release A
• • •
Release B
• • •

**Process Q**

• • •
Get B
• • •
Get A
• • •
Release B
• • •
Release A
• • •

## Situation - 2

**Process P**

• • •
Get A
• • •
Release A
• • •
Get B
• • •
Release B
• • •

**Process Q**

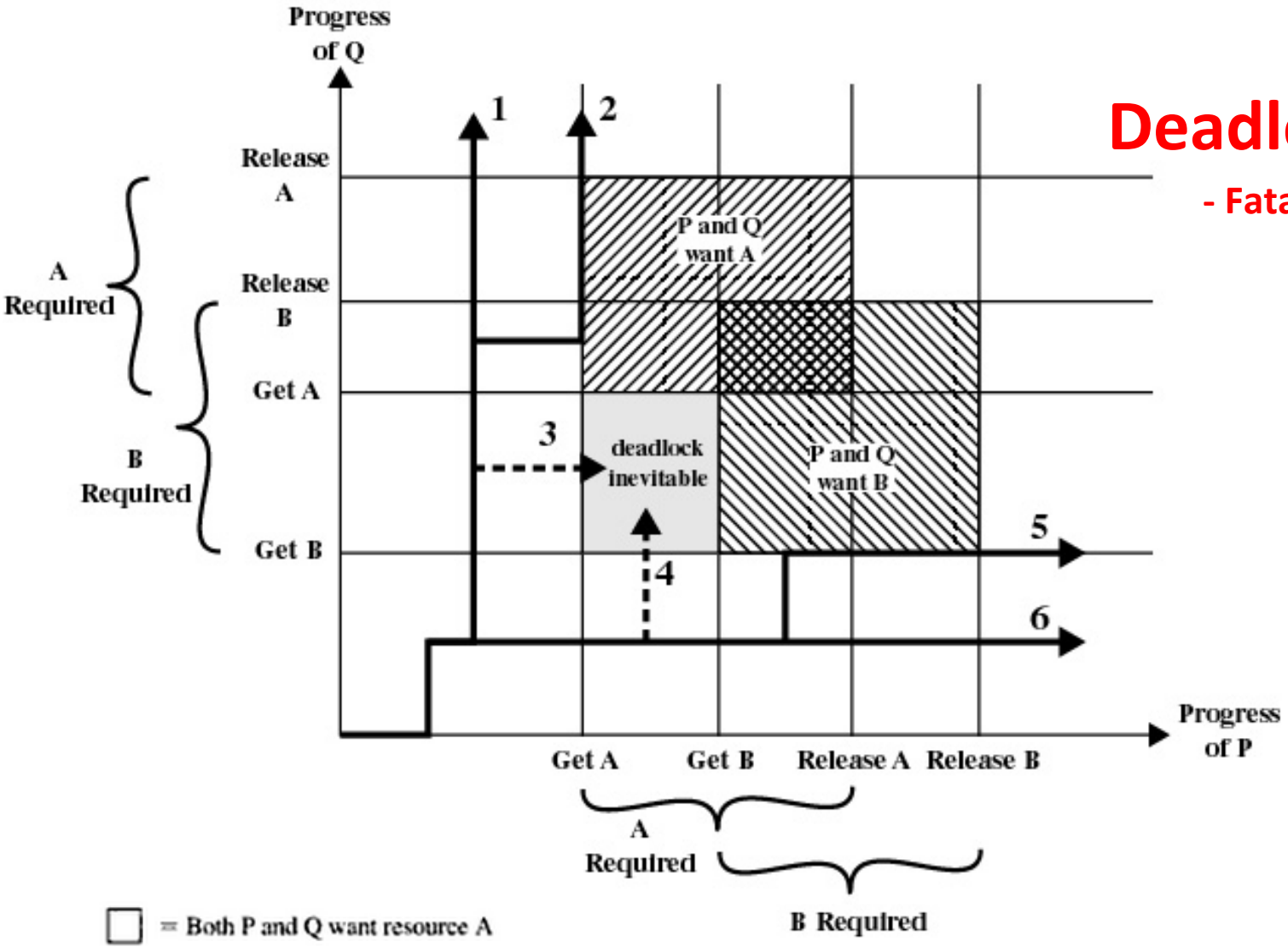
• • •
Get B
• • •
Get A
• • •
Release B
• • •
Release A
• • •



# Explanation

1. Q acquires B and then A and then releases B and A. When P resumes execution, it will be able to acquire both resources.
2. Q acquires B and then A. P executes and blocks on a request for A. Q releases B and A. When P resumes execution, it will be able to acquire both resources.
3. Q acquires B and then P acquires A. Deadlock is inevitable, because as execution proceeds, Q will block on A and P will block on B.
4. P acquires A and then Q acquires B. Deadlock is inevitable, because as execution proceeds, Q will block on A and P will block on B.
5. P acquires A and then B. Q executes and blocks on a request for B. P releases A and B. When Q resumes execution, it will be able to acquire both resources.
6. P acquires A and then B and then releases A and B. When Q resumes execution, it will be able to acquire both resources.

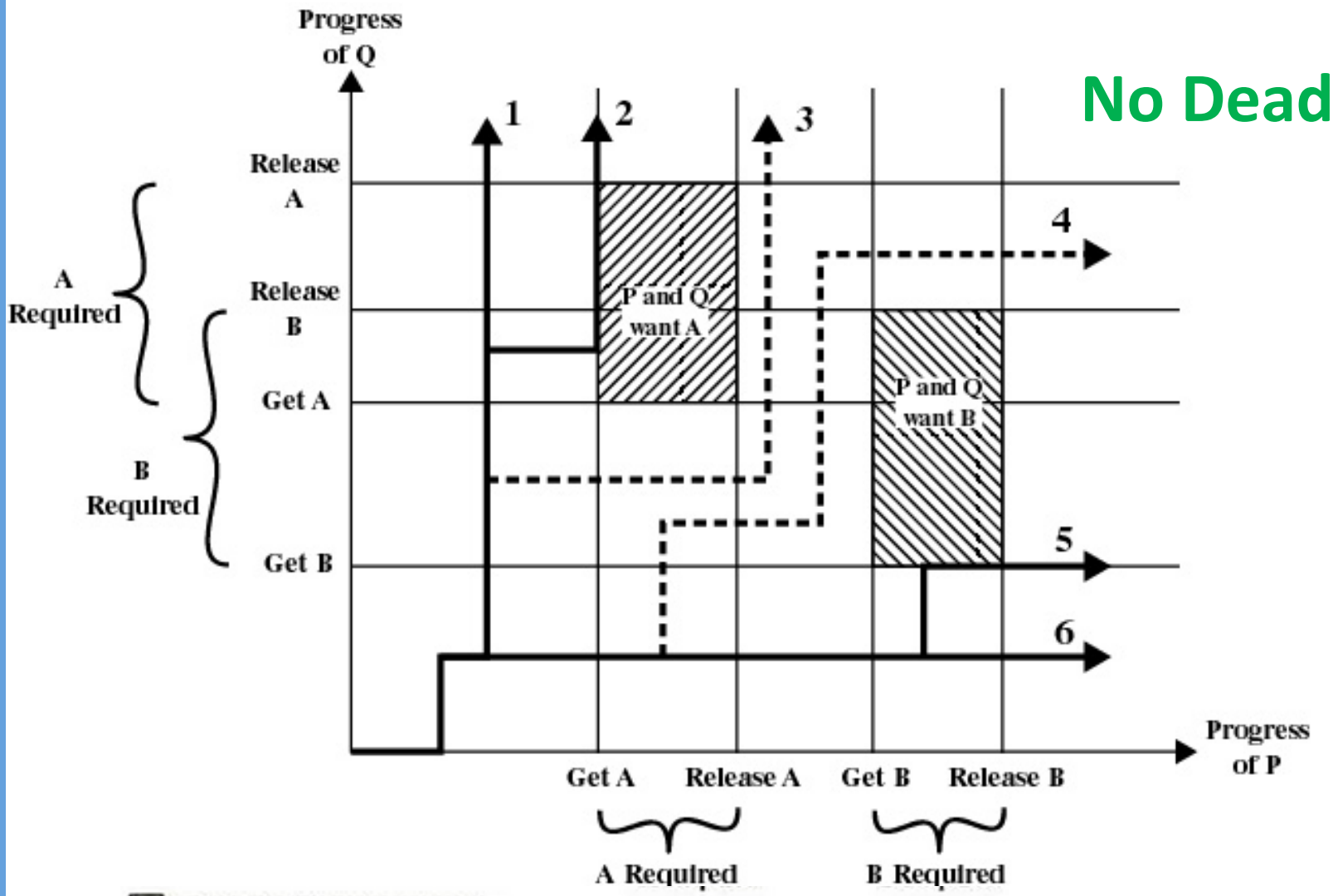
# Joint Progress Diagram




**Deadlock**

- Fatal region

# Joint Progress Diagram




**No Deadlock**

# When will Deadlock Occur?

1. Mutual exclusion
2. Hold and wait
3. No Preemption
4. Circular wait

**Necessary Conditions for a Deadlock to Occur**



# Conditions for Deadlock

- Mutual exclusion
  - only one process may use a resource at a time
- Hold-and-wait
  - A process request all of its required resources at one time



# Conditions for Deadlock

- No preemption
  - If a process holding certain resources is denied a further request, that process must release its original resources
  - If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources

# Conditions for Deadlock

- Circular wait
  - Prevented by defining a linear ordering of resource types

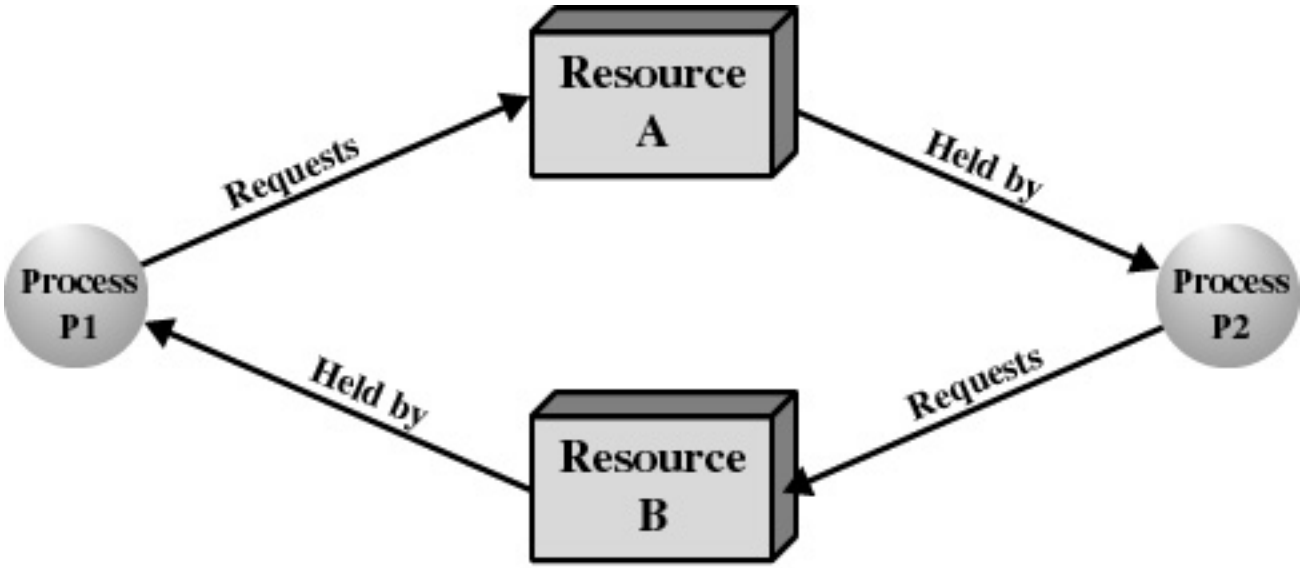
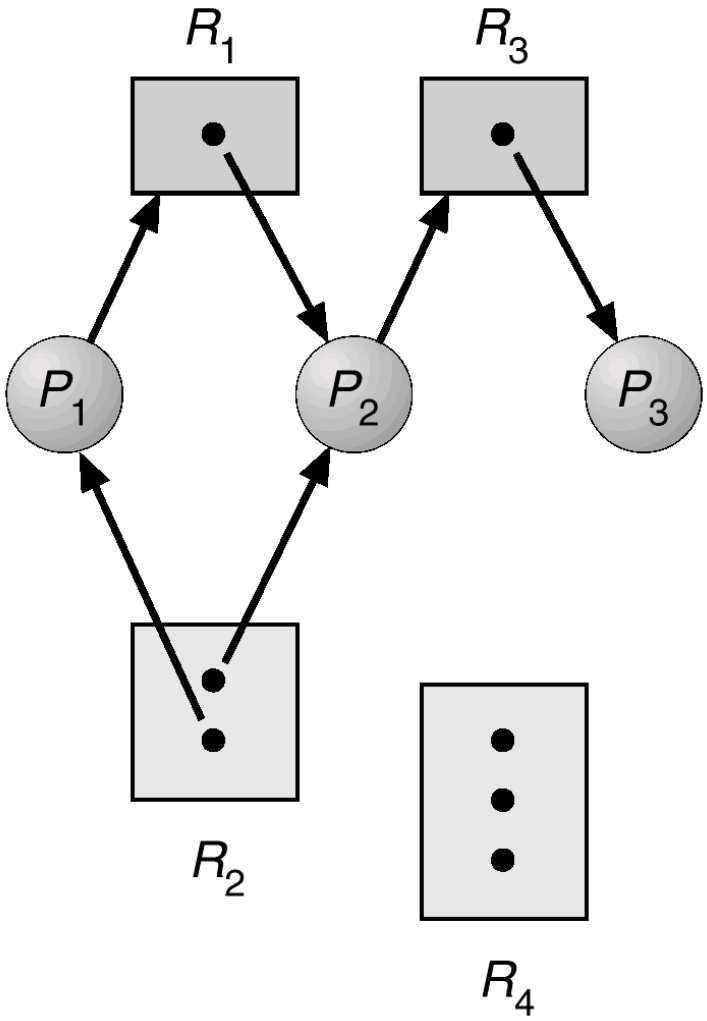
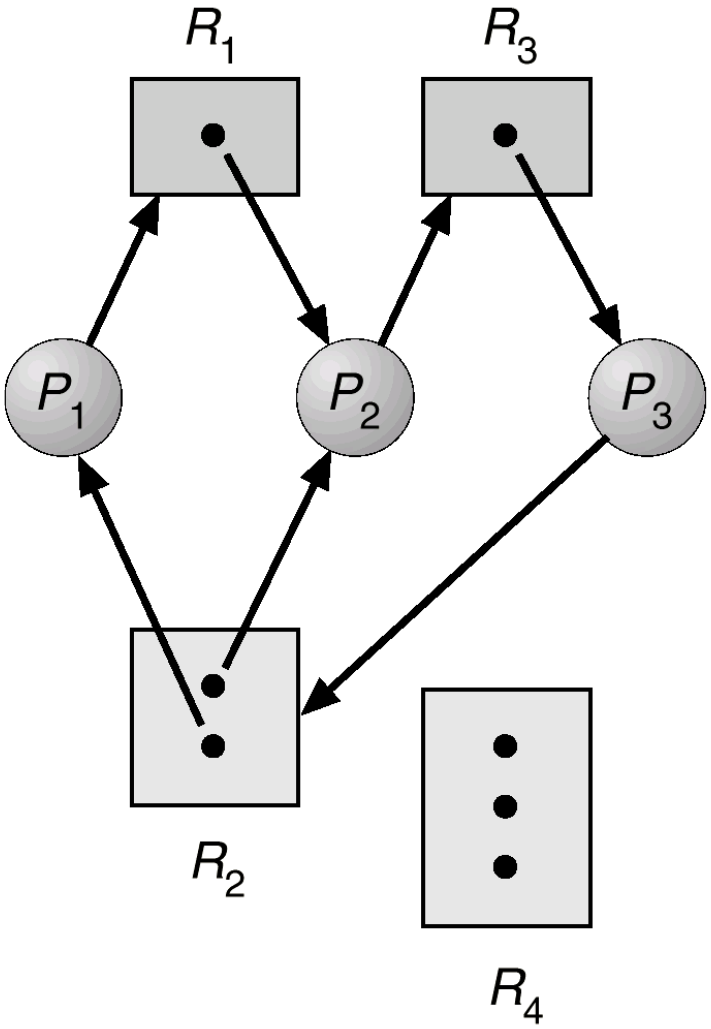



Figure 6.5 Circular Wait

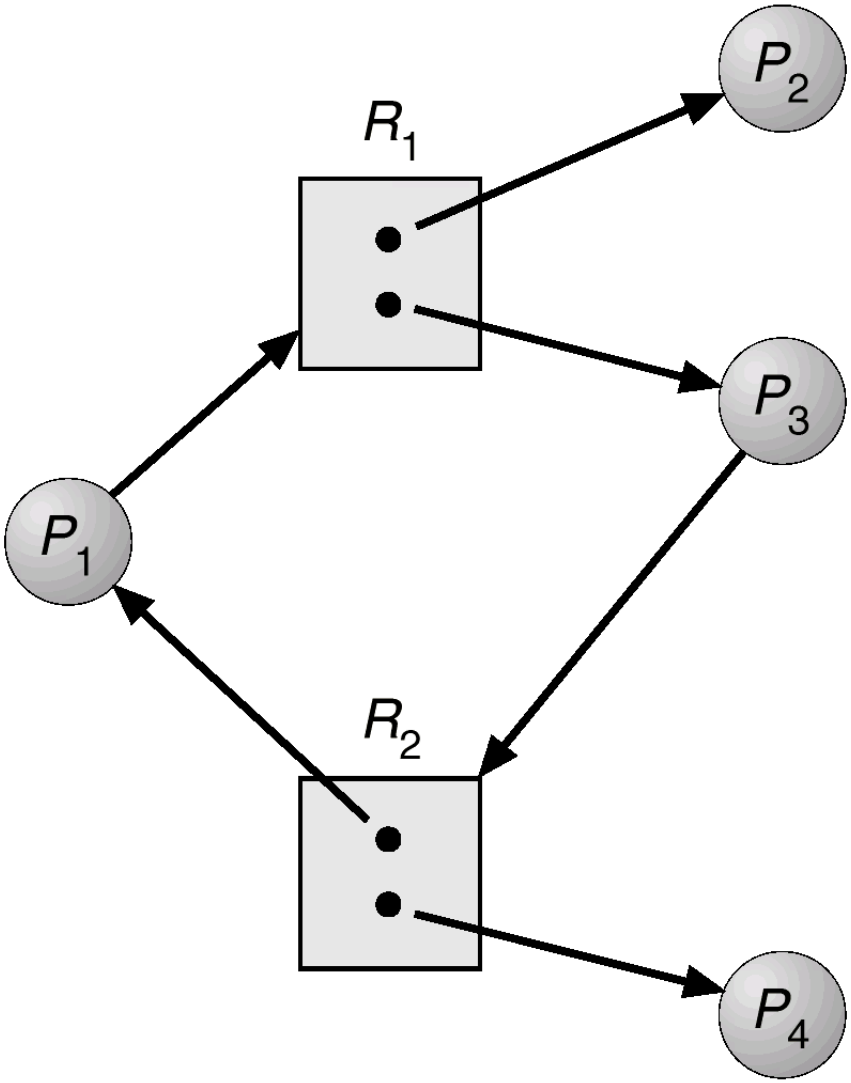
# Example - RAG



# RAG with Deadlock



# RAG with Cycle (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.

# Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state. **(Prevention)** **(Avoidance)**

  obstacle escape

- Allow the system to enter a deadlock state and then recover. **(Detection)** **(Recovery)**

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

# Deadlock Prevention

Restrain the ways request can be made.

(see that at least one of the 4 conditions fail)

- **Mutual Exclusion**
  - not required for sharable resources;
    - Ex: Files for reading (sharable) – allow simultaneous access
  - must hold for non-sharable resources.
    - Printer (non-sharable) – stop simultaneous access


# Deadlock Prevention

Restrain the ways request can be made.

- **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**.
  - **Low resource utilization; starvation possible.**

# Deadlock Prevention

Restrain the ways request can be made.

- **No Preemption**
  - **So, preempt the resource**
  - 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.

# Deadlock Prevention

Restrain the ways request can be made.

- **Circular Wait**
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

$R = \{R_1, R_2, \ldots, R_m\}$ R = {tape drives, disk drives, and printers}

$F: R \rightarrow N$ N is a natural number

$F(\text{tape drive}) = 1$
$F(\text{disk drive}) = 5$
$F(\text{printer}) = 12$

$$F(R_j) > F(R_i)$$



# 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 $<P_1, P_2, \ldots, P_n>$ is safe if for each $P_i$, the resources that *Pi* can still request can be satisfied by currently available resources + resources held by all the $P_j$, with $j<I$.

# Safe State

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

# Deadlock Avoidance

## Resource Allocation Graph (RAG)

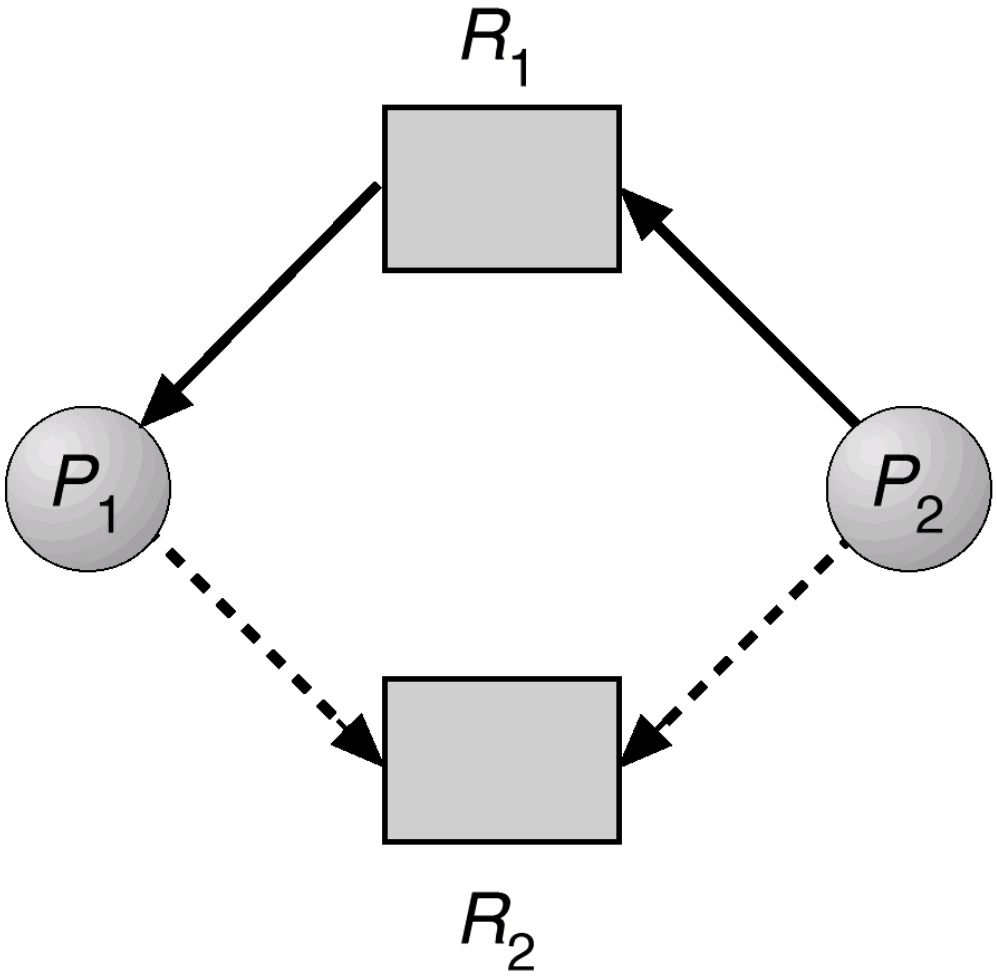
- ***Claim edge***
  - $P_i \rightarrow R_j$ indicated that process $P_j$ may request resource $R_j$; represented by a dashed line.

- Claim edge converts to request edge when a process requests a resource.

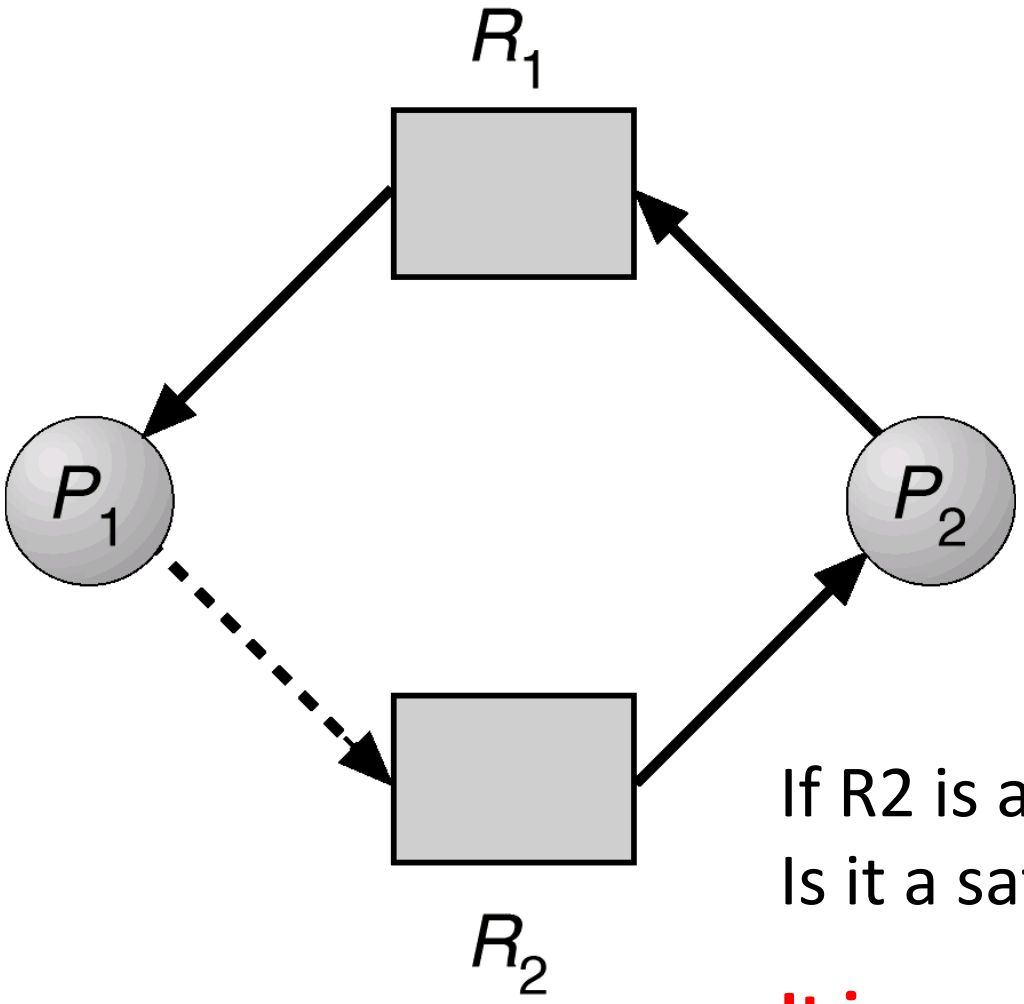
- When a resource is released by a process, assignment edge reconverts to a claim edge.

- Resources must be claimed *a priori* in the system.

# RAG for Deadlock Avoidance



# RAG for Deadlock Avoidance




If R2 is assigned to P2
Is it a safe state ?

**It is an unsafe state**



# RAG for Deadlock Avoidance

- Works **only for single instance** of a resource
- It does not work for multiple instances of resources

# Illustration of a Situation- Bank

**One Resource – Loan (say)**

Allocated

Need

| Customer | Credit Used | Credit Line |
|---|---|---|
| Andy | 0 | 6 |
| Barb | 0 | 5 |
| Marv | 0 | 4 |
| Sue | 0 | 7 |
| Funds Available | 10 | |
| Max Commitment | | 22 |

$t_o$

citibank

STATE BANK OF INDIA

# Illustration of a Situation- Bank

**One Resource – Loan (say)**

Allocated

Need

| Customer | Credit Used | Credit Line |
|---|---|---|
| Andy | 1 | 6 |
| Barb | 1 | 5 |
| Marv | 2 | 4 |
| Sue | 4 | 7 |
| Funds Available | 2 | |
| Max Commitment | | 22 |

$t_1$

# Illustration of a Situation- Bank

**One Resource – Loan (say)**

Allocated

Need

| Customer | Credit Used | Credit Line |
|---|---|---|
| Andy | 1 | 6 |
| Barb | 2 | 5 |
| Marv | 2 | 4 |
| Sue | 4 | 7 |
| Funds Available | 1 | |
| Max Commitment | | 22 |

$t_2$

# Illustration of a Situation- General

## Multiple Resources

Existing

$E = (6, 3, 4, 2)$

Possessed

$P = (5, 3, 2, 2)$

Available

$A = (1, 0, 2, 0)$

$$A = E - P$$

| Resources Assigned | | | | |
|---|---|---|---|---|
| Processes | Tapes | Plotters | Printers | Toasters |
| A | 3 | 0 | 1 | 1 |
| B | 0 | 1 | 0 | 0 |
| C | 1 | 1 | 1 | 0 |
| D | 1 | 1 | 0 | 1 |
| E | 0 | 0 | 0 | 0 |
| Total Existing | 6 | 3 | 4 | 2 |
| Total Claimed by Processes | 5 | 3 | 2 | 2 |
| Remaining Unclaimed | 1 | 0 | 2 | 0 |

| Resources Still Needed | | | | |
|---|---|---|---|---|
| Processes | Tapes | Plotters | Printers | Toasters |
| A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 1 | 2 |
| C | 3 | 1 | 0 | 0 |
| D | 0 | 0 | 1 | 0 |
| E | 2 | 1 | 1 | 0 |

# Illustration of a Situation- General

1. **Look** for a row whose unmet needs don't exceed what's available. i.e., a row **where P<=A** if **no** such row exists, the system **is deadlock**ed because no process can acquire the resources it needs to run to completion. If **there's** more than one such row, just **pick one**

2. Assume that the process chosen in 1 **acquires** all the resources it needs and runs to completion, thereby **releas**ing its resources. **Mark that process** as virtually **terminated** and **add** its resources to A.

3. Repeat 1 and 2 until all processes are either virtually terminated (**safe state**), or a deadlock is detected (**unsafe state**)

| Resources Assigned | | | | |
|---|---|---|---|---|
| Processes | Tapes | Plotters | Printers | Toasters |
| A | 3 | 0 | 1 | 1 |
| B | 0 | 1 | 0 | 0 |
| C | 1 | 1 | 1 | 0 |
| D | 1 | 1 | 0 | 1 |
| E | 0 | 0 | 0 | 0 |
| Total Existing | 6 | 3 | 4 | 2 |
| Total Claimed by Processes | 5 | 3 | 2 | 2 |
| Remaining Unclaimed | 1 | 0 | 2 | 0 |

| Resources Still Needed | | | | |
|---|---|---|---|---|
| Processes | Tapes | Plotters | Printers | Toasters |
| A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 1 | 2 |
| C | 3 | 1 | 0 | 0 |
| D | 0 | 0 | 1 | 0 |
| E | 2 | 1 | 1 | 0 |

# Illustration of a Situation- General

- As per the Algorithm, the process D's requirements are smaller than A.

$$E=\{6,3,4,2\}$$

$$P=\{5,3,2,2\} - \{1,1,0,1\} = \{4,2,2,1\}\}$$

$$A = \{1,0,2,0\} + \{1,1,0,1\} = \{2,1,2,1\}$$

- Now, A's requirements are less than A. So, do the same thing with A:

$$P = \{4,2,2,1\} - \{3,0,1,1\} = \{1,2,1,0\}$$

$$A=\{2,1,2,1\} + \{3,0,1,1\} = \{5,1,3,2\}$$

- At this point, we see that there are no remaining processes that can't be satisfied from available resources. So the illustrated state is **safe.**

| Resources Assigned | | | | |
|---|---|---|---|---|
| Processes | Tapes | Plotters | Printers | Toasters |
| A | 3 | 0 | 1 | 1 |
| B | 0 | 1 | 0 | 0 |
| C | 1 | 1 | 1 | 0 |
| D | 1 | 1 | 0 | 1 |
| E | 0 | 0 | 0 | 0 |
| Total Existing | 6 | 3 | 4 | 2 |
| Total Claimed by Processes | 5 | 3 | 2 | 2 |
| Remaining Unclaimed | 1 | 0 | 2 | 0 |

| Resources Still Needed | | | | |
|---|---|---|---|---|
| Processes | Tapes | Plotters | Printers | Toasters |
| A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 1 | 2 |
| C | 3 | 1 | 0 | 0 |
| D | 0 | 0 | 1 | 0 |
| E | 2 | 1 | 1 | 0 |



# Banker's Algorithm for Deadlock Avoidance

- 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.



# Banker's Algorithm for Deadlock Avoidance

- Resource – Request Algorithm
  - Verifies if the requested resource can be safely granted.
- Safety Algorithm
  - To check whether or not a system is in a safe state.

# Banker's Algorithm for Deadlock Avoidance

**Data Structures:**

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.

- *Allocation:* *n* x *m* matrix.

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

- *Max: n x m* matrix.

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

- *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.

| Available | | Allocation | Max | Need |
|---|---|---|---|---|
| A B C | | A B C | A B C | A B C |
| 3 3 2 | $P_0$ | 0 1 0 | 7 5 3 | 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 |

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

# Banker's Algorithm for Deadlock Avoidance

## Resource Request Algorithm

*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 (error?). Otherwise, **raise error** condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3 (min?). 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: (safe?)

$Available = Available - Request_i;$
$Allocation_i = Allocation_i + Request_i;$
$Need_i = Need_i - Request_i;;$

| Available | | Allocation | Max | Need |
|---|---|---|---|---|
| A B C | | A B C | A B C | A B C |
| 3 3 2 | $P_0$ | 0 1 0 | 7 5 3 | 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 |

- *If safe ⇒ the resources are **allocated** to $P_i$. (Run the Safety Algorithm)*
- *If unsafe ⇒ $P_i$ **must wait**, and the old resource-allocation state is restored*

# Banker's Algorithm for Deadlock Avoidance

## Safety Algorithm

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

   *m = No. of Resources  n = No. of Processes*

   *Work = Available*

   *Finish* [*i*] = *false* for *i* - 1,2,3, …, *n*.

2. Find an *i* (process) 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.

| Available | | Allocation | Max | Need |
|---|---|---|---|---|
| A B C | | A B C | A B C | A B C |
| 3 3 2 | $P_0$ | 0 1 0 | 7 5 3 | 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 |

# Example of Banker's Algorithm

- 5 processes $P_0$ through $P_4$;
- 3 resource types *A* (10 instances), *B* (5instances, and *C* (7 instances).

| Available | | Allocation | | | Max | | | Need | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | | A | B | C | A | B | C | A | B | C |
| 3 | 3 | 2 | $P_0$ | 0 | 1 | 0 | 7 | 5 | 3 | 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 |

Snapshot at time $T_0$

**Safe?** $< P_1, P_3, P_4, P_2, P_0 >$

# Example of Banker's Algorithm

**Suppose $P_1$ now requests( 1,0,2)** Available = (3,3,2)

Check that Request ≤ Available ? (1,0,2) ≤ (3,3,2) ⇒ ***true.***

| Available | | Allocation | | | Max | | | Need | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | | A | B | C | A | B | C | A | B | C |
| 2 | 3 | 0 | $P_0$ | 0 | 1 | 0 | 7 | 5 | 3 | 7 | 4 | 3 |
| 3 | 3 | 2 | $P_1$ | 3 | 0 | 2 | 3 | 2 | 2 | 0 | 2 | 0 |
| | | | $P_2$ | 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 |

- Executing safety algorithm

  shows that sequence **$<P_1, P_3, P_4, P_0, P_2>$** satisfies safety requirement.

  - Can request for (3,3,0) by $P_4$ be granted?
  - Can request for (0,2,0) by $P_0$ be granted?