# UNIT 5

# Resource Sharing & Management

## Introduction

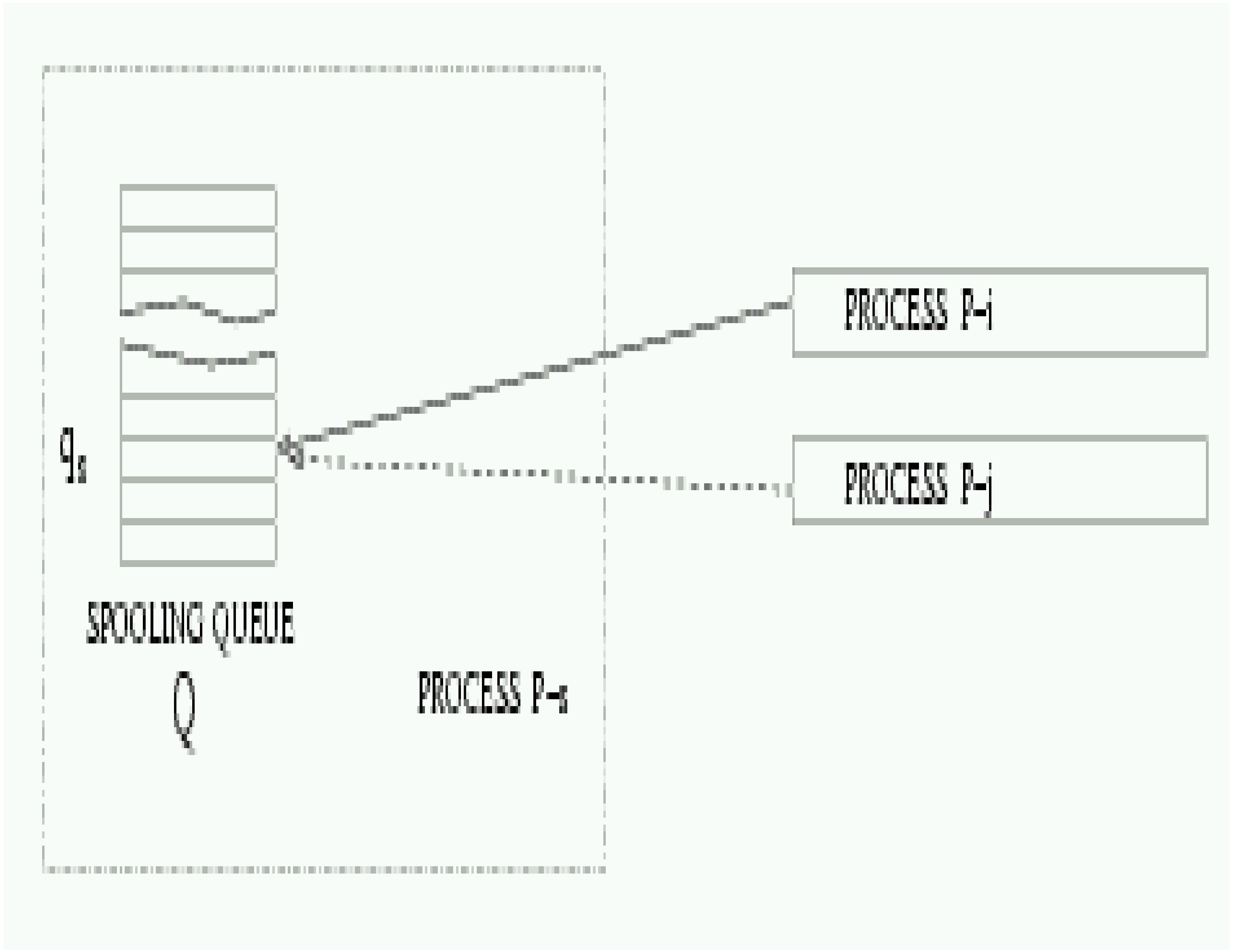
- Some of the resources connected to a computer system (*image processing resource) may be expensive*.

- These resources may be *shared among users or processes*.

# MUTUAL EXCLUSION

- Each resource can be assigned to at most one process only.
- Mutual Exclusion is required in many situations in the OS design.
- Consider the context of *management of a print request queue*.
- Processes that need to print a file, deposit the *file address into* this queue. Printer spooler process picks the file address from this queue to print files.

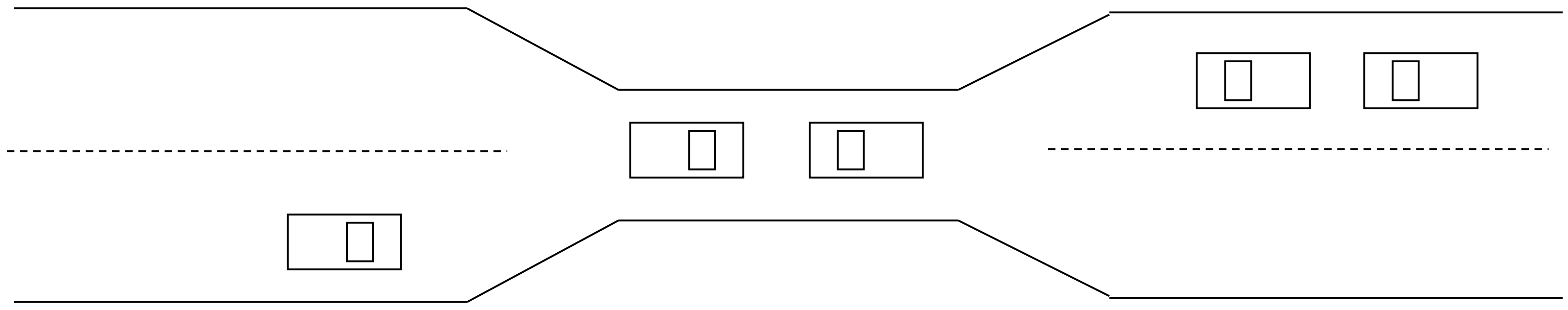
# Mutual Exclusion



## Mutual Exclusion

- Both processes *Pi and Pj think their print jobs* are spooled.
- *Q can be considered as a shared memory area between* processes *Pi, Pj and Ps*.
- *Inter Process Communication can be established* between processes that need printing and that which does printing.

## 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 and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

## 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 (If the request cannot be granted immediately[for e.g. if the resource is being used by another process], then the requesting process must wait until it can acquire the resource)
    - Use (The process can operate on the resource[for e.g. if the resource is a printer, the process can print on the printer])
    - Release (The process releases the resource)

## System Model

- The request and release of resources are system calls.

- Examples :-
  1. request () and release () device
  2. open () and close () file
  3. allocate () and free () memory

- Request and release of resources that are not managed by the OS can be accomplished through the wait() and signal() operations on semaphores or through acquisition (demand) and release of a mutux lock.

## DEADLOCK

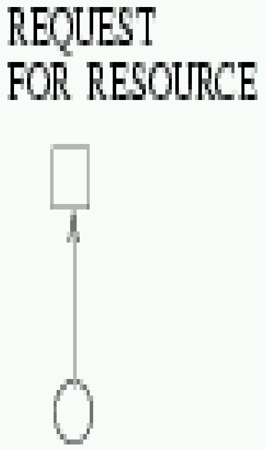
- Consider an example in which process *P1 needs 3* resources *r1, r2 and r3 to make any progress.*
- Similarly, *P2 needs resources r2 and r3.*
- Suppose *P1 gets r1 and r3; P2 gets r3.*
- *P2 is waiting for r2 to be released; P1 is waiting for r3 to* be released …… *deadlock.*
- A *dead-lock is a condition that may involve two or* more processes in a state such that *each is waiting for release of a resource currently held by some other process.*

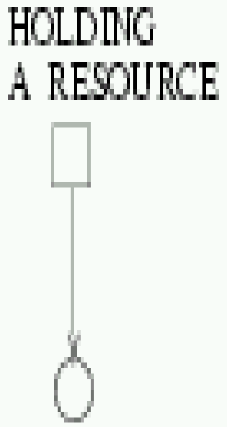
## Deadlock

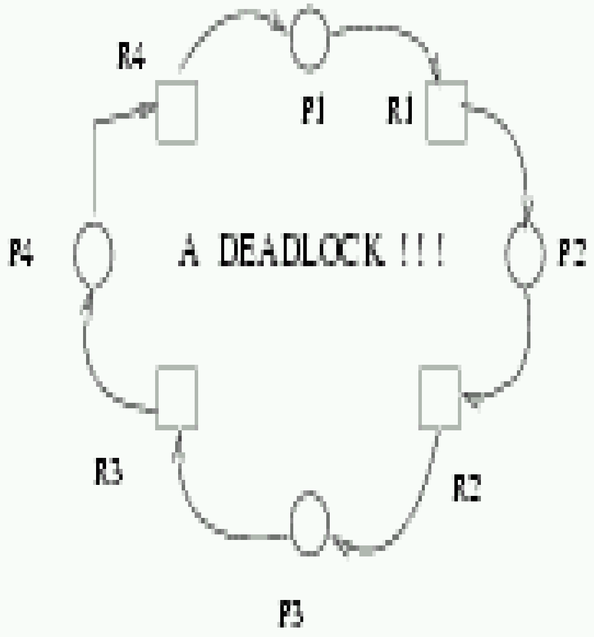
A PROCESS IS DENOTED BY A CIRCLE ◯ A RESOURCE IS DENOTED BY A SQUARE □

AN EDGE FROM A PROCESS TO A RESOURCE DENOTES A REQUEST FOR A RESOURCE

AN EDGE FROM A RESOURCE TO A PROCESS DENOTES THAT PROCESS HOLDS THE RESOURCE







## Deadlock

- Formally, a deadlock occurs when the following conditions are present simultaneously
  - *Mutual Exclusion(Each resource can be assigned to at most one process only)*
  - *Hold and Wait(Processes hold a resource & may seek an additional resource)*
  - *No preemption(Processes that have been given a resource cannot be pre-empted to release their resources)*
  - *Circular Wait(Every process awaits release of at least one resource held by some other processes)*

## Deadlock

- **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.
- **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 $P_2$, …, $P_{n-1}$ is waiting for a resource that is held by $P_n$, and $P_n$ is waiting for a resource that is held by $P_0$.

## 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.

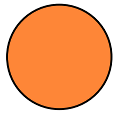
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

- request edge – directed edge $P_1 \rightarrow R_j$

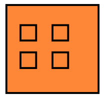
- assignment edge – directed edge $R_j \rightarrow P_i$

## Resource allocation graph

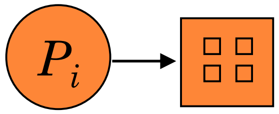
- Process



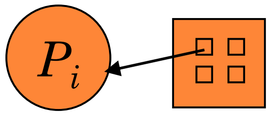
- Resource Type with 4 instances



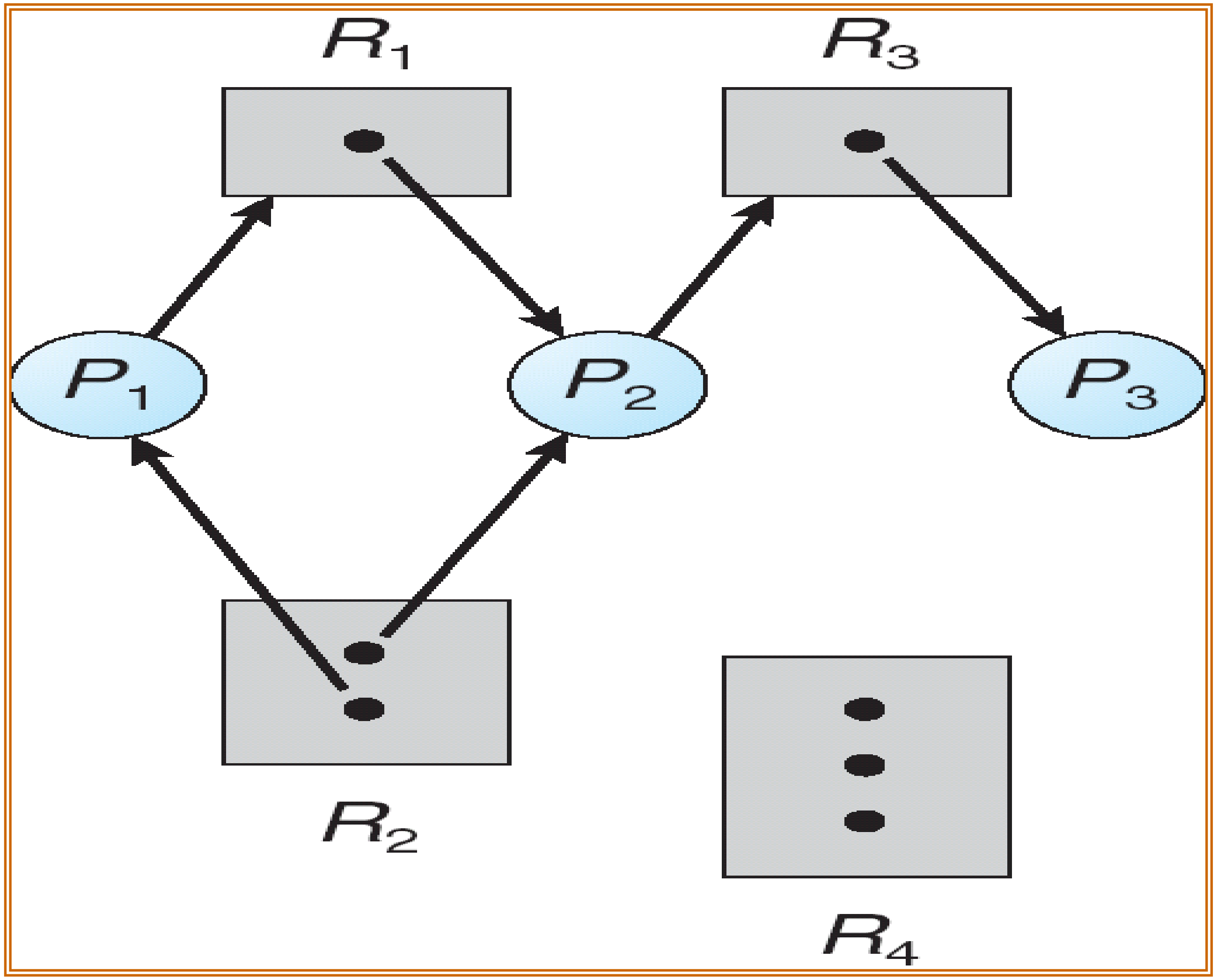
- $P_i$ requests instance of $R_j$



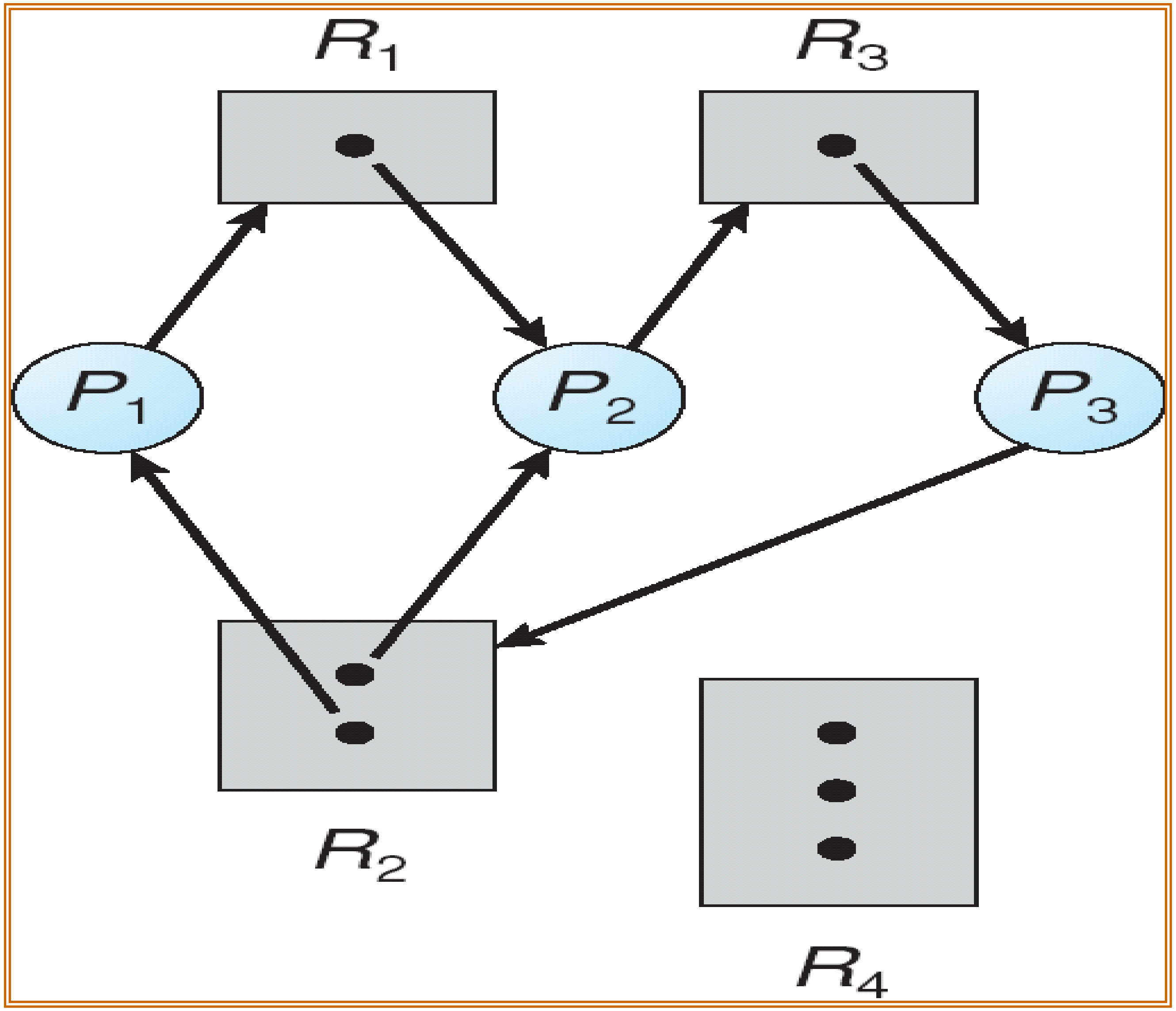
- $P_i$ is holding an instance of $R_j$



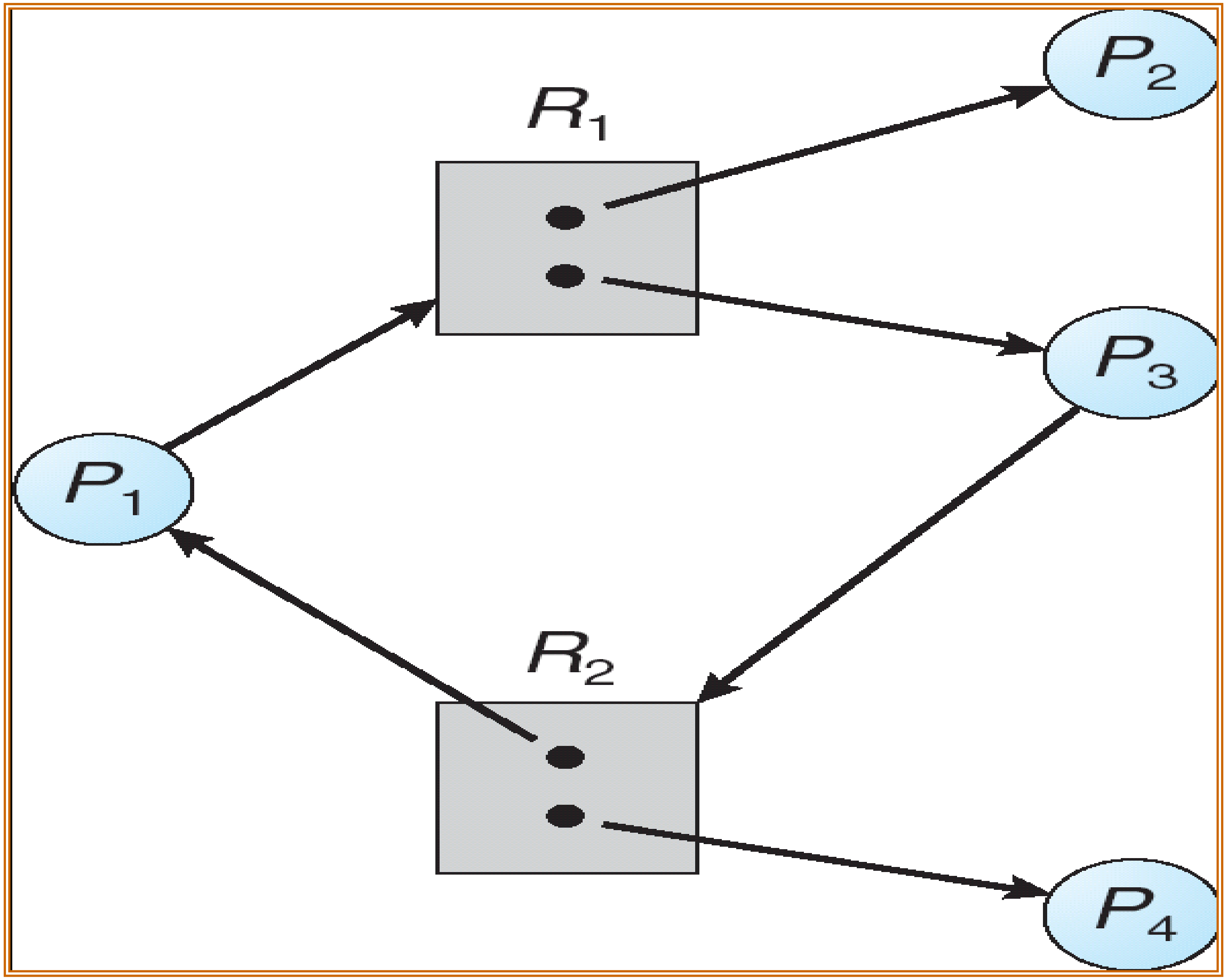
## Resource allocation graph example



## Resource allocation graph with a deadlock



## Resource allocation graph with a cycle but no deadlock



## BASIC FACTS

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

## METHODS FOR HANDLING DEADLOCK

- Ensure that the system will *never* enter a deadlock state. (<u>Deadlock Prevention or Deadlock Avoidance</u>)
- Allow the system to enter a deadlock state and then recover (<u>Deadlock Detection and Deadlock Recovery</u>).
- 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.

- **Mutual Exclusion** – not required for sharable resources; 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.
    - Low resource utilization; starvation possible.

## 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 Prevention

- We have *multiple resources and processes that can* request *multiple copies of each resource.*
- It is difficult modeling this as a *graph*.
- We use the *matrix method to model this scenario.*

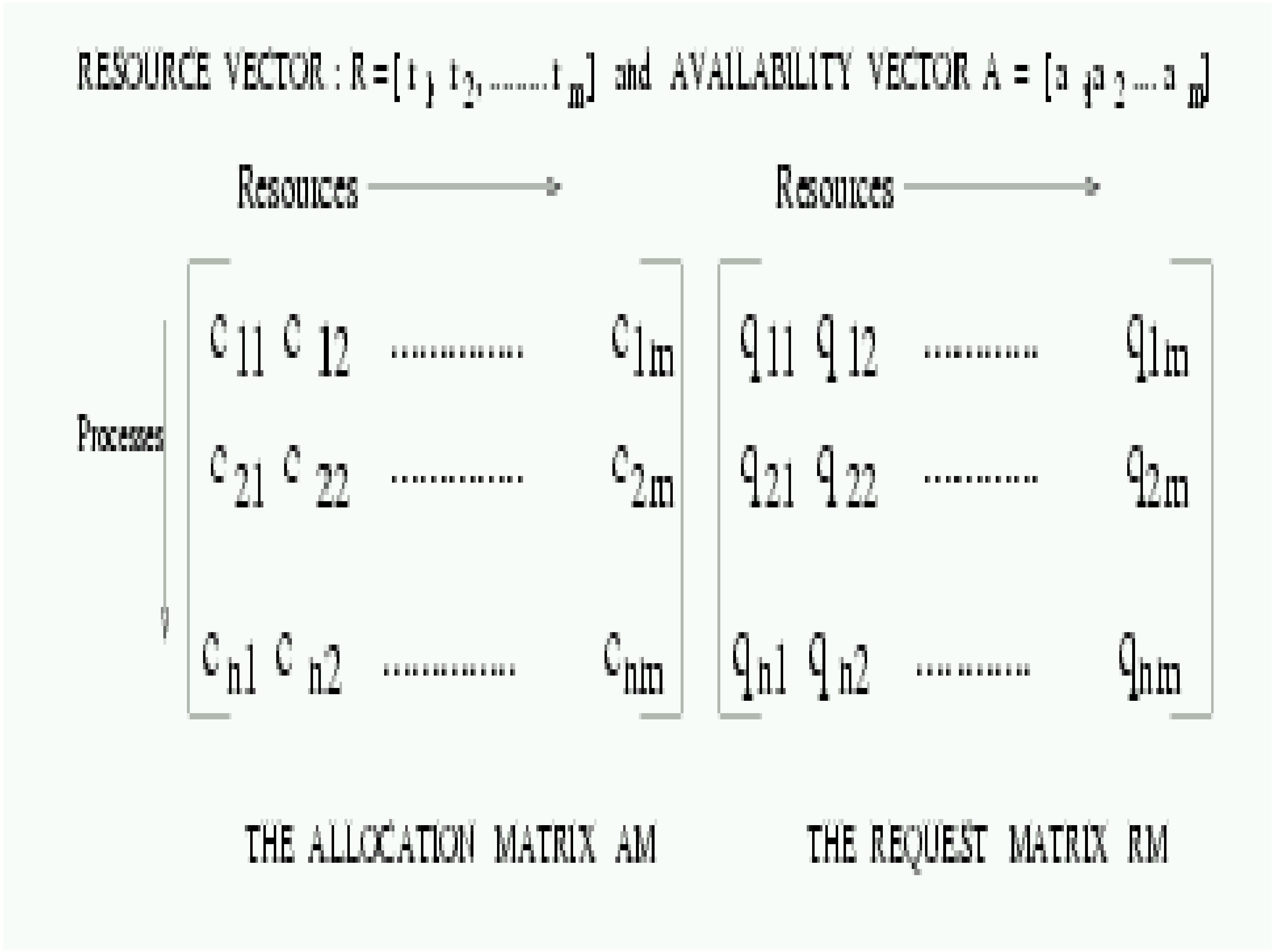
## Deadlock Prevention

- Assume *n processes and m kinds of resources.*
- We denote the *ith resource with ri.*
- We define 2 vectors each of size m,

  *Vector R = (r1, r2,……., rm)*

  *Vector A = (a1, a2,……, am) where ai is the resource of* type *i available for allocation.*
- We define 2 matrices for *allocations made (AM) and the requests pending for resources (RM).*

# Matrix model of Requests and Allocation

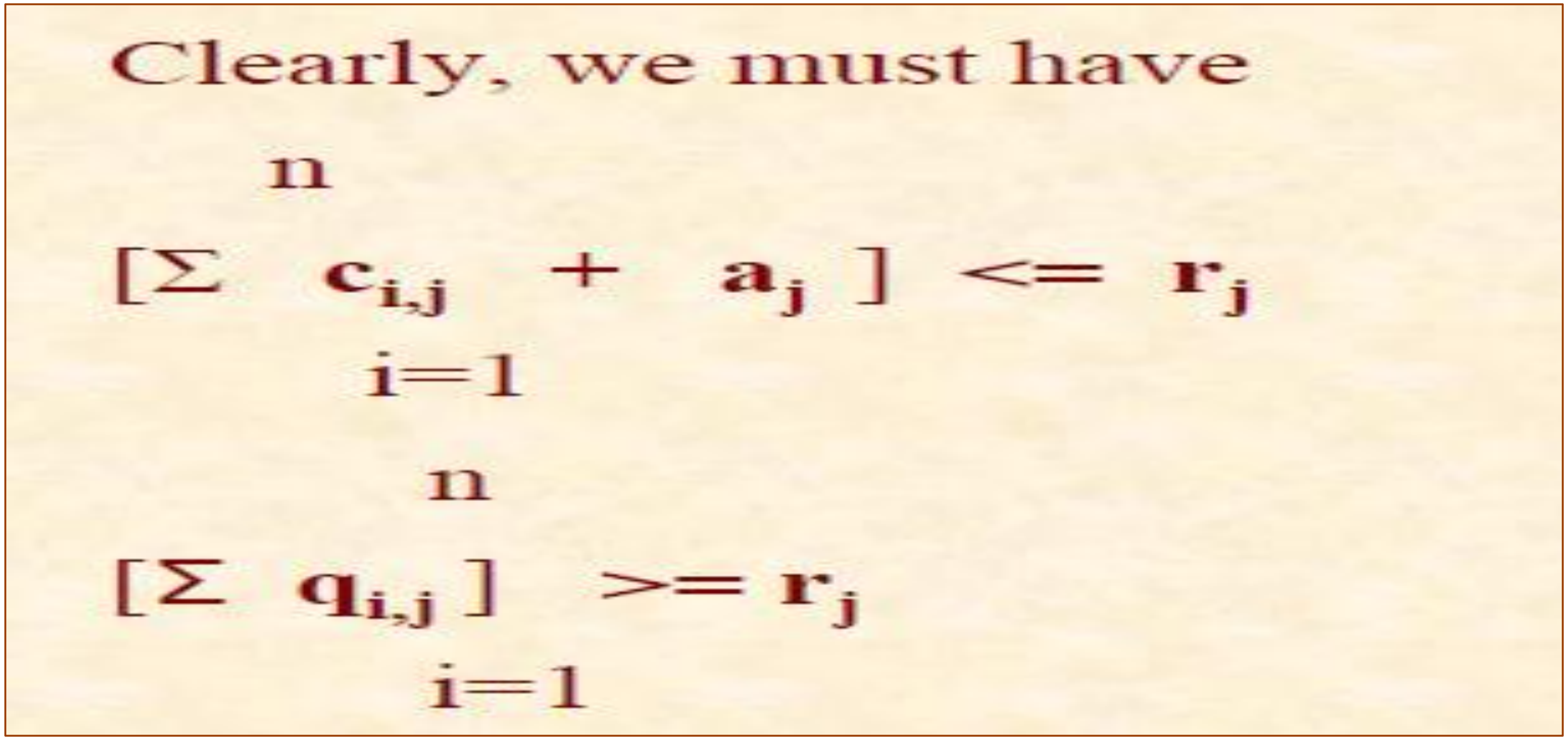


RESOURCE VECTOR : $R = [r_1, r_2, \ldots\ldots r_m]$ and AVAILABILITY VECTOR $A = [a_1, a_2 \ldots a_m]$

$$\begin{array}{c|c}
 & \text{Resources} \longrightarrow \\
\text{Processes} \downarrow & \begin{bmatrix} c_{11} & c_{12} & \cdots\cdots & c_{1m} \\ c_{21} & c_{22} & \cdots\cdots & c_{2m} \\ & & & \\ c_{n1} & c_{n2} & \cdots\cdots & c_{nm} \end{bmatrix}
\end{array}
\qquad
\begin{array}{c}
\text{Resources} \longrightarrow \\
\begin{bmatrix} q_{11} & q_{12} & \cdots\cdots & q_{1m} \\ q_{21} & q_{22} & \cdots\cdots & q_{2m} \\ & & & \\ q_{n1} & q_{n2} & \cdots\cdots & q_{nm} \end{bmatrix}
\end{array}$$

THE ALLOCATION MATRIX AM THE REQUEST MATRIX RM

## Deadlock prevention



Clearly, we must have

$$[\sum_{i=1}^{n} c_{i,j} + a_j] <= r_j$$

$$[\sum_{i=1}^{n} q_{i,j}] >= r_j$$

## 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.

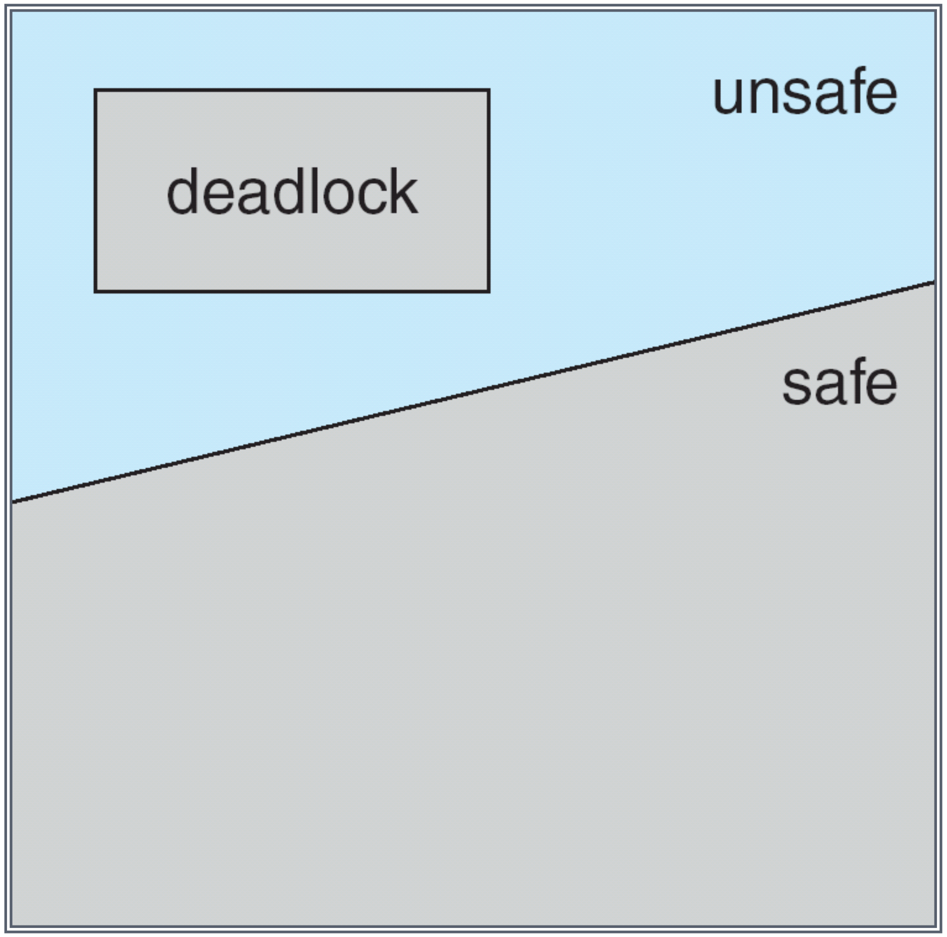
- System is in safe state if there exists a safe sequence of all processes.

- Sequence <$P_1$, $P_2$, …, $P_n$> is safe if 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.
  - 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.

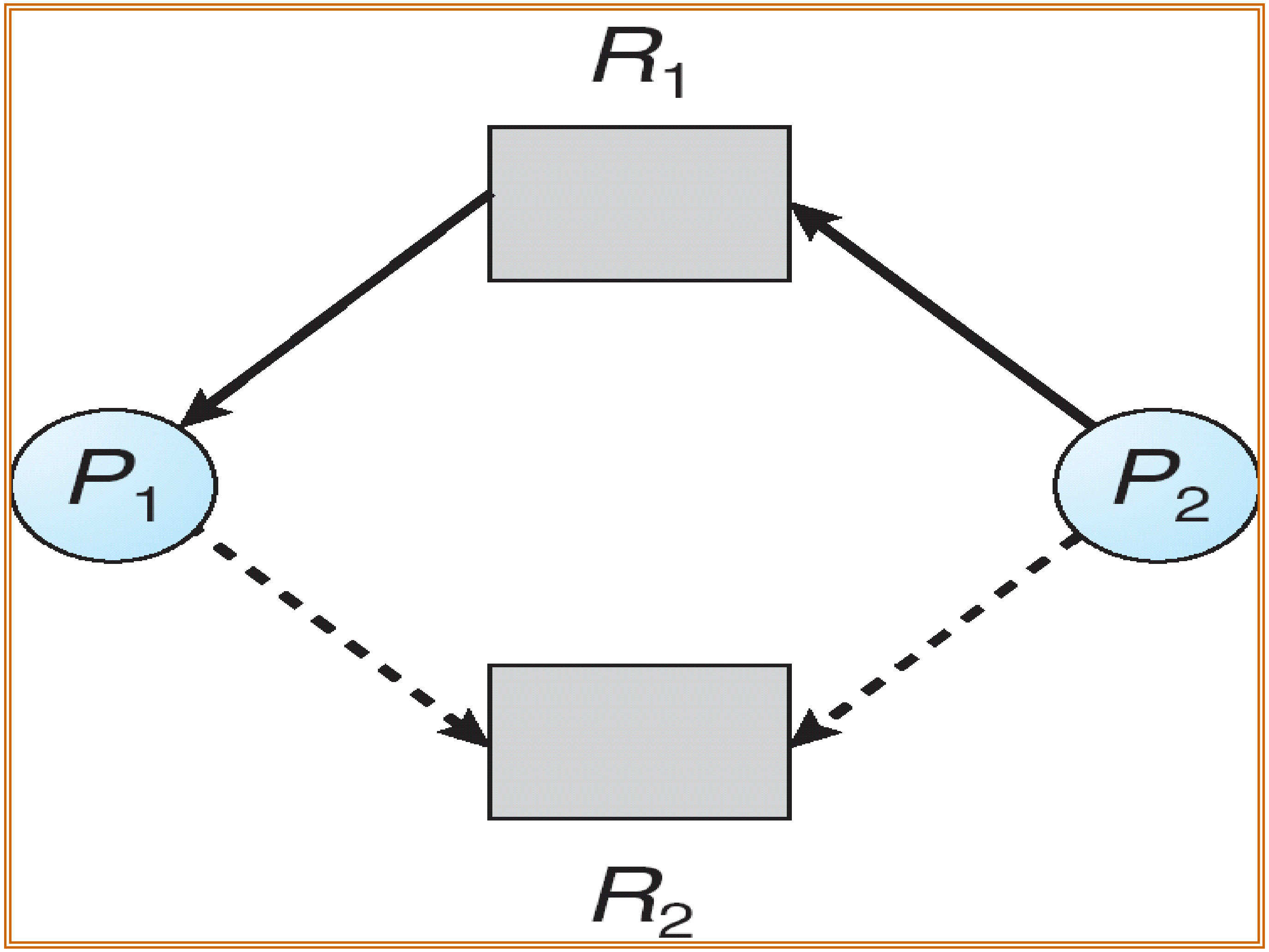
## Safe, Unsafe , Deadlock State



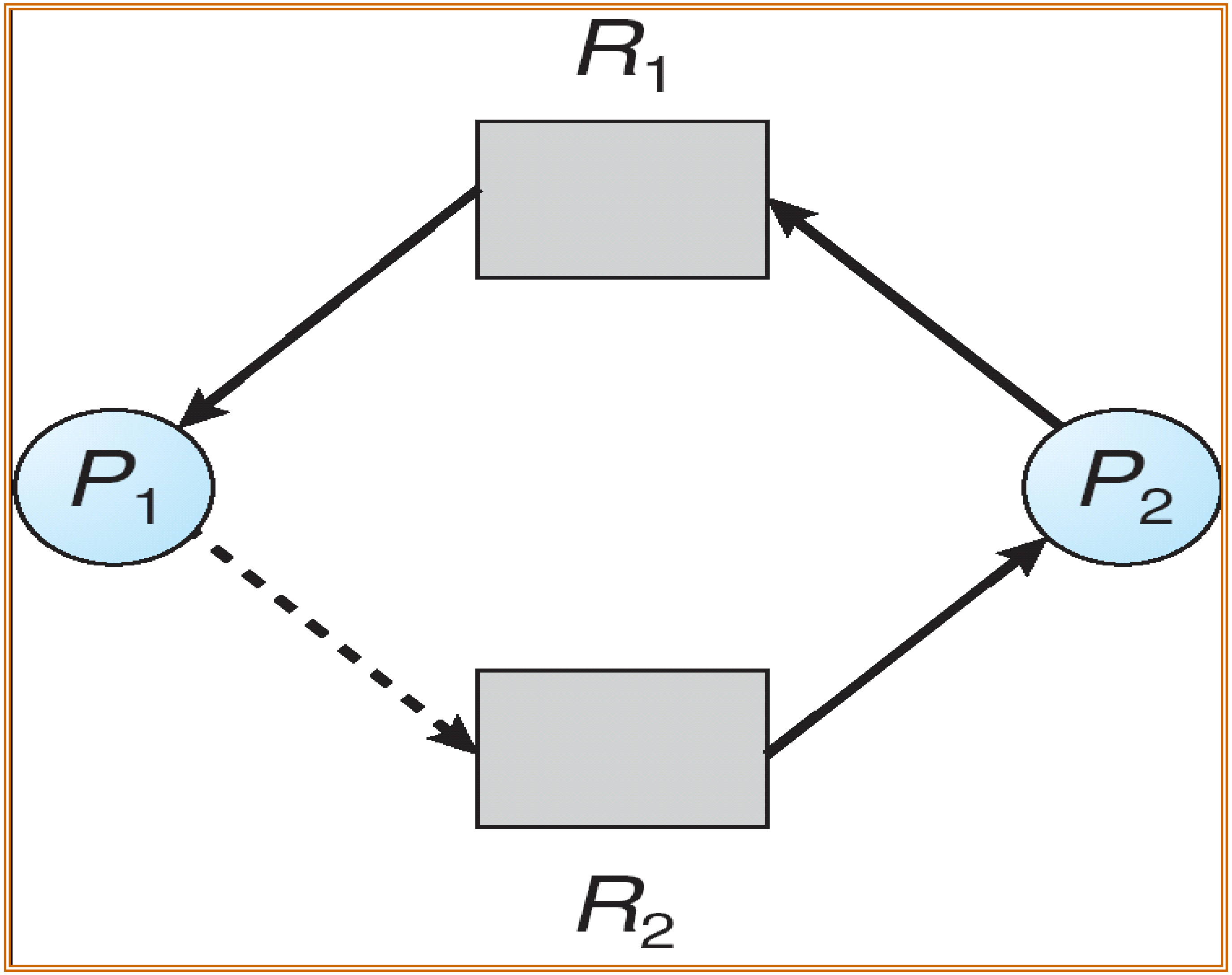
## Resource-Allocation Graph Algorithm

- Claim edge $P_i \rightarrow$ Rj indicated that process Pi may request resource Rj; 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.

## Resource-Allocation Graph For Deadlock Avoidance



## Unsafe State In Resource-Allocation Graph



## Dead-lock Avoidance

- Conditions for dead-lock to occur are mutual exclusion, hold and wait, no preemption and circular wait.

- The first 3 conditions for dead-lock are necessary conditions. Circular Wait implies Hold and Wait.

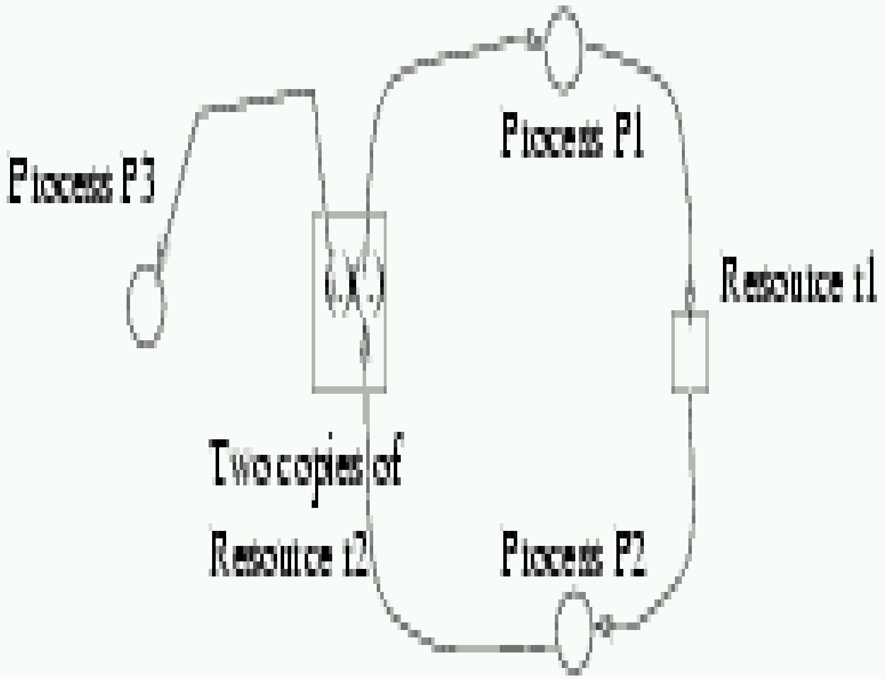
- How does one avoid having a dead-lock??

## Infinite Resource Argument

- One possibility is to have multiple resources of the same kind.
- Sometimes, we may be able to break a dead-lock by having a few additional copies of a resource.
- When one copy is taken, there is always another copy of that resource.

## Infinite Resource Argument

A PROCESS IS DENOTED BY A CIRCLE ◯ A RESOURCE IS DENOTED BY A SQUARE ◻



Process P1 has one r2 and requests r1

Process P2 has r1 and request r2

Process P3 has r2, which it will release on completion

The deadlock is broken when P3 terminates.

## Infinite Resource Argument

- The pertinent question is, how many copies of each resource do we need??

- Unfortunately, theoretically, we need infinite number of resources!!!

- In the example, if P3 is deadlocked, the deadlock between P1 and P3 cannot be broken.

## Never let the conditions occur

- It takes 4 conditions for dead-lock to occur. This dead-lock avoidance simply states do not let conditions occur:
- *Mutual exclusion - unfortunately many resources* require many exclusion!!
- *Hold and Wait - since this is implied by Circular Wait,* we may possibly avoid Circular Wait.
- *Preemption - may not be the best policy to avoid deadlock* but works and is clearly enforceable in many situations.

## 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.

## Bankers Algorithm

- This algorithm based on resource denial if there is a suspected risk of a deadlock.
- A request of a process is assessed if the process resources can be met from the available resources RMi,j <= ai for all j.
- Once the process is run, it shall return all the resources it held.

## Bankers Algorithm

- Note that Banker's Algorithm makes sure that only processes that will run to completion are scheduled to run.
- However, if there are deadlocked processes, the will remain deadlocked.
- Banker's Algorithm does not eliminate a deadlock.

## BANKERS ALGORITHM

- Banker's Algorithm makes some unrealistic assumptions– resource requirements for processes is known in advance.

- The algorithm requires that there is no specific order in which the processes should be run.

- It assumes that there is a fixed number of resources available on the system.

## A Graph Based Detection Algorithm

- In the digraph model with one resource of one kind, we are required to detect a directed cycle in a processor resource digraph.

- For each process, use the process node as root and traverse the digraph in depth first mode marking the nodes. If a marked node is revisited, deadlock exists.

## BANKERS ALGORITHM

- Consider a process Pi and its corresponding row in matrix RM.

- If vector RM <= A then every resource request of process Pi can be met from the available set of resources.

- On completion, this process can return its current allocation in row AMi for another process.

## Bankers Algorithm

- Deadlock detection algorithm is in the following steps :
- *Step 0* : *Assume that all processes are unmarked initially.*
- *Step 1*: *While there are unmarked processes, choose an* unmarked process with RMi <= A. process Step 2 else go to Step 3.
- *Step 2* : *Add row AMi to A and mark the process.*
- *Step 3* : *If there is no such process the algorithm terminates.*
- If all processes are marked, no deadlock.
- If there is a set of processes that remain unmarked, then this set of processes have a deadlock.

## Bankers Algorithm

- Note that notwithstanding the non-deterministic nature of the algorithm it always detects a deadlock.
- The method detects a deadlock if present; it does not eliminate a deadlock.
- Deadlock elimination may require pre-emption or release of resources.

## 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.

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

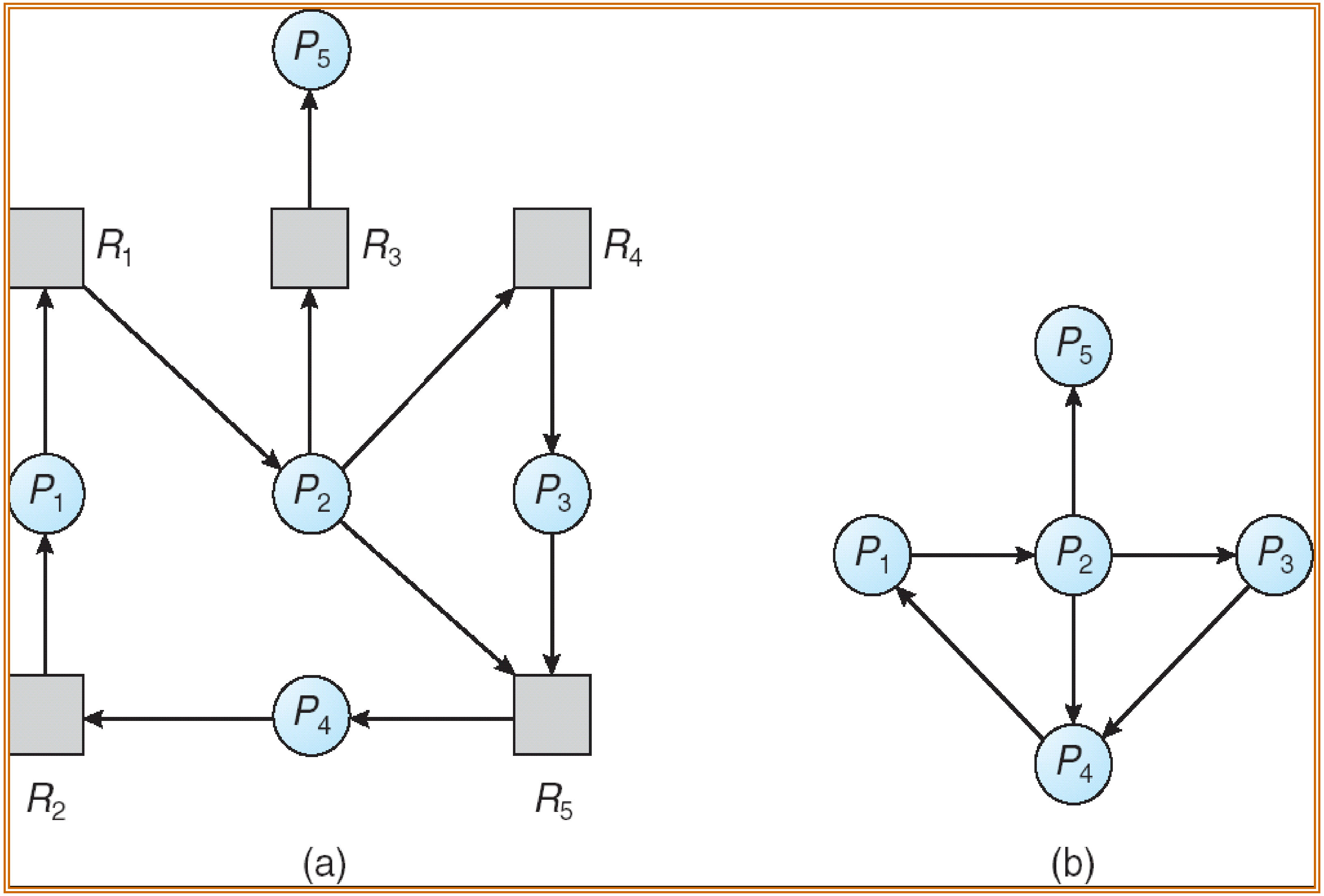
## Deadlock Detection

- 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.
-

## 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. $R_j$.

## 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.