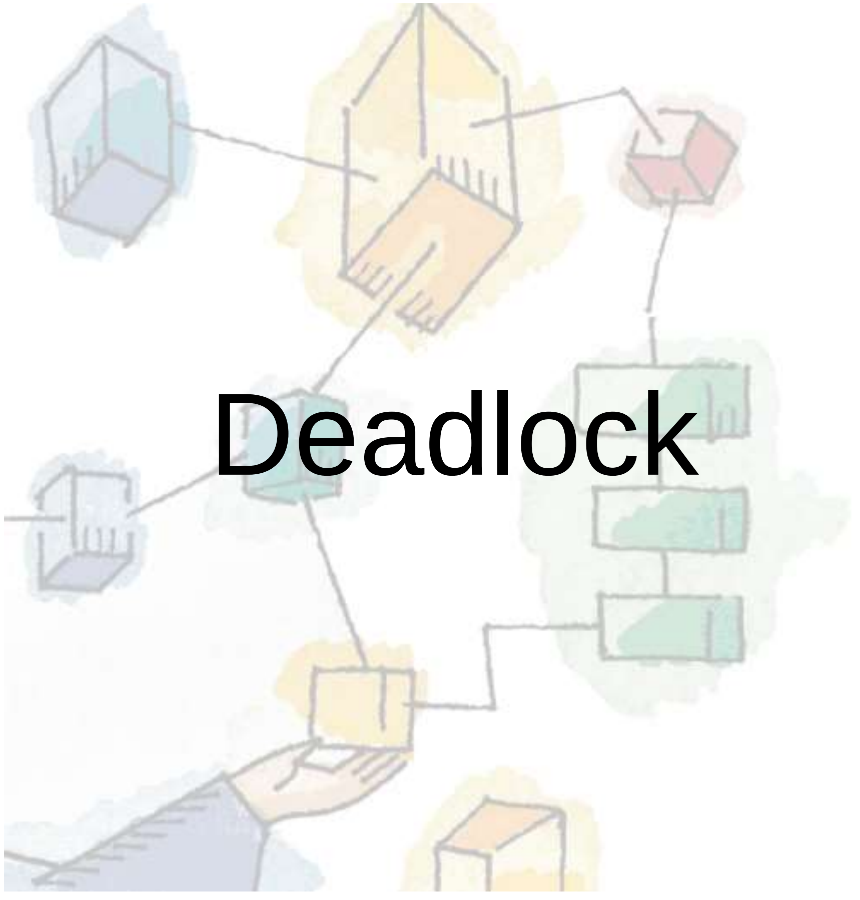


# Deadlock

- Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Dining philosopher's problem

# Deadlock

- Permanent blocking of a set of processes that compete for system resources
- A set of processes is deadlocked when each of the process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.
- Involve conflicting needs for resources by two or more processes

# Deadlock

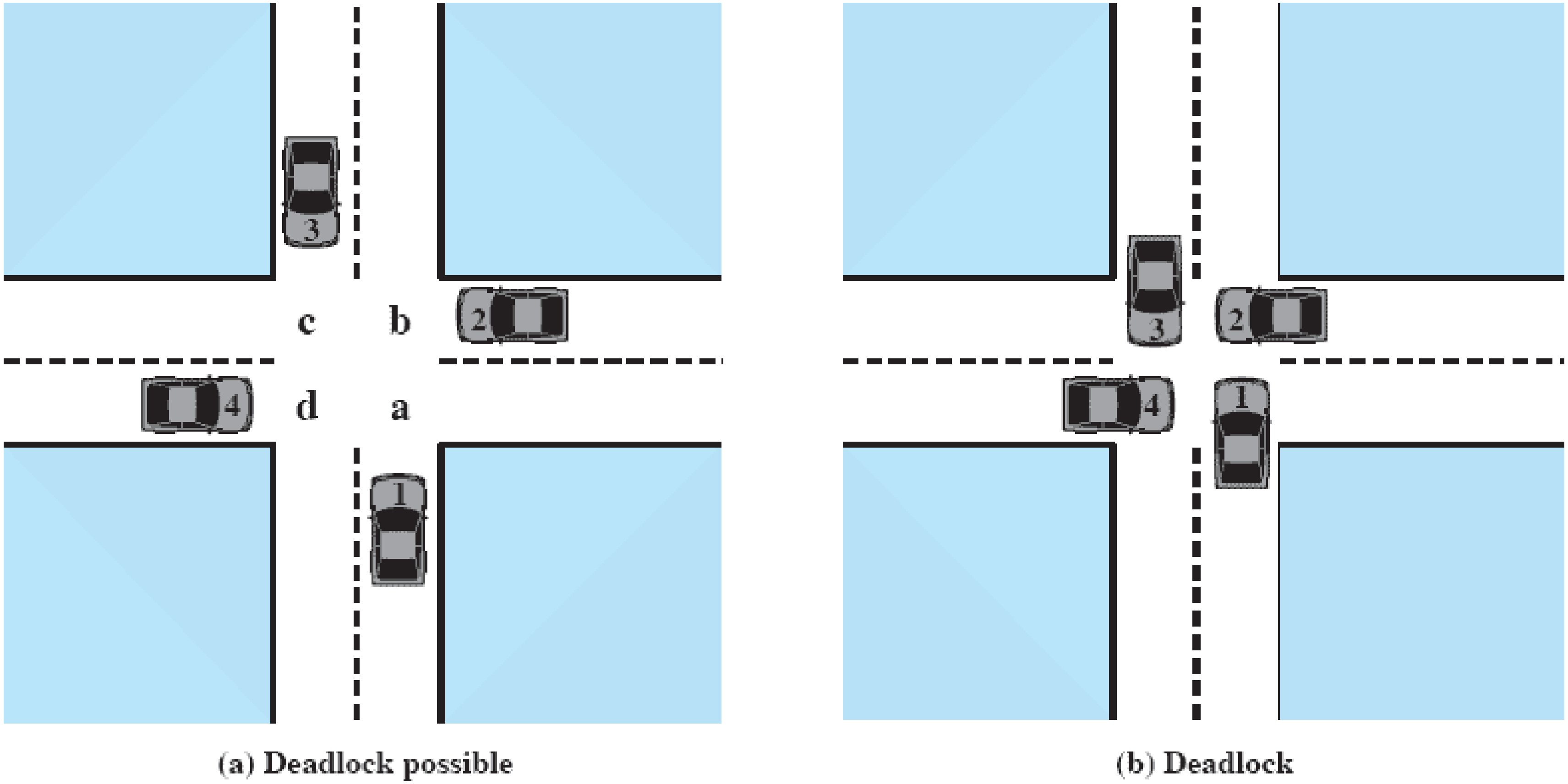



Figure 6.1 Illustration of Deadlock

- Under normal mode of operation, a process may utilize a resource in following sequence:

1. Request
2. Use
3. Release

**System Table**

→ Records whether each resource is free or allocated

→If allocated then to which process

→If a process requests a resource currently allocated to another process then it is added to a queue of processes waiting for that resource.

# Consumable Resources

- Created (produced) and destroyed (consumed)
- There is no limit on no of consumable resources of a particular type.
- Information in I/O buffers, Interrupts, signals, messages

# Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases
- Deadlock occurs if each process holds one resource and requests the other

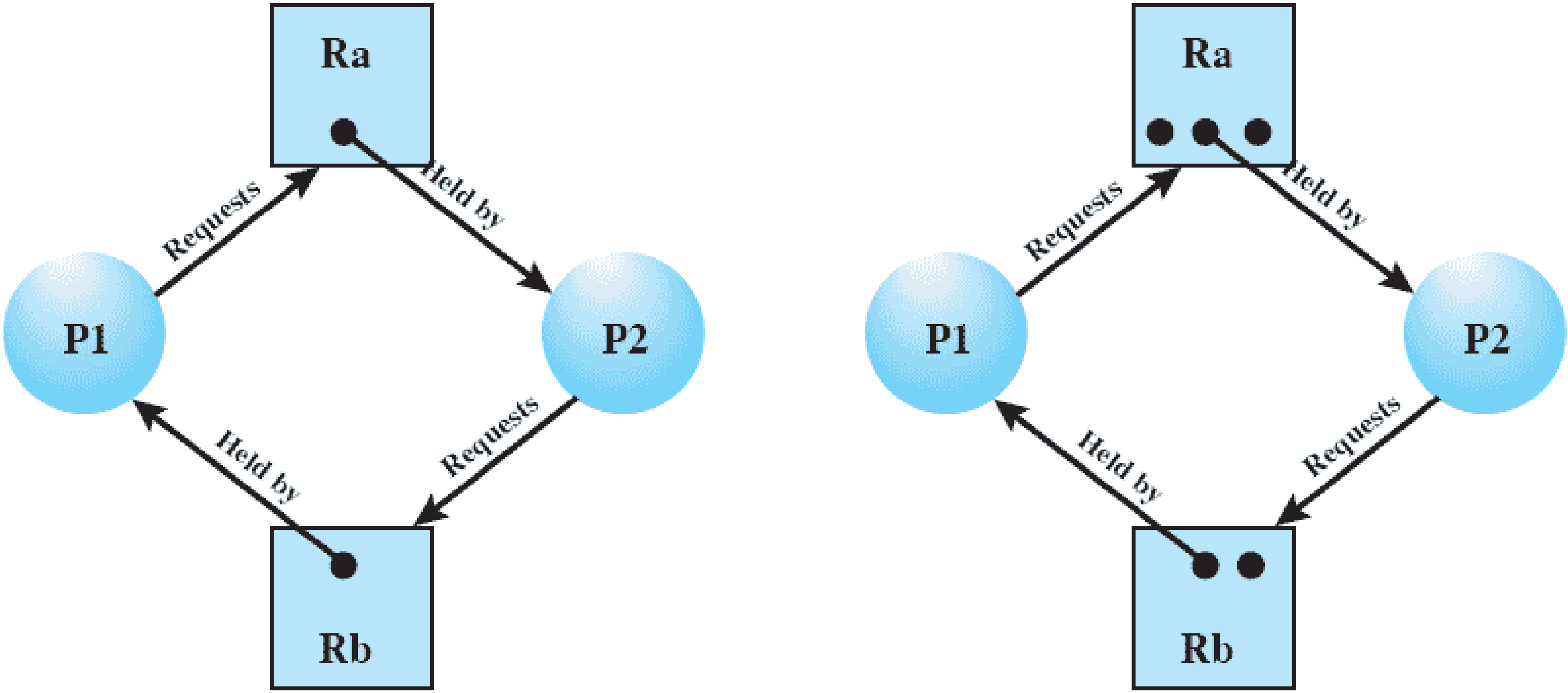
# Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes


(a) Resouce is requested

(b) Resource is held

# Resource Allocation Graphs




(c) Circular wait

(d) No deadlock

# Resource Allocation Graphs

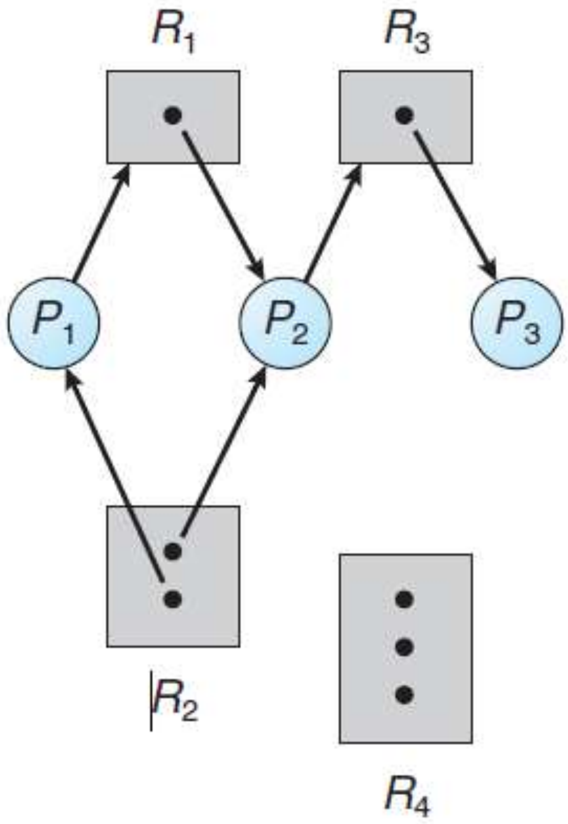



**Figure 7.1** Resource-allocation graph.

# Resource Allocation Graphs

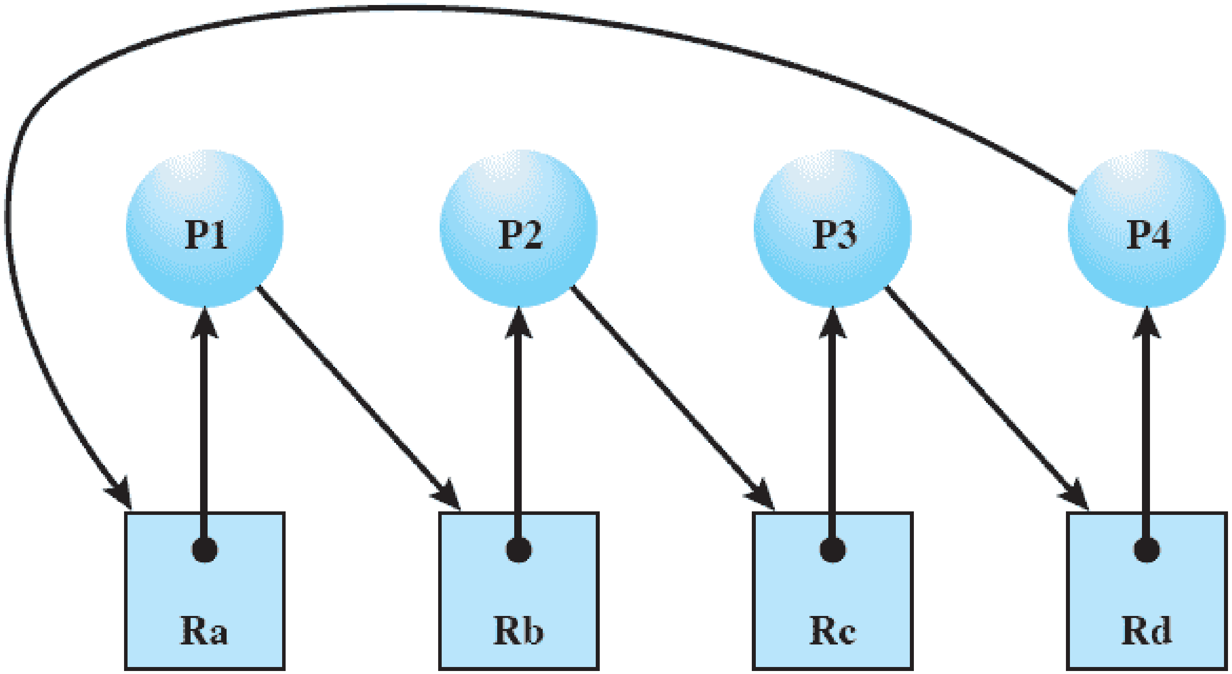



Figure 6.6 Resource Allocation Graph for Figure 6.1b

# Conditions for Deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others

# Conditions for Deadlock

- No preemption
  - No resource can be forcibly removed form a process holding it
- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

# Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait

# Existence of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

# Handling Deadlocks

- Prevent
- Avoid
- Detect

# Deadlock Prevention

## Indirect Method

Prevent occurrence of conditions 1 to 3

## Direct Method

Prevent occurrence of condition 4 i.e. Circular Wait

# Deadlock Prevention

- Mutual Exclusion
  - Must be supported by the OS
  - Sharable resources
  - Non Sharable resources

## Hold and Wait

- Require a process request all of its required resources at one time
- A process may request some resources and may use them.
- Before it can request any other resource it must release all the resources that is currently allocated.
- Disadvantages
  - Starvation
  - Low resource utilization
  - No prior knowledge of resources required

## No Preemption

- Process must release resource and request again
- OS may preempt a process to require it releases its resources

# Circular Wait

- Define a linear ordering of resource types
- Circular wait can be prevented by defining a linear ordering of resource types and to require that each process requires resource in an increasing order of enumeration.

# 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
- Deadlock Avoidance Algorithms –
  - Safe State Algorithm
  - Banker's Algorithm

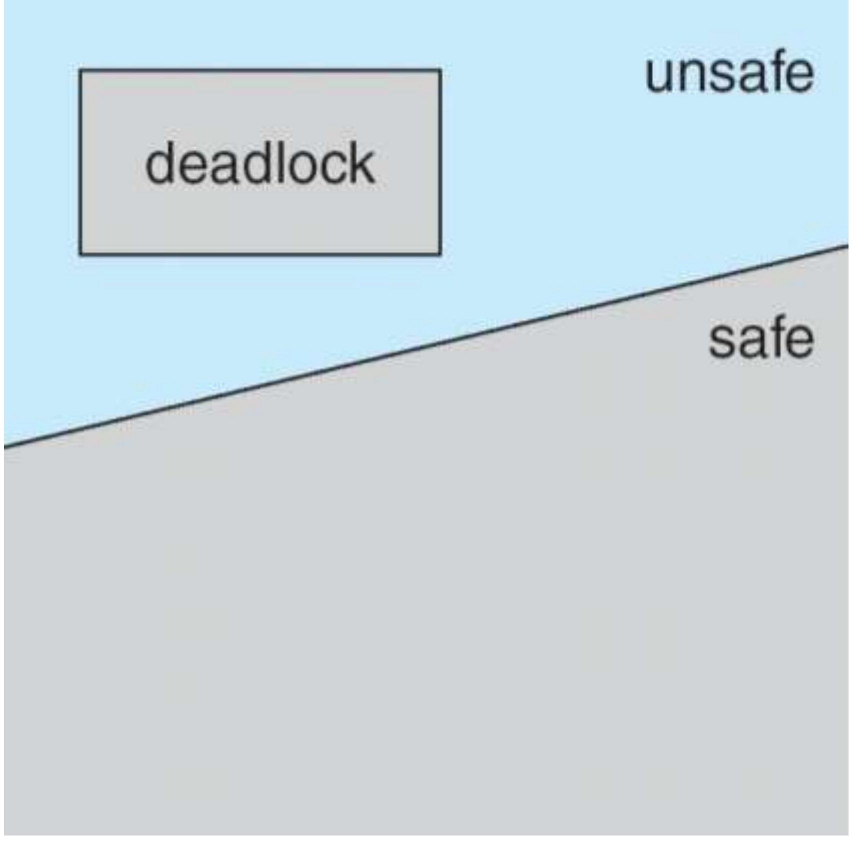
# Safe State

- 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 sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems 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 $j < I$
- That is:
  - 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



# Safe State

- Example : Consider 3 processes P1, P2, P3 and one resource R1 with 12 instances

| Process | Max. Need | Current Allocation | Need | Available (After process execution) |
|---|---|---|---|---|
| P1 | 10 | 5 | 5 | 10 |
| P2 | 4 | 2 | 2 | 5 |
| P3 | 9 | 2 | 7 | 12 |
| **Total :** | **23** | **9** | | |
| **Free resources :** | | **3** | | |

Free resources = Total resources-current allocation = 12-9 = 3

Safe sequence = <P2, P1, P3>

Available = Allocated free resources + current allocation

# 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

# 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**: $n \times m$ matrix. If $Max\ [i,j] = k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$

- **Allocation**: $n \times m$ matrix. If Allocation$[i,j] = k$ then $P_i$ is currently allocated $k$ instances of $R_j$

- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then $P_i$ may need $k$ more instances of $R_j$ to complete its task

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

# Safety Algorithm

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

   ***Work = Available***

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

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

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

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

   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

# Resource-Request Algorithm for Process $P_i$

***Request**$_i$* = 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. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request**$_i$* $\leq$ ***Available***, go to step 3. 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:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to ***P**$_i$*
- If unsafe $\Rightarrow$ ***P**$_i$* must wait, and the old resource-allocation state is restored

# Example of Banker's Algorithm

- 5 processes $P_1$ through $P_5$;     P2(4,2,1)
- 3 resource types:

  *A* (10 instances), *B* (5 instances), and *C* (7 instances)
- Snapshot at time $T_0$:

| *Need* | | *Allocation* | *Max* | *Available(Work)* |
|---|---|---|---|---|
| *A B C* | | *A B C* | *A B C* | *A B C* |
| 7 4 3 | $P_1$ | 0 1 0 | 7 5 3 | 3 3 2 |
| 1 2 2 | $P_2$ | 2 0 0 | 3 2 2 | |
| 6 0 0 | $P_3$ | 3 0 2 | 9 0 2 | |
| 0 1 1 | $P_4$ | 2 1 1 | 2 2 2 | |
| 4 3 1 | $P_5$ | 0 0 2 | 4 3 3 | |

# Example (Cont.)

- The content of the matrix ***Need*** is defined to be
  ***Need=Max – Allocation***

| | *Need* |
|---|---|
| | *A B C* |
| $P_1$ | 7 4 3 |
| $P_2$ | 1 2 2 |
| $P_3$ | 6 0 0 |
| $P_4$ | 0 1 1 |
| $P_5$ | 4 3 1 |

- The system is in a safe state since the sequence < $P_2$, $P_4$, $P_5$, $P_1$, $P_3$> satisfies safety criteria

# Example of Banker's Algorithm

- 5 processes $P_1$ through $P_5$;
  3 resource types:
  $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances)
- Snapshot at time $T_0$:

| Need | | Allocation | Max | Available(Work) | |
|---|---|---|---|---|---|
| *A B C* | | *A B C* | *A B C* | *A B C* | |
| 7 4 3 | $P_1$ | 0 1 0 | 7 5 3 | 3 3 2 | |
| 1 2 2 | $P_2$ | 2 0 0 | 3 2 2 | 5 3 2 | |
| 6 0 0 | $P_3$ | 3 0 2 | 9 0 2 | 7 4 3 | |
| 0 1 1 | $P_4$ | 2 1 1 | 2 2 2 | 7 4 5 | |
| 4 3 1 | $P_5$ | 0 0 2 | 4 3 3 | 7 5 5 | 10 5 7 |

Available=Total resources-Total allocation
= (10,5,7)-(7,2,5) = (3,3,2)

Safe Sequence = <P2, P4, P5, P1, P3>

# Example: $P_1$ Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

| | *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 | |

- 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?

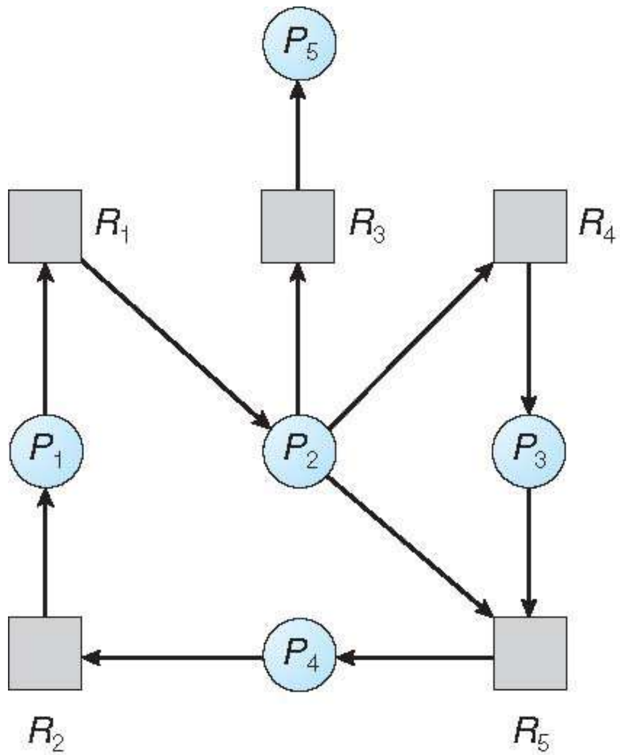
# Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

# Single Instance of Each Resource Type

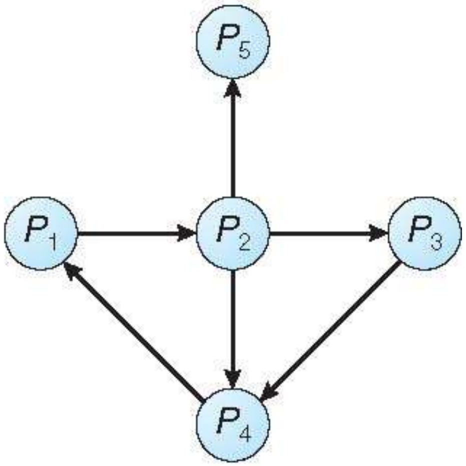
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph

# Resource-Allocation Graph and Wait-for Graph




(a)

Resource-Allocation Graph




(b)

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

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

# Detection Algorithm (Cont.)

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. Moreover, if ***Finish*[*i*] == *false***, then $P_i$ is deadlocked

# 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?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated

# 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

# Dining Philosophers Problem

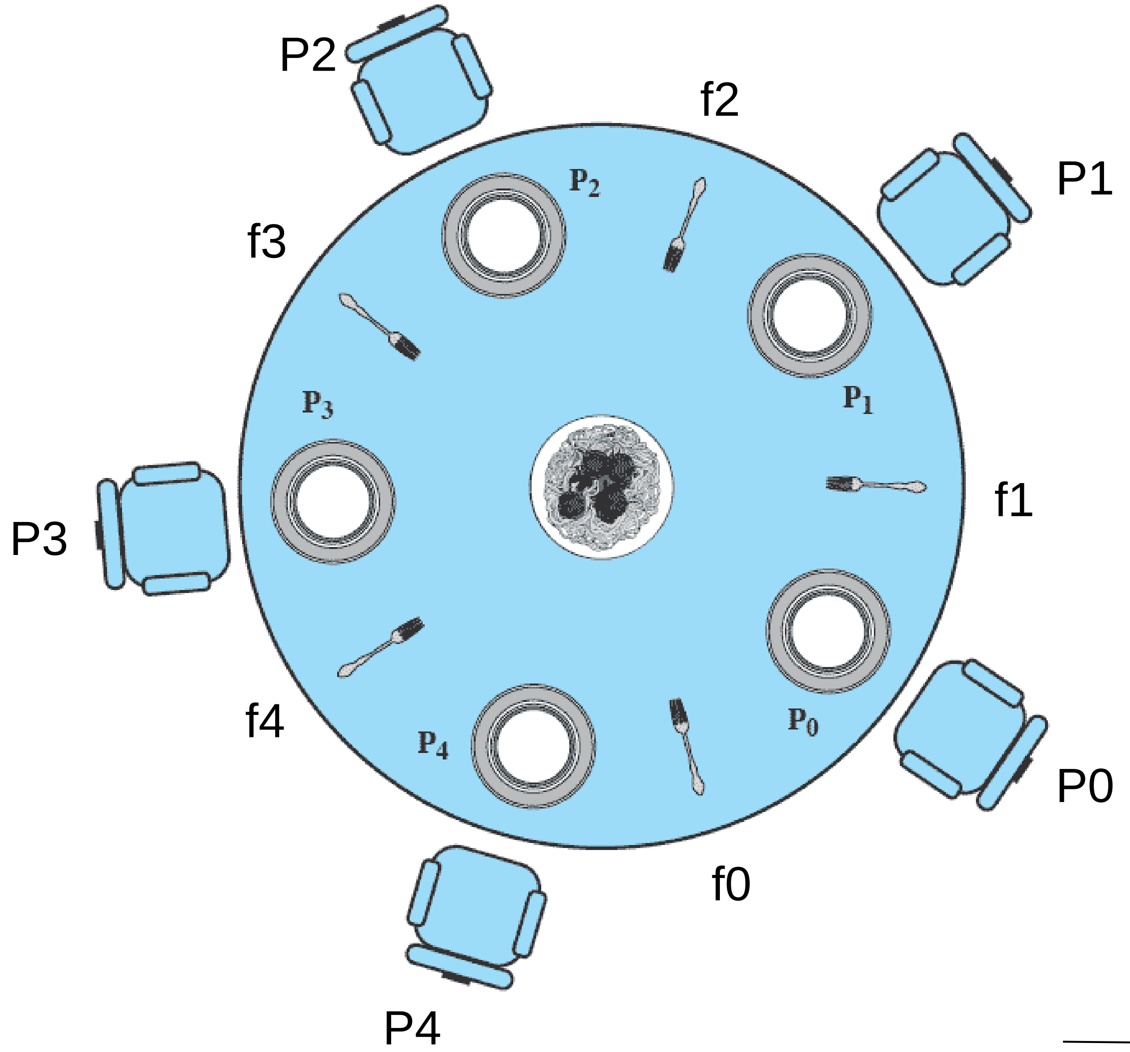



Figure 6.11 Dining Arrangement for Philosophers

# Dining Philosophers Problem

```
semaphore s[5]=1;
void philosopher (void)
{
while(true)
{
Thinking();
wait(take_fork(si));
wait(take_fork((si+1)%N));
Eat();     // Critical Section
signal(put_fork(si));
signal(put_fork((si+1)%N));
}
}
```

| S0 | S1 | S2 | S3 | S4 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| | | | | |
| | | | | |

P0 – f0 , f1 – S0 , S1
P1 - f1 , f2 - S1 , S2
P2 - f2 , f3 - S2 , S3
P3 – f3 , f4 - S3 , S4
P4 - f4 , f0 - S4, S0

Consider the following snapshot of a system
Answer the following using Banker's Algorithm

- What is the content of Need Matrix?
- Is the system in safe state?
- If the request from P1 arrives for (0, 4, 2, 0); can the request be granted immediately?

| | Allocation | | | | Max | | | | Available | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | A | B | C | D | A | B | C | D |
| **P0** | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | 1 | 5 | 2 | 0 |
| **P1** | 1 | 0 | 0 | 0 | 1 | 7 | 5 | 0 | | | | |
| **P2** | 1 | 3 | 5 | 4 | 2 | 3 | 5 | 6 | | | | |
| **P3** | 0 | 6 | 3 | 2 | 0 | 6 | 5 | 2 | | | | |
| **P4** | 0 | 0 | 1 | 4 | 0 | 6 | 5 | 6 | | | | |