# Unit 3 - Process Deadlock

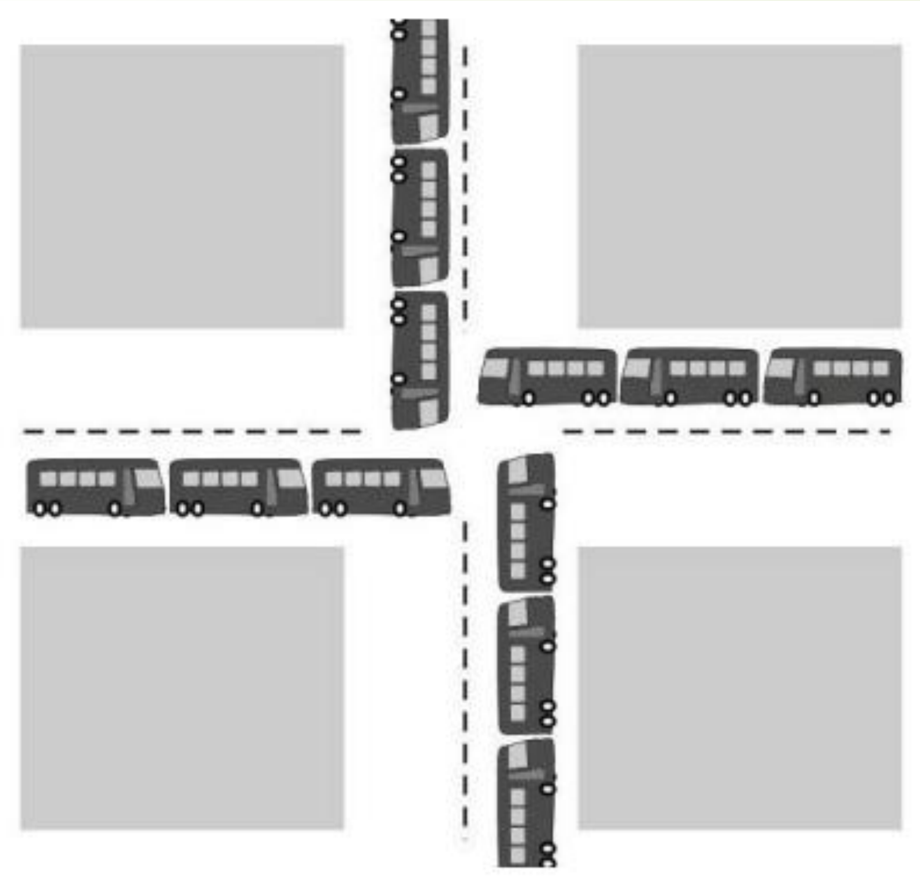
# Introduction

- System consists of resources
- 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**

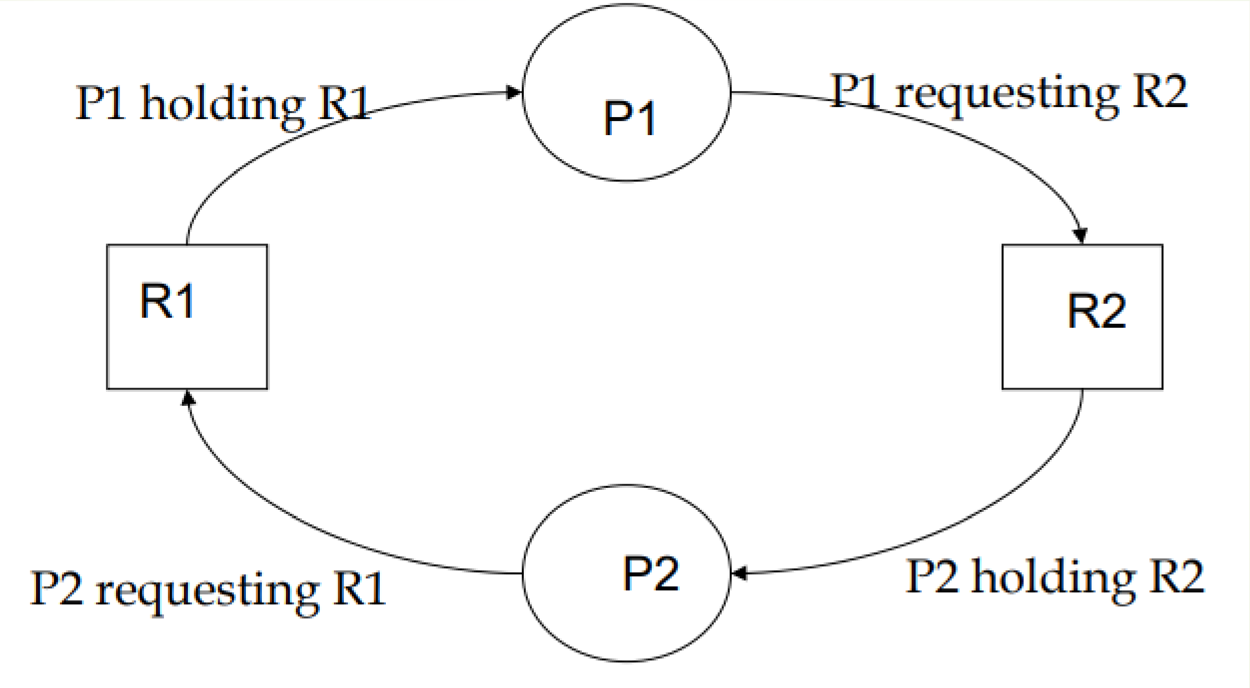
# Introduction (contd.)

- A process in a multiprogramming system is said to be in dead lock if it is waiting for a particular event that will never occur
- Example:
  - All automobiles trying to cross
  - Traffic Completely stopped
  - Not possible without backing some



# Resource Deadlock

- A process request a resource before using it, and release after using it
- If the resource is not available when it is requested, the requesting process is force to wait
- Process P1 holds resource R1 and needs resource R2 to continue; Process P2 holds resource R2 and needs resource R1 to continue – deadlock



# Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously.

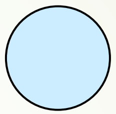
  - **Mutual exclusion:** only one process at a time can use a resource
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ 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

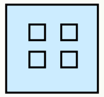
- A set of vertices $V$ and a set of edges $E$.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

# Resource-Allocation Graph (Cont.)

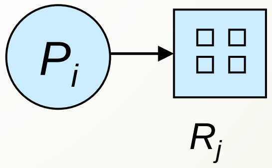
- Process



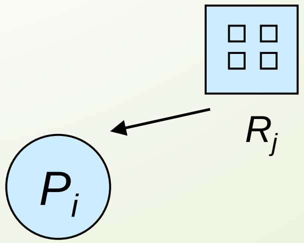
- Resource Type with 4 instances



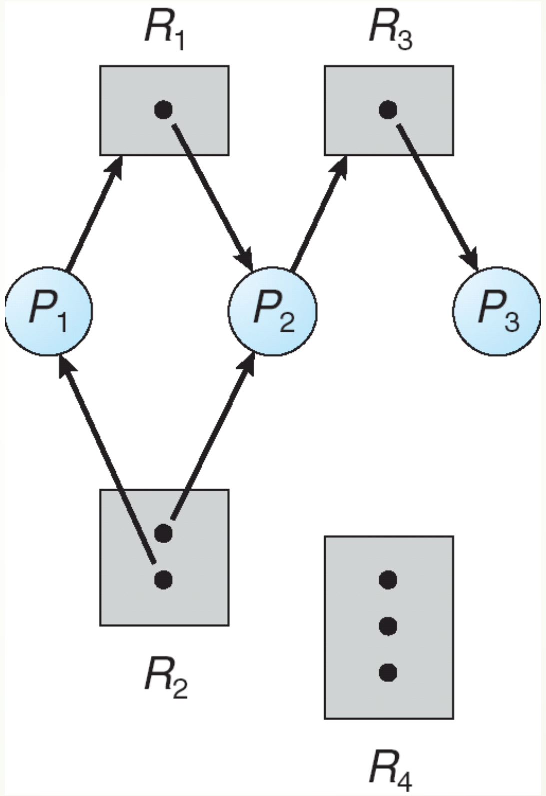
- $P_i$ requests instance of $R_j$



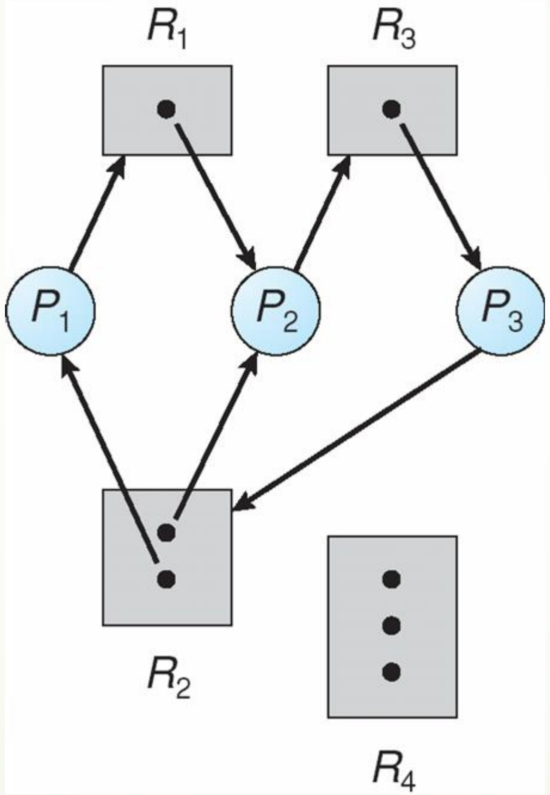

- $P_i$ is holding an instance of $R_j$



# Example of a Resource Allocation Graph



# Resource Allocation Graph With A 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 Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

# Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
  - (Making all resources shareable)
- **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 allocated to it.
  - Low resource utilization; starvation possible

# Deadlock Prevention (contd.)

- **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

# Ostrich Algorithm

- Deadlock Ignorance Method
- A strategy of ignoring problem on the basis that they may be exceedingly rare

- “To stick one’s head in the sand and pretend there is no problem”

- Used when it is cost-effective to allow the problem rather than its prevention

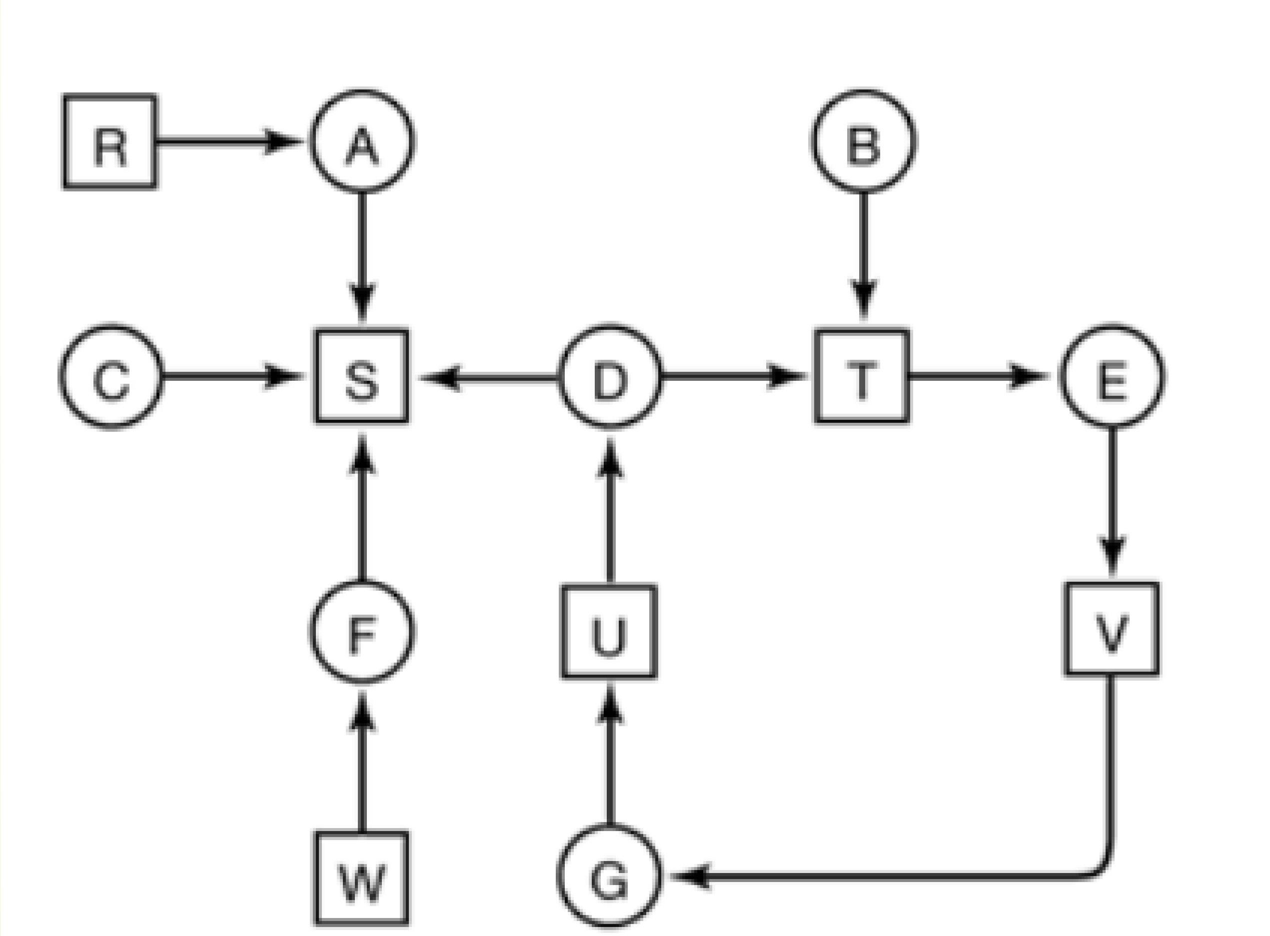
# Deadlock Detection

- The system does not attempt to prevent deadlocks from occurring
- Instead, it lets them occur, tries to detect when this happens, and then takes some action to recover after the fact
- Different approaches:
  - Deadlock Detection with One Resource of Each Type
  - Deadlock Detection with Multiple Resource of Each Type

# Deadlock Detection with One Resource of Each Type

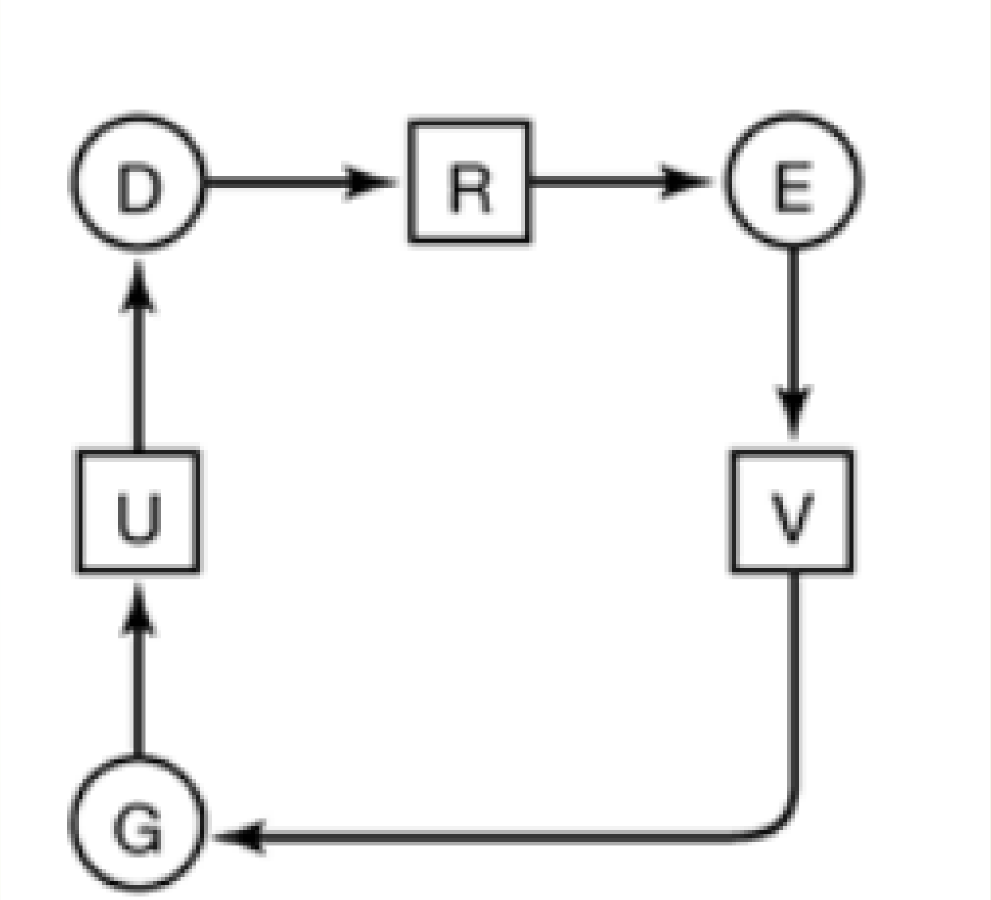
- If resource graph consists one or more cycle then deadlock occurs
- If no cycle then no deadlock
- Here, one resource of each type exists (one scanner, one printer, etc.)
- Is this System Deadlocked? If so which process are involved?
  - Process A holds R and wants S
  - Process B holds nothing but wants T
  - Process C holds nothing but wants S
  - Process D holds U and wants S and T
  - Process E holds T and wants V
  - Process F holds W and wants S
  - Process G holds V and wants U

# Deadlock Detection-One Resource (contd.)



# Deadlock Detection-One Resource (contd.)

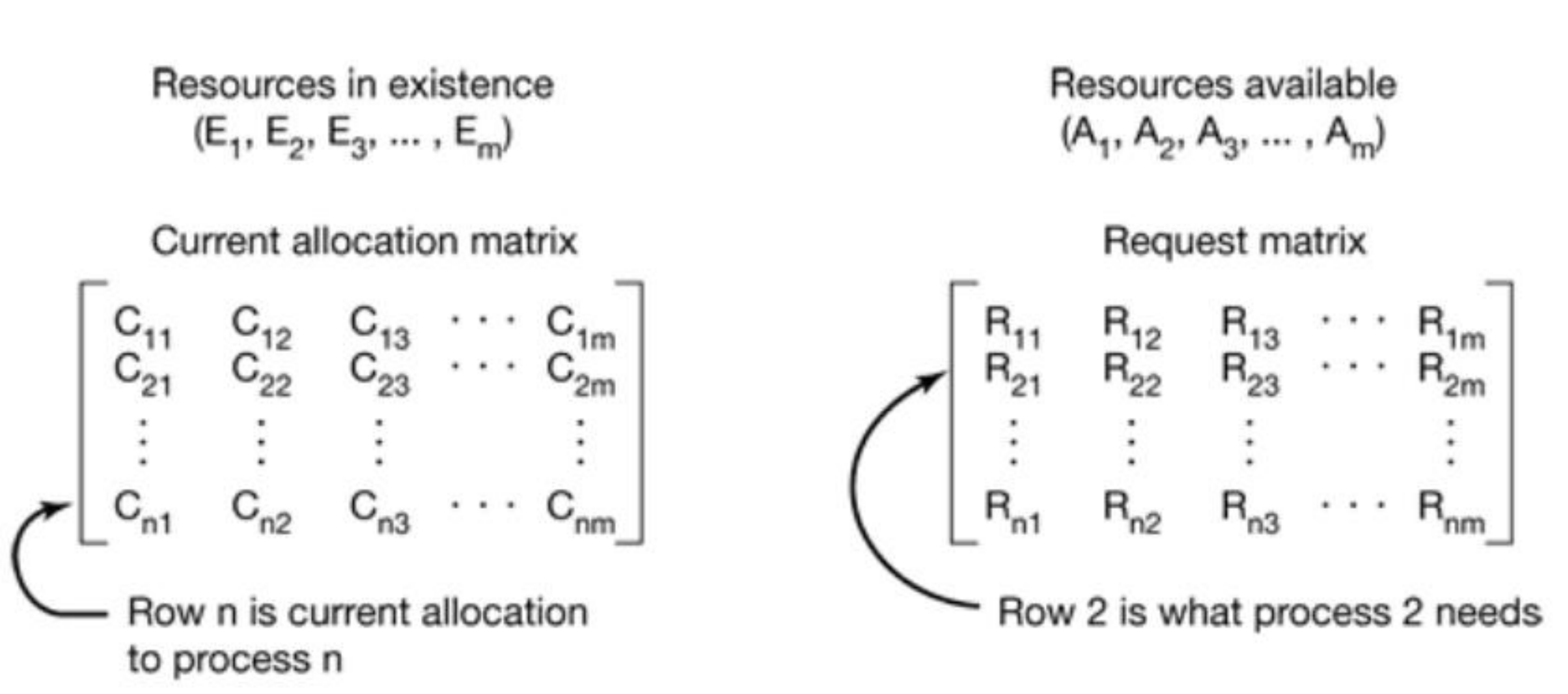
- After Extraction:



# Deadlock Detection with Multiple Resources of Each Type

- Multiple copies of resources available
- A matrix-based algorithm in order to detect deadlock
- n processes. P1 through Pn
- E is the existing resource vector
- A be the available resource vector
- two arrays, C, the current allocation matrix, and R, the request matrix
- Cij is the number of instances of resource j that are held by process i
- Rij is the number of instances of resource j that Pi wants

# Deadlock Detection - Multiple Resources (contd.)



# Deadlock Detection - Multiple Resources (contd.)

- The deadlock detection algorithm can now be given, as follows.
  - Look for an unmarked process, Pi , for which the i-th row of R is less than or equal to A
  - If such a process is found, add the i-th row of C to A, mark the process, and go back to step 1
  - If no such process exists, the algorithm terminates.

# Example

| Process | Allocated | Request | Available |
|---|---|---|---|
| P0 | 0 1 0 | 0 0 0 | 0 0 0 |
| P1 | 2 0 0 | 2 0 2 | 0 1 0 |
| P2 | 3 0 3 | 0 0 0 | 3 1 3 |
| P3 | 2 1 1 | 1 0 0 | 5 2 4 |
| P4 | 0 0 2 | 0 0 2 | 5 2 6<br>7 2 6 |

| A | B | C |
|---|---|---|
| 7 | 2 | 6 |

Work = available

# Recovery from Deadlock

- **Recovery through Preemption**
  - Temporarily take a resource away from its current owner and give it to another process
- **Recovery through Rollback**
  - When a deadlock is detected, it is easy to see which resources are needed.
  - . To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints
- **Recovery through Killing Processes**
  - Kill one or more processes

# Deadlock Avoidance - Safe and Unsafe State

- A state is said to be safe if it is not deadlocked and there is a some scheduling order in which every process can run to completion
- In a safe state, system can guarantee that all processes will finish – no deadlock occur
- For an unsafe state, no such guarantee can be given – deadlock may occur.

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

Free: 3
(a)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 4 | 4 |
| C | 2 | 7 |

Free: 1
(b)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 2 | 7 |

Free: 5
(c)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 7 | 7 |

Free: 0
(d)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 0 | – |

Free: 7
(e)

# Banker's Algorithm

- This approach to the deadlock problem anticipates a deadlock before it actually occurs
- This approach employs an algorithm to access the possibility that deadlock could occur and act accordingly
- It is named as Banker's algorithm because the process is analogous to that used by a banker in deciding if a loan can be safely made a not

# Banker’s Algorithm (contd.)

- The following are the features that are to be considered for avoidance of the deadlock s per the Banker’s Algorithms
- Each process declares maximum number of resources of each type that it may need
- Keep the system in a safe state in which we can allocate resources to each process in some order and avoid deadlock
- Check for the safe state by finding a safe sequence: where resources that Pi needs can be satisfied by available resources plus resources held by Pj where j < I
- Resource allocation graph algorithm uses claim edges to check for a safe state

# Banker's Algorithm (contd.)

## For Single Resource

- Consider an analogy in which 4 processes (P1, P2, P3 and P4) can be compared with the customers in a bank, resources such as printers etc. as cash available in the bank and the Operating system as the Banker

| Processes | Resources used | Maximum resources |
|---|---|---|
| P1 | 0 | 6 |
| P2 | 0 | 5 |
| P3 | 0 | 4 |
| P4 | 0 | 7 |

| Processes | Resources used | Maximum resources |
|---|---|---|
| P1 | 1 | 6 |
| P2 | 1 | 5 |
| P3 | 2 | 4 |
| P4 | 4 | 7 |

# Banker's Algorithm (contd.)

## For Multiple Resource

| Process | Allocated | Request | Available |
|---|---|---|---|
| P0 | 0 1 0 | 0 0 0 | 0 0 0 |
| P1 | 2 0 0 | 2 0 2 | 0 1 0 |
| P2 | 3 0 3 | 0 0 0 | 3 1 3 |
| P3 | 2 1 1 | 1 0 0 | 5 2 4 |
| P4 | 0 0 2 | 0 0 2 | 5 2 6<br>7 2 6 |

| A | B | C |
|---|---|---|
| 7 | 2 | 6 |

Work = available

# Deadlock VS Starvation

| Deadlock | Starvation |
|---|---|
| 1) A deadlock is a condition in operating systems in which no process proceeds for execution and wait for resources that have been acquired by some other processes. It is also known as circular wait. | 1) Starvation is a situation when a process keeps waiting (and starving) for a resource that is being held by other high priority processes. |
| 2) In a deadlock, all the involved processes keep waiting for each other to get completed. | 2) In this situation, the high priority processes are executed, and low priority processes are blocked. |
| 3) In a deadlock, none of them can execute because they are waiting for the other process to complete. | 4) In starvation, the low priority process starves due to lack of resources. |
| 5) The resources are blocked by the process. | 5) Resources are utilized by high priority process continuously. |
| 6) Deadlock can be prevented by avoiding conditions such as mutual exclusion, hold and wait, no preemption and circular wait. | Starvation can be prevented using the "Aging" technique. |

# Bankers algorithm Example

| PID | Allocation<br>A B C D | Max<br>A B C D | Available<br>A B C D |
|---|---|---|---|
| P0 | 2 0 0 1 | 4 2 1 2 | 3 3 2 1 |
| P1 | 3 1 2 1 | 5 2 5 2 | |
| P2 | 2 1 0 2 | 2 3 1 6 | |
| P3 | 1 3 1 2 | 1 4 2 4 | |
| P4 | 1 4 3 2 | 3 6 6 5 | |

1) What is Need Matrix?
2) Is the system is safe state? Is safe find the safe sequence
3) If request from p1 arrives for(1,0,0,0) can request be immediately granted?
4) If request from P4 arrives  for (0,0,2,0) can  it be immediately granted?