Operating Systems & Systems Programming
Module 4
Deadlock

Dr. Vikash

Jaypee Institute of Information Technology, Noida

Overview

- System consists of resources
- Resource types $R_1, R_2, ..., R_m$, 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**

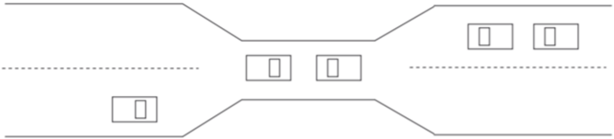
Deadlock Characterization

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 $\{P_0, P_1, ..., P_n\}$ 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_{n1} 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.

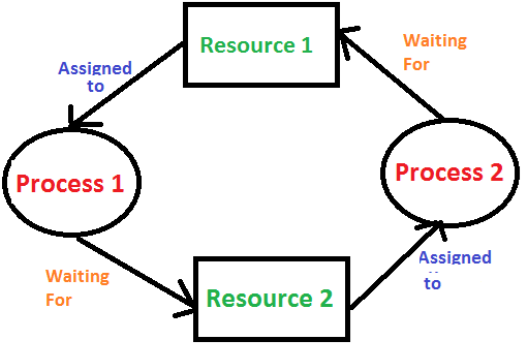
DEADLOCKS

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.

In a computer system deadlocks arise when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the group.



What is a deadlock ?

- Formal definition:
  - **“A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.”**
- Usually, the event is release of a currently held resource
- In deadlock, none of the processes can
  - Run
  - Release resources
  - Be awakened

Resource-Allocation Graph

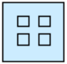
A set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$ the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, ..., R_n\}$ the set consisting of all the resources in the system.
- request edge - directed edge $P_i \rightarrow R_j$
- assignment edge - directed edge $R_j \rightarrow P_i$

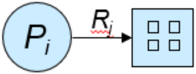
Resource-Allocation Graph

Process

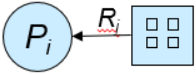
Resource Type with 4 instances



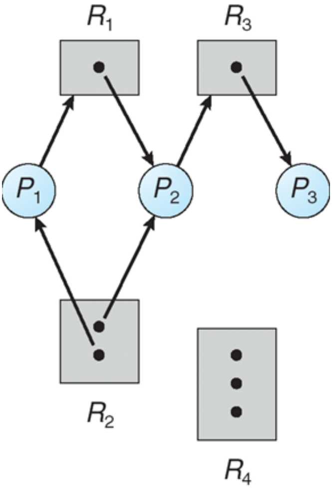
P_i requests instance of R_j



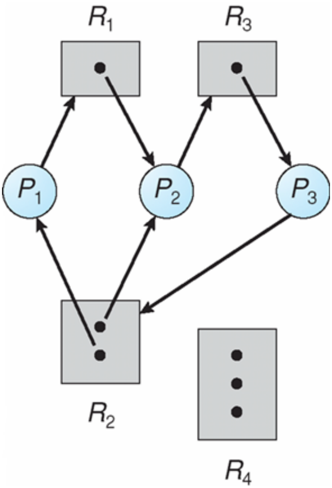
P_i is holding an instance of R_j



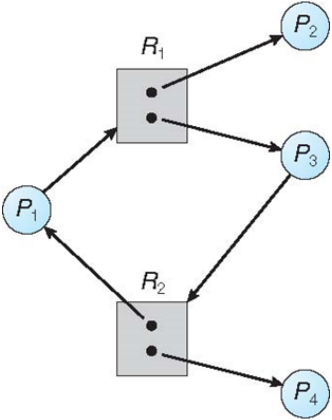
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



- 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
  - Deadlock Detection and Recovery
  - Deadlock Ignorance
- 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

Prevent Deadlock by eliminating of the four conditions (ME, Hold & wait, No-preemption, circular wait).

- **Mutual Exclusion** - not required for sharable resources (e.g., read-only files); 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 allocated to it.
  - Low resource utilization; starvation possible

- **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 Avoidance

Require 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

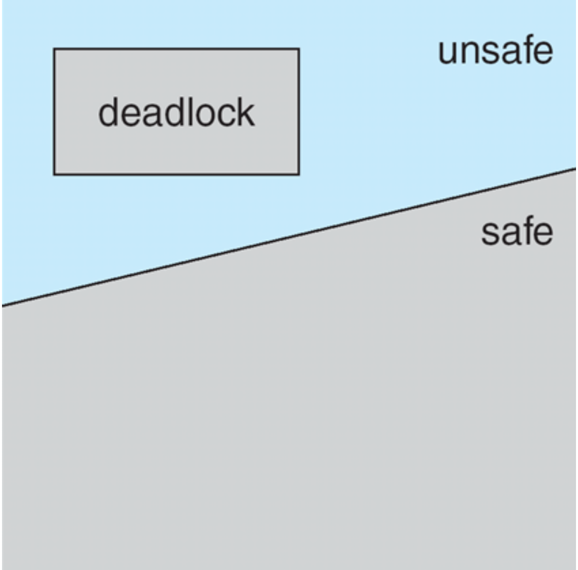
Safe State

- A state is **safe** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a **safe sequence** of processes $\{P_0, P_1, P_2, ..., P_N\}$ such that all of the resource requests for Pi can be granted using the resources currently allocated to P_i and all processes P_j where j $<$ i. (i.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which MAY lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

Basic Facts

- if a system is in safe state $\rightarrow$ no deadlocks
- if a system is in unsafe state $\rightarrow$ possibility of deadlocks
- Avoidance $\rightarrow$ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



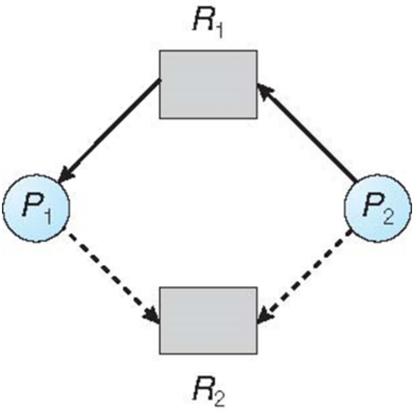
Avoidance Algorithms

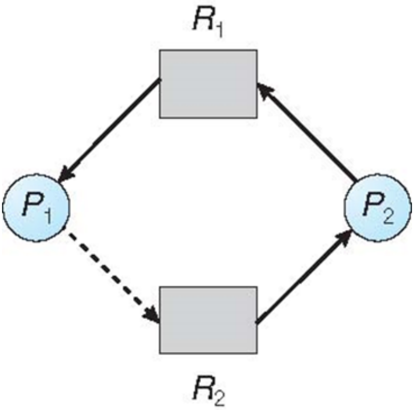
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the bankers algorithm

Resource-Allocation Graph Scheme

- **Claim** edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line
- Claim edge converts to **request** edge when a process requests a resource
- Request edge converted to an **assignment** edge when the resource is allocated to the process
- 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




R_1
P_1
P_2
R_2

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resources it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

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 processes 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]

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:

$$\begin{aligned} \text{Available} &= \text{Available} - Request_i; \\ Allocation_i &= Allocation_i + Request_i; \\ Need_i &= Need_i - Request_i; \end{aligned}$$

- 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

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:
   Finish [i] = false
   $Need_i \leq Work$ 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

Example of Banker's Algorithm

5 processes P_0 through P_4;

3 resource types:

A (10 instances), B (5instances), and C (7 instances)

Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example of Banker's Algorithm

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

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P1 Request (1,0,2)

Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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 < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement

Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?

Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

A single processor system has three resource types X, Y and Z, which are shared by three processes. There are 5 units of each resource type. Consider the following scenario, where the column alloc denotes the number of units of each resource type allocated to each process, and the column request denotes the number of units of each resource type requested by a process in order to complete execution. Which of these processes will finish LAST?

1. P0
2. P1
3. P2
4. None of the above since the system is in a deadlock

	Alloc			Request		
	X	**Y**	**Z**	**X**	**Y**	**Z**
P0	1	2	1	1	0	3
P1	2	0	1	0	1	2
P2	2	2	1	1	2	0

Practice Problem 2

An operating system uses the banker's algorithm for deadlock avoidance when managing the allocation of three resource types X, Y and Z to three processes P0, P1 and P2. The table given below presents the current system state. Here, the Allocation matrix shows the current number of resources of each type allocated to each process and the Max matrix shows the maximum number of resources of each type required by each process during its execution.

	Allocation			Max		
	X	**Y**	**Z**	**X**	**Y**	**Z**
P0	0	0	1	8	4	3
P1	3	2	0	6	2	0
P2	2	1	1	3	3	3

There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in safe state. Consider the following independent requests for additional resources in the current state-

REQ1: P0 requests 0 units of X, 0 units of Y and 2 units of Z

REQ2: P1 requests 2 units of X, 0 units of Y and 0 units of Z

A system has 4 processes and 5 allocatable resource. The current allocation and maximum needs are as follows-

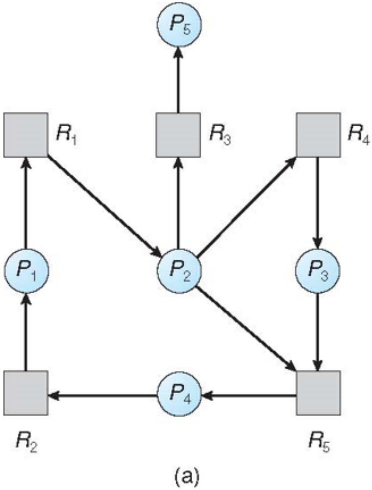
	Allocated					Maximum				
A	1	0	2	1	1	1	1	2	1	3
B	2	0	1	1	0	2	2	2	1	0
C	1	1	0	1	1	2	1	3	1	1
D	1	1	1	1	0	1	1	2	2	0

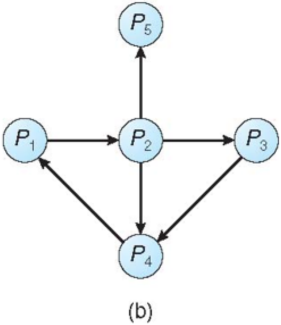
If Available = [0 0 X 1 1], what is the smallest value of x for which this is a safe state?

- 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

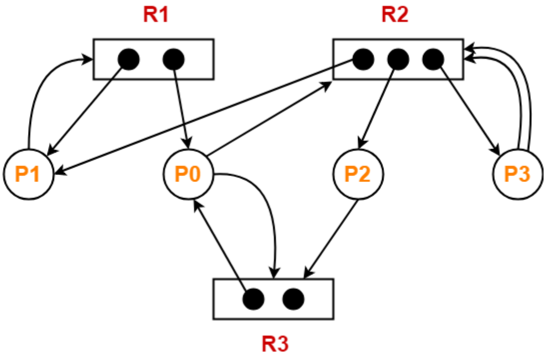




Resource-Allocation Graph

Corresponding wait-for graph

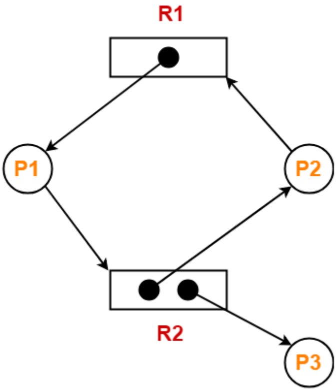
Consider the resource allocation graph in the figure-



Find if the system is in a deadlock state otherwise find a safe sequence.

Example Problem 2

Consider the resource allocation graph in the figure-



Find if the system is in a deadlock state otherwise find a safe sequence.

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 $Allocation_i \neq 0$, then **Finish[i]=false**; otherwise, **Finish[i]=true**
2. Find an index i such that both:
   a. **Finish[i] == false**
   b. $Request_i \leq$ **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] == 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

Example of Detection Algorithm

Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm

P_2 requests an additional instance of type C

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

State of system?

- Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
- Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

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 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
  6. 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

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Example

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

Thank You!!!