L22- DEADLOCK AVOIDANCE AND TUTORIAL

Overview of Deadlock Management Section
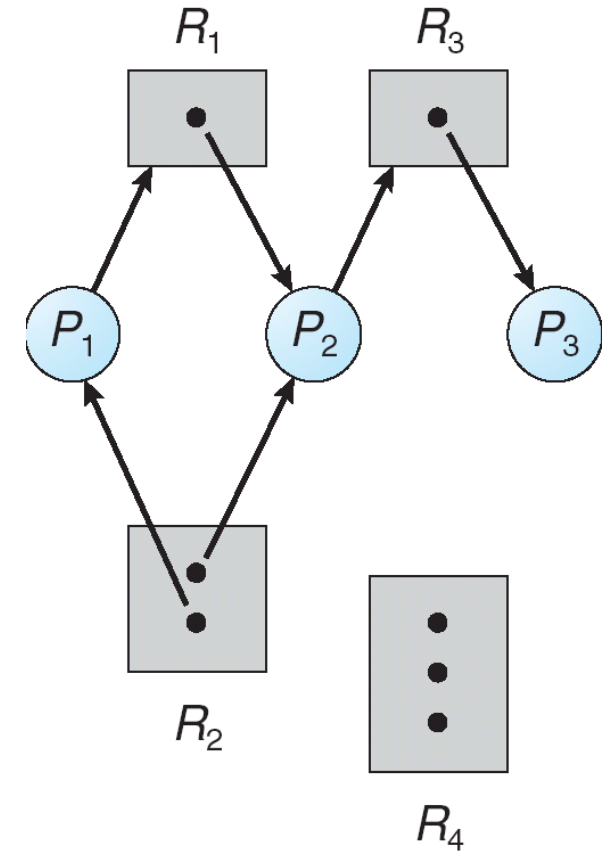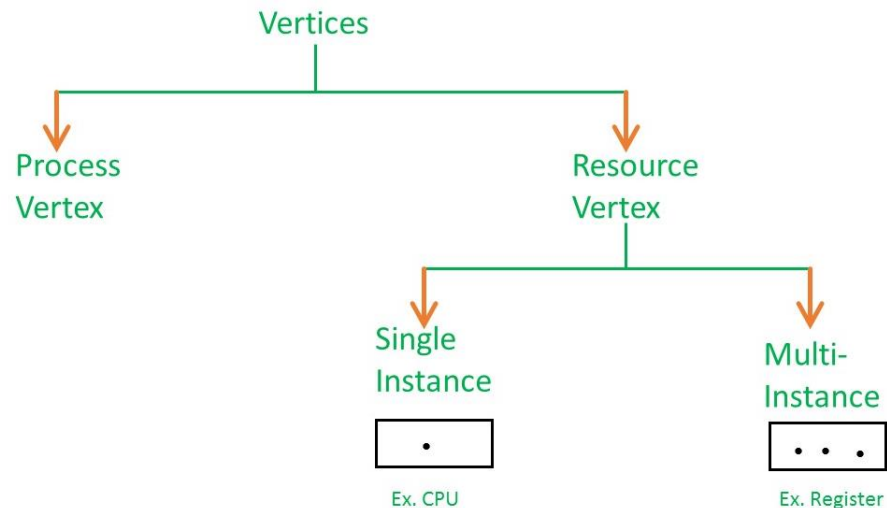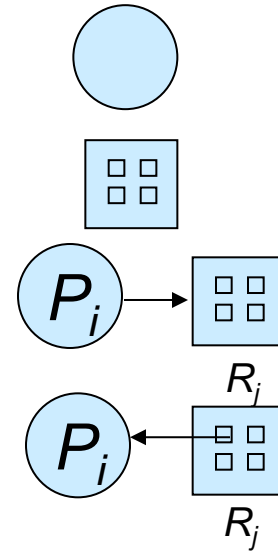
❖ System Model

❖ Deadlock Characterization

❖ Methods for Handling Deadlocks

❖ Deadlock Prevention

❖ Deadlock Avoidance

Deadlock Characterization

❖ Deadlock can arise if the following 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, $\ldots$, 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

❖ Process

❖ Resource Type with 4 instances

❖ P_i requests an instance of R_j

❖ P_i is holding an instance of R_j

$P_i \rightarrow R_j$

$P_i \leftarrow R_j$

Vertices

Process Vertex

Resource Vertex

Single Instance

Ex. CPU

Multi-Instance

Ex. Register

R_1 R_3

P_1 P_2 P_3

R_2

R_4

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 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

Safe State & Deadlock

❖ 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.
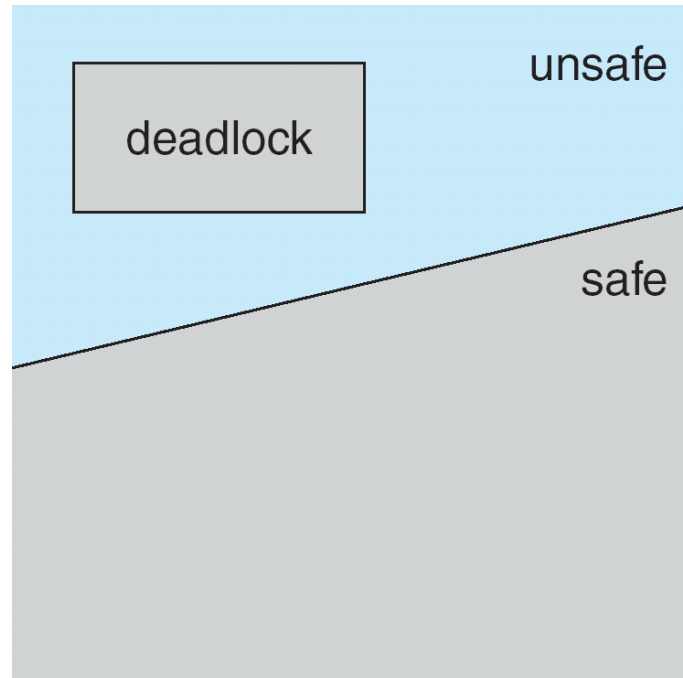
Avoidance Algorithms

❖ Single instance of a resource type

  ❖ Use a resource-allocation graph

❖ Multiple instances of a resource type

  ❖ Use the banker's algorithm

Resource-Allocation Graph Scheme

❖ **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j

❖ Claim edge is 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 & Unsafe State
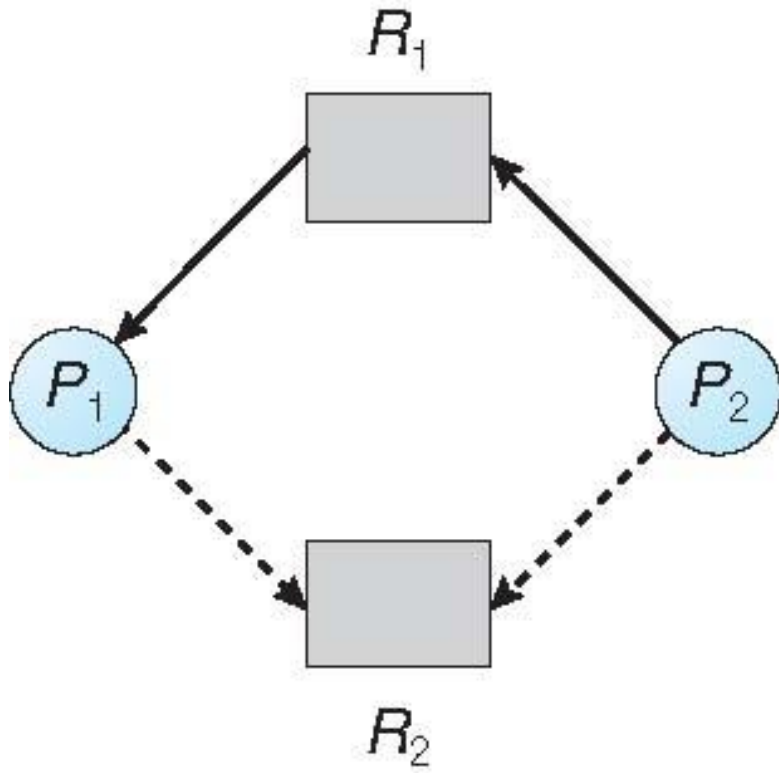
Resource-Allocation Graph
with Claim Edges
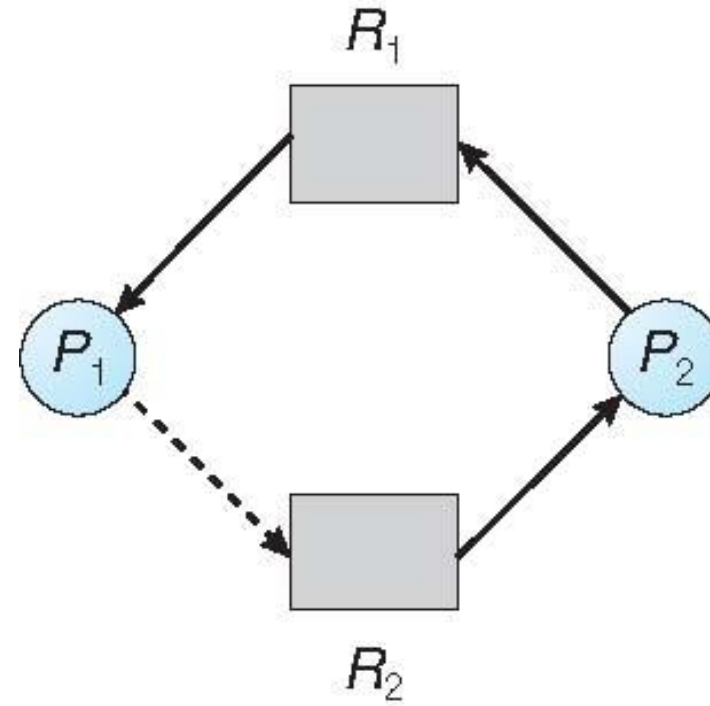
Resource-Allocation Graph Algorithm

❖ 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

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 type

❖ **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]

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) **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$_i$** ≤ **Need$_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If **Request$_i$** ≤ **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 states

$$\textbf{Available = Available – Request}_i;$$

$$\textbf{Allocation}_i \textbf{= Allocation}_i \textbf{+ Request}_i;$$

$$\textbf{Need}_i \textbf{= Need}_i \textbf{– Request}_i;$$

❖ If safe ⇒ the resources are allocated to **P$_i$**

❖ If unsafe ⇒ **P$_i$** must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

❖ 5 [P_0 -P_4] & 3 resource types: A (10), B (5), and C (7)

❖ 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	

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

<div align="center">

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

</div>

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

❖ 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 < **P$_1$**, **P$_3$**, **P$_4$**, **P$_0$**, **P$_2$**> satisfies safety requirement

Q1: Consider two processes P and Q with an event Pe and Qe, respectively executing inside a loop. P and Q are executing in a multiprogramming environment on a single processor where context switching can happen arbitrarily at any time. We want events Pe and Qe to strictly execute alternatively in the sequence Pe, Qe, Pe, Qe .. (10 times). Give an algorithmic solution to ensure this using semaphores.

Process P:

```
for (i=10; i>0; i--)

{

  WAIT (X)

  Pe

   SIGNAL (Y)

}
```

Process Q:

```
for (j=10; i>0; j--)

{

    WAIT(Y)

    Qe

    SIGNAL (X)

}
```

Q2: Consider a system with 4 processes P1, P2, P3 & P4 and 3 resource types R1, R2 & R3. The current Allocation matrix and Need matrix are given below. If there is exactly one instance of all resource types available now, is the system in safe state now? If so, give a safe sequence.

	Allocation			Need		
	R1	R2	R3	R1	R2	R3
P1	1	0	2	1	2	1
P2	1	2	1	2	3	1
P3	0	1	1	1	0	1
P4	0	1	0	2	2	0

	Allocation			Need		
	R1	R2	R3	R1	R2	R3
P1	1	0	2	1	2	1
P2	1	2	1	2	3	1
P3	0	1	1	1	0	1
P4	0	1	0	2	2	0

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) **Need$_i$ ≤ 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

Q3: Consider a system with 4 processes P1, P2, P3 & P4 and 3 resource types R1, R2 & R3. The current Allocation matrix and Need matrix are given below. If there is exactly one instance of all resource types available now, is the system in safe state now? If so, give a safe sequence.

	Allocation			Need		
	R1	R2	R3	R1	R2	R3
P1	1	0	2	1	2	1
P2	1	2	1	2	3	1
P3	0	1	1	1	0	1
P4	1	0	1	2	2	0

Safe state and Safe sequence

	Allocation			Need		
	R1	R2	R3	R1	R2	R3
P1	1	0	2	1	2	1
P2	1	2	1	2	3	1
P3	0	1	1	1	0	1
P4	1	0	1	2	2	0

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) **Need$_i$ ≤ 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

Q4. Each of a set of n processes executes the following code using two semaphores a and b initialized to 1 and 0, respectively. Assume that count is a shared variable, initialized to 0 and not used in CODE SECTION P. What does the code achieve?

```
CODE SECTION P

wait(a); count=count+1;
if (count==n) signal(b);
signal(a); wait(b); signal(b);

CODE SECTION Q
```

(a) It ensures that at most n-1 process are in CODE SECTION Q at any time.

(b) It ensures that all process executes CODE SECTION Q mutually exclusively.

(c) It ensures that at most n-1 process are in CODE SECTION P at any time.

(d) It ensures that no process executes CODE SECTION Q before every process has finished CODE SECTION P

Thank You