CS343 Operating Systems

Lecture 12

Deadlock Avoidance, Detection and Recovery

John Jose

Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati

Methods for Handling Deadlocks

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
- 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

Deadlock Prevention

- Deadlock prevention is done by ensuring that at least one of the necessary 4 conditions for deadlock is not met.
- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

Deadlock Avoidance

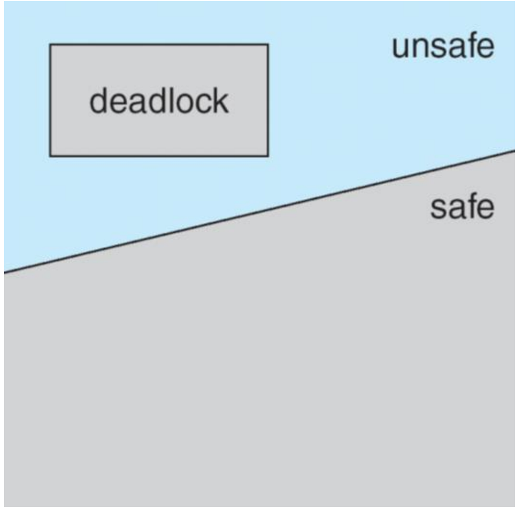
- Requires that the system has some additional ***a priori*** information available
- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state 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, …, 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.



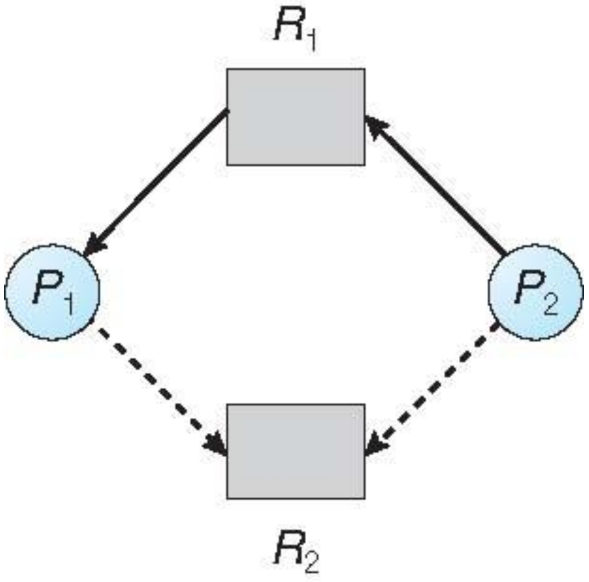
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

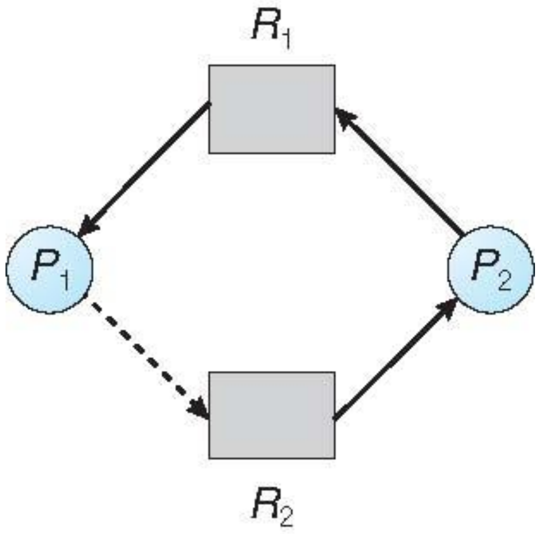
Resource-Allocation Graph Scheme

- **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 *apriori* in the system

Resource-Allocation Graph & Unsafe State



Resource-Allocation Graph
with Claim Edges



Unsafe State In
Resource-Allocation Graph

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 apriori 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**$_i$ $\leq$ **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 states

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

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

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

- 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 [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	

Example of Banker's Algorithm contd…

- 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: 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

Deadlock Detection

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

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?
  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

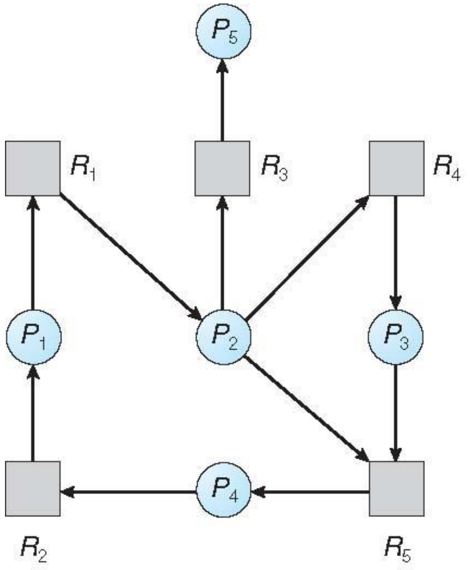
Detection in Single Instance Resource Types

- Maintain **wait-for** graph
  - Nodes are processes
  - $\mathbf{P_i} \rightarrow \mathbf{P_j}$ if $\mathbf{P_i}$ is waiting for $\mathbf{P_j}$

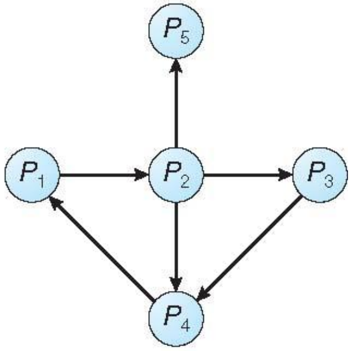
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of $\mathbf{n^2}$ operations, where **n** is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

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 $\mathbf{P_i}$ is requesting **k** more instances of resource type $\mathbf{R_j}$.

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 $\mathbf{Allocation_i \neq 0}$, then **Finish[i] = false**; otherwise, **Finish[i] = true**

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

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

   (b) $\mathbf{Request_i \leq Work}$

   If no such **i** exists, go to step 4

Detection Algorithm contd..

3. **Work = Work + Allocation$_i$**
   Finish[i] = true
   go to step 2

4. If **Finish[i] == false**, for some **i**, $1 \leq \mathbf{i} \leq \mathbf{n}$, then the system is in deadlock state. Moreover, if **Finish[i] == false**, then **P$_i$** is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes $\mathbf{P_0}$ through $\mathbf{P_4}$; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time $\mathbf{T_0}$:

	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	

Example of Detection Algorithm contd..

- ***P_2*** requests an additional instance of type ***C***

<u>Request</u>

	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***

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/