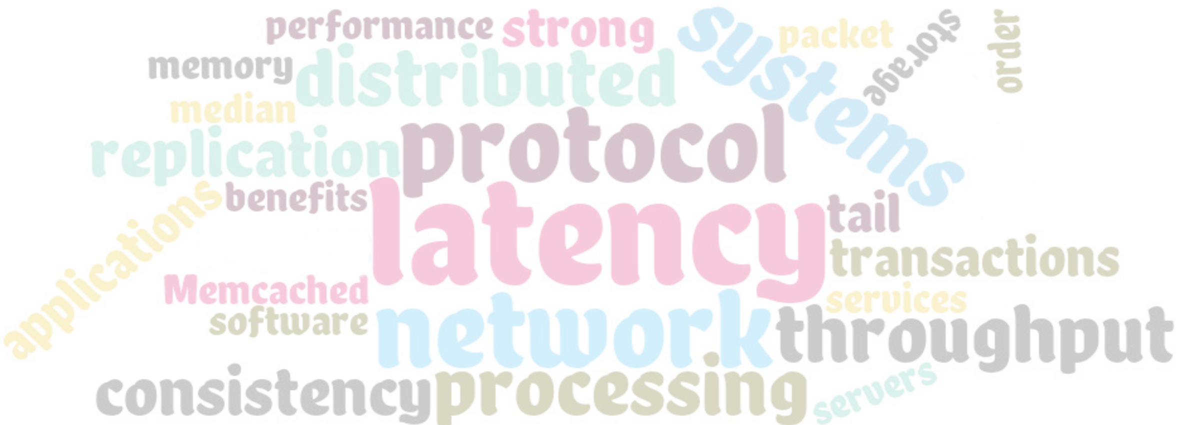


Deadlocks Detection

Course: Distributed Computing

Faculty: Dr. Rajendra Prasath

Spring 2019

About this topic

This course covers various concepts in **Mutual Exclusion in Distributed Systems.** We will also focus on different types of distributed mutual exclusion algorithms in distributed contexts and their analysis

RECAP

What did you learn so far?

- Challenges in Message Passing systems
- Distributed Sorting
- Space-Time Diagram
- Partial Ordering / Causal Ordering
- Concurrent Events
- Local Clocks and Vector Clocks
- Distributed Snapshots
- Termination Detection
- Topology Abstraction and Overlays
- Leader Election Problem in Rings
- Message Ordering / Group Communications
- Distributed Mutual Exclusion Algorithms

Topics to focus on …

- ➔ Distributed Mutual Exclusion
- ➔ Deadlock Detection
- ➔ Check pointing and rollback recovery
- ➔ Self-Stabilization
- ➔ Distributed Consensus
- ➔ Reasoning with Knowledge
- ➔ Peer – to – peer computing and Overlays
- ➔ Authentication in Distributed Systems

For End Semester

Deadlocks

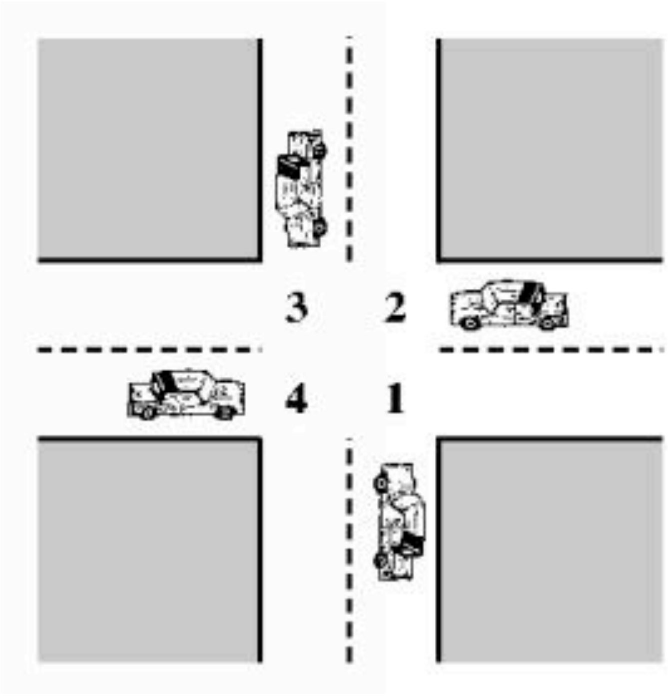
Let us explore deadlock detection, prevention and avoidance algorithms in distributed systems

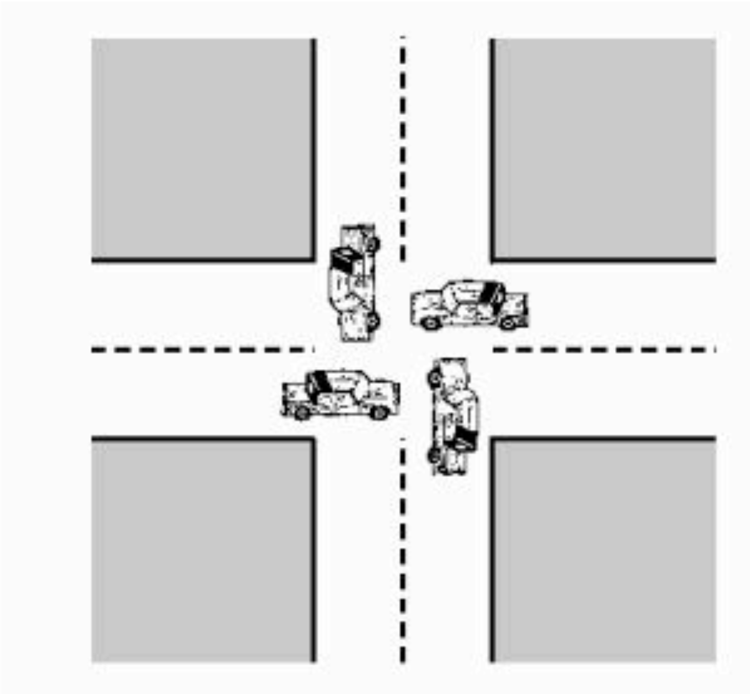
Distributed Mutual Exclusion (recap)

- **No Deadlocks** – No processes should be permanently blocked, waiting for messages (Resources) from other sites
- **No starvation** – no site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
- **Fairness** - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation)
- **Fault Tolerance** – the algorithm is able to survive a failure at one or more sites

Deadlock – A Simple Example

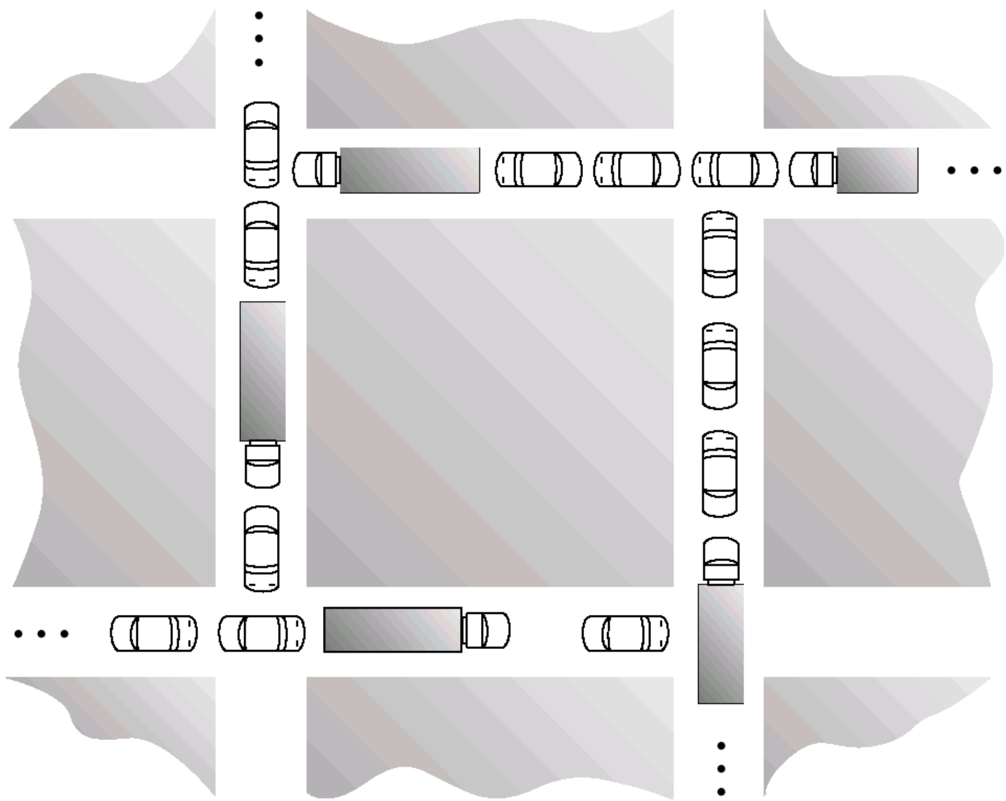
- Vehicular Traffic at a signal





Deadlock – Another Example

➔ **Vehicular Traffic – Another Scenario**



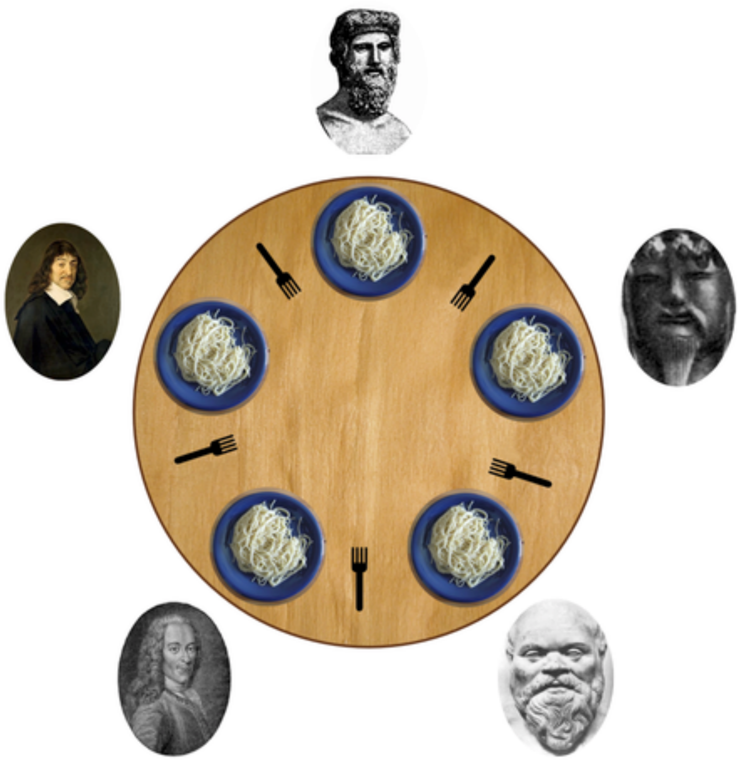
Deadlock – Illustrated

- Vehicular Traffic – A real-time scenario

Dining Philosophers' Problem

- ➔ Each philosopher must alternately think and eat
- ➔ A philosopher can only eat when they have both left and right forks

➔ Suggest a Simple Solution ??

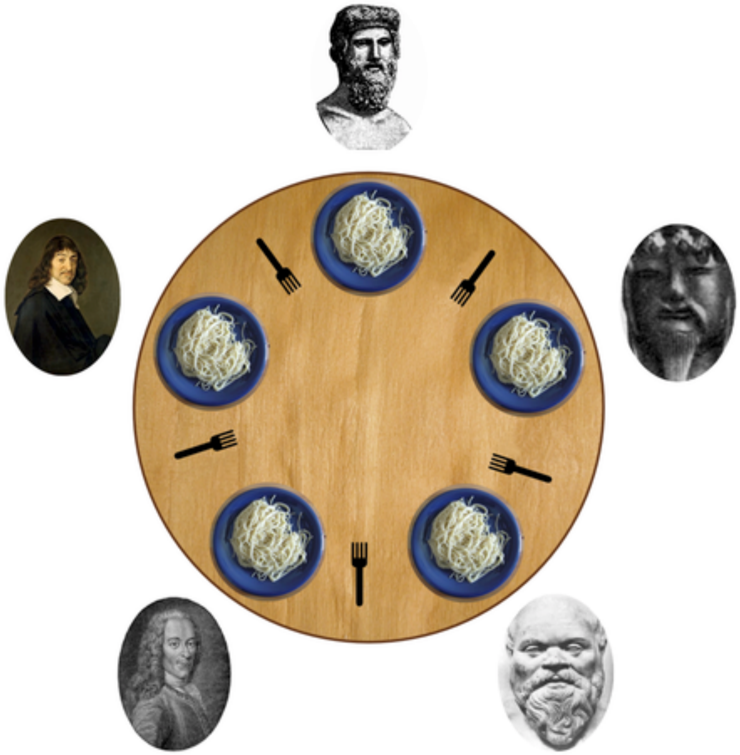


- ➔ Problem: How to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve?

Dining Philosophers' Problem

➔ Deadlock-Free Solutions !!

- ➔ Soln - 1: Forks will be numbered 1 through 5 and each philosopher will always pick up the lower-numbered fork first, and then the higher-numbered fork
- ➔ Soln - 2: Use Arbitrator (waiter) to grant permission to pick up both forks



Deadlocks in Distributed Sytems

Definition

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

- No process can progress in the system
- Competing processes may WAIT indefinitely for resources
- How do we manage resources among the competing tasks efficiently?

Deadlocks – A few more examples

Tape Drives

- Assume that a system has two Tape Drives
- There are two processes P_1 and P_2 each hold one drive
- Now each process needs access to another tape drive
- P_1 does not get access to the resource held by P_2 and vice versa.
- This implies DEADLOCK … neither P_1 nor P_2 succeeds in its attempt

Deadlocks – A few more examples

Semaphores

- Semaphores A and B

P_1	P_2
wait (A)	wait(B)

OR

wait (B)	wait(A)

- This implies DEADLOCK … neither P_1 nor P_2 succeeds in its attempt

Deadlock – Characterization

- **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 $\{P_0, P_1, \cdots, P_0\}$ of waiting processes such that P_i is waiting for a resource that is held by $P_j \ (mod\ n)$ where n is the total number of resources

System Model

- 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 (Critical Section)
  - RELEASE

- Recall - Distributed Exclusion Algorithms

Resource Allocation Graph (RAG)

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

- V is partitioned into two types:
  - Set consisting of all processes

    $$P = \{P_1, P_2, \ldots, P_n\}$$

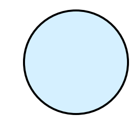
  - Set consisting of all resource types

    $$R = \{R_1, R_2, \ldots, R_m\}$$

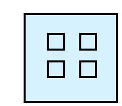
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph (contd)

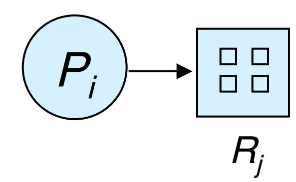
- Process



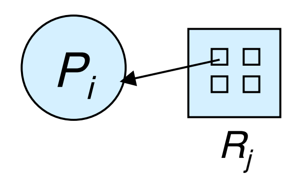
- Resource type with 4 instances



- P_i requests an instance of R_j

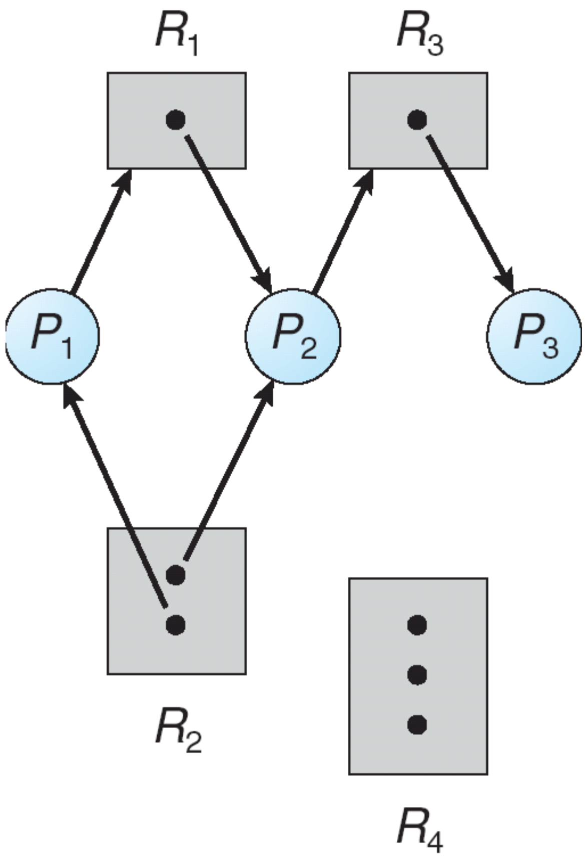


- P_i is holding an instance of R_j



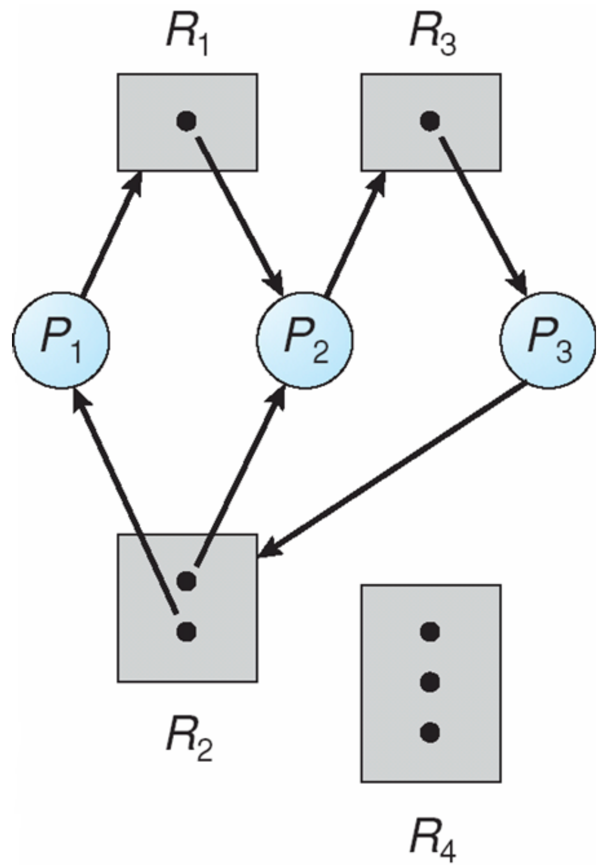
RAG – An example

- Look at this graph
- Resources:
  - R_1 - 1 unit
  - R_2 - 2 units
  - R_3 - 1 unit
  - R_4 - 3 units
- Requests:
  - P_1, P_2, P_3



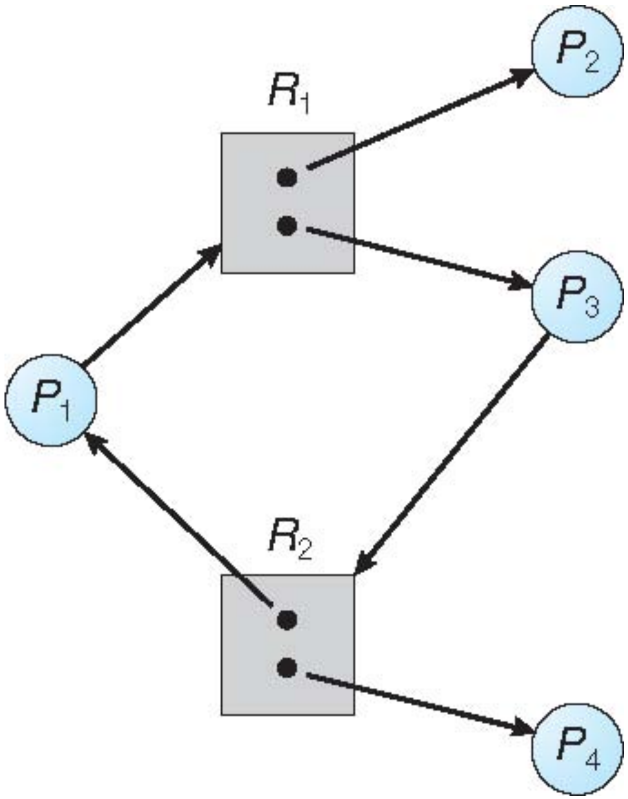
RAG with a Deadlock

- Look at this graph
  - P_1 needs R_1 which in turn used by P_2 and P_2 is requesting R_3 which is currently being accessed by P_3 and P_3 needs R_2 which is being locked by P_1 and P_2
- This implies Deadlock



RAG with a cycle but NO Deadlock

- Look at this graph
  - P_2 and P_4 may release the resource R_3 in finite time as they do not depend on other competing processes
  - There exists a cycle but may not be 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

How to handle Deadlocks?

- Ensure that the system will never enter a deadlock state
  - Deadlock Prevention – Stop before it happens!
  - Deadlock Avoidance – Precautions !!
  - Deadlock Detection – How to overcome?
- 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

4 Conditions to occur Deadlocks:

- ➔ Mutual Exclusion – Exclusive access – when a process accesses a resource, it is granted exclusive use of that resource
- ➔ Hold and wait – a process is allowed to hold onto some resources while waiting for other resources
- ➔ No preemption – a process cannot preempt or take away the resources held by another process
- ➔ Cyclical wait – There is a circular chain of waiting processes, each waiting for a resource held by the next process in the chain

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
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

Safe State

- When a process requests an available resource, system must decide whether the allocation immediate 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 processes such that for each P_i, the resources that P_i can still request, can be satisfied by available resources + resources held by all P_j, j < I
  - 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 get resources and so on

Basic Facts

- If a system is in safe state

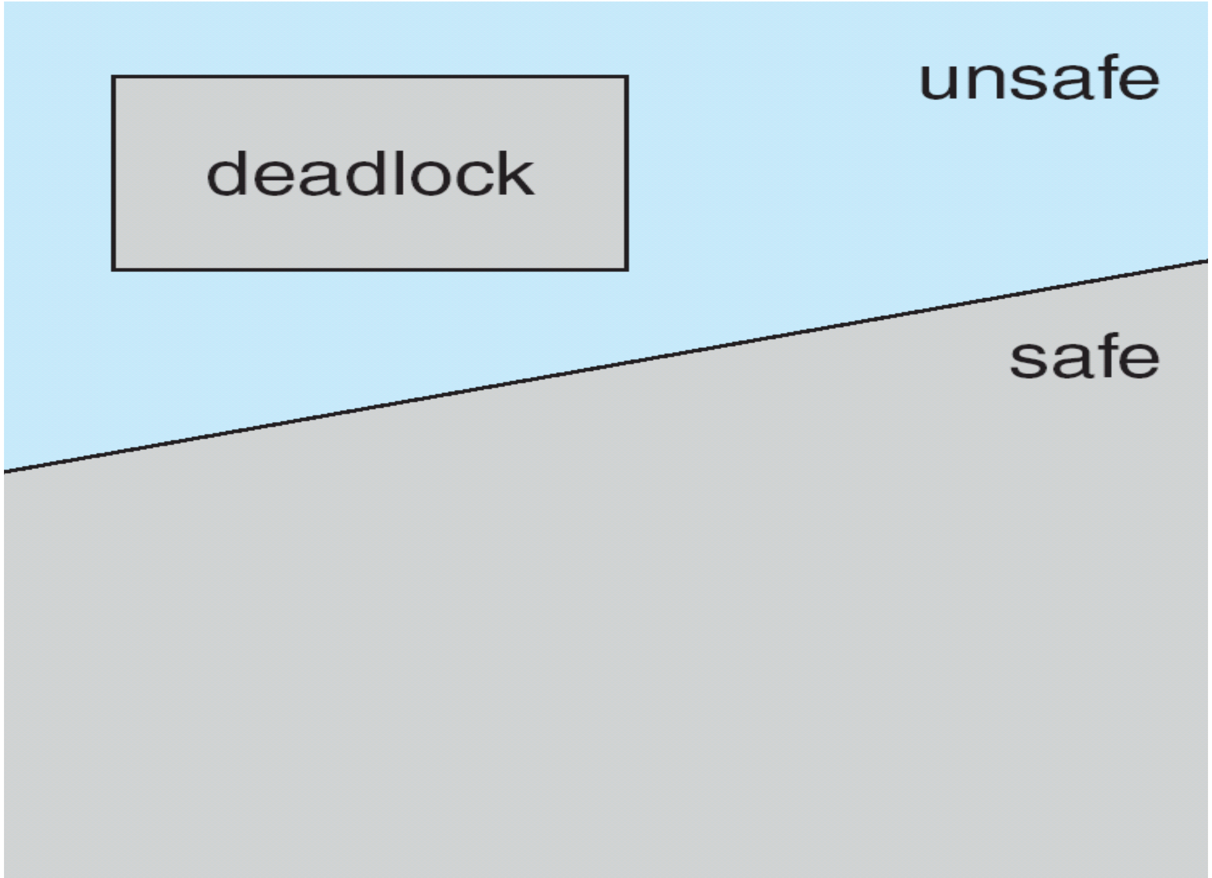
  $\Rightarrow$ no deadlocks
- If a system is in unsafe state

  $\Rightarrow$ possibility of deadlock
- Avoidance

  $\Rightarrow$ ensure that a system will never enter an unsafe state

Safe / Unsafe / Deadlock State

➔ Illustration of safe, unsafe and deadlock state



Deadlock Avoidance Algorithms

Avoidance Algorithms

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

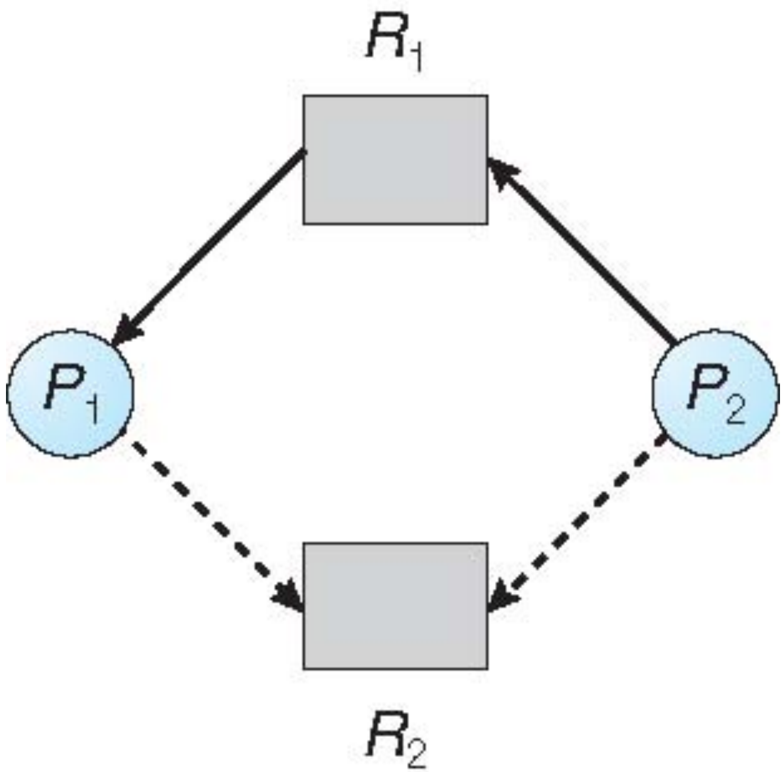
Resource-Allocation Graph Scheme

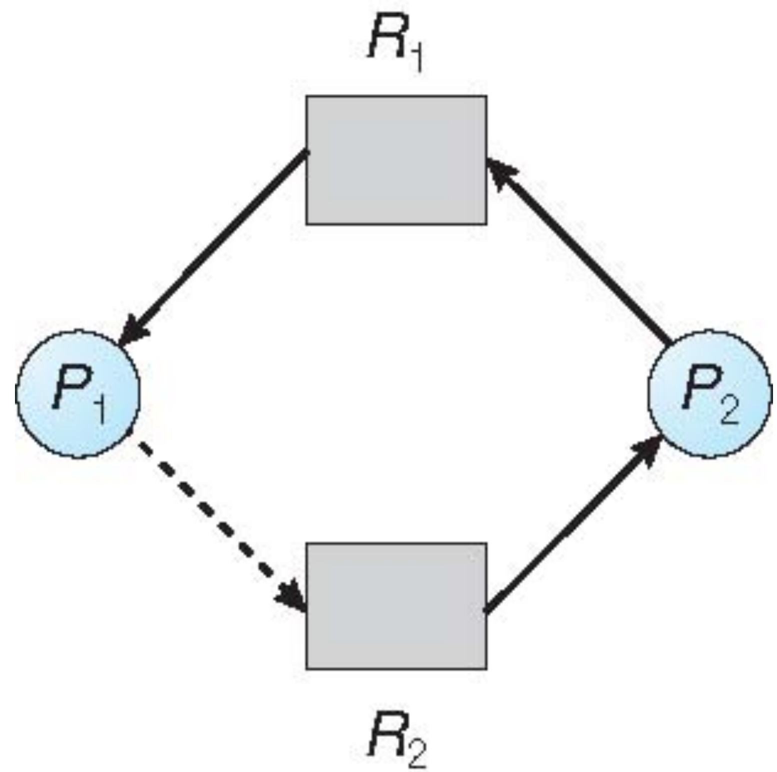
- Claim edge $P_i \rightarrow R_j$ indicate:
  - (i) process P_j may request resource R_j
  - (ii) represented by a dashed line
- Claim edge converts to Request edge when a process requests for a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When 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

- An Example

Unsafe state





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 of Resources
- 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 - Banker's Algorithm

Let n = number of processes;
    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 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 and 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 Algo 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 $Request_i \leq Need_i$ then go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, then 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_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 (contd)

- ➔ Need matrix is defined to be as follows:

  Need = Max – Allocation

	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	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, $(1,0,2) \le (3,3,2) \Rightarrow$ 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
- 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

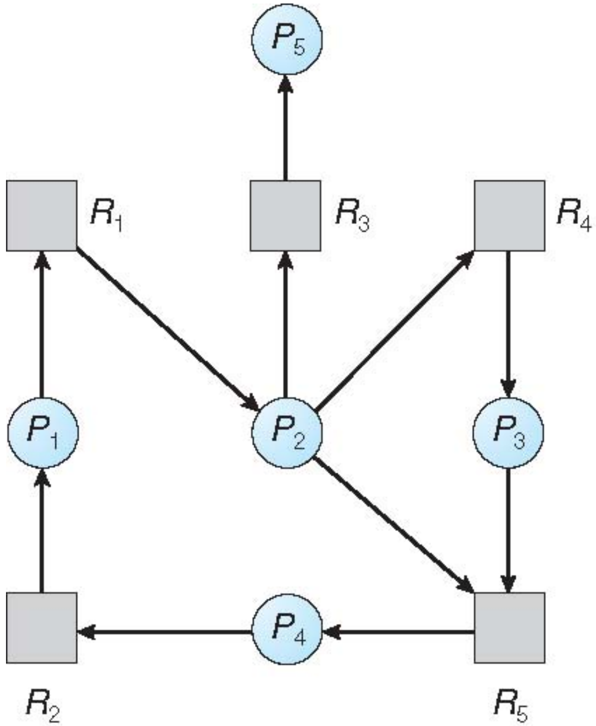
Single Instance of Each Resource Type

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

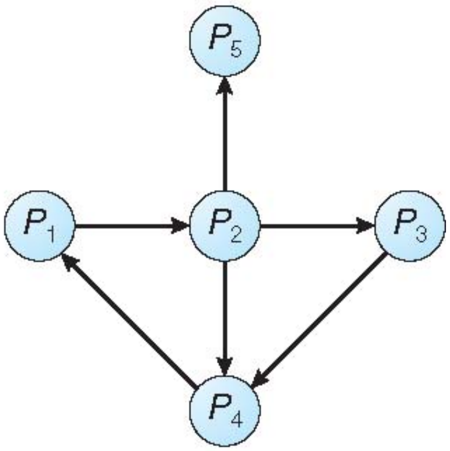
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- 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 / Wait-for Graph



(a)



(b)

Resource allocation Graph

Wait For Graph (WFG)

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; else 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

Detection Algorithm (contd)

- (3) $Work = Work + Allocation_i$
  $Finish[i] = true$; go to step 2

- (4) If $Finish[i] == false$, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked

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

Detection Algorithm - Example

- 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

Detection Algorithm – Example (contd)

- 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 the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - 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

Resource Links

- Distributed Deadlock Detection
  - http://www.cse.scu.edu/~jholliday/dd_9_16.htm

- Coffman et. al., System Deadlocks, ACM Computing Surveys. 3 (2) (1971): 67–78. DOI: 10.1145/356586.356588

- Havender, James W., Avoiding deadlock in multitasking systems, IBM Systems Journal. 7 (2) (1968): 74. DOI: 10.1147/sj.72.0074

- Knapp, Edgar, Deadlock detection in distributed databases, ACM Computing Surveys, 19 (4) (1987): 303–328. DOI:10.1145/45075.46163. ISSN 0360-0300

Summary

- Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
    - Resource Allocation Graphs
    - Banker's Algorithm
  - Recovery from Deadlocks

  - Performance Metrics

- Stay tuned ... More to come up ... !!

How to reach me?

- ➔ Please leave me an email:
  rajendra [DOT] prasath [AT] iiits [DOT] in

- ➔ Visit my homepage @

  - ➔ http://www.iiits.ac.in/FacPages/index-rajendra.html

  OR

  - ➔ http://rajendra.2power3.com

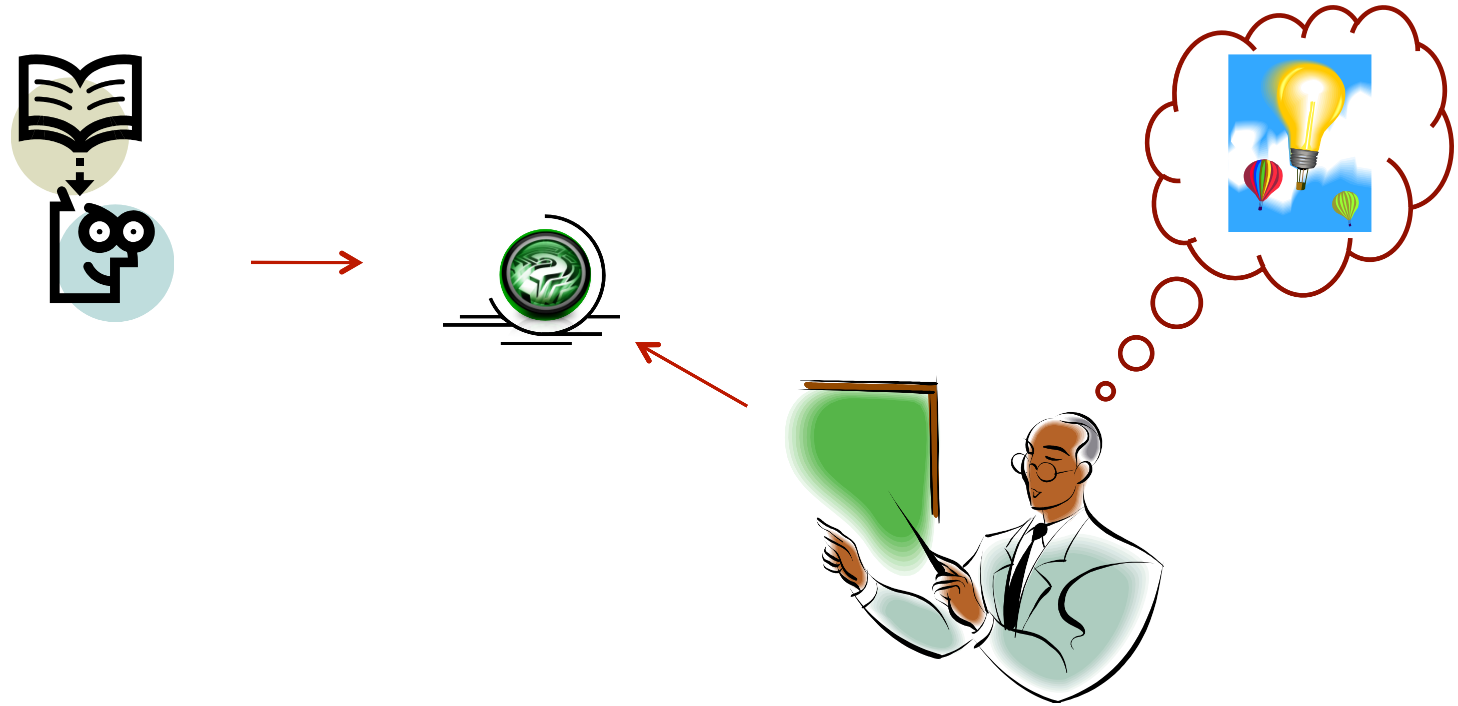
Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

Thanks …



… Questions ???