African University of Science and Technology

Deadlocks

Prof. Ousmane THIARE

[www.ousmanethiare.com]

June 22, 2015

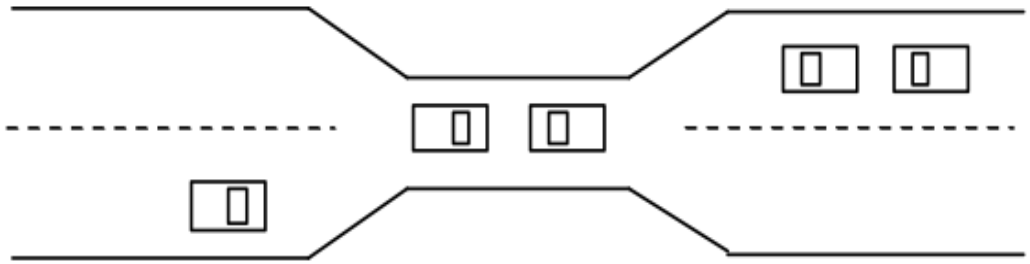
Outline

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - semaphores A and B, initialized to 1
  - P_0 executes wait (A), A=0
  - P_1 executes wait (B), B=0
  - P_0 and P_1 can not go further

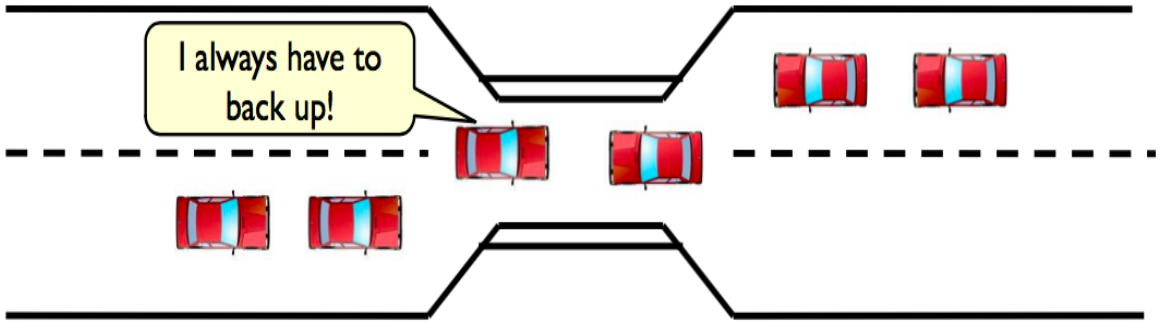
P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

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.

Bridge Crossing Example



- Deadlock vs. Starvation
  - Starvation = Indefinitely postponed
    - Delayed repeatedly over a long period of time while the attention of the system is given to other processes
    - Logically, the process may proceed but the system never gives it the CPU (unfortunate scheduling)
  - Deadlock = no hope
    - All processes blocked; scheduling change won't help

System Model

- Resource types $R_1, R_2, \cdots, R_m$
  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. At least one of the resources is non-sharable (that is; only a limited number of processes can use it at a time and if it is requested by a process while it is being used by another one, the requesting process has to wait until the resource is released.).
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes. There must be at least one process that is holding at least one resource and waiting for other resources that are being hold by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task. No resource can be preempted before the holding process completes its task with that resource.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Circular wait:** there exists a set $\{P_0, P_1, \cdots, P_0\}$ 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 $P_2, \cdots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Methods for handling deadlocks are:

- Deadlock prevention;
- Deadlock avoidance;
- Deadlock detection and recovery.

Resource-Allocation Graph

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

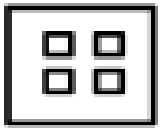
- V is partitioned into two types:
  - $P = \{P_1, P_2, \cdots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \cdots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph

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

- Process

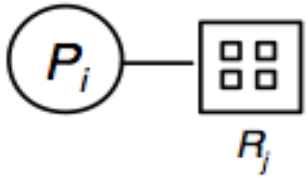
- Resource Type with 4 instances



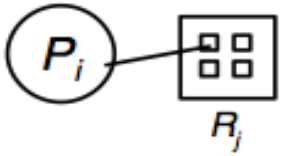
Resource-Allocation Graph

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

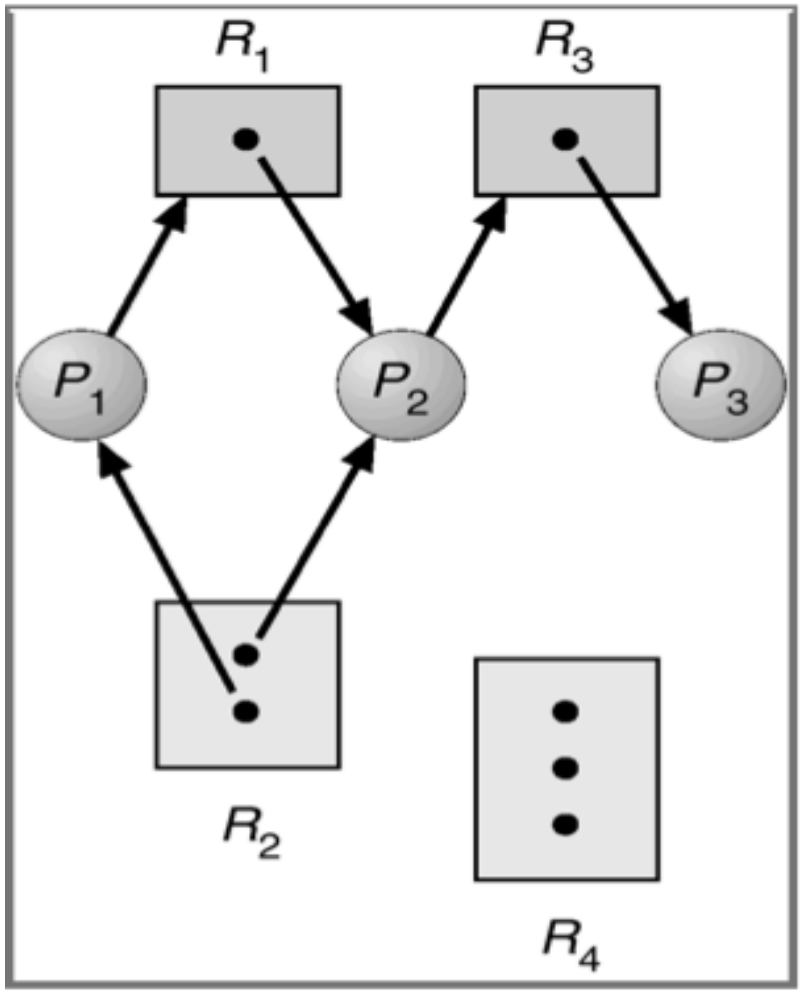
- P_i requests instance of R_j



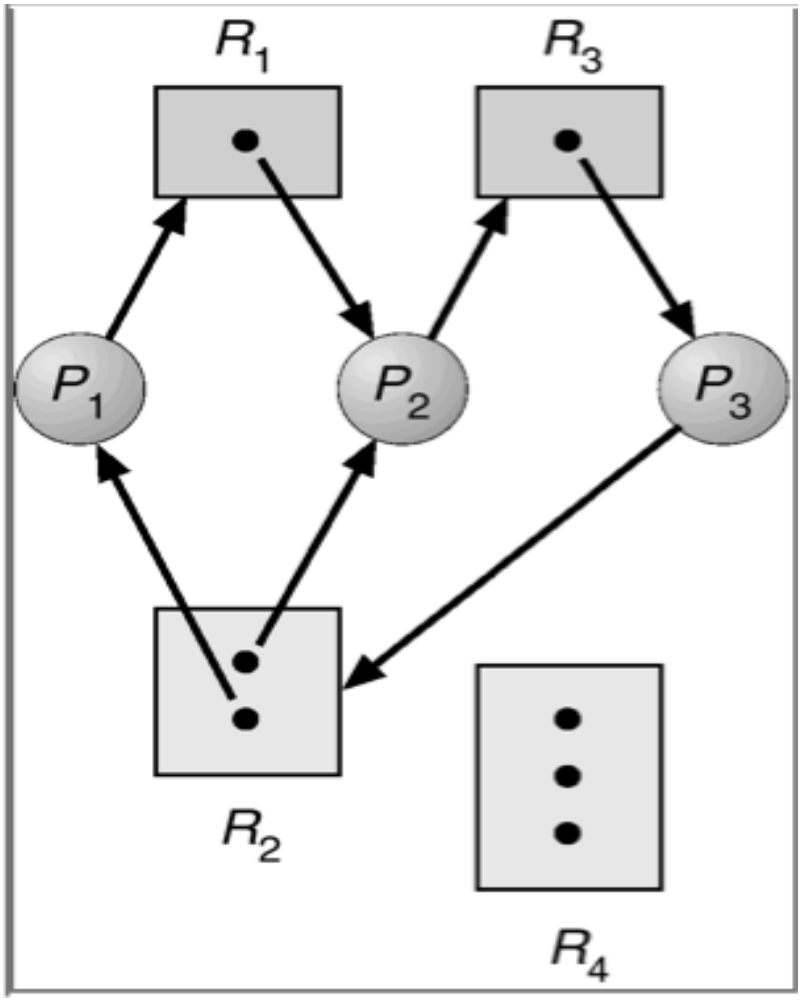
- P_i is holding an instance of R_j



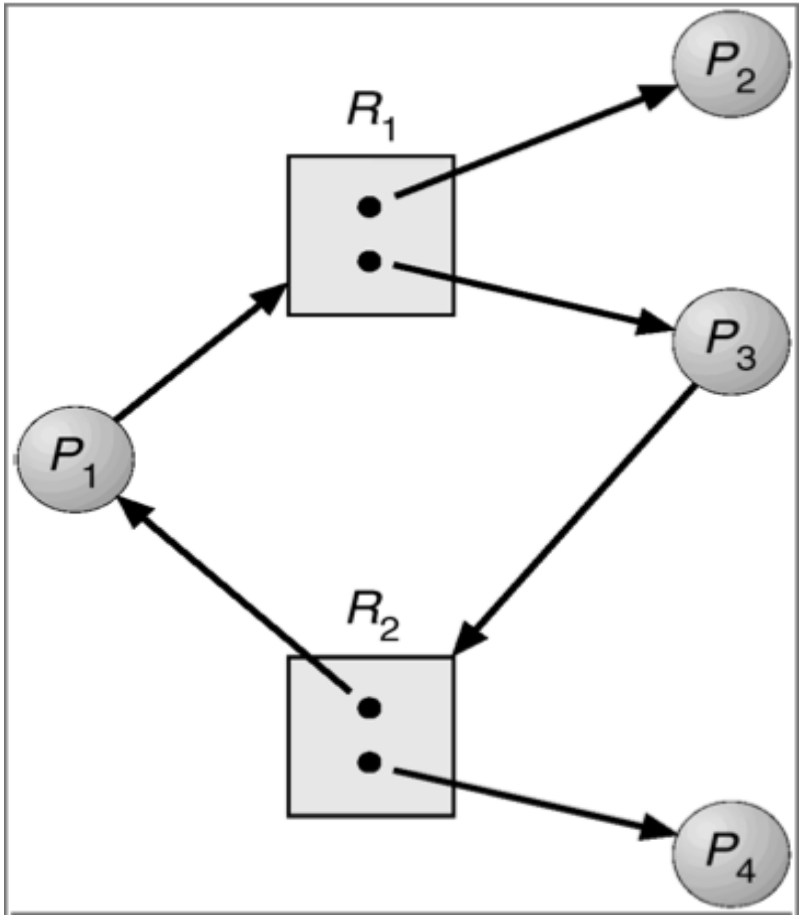
Resource-Allocation Graph



Resource-Allocation Graph With A Deadlock

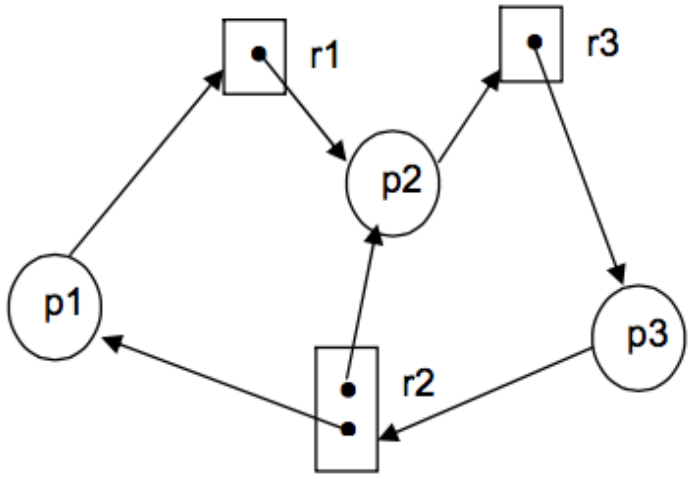


Resource-Allocation Graph With A Cycle But No Deadlock



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.



There are three cycles, so a deadlock may exists. Actually p1, p2 and p3 are deadlocked

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state.
- 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

Restrain the ways request can be made. To prevent the system from deadlocks, one of the four discussed conditions that may create a deadlock should be discarded. The methods for those conditions are as follows:

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources. In general, we do not have systems with all resources being sharable. Some resources like printers, processing units are non-sharable. So it is not possible to prevent deadlocks by denying mutual exclusion.
- **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.
  - Low resource utilization; starvation possible.

Deadlock Prevention

Restrain the ways request can be made.

- **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.
  For example, set priorities for r1 = 1, r2 = 2, r3 = 3, and r4 = 4. With these priorities, if process P wants to use r1 and r3, it should first request r1, then r3.

Deadlock Avoidance

Given some additional information on how each process will request resources, it is possible to construct an algorithm that will avoid deadlock states. The algorithm will dynamically examine the resource allocation operations to ensure that there won't be a circular wait on resources.

When a process requests a resource that is already available, the system must decide whether that resource can immediately be allocated or not. The resource is immediately allocated only if it leaves the system in a safe state.

A state is safe if the system can allocate resources to each process in some order avoiding a deadlock. A deadlock state is an unsafe state.

Deadlock Avoidance

Example:
Consider a system with 12 tape drives. Assume there are three processes : p1, p2, p3. Assume we know the maximum number of tape drives that each process may request:
p1 : 10, p2 : 4, p3 : 9
Suppose at time t_{now}, 9 tape drives are allocated as follows :
p1 : 5, p2 : 2, p3 : 2
So, we have three more tape drives which are free.
This system is in a safe state because it we sequence processes as: $< p2, p1, p3 >$, then p2 can get two more tape drives and it finishes its job, and returns four tape drives to the system. Then the system will have 5 free tape drives. Allocate all of them to p1, it gets 10 tape drives and finishes its job. p1 then returns all 10 drives to the system. Then p3 can get 7 more tape drives and it does its job.
It is possible to go from a safe state to an unsafe state:

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.

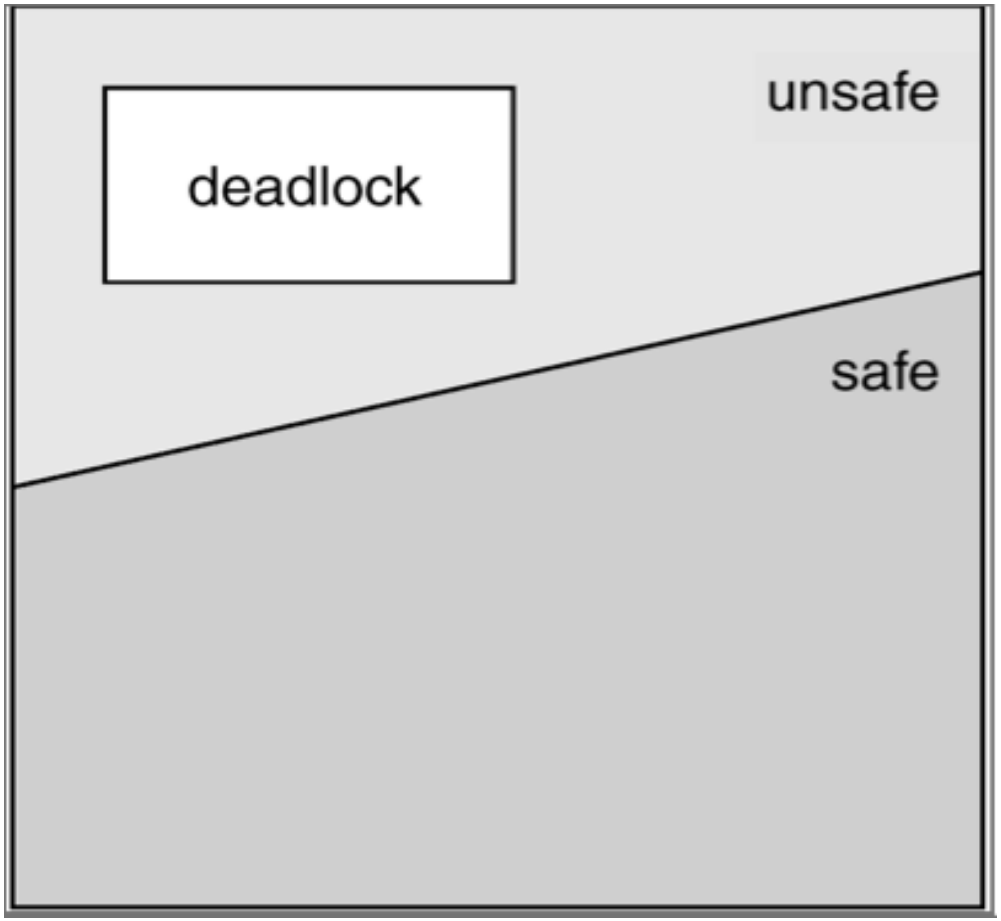
Safe State

- 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 safe sequence of all processes.
- Sequence $\langle P_1, P_2, \cdots, P_n \rangle$ is safe if 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 $j < i$.
  - 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.

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.

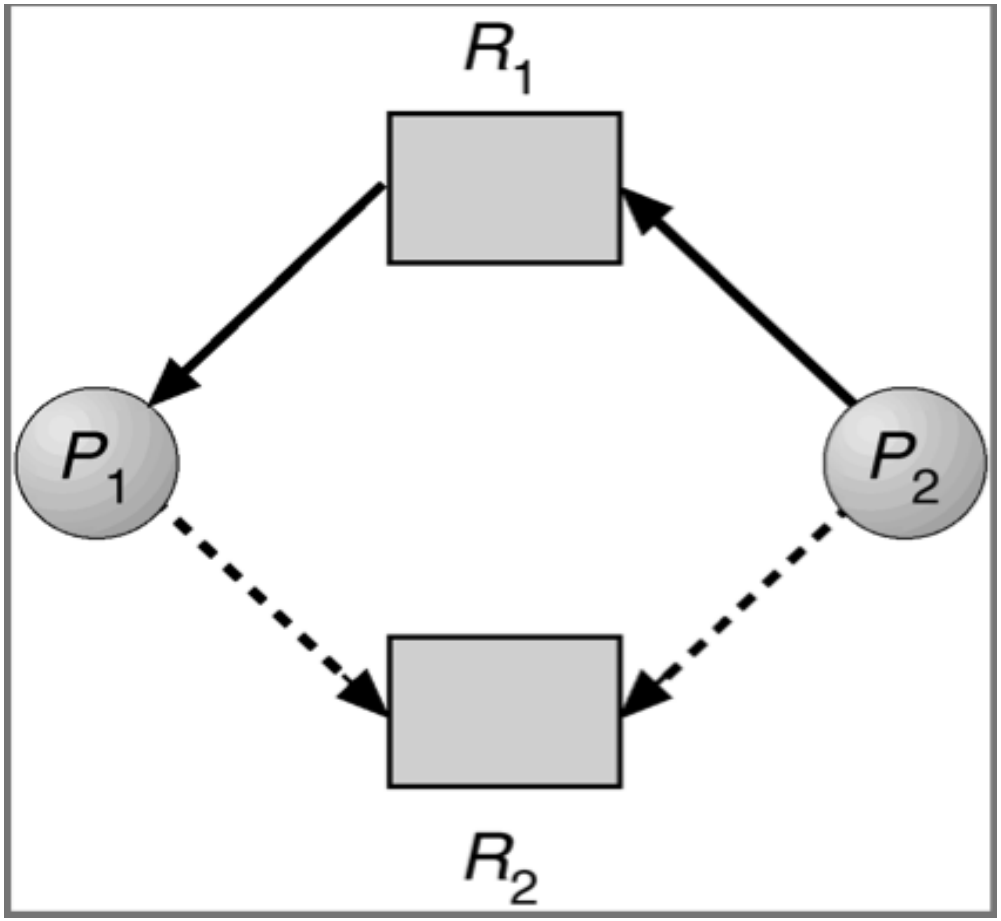
Safe, Unsafe , Deadlock State



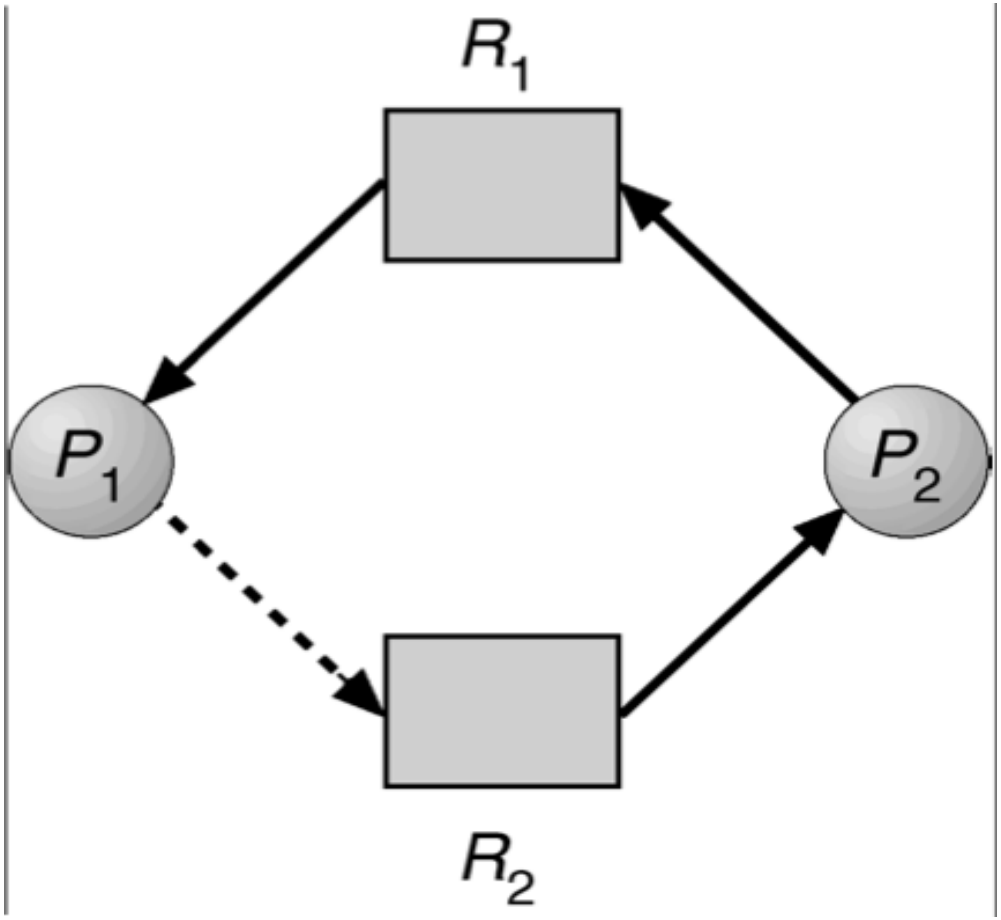
Resource-Allocation Graph Algorithm

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- 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 For Deadlock Avoidance



Unsafe State In 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 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.

$$\textbf{Need}[i,j] = \textbf{Max}[i,j] - \textbf{Allocation}[i,j].$$

Safety Algorithm

Let Work and Finish be vectors of length m and n, respectively.

1. Initialize Work = Available, Finish [j] = false, for all j.

2. Find an i such that Finish [i] = false and Need(i) ≤ Work

   If no such i is found, go to step **4.**

3. If an i is found, then for that i, do :

   Work = Work + Allocation(i)
   Finish [i] = true

   Go to step **2.**

4. If Finish [j] = true for all j, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request=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:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

   - 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 P0 through P4; 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

- 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* $\leq$ *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 1	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?

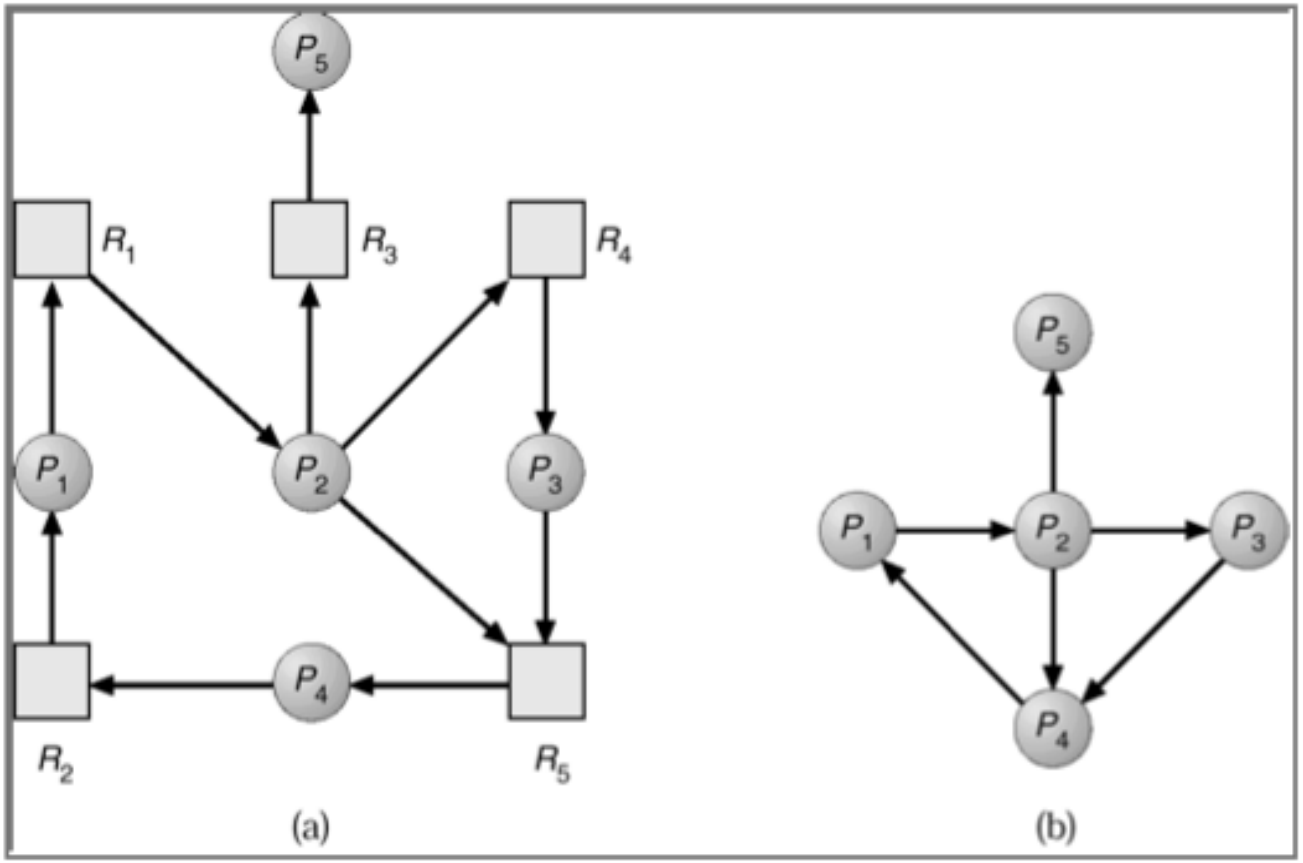
Deadlock Detection

- 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.
- 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 and Wait-for Graph



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 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 $i = 1,2, \ldots, n$, 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.

Detection Algorithm

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 $O(mxn^2)$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P0 through P4; 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

- P_2 requests an additional instance of type C.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
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. (Which resource(s) is/are to be preempted from which process?)
- Rollback – return to some safe state, restart process for that state. If we preempt a resource from a process, roll the process back to some safe state and make it continue.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

    allowing the use of the optimal approach for each of resources in the system.
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.

Traffic Deadlock