Deadlocks

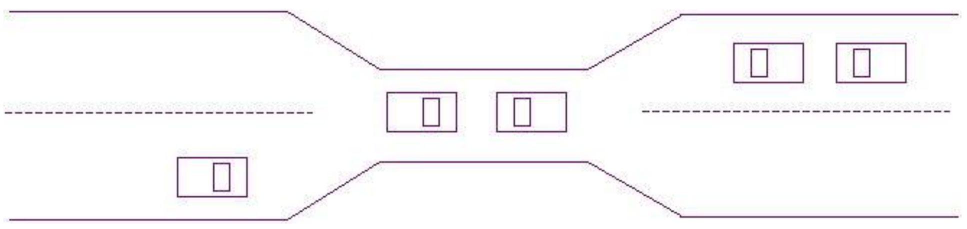
The Deadlock Problem

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. P1 and P2 each hold one tape drive and each needs another one.
- Example
  - semaphores A and B, initialized to 1

P1	P2
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.

System Model

- Resource types R1, R2, . . ., Rm
  - CPU cycles, memory space, I/O devices
- Each resource type Ri has Wi instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

1. Mutual exclusion:

Only one process at a time can use a resource.

2. Hold and Wait:

A process holding at least one resource is waiting to acquire additional resources held by other processes

3. No preemption

A resource can be released only voluntarily by the process holding it, after that process has completed its task.

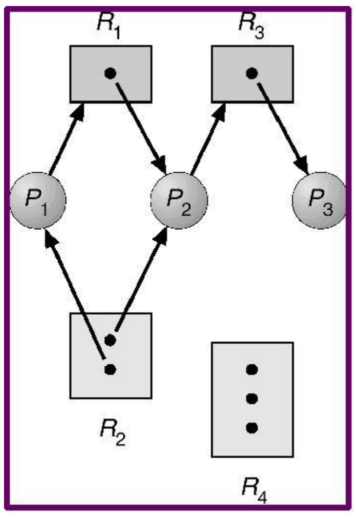
4. Circular wait:

There exists a set {P0, P1, …, P0} of waiting processes such that:
P0 is waiting for a resource that is held by P1,
P1 is waiting for a resource that is held by P2,
…,
Pn–1 is waiting for a resource that is held by Pn, and
Pn is waiting for p0

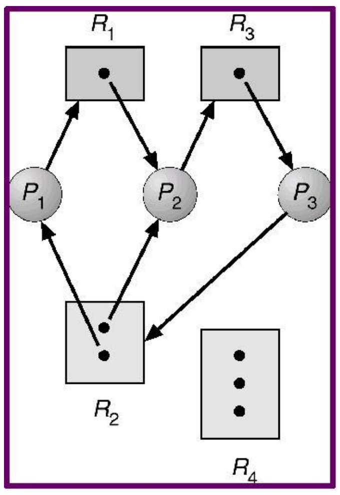
Resource Allocation Graph

- V is partitioned into two types:
  - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
  - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.
- request edge – directed edge P1 --> Rj
- assignment edge – directed edge Rj --> Pi

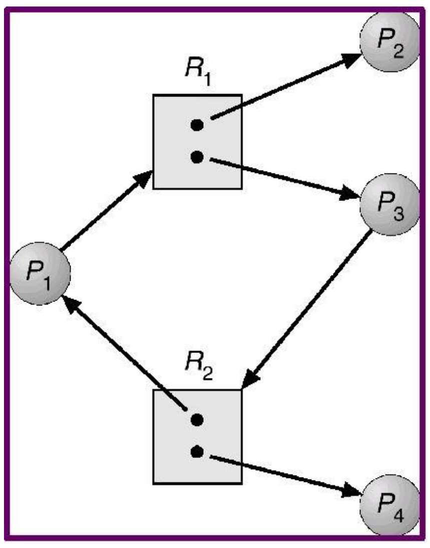
Resource Allocation Graph Example



Resource Allocation Graph with a Deadlock



Resource Allocation Graph: Cycle but no Deadlock



Basics

- If graph contains no cycles: no deadlock.
- If graph contains a cycle:
    - If only one instance per resource type, then deadlock.
    - If several instances per resource type, possibility of deadlock.

Methods of Handling Deadlock

Methods of Handling Deadlocks

1. Ensure that the system will never enter a deadlock state.
2. Allow the system to enter a deadlock state and then recover.
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX and Windows.

Requirements

Mutual Exclusion

Not required for sharable resources;
Must hold for non-sharable resources.

Deadlock Prevention: 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.
- Downside: Low resource utilization; starvation possible.
- Problematic with dynamic resource requests

Deadlock Prevention: 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

Deadlock Avoidance

- 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 <P1, P2, …, Pn> is safe if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j<I.
  - If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
  - When Pi terminates, Pi+1 can obtain its needed resources, and so on.

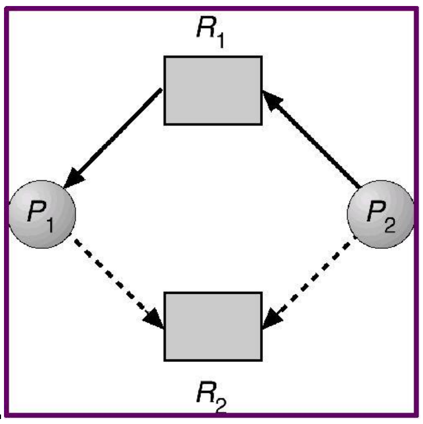
Basic Facts

- If a system is in safe state ==> no deadlocks.
- If a system is in unsafe state ==> possibility of deadlock.
- Avoidance ==> ensure that a system will never enter an unsafe state.

Resource Allocation Graph Algorithm

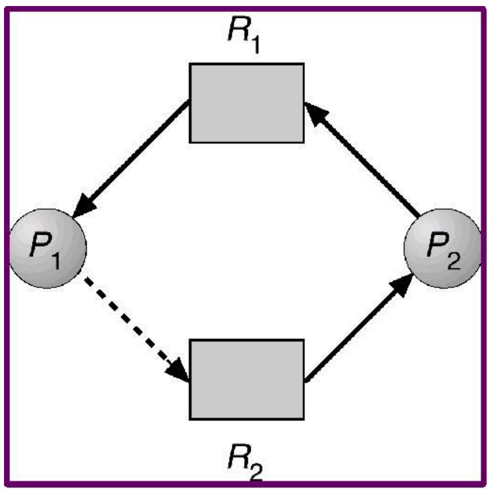
- Claim edge Pi --> Rj indicated that process Pj may request resource Rj; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed a priori in the system

Resource-Allocation Graph For Deadlock Avoidance



.

Unsafe State



Banker's Algorithm

The definitive algorithm for preventing deadlock. It detects dangerous situations, and denies requests for resources if they could place the system in a dangerous state.

- Each process must request maximum required resources at start
- The Banker's Algorithm checks for safety when a request is made
- When a process requests a resource it may have to wait if it would create an unsafe state
- 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 Rj available.
- **Max**: n x m matrix. If Max [i,j]== k, then process Pi may request at most k instances of resource type Rj.
- **Allocation**: n x m matrix. If Allocation[i,j] == k then Pi is currently allocated k instances of Rj.
- **Need**: n x m matrix. If Need[i,j] == k, then Pi may need k more instances of Rj 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 = 1,2, …, n.
2. Find and i such that both:
   - Finish [i] = false
   - Need[i] <= Work[i]
   - If no such i exists, go to step 4.
3. Work = Work + Allocation
   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

Request_i = request vector for process Pi. If Request_i [j] == k then process Pi wants k instances of resource type Rj.

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 state as follows:
   Available = Available - Request_i;
   Allocation_i = Allocation_i + Request_i;
   Need_i = Need_i – Request_i;
   - if (safe) allocate the resources to P_i
   - if (unsafe) P_i must wait and the previous state is restored

Example

- 5 processes P0 through P4; 3 resource types A (10 instances), B (5 instances), and C (7 instances).
- Snapshot at time T0:

	Alloc	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

Example (cont)

- The content of the matrix. Need is defined to be 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 < P1, P3, P4, P2, P0> satisfies safety criteria.

Example (cont)

P1 Request (1,0,2)

- Check that Request <= Available (that is, (1,0,2) <= (3,3,2): true.

	Alloc	Need	Available
	A B C	A B C	A B C
P0	0 1 0	7 4 3	2 3 0
P1	3 0 2	0 2 0	
P2	3 0 1	6 0 0	
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Deadlock Detectioin

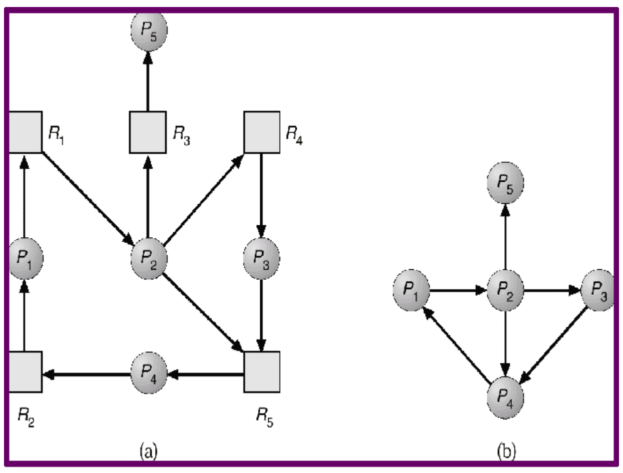
Deadlock Detection

- Allow system to enter deadlock state
- Use a Detection algorithm to detect deadlock
- Employ a Recovery scheme

Single Instance of Each Resource Type

- Maintain wait-for graph
- Nodes are processes.
- Pi --> Pj if Pi is waiting for Pj.
- 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 and Wait For Graph



Resource Graph **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 Pi is requesting k more instances of resource type. Rj.

Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively Initialize:
    1. Work = Available
    2. For i = 1,2, …, n, if Allocation[i] != 0, then
       Finish[i] = false;
       otherwise, Finish[i] = true.
2. Find an index i such that both:
    1. Finish[i] == false
    2. (b) Request[i] <= Work
    3. If no such i exists, go to step 4.
3. Work = Work + Allocationi
   Finish[i] = true
   go to step 2.
4. If Finish[i] == false, for some i, 1 <= i <= n, then the system is in deadlock state
   . Moreover, if Finish[i] == false, then Pi is deadlocked.

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

Example

- Five processes P0 through P4; three resource types
- A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T0:

	Allocat	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

- Sequence <P0, P2, P3, P1, P4> will result in Finish[i] = true for all i.

Example (cont)

- P2 requests an additional instance of type C.

	Request
	A B C
P0	0 0 0
P1	2 0 1
P2	0 0 1
P3	1 0 0
P4	0 0 2

- State of system?

    - Can reclaim resources held by process P0, but insufficient resources to fulfill other processes; requests.
    - Deadlock exists, consisting of processes P1, P2, P3, and P4.

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

Recovery From Deadlock

- Processs Termination
- Resource Preemption

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?

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.

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.