ICS 143 - Principles of Operating Systems

Lectures 8 and 9 - Deadlocks

Prof. Nalini Venkatasubramanian
nalini@ics.uci.edu

Outline

- System Model
- Deadlock Characterization
- Methods for handling deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

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 1
    - System has 2 tape drives. P1 and P2 each hold one tape drive and each needs the other one.
  - Example 2
    - Semaphores A and B each initialized to 1

P0	*P1*
wait(A)	*wait(B)*
wait(B)	*wait(A)*

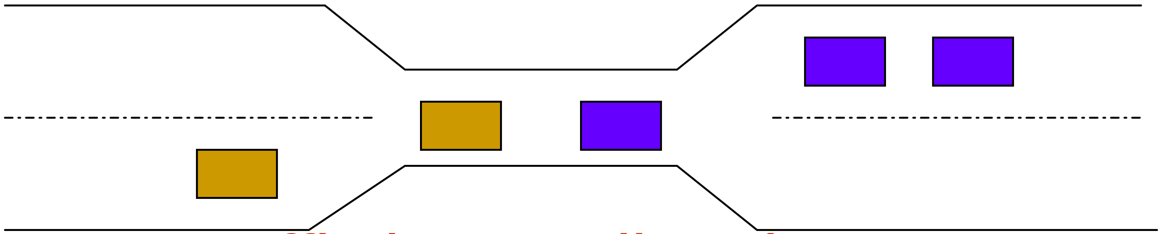
Definitions

- A process is *deadlocked* if it is waiting for an event that will never occur.

  Typically, more than one process will be involved in a deadlock (the deadly embrace).

- A process is *indefinitely postponed* if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes,
  - i.e. the process is ready to proceed but never gets the CPU.

Example - Bridge Crossing



- Assume traffic in one direction.
  - Each section of the bridge is viewed as a resource.
- If a deadlock occurs, it can be resolved only if one car backs up (preempt resources and rollback).
  - Several cars may have to be backed up if a deadlock occurs.
  - Starvation is possible

Resources

- Resource
  - commodity required by a process to execute
- Resources can be of several types
  - Serially Reusable Resources
    - CPU cycles, memory space, I/O devices, files
    - acquire -> use -> release
  - Consumable Resources
    - Produced by a process, needed by a process - e.g. Messages, buffers of information, interrupts
    - create ->acquire ->use
    - Resource ceases to exist after it has been used

System Model

- Resource types
  - $R_1, R_2, \ldots. R_m$
- Each resource type R_i has W_i instances
- Assume serially reusable resources
  - request -> use -> release

Conditions for Deadlock

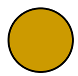
- The following 4 conditions are necessary and sufficient for deadlock (must hold simultaneously)
  - Mutual Exclusion:
    - Only once process at a time can use the resource.
  - Hold and Wait:
    - Processes hold resources already allocated to them while waiting for other resources.
  - No preemption:
    - Resources are released by processes holding them only after that process has completed its task.
  - Circular wait:
    - A circular chain of processes exists in which each process waits for one or more resources held by the next process in the chain.

Resource Allocation Graph

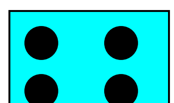
- A set of vertices V and a set of edges E
- V is partitioned into 2 types
  - P = {P1, P2,…,Pn} - the set of processes in the system
  - R = {R1, R2,…,Rn} - the set of resource types in the system
- Two kinds of edges
  - Request edge - Directed edge Pi ---> Rj
  - Assignment edge - Directed edge Rj ----> Pi

Resource Allocation Graph

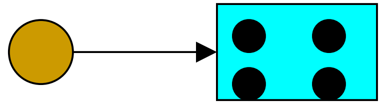
- Process



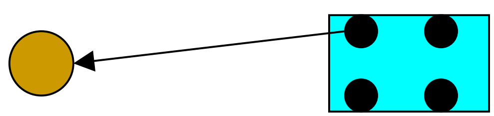
- Resource type with 4 instances



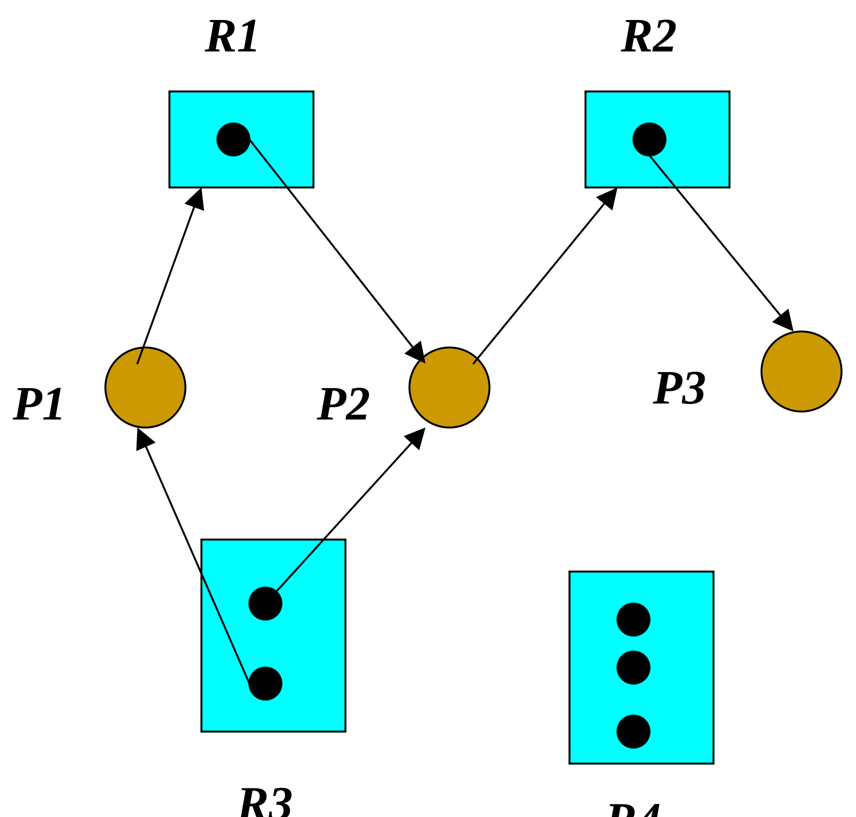
- Pi requests instance of Rj



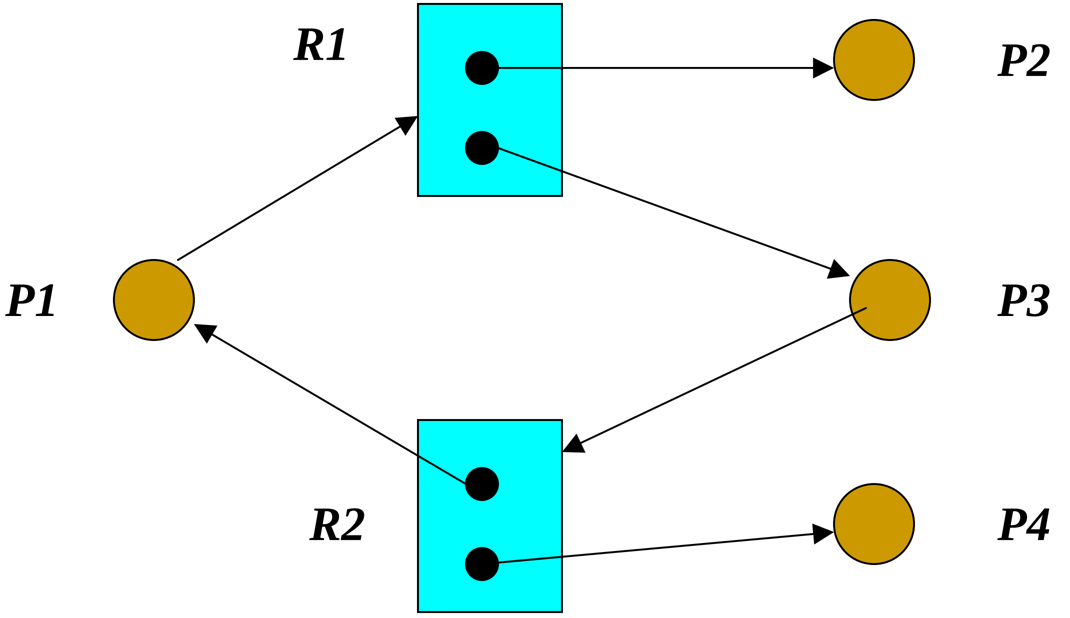
- Pi is holding an instance of Rj



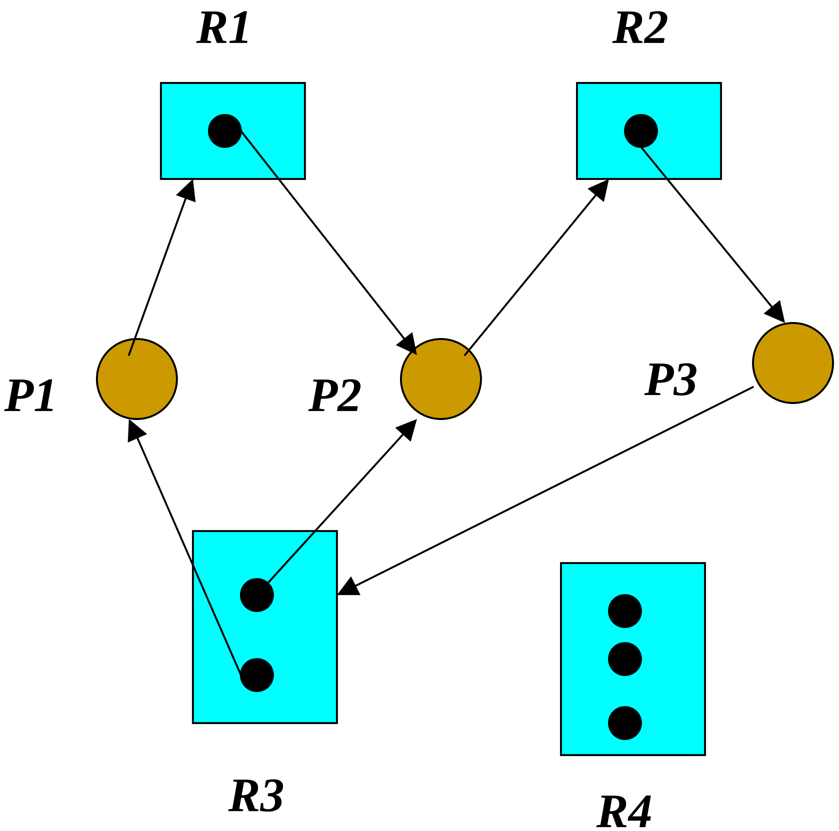
Graph with no cycles

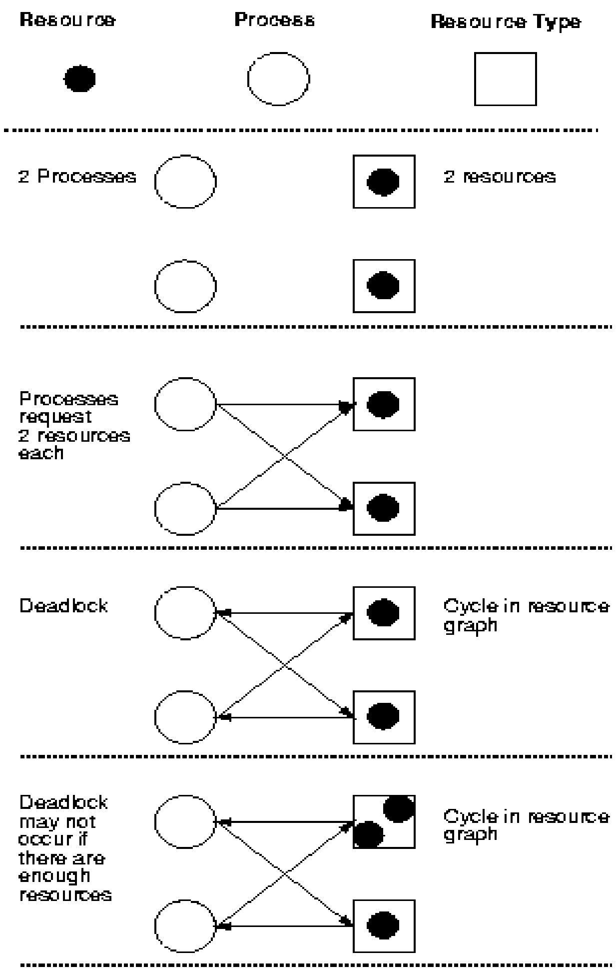


Graph with cycles



Graph with cycles and deadlock




Resource
Process
Resource Type
2 Processes
2 resources
Processes request 2 resources each
Deadlock
Cycle in resource graph
Deadlock may not occur if there are enough resources
Cycle in resource graph

Basic facts

- 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 for handling deadlocks

- Ensure that the system will never enter a deadlock state.
- Allow the system to potentially enter a deadlock state, detect it and then recover
- Ignore the problem and pretend that deadlocks never occur in the system;
  - Used by many operating systems, e.g. UNIX

Deadlock Management

- Prevention
  - Design the system in such a way that deadlocks can never occur
- Avoidance
  - Impose less stringent conditions than for prevention, allowing the possibility of deadlock but sidestepping it as it occurs.
- Detection
  - Allow possibility of deadlock, determine if deadlock has occurred and which processes and resources are involved.
- Recovery
  - After detection, clear the problem, allow processes to complete and resources to be reused. May involve destroying and restarting processes.

Deadlock Prevention

- If any one of the conditions for deadlock (with reusable resources) is denied, deadlock is impossible.
- Restrain ways in which requests can be made
  - Mutual Exclusion
    - non-issue for sharable resources
    - cannot deny this for non-sharable resources (important)
  - Hold and Wait - guarantee that when a process requests a resource, it does not hold other resources.
    - Force each process to acquire all the required resources at once. Process cannot proceed until all resources have been acquired.
    - Low resource utilization, starvation possible

Deadlock Prevention (cont.)

- No Preemption
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, the process releases the resources currently being held.
  - 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.
  - Require that processes request resources in increasing order of enumeration; if a resource of type N is held, process can only request resources of types > N.

Deadlock Avoidance

- Set of resources, set of customers, banker
- Rules
  - Each customer tells banker maximum number of resources it needs.
  - Customer borrows resources from banker.
  - Customer returns resources to banker.
  - Customer eventually pays back loan.
- Banker only lends resources if the system will be in a *safe state* after the loan.

Deadlock Avoidance

- Requires that the system has some additional apriori 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 <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 Pj with j<i.
  - If Pi resource needs are not available, 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...

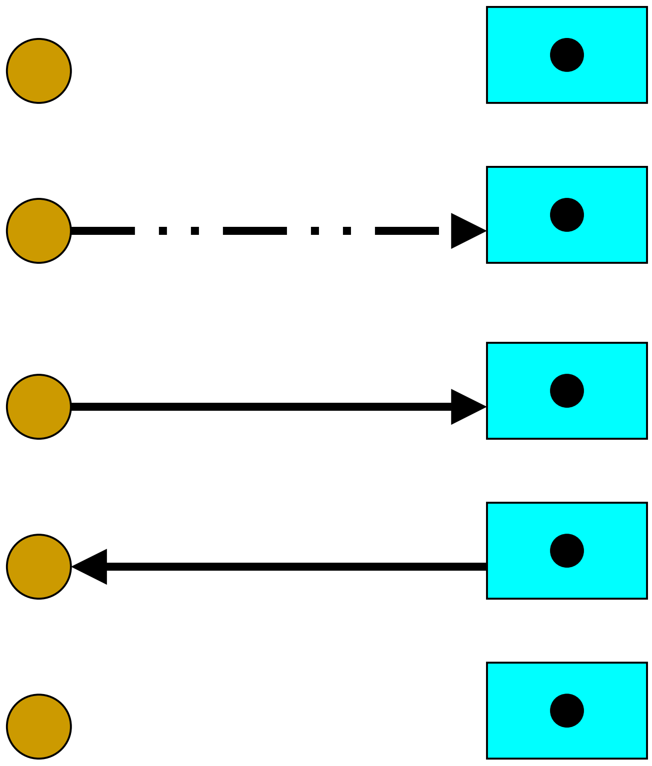
Basic Facts

- If a system is in a safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never reach an unsafe state.

Resource Allocation Graph Algorithm

- Used for deadlock avoidance when there is only one instance of each resource type.
  - Claim edge: $P_i \rightarrow R_j$ indicates that process Pi 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 claim edge.
  - Resources must be claimed a priori in the system.
- If request assignment does not result in the formation of a cycle in the resource allocation graph - safe state, else unsafe state.

Claim Graph



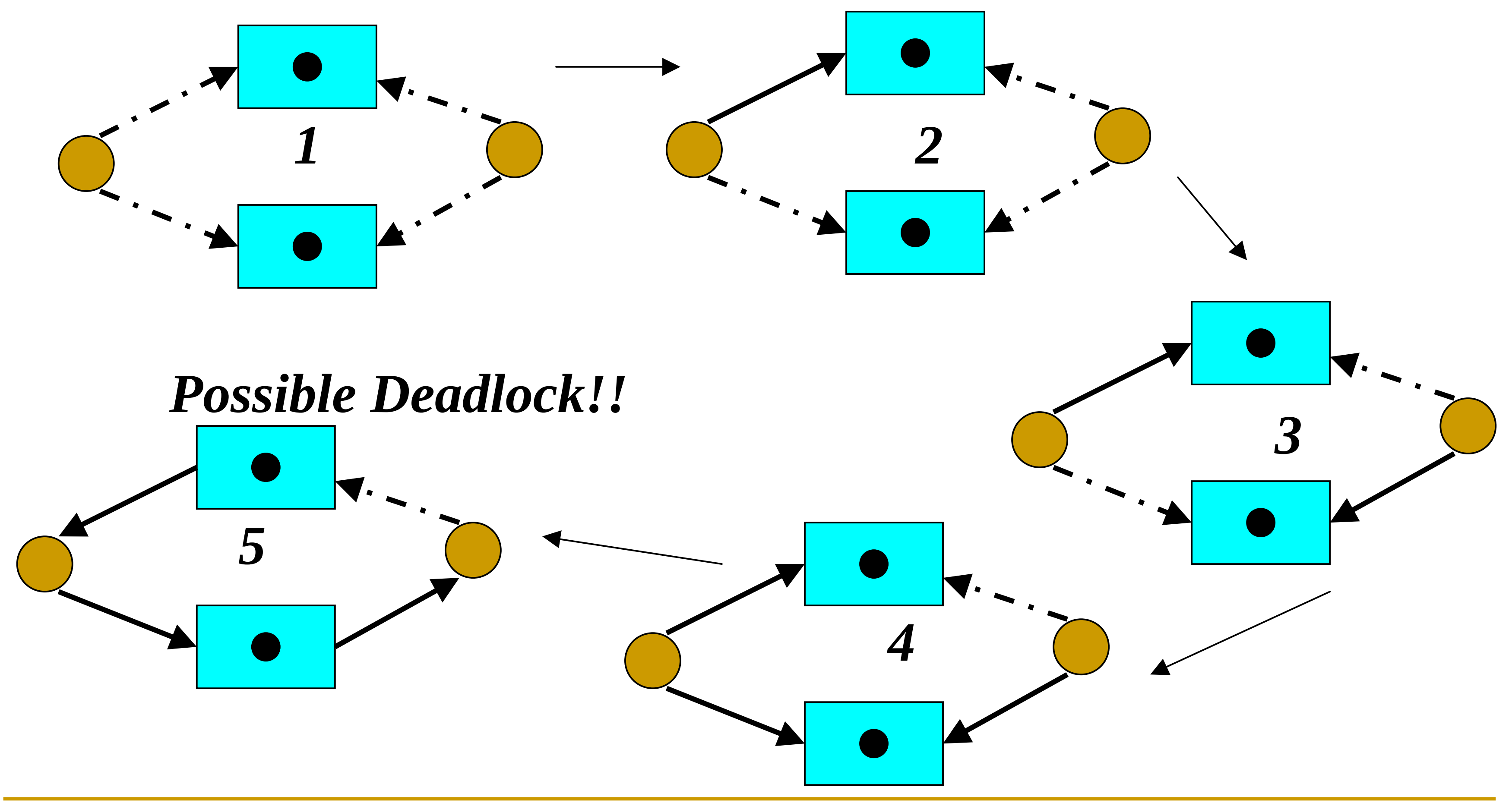
Process claims resource

Process requests resource

Process is assigned resource

Process releases resource

Claim Graph



Banker's Algorithm

- Used for multiple instances of each resource type.
- Each process must a priori claim maximum use of each resource type.
- 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 resource types.
  - *Available*: Vector of length m. If *Available*[j] = k, there are k instances of resource type R_j available.
  - *Max*: $n \times m$ matrix. If *Max*[i,j] = k, then process P_i may request at most k instances of resource type R_j.
  - *Allocation*: $n \times m$ matrix. If *Allocation*[i,j] = k, then process P_i is currently allocated k instances of resource type R_j.
  - *Need*: $n \times m$ matrix. If *Need*[i,j] = k, then process P_i may need k more instances of resource type R_j to complete its task.

Need[i,j] = *Max*[i,j] - *Allocation*[i,j]

Safety Algorithm

- Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize
  - *Work* := *Available*
  - *Finish*[*i*] := *false* for *i* = 1,2,…,n.
- Find an *i* (i.e. process *Pi*) such that both:
  - *Finish*[*i*] = *false*
  - *Need_i* <= *Work*
  - If no such *i* exists, go to step 4.
- *Work* := *Work* + Allocation_*i*
  - *Finish*[*i*] := *true*
  - go to step 2
- If *Finish*[*i*] = *true* for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process *Pi*

- Request_i = request vector for process Pi. If Request_i[j] = k, then process Pi wants k instances of resource type Rj.
  - STEP 1: If *Request*(*i*) ≤ *Need*(*i*), go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
  - STEP 2: If *Request*(*i*) ≤ *Available*, go to step 3. Otherwise, Pi must wait since resources are not available.
  - STEP 3: Pretend to allocate requested resources to Pi by modifying the state as follows:

    Available := *Available* - *Request* (i)*;*

    Allocation (*i*) := *Allocation* (*i*) + *Request* (*i*)*;*

    Need (*i*) := *Need* (*i*) - *Request* (*i*)*;*
  - If safe ⇒ resources are allocated to *Pi*.
  - If unsafe ⇒ *Pi* must wait and the old resource-allocation state is restored.

Example of Banker's Algorithm

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

	Allocation			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*.
- The system is in a safe state since the sequence <*P*1,*P*3,*P*4,*P*2,*P*0> satisfies safety criteria.

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

Example: P1 requests (1,0,2)

- Check to see that Request ≤ Available
  - ((1,0,2) ≤ (3,3,2)) ⇒ true.

	Allocation			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	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Example (cont.)

- Executing the 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 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.

Several instances of a resource type

- Data Structures
  - *Available*: Vector of length m. If *Available*[j] = k, there are k instances of resource type R_j available.
  - *Allocation*: $n \times m$ matrix. If *Allocation*[i,j] = k, then process P_i is currently allocated k instances of resource type R_j.
  - *Request* : An $n \times m$ 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 .

Deadlock Detection Algorithm

- Step 1: Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize
  - *Work* := *Available*
  - For $i = 1,2,\ldots,n$, if $Allocation(i) \neq 0$, then $Finish[i]$:= *false*, otherwise $Finish[i]$:= *true.*
- Step 2: Find an index *i* such that both:
  - $Finish[i]$ = *false*
  - $Request\ (i) \leq Work$
  - If no such *i* exists, go to step 4.

Deadlock Detection Algorithm

- Step 3: *Work* := *Work* + Allocation(*i*)
  - *Finish*[*i*] := *true*
  - go to step 2
- Step 4: If *Finish*[*i*] = *false* for some *i*, $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if *Finish*[*i*] = *false*, then *Pi* is deadlocked.

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

Example of Detection Algorithm

- 5 processes - *P*0 - *P*4; 3 resource types - *A*(7 instances), *B*(2 instances), *C*(6 instances)
- Snapshot at time *T* 0: <*P*0,*P*2,*P*3,*P*1,*P*4> will result in *Finish*[*i*] = true for all *i*.

	Allocation			Max			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			

Example (cont.)

- P2 requests an additional instance of type C.
- State of system
  - Can reclaim resources held by process P0, but insufficient resources to fulfill other processes' requests.
  - Deadlock exists, consisting of P 1,P 2,P 3 and P 4.

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

Detection-Algorithm Use

- 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
- How often --
  - Every time a request for allocation cannot be granted immediately
    - Allows us to detect set of deadlocked processes and process that “caused” deadlock. Extra overhead.
    - Every hour or whenever CPU utilization drops.
  - With arbitrary invocation there may be many cycles in the resource graph and 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 the 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 from 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 class of resources in the system.
- Partition resources into hierarchically ordered classes.
  - Use most appropriate technique for handling deadlocks within each class.