Deadlocks

Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

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 sharable resources or objects such as devices, files or memory.
  - P_1 and P_2 each hold one resource and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1: deadlock!

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

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU , memory, I/O devices, files, data, etc…
- Each resource type R_i may have a number of instances.
- Each process can conduct three basic operations on a resource:
  - request
  - use
  - release

Deadlock Characterization

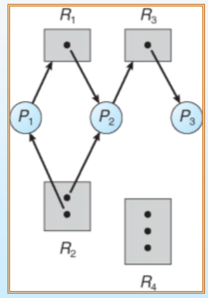
Deadlocks can occurs if the following four conditions hold simultaneously.

- **Mutual exclusion:** only one process can use a resource at a time.
- **Hold and wait:** a process holding a resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released by the process holding it, only after it has finished with it.
- **Circular wait:** Given a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes, in which P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

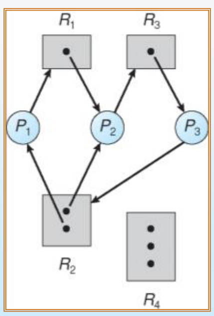
Resource-Allocation can be represented by a Graph

- Let
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- Let
  - request be directed edge $P_1 \rightarrow R_j$
  - assignment be directed edge $R_j \rightarrow P_i$

Multi Instance Resource Allocation Graph: Example: No deadlock



Resource Allocation Graph: Example: with deadlock



Methods for Handling Deadlocks

- There may be three approaches:
  - 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;. When the system is locked, restart it… No guarantee for consistency…
    - used by most operating systems, including UNIX

Deadlock Prevention

- **Restrain the ways request can be made:**
  - **Do not use Mutual Exclusion**;
  - **Prevent Hold and Wait** :-
    - must guarantee that whenever a process requests a resource, it does not hold any resources.
      - Require process to request and be allocated all its resources before it begins execution.
        - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **Alow 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 all the resources it is requesting.
- **Prevent Circular Wait** :-
  - impose a total ordering of all resource types, and require that each process requests resources, for example in an increasing order of enumeration.

Deadlock Avoidance

- Requires that the system has some *a priori* information available for the resource requests
- This may not be practical, but it is possible…
  - Simplest 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 not 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 regarding resource allocation

- When a process requests an available resource, system must decide if immediate allocation will leave the system in a **safe state**, no deadlock state.
- System is in safe state if there exists a deadlock free safe sequence of resource allocation for of all processes.

Determination of Safe State

- Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the required resources 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.

Summary of safe state

- 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**, with proper order of resource allocation.

Algorithms

to establish

the safe state

Banker's Algorithm

- Principle:
  - Resources have multiple instances.
  - Each process must have a priori claim for maximum instances.
  - When a process requests a resource it must be prepared to wait until it is available.
  - When a process gets all its resources it must return them, in a finite length of time.

Banker's Algorithm: Data Structures

- Size:
  - *n* = number of processes,
  - *m* = number of resource types.
- *Available resource vector, R* [] of length *m*.
  - If *R* [*j*] = *k*, there are *k* instances of resource type R_j available.
- *Max resource matrix, of size n x m*, Max[,].
  - If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.
- *Allocation resource matrix of size n* x *m, Allocation[,]*.
  - If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j.
- *Need matrix, of size n* x *m Need[,]*.
  - 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*].

Banker's Algorithm: Safe state test

Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. At the beginning of the test initialize:

Work[] = Available[], at a point in time

Finish [*i*] = *false* for *I* =0, 1,2,3 …, *n-1.*

do for all processes,i=0,…,n-1

{ If (*Finish* [*i*] = *false && $Need_i \le$ Work)*

{$Work_i = Work_i + Allocation_i$ *Finish*[*i*] = *true}*

}

If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Otherwise the system is unsafe

Request handling algorithm for Process P_i

If $Request_i$ [*j*] = *k* then process P_i wants *k* instances of resource type R_j.

1. If $Request_i > Need_i$, process has exceeded its maximum claim, error case, otherwise next step
2. If $Request_i \le Available$.

   {Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available = Available - Request_i$;

   $Allocation_i = Allocation_i + Request_i$;

   $Need_i = Need_i – Request_i$;

   Test the state:

   If safe state ⇒ the resources are allocated to Pi.

   If unsafe state ⇒ Pi must wait, and the old resource-allocation state is restored}

   Else

   Pi must wait

Example of Banker's Algorithm

- n=5, 3
- m= 3 resource types *A, B, and C of instances 10, 5, and 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 (Cont.)

- 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

- Test if the system is safe:
- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Example: Assume P_1 Request (1,0,2) (Cont.)

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

Deadlock Detection and Recovery

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

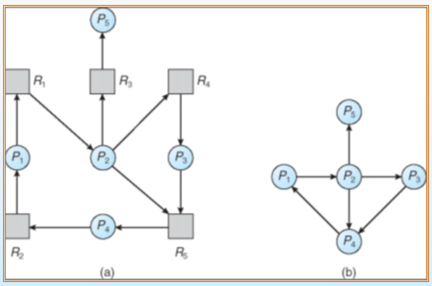
Wait-for-graph: Assume Single Instance for Each Resource Type

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:
   - *Finish*[*i*] == *false*
   - $Request_i \leq Work$,

   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*] == 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.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types with instances A (7), *B* (2), and *C* (6).
- 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	

- Test the state:
- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- 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

- Test the state: ?
  - 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

- Option1: Abort all deadlocked processes.
- Option2: Abort one process at a time until the deadlock cycle is eliminated.
- Aborting order:
  - 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, because of this particular abortion.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Select a victim, so that the cost is minimized.
- Rollback, so that the system returns to some safe state to restart the process from that state onwards.
- Starvation is possible if the same process is always picked up as the victim.
  - In the case number of rollbacks may be included in the decision.

End