Chapter 7: Deadlocks

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 disk drives
  - \( P_1 \) and \( P_2 \) each hold one disk drive and each needs another one
- Example
  - semaphores \( A \) and \( B \), initialized to 1
  - \( P_0 \), \( P_1 \)
  - \( P_0 \) wait (\( A \)); \( P_1 \) wait (\( B \));
  - \( P_1 \) wait (\( B \)); \( P_0 \) wait (\( A \))

Seems like not so big of a deal? Well...

Complicating consideration

- At first glance, it would seem that an obvious solution would exist:
  - “Give me that drive – I need it”
  - “I’m higher priority – release your lock and let me use the resource!”
- These ignore a couple of critical issues:
  - Processes typically have a local view of the world: they just see themselves and the system as presented by the OS.
  - They are unaware of other processes.
  - Locks are often acquired in groups – atomic operations of any complexity require more than one lock.
  - A high priority process forcing a lower one to yield one of the group of locks could violate atomicity within the lower.

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
- Note – Most OSes do not prevent or deal with deadlocks

Deadlock

- One of the main concurrency-related errors that we wish to either:
  - Avoid
  - Detect
  - Recover from

This chapter we look in detail at what leads to deadlock, and how we can attempt to detect and avoid it, or if all else fails, recover from.
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
  - release

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- 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 {$P_0, P_1, \ldots, P_n$} 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 $P_2$, …, $P_n$ is waiting for a resource that is held by $P_0$.

Resource-Allocation Graph

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

- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, 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
- request edge = directed edge $P_i \rightarrow R_j$
- assignment edge = directed edge $R_j \rightarrow P_i$

Terminology: This is a bipartite graph.

Resource-Allocation Graph (Cont.)

- Process
- Resource Type with 4 instances
- $P_i$ requests instance of $R_j$
- $P_i$ is holding an instance of $R_j$

Example of a Resource Allocation Graph

Resource Allocation Graph With A Deadlock

Observation: We have a cycle now!

Does cycle = deadlock?
Graph With A Cycle But No Deadlock

This graph has a cycle but no deadlock.

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
  - Avoidance
- Allow the system to enter a deadlock state and then recover
  - Detection and recovery
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX
  - Make it the application programmers problem.

Prevention

- One way to avoid deadlock is to prevent one of the four conditions that lead to deadlock from being possible.

Deadlock Prevention

Restrain the ways request can be made

- Eliminate Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources
- Eliminate 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

“Hold and wait” similar to suggestion to ask for all locks within a transaction at the beginning instead of incrementally in two-phase scheme. It works, but at the cost of performance and liveness.

Deadlock Prevention (Cont.)

- Eliminate 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
  - Prempted 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
  - Look familiar? (Consider the CPU as a resource)

- Eliminate Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - Don’t allow a request for a low-ordered resource after a higher-ordered one
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 sequence <P_1, P_2, ..., P_n> of ALL the processes is the systems such that 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
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_i 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 → no deadlocks
- If a system is in unsafe state → possibility of deadlock
- Avoidance → ensure that a system will never enter an unsafe state.

Avoidance algorithms

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

Safe, Unsafe, Deadlock State

Resource-Allocation Graph Scheme

- Claim edge P_i → R_j indicated that process P_i may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests 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

Unsafe State In Resource-Allocation Graph

Cycle created by granting resource to P2

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
- 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.

- \( \text{Available} \): Vector of length \( m \). If available \([j] = k \), there are \( k \) instances of resource type \( R_j \) available
- \( \text{Max} \): \( n \times m \) matrix. If \( \text{Max}[i,j] = k \), then process \( P_i \) may request at most \( k \) instances of resource type \( R_j \)
- \( \text{Allocation} \): \( n \times m \) matrix. If \( \text{Allocation}[i,j] = k \), then process \( P_i \) is currently allocated \( k \) instances of \( R_j \)
- \( \text{Need} \): \( n \times m \) matrix. If \( \text{Need}[i,j] = k \), then \( P_i \) may need \( k \) more instances of \( R_j \) to complete its task

Safety Algorithm

1. Let \( \text{Work} \) and \( \text{Finish} \) be vectors of length \( m \) and \( n \), respectively. Initialize:
   \( \text{Work} = \text{Available} \)
   \( \text{Finish}[i] = \text{false} \) for \( i = 0, 1, \ldots, n-1 \)
2. Find and \( i \) such that both:
   (a) \( \text{Finish}[i] = \text{false} \)
   (b) \( \text{Need}[i] \leq \text{Work} \)
   If no such \( i \) exists, go to step 4
3. \( \text{Work} = \text{Work} + \text{Allocation}[i] \)
   \( \text{Finish}[i] = \text{true} \)
   go to step 2
4. If \( \text{Finish}[i] = \text{true} \) for all \( i \), then the system is in a safe state
Looking closer

- Remember safe state means a sequence exists <P0, P1, ..., PN> such that PJ can execute if all P0...PJ(J-1) complete and release their resources for J in range 0...N.
- Bankers safe-check algorithm basically says:
  - Start with current set of free resources
  - Find a process that can use them that hasn’t run yet, put it at the end of the sequence, and contribute what it has allocated to the free pool.
  - To subsequent processes, it will have finished and freed them.
  - Repeat until either
    - All processes get put in sequence -> Safe!
    - We reach a set of processes that simply cannot proceed -> Unsafe!

Resource-Request Algorithm for Process Pi

Request = request vector for process Pi. If Request[i] = k then process Pi wants k instances of resource type R_i

1. If Request, = Need, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If Request, = Available, go to step 3. Otherwise Pi must wait, since resources are not available
3. Pretend to allocate requested resources to Pi by modifying the state as follows:
   - Available = Available – Request;
   - Allocation = Allocation + Request;
   - Need = Need – Request;
- If safe -> the 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 P_i, through P_4;
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_c:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P_0</td>
<td>0 1 0</td>
<td>7 5 3 A B C</td>
</tr>
<tr>
<td>P_1</td>
<td>2 0 0</td>
<td>3 2 2 A B C</td>
</tr>
<tr>
<td>P_2</td>
<td>3 0 2</td>
<td>9 0 2 A B C</td>
</tr>
<tr>
<td>P_3</td>
<td>2 1 1</td>
<td>2 2 2 A B C</td>
</tr>
<tr>
<td>P_4</td>
<td>0 0 2</td>
<td>4 3 3 A B C</td>
</tr>
</tbody>
</table>

To the blackboard!

Example (Cont.)

- The content of the matrix Need is defined to be Max – Allocation

<table>
<thead>
<tr>
<th>Need</th>
<th>A B C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P_1</td>
<td>1 2 2</td>
</tr>
<tr>
<td>P_2</td>
<td>6 0 0</td>
</tr>
<tr>
<td>P_3</td>
<td>0 1 1</td>
</tr>
<tr>
<td>P_4</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>

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

Example: P_i Request (1,0,2)

- Check that Request = Available (that is, (1,0,2) x (3,3,2) = true

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P_0</td>
<td>0 1 0</td>
<td>7 4 3 A B C</td>
</tr>
<tr>
<td>P_1</td>
<td>3 0 2</td>
<td>0 2 0 A B C</td>
</tr>
<tr>
<td>P_2</td>
<td>3 0 1</td>
<td>6 0 0 A B C</td>
</tr>
<tr>
<td>P_3</td>
<td>2 1 1</td>
<td>0 1 1 A B C</td>
</tr>
<tr>
<td>P_4</td>
<td>0 0 2</td>
<td>4 3 1 A B C</td>
</tr>
</tbody>
</table>

- Executing safety algorithm shows that sequence < P_0, P_1, P_2, P_3, P_4 > satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_4 be granted?

Left here
Logistics

- HWK#4 posted today after class
  - Covers ch. 6 and 7
- PA#2 will be posted Thursday
  - You will have two weeks
- Midterm
  - Grading is proceeding. Expect them by discussion this week

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 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 \( n \times m \) matrix defines the number of resources of each type currently allocated to each process.
- Request: An \( n \times m \) matrix indicates the current request of each process. If Request \( \left[ i \right] = k \), 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 \( i \) = 0, then Finish\( [i] \) = false; otherwise, Finish\( [i] \) = true
2. Find an index \( i \) such that both:
   - (a) Finish\( [i] \) = false
   - (b) Request \( \left[ i \right] \) = Work
   - If no such \( i \) exists, go to step 4
Detection Algorithm (Cont.)

3. Work = Work + Allocation
   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 a 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 T0:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0 0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P1 2 0 0</td>
<td>2 0 2</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P2 3 0 3</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>P3 2 1 1</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>P4 0 0 2</td>
<td>0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

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

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur? Detection frequency should be a function of deadlock frequency.
  - 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
  - Very hard to infer "some safe state" from arbitrary program.
  - Safest bet is "safe state" corresponding to just restarting the process.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor