Chapter 3

Deadlocks

3.1. Resource
3.2. Introduction to deadlocks
3.3. The ostrich algorithm
3.4. Deadlock detection and recovery
3.5. Deadlock avoidance
3.6. Deadlock prevention
3.7. Other issues
Introduction

• Parallel operation among many devices driven by concurrent processes contribute significantly to high performance. But concurrency also results in contention for resources and possibility of deadlock among the vying processes.

• *Deadlock* is a situation where a group of processes are permanently blocked waiting for the resources held by each other in the group.

• Typical application where deadlock is a serious problem: Operating system, data base accesses, and distributed processing.
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 \( \{P_0, P_1, \ldots, P_0\} \) 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 \( P_2 \), \ldots, \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_0 \) is waiting for a resource that is held by \( P_0 \).
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 $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
Resource-Allocation Graph (Cont.)

- Process

- Resource Type with 4 instances

- $P_i$ requests instance of $R_j$

- $P_i$ is holding an instance of $R_j$
Resource Allocation Graph with a Deadlock
Resource Allocation Graph with a cycle but no deadlock.
Deadlock Modeling

A                         B                        C
Request R  
Request S  
Request T  
Release R  
Release S  
Release T  

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

How deadlock occurs
Methods for Handling Deadlocks

• Ensure that the system will *never* enter a deadlock state. (pessimistic)
• Allow the system to enter a deadlock state and then recover. Database systems;
• Ignore the problem and pretend that deadlocks never occur in the system; Older operating systems; (ostrich algorithm: optimistic)
Dealing with Deadlock

Strategies for dealing with Deadlocks

1. just ignore the problem altogether
2. detection and recovery
3. dynamic avoidance
   - careful resource allocation
4. prevention
   - negating one of the four necessary conditions
The Ostrich Algorithm

• Pretend there is no problem
• Reasonable if
  – deadlocks occur very rarely
  – cost of prevention is high
• UNIX and Windows takes this approach
• It is a trade off between
  – convenience
  – correctness
Detection with One Resource of Each Type (1)

- Note the resource ownership and requests.
- A cycle can be found within the graph, denoting deadlock.
Recovery from Deadlock (1)

• Recovery through preemption
  – take a resource from some other process
  – depends on nature of the resource
• Recovery through rollback
  – checkpoint a process periodically
  – use this saved state
  – restart the process if it is found deadlocked
Recovery from Deadlock (2)

- Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun from the beginning
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 \(<P_1, P_2, \ldots, P_n>\) 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.
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 \textit{a priori} in the system.
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:** $n \times m$ 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 $P_i$ is currently allocated $k$ instances of $R_j$.
- **Need:** $n \times m$ matrix. If $Need[i,j] = k$, then $P_i$ may need $k$ more instances of $R_j$ to complete its task.

\[
Need[i,j] = Max[i,j] - Allocation[i,j].
\]
Safety Algorithm

1. Let \( Work \) and \( Finish \) be vectors of length \( m \) and \( n \), respectively. Initialize:
   
   \[
   Work = Available \\
   Finish[i] = false \text{ for } i = 1, 3, ..., n.
   \]

2. Find and \( i \) such that both:
   
   (a) \( Finish[i] = false \)
   
   (b) \( Need_i \leq Work \)

   If no such \( i \) exists, go to step 4.

3. \( Work = Work + Allocation_i \)

   \( Finish[i] = true \)

   go to step 2.

4. If \( Finish[i] == true \) for all \( i \), then the system is in a safe state.
Resource-Request Algorithm for Process $P_i$

$Request = \text{request vector for process } P_i$. 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:
   
   $\text{Available} = \text{Available} = \text{Request}_i$;
   $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$;
   $\text{Need}_i = \text{Need}_i - \text{Request}_i$;

   • 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 $P_0$ through $P_4$; 3 resource types $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances).

• Snapshot at time $T_0$:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example (Cont.)

• The content of the matrix. Need is defined to be $\text{Max} - \text{Allocation}$.

\begin{center}
\begin{tabular}{c c c c}
\textit{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 \\
\end{tabular}
\end{center}

• 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) (Cont.)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow true$.

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

- 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 Prevention

Restrain the ways request can be made.

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

- **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 (Cont.)

• **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 Prevention
Attacking the Mutual Exclusion Condition

• Some devices (such as printer) can be spooled
  – only the printer daemon uses printer resource
  – thus deadlock for printer eliminated

• Not all devices can be spooled

• Principle:
  – avoid assigning resource when not absolutely necessary
  – as few processes as possible actually claim the resource
Attacking the Hold and Wait Condition

• Require processes to request resources before starting
  – a process never has to wait for what it needs

• Problems
  – may not know required resources at start of run
  – also ties up resources other processes could be using

• Variation:
  – process must give up all resources
  – then request all immediately needed
Attacking the No Preemption Condition

• This is not a viable option
• Consider a process given the printer
  – halfway through its job
  – now forcibly take away printer
  – !!??
Attacking the Circular Wait Condition (1)

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

- Normally ordered resources
- A resource graph
Attacking the Circular Wait Condition (1)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual exclusion</td>
<td>Spool everything</td>
</tr>
<tr>
<td>Hold and wait</td>
<td>Request all resources initially</td>
</tr>
<tr>
<td>No preemption</td>
<td>Take resources away</td>
</tr>
<tr>
<td>Circular wait</td>
<td>Order resources numerically</td>
</tr>
</tbody>
</table>

Summary of approaches to deadlock prevention
Other Issues
Two-Phase Locking

• Phase One
  – process tries to lock all records it needs, one at a time
  – if needed record found locked, start over
  – (no real work done in phase one)

• If phase one succeeds, it starts second phase,
  – performing updates
  – releasing locks

• Note similarity to requesting all resources at once

• Algorithm works where programmer can arrange
Nonresource Deadlocks

• Possible for two processes to deadlock
  – each is waiting for the other to do some task

• Can happen with semaphores
  – each process required to do a down() on two semaphores (mutex and another)
  – if done in wrong order, deadlock results
Starvation

• Algorithm to allocate a resource
  – may be to give to shortest job first

• Works great for multiple short jobs in a system

• May cause long job to be postponed indefinitely
  – even though not blocked

• Solution:
  - (No additional solution provided)