Chapter 7: Deadlocks
Deadlock

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

- This chapter we look in detail at what leads to deadlock, and how we can attempt to detect and avoid it.
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

\[
\begin{align*}
P_0 & \quad P_1 \\
\text{wait (A)}; & \quad \text{wait(B)} \\
\text{wait (B)}; & \quad \text{wait(A)}
\end{align*}
\]

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

- We can reason about the properties of a system with respect to deadlock if we formalize it.

- The formalism describes:
  - What resources are held by the processes
  - What resources are processes waiting for

- We formalize it by building a model based on directed graphs and their properties.
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 \( P_0 \).
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
Observation: We have a cycle now!

Does cycle = deadlock?
This graph has a cycle but no deadlock.
Basic Facts

- If graph contains no cycles \(\Rightarrow\) no deadlock

- If graph contains a cycle \(\Rightarrow\)
  - 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
  - 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
  - **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 resources after a higher-ordered one
Requires that the system has some additional \textit{a priori} information available

- Simplest and most useful model requires that each process declare the \textit{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 \textit{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 \(<P_1, P_2, \ldots, P_n>\) 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 \(P_j\) 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.
Basic Facts

- 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.
Safe, Unsafe, Deadlock 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
Resource-Allocation Graph Scheme

- **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.

- 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

- Process $P_1$
- Process $P_2$
- Resource $R_1$
- Resource $R_2$
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.

- **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:

   \[ Work = Available \]

   \[ Finish[i] = false \text{ for } i = 0, 1, \ldots, n-1 \]

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
Looking closer

- Remember safe state means a sequence exists \(<P_0, P_1, \ldots, P_N>\) such that \(P_J\) can execute if all \(P_0\ldots P(J-1)\) complete and release their resources for \(J\) in range \(0\ldots N\).

- 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 \(\rightarrow\) Safe!
    - We reach a set of processes that simply cannot proceed \(\rightarrow\) Unsafe!
Resource-Request Algorithm for Process \( P_i \)

\( \text{Request} = \text{request vector for process } P_i. \) If \( \text{Request}_i[j] = k \) then process \( P_i \) wants \( k \) instances of resource type \( R_j \)

1. If \( \text{Request}_i \leq \text{Need}_i \) go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If \( \text{Request}_i \leq \text{Available} \), 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:

\[
\begin{align*}
\text{Available} &= \text{Available} - \text{Request}; \\
\text{Allocation}_i &= \text{Allocation}_i + \text{Request}_i; \\
\text{Need}_i &= \text{Need}_i - \text{Request}_i;
\end{align*}
\]

- **If safe** ⇒ the resources are allocated to \( P_i \)
- **If unsafe** ⇒ \( P_i \) 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>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A$ $B$ $C$</td>
<td>$A$ $B$ $C$</td>
<td>$A$ $B$ $C$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td></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</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

- 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)

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

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