Deadlock

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
FAQ

• Does a cycle in a resource allocation graph signify “circular wait”? Only if there is only one instance of a resource.
• Does the MAC spinning wheel indicate a deadlock?
• Are application hangs caused by deadlocks? OS timer perhaps 5 sec.
• How much damage can deadlocked apps cause?
Basic Facts

• If graph contains no cycles $\implies$ no deadlock

• If graph contains a cycle $\implies$
  – 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:
  – Deadlock prevention
    • ensuring that at least one of the 4 conditions cannot hold
  – Deadlock avoidance
    • Dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

• Allow the system to enter a deadlock state
  – Detect and then recover. Hope is that it happens rarely.

• Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
Methods for Handling Deadlocks

<table>
<thead>
<tr>
<th>Approach</th>
<th>Resource allocation policy</th>
<th>Scheme</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prevention</td>
<td>Conservative, undercommits resources</td>
<td>Requesting all resources at once</td>
<td>Good for processes with a single burst of activity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preemption</td>
<td>Good when preemption cost is small</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Resource ordering</td>
<td>Compile time enforcement possible</td>
</tr>
<tr>
<td>Avoidance</td>
<td>midway</td>
<td>Find at least one safe path (dynamic)</td>
<td>Future max requirement must be known</td>
</tr>
<tr>
<td>Detection</td>
<td>Liberal</td>
<td>Invoked periodically</td>
<td>Preemption may be needed</td>
</tr>
</tbody>
</table>
Ostrich algorithm: Stick your head in the sand; pretend there is no problem at all.

Advantages:
- Cheaper, rarely needed anyway
- Prevention, avoidance, detection and recovery
  - Need to run constantly

Disadvantages:
- Resources held by processes that cannot run
- More and more processes enter deadlocked state
  - When they request more resources
- Deterioration in system performance
  - Requires restart

To be fair to the ostriches, let me say that ...
Deadlock Prevention: Limit Mutual Exclusion

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

Restrain the ways request can be made:

- **Limit Mutual Exclusion** –
  - not required for sharable resources (e.g., read-only files)
  - (Mutual Exclusion must hold for non-sharable resources)
• **Limit Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  
  1. Require process to request and be allocated all its resources before it begins execution
Deadlock Prevention: Limit Hold and Wait

- **Limit Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  2. Allow a process to request resources when it is holding none.
  Ex: Copy data from DVD, sort file, and print
  – First request DVD and disk file
  – Then request file and printer
- **Disadvantage:** starvation possible
• **Limit 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
Deadlock Prevention: Limit Circular Wait

• **Limit Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

• Assign each resource a unique number
  – Disk drive: 1
  – Printer: 2 …
  – Request resources in increasing order
    • *Example soon*
Dining philosophers problem: Necessary conditions for deadlock

- Mutual exclusion
  - 2 philosophers cannot share the same chopstick
- Hold-and-wait
  - A philosopher picks up one chopstick at a time
  - Will not let go of the first while it waits for the second one
- No preemption
  - A philosopher does not snatch chopsticks held by some other philosopher
- Circular wait
  - Could happen if each philosopher picks chopstick with the same hand first
Deadlock Example

Assume that thread one is the first to acquire the locks and does so in the order (1) first mutex, (2) second mutex. If thread two later acquires the locks out of order, witness generates a warning message on the system console.

Solution: Lock-order verifier 
“Witness” records the relationship that first mutex must be acquired before second mutex. If thread two later acquires the locks out of order, witness generates a warning message on the system console.

```c
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```
Deadlock may happen even with Lock Ordering

```c
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
        acquire(lock2);
            withdraw(from, amount);
            deposit(to, amount);
        release(lock2);
    release(lock1);
}
```

Ex: Transactions 1 and 2 execute concurrently.
    Transaction 1 transfers $25 from account A to account B, and
    Transaction 2 transfers $50 from account B to account A.
Deadlock is possible, even with lock ordering.

Lock ordering:
First `from` lock, then `to` lock
Deadlock Avoidance

Manage resource allocation to ensure the system never enters an unsafe state.
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
Deadlock Avoidance

• Require additional information about how resources are to be requested
• Knowledge about sequence of requests and releases for processes
  – Allows us to decide if resource allocation could cause a future deadlock
    • Process P: Tape drive, then printer
    • Process Q: Printer, then tape drive
Deadlock Avoidance: Handling resource requests

• For each resource request:
  – Decide whether or not process should wait
    • To avoid possible future deadlock

• Predicated on:
  1. Currently available resources
  2. Currently allocated resources
  3. Future requests and releases of each process
Avoidance: amount and type of information needed

• **Resource allocation state**
  - Number of available and allocated resources
  - Maximum demands of processes

• *Dynamically* examine resource allocation state
  - Ensure circular-wait cannot exist

• Simplest model:
  - Declare maximum number of resources for each type
  - Use information to avoid deadlock
Safe Sequence

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 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 and released resources
    - When $P_i$ terminates, $P_{i+1}$ can obtain its needed resources, and so on

- If no such sequence exists: system state is **unsafe**
Deadlock avoidance: Safe states

• If the system can:
  – Allocate resources to each process in some order
    • Up to the maximum for the process
  – Still avoid deadlock
  – Then it is in a safe state

• A system is safe ONLY IF there is a safe sequence

• A safe state is not a deadlocked state
  – Deadlocked state is an unsafe state
  – Not all unsafe states are deadlock
Safe, Unsafe, Deadlock State

Examples of safe and unsafe states in next 3 slides
Example A: Assume 12 Units in the system

<table>
<thead>
<tr>
<th></th>
<th>Max need</th>
<th>Current holding</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

At T0:
9 units allocated
3 (12-9) units available

A unit could be a drive, a block of memory etc.

• At time T0 is the system is in a safe state?
  – Try sequence <P1, P0 , P2>
  – P1 can be given 2 units
  – When P1 releases its resources; there are 5 units
  – P0 uses 5 and subsequently releases them (# 10 now)
  – P2 can then proceed.

• Thus <P1, P0 , P2> is a safe sequence, and at T0 system was in a safe state
Example A: Assume 12 Units in the system (timing)

Is the state at T0 safe?

<table>
<thead>
<tr>
<th>Max need</th>
<th>Current holding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T0</td>
</tr>
<tr>
<td>av</td>
<td>3</td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
</tr>
</tbody>
</table>

Thus the state at T0 is safe.
Example B: 12 Units available in the system

<table>
<thead>
<tr>
<th>Max need</th>
<th>T0</th>
<th>T1 safe?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Av</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3 Is that OK?</td>
</tr>
</tbody>
</table>

Before T1:
3 units available

At T1:
2 units available

- At time **T1**, P2 is allocated 1 more units. Is that a good decision?
  - Now only P1 can proceed.
  - When P1 releases its resources; there are 4 units
  - P0 needs 5 more, P2 needs 6 more. Deadlock.
    - **Mistake** in granting P2 the additional unit.
- The state at **T1** is not a safe state.
Avoidance Algorithms

- **Single instance** of a resource type
  - Use a **resource-allocation graph scheme**

- **Multiple instances** of a resource type
  - Use the **banker’s algorithm (Dijkstra)**
• **Claim edge** $P_i \rightarrow R_j$ 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
Suppose $P_2$ requests $R_2$. Although $R_2$ is currently free, we cannot allocate it to $P_2$, since this action will create a cycle getting system in an unsafe state. If $P_1$ requests $R_2$, and $P_2$ requests $R_1$, then a deadlock will occur.
• 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: examining a request

• Multiple instances of resources.
• Each process must a priori claim maximum use
• When a process requests a resource
  – it may have to wait (resource request algorithm)
  – Request not granted if the resulting system state is unsafe (safety algorithm)
• When a process gets all its resources it must return them in a finite amount of time
• Modeled after a banker in a small town making loans
Let $n = \text{number of processes, and } m = \text{number of resources types}$.

- **Available**: Vector of length $m$. If available $[j] = k$, there are $k$ instances of resource type $R_j$ available.

**Processes vs resources:**

- **Max**: $n \times m$ matrix. If $Max[\cdot, \cdot] = k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$.

- **Allocation**: $n \times m$ matrix. If $Allocation[\cdot, \cdot] = k$ then $P_i$ is currently allocated $k$ instances of $R_j$.

- **Need**: $n \times m$ matrix. If $Need[\cdot, \cdot] = k$, 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: Is System in safe state?

1. Let \textbf{Work} and \textbf{Finish} be vectors of length \( m \) and \( n \), respectively. Initialize:
   \[ \text{Work} = \text{Initially Available resources} \]
   \[ \text{Finish}[i] = \text{false} \quad \text{for} \quad i = 0, 1, \ldots, n-1 \]

2. Find a process \( i \) 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

\( n = \text{number of processes}, \)
\( m = \text{number of resources types} \)
\( \text{Need}_i: \text{additional res needed} \)
\( \text{Work}: \text{res currently free} \)
\( \text{Finish}_i: \text{processes finished} \)
\( \text{Allocation}_i: \text{allocated to i} \)
Resource-Request Algorithm for Process $P_i$

Notation: $Request_i = request$ vector for process $P_i$. If $Request_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$

Algorithm: Should the allocation request be granted?

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. Is allocation safe?: Pretend to allocate requested resources to $P_i$ by modifying the state as follows:
   
   $Available = Available - Request_i$
   $Allocation_i = Allocation_i + Request_i$
   $Need_i = Need_i - Request_i$
   
   - If safe $\Rightarrow$ the resources are allocated to $P_i$
   - If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is preserved.

Safety Algorithm
Example A: 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>Process</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>

- Is it a safe state?
Example (Cont.)

- The matrix \textit{Need} is \textbf{Max – Allocation}

\begin{align*}
\text{Need} \\
\begin{array}{ccc}
A & B & C \\
\hline \\
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{array}
\end{align*}

- Next we show that the system is in a safe state since the sequence \(< P_1, P_3, P_4, P_2, P_0 >\) satisfies safety criteria.
The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria, since:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</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>$P_0$</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>1 2 2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</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>

P1 run to completion. Available becomes $[3 3 2] + [2 0 0] = [5 3 2]$

P3 run to completion. Available becomes $[5 3 2] + [2 1 1] = [7 4 3]$

P4 run to completion. Available becomes $[7 4 3] + [0 0 2] = [7 4 5]$

P2 run to completion. Available becomes $[7 4 5] + [3 0 2] = [10 4 7]$

P0 run to completion. Available becomes $[10 4 7] + [0 1 0] = [10 5 7]$

**Hence state above is safe**
Ex B: Assume now $P_1$ Requests (1,0,2)

• Check that Request $\leq$ Available
  
  - $(1,0,2) \leq (3,3,2) \Rightarrow \text{true. Check for safety after pretend allocation.}$  
  
  $P_1$ allocation would be $(2 \ 0 \ 0) + (1 \ 0 \ 2) = 302$

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</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>$P_0$</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 2</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. Yes, safe state.
• Given State is *(previous slide)*

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

• $P_4$ request for (3,3,0): cannot be granted - resources are not available.

• $P_0$ request for (0,2,0): cannot be granted since the resulting state is unsafe.
Deadlock Detection

• Allow system to enter deadlock state

• Detection algorithm
  – Single instance of each resource:
    • wait-for graph
  – Multiple instances:
    • detection algorithm (based on Banker’s algorithm)

• Recovery scheme