CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 1018 L14
Deadlocks

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

- Mutex vs Semaphore
• 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?
A contemporary example
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 \( \{P_0, P_1, ..., P_n\} \) 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 \( P_2 \), ..., \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_0 \).
A set of vertices $V$ and a set of edges $E$.

- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system

- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$
• 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

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Does a deadlock exist here?

P3 will eventually be done with R3, letting P2 use it.

Thus P2 will be eventually done, releasing R1. ...

P1 holds an instance of R2, and is requesting R1.
At this point, two minimal cycles exist in the system:

\[ P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1 \]

\[ P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2 \]

Processes \( P1 \), \( P2 \), and \( P3 \) are deadlocked.
Graph With A Cycle But No Deadlock

P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle. Thus there is no deadlock.

If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.
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 \textit{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.
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.

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.
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.
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 \(<P_1, P_2, ..., P_n>\) 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></td>
</tr>
<tr>
<td></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>2</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)
Resource-Allocation Graph Scheme

- **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 $Max [i,j] = k$, 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: Is System in safe state?

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

2. Find a process \(i\) such that both:
   \begin{enumerate}
   \item (a) \text{Finish} [i] = \text{false}
   \item (b) \text{Need}_i \leq \text{Work}
   \end{enumerate}
   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

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

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

Algorithm: Should the allocation request be granted?

1. If $\textit{Request}_i \leq \textit{Need}_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\textit{Request}_i \leq \textit{Available}$, 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:
   \begin{align*}
   \textit{Available} &= \textit{Available} - \textit{Request}_i; \\
   \textit{Allocation}_i &= \textit{Allocation}_i + \textit{Request}_i; \\
   \textit{Need}_i &= \textit{Need}_i - \textit{Request}_i;
   \end{align*}
   \begin{itemize}
   \item If safe $\Rightarrow$ the resources are allocated to $P_i$
   \item If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is preserved.
   \end{itemize}
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)
- Is it a safe state?

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>available</td>
<td></td>
<td>3 3 2</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5 3</td>
<td>0 1 0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>4 3</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2 2</td>
<td>2 0 0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0 2</td>
<td>3 0 2</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2 2</td>
<td>2 1 1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3 3</td>
<td>0 0 2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3 1</td>
<td></td>
</tr>
</tbody>
</table>
Example A: Banker’s Algorithm

• Is it a safe state?
• Yes, since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>available</td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</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. $(1,0,2) \leq (3,3,2) \rightarrow$ true.
- Check for safety after pretend allocation. $P_1$ allocation would be $(2 \ 0 \ 0) + (1 \ 0 \ 2) = 302$

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>available</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
Hence state above is safe, thus the allocation would be safe.
Ex C,D: Additional Requests ..

- Given State is (same as previous slide)

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>available</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

P4 request for (3,3,0): cannot be granted - resources are not available.
P0 request for (0,2,0): cannot be granted since the resulting state is unsafe.
Bankers Algorithm: Practical Issues

• Processes rarely know in advance about their maximum resource needs
• Number of processes is not fixed
  – Varies dynamically
• Resources thought to be available can disappear
• Few systems use this algorithm
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
• Maintain **wait-for graph** (based on resource allocation graph)
  – Nodes are processes
  – $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$

  – **Deadlock if cycles**

• 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

- \( P_1 \)
- \( P_2 \)
- \( P_3 \)
- \( P_4 \)
- \( P_5 \)
- \( R_1 \)
- \( R_2 \)
- \( R_3 \)
- \( R_4 \)
- \( R_5 \)
- \( R_6 \)

Corresponding wait-for graph

- \( P_1 \)
- \( P_2 \)
- \( P_3 \)
- \( P_4 \)
- \( P_5 \)

Has cycles. Deadlock.
Several Instances of a Resource Type

Banker’s algorithm: Can requests by all process be satisfied?

• Available: A vector of length $m$ indicates the number of available (currently free) 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 $\text{Request} [i][j] = k$, then process $P_i$ is requesting $k$ more instances of resource type $R_j$. 