Deadlock
Deadlocks

• Computer resources are full of resources that can only be used by one process at a time
• Unpredictable results can occur if two or more processes use the same resource at the same time
• OS control this problem by arbitrating resource requests
• OS grant exclusive access and manage access to system resources
• By allowing processes to have exclusive access to resources problems may occur:
  – Starvation
  – Deadlock
Deadlocks

• Deadlock can occur if two or more processes are competing for two or more common resources

• Deadlock causes all processes involved in the deadlock to become permanently blocked
  – Each process is waiting on resources owned by other processes that are also deadlocked

• Deadlock may be difficult to detect
  – Often timing sensitive

• What to do about deadlocks:
  – Understand them
  – Model them
  – Develop ways to detect them
  – Develop ways to avoid them
Resources

• Deadlocks occur when processes have been granted exclusive access to resources
  – Resources include semaphores, devices, files, ...

• Resources come in two types:
  – Preemptive resource: A resource that can be taken away from a process without any ill effects.
    • Example: memory
  – Non-preemptive resource: A resource that cannot be taken safely away from a process
    • Example: printer

• Deadlocks generally involve non-preemptive resources

• Deadlocks can be easily fixed with preemptive resources
  – Take resources away from one or more deadlocked processes
Using Resources

- Access to resources should follow the model presented when we discussed critical sections
  - Request the resource
  - Use the resource
  - Release the resource
- If a resource is not available when requested, the requesting process should block
- It is often difficult to develop a program that can continue if needed resources are not available
- This model relies on well-behaved processes
  - What happens if a process does not release resources
    - Timeout?
Conditions for Deadlock

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

- Conditions for deadlock:
  - Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available
  - Hold and wait condition. Processes currently holding resources granted earlier can request new resources
  - No preemption condition. Resources previously granted cannot be forcibly taken away from a process
  - Circular wait condition. There must be a circular chain of two or more processes where each is waiting for a resource held by the next member of the chain
Deadlock Modeling

- Deadlocks can be modeled by creating simple graphs.
  - Circles are processes
  - Squares are resources
  - Edges represent a resource request or a resource being held by a process

\[ \text{Process A holds resource B} \quad \text{Process C requests resource D} \quad \text{Deadlock} \]
Deadlock Modeling

- Draw all resources and processes
- Add edges as resource requests and releases are requested
- Check for a cycle in the graph to detect deadlock

Process A: Request R, S; Release R, S
Process B: Request S, T; Release S, T
Process C: Request T, R; Release T, R

A requests R  B requests S  C requests T
A requests S  B requests T
C requests R  DEADLOCK
How To Deal With Deadlocks

- One way to prevent deadlocks is to prevent resource request from being granted that may lead to potential deadlock
- Resource graphs are useful for seeing if a set of resource requests may lead to deadlock
  - Graphs are based on a particular ordering
  - Not all orderings of the same requests may lead to deadlock
- Need ways to deal with deadlock
  - Just ignore the problem altogether
  - Detect a deadlock and recover from the deadlock, once detected
  - Avoid deadlocks by carefully allocate resources
  - Prevent deadlocks by structurally negating one of the four necessary deadlock conditions
The Ostrich Algorithm

• Do nothing to prevent deadlocks

• Good when deadlocks are rare

• Do not want to pay penalties for implementing expensive deadlock avoidance or protection mechanisms if deadlocks are rare

• The ostrich algorithm may be appropriate for non safety-critical systems

• Should still make an effort to communicate the possibility of deadlocks to customers and users
Deadlock Detection and Recovery

• System does not attempt to prevent deadlocks
• System instead is capable of detecting deadlocks, and if found, can recover from the deadlock
• Have to answer the questions:
  – Is the system deadlocked?
  – If so, what processes are involved in the deadlock
• Detection can be handled by keeping an up-to-date resource allocation graph and check for cycles in this graph
  – Many graph cycle detection algorithms exist
• The cycle detection approach works well when all resources are of the same type
  – Does not scale to handle multiple resource types
Deadlock Detection: Single Resource Type

What processes are deadlocked?
Deadlock Detection: Multiple Resource Types

- Need 2 vectors, and 2 matrices
  - $E =$ Resource request vector
  - $A =$ Resource available vector
  - $C =$ Current allocation matrix
  - $R =$ Request matrix

- Invariant: Every resource must either be allocated or available

- Deadlock detection algorithm works by iteratively updating the matrices and vectors to detect if a deadlock occurs

- Algorithm starts by initializing all processes to be unmarked

- The algorithm proceeds by marking processes that are able to obtain all of their resources

- Any unmarked processes remaining when the algorithm terminates are deadlocked
Deadlock Detection

Algorithm: Multiple Resources

• Algorithm
  – Look for an unmarked processes, Pi for which the i-th row of R is less then A
  – If such a process is found, add the i-th row of C to A, mark the process and go back to the previous step
  – If no such process, exists, the algorithm terminates
Deadlock Detection
Algorithm: Multiple Resources

Example:

E=(4 3 2 1)  A=(2 1 0 0)

\[
C = \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0 \\
\end{bmatrix}
\]

R = \[
\begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 \\
\end{bmatrix}
\]


E=(4 3 2 1)  A=(2 2 2 0)

\[
C = \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

R = \[
\begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

Deadlock Detection
Algorithm: Multiple Resources

Example:

\[ E=(4 \ 3 \ 2 \ 1) \quad A=(4 \ 2 \ 2 \ 1) \]
\[
C = \begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\quad R = \begin{bmatrix}
2 & 0 & 1 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

Process 1 can run, \( R[1] \leq A \), Mark Process 1

\[ E=(4 \ 3 \ 2 \ 1) \quad A=(2 \ 2 \ 2 \ 0) \]
\[
C = \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\quad R = \begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

All processes have been marked, all resource requests have been granted, no DEADLOCK!
Deadlock Detection
Algorithm: Multiple Resources

Example:

\[ E = (4 \ 3 \ 2 \ 1) \quad A = (2 \ 1 \ 0 \ 0) \]

\[
C = \begin{pmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{pmatrix}
\]

\[
R = \begin{pmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 \\
2 & 1 & 0 & 0
\end{pmatrix}
\]

Process 3 can run, \( R[3] \leq A \), Mark Process 3

\[
E = (4 \ 3 \ 2 \ 1) \quad A = (2 \ 2 \ 2 \ 0)
\]

\[
C = \begin{pmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

\[
R = \begin{pmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

Neither process 1 or process 2 can run, both need a resource that is not available, DEADLOCK
Recovering From Deadlock

- **Recovery through preemption**
  - In some cases it might be possible to take a resource away from its current owner
  - May break deadlock by selecting one or more resources held by a subset of processes participating in the deadlock
  - Since most resources are NOT preemptable, it may be difficult or impossible to implement this solution

- **Recovery through rollback**
  - Have processes take periodic checkpoints
  - Checkpointing involves saving a processes state (perhaps in a file)
  - If a deadlock is detected, the OS can rollback the state of one or more of the deadlock processes
  - Commonly used in database technology
Recovering from Deadlock

• Another technique for recovering from a deadlock is to carefully select and terminate one or more processes that are participating in the deadlock

• Although a process has been terminated, it allows the other processes to continue

• May choose to automatically rerun the terminated process

• Not all processes can safely be rerun
  – A process may update a database
  – Rerunning this process may result in updates being applied multiple times
  – Must ensure that rerunning the process does not have any unwanted side effects
  – With database processes we can restore the state of the database by using the transaction log
Deadlock Avoidance

- Goal is to avoid deadlocks by carefully investigating if honoring a resource request is safe
- Need to ensure that the result of honoring a resource request leaves the system in a safe state
- A safe state is when the system is not deadlocked and there is a way to satisfy all pending resource requests in some order
- May implement a deadlock avoidance policy for a single resource type by keeping track of
  - Resources currently held by each process
  - Maximum number of resources needed by each process
  - Number of free resources remaining
- Bankers Algorithm
### Bankers Algorithm Example: Deadlock Avoidance

<table>
<thead>
<tr>
<th></th>
<th>HAS</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**Free:** 3

<table>
<thead>
<tr>
<th></th>
<th>HAS</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**Run B, Free:** 1

<table>
<thead>
<tr>
<th></th>
<th>HAS</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**Free:** 5

<table>
<thead>
<tr>
<th></th>
<th>HAS</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

**Run C, Free:** 0

<table>
<thead>
<tr>
<th></th>
<th>HAS</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

**Free:** 7

**Run A, Free:** 1

<table>
<thead>
<tr>
<th></th>
<th>HAS</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

**NO DEADLOCK**

**Free:** 9
Bankers Algorithm: Multiple Resource Types

• The basic bankers algorithm can be extended to handle multiple resource types

• Need:
  – A resource assigned matrix
  – A resource still needed matrix
  – An existing resource vector (E)
  – A possessed resource vector (P)
  – An available resource vector (A)
  – Invariant: $P + A = E$

• Algorithm
  – Look at the resources still needed matrix and see if you can find a row $\leq A$
  – If one exists, mark the process and add its held resources to $A$
  – Terminate if no process has a row $\leq A$ (deadlock), or if all processes have been marked (no deadlock)
Bankers Algorithm Example

**Initial State**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Resources Assigned

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>P</td>
<td>(5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>(1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Current state is safe!
Run Order D, E, A, C, B
Bankers Algorithm Example

Process B wants resource 3

<table>
<thead>
<tr>
<th></th>
<th>Resources Assigned</th>
<th>Resources Still Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3 0 1 1</td>
<td>A 1 1 0 0</td>
</tr>
<tr>
<td>B</td>
<td>0 1 1 0</td>
<td>B 0 1 0 2</td>
</tr>
<tr>
<td>C</td>
<td>1 1 1 0</td>
<td>C 3 1 0 0</td>
</tr>
<tr>
<td>D</td>
<td>1 1 0 1</td>
<td>D 0 0 1 0</td>
</tr>
<tr>
<td>E</td>
<td>0 0 0 0</td>
<td>E 2 1 1 0</td>
</tr>
</tbody>
</table>

E = (6 3 4 2)
P = (5 3 2 2)
A = (1 0 1 0)

This is a safe request!
Run Order D, E, A, C, B
### Bankers Algorithm Example

**Process E wants resource 3**

<table>
<thead>
<tr>
<th></th>
<th>Assigned</th>
<th>Still Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3 0 1 1</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>B</td>
<td>0 1 1 0</td>
<td>0 1 0 2</td>
</tr>
<tr>
<td>C</td>
<td>1 1 1 0</td>
<td>3 1 0 0</td>
</tr>
<tr>
<td>D</td>
<td>1 1 0 1</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>E</td>
<td>0 0 1 0</td>
<td>2 1 0 0</td>
</tr>
</tbody>
</table>

E = (6 3 4 2)  
P = (5 3 2 2)  
A= (1 0 0 0)

This is not a safe request!  
No run order exists because  
A = (1 0 0 0)  
Because this is an unsafe state the Bankers algorithm defers this request!
Deadlock Prevention

- Recall that deadlock prevention involves ensuring that at least one of the four deadlock conditions can not occur

- Performed by creating a resource allocation policy to prevent one of these conditions from occurring

- Recall that the four conditions necessary to have deadlock are:
  - Mutual exclusion condition
  - Hold and wait condition
  - No preemption condition
  - Circular wait condition
Deadlock Prevention

• Attacking the mutual exclusion condition
  – Prevent a resource from being mutually assigned to a single process
  – May use a spooling technique
  – Unfortunately, not all devices can be spooled

• Attacking the hold and wait condition
  – Prevent processes that are holding resources from having to wait for additional resources
  – Accomplished by requesting all resources in advance
  – Either get all resources, or get no resources
  – Problems:
    • Processes may not know about all of the resources that they need in advance
    • Does not lead to optimal allocation of resources
Deadlock Prevention

• Attacking the no preemption condition
  – Very difficult
  – Not all resources may be safely taken away from a process once the process obtains exclusive access to the resource

• Attacking the circular wait condition
  – Solved by assigning all resources a unique numeric identifier
  – Require that all processes request resources in some order
    • Increasing, decreasing order
  – By all processes agreeing to request resources in some order it will be impossible to get a cycle in the resource allocation graph
    • We saw this in the midterm
  – Problem: Technique prevents deadlock, but not starvation
Starvation

- Starvation is a problem closely related to deadlocks
- In a dynamic system, resource requests are happening all of the time
- A policy is needed to select which process gets which resource
- It may be possible that this policy prevents non-deadlocked processes from getting resources for an extended period of time
  - Example: Cycle prevention technique by providing a global ordering to resource requests
- Need the resource allocation policy to ensure that no process needs to wait too long for resources
  - Perhaps use timestamps, or first-come, first served technique