CS 333
Introduction to Operating Systems

Class 7 - Deadlock

Jonathan Walpole
Computer Science
Portland State University
Continued from Class 6

- Implementing Hoare semantics
- Reentrancy
- Message Passing
“Hoare Semantics”

What happens when a Signal is performed?
   The signaling thread (A) is suspended.
   The signaled thread (B) wakes up and runs immediately.
      B can assume the condition is now true/satisfied

From the original Hoare Paper:

“No other thread can intervene [and enter the monitor] between
the signal and the continuation of exactly one waiting thread.”

“If more than one thread is waiting on a condition, we postulate
that the signal operation will reactivate the longest waiting
thread. This gives a simple neutral queuing discipline which
ensures that every waiting thread will eventually get its turn.”
Implementing Hoare Semantics

- Thread A holds the monitor lock
- Thread A **signals** a condition that thread B was waiting on
- Thread B is moved back to the ready queue?
  - B should run immediately
  - Thread A must be suspended...
  - The monitor lock must be passed from A to B!
- When B finishes it releases the monitor lock
- Thread A must re-acquire the lock
  - Perhaps A is blocked, waiting to re-acquire the lock
Implementing Hoare Semantics

- Problem:
  - Possession of the monitor lock must be passed directly from A to B and then eventually back to A
Implementing Hoare Semantics

- Implementation Ideas:
  - Hand off mutex directly from A to B in signal
  - Consider signaling thread A to be “urgent” after it hands off the monitor mutex to B
    - Thread C trying to gain initial entry to the monitor is not “urgent”
    - Thread A should get preference when trying to reacquire the mutex
  - Consider two wait lists associated with each `MonitorLock` (so now this is not exactly a mutex)
    - `UrgentlyWaitingThreads`
    - `NonurgentlyWaitingThreads`
  - Want to wake up urgent threads first, if any
Implementing Hoare Semantics

- Recommendation for Project 4 implementation:
  - Do not modify the methods provided, because future code will use them
  - Create new classes:
    - MonitorLock -- similar to Mutex
    - HoareCondition -- similar to Condition
Reentrancy
Reentrant code

- A function/method is said to be reentrant if...
  
  *A function that has been invoked may be invoked again before the first invocation has returned, and will still work correctly*

- Recursive routines are reentrant

- In the context of concurrent programming...
  
  *A reentrant function can be executed simultaneously by more than one thread, with no ill effects*
Consider this function...

```plaintext
var count: int = 0
function GetUnique () returns int
    count = count + 1
    return count
endFunction
```

- Is it reentrant?
- What happens if it is executed by different threads concurrently?
When is code reentrant?

- **Some variables are**
  - “local” -- to the function/method/routine
  - “global” -- sometimes called “static”

- **Access to local variables?**
  - A new stack frame is created for each invocation
  - Each thread has its own stack

- **What about access to global variables?**
  - **Must use synchronization!**
Does this work?

```plaintext
var count: int = 0

myLock: Mutex

function GetUnique () returns int
    myLock.Lock()
    count = count + 1
    myLock.Unlock()
    return count
endFunction
```
Making this function reentrant

```plaintext
var count: int = 0
myLock: Mutex

function GetUnique () returns int
    var i: int
    myLock.Lock()
    count = count + 1
    i = count
    myLock.Unlock()
    return i
endFunction
```
Message Passing
Message Passing

- Synchronization requires Interprocess Communication
  - via shared memory
  - across machine boundaries
  - message passing can be used for both

- Processes synchronize with send and receive primitives
  - receive can block (like waiting on a Semaphore)
  - send unblocks a process blocked on receive (just as a signal unblocks a waiting process)
The basic idea:
  - The producer sends the data to the consumer in a message
  - The system buffers messages
    - The producer can out-run the consumer
    - The messages will be kept in order
  - But how does the producer avoid overflowing the buffer?
    - After consuming the data, the consumer sends back an “empty” message
  - A fixed number of messages (N=100)
  - The messages circulate back and forth.
Producer-consumer with message passing

const N = 100           -- Size of message buffer
var em: char
for i = 1 to N          -- Get things started by
    Send (producer, &em) -- sending N empty messages
endFor

thread consumer
    var c, em: char
    while true
        Receive(producer, &c)  -- Wait for a char
        Send(producer, &em)    -- Send empty message back
        // Consume char...
    endwhile
end
Producer-consumer with message passing

thread producer
  var c, em: char
  while true
    // Produce char c...
    Receive(consumer, &em) -- Wait for an empty msg
    Send(consumer, &c) -- Send c to consumer
  endwhile
end
OS design choices for message passing

- **Option 1: Mailboxes**
  - System maintains a buffer of sent, but not yet received, messages
  - Must specify the size of the mailbox ahead of time
  - Sender will be blocked if the buffer is full
  - Receiver will be blocked if the buffer is empty
OS design choices for message passing

- **Option 2: No buffering**
  - If Send happens first, the sending thread blocks
  - If Receive happens first, the receiving thread blocks
  - Sender and receiver must **Rendezvous** (i.e. meet)
  - Both threads are ready for the transfer
  - The data is copied / transmitted
  - Both threads are then allowed to proceed
DEADLOCK
Resources and deadlocks

- Processes need access to resources in order to make progress

- Examples of computer resources
  - printers
  - disk drives
  - kernel data structures (process & file table entries ...)
  - locks/semaphores to protect critical sections

- Suppose a process holds resource A and requests resource B
  - at the same time another process holds B and requests A
  - both are blocked and remain so ... this is deadlock
Deadlock modeling: resource usage model

- **Sequence of events required to use a resource**
  - request the resource (like acquiring a mutex lock)
  - use the resource
  - release the resource (like releasing a mutex lock)

- **Must wait if request is denied**
  - block
  - busy wait
  - fail with error code
Preemptable vs nonpreemptable resources

- **Preemptable resources**
  - can be taken away from a process with no ill effects

- **Nonpreemptable resources**
  - will cause the holding process to fail if taken away

- **Deadlocks occur when processes are granted exclusive access to non-preemptable resources and wait when the resource is not available**
Definition of 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

- Usually the event is the release of a currently held resource
- None of the processes can ...
  - be awakened
  - run
  - release resources
Deadlock conditions

- A deadlock situation can occur if and only if the following conditions hold simultaneously
  - Mutual exclusion condition - resource assigned to one process
  - Hold and wait condition - processes can get more than one resource
  - No preemption condition
  - Circular wait condition - chain of two or more processes (must be waiting for resource from next one in chain)
Examples of deadlock
Resource acquisition scenarios

**Thread A:**

acquire (resource_1)
use resource_1
release (resource_1)

**Example:**

```go
var r1_mutex: Mutex
...

r1_mutex.Lock()
Use resource_1
r1_mutex.Unlock()
```
Resource acquisition scenarios

**Thread A:**

- acquire (resource_1)
- use resource_1
- release (resource_1)

**Another Example:**

```php
var r1_sem: Semaphore
r1_sem.Signal()
...
r1_sem.Wait()
Use resource_1
r1_sem.Signal()
```
## Resource acquisition scenarios

<table>
<thead>
<tr>
<th><strong>Thread A:</strong></th>
<th><strong>Thread B:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire (resource_1)</td>
<td>acquire (resource_2)</td>
</tr>
<tr>
<td>use resource_1</td>
<td>use resource_2</td>
</tr>
<tr>
<td>release (resource_1)</td>
<td>release (resource_2)</td>
</tr>
</tbody>
</table>
Resource acquisition scenarios

**Thread A:**
- acquire (resource_1)
- use resource_1
- release (resource_1)

**Thread B:**
- acquire (resource_2)
- use resource_2
- release (resource_2)

*No deadlock can occur here!*
## Resource acquisition scenarios: 2 resources

**Thread A:**
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

**Thread B:**
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)
Resource acquisition scenarios: 2 resources

**Thread A:**
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

**Thread B:**
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

No deadlock can occur here!
## Resource acquisition scenarios: 2 resources

<table>
<thead>
<tr>
<th><strong>Thread A:</strong></th>
<th><strong>Thread B:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire (resource_1)</td>
<td>acquire (resource_2)</td>
</tr>
<tr>
<td>use resources 1</td>
<td>use resources 2</td>
</tr>
<tr>
<td>release (resource_1)</td>
<td>release (resource_2)</td>
</tr>
<tr>
<td>acquire (resource_2)</td>
<td>acquire (resource_1)</td>
</tr>
<tr>
<td>use resource 2</td>
<td>use resource 1</td>
</tr>
<tr>
<td>release (resource_2)</td>
<td>release (resource_1)</td>
</tr>
</tbody>
</table>
Resource acquisition scenarios: 2 resources

**Thread A:**
- acquire (resource_1)
- use resources 1
- release (resource_1)
- acquire (resource_2)
- use resource 2
- release (resource_2)

**Thread B:**
- acquire (resource_2)
- use resources 2
- release (resource_2)
- acquire (resource_1)
- use resource 1
- release (resource_1)

No deadlock can occur here!
## Resource acquisition scenarios: 2 resources

<table>
<thead>
<tr>
<th><strong>Thread A:</strong></th>
<th><strong>Thread B:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire (resource_1)</td>
<td>acquire (resource_2)</td>
</tr>
<tr>
<td>acquire (resource_2)</td>
<td>acquire (resource_1)</td>
</tr>
<tr>
<td>use resources 1 &amp; 2</td>
<td>use resources 1 &amp; 2</td>
</tr>
<tr>
<td>release (resource_2)</td>
<td>release (resource_1)</td>
</tr>
<tr>
<td>release (resource_1)</td>
<td>release (resource_2)</td>
</tr>
</tbody>
</table>
Resource acquisition scenarios: 2 resources

<table>
<thead>
<tr>
<th>Thread A:</th>
<th>Thread B:</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire (resource_1)</td>
<td>acquire (resource_2)</td>
</tr>
<tr>
<td>acquire (resource_2)</td>
<td>acquire (resource_1)</td>
</tr>
<tr>
<td>use resources 1 &amp; 2</td>
<td>use resources 1 &amp; 2</td>
</tr>
<tr>
<td>release (resource_2)</td>
<td>release (resource_1)</td>
</tr>
<tr>
<td>release (resource_1)</td>
<td>release (resource_2)</td>
</tr>
</tbody>
</table>

Deadlock is possible!
Consequences of deadlock

- **Deadlock occurs in a single program**
  - Programmer creates a situation that deadlocks
  - Kill the program and move on
  - Not a big deal

- **Deadlock occurs in the Operating System**
  - Spin locks and locking mechanisms are mismanaged within the OS
  - Threads become frozen
  - System hangs or crashes
  - Must restart the system and kill all applications
Dealing with deadlock

- **Four general strategies**
  - Ignore the problem
    - Hmm... advantages, disadvantages?
  - Detection and recovery
  - Dynamic avoidance through resource allocation
  - Prevention, by structurally negating one of the four conditions
Deadlock detection

- Let the problem happen, then recover
- How do you know it happened?
- Do a depth-first-search on the resource allocation graph
Detection: Resource Allocation Graphs

Process/Thread

Resource

\( A \)

\( R \)
Detection: Resource Allocation Graphs

Process/Thread

Resource

A

"is held by"

R
Detection: Resource Allocation Graphs

Process/Thread → A → S → "is requesting" → Resource

Resource
Detection: Resource Allocation Graphs
Detection: Resource Allocation Graphs

Deadlock
Detection: Resource Allocation Graphs

Deadlock = a cycle in the graph
Deadlock detection (1 resource of each)

- Do a depth-first-search on the resource allocation graph
Deadlock detection (1 resource of each)

- Do a depth-first-search on the resource allocation graph
Deadlock detection (1 resource of each)

- Do a depth-first-search on the resource allocation graph

![Diagram of resource allocation graph]

(a) (b)
Deadlock detection (1 resource of each)

- Do a depth-first-search on the resource allocation graph
Deadlock detection (1 resource of each)

- Do a depth-first-search on the resource allocation graph

![Diagram of resource allocation graph with a deadlock cycle highlighted]

(a) (b)
Multiple units of a resource

- Some resources have only one “unit”.
  - Only one thread at a time may hold the resource.
    - Printer
    - Lock on ReadyQueue

- Some resources have several units.
  - All units are considered equal; any one will do.
    - Page Frames
    - Dice in the Gaming Parlor problem
  - A thread requests “k” units of the resource.
  - Several requests may be satisfied simultaneously.
Deadlock modeling with multiple resources

- **Theorem**: *If a graph does not contain a cycle then no processes are deadlocked*
- A cycle in a RAG is a **necessary** condition for deadlock
- Is it a **sufficient** condition?
Deadlock modeling with multiple resources

- **Theorem:** *If a graph does not contain a cycle then no processes are deadlocked*
  - A cycle in a RAG is a necessary condition for deadlock
  - Is it a sufficient condition?
Deadlock detection issues

- How often should the algorithm run?
  - On every resource request?
  - Periodically?
  - When CPU utilization is low?
  - When we suspect deadlock because some thread has been asleep for a long period of time?
Recovery from deadlock

- If we detect deadlock, what should be done to recover?
  - Abort deadlocked processes and reclaim resources
  - Abort one process at a time until deadlock cycle is eliminated

- Where to start?
  - Lowest priority process?
  - Shortest running process?
  - Process with fewest resources held?
  - Batch processes before interactive processes?
  - Minimize number of processes to be terminated?
Other deadlock recovery techniques

- Recovery through preemption and rollback
  - Save state periodically
    - take a checkpoint
    - start computation again from checkpoint
      - Checkpoint must be prior to resource acquisition!
  - Useful for long-lived computation systems
Deadlock avoidance

Detection vs. avoidance...

- Detection - “optimistic” approach
  - Allocate resources
  - “Break” system to fix the problem

- Avoidance - “pessimistic” approach
  - Don’t allocate resource if it may lead to deadlock
  - If a process requests a resource…
    … make it wait until you are sure it’s OK

- Which one to use depends upon the application
  - How easy is it to recover from deadlock?
Avoidance using process-resource trajectories
Avoidance using process-resource trajectories

Process A

t_1 t_2 t_3 t_4

- Requests Printer
- Requests CD-RW
- Releases Printer
- Releases CD-RW

time
Avoidance using process-resource trajectories
Avoidance using process-resource trajectories

Requests CD-RW

Requests Printer

Releases CD-RW

Releases Printer
Avoidance using process-resource trajectories

Process B

\( t_Z \)
\( t_Y \)
\( t_X \)
\( t_W \)

Process A

\( t_1 \) \( t_2 \) \( t_3 \) \( t_4 \)

Time
Avoidance using process-resource trajectories

Both processes hold CD-RW
Avoidance using process-resource trajectories

Both processes hold Printer
Avoidance using process-resource trajectories

Process B

\[ t_W, t_X, t_Y, t_Z \]

Process A

\[ t_1, t_2, t_3, t_4 \]

Forbidden Zone
Avoidance using process-resource trajectories

Trajectory showing system progress
Avoidance using process-resource trajectories

Process A

Process B

t_W
t_X
t_Y
t_Z

t_1 t_2 t_3 t_4

B makes progress, A is not running
Avoidance using process-resource trajectories

Process B

\( t_1, t_2, t_3, t_4 \)

Process A

\( t_1, t_2, t_3, t_4 \)

B requests the CD-RW
Avoidance using process-resource trajectories

Process B

Process A

time

t1 t2 t3 t4

Request is granted
Avoidance using process-resource trajectories

Process A

Process B

A runs & makes a request for printer
Avoidance using process-resource trajectories

Request is granted; A proceeds
Avoidance using process-resource trajectories

Process B

Process A

B runs & requests the printer... MUST WAIT!
Avoidance using process-resource trajectories

A runs & requests the CD-RW
Avoidance using process-resource trajectories

A...
holds printer
requests CD-RW

B...
holds CD-RW
requests printer
Avoidance using process-resource trajectories

A...
holds printer
requests CD-RW

B...
holds CD-RW
requests printer

DEADLOCK!
Avoidance using process-resource trajectories

Process B
\[ t_W, t_X, t_Y, t_Z \]

Process A
\[ t_1, t_2, t_3, t_4 \]

A danger occurred here.

Should the OS give A the printer, or make it wait???
Avoidance using process-resource trajectories

This area is “unsafe”
Avoidance using process-resource trajectories

Within the “unsafe” area, deadlock is inevitable. We don’t want to enter this area. The OS should make A wait at this point!
Avoidance using process-resource trajectories

B requests the printer, B releases CD-RW, B releases printer, then A runs to completion!
Safe states

- **The current state:**
  "which processes hold which resources"

- **A “safe” state:**
  - No deadlock, and
  - There is some scheduling order in which every process can run to completion even if all of them request their maximum number of units immediately

- **The Banker’s Algorithm:**
  - **Goal:** Avoid unsafe states!!!
  - *When a process requests more units, should the system grant the request or make it wait?*
Avoidance with multiple resources

<table>
<thead>
<tr>
<th>Total resource vector</th>
<th>Available resource vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources in existence (E_1, E_2, E_3, ..., E_m)</td>
<td>Resources available (A_1, A_2, A_3, ..., A_m)</td>
</tr>
<tr>
<td>Current allocation matrix</td>
<td>Maximum Request Vector</td>
</tr>
</tbody>
</table>
| \[
\begin{bmatrix}
C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\
C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm}
\end{bmatrix}
| \[
\begin{bmatrix}
R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\
R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm}
\end{bmatrix}
|

Row n is current allocation to process n

Row 2 is what process 2 might need

Note: These are the max. possible requests, which we assume are known ahead of time!
Banker’s algorithm for multiple resources

- Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion.

- Assume the process of the row chosen requests all the resources that it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to A vector.

- Repeat steps 1 and 2, until either all process are marked terminated, in which case the initial state was safe, or until deadlock occurs, in which case it was not.
# Avoidance with multiple resources

### Total resource vector

Resources in existence  
\((E_1, E_2, E_3, \ldots, E_m)\)

Current allocation matrix

\[
\begin{bmatrix}
C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\
C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm}
\end{bmatrix}
\]

Row \(n\) is current allocation to process \(n\)

### Available resource vector

Resources available  
\((A_1, A_2, A_3, \ldots, A_m)\)

Maximum Request Vector

\[
\begin{bmatrix}
R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\
R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm}
\end{bmatrix}
\]

Row 2 is what process 2 might need

---

**Run algorithm on every resource request!**
Avoidance with multiple resources

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(4 2 3 1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(2 1 0 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current allocation matrix

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

Max request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 \\
\end{bmatrix}
\]
**Avoidance with multiple resources**

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>E = ( 4, 2, 3, 1 )</td>
<td>A = ( 2, 1, 0, 0 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Current allocation matrix**

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

**Max request matrix**

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$
Avoidance with multiple resources

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

\[
A = (2 \quad 1 \quad 0 \quad 0)
\]

Current allocation matrix

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

Max request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]
Avoidance with multiple resources

\[ E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix} \]

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

Current allocation matrix

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

Max request matrix

\[ R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \]
Avoidance with multiple resources

E = (4 2 3 1)

A = (2 1 0 0)

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

\[
\begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]
## Avoidance with multiple resources

### Max request matrix

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### Current allocation matrix

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

### Max request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 \\
\end{bmatrix}
\]
Problems with deadlock avoidance

- **Deadlock avoidance is often impossible**
  - because you don’t know in advance what resources a process will need!

- **Alternative approach “deadlock prevention”**
  - Make deadlock *impossible!*
  - Attack one of the four conditions that are necessary for deadlock to be possible
Deadlock prevention

- **Conditions necessary for deadlock:**
  - Mutual exclusion condition
  - Hold and wait condition
  - No preemption condition
  - Circular wait condition
Deadlock prevention

- **Attacking mutual exclusion?**
  - a bad idea for some resource types
    - resource could be corrupted
  - works for some kinds of resources in certain situations
    - eg., when a resource can be partitioned

- **Attacking no preemption?**
  - a bad idea for some resource types
    - resource may be left in an inconsistent state
  - may work in some situations
    - checkpointing and rollback of idempotent operations
Deadlock prevention

- **Attacking hold and wait?**
  - Require processes to request all resources before they begin!
  - Process must know ahead of time
  - Process must tell system its “max potential needs”
    - eg., like in the bankers algorithm
    - When problems occur a process must release all its resources and start again
Attacking the conditions

- Attacking circular waiting?
  - Number each of the resources
  - Require each process to acquire lower numbered resources before higher numbered resources
  - More precisely: "A process is not allowed to request a resource whose number is lower than the highest numbered resource it currently holds"
Recall this example of deadlock

Thread A:
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

Thread B:
- acquire (resource_2)
- acquire (resource_1)
- use resources 1 & 2
- release (resource_1)
- release (resource_2)

Assume that resources are ordered:
1. Resource_1
2. Resource_2
3. ...etc...
Recall this example of deadlock

**Thread A:**
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

**Thread B:**
- acquire (resource_2)
- acquire (resource_1)
- use resources 1 & 2
- release (resource_1)
- release (resource_2)

- Assume that resources are ordered:
  1. Resource_1
  2. Resource_2
  3. ...etc...
- Thread B violates the ordering!
Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- Process A
  - holds X
  - requests Y

- Process B
  - holds Y
  - requests Z

- Process C
  - holds Z
  - requests X
Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- **Process A**
  - holds X
  - requests Y

- **Process B**
  - holds Y
  - requests Z

- **Process C**
  - holds Z
  - requests X
Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- **Process A**
  - holds X
  - requests Y

- **Process B**
  - holds Y
  - requests Z

- **Process C**
  - holds Z
  - requests X
Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- **Process A**
  - holds X
  - requests Y
  - $X < Y$

- **Process B**
  - holds Y
  - requests Z
  - $Y < Z$

- **Process C**
  - holds Z
  - requests X
  - $Z < X$
Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- **Process A**
  - holds X
  - requests Y

- **Process B**
  - holds Y
  - requests Z

- **Process C**
  - holds Z
  - requests X

This is impossible!
Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- **Process A**
  - holds X
  - requests Y

- **Process B**
  - holds Y
  - requests Z

- **Process C**
  - holds Z
  - requests X

\[ X < Y \]
\[ Y < Z \]
\[ Z < X \]

This is impossible! Therefore the assumption must be false!
Resource Ordering

- The chief problem:
  - It may be hard to come up with an acceptable ordering of resources!

- Still, this is the most useful approach in an OS
  1. ProcessControlBlock
  2. FileControlBlock
  3. Page Frames

- Also, the problem of resources with multiple units is not addressed.
A word on starvation

- Starvation and deadlock are two different things
  - With deadlock - no work is being accomplished for the processes that are deadlocked, because processes are waiting for each other. Once present, it will not go away.

  - With starvation - work (progress) is getting done, however, a particular set of processes may not be getting any work done because they cannot obtain the resource they need
Quiz

- What is deadlock?
- What conditions must hold for deadlock to be possible?
- What are the main approaches for dealing with deadlock?
- Why does resource ordering help?