A common project 2 error

<table>
<thead>
<tr>
<th>method Lock ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
</tr>
<tr>
<td>oldIntStat: int</td>
</tr>
<tr>
<td>oldIntStat = SetInterruptsTo (DISABLED)</td>
</tr>
<tr>
<td>if mutex == 0</td>
</tr>
<tr>
<td>mutex = 1</td>
</tr>
<tr>
<td>heldBy = currentThread</td>
</tr>
<tr>
<td>oldIntStat = SetInterruptsTo (oldIntStat)</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>sleepThreads.AddToEnd(currentThread)</td>
</tr>
<tr>
<td>currentThread.Sleep()</td>
</tr>
<tr>
<td>oldIntStat = SetInterruptsTo (oldIntStat)</td>
</tr>
<tr>
<td>endIf</td>
</tr>
<tr>
<td>endMethod</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>method Unlock ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
</tr>
<tr>
<td>oldIntStat: int</td>
</tr>
<tr>
<td>t: ptr to Thread</td>
</tr>
<tr>
<td>oldIntStat = SetInterruptsTo (DISABLED)</td>
</tr>
<tr>
<td>mutex = 0</td>
</tr>
<tr>
<td>if sleepThreads.IsEmpty() == false</td>
</tr>
<tr>
<td>t = sleepThreads.Remove()</td>
</tr>
<tr>
<td>t.status = READY</td>
</tr>
<tr>
<td>readyList.AddToEnd(t)</td>
</tr>
<tr>
<td>endIf</td>
</tr>
<tr>
<td>oldIntStat = SetInterruptsTo (oldIntStat)</td>
</tr>
<tr>
<td>endMethod</td>
</tr>
</tbody>
</table>
Quiz

- What is the difference between a monitor and a semaphore?
  - Why might you prefer one over the other?
- How do the wait/signal methods of a condition variable differ from the wait/signal methods of a semaphore?
- What is the difference between Hoare and Mesa semantics for condition variables?
  - What implications does this difference have for code surrounding a wait() call?
Message Passing

- Interprocess Communication
  - via shared memory
  - across machine boundaries

- Message passing can be used for synchronization or general communication

- Processes use 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)
Producer-consumer with message passing

- **The basic idea:**
  - The producer sends the data to 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
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
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** (ie. meet)
  - Both threads are ready for the transfer
  - The data is copied / transmitted
  - Both threads are then allowed to proceed
CS 333
Introduction to Operating Systems
Class 7 - Deadlock

Jonathan Walpole
Computer Science
Portland State University
Resources and deadlocks

- Processes need access to resources in order to make progress

- Examples of computer resources
  - printers
  - tape 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
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 non-preemptable resources

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

- Non-preemptable resources
  - will cause the 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)
Resource acquisition scenarios

**Thread A:**

<table>
<thead>
<tr>
<th>acquire (resource_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>use resource_1</td>
</tr>
<tr>
<td>release (resource_1)</td>
</tr>
</tbody>
</table>

**Example:**

```go
var r1_mutex: Mutex
...
rl_mutex.Lock()
Use resource_1
rl_mutex.Unlock()
```
Resource acquisition scenarios

**Thread A:**

```plaintext
acquire (resource_1)
use resource_1
release (resource_1)
```

**Another Example:**

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

<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>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

**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)
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>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>
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_2)
- acquire (resource_1)
- use resources 1 & 2
- release (resource_1)
- release (resource_2)

*Deadlock is possible!*
Examples 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
Other examples of deadlock
Resource Allocation Graphs

Process/Thread

Resource
Resource Allocation Graphs

Process/Thread

Resource

"is held by"
Resource Allocation Graphs

Process/Thread

Resource

"is requesting"
Resource Allocation Graphs
Resource Allocation Graphs

\[
\begin{array}{c}
A \\
\downarrow \\
R \\
\downarrow \\
B \\
\uparrow \\
S \\
\uparrow \\
A
\end{array}
\]

Deadlock
Resource Allocation Graphs

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

- Let the problem happen, then recover
- How do you know it happened?
- Do a depth-first-search on the resource allocation graph
Deadlock detection (1 resource of each)

- Let the problem happen, then recover
- How do you know it happened?
- Do a depth-first-search on the resource allocation graph
Deadlock detection (1 resource of each)

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

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

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

![Depth-First-Search Diagram]

(a) Diagram showing resource allocation and potential deadlock.

(b) Another diagram illustrating the resolution process.
Deadlock detection (1 resource of each)

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

- **What should be done to recover?**
  - Abort deadlocked processes and reclaim resources
  - Temporarily reclaim resource, if possible
  - 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?
  - Number of processes that must be terminated?
Other deadlock recovery techniques

- Recovery through rollback (preemption)
  - Save state periodically
    - take a checkpoint
    - start computation again from checkpoint
  - Done for large 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
Process-resource trajectories

$\text{Process A}$

$t_1$ $t_2$ $t_3$ $t_4$

time
Process-resource trajectories

Requests Printer

Requests CD-RW

Releases Printer

Releases CD-RW

Process A

t_1  t_2  t_3  t_4

time
Process-resource trajectories

Process B

\( t_W \)
\( t_X \)
\( t_Y \)
\( t_Z \)
Process-resource trajectories

requests CD-RW
requests Printer
releases CD-RW
releases Printer
Process-resource trajectories

Process A

Process B

\( t_W \)

\( t_X \)

\( t_Y \)

\( t_Z \)

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

time

time
Process-resource trajectories

Both processes hold CD-RW
Process-resource trajectories

Both processes hold Printer
Process-resource trajectories

Forbidden Zone
Process-resource trajectories

Trajectory showing system progress
Process-resource trajectories

B makes progress, A is not running
Process-resource trajectories

Process A

Process B

$B$ requests the CD-RW
Process-resource trajectories

Request is granted
Process-resource trajectories

A runs & makes a request for printer
Process-resource trajectories

Request is granted; A proceeds
Process-resource trajectories

Process B

$\tau_W$

$\tau_X$

$\tau_Y$

$\tau_Z$

Process A

$t_1$

$t_2$

$t_3$

$t_4$

B runs & requests the printer... MUST WAIT!
Process-resource trajectories

A runs & requests the CD-RW
Process-resource trajectories

A...
holds printer
requests CD-RW

B...
holds CD-RW
requests printer
Process-resource trajectories

Process A

Process B

A...
holds printer
requests CD-RW

B...
holds CD-RW
requests printer

DEADLOCK!
Process-resource trajectories

A danger occurred here.

Should the OS give A the printer, or make it wait???
Process-resource trajectories

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

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

Current allocation matrix

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

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

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

Request matrix

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

Row 2 is what process 2 needs

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 modeling

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 algorithm

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

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 algorithm

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

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

Current allocation matrix

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

Max request matrix

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

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

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

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

\[ E = (4 \ 2 \ 3 \ 1) \]
\[ A = (2 \ 1 \ 0 \ 0) \]
# Avoidance algorithm

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

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>
</tbody>
</table>

Current allocation matrix:

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

Max request matrix:

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

<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></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 = (2 1 0 0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2 2 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 2 2 1</td>
<td></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

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

- Conditions necessary for deadlock:
  - Mutual exclusion condition
  - Hold and wait condition
  - No preemption condition
  - Circular wait condition
Attacking the conditions

- **Attacking mutual exclusion?**
  - a bad idea for some resource types
    - Because resource could be corrupted
  - may work for some resources in certain situations

- **Attacking no preemption?**
  - a bad idea for some resource types
    - Because resource may be left in an inconsistent state
  - may work in some situations
Attacking the conditions

- 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”
Attacking the conditions

- If a process decides it wants more than its initial declared needs, it must...
  - Release all resources
  - Give the system a new “max potential needs”
  - Resume execution

- **Issues:**
  - Under-allocation of resources
  - Resource needs not known in advance
Attacking the conditions

- **Attacking circular wait?**
  - 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

<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>

Assume that resources are ordered:

1. Resource_1
2. Resource_2
3. …etc…
Recall this example of deadlock

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

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

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

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?