Administrivia

- Assignments 0 & 1
- Class contact for the next two weeks
- Next week
  - midterm exam
  - project review & discussion (Chris Chambers)
- Following week
  - Memory Management (Wuchi Feng)
CSE 513
Introduction to Operating Systems

Class 4 - IPC & Synchronization (2)
Deadlock

Jonathan Walpole
Dept. of Comp. Sci. and Eng.
Oregon Health and Science University
Counting semaphores

- A binary semaphore can only take on the values of \([0, 1]\).

- Class exercise: create a counting semaphore (generalized semaphore that we discussed previously) using just a binary semaphore!!
Possible solution

Semaphore S1, S2, S3; // BINARY!!
int C = N;   // N is # locks

down_c(sem)
{
    downB(S3);
    downB(S1);
    C = C - 1;
    if (C<0) {
        upB(S1);
        downB(S2);
    }
    else {
        upB(S1);
    }
    upB(S3);
}

up_c(sem)
{
    downB(S1);
    C = C + 1;
    if (C<=0) {
        upB(S2);
    }
    upB(S1);
}
Monitors

- It is difficult to produce correct programs using semaphores
  - correct ordering of up and down is tricky!
  - avoiding deadlock is tricky!
  - boundary conditions are tricky!

- Can we get the compiler to generate the correct semaphore code for us?
  - what are suitable higher level abstractions for synchronization?
Monitors

- Collect related shared objects together in a monitor

- Encapsulation and mutual exclusion
  - Local data variables are accessible only via the monitor’s procedures
  - Processes enter the monitor by invoking one of its procedures
  - Only one process may execute within the monitor at a given time

- Condition variables (cv)
  - Wait(cv) - process blocked (queued) until condition holds
  - Signal(cv) - signals the condition and unblocks (dequeues) a process
Monitor structures

- Monitor operations
- Initialization code
- Shared data
- Condition queues
- Monitor entry queue
Monitor example for mutual exclusion

process Producer
begin
  loop
    <produce char "c">
    BoundedBuffer.deposit(c)
  end loop
end Producer

process Consumer
begin
  loop
    BoundedBuffer.remove(c)
    <consume char "c">
  end loop
end Consumer

monitor: BoundedBuffer
var buffer : ...;
  nextIn, nextOut : ...;
entry deposit(c: char)
begin
  ...
end
entry remove(var c: char)
begin
  ...
end
end BoundedBuffer

OGI SCHOOL OF SCIENCE & ENGINEERING
OREGON HEALTH & SCIENCE UNIVERSITY
Monitor example with condition variables

monitor : BoundedBuffer
var buffer : array[0..n-1] of char
nextIn, nextOut : 0..n-1 := 0
fullCount : 0..n := 0
notEmpty, notFull : condition

entry deposit(c:char)
begin
  if (fullCount = n) then
    wait(notFull)
  end if
  buffer[nextIn] := c
  nextIn := nextIn+1 mod n
  fullCount := fullCount+1
  signal(notEmpty)
end deposit

end BoundedBuffer

entry remove(var c: char)
begin
  if (fullCount = n) then
    wait(notEmpty)
  end if
  c := buffer[nextOut]
  nextOut := nextOut+1 mod n
  fullCount := fullCount-1
  signal(notFull)
end remove
Monitor design choices

- **Condition variables introduce a problem for mutual exclusion**
  - only one process active in the monitor at a time, so what to do when a process is unblocked on signal?
  - must not block holding the mutex, so what to do when a process blocks on wait?

- **Should signals be stored/remembered?**
  - signals are not stored
  - if signal occurs before wait, signal is lost!

- **Should condition variables count?**
Monitor design choices

- **Choices when A signals a condition that unblocks B**
  - A waits for B to exit the monitor or blocks again
  - B waits for A to exit the monitor or block
  - Signal causes A to immediately exit the monitor or block (on what condition?)

- **Choices when A signals a condition that unblocks B & C**
  - B is unblocked, but C remains blocked
  - C is unblocked, but B remains blocked

- **Choices when A calls wait and blocks**
  - a new external process is allowed to enter
  - but which one?
Common monitor semantics

- **Hoare semantics**
  - On signal, allow signaled process to run; upon its exit from the monitor, signaler process continues

- **Brinch Hansen semantics**
  - Signaluer must immediately exit following signal
Message Passing

- Interprocess communication
  - via shared memory
  - across machine boundaries

- Message passing can be used locally or remotely for synchronization or general communication
  - processes use send and receive primitives
  - receive can block like wait
  - send unblocks a process blocked on receive (like signal unblocking a waiting process)
Producer consumer with message passing

#define N 100 /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m; /* message buffer */

    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        send(consumer, &m); /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&m); /* extract item from message */
        send(producer, &m); /* send back empty reply */
        consume_item(item); /* do something with the item */
    }
}
Design Choices for Message Passing

- **Mailboxes**
  - system maintains a buffer of sent, but not yet received, messages

- **Rendezvous**
  - sender and receiver must be active at the same time
  - receive must be blocked before send occurs
  - kernel does no buffering
  - when does the send return?
Barriers

- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through
Deadlock
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 Nonpreemptable Resources

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

- Nonpreemptable 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

down (resource_1);
  use resource_1;
up (resource_1);

down (resource_2);
  use resource_2;
up (resource_2);

down (resource_1);
down (resource_2);
  use both resources;
up (resource_2);
up (resource_1);

down (resource_1);
down (resource_2);
  use both resources;
up (resource_2);
up (resource_1);
Resource acquisition scenarios

down (resource_1);
  use resource;
up (resource_1);
down (resource_2);
  use resource;
up (resource_2);

down (resource_2);
  use resource;
up (resource_2);
down (resource_1);
  use resource;
up (resource_1);

down (resource_1);
down (resource_2);
  use resources;
up (resource_2);
up (resource_1);

down (resource_2);
down (resource_1);
  use resources;
up (resource_1);
up (resource_2);
Flavors of Deadlock

- **Not so bad**
  - Programmer creates a situation that deadlocks
  - Kill the program and move on

- **Worse**
  - Spin locks and locking mechanisms within the OS
Other examples of deadlock
Deadlock modeling

- **Resource Allocation Graphs (RAGs)**

  - Resource R assigned to process A
  - Process B waiting for resource S
  - Process C and D are deadlocked over T & U
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
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 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 the existence of a cycle 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 the existence of a cycle a sufficient condition?
Deadlock detection (multiple resources)

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 \(n\)

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

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

Row 2 is what process 2 needs
Deadlock detection (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
\((A_1, A_2, A_3, \ldots, A_m)\)

Request matrix

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

Row 2 is what process 2 needs
Deadlock detection (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 \(n\)

**What I have (now!)**

**Available resource vector**

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

Request matrix

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

Row 2 is what process 2 needs

**What I am requesting now**
Detection algorithm

- Is there a sequence of running process such that all the resources will be returned?

\[
\begin{align*}
E &= (4 \ 2 \ 3 \ 1) \\
A &= (2 \ 1 \ 0 \ 0)
\end{align*}
\]

\[
\begin{align*}
C &= \begin{bmatrix}
0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 \\
0 & 1 & 2 & 0
\end{bmatrix} \\
R &= \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\end{align*}
\]
Detection algorithm

1. Look for an unmarked process $P_i$, for which the $i$-th row of $R$ is less than or equal to $A$

2. If such a process is found, add the $i$-th row of $C$ to $A$, mark the process and go back to step 1

3. If no such process exists the algorithm terminates

If all marked, no deadlock
Detection 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>
</tbody>
</table>

Current allocation matrix

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

Request matrix

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

\[
E = \begin{pmatrix}
4 & 2 & 3 & 1
\end{pmatrix}
\quad
A = \begin{pmatrix}
2 & 1 & 0 & 0
\end{pmatrix}
\]

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

Request matrix
\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]
Detection algorithm

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

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

Current allocation matrix

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

Request matrix

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

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

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

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

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

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

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

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

Request matrix
\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]
Detection algorithm

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

\[ A = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 \\ 4 & 2 & 2 & 1 \end{pmatrix} \]

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

Request matrix
\[ R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix} \]
Detection algorithm

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

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

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

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

No deadlock!
Detection algorithms

- How often should the algorithm run?
  - After every resource request?
  - Periodically?
  - When CPU utilization is low?
  - When we suspect deadlock?
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?**
  - Low priority process
  - How long process has been executing
  - How many resources a process holds
  - Batch or interactive
  - Number of processes that must be terminated
Other deadlock recovery techniques

- Recovery through rollback
  - Save state periodically
  - Start computation again from “checkpoint”
  - Done for large computation systems
Deadlock avoidance

- Detection vs. avoidance...
  - Detection - “optimistic” approach
    - Allocate resources
    - “Break” system to fix it
  - Avoidance - “pessimistic” approach
    - Don’t allocate resource if it may lead to deadlock

- Which one to use depends upon the application
Resource allocation plot
Resource allocation graph

- Printer
- Plotter

B

l_8
l_7
l_6
l_5

p q l_1 l_2 l_3 l_4

Printer

Plotter

u (Both processes finished)
**Safe states**

- Safe state – “when system is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resource immediately”

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

- 10 total

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>7</td>
</tr>
</tbody>
</table>

- Free: 3

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

- Free: 1

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>7</td>
</tr>
</tbody>
</table>

- Free: 5

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
</tbody>
</table>

- Free: 0

<table>
<thead>
<tr>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

- Free: 7
## Unsafe states

<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

10 total

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</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: 2

Unsafe

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</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>

Free: 0

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Free: 4
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 processes 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

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

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

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 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>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 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

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

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

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

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

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

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

Current allocation matrix:

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

Max request matrix:

R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}
## 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>2 2 2 0</td>
<td>4 2 2 1</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 usually impossible
  - because you don’t know in advance what resources a process will need!

- Alternative approach “deadlock prevention”
  - Prevent the situation in which deadlock might occur for all time!
  - Attack one of the four conditions that are necessary for deadlock to be possible
Attacking the conditions

- **Attacking mutual exclusion?**
  - a bad idea for some resource types
  - may work for others

- **Attacking no preemption?**
  - a bad idea for some resource types
  - may work for others
Attacking the conditions

- Attacking hold and wait
  - have processes request all resources before beginning
    - Underallocation of resources
    - Unknown requests
  - if new request, deallocate and reallocate!
Attacking the conditions

- **Attacking circular wait**
  - same problem/solution in the dining philosophers
  - number all resources and acquire in the same order
  - may be hard to get an ordering that everyone likes

1  2  3  4  5  6  7
Attacking the conditions

- **Attacking circular wait**
  - Same problem in the dining philosophers
  - Number all resources
  - Typically hard to get an ordering that everyone likes
Attacking the conditions

- Attacking circular wait
  - Same problem in the dining philosophers
  - Number all resources
  - Typically hard to get an ordering that everyone likes

1  2  3  4  5  6  7
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
  - 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 are trying to get
Summary

- What is deadlock?
- Deadlock detection algorithms

Read
- Chapter 3
- Sample problems
  - Chap 3: 15, 20, 21
Spare slides

Solution to sleeping barber problem.