Announcements

- Homework #1
  - Show your work, but be concise; if you have more than 1 page for an answer, you are probably writing too much.
  - Question 1: Use the “speedup” definition that was provided in lecture ... the ratio of the processing time for $n$ instructions for a non-pipelined versus pipelined architecture.
  - Question 4: possible “diagram” (shown right) to help you explain your answer.
From last time ...

Threading lab

Q: Why did you experience inconsistent results (outputs) during subsequent invocations of your threeThreads.cpp program?
From last time ...

Threading lab

Q: Why did you experience inconsistent results (outputs) during subsequent invocations of your threeThreads.cpp program?

Task: Be able to answer the above question with the aid of the pipeline diagram we’ve seen in lecture.
From last time ...

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>I1: count = count + 1</code></td>
<td><code>I2: count = count +1</code></td>
</tr>
</tbody>
</table>

Q: When Thread A and Thread B are executed concurrently, what issues might arise?
From last time …

When concurrently executing threads that share a variable, you must impose a specific order on instruction execution to achieve the desired use of the shared variable(s).

**Thread A**

- \( I_1: \text{count} = \text{count} + 1 \)
- \( a_1: \text{load count} \)
- \( a_2: \text{add 1} \)
- \( a_3: \text{store count} \)

**Thread B**

- \( I_2: \text{count} = \text{count} + 1 \)
- \( b_1: \text{load count} \)
- \( b_2: \text{add 1} \)
- \( b_3: \text{store count} \)

Q: What are histories for instruction execution in which the count updates behave as expected?

Q: What are histories for instruction execution in which the count updates do NOT behave as expected?
From last time ...

When concurrently executing threads that share a variable, you must impose a specific order on instruction execution to achieve the desired use of the shared variable(s).

```
I1: count = count + 1
a1: load count
a2: add 1
a3: store count
b1: load count
b2: add 1
b3: store count
```

Task: Be able to explain why the two “incorrect” histories shown left produce undesirable results, but the bottom two are “correct”.

“Incorrect” history: a1 < a2 < b1 < a3 < b2 < b3

“Incorrect” history: a1 < b1 < a3 < a2 < b2 < b3

Sample solution: a1 < a2 < a3 < b1 < b2 < b3

Sample solution: b1 < b2 < b3 < a1 < a2 < a3

Q: When an instruction set CANNOT be interrupted, we say that it is _________
From last time ...

When concurrently executing threads that share a variable, you must impose a specific order on instruction execution to achieve the desired use of the shared variable(s)

Sample “solution”:
- $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$
- $b_1 < b_2 < b_3 < a_1 < a_2 < a_3$

Q: When an instruction set CANNOT be interrupted, we say that it is ATOMIC
Today

Semaphores
We’ve discussed now several examples (you and Bob not eating lunch at the same time) whose concurrency problems can be “solved” by one thread “waiting” for another.

Q: How might you enforce a certain order of instruction execution?
We’ve discussed now several examples (you and Bob not eating lunch at the same time) whose concurrency problems can be “solved” by one thread “waiting” for another.

Q: How might you enforce a certain order of instruction execution?

```
sleep(3) - Linux man page

Name
sleep - sleep for the specified number of seconds

Synopsis
#include <unistd.h>
unsigned int sleep(unsigned int seconds);

Description
sleep() makes the calling thread sleep until seconds seconds have elapsed or a signal arrives which is not ignored.
```
### Question

What are the possible orderings, histories, of A1-A3 and M1-M3 if threads A and M are run concurrently?
Q: What are the possible orderings, histories, of A1-A3 and M1-M3 if threads A and M are run concurrently?

Desired execution order: A1 < A2 < M1 < M2 < M3 < A3
Desired execution order: A1 < M1 < A2 < M2 < M3 < A3

(on the board discussion)
Q: What are the possible orderings, histories, of A1-A3 and M1-M3 if threads A and M are run concurrently?

Desired execution order: A1 < A2 < M1 < M2 < M3 < A3
Desired execution order: A1 < M1 < A2 < M2 < M3 < A3

(on the board discussion)

Q: Why is using `sleep()` impractical?
Q: How is it done in the real world?
Q: How is it done in the real world?

Semaphores are the data structure implementation of message passing among threads, invented by Edsger Dijkstra

(Chapter 4 of the textbook)
Semaphores

- A data structure that contains only a single non-negative integer as a datum
Semaphores

- A data structure that contains only a single non-negative integer as a datum
- The integer can be initialized to any value, but once declared and set, its value can be modified only by several methods (there are no setter and getter methods)

Q: Why are there no setter and getter methods?

(HW2 question)
Semaphores

- A data structure that contains only a single non-negative integer as a datum
- The integer can be initialized to any value, but once declared and set, its value can be modified only by several methods (there are no setter and getter methods)
- Operation 1: increment
- Operation 2: decrement

<table>
<thead>
<tr>
<th>Semaphore</th>
<th>int value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ increment + decrement</td>
</tr>
</tbody>
</table>
Semaphores

- A data structure that contains only a single non-negative integer as a datum
- The integer can be initialized to any value, but once declared and set, its value can be modified only by several methods (there are no setter and getter methods)
- Operation 1: **increment**
- Operation 2: **decrement**
- The method Semaphore is the constructor; it creates and returns a reference variable to a new Semaphore

<table>
<thead>
<tr>
<th>Semaphore</th>
<th>int value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ Semaphore(int)</td>
</tr>
<tr>
<td></td>
<td>+ increment</td>
</tr>
<tr>
<td></td>
<td>+ decrement</td>
</tr>
</tbody>
</table>
Semaphores

- A data structure that contains only a single non-negative integer as a datum
- The integer can be initialized to any value, but once declared and set, its value can be modified only by several methods (there are no setter and getter methods)
- Operation 1: **increment**
- Operation 2: **decrement**
- The method Semaphore is the constructor; it creates and returns a reference variable to a new Semaphore

Q: Now that we have the structure, what is the behavior of the Semaphore? Or, how is it used?
If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks** itself and will not continue until ANOTHER thread increments the semaphore.

<table>
<thead>
<tr>
<th>Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>int value</td>
</tr>
<tr>
<td>+ Semaphore(int)</td>
</tr>
<tr>
<td>+ increment</td>
</tr>
<tr>
<td>+ decrement</td>
</tr>
</tbody>
</table>
If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks itself** and will not continue until ANOTHER thread increments the semaphore.

<table>
<thead>
<tr>
<th>Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>int value</td>
</tr>
<tr>
<td>+ Semaphore(int)</td>
</tr>
<tr>
<td>+ increment</td>
</tr>
<tr>
<td>+ decrement</td>
</tr>
</tbody>
</table>

Thread notifies the scheduler that it cannot proceed. Scheduler will prevent thread from running until an event occurs that causes the thread to become unblocked.
Semaphores

If decrement would result in the datum being negative, then the thread that issued the decrement blocks itself and will not continue until ANOTHER thread increments the semaphore.

If increment occurs, an already waiting thread is unblocked.

<table>
<thead>
<tr>
<th>Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>int value</td>
</tr>
<tr>
<td>+ Semaphore(int)</td>
</tr>
<tr>
<td>+ increment</td>
</tr>
<tr>
<td>+ decrement</td>
</tr>
</tbody>
</table>
Semaphores

If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks** itself and will not continue until ANOTHER thread increments the semaphore.

If increment occurs, an **already** waiting thread is **unblocked**.

Sometimes referred to as **waking**

<table>
<thead>
<tr>
<th>Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>int value</td>
</tr>
<tr>
<td>+ Semaphore(int)</td>
</tr>
<tr>
<td>+ increment</td>
</tr>
<tr>
<td>+ decrement</td>
</tr>
</tbody>
</table>
Semaphores

If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks** itself and will not continue until ANOTHER thread increments the semaphore.

If increment occurs, an **already** waiting thread is unblocked.

**Q:** Which thread is executed after a shared Semaphore’s value is incremented?
Semaphores

If decrement would result in the datum being \textbf{negative}, then the thread that issued the decrement \textbf{blocks} itself and will not continue until ANOTHER thread increments the semaphore.

If increment occurs, an \textbf{already} waiting thread is unblocked.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Semaphore} & \\
\hline
int value & \\
+ Semaphore(int) & \\
+ increment & \\
+ decrement & \\
\hline
\end{tabular}
\end{center}

Q: Which thread is executed after a shared Semaphore’s value is incremented?

A: Both the thread that is unblocked \textbf{AND} the thread that issued the increment \textbf{run (are scheduled)} concurrently ... Q: In which order?
Semaphores

If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks** itself and will not continue until ANOTHER thread increments the semaphore.

If increment occurs, an **already** waiting thread is unblocked.

**When a thread signals a Semaphore, it has no knowledge of how many other threads are waiting**
Semaphores

If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks** itself and will not continue until ANOTHER thread increments the semaphore.

If increment occurs, an **already** waiting thread is unblocked.

<table>
<thead>
<tr>
<th>Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>int value</td>
</tr>
<tr>
<td>+ Semaphore(int)</td>
</tr>
<tr>
<td>+ increment</td>
</tr>
<tr>
<td>+ decrement</td>
</tr>
</tbody>
</table>

Although a Semaphore has no getter or setter “methods,” the increment and decrement operations “return” the value in the Semaphore.
If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks** itself and will not continue until ANOTHER thread increments the semaphore.

If increment occurs, an **already** waiting thread is unblocked.

**Although a Semaphore has no getter or setter “methods,”** the increment and decrement operations “return” the value in the Semaphore.

- Incrementing may affect other threads
- Decrementing directly affects only the thread that issued the call
Semaphores

If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks** itself and will not continue until ANOTHER thread increments the semaphore.

If increment occurs, an **already** waiting thread is unblocked.

**Although a Semaphore has no getter or setter “methods,”** the increment and decrement operations “return” the value in the Semaphore.

Incrementing is therefore also referred to as **signal()**

Decrementing is therefore also referred to as **wait()**
Semaphores

If decrement would result in the datum being **negative**, then the thread that issued the decrement **blocks** itself and will not continue until ANOTHER thread increments the semaphore

If increment occurs, an **already** waiting thread is unblocked.

**Increment** and **decrement** refer to what a Semaphore DOES (to the value) **Signal** and **Wait** describe what they are USED FOR

Dijkstra, because of this “confusion” referred to increment as V, and decrement as P ... “use a meaningless name rather than a confusing one”
Syntax is straight-forward ... but it is OS specific, so the “pseudocode” description is used in most textbooks

Thread A

```java
aSem = Semaphore(3)
aSem.increment()
aSem.decrement()
```
Semaphores

Syntax is straight-forward ... but it is OS specific, so the “pseudocode” description is used in most textbooks

Thread A

```
aSem = Semaphore(3)
aSem.increment()
aSem.decrement()
```

Task: Draw the variable, object reference diagram that results after this line of code is executed
Semaphores

Syntax is straight-forward ... but it is OS specific, so the “pseudocode” description is used in most textbooks

Thread A

```java
aSem = Semaphore(3)
aSem.increment()
aSem.decrement()
```

Semaphore

value = 3
Semaphores

Syntax is straight-forward ... but it is OS specific, so the “pseudocode” description is used in most textbooks

Thread A

```java
aSem = Semaphore(3)
aSem.increment()
aSem.decrement()
```

Semaphore
value = 4

Q: When aSem’s value is incremented from 3 to to 4, what effect does that have on Thread A?
Semaphores

Syntax is straight-forward ... but it is OS specific, so the “pseudocode” description is used in most textbooks

Thread A

```python
aSem = Semaphore(3)
aSem.increment()
aSem.decrement()
```

Q: When aSem’s value is incremented from 3 to to 4, what effect does that have on Thread A?

Q: When aSem’s value is incremented to 4, what effect does that have on other threads?
Semaphores

Syntax is straight-forward ... but it is OS specific, so the “pseudocode” description is used in most textbooks

Thread A

```python
aSem = Semaphore(3)
aSem.increment()
aSem.decrement()
```

Semaphore

value = 3

Q: When `aSem`’s value is incremented from 3 to 4, what effect does that have on Thread A?

Q: When `aSem`’s value is incremented to 4, what effect does that have on other threads?

Q: When `aSem`’s value is decremented from 4 to 3, what effect does that have on Thread A and on other threads?
Semaphores

Another example ...

Thread A

```python
mySem = Semaphore(0)
mySem.increment()
mySem.decrement()
mySem.decrement()
mySem.decrement()
print("hello")
```

Q: What is achieved when the code in the yellow box is executed?
Another example...

Thread A

```python
mySem = Semaphore(0)
mySem.increment()
mySem.decrement()
mySem.decrement()
mySem.decrement()
print("hello")
```
Semaphores

Another example ...

Thread A

```python
mySem = Semaphore(0)
mySem.increment()
mySem.decrement()
mySem.decrement()
mySem.decrement()
print("hello")
```

Q: What is achieved when the code in the yellow box is executed?
Another example ...

Thread A

```python
mySem = Semaphore(0)
mySem.increment()
mySem.decrement()
mySem.decrement()
mySem.decrement()
print("hello")
```
Another example ...

Q: What is achieved when the code in the yellow box is executed?

```python
mySem = Semaphore(0)
mySem.increment()
mySem.decrement()  # mySem.value = 1
mySem.decrement()  # mySem.value = 0
mySem.decrement()  # mySem.value = -1
print("hello")
```
Semaphores

Another example ...

Thread A

mySem = Semaphore(0)
mySem.increment()
mySem.decrement()
mySem.decrement()
mySem.decrement()
print("hello")

Semaphore
value = 0
Semaphores

Another example ...

Thread A

```java
mySem = Semaphore(0)
mySem.increment()
mySem.decrement()
mySem.decrement()
mySem.decrement()
print("hello")
```

Semaphore

Q: What is achieved when the code in the yellow box is executed?
Another example...

Thread A

```python
mySem = Semaphore(0)
mySem.increment()
mySem.decrement()
mySem.decrement()
mySem.decrement()
print("hello")
```

Semaphore

value = 0

Thread A self stalls ... is the last decrement executed? Is the print statement executed?
Semaphores

Semaphores are used to “halt” a thread and to enforce running one thread in a specific sequence relative to another.
Semaphores

Assume $a_1$ writes to a file, and $b_1$ prints a line from the file (hence reads from the file).

Goal: We want $a_1$ to complete before $b_1$ begins.
Assume $a_1$ writes to a file, and $b_1$ prints a line from the file (hence reads from the file)

**Goal:** We want $a_1$ to complete before $b_1$ begins

```
sem = Semaphore(0)
```

In-class exercise
Semaphores

Assume a1 writes to a file, and b1 prints a line from the file (hence reads from the file)

Goal: We want a1 to complete before b1 begins

Already run code

```python
sem = Semaphore(0)
```

Thread A

```python
a1
sem.signal()
```

Thread B

```python
sem.wait()
b1
```

Task: Be able to explain why the above use of the Semaphore ensures the goal
Assume \texttt{a1} writes to a file, and \texttt{b1} prints a line from the file (hence reads from the file)

\textbf{Goal:} We want \texttt{a1} to complete before \texttt{b1} begins

\begin{center}
\begin{tabular}{ll}
\multicolumn{2}{c}{Already run code} \\
\texttt{sem = Semaphore(0)} & \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{ll}
\texttt{Thread A} & \texttt{Thread B} \\
\texttt{a1} & \texttt{sem.wait()} \\
\texttt{sem.signal()} & \texttt{b1} \\
\end{tabular}
\end{center}

\textbf{Q:} Does the use of the Semaphore by the two threads guarantee that Thread A will “complete” prior to Thread B executing?
Semaphores

Assume $a_1$ writes to a file, and $b_1$ prints a line from the file (hence reads from the file)

**Goal:** We want $a_1$ to complete before $b_1$ begins

Already run code

```python
sem = Semaphore(0)
```

Thread A

```python
a1
sem.signal()
```

Thread B

```python
sem.wait()
b1
```

We do not know how the OS will schedule the two threads.

**Q:** What are the scheduling choices?
Semaphores

Assume \( a_1 \) writes to a file, and \( b_1 \) prints a line from the file (hence reads from the file)

Goal: We want \( a_1 \) to complete before \( b_1 \) begins

Already run code

```python
sem = Semaphore(0)
```

Thread A

```python
a1
sem.signal()
```

Choice 1: Thread A < Thread B
Choice 2: Thread B < Thread A

Thread B

```python
sem.wait()
b1
```

Task: Be sure you understand both execution orders
Semaphores

Assume \( a_1 \) writes to a file, and \( b_1 \) prints a line from the file (hence reads from the file)

**Goal**: We want \( a_1 \) to complete before \( b_1 \) begins

**Already run code**

\[
\text{sem} = \text{Semaphore}(0)
\]

**Thread A**

\[
\begin{align*}
\text{a}_1 \\
\text{sem}.\text{signal}()
\end{align*}
\]

**Thread B**

\[
\begin{align*}
\text{b}_1 \\
\text{sem}.\text{wait}()
\end{align*}
\]

Choice 1: Thread A < Thread B
Choice 2: Thread B < Thread A

Q: What is the sequence of events when Thread A executes first? When Thread B executes first? (on the board walk through)
Semaphores

In-class exercise

Thread A

\[
\begin{array}{c}
\text{a1} \\
\text{a2}
\end{array}
\]

Thread B

\[
\begin{array}{c}
\text{b1} \\
\text{b2}
\end{array}
\]

Goals

- a1 must happen before b2
- b1 must happen before a2

Hint : Q: How many semaphores are needed?
Semaphores

Sample solution (in class)

Already run code

Thread A

- a1
- a2

Thread B

- b1
- b2

Goals

- a1 must happen before b2
- b1 must happen before a2
Take-home exercise / question

We’ve seen this: \( \text{sem} = \text{Semaphore}(0) \)
Take-home exercise / question

We’ve seen this: \( \text{sem} = \text{Semaphore}(0) \)

Why would we want to do this: \( \text{sem} = \text{Semaphore}(2) \)
Up Next ...

Multiplex
Mutex