From last time ...

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

Q: What are some of the possible orderings, histories, of A1-A3 and M1-M3 if threads A and M are run concurrently, such that A3 < M3?
From last time ...

• What is a semaphore?
• How/when is a semaphore used?
• What are the two methods common to all semaphore “implementations”? 

From last time ...

- A data structure that contains only a non-negative integer as a datum
- The integer can be initialized to any non-negative 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** or **signal**
- Operation 2: **decrement** or **wait**
- The method Semaphore is the constructor; it creates and returns a reference variable to a new 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>
From last time ...

- A data structure that contains only a non-negative integer as a datum
- The integer can be initialized to any non-negative 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 or signal
- Operation 2: decrement or wait
- The method Semaphore is the constructor; it creates and returns a reference variable to a new 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 A

```java
aSem = Semaphore(3)
daSem.increment()
daSem.decrement()
```
From last time ...

• A data structure that contains only a non-negative integer as a datum
• The integer can be initialized to any non-negative 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 or signal
• Operation 2: decrement or wait
• The method Semaphore is the constructor; it creates and returns a reference variable to a new Semaphore

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

Semaphore

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

Thread A

```
value = 3
```
From last time ...

- A data structure that contains only a non-negative integer as a datum
- The integer can be initialized to any non-negative 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** or **signal**
- Operation 2: **decrement** or **wait**
- The method Semaphore is the constructor; it creates and returns a reference variable to a new Semaphore

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

Thread A
From last time ...

• A data structure that contains only a non-negative integer as a datum
• The integer can be initialized to any non-negative 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 or signal
• Operation 2: decrement or wait
• The method Semaphore is the constructor; it creates and returns a reference variable to a new Semaphore

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

```
<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 A

```java
aSem = Semaphore(3)
aSem.increment()
aSem.decrement()
```
Task: Use semaphore(s) to ensure that a1 occurs before b1 occurs?
Task: Use semaphore(s) to ensure that \( a_1 \) occurs before \( b_1 \) occurs?

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

- Thread A
  - \( a_1 \)
  - \text{sem.signal}()

- Thread B
  - \text{sem.wait}()
  - \( b_1 \)

- Thread A < Thread B
- Thread B < Thread A

Q: How does using semaphores enable either of the execution orders to maintain the constraint?
From last time ...

Task: Use semaphore(s) to ensure that \texttt{a1} occurs before \texttt{b1} occurs?

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

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a1} \texttt{sem.signal()}</td>
<td>\texttt{sem.wait()} \texttt{b1}</td>
</tr>
</tbody>
</table>

- \textbf{Thread A < Thread B} A executes \texttt{a1}, then \texttt{sem} incremented to 1
- \textbf{Thread B < Thread A} \texttt{B decrements} \texttt{sem} to 0 (no self block), then \texttt{b1} executed
From last time ...

Task: Use semaphore(s) to ensure that \texttt{a1} occurs before \texttt{b1} occurs?

\begin{align*}
\text{sem} & = \text{Semaphore}(0) \\
\text{Thread A} & \\
\text{a1} & \\
\text{sem.signal()} & \\
\text{Thread B} & \\
\text{sem.wait()} & \\
\text{b1} & \\
\end{align*}

- Thread A < Thread B
- **Thread B < Thread A**

B decrements sem to -1 (attempts, self blocks), then waits
A executes a1, then increments sem to 1
B is unblocked, decrements sem to 0, then b1 is executed
Task: Use semaphore(s) to ensure the history requirements

Requirements:

- a1 must happen before b2
- b1 must happen before a2
From last time ...

```
s1 = Semaphore(0)
s2 = Semaphore(0)
```

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>s1.signal()</td>
<td>s2.signal()</td>
</tr>
<tr>
<td>a2</td>
<td>s1.wait()</td>
</tr>
<tr>
<td>s2.wait()</td>
<td>b2</td>
</tr>
</tbody>
</table>
From last time ...

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

```
sem1 = Semaphore(0)
sem2 = Semaphore(0)
```

Q: What are the order of execution choices?

Thread A

```
a1
sem1.signal()
sem2.wait()
a2
```

Thread B

```
b1
sem2.signal()
sem1.wait()
b2
```
From last time ...

- \( a_1 \) must happen before \( b_2 \)
- \( b_1 \) must happen before \( a_2 \)

\[
\begin{align*}
\text{sem1} &= \text{Semaphore}(0) \\
\text{sem2} &= \text{Semaphore}(0)
\end{align*}
\]

Thread A

\[
\begin{align*}
& a_1 \\
& \text{sem1}.\text{signal}() \\
& \text{sem2}.\text{wait}() \\
& a_2
\end{align*}
\]

Thread B

\[
\begin{align*}
& b_1 \\
& \text{sem2}.\text{signal}() \\
& \text{sem1}.\text{wait}() \\
& b_2
\end{align*}
\]

Q: What are the order of execution choices?

Thread A \(<\) Thread B

Thread B \(<\) Thread A
From last time ...

- \( a_1 \) must happen before \( b_2 \)
- \( b_1 \) must happen before \( a_2 \)

\[
\begin{align*}
\text{sem1} &= \text{Semaphore}(0) \\
\text{sem2} &= \text{Semaphore}(0)
\end{align*}
\]

Thread A:
- \( a_1 \) signal()
- \( \text{sem2.wait()} \)
- \( a_2 \)

Thread B:
- \( b_1 \) signal()
- \( \text{sem2.wait()} \)
- \( b_2 \)

Semaphore 0
Semaphore 0

Thread A < Thread B
Thread B < Thread A
From last time ...

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

```python
sem1 = Semaphore(0)
sem2 = Semaphore(0)
```

Thread A:
- `a1`
- `sem1.signal()`
- `sem2.wait()`
- `a2`

Thread B:
- `b1`
- `sem2.signal()`
- `sem1.wait()`
- `b2`

Semaphore 1
Semaphore 0

Thread A < Thread B
Thread B < Thread A

`a1` must happen before `b2`
`b1` must happen before `a2`
From last time ...

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

\[
\text{sem1} = \text{Semaphore}(0) \quad \text{sem2} = \text{Semaphore}(0)
\]

Semaphore 1
Semaphore 0

Thread A < Thread B
Thread B < Thread A

Because Thread A self blocks, Thread B will “eventually” start up, regardless how long “after” Thread A it is scheduled.
From last time ...

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

```
sem1 = Semaphore(0)
sem2 = Semaphore(0)
```

Thread A
```
\begin{align*}
a1 & \\
sem1.signal() & \\
sem2.wait() & \\
a2 & \\
\end{align*}
```

Thread B
```
\begin{align*}
b1 & \\
sem2.signal() & \\
sem1.wait() & \\
b2 & \\
\end{align*}
```

Thread A < Thread B
Thread B < Thread A
From last time …

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

Semaphore 0

Thread A

a1
sem1.signal()

sem2.wait()
a2

Thread B

b1
sem2.signal()

sem1.wait()
b2

• a1 must happen before b2
• b1 must happen before a2

Thread B increments sem2 (from 0 to 1), and Thread A, because it is “waiting” will right away decrement sem2 back to 0
From last time …

- $a_1$ must happen before $b_2$
- $b_1$ must happen before $a_2$

$\text{sem1} = \text{Semaphore}(0)$
$\text{sem2} = \text{Semaphore}(0)$

Thread A

- $a_1$
- \text{sem1.signal()}
- \text{sem2.wait()}
- $a_2$

Thread B

- $b_1$
- \text{sem2.signal()}
- \text{sem1.wait()}
- $b_2$

Semaphore

0

Semaphore

0

Thread A < Thread B

Thread B < Thread A

- $a_1$\n- $\text{sem1} \rightarrow 1$
- $\text{sem2} \rightarrow -1$ (attempt, Thread A self blocks)
- $b_1$
- $\text{sem2} \rightarrow 0$ (Thread A is unblocked, A and B run concurrently)
- $\text{sem1} \rightarrow 0$ then $a_2$, or vice versa
From last time ...

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

The diagram shows the following:

Semaphore = Semaphore(0)
Semaphore = Semaphore(0)

Thread A
- a1
- sem1.signal()
- sem2.wait()
- a2

Thread B
- b1
- sem2.signal()
- sem1.wait()
- b2

• a1 must happen before b2
• b1 must happen before a2
From last time …

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

Q: Are the requirements satisfied?

Thread A

- a1
- sem1.signal()
- sem2.wait()
- a2

Thread B

- b1
- sem2.signal()
- sem1.wait()
- b2

a1
sem1 -> 1
sem2 -> -1 (attempt, Thread A self blocks)
b1
sem2 -> 0 (Thread A is unblocked, A and B run concurrently)
sem1 -> 0 then a2, or vice versa
b2
From last time ...

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

Semaphore
0

Semaphore
0

Q: What is the execution order for Thread B < Thread A
From last time ...

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

```
sem1 = Semaphore(0)
sem2 = Semaphore(0)

Thread A
a1
sem1.signal()
sem2.wait()
a2

Thread B
b1
sem2.signal()
sem1.wait()
b2
```

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

Semaphore

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore (0)</td>
</tr>
</tbody>
</table>

Semaphore

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore (0)</td>
</tr>
</tbody>
</table>

Thread A < Thread B

Thread B < Thread A

b1
sem2 -> 1 (no blocking)
sem1-> -1 (attempt, Thread B self blocks)
From last time ...

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

**Thread A**

- a1
- sem1.signal()
- sem2.wait()
- a2

**Thread B**

- b1
- sem2.signal()
- sem1.wait()
- b2

**Q: Are the requirements satisfied?**

- b1
- sem2 -> 1 (no blocking)
- sem1-> -1 (attempt, Thread B self blocks)
- a1
- sem1 -> 0 (Thread B is unblocked, B and A run concurrently)
- b2 or sem2 -> 0 (no blocking), both executed
- a2
From last time ...

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

Semaphore sem1 = Semaphore(0)
Semaphore sem2 = Semaphore(0)

Thread A
- a1
- sem1.signal()
- sem2.wait()
- a2

Thread B
- b1
- sem2.signal()
- sem1.wait()
- b2

Q: For 4 different instructions (a1, a2, b1, b2) among 2 threads, how many orderings are there such that a1 < a2 and b1 < b2?

Thread A < Thread B
Thread B < Thread A
From last time …

- $a_1$ must happen before $b_2$
- $b_1$ must happen before $a_2$

```python
sem1 = Semaphore(0)
sem2 = Semaphore(0)
```

**Thread A**

```python
a1
sem1.signal()
sem2.wait()
a2
```

**Thread B**

```python
b1
sem2.signal()
sem1.wait()
b2
```

**Semaphore**

- $sem_1 = Semaphore(0)$
- $sem_2 = Semaphore(0)$

Thread A $<$ Thread B

Thread B $<$ Thread A

**Q:** Of these, how many “satisfy” the original requirements?
From last time ...

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

Semaphore

\[
\begin{align*}
\text{sem1} &= \text{Semaphore}(0) \\
\text{sem2} &= \text{Semaphore}(0)
\end{align*}
\]

Semaphore

0

Semaphore

0

Thread A

\[
\begin{align*}
a1 \\
\text{sem1}.\text{signal}() \\
\text{sem2}.\text{wait}() \\
a2
\end{align*}
\]

Thread B

\[
\begin{align*}
b1 \\
\text{sem2}.\text{signal}() \\
\text{sem1}.\text{wait}() \\
b2
\end{align*}
\]

Thread A < Thread B

Thread B < Thread A

Thread A

\[
\begin{align*}
b1 \\
\text{sem2} \to 1 \text{ (no blocking)} \\
\text{sem1} \to -1 \text{ (attempt, Thread B self blocks)} \\
a1 \\
\text{sem1} \to 0 \text{ (Thread B is unblocked, B and A run concurrently)} \\
b2 \text{ or sem2} \to 0 \text{ (no blocking), both executed} \\
a2
\end{align*}
\]

Thread B

\[
\begin{align*}
b1 \\
\text{sem1} \to 1 \text{ (attempt, Thread A self blocks)} \\
b1 \\
\text{sem2} \to 0 \text{ (Thread A is unblocked, A and B run concurrently)} \\
\text{sem1} \to 0 \text{ then a2, or vice versa} \\
b2
\end{align*}
\]
Today

Mutex
Multiplex
Performance Gain
You've been given a program whose pseudocode is shown right. The /beginConcurrent, /endConcurrent specify the start and end portions of the code that is run concurrently.

Add semaphore(s) as needed, and invoke the wait() and signal() methods as needed, so when run to completion the program outputs the following:

x=13, y=11, z=24

Assume that the variables x, y, and z are global variables, and that each of the two statements in Procedure1 and Procedure2 are atomic.
In already seen examples, semaphores were used to enforce a specific instruction history.

Q: Can semaphores be used for other purposes?

Q: Must semaphores be initialized to zero?
Mutex

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

Why would we want to do these:

\( \text{semB} = \text{Semaphore}(2) \)

\( \text{semC} = \text{Semaphore}(3) \)

\( \text{semD} = \text{Semaphore}(44) \)
Mutex

Define mutex
Mutex

You : eat breakfast, work, eat lunch

Scenario : You and Bob are bitter enemies (no sending of messages), AND you do NOT want to be in the lunch room at the same time ... how can this be achieved?

Bob : eat breakfast, work, eat lunch
Mutex

Scenario: You and Bob are bitter enemies (no sending of messages), AND you do NOT want to be in the lunch room at the same time ... how can this be achieved?

You do not want the “eat lunch” events to occur at the same time.
### Mutex

**Scenario:** You and Bob are bitter enemies (no sending of messages), AND you do NOT want to be in the lunch room at the same time ... how can this be achieved?

You do not want the “eat lunch” events to occur at the same time.

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eat breakfast</td>
<td>Eat breakfast</td>
</tr>
<tr>
<td>Retrieve conch shell</td>
<td>Retrieve conch shell</td>
</tr>
<tr>
<td>Eat Lunch</td>
<td>Eat Lunch</td>
</tr>
<tr>
<td>Put conch shell back</td>
<td>Put conch shell back</td>
</tr>
</tbody>
</table>

**Task:** use semaphore(s) to implement the mutex solution?

**You:**
- eat breakfast
- work
- eat lunch

**Bob:**
- eat breakfast
- work
- eat lunch

**Keep a conch shell in the office ... ONLY if you have the conch shell can you go into the lunch room**
### In-class exercise

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

Add semaphores to the above two threads to enforce mutual exclusion to the shared variable `count`.

Order does not matter (either Thread A or Thread B can “go first”).

**Hint**: This can be implemented with a single semaphore.
Mutex

mutex = Semaphore(1)

Semaphore
1

Thread A
mutex.wait()
count = count + 1
mutex.signal()

Thread B
mutex.wait()
count = count + 1
mutex.signal()

Q: What are possible sequences of scheduling/execution?
Mutex

mutex = Semaphore(1)

Semaphore

1

Thread A
mutex.wait()
count = count + 1
mutex.signal()

Thread B
mutex.wait()
count = count + 1
mutex.signal()

Thread A < Thread B
Thread B < Thread A
Mutex

Thread A
- `mutex.wait()`
- `count = count + 1`
- `mutex.signal()`

Thread B
- `mutex.wait()`
- `count = count + 1`
- `mutex.signal()`

A decrements: `mutex` -> 0

Q: Does this self-block A?
Mutex

mutex = Semaphore(1)

Semaphore
0

Thread A < Thread B
Thread B < Thread A

A decrements: mutex -> 0

Q: Because of duplicated pipelines, threading, etc., Thread A and Thread B are being executed concurrently ... at this point, what happens if B for some reason is scheduled?
Mutex

- `mutex = Semaphore(1)`
- Semaphore
  - 0

Thread A:
- `mutex.wait()`
- `count = count + 1`
- `mutex.signal()`

Thread B:
- `mutex.wait()`
- `count = count + 1`
- `mutex.signal()`

A decrements: `mutex -> 0`
B decrements: `mutex -> -1` (attempt, blocks)

Thread A < Thread B
Thread B < Thread A
Mutex

\[ \text{mutex} = \text{Semaphore}(1) \]

Semaphore

0

Thread A < Thread B

Thread B < Thread A

Thread A

mutex.wait()

\[
\text{count} = \text{count} + 1
\]

mutex.signal()

Thread B

mutex.wait()

\[
\text{count} = \text{count} + 1
\]

mutex.signal()

A decrements : mutex -> 0
B decrements : mutex -> -1 (attempt, blocks)
A updates count
Mutex

```
mutex = Semaphore(1)
```

Semaphore

\[
\begin{align*}
\text{Thread A} & : \text{mutex.wait()} \\
& : \text{count} = \text{count} + 1 \\
& : \text{mutex.signal()}
\end{align*}
\]

\[
\begin{align*}
\text{Thread B} & : \text{mutex.wait()} \\
& : \text{count} = \text{count} + 1 \\
& : \text{mutex.signal()}
\end{align*}
\]

A decrements : mutex -> 0
B decrements : mutex -> -1 (attempt, blocks)
A updates count
A increments : mutex -> 0 (B unblocked)
Mutex

\[
\text{mutex} = \text{Semaphore}(1)
\]

Semaphore

\[
\text{count} = \text{count} + 1
\]

\[
\text{mutex}\text{.signal}()
\]

\[
\text{mutex}\text{.wait}()
\]

Thread A

\[
\text{count} = \text{count} + 1
\]

\[
\text{mutex}\text{.signal}()
\]

Thread B

A decrements: \text{mutex} \rightarrow 0

B decrements: \text{mutex} \rightarrow -1 (attempt, blocks)

A updates count

A increments: \text{mutex} \rightarrow 0 (B unblocked)

B updates count
Mutex

\[ \text{mutex} = \text{Semaphore}(1) \]

| Semaphore | 1 |

**Thread A**
- `mutex.wait()`
- `count = count + 1`
- `mutex.signal()`

**Thread B**
- `mutex.wait()`
- `count = count + 1`
- `mutex.signal()`

A decrements: \( \text{mutex} \rightarrow 0 \)
B decrements: \( \text{mutex} \rightarrow -1 \) (self blocks)
A updates count
A increments: \( \text{mutex} \rightarrow 0 \) (B unblocked)
B updates count
B increments: \( \text{mutex} \rightarrow 1 \)

Thread A < Thread B
Thread B < Thread A
Mutex

mutex = Semaphore(1)

Semaphore 1

Thread A
mutex.wait()
count = count + 1
mutex.signal()

Thread B
mutex.wait()
count = count + 1
mutex.signal()

Thread A < Thread B

Thread B < Thread A

Step through the same code for the other sequence
Q: What is one “functional” purpose for initializing a semaphore to a non-zero value when 1, 2, 3, or $n$ threads use that semaphore?

```plaintext
aSemaphore = Semaphore(8)
```

Diagram:
- **Semaphore**
  - 8
In class exercise

Thread C: \( x = x + 1 \)
Thread D: \( x = x - 12 \)
Thread E: \( x = 4 \)

Create one or more semaphores and invoke the wait and signal methods among Threads C-E to enforce an upper limit of 2 for the number of threads that can access concurrently the shared variable \( x \).
Multiplex

\[ \text{multiplex} = \text{Semaphore}(2) \]

Semaphore

2

<table>
<thead>
<tr>
<th>Thread C</th>
<th>Thread D</th>
<th>Thread E</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplex.wait()</td>
<td>multiplex.wait()</td>
<td>multiplex.wait()</td>
</tr>
<tr>
<td>( x = x + 1 )</td>
<td>( x = x - 12 )</td>
<td>( x = 4 )</td>
</tr>
<tr>
<td>multiplex.signal()</td>
<td>multiplex.signal()</td>
<td>multiplex.signal()</td>
</tr>
</tbody>
</table>

**Q: How many possible choices of scheduling Threads?**

- Thread A < Thread B < Thread C
- Thread A < Thread C < Thread B
- Thread B < Thread A < Thread C
- Thread B < Thread C < Thread A
- Thread C < Thread A < Thread B
- Thread C < Thread B < Thread A
Multiplex

\[ \text{multiplex} = \text{Semaphore}(2) \]

Thread C
- multiplex.wait()
- \( x = x + 1 \)
- multiplex.signal()

Thread D
- multiplex.wait()
- \( x = x - 12 \)
- multiplex.signal()

Thread E
- multiplex.wait()
- \( x = 4 \)
- multiplex.signal()

Q: How many possible choices of scheduling Threads?

- Thread C < Thread D < Thread E
- Thread C < Thread E < Thread D
- Thread D < Thread C < Thread E
- Thread D < Thread E < Thread C
- Thread E < Thread C < Thread D
- Thread E < Thread D < Thread C

Regardless of the scheduling, only 2 Threads at any one time may “access” the code that updates the value of \( x \).
Q: If there is more than one “correct” instruction execution history, does it matter which one we use?
Performance Gain

Q: If there is more than one “correct” instruction execution history, does it matter which one we use?

Q: If a program requires the use of 50 semaphores or 500 semaphores (both impose the “correct” instruction history), which one should we choose, and why?
Modern processors have large but slow “main” memory.

Smaller (than memory) but faster (than memory) caches (multiple levels).

Limited size but very fast registers and ALUs.
Modern processors have large but slow “main” memory. Smaller (than memory) but faster (than memory) caches (multiple levels). Limited size but very fast registers and ALUs.

Q: How does this modern processor architecture dictate how we should design software?
Modern processors have large but slow “main” memory

Smaller (than memory) but faster (than memory) caches (multiple levels)

Limited size but very fast registers and ALUs

Q: How does this modern processor architecture dictate how we should design software?

A program that is “too large” to fit into cache must be fetched from main memory, which is slow. Hence the goal is to write programs that are cache friendly.
But even IF a program can fit in its entirety in the cache ...
Scenario: For a hypothetical program that can fit in cache, assume that Register 2 (R2) needs such-and-such data ... where does the computer/program look for that data, and if not found, where does it get it from?
But even IF a program can fit in its entirety in the cache ...

Scenario: For a hypothetical program that can fit in cache, assume that Register 2 (R2) needs such-and-such data ... where does the computer/program look for that data, and if not found, where does it get it from?

When the CPU issues a read (load the registers) request, first-level cache logic checks whether that data item is already in cache. If ...

- Yes -> ________
- No -> ________
But even IF a program can fit in its entirety in the cache ...

Scenario: For a hypothetical program that can fit in cache, assume that Register 2 (R2) needs such-and-such data ... where does the computer/program look for that data, and if not found, where does it get it from?

When the CPU issues a read (load the registers) request, first-level cache logic checks whether that data item is already in cache. If ...

- Yes -> Cache hit
- No -> Cache miss

Q: If a cache miss occurs, where is the sought-after item retrieved from?
Modern Processors - Cache

But even IF a program can fit in its entirety in the cache ...

If there is a level 1 cache miss, level 2 cache is checked. If that is a miss, then main memory, and potentially electronic disc (Hard Disk) is checked.

Q: If data is retrieved from L2 cache, what does that mean?

Q: Where in L1 cache should data be placed?
But even IF a program can fit in its entirety in the cache ...

**Miss rate**: typically 3-10% for L1 cache, and less than 2% for L2 cache

**Hit time**: Time to deliver a line in cache to the processor. Typically 1 clock cycle to L1 cache, 3-8 clock cycles for L2 cache
But even IF a program can fit in its entirety in the cache ...

**Miss rate**: typically 3-10% for L1 cache, and less than 2% for L2 cache

**Hit time**: Time to deliver a line in cache to the processor. Typically 1 clock cycle to L1 cache, 3-8 clock cycles for L2 cache

**Miss penalty**: time incurred because of cache miss. Typically 25-100 clock cycles

**Average access time** = hit time + miss rate * miss penalty
Q: How does cache improve performance?

Q: If cache access is 20ns, and memory access is 40ns, is a computer with cache twice as fast as a computer without cache?
Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of $\tau$ “faster” than memory?

Q: What factors other than cache and memory access speed determine the speed of execution of a program?
Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of $\tau$ “faster” than memory?

$\beta$ = cache reuse ratio; the fraction of loads or reads that can be reused from cache

Q: Intuitively, do we want a higher or lower $\beta$?
Modern Processors - Cache

Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of \( \tau \) “faster” than memory?

\[ \beta = \text{cache reuse ratio; the fraction of loads or reads that can be reused from cache} \]
\[ T_m = \text{access time to main memory} \]
\[ T_c \]

Q: What is \( T_c \) in terms of \( T_m \)?
Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of $\tau$ “faster” than memory?

$\beta$ = cache reuse ratio; the fraction of loads or reads that can be reused from cache

$T_m$ = access time to main memory

$T_c = T_m / \tau$ = access time for cache

Q: What is $\tau$?
Modern Processors - Cache

Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of $\tau$ “faster” than memory?

$\beta = \text{cache reuse ratio; the fraction of loads or reads that can be reused from cache}$

$T_m = \text{access time to main memory}$

$T_c = T_m / \tau = \text{access time for cache}$

$\tau$ is the ratio of $T_m/T_c$

Q: What is the average access time of data in terms of $T_m$, $T_c$, and $\beta$?
Modern Processors - Cache

Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of $\tau$ “faster” than memory?

$\beta = \text{cache reuse ratio; the fraction of loads or reads that can be reused from cache}$

$T_m = \text{access time to main memory}$

$T_c = T_m / \tau = \text{access time for cache}$

$T_{av} = \beta T_c + (1 - \beta) T_m$

Q: What value should go here?
Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of $\tau$ “faster” than memory?

$\beta$ = cache reuse ratio; the fraction of loads or reads that can be reused from cache

$T_m$ = access time to main memory

$T_c = T_m / \tau$ = access time for cache

$T_{av} = \beta T_c + (1-\beta) T_m$
Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of τ “faster” than memory?

\[ \beta = \text{cache reuse ratio; the fraction of loads or reads that can be reused from cache} \]

\[ T_m = \text{access time to main memory} \]

\[ T_c = \frac{T_m}{\tau} = \text{access time for cache} \]

\[ T_{av} = \beta T_c + (1-\beta)T_m \]

\[ G(\tau,\beta) = \frac{T_m}{T_{av}} = \text{access performance gain} \]

(derivation, on the board)
Modern Processors - Cache

Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of $\tau$ “faster” than memory?

$\beta$ = cache reuse ratio; the fraction of loads or reads that can be reused from cache

$T_m$ = access time to main memory

$T_c = T_m / \tau$ = access time for cache

$T_{av} = \beta T_c + (1-\beta)T_m$

$G(\tau, \beta) = T_m / T_{av} = \text{access performance gain}$

$= \tau T_c / (\beta T_c + (1-\beta)\tau T_c)$

$= \tau / (\beta + \tau (1-\beta))$

Q: Using this formula, under what conditions does using cache have a significant improvement over a system that does not use cache?
Q: How does cache improve performance?

Q: How do we estimate the performance gain of a cache that is a factor of $\tau$ “faster” than memory?

$\beta = \text{cache reuse ratio; the fraction of loads or reads that can be reused from cache}$

$T_m = \text{access time to main memory}$

$T_c = T_m / \tau = \text{access time for cache}$

$T_{av} = \beta T_c + (1-\beta)T_m$

$G(\tau, \beta) = T_m / T_{av} = \text{access performance gain}$

$= \tau T_c / (\beta T_c + (1-\beta)\tau T_c)$

$= \tau / (\beta + \tau(1-\beta))$

Q: Is cache that is twice as fast going to result in a system that is twice as good? (the salesman pitch)
Modern Processors - Cache

\[ G(\tau, \beta) = \frac{\tau}{\beta + \tau(1-\beta)} \]
\[ \beta = \text{cache reuse ratio} \]
\[ \tau = \frac{T_m}{T_c} \]

Assume a program with cache reuse ratio of 0.6, and a computer with 60ns cache access time and 120ns access time to main memory.

**Q:** What performance improvement can you expect if you upgrade your cache so that it has a 30ns access time?

In class exercise
Q: When $\tau$ is increased from 5 to 10, how is performance improved?
Up Next

- Concurrent Matrix Multiplication
- Shared Variable Programming
- Cache Friendly code