Announcements

Homework #1 – Hints and additional explanation

• Question 3
  
  • What are the possible starting values for the variable x?

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA1 : x = x+1</td>
<td>IB1 : x = x-2</td>
<td>IC1 : x = x+2</td>
</tr>
<tr>
<td>IA2 : print(x)</td>
<td>IB2 : print(x)</td>
<td>IC2 : print(x)</td>
</tr>
</tbody>
</table>

Homework #2

• Will be posted to the Course Website on Thursday
• A mix of book questions and a programming task
Announcements - Lab

Last week’s lab

• Issue threads
• No “coordination” among threads
Announcements - Lab

Last week’s lab

- Issue threads
- No “coordination” among threads

This week’s lab

- C++, asynchronous execution
- “collecting” the outputs of multiple threads
From last time ...

• What is a semaphore?
• How/when is a semaphore used?
• 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
• Operation 2: decrement
• The method Semaphore is the constructor; it creates and returns a reference variable to a new Semaphore

Thread A

```
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>

Semaphore

<table>
<thead>
<tr>
<th>value = 3</th>
</tr>
</thead>
</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
- Operation 2: decrement
- The method Semaphore is the constructor; it creates and returns a reference variable to a new Semaphore.

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

Thread A

```
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>

Semaphore

<table>
<thead>
<tr>
<th>value = 4</th>
</tr>
</thead>
</table>

aSem
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
- Operation 2: decrement
- The method Semaphore is the constructor; it creates and returns a reference variable to a new Semaphore

```
Semaphore
int value
+ Semaphore(int)
+ increment
+ decrement
```

Thread A

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

```
sem = Semaphore(0)

Thread A
a1
sem.signal()

Thread B
sem.wait()
b1
```

Assume **a1 must occur before b1**. Then for threads A and B, execution choices are

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

You should be able to understand how/why using semaphores will enable either of the execution orders to maintain the constraint.
From last time...

Assume \textbf{a1 must occur before b1}. Then for threads A and B, execution choices are

- **Thread A < Thread B**
  - A executes a1, then sem incremented to 1
  - B decrements sem to 0 (no self block), then b1 executed

- **Thread B < Thread A**
  - B decrements sem to -1 (attempt, self blocks), then waits
  - A executes a1, then increments sem to 1
  - B is unblocked, decrements sem to 0, then b1 is executed
From last time ...

Requirements:

- $a_1$ must happen before $b_2$
- $b_1$ must happen before $a_2$
sem1 = Semaphore(0)
sem2 = Semaphore(0)

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 $a_2$

Q: What are the order of execution choices?

Thread A

1. $a_1$
2. $sem1$.signal()
3. $sem2$.wait()
4. $a_2$

Thread B

1. $b_1$
2. $sem2$.signal()
3. $sem1$.wait()
4. $b_2$

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

Q: What are the order of execution choices?

Thread A < Thread B
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
```

Semaphore 0
Semaphore 0

Thread A < Thread B

Thread B < Thread A
From last time ...

- \(a1\) must happen before \(b2\)
- \(b1\) must happen before \(a2\)

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

Semaphore 1  Semaphore 0

Thread A < Thread B

Thread B < Thread A

Thread A

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

Thread B

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

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

Semaphore

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

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

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

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

Semaphore

- sem1 = Semaphore(0)
- sem2 = 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

Semaphore

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

a1
sem1 -> 1
sem2 -> -1 (attempt, Thread A self blocks)
b1
From last time ...

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

Semaphore 1
Semaphore 0

Thread A

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

Thread B

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

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 …

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

Semaphore 0
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

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

Semaphore
0

Semaphore
0

Thread A

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

Thread B

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

\[a1 \rightarrow 1\]
\[\text{sem1} \rightarrow -1 \text{ (attempt, Thread A self blocks)}\]
\[b1\]
\[\text{sem2} \rightarrow 0 \text{ (Thread A is unblocked, A and B run concurrently)}\]
\[\text{sem1} \rightarrow 0 \text{ then a2, or vice versa}\]
\[b2\]
From last time …

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

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

**Thread A**
- \text{a1}
- \text{sem1.signal()}
- \text{sem2.wait()}
- \text{a2}

**Thread B**
- \text{b1}
- \text{sem2.signal()}
- \text{sem1.wait()}
- \text{b2}

**Q: Are the requirements satisfied?**

- a1
- \text{sem1} \rightarrow 1
- \text{sem2} \rightarrow -1 \text{ (attempt, Thread A self blocks)}
- b1
- \text{sem2} \rightarrow 0 \text{ (Thread A is unblocked, A and B run concurrently)}
- \text{sem1} \rightarrow 0 \text{ then a2, or vice versa}
- b2

Thread A < Thread B

Thread B < Thread A
From last time ...

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

Semaphore 0
Semaphore 0

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

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

Q: What is the execution order for Thread B < Thread A

Thread A < Thread B

Thread B < Thread A
From last time ...

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

Semaphore

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

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

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

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 \(a2\)

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

**Q: Are the requirements satisfied?**

- \(b1\) sem2 -> 1 (no blocking)
- sem1 -> -1 (attempt, Thread B \textbf{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 …

- \(a_1\) 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**
- \(b_1\)
- \(\text{sem2.signal()}\)
- \(\text{sem1.wait()}\)
- \(b_2\)

Q: Are there other instruction histories in addition to the below 2 for which the requirements are maintained?

- \(b_1\) \(\text{sem2} \rightarrow 1\) (no blocking)
- \(\text{sem1} \rightarrow -1\) (attempt, Thread B self blocks)
- \(a_1\)
- \(\text{sem1} \rightarrow 0\) (Thread B is unblocked, B and A run concurrently)
- \(b_2\) or \(\text{sem2} \rightarrow 0\) (no blocking), both executed
- \(a_2\)

- \(a_1\)
- \(\text{sem1} \rightarrow 1\)
- \(\text{sem2} \rightarrow -1\) (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
- \(b_2\)
From last time …

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

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

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

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), how many orderings are there such that `a1 < a2` and `b1 < b2`?

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

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

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

\[
\begin{array}{c}
\text{Semaphore} \\
0
\end{array}
\quad \begin{array}{c}
\text{Semaphore} \\
0
\end{array}
\]

Thread A < Thread B
Thread B < Thread A

\[
\begin{align*}
\text{Thread A} & \quad \text{Thread B} \\
a1 & \quad b1 \\
\text{sem1.signal()} & \quad \text{sem2.signal()} \\
\text{sem1.wait()} & \quad \text{sem2.wait()} \\
a2 & \quad b2
\end{align*}
\]

a1 < a2 < b1 < b2  \quad b1 < b2 < a1 < a2
a1 < b1 < a2 < b2  \quad b1 < a1 < b2 < a2
a1 < b1 < b2 < a2  \quad b1 < a1 < a2 < b2

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

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

$$\text{sem}_1 = \text{Semaphore}(0)$$
$$\text{sem}_2 = \text{Semaphore}(0)$$

Thread A

1. $a_1$
2. $\text{sem}_1$.signal()
3. $\text{sem}_2$.wait()
4. $a_2$

Thread B

1. $b_1$
2. $\text{sem}_2$.signal()
3. $\text{sem}_1$.wait()
4. $b_2$

$a_1 < a_2 < b_1 < b_2$

$b_1 < b_2 < a_1 < a_2$

Thread A < Thread B

Thread B < Thread A

- $b_1$
  - sem2 -> 1 (no blocking)
  - sem1-> -1 (attempt, Thread B self blocks)
- $a_1$
  - sem1 -> 0 (Thread B is unblocked, B and A run concurrently)
- $b_2$ or sem2 -> 0 (no blocking), both executed
- $a_2$

- $a_1$
  - sem1 -> 1
  - sem2 -> -1 (attempt, Thread A self blocks)
- $b_1$
  - sem2 -> 0 (Thread A is unblocked, A and B run concurrently)
- $\text{sem}_1$ -> 0 then $a_2$, or vice versa
- $b_2$
From last time ...

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

Why would we want to do this: \[
\text{sem} = \text{Semaphore}(2) \]
Deadlock
Mutex
Multiplex
Deadlock

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

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

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

We saw that the above use of two semaphores enforced that a1 happen before b2, and b1 happen before a2

Q: Is use of semaphores always “safe”?
We saw that the above use of two semaphores enforced that $a_1$ happen before $b_2$, and $b_1$ happen before $a_2$.

Q: Is use of semaphores always “safe”?

Task: Add method calls to sem1 and sem2 such that Thread A and Thread B do not execute to completion.
**Deadlock**

Thread A

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

Thread B

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

Q: What is the result of the execution of the two Threads, assuming A is scheduled first?
Deadlock

Thread A

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

Thread B

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

Q: What is the result of the execution of the two Threads, assuming A is scheduled first?

```
a1
```
Deadlock

Thread A

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

Thread B

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

Q: What is the result of the execution of the two Threads, assuming A is scheduled first?

a1
sem1 -> -1 (attempt, A self blocks)
Deadlock

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

Thread A

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

Thread B

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

Q: What is the result of the execution of the two Threads, assuming A is scheduled first?

\[
\begin{align*}
&\text{a1} \\
&\text{sem1} \rightarrow -1 \text{ (attempt, A self blocks)} \\
&\text{b1}
\end{align*}
\]
Deadlock

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

Semaphore -1
Semaphore -1

Thread A

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

Thread B

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

Q: What is the result of the execution of the two Threads, assuming A is scheduled first?

a1
sem1 -> -1 (attempt, A self blocks)
b1
sem2 -> -1 (attempt, B self blocks)
Deadlock

Thread A

sem1.wait()
sem2.signal()

Thread B

sem2.wait()
sem1.signal()

Q: What is the result of the execution of the two Threads, assuming A is scheduled first?

a1
sem1.wait() (attempt, A self blocks)

b1
sem2.wait() (attempt, B self blocks)

Q: At this point what happens?
Deadlock

Thread A

sem1.wait()
sem2.signal()

Thread B

b1
sem2.wait()
sem1.signal()

Deadlock

Q: What is the result of the execution of the two Threads, assuming A is scheduled first?

a1
sem1 -> -1 (attempt, A self blocks)
b1
sem2 -> -1 (attempt, B self blocks)

Deadlock
Deadlock

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

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

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

Q: What is the result of the execution of the two Threads, assuming B is scheduled first?
Deadlock

Q: What is the result of the execution of the two Threads, assuming B is scheduled first?

Also a Deadlock

Q: Is it possible for two threads to implement semaphores such that some execution order results in a deadlock, and another does not?

(HW2 question)
Mutex

From a previous lecture ....

You: eat breakfast, work, eat lunch, call Bob

Bob: eat breakfast, wait for call, 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?
**Mutex**

From a previous lecture ....

You: eat breakfast, work, eat lunch, call Bob

Bob: eat breakfast, wait for call, 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?

Keep a conch shell in the office ... ONLY if you have the conch shell can you go into the lunch room

<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>

Q: How can using semaphores “implement” the mutex solution
In class exercise

Thread A

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

Thread B

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

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

Hint: This can be implemented with a single semaphore
In class exercise

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

In class exercise

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

In class exercise

```python
mutex = Semaphore(1)

Semaphore
0

Thread A < Thread B
Thread B < Thread A

A decrements : mutex -> 0

Q: Does this self-block A?
```

Thread A

```python
mutex.wait()
count = count + 1
mutex.signal()
```

Thread B

```python
mutex.wait()
count = count + 1
mutex.signal()
```
Mutex

In class exercise

Thread A

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

Thread B

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

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

**In class exercise**

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

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

**mutex = Semaphore(1)**

- Semaphore
  - 0

**Thread A < Thread B**

- A decrements: `mutex -> 0`

**Thread B < Thread A**

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

In class exercise

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)
A updates count

mutex = Semaphore(1)
Semaphore 0

Thread A < Thread B
Thread B < Thread A
Mutex

In class exercise

`mutex = Semaphore(1)`

Semaphore

0

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

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

In class exercise

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)
A updates count
A increments : mutex -> 0 (B unblocked)
B updates count

mutex = Semaphore(1)
Semaphore 0

Thread A < Thread B
Thread B < Thread A
Mutex

In class exercise

```
mutex = Semaphore(1)

Semaphore

1

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

A decrements: mutex -> 0
A updates count
A increments: mutex -> 0 (B unblocked)
B updates count
B increments: mutex -> 1
```

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

B decrements: mutex -> -1 (self blocks)
B updates count
```

Thread A < Thread B
Thread B < Thread A
Mutex

In class exercise

```
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?

\[ \text{aSemaphore} = \text{Semaphore}(8) \]
In class exercise

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = x + 1$</td>
<td>$x = x - 12$</td>
<td>$x = 4$</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Semaphore</th>
<th>2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</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

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

Semaphore

| 2 |

Thread A
- `multiplex.wait()`
- `x = x + 1`
- `multiplex.signal()`

Thread B
- `multiplex.wait()`
- `x = x - 12`
- `multiplex.signal()`

Thread C
- `multiplex.wait()`
- `x = 4`
- `multiplex.signal()`

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

Regardless of the scheduling, only 2 Threads at any one time may “access” the code that updates the value of x.
Lab on Friday