Today: Monitors and Condition Variables

- Monitors
  - What is wrong with semaphores?
- Two types of monitors: Mesa and Hoare
  - How do we implement monitors?
  - What are they?

Last Class: Synchronization for Readers/Writers

- Statement is possible in either case:
  - Favor writers
  - Favor readers
- Two possible solutions using semaphores
  - Allow only one writer at a time
  - Allow multiple readers to concurrently access a data
- Readers/Writers problem:
What is a Monitor?

- Monitors require all data to be private.
- Monitor method at a time.
- Monitors guarantee mutual exclusion, i.e., only one thread may execute a given
  unlike classes.

Unrelated, the synchronization operations all together,
A monitor is similar to a C++ class that ties the data, operations, and in

What's Wrong with Semaphores?

- There is no control or guarantee of proper usage.
- They serve two purposes: mutual exclusion and scheduling constraints.
- Access to semaphores can come from anywhere in a program.
- Semaphores control access.
- There is no linguistically convenient between the Semaphore and the data to
  which the
  implementation, but have the following drawbacks.
- Semaphores are a huge step up from the equivalent file I/O or

Solutions: use a higher level primitive called monitors
It is simple to turn a Java class into a monitor:

**Implementing Monitors in Java**

- Makes all methods synchronized (or at least the non-private ones).
- Make all data private.

**Monitor operations:**

- Enables new threads to start.
- Temporarily releases the monitor to complete.
- Acquires the mutex on the start.
- Encapsulates the shared data you want to protect.
- Managed concurrent access to shared data.
- The monitor uses the lock to ensure that only a single thread is active in the monitor.

- Condition variables enable threads to go to sleep inside of critical sections.
- The lock allows providers mutual exclusion for shared data.
- The lock is not a new instance of any inference.

A monitor defines a lock and zero or more condition variables for

**Monitors: A Formal Definition**
Operations

Rule: thread must hold the lock when doing condition variable

1. \texttt{Wait(lock)}: atomic (release lock, go to sleep), when the process wakes up it waits for something in the queue

Condition variables support these operations:

- \texttt{Signal()}: wake up all waiting threads
- \texttt{Broadcast()} wake up all waiting threads

Critical section:

Condition variable: is a queue of threads waiting for something inside a critical section

\begin{tikzpicture}[scale=0.7]
\begin{scope}[every node/.style={draw, rounded corners, text width=10cm, align=center}]
\node (condition) {Condition Variables};
\end{scope}
\end{tikzpicture}

Operations on Condition Variables

\begin{tikzpicture}[scale=0.7]
\begin{scope}[every node/.style={draw, rounded corners, text width=10cm, align=center}]
\node (condition) {Condition Variables};
\end{scope}
\end{tikzpicture}

- Any lock held by the thread is automatically released when the thread is put to sleep
- Condition variables enable a thread to sleep inside a critical section

Solution: use condition variables

- The thread could sleep forever because it doesn't know when the thread is awoken.
- \texttt{Wait(lock)} can be used to ensure the thread wakes up only when it is ready to proceed.

How can we change \texttt{remove()} to wait until something is on the queue?

\begin{tikzpicture}[scale=0.7]
\begin{scope}[every node/.style={draw, rounded corners, text width=10cm, align=center}]
\node (condition) {Condition Variables};
\end{scope}
\end{tikzpicture}
When the thread that was waiting and is now executing exits or waits again, it
• The thread that signals gives up the lock and the waiting thread gets the lock.

Hoare-style: (most textbooks)
• The waiting thread waits for the lock.
• The thread that signs keeps the lock (and thus the processor).

Mesas-styler: (Nacho's Java, and most real operating systems)
• If there is a waiting thread, one of the threads starts executing; others must wait.
  • what happens with semaphores.
• No waiting threads — the signal continues and the signal is effectively lost (unlike

What should happen when signal() is called?

Mesas versus Hoare Monitors

{  
  remove and return item;  
  wait () // : give up lock and go to sleep  
  while queue is empty  
} ( )  

public Object synchronized remove() 
{
  ;
  notify () // put item on queue;
} ( )

public void synchronized add( Object item ) ( 

private queue // : queue data
}  

class queue

: effectively one condition variable per object.
: use notifyAll() to wake up all waiting threads.
: use notify() to signal that the condition a thread is waiting on is satisfied.
: use wait() to give up the lock

Condition Variables in Java
Readers/Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```

Readers-Write using Monitors (Java)

```java
class ReaderWriter
{
    private synchronized void synchronizeReaders()
    {
        // read
        if (numReaders == 0)
            numReaders++; // notice

        private synchronized void prepareReaders()
        {
            // prepare
            numReaders = 0;
        }
    }
}
```
The class must explicitly provide the lock, acquire and release it correctly.

- No synchronization keyword
- Monitors in C++ are more complicated.

```java
void wait() { waiters += 1; }
void signal() { waiters -= 1; }
```
Bounded Buffer using Hoare-style condition-style condition variables

```c
{  
    Look.'<queue'><.'()  
    empty.<queue>()  
    count = count-1  
    item = buffer[(last-count) mod N]  
    while (not-empty(look))  
    {  
        if (count == 0)  
        {  
            Look.<queue><.append(look)  
        }  
        Look.<queue><.remove(look)  
        count++  
    }  
    last = (last + 1) mod N  
    buffer[look] = item  
    empty.<queue><.remove(look)  
    count = count  
    Look.<queue><.append(look)  
}
```

Monitors in C++: Example

```c
{  
    Look.'queue'<.remove()  
    Look.<queue><.get()  
    while queue is empty  
    Look.<queue><.get()  
    item = queue.<queue><.get()  
    Remove.<queue>()  
    Look.<queue><.get()  
    queue.<queue><.add()  
}
```

It is possible to implement monitors with semaphores:

- Condition to access state variables and do their job.
- Condition variables are not and so a result they must be in a critical
- Regardless of the order of execution
- Semaphores—Wait and Signal are commutative, the result is the same

**Thread continues**

- If a thread does a semaphores—Wait, the value is decremented and the
- On a semaphore Signal, if no one is waiting, the value of the semaphore is
- If a thread does a condition—Wait, it waits.
- On a condition variable Signal, if no one is waiting, the signal is a no-op.
- Condition variables do not have any history, but semaphores do.

---

### Semaphores versus Condition Variables

---

```c
{ 
  semaphore<signal
  } ()

  condition<signal

  
  lock=>acquire()
  semaphore<wait()}

  lock=release()

  condition=wait(lock)
}
```

**How about this?**

May get deadlock. Why?

But condition variables only work inside a lock. If we use semaphores inside a lock, we have

```c
{ 
  semaphore<signal
  } ()

  condition<signal

  
  lock=wait(lock)
}
```

**Can we build monitors out of semaphores?** After all, semaphores provide atomic operations

---

### Semaphores versus Monitors

---
Implementing Monitors with Semaphores

```java
public class MyMonitor {
    private int nextCount = 0;
    private int waiters = 0;
    private Semaphore wait;
    private Semaphore signal;

    public MyMonitor() {
        this.wait = new Semaphore(0);
        this.signal = new Semaphore(0);
    }

    public void enter() {
        if (waiters >= 0) {
            waiters += 1;
            signal.acquire();
            wait.release();
        }
    }

    public void leave() {
        waiters -= 1;
        wait.acquire();
        if (waiters == 0) {
            signal.release();
        }
    }

    // Other methods...
}
```

Implementing Monitors with Semaphores

```java
public class MyMonitor {
    private int nextCount = 0;
    private int waiters = 0;
    private Semaphore wait;
    private Semaphore signal;

    public MyMonitor() {
        this.wait = new Semaphore(0);
        this.signal = new Semaphore(0);
    }

    public void enter() {
        if (waiters >= 0) {
            waiters += 1;
            signal.acquire();
            wait.release();
        }
    }

    public void leave() {
        waiters -= 1;
        wait.acquire();
        if (waiters == 0) {
            signal.release();
        }
    }

    // Other methods...
}
```
It is possible to implement monitors with semaphores

- Locks implemented by following the monitor rules, for acquiring and releasing
- C++ does not provide a monitor construct, but monitors can be
  implemented by following the monitor rules, for acquiring and releasing
- Condition variables release mutex temporarily
- Monitor wraps operations with a mutex

**Summary**

- Other semantics?
  - Is this more semantics of Mesa semantics? What would you change to provide this

```cpp
lock->wait(); // Wrap a new thread into the monitor
  the
next->wait(); // Resume a suspended thread
  if (nextCount > 0)
}{
  if (nextCount > 0)
    lock->waite() // Wrapper code for all methods on the shared data

  lock->waite() //

  Monitor::someMethod() //

  lock->waite() //
}
```

**Using The Monitor Class**
Homework 2: due Oct 17

Lab 2: due Oct 18

Exam 1: Oct 24 (6:15-7:45, room FERN II)