Compare semaphores and monitors

- Two types of monitors: Mesa and Hoare
  - How do we implement monitors?
  - What are they?

Monitors

What is wrong with semaphores?

Today: Monitors and Condition Variables

Synchronization 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 data

Readers/writers problem:

Last Class: Synchronization for Readers/Writers
- 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
    method.

Unlike classes, •

particularly, the synchronization operations all in

•

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

What is a Monitor?

Department of Computer Science, Ulster University

Solution: use a higher level primitive called monitors

- 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.
  - Semaphore controls access.
  - There is no linguistic connection between the semaphore and the data to which the
    implementation, but have the following drawbacks.

What's wrong with Semaphores?

Semaphore are a huge step up from the equivalent load/store
Implementing Monitors in Java

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

It is simple to turn a Java class into a monitor:

Monitor operations:
- Releasing a lock at the same time it picks the thread to sleep.
- Condition variables enable threads to go to sleep inside critical sections, by
  - The lock also provides mutual exclusion for shared data.
  - The monitor uses the lock to ensure that only a single thread is active in the monitor
  - Managing concurrent access to shared data.

A Monitor defines a lock and zero or more condition variables for
Operations.

**Rule:** thread must hold the lock when doing condition variable operations.

1. `Wait(lock)`: atomic (release lock, go to sleep), when the process wakes up it
   Condition variable: is a queue of threads waiting for something inside a

**Operations on Condition Variables**

- 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
- queue, add an item to it, and wake up the sleeping thread
- But if we hold on to the lock and sleep, then other threads cannot access the shared
- Logically, we want to go to sleep inside of the critical section

How can we change `remove()` to wait until something is on the queue?

**Condition Variables**
When the thread that was waiting and is now executing exits or waits again, it
• The thread that signaled gives up the lock and the waiting thread gets the lock.

**Hoare-style (most textbooks)**

The waiting thread waits for the lock.

• The waiting thread keeps the lock (and thus the processor).

** Mesa-style (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 signaler continues and the signal is effectively lost (unlike

What should happen when signal() is called?

---

** Mesa versus Hoare Monitors**

---

```java
{ 
  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/ Writers using Monitors (Java)

```java
class ReaderWriter {
    private volatile int numberOfReaders = 0;
    private volatile int numberOfWriters = 0;

    public synchronized void prepareReader() {
        ++numberOfReaders;
        if (numberOfReaders == 0) notify();
    }

    public synchronized void prepareWriter() {
        ++numberOfWriters;
        if (numberOfWriters == 0) notify();
    }

    public synchronized void removeAndReturnItem() {
        wait();
        if (queue.isEmpty()) notify();
    }

    public Object remove() {
        return queue.poll();
    }

    public synchronized void add(Object item) {
        queue.add(item);
        if (numberOfReaders > 0) notifyAll();
    }

    public synchronized void queueData() {
        // ... some data...
    }

    private static class Queue {
        private final LinkedBlockingQueue<Object> queue = new LinkedBlockingQueue();
    }
}
```

Hoare-style: we can change the value in remove to null because the waiting thread
runs immediately after an item is added to the queue.

Hoare-style: we can change the while loop in remove to an if statement because the waiting thread

some other thread could add the lock and remove the item before it gets to run.

Mesz-style: the waiting thread may need to wait again after it is awakened, because

but we can simplify it for Hoare-style semantics:

Mesz versus Hoare Monitors (cont.)

The synchronized queueing example above works for either style of monitor.
The class must explicitly provide the lock, acquire, and release it correctly.

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

```
 Monitors in C++
```

Readers/Writers using Monitors (Java)
Monitors in C++: Example

```cpp
class Queue {
public:
  void Add();  // lock before using data
  void Remove();  // ok to access shared data
  void Lock();  // lock before using data
  void Unlock();  // unlock after access
private:
  int queueData();  // queue item from queue
  lock->Acquire();  // lock before using data
  lock->Release();  // unlock after access
}

Monitors using Hoare-style condition variables

```
It is possible to implement monitors with semaphores:

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

Thread continues:
- If a thread does a semaphore → Wait, the value is decremented and the thread continues.
- On a semaphore Signal, if no one is waiting, the value of the semaphore is incremented.
- 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->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();
    semaphore->wait();
} ( )
```

and again. Does the following work?

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

Semaphores versus Monitors
Implementing Monitors with Semaphores

```java
class Monitor
{
    private Semaaphore wait
    private Semaaphore signal

    public void monitor()
    {
        // code to be protected by monitor
    }
}

newCount = 1;
while (true)
{
    wait();
    count = 1;
}
```

Implementing Monitors with Semaphores

```java
public class Monitor
{
    private semaaphore wait
    private semaaphore signal

    public void monitor()
    {
        // code to be protected by monitor
    }
}
```
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
- Condition variables release mutex temporarily
- Monitor wraps operations with a mutex

### Summary

#### Other semantics?

Is this more semantics or Mesa semantics? What would you change to provide the

```c
{
    lock<->synchronized();
    add a new thread into the monitor
    else
        resume a suspended thread
    if (nextCount < 0)
        <ope on data and call to conditionwait() and conditionsignal()>
    lock the monitor
    // lock<->wait();
    Monitor::synchronized();
    wrapper code for all methods on the shared data
    //
```