Compare semaphore and monitors

- Two types of monitors: Mesas and Hore
- How do we implement monitors?
- What are they?

Monitors

What is wrong with semaphores?

Today: Monitors and Condition Varnibles

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 a data

Readers/Write 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
    exclusive method.

Unlike classes,

- A particular synchronization operation all together,
  - A monitor is similar to a class that ties the data, operations, and in

What is a Monitor?

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.
- Semaphores control access.
- There is no linguistic connection between the semaphore and the data to which the
  implementation, but have the following drawbacks.
- Semaphores are a huge step up from the equivalent load/store

What's Wrong with Semaphores?
{ return item; 
remove item; 
if queue not empty 
} 
public Object synchronize() 
{ 
put item on queue; 
} } 
}

private 
queue data // Queue data

class Queue

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

**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:

**Implementing Monitors in Java**

- Releases the mutex at the end.
- Releases the mutex when it can continue.
- Temporarily releases the mutex if it can't complete.
- Temporarily releases the mutex at the start.
- Temporarily releases the shared data.
- Releases the mutex at the end.
- Temporarily releases the shared data you want to protect.

**Monitor Operations:**

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

**A Monitor: A Formal Definition**

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

Rule: thread must hold the lock when doing condition variable

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

Operations on Condition Variables

- Any lock held by the thread is atomically 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?
released the lock back to the signaling thread.

When the thread that was waiting and is now executing exits or waits again, it

- The thread that signals releases the lock and the waiting thread gets the lock.

**Hoare-style (most textbooks)**

The waiting thread waits for the lock.

- The thread that signals keeps the lock (and thus the processor).

**Mesas-style (Nachos, 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?

**Mesas versus Hoare Monitors**

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

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

public 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**
class ReaderWriter
{
    private synchronized void prepareReaders()
    {
        if (numReaders == 0) notify();
        numReaders += 1;
    }

    private synchronized void preparewriters()
    {
        if (numWriters == 0) notify();
        numWriters += 1;
    }

    // Reader
    public void read() { ... }

    // Writer
    public void write() { ... }
}

class Reader
{
    public void read() { ... }
}

class Writer
{
    public void write() { ... }
}

// Readers/Writers using Monitors (Java)

private synchronized void prepareReaders()
{
    if (numReaders == 0) notify();
    numReaders += 1;
}

private synchronized void prepareWriters()
{
    if (numWriters == 0) notify();
    numWriters += 1;
}

// Reader
public void read()
{
    while (true) // while queue becomes empty
    { // public synchronized remove()
        private synchronized remove();
        // put item on queue!
    }
}

// Writer
public void write()
{
    synchronized (monitor) // public void synchronized add( Object item )
    {
        // queue data
        // private void synchronized read() public void synchronized remove()
        notify();
    }
}

// runs immediately after an item is added to the queue.
Hoare-Style: we can change the "while" to "if" because the waiting
thread can change the "if" to "while" to get to run.

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

Hoare-Style: we can simplify it for Hoare-Style semantics:

The synchronized queuing example above works for either Style of monitor,

Meze Versus Hoare Monitors (cont.)
The class must explicitly provide the lock, acquire and release it correctly:

- No synchronization keyword

Monitors in C++ are more complicated.

Monitors in C++
Bounded Buffer using Hoare-style condition-style condition variables

Monitors in C++: Example
It is possible to implement monitors with semaphores.

section to access state variables and do their job.

Condition variables are not, and as a result they must be in a critical region of the order of execution.

Semaphore→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
  increased.
- If a thread does a Condition→Wait, it waits.
- On a Condition→Signal, if no one is waiting, the signal is a nop.

Condition variables do not have any history, but semaphores do.

### Semaphores versus Condition Variables

```c
{ Semaphore<signal>() ; 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>() ; Semaphore<signal>() ;
  Condition<signal>() ;
  { Semaphore<wait>() ; Semaphore<wait>() ;
    } ;
}
```

and guessing. Does the following work?

Can we build monitors out of semaphores? After all, semaphores provide atomic operations.
Implementing Monitors with Semaphores

nextCount = 0

Semaphore wait // next Count - 1
Semaphore signal // next Count += 1

don't signal if nobody is waiting

do signal if no thread is waiting

wait on the condition

allow a new thread in the monitor

resume a suspended thread

condition wait // next Count > 0

condition signal // next Count < 0

wait for < wait

next Count += 1

wait for < wait

next Count = waiters = 0

lock Monitor // nobody in the monitor
call = 0

condition wait on condition variable

monitor: Monitor

next Count = 0

Semaphore wait // number of threads suspended
Semaphore signal // suspend this thread when another

Semaphore wait // number of threads waiting on condition
Semaphore lock // a case for every condition

dez-behete protected by monitor

Semaphore signal // suspend a thread on a wait
Semaphore wait // shared data

protected: void conditionWait()

protected: void conditionSignal()

class Monitor
It is possible to implement monitors with semaphores.

Locks are implemented by following the monitor rules for acquiring and releasing them.

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 how semantics or Mesa semantics? What would you change to provide the

```c
{ 
    // Allocate a new thread into the monitor
    lock->start(true); 
    // Resume a suspended thread
    next->start();
    if (next->count < 0) 
        // Copy data and calls to conditionwait() and conditionsignal() 
        lock->wait();
    } 
}
```

Using the Monitor Class
Announcements

- Homework 2: due Oct 16
- Lab 2: due Oct 24
- Exam I: Oct 24 (6:15-7:45, room FERN II)