[Return to the lecture notes index](https://www.andrew.cmu.edu/course/15-440-kesden/applications/ln/index.html)

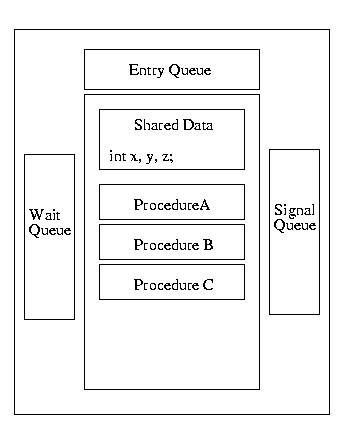
**Lecture 6 (September 11, 2008)**

**Monitors**

A *monitor* is a synchronization tool designed to make a programmer's life simple. A monitor can be thought of as a conceptual box. A programmer can put functions/procedures/methods into this box and the monitor makes him a very simple guarantee: only one function within the monitor will execute at a time -- mutual exclusion will be guaranteed.

Furthermore, the monitor can protect shared data. Data items declared within the monitor can only be accessed by functions/procedures/methods within the monitor. Therefore mutual exclusion is guaranteed for these data items. Functions/procedures/methods outside of the monitor can not corrupt them.

If nothing is executing within the monitor, a thread can execute one of its procedures/methods/functions. Otherwise, the thread is put into the entry queue and put to sleep. As soon as a thread exits the monitor, it wakes up the next process in the entry queue.



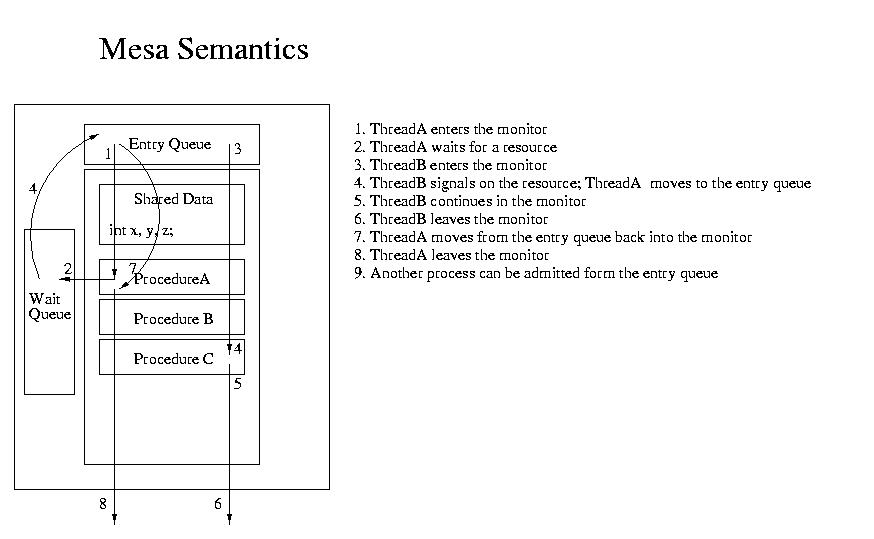
The picture gets a bit messier when we consider that threads executing within the monitor may require an unavailable resource. When this happens, the thread waits for this resource, using the wait operation of a condition variable. At this point, another thread is free to enter the monitor.

Now let me suggest that while a second thread is running in the monitor, it frees a resource required by the first. It signals that the resource that the first thread is waiting for becomes available. What should happen? Should the first thread be immediately awakened or should the second thread finish first? This situation gives rise to different versions of monitor sematics.

We're going to run through three different models first. But, to be honest, I think the one example implemented three different ways, is more than worth these thousand words.

**Mesa Semantics**

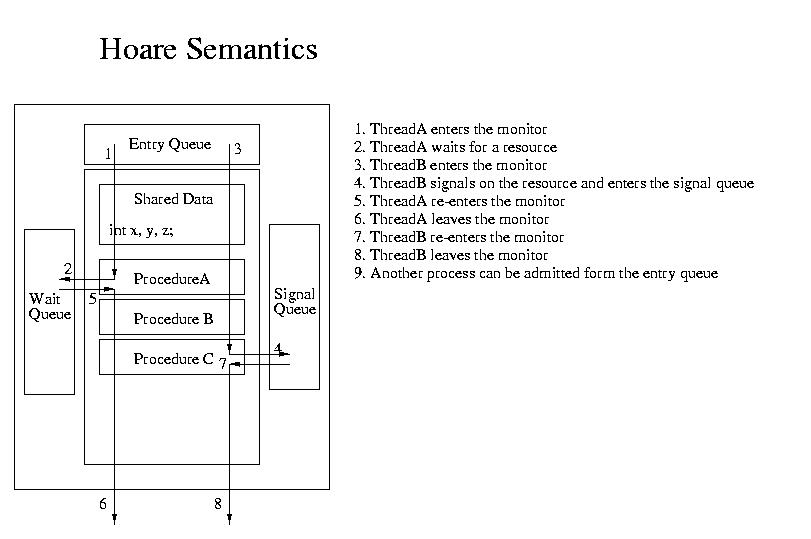
With Mesa semantics, when the first thread's resource becomes available, it is moved from the signal queue back to the entry queue. The second thread finishes executing, and the first thread will eventually be processes from the entry queue.



This interpretation while simple, has a slight glitches. Perhaps by the time that the first process begins to execute again, the event it was waiting for has passed and the resource is again unavailable. The cycle may repeat.

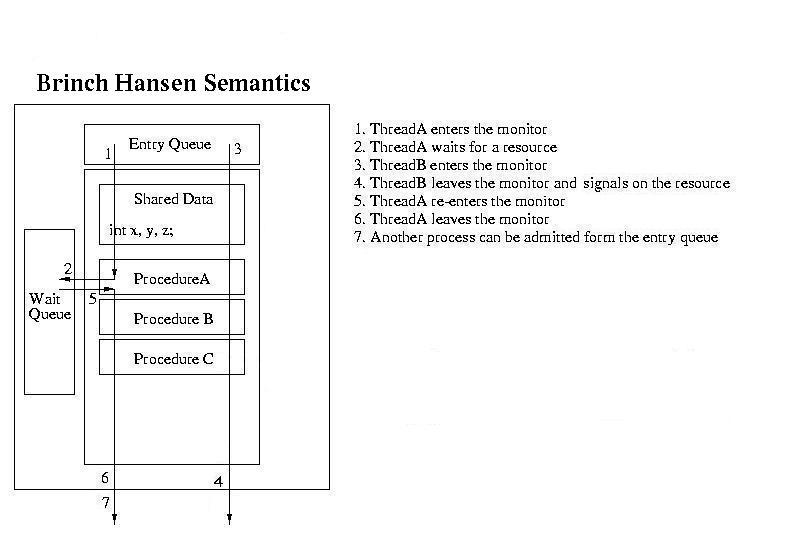
**Hoare Semantics**

Hoare advocates a slightly different interpretation. If the first thread's condition is satisfied, it should immediatly execute. The second thread is put into a signal queue. When a thread exits the monitor, a thread from the signal queue is restarted. Only when the signal queue is empty, is the entry queue used. This approach is more compilcated, but leads to some nicer proofs.



**Brinch Hanson Semantics**

There is also a third type of semantics that often exists in monitors, Brinch Hanson semantics. It incorporates only the union of mesa and Hoare semantics. The Brinch Hansen monitor only allows a process or thread to signal upon exit from the monitor. At that point, the signaled process or thread can run; the signalling process has already left the monitor.



**A Monitor Question From An Old Operating Systems Midterm**

3. [15 points] Fire? What fire?

Fire! The 15-412 students drop their exams and rush to evacuate via the only exterior doorway. Meanwhile, the valiant civil servants of the Pittsburgh Fire Bureau rush toward the same doorway to douse the fiery inferno. But alas, the celebrating students can't exit through the doorway at the same time the firefighters enter. Unless the situation is controlled, the firefighters will enjoy a picnic lunch outdoors and the students will become toast.

You must model a solution to this terrible plight that allows the students to escape and enjoy their weekend and allows the firefighters to enter the building and prevent any more singed concrete.

Below are the rules that allow maximum use of the doorway, without deadlock:

* There is only one doorway.
* The doorway can accommodate no more than one student at a time.
* The doorway can accommodate no more than two firefighters at a time.
* The doorway cannot be shared by students and firefighters at the same time.
* A student should not enter the doorway if a firefighter is currently trying to enter.

Furthermore, your solution must ensure the following:

* Firefighters cannot be starved.
* Students can only be starved by firefighters.
* There is no deadlock.

Your solution should use a single monitor as the tool for guaranteeing mutual exclusion and synchronization. Please state which semantics are required by the monitor used in your solution: Hoare, Mesa, or Brinch Hansen. Be sure to declare an initialize all shared variables.

Your solution should have a single monitor with the following entry procedures:

* FirefighterEnterDoorway
* FireFighterExitDoorway
* StudentEnterDoorway
* StudentExitDoorway

**A Mesa Semantics Solution**

In order to solve this problem, we must begin by asking ourselves the question, "What is shared?" The answer to this question is the doorway -- out monitor must control access to the doorway.

What is the sharing discipline? It is not mututal exclusion. It is two firefighters or one student, such that firefighters can starve students, but not vice versa.

To do this we can envision a system where students and firefighters get in line in front of the doorway. If the doorway is empty, the arriving person goes right into it. Otherwise he/she gets into the right line. Upon exiting the doorway, the person signals a person or two people to enter, as appropriate.

MESA MONITOR doorway

{

   int firefighterInDoorway = 0;

   int firefighterWaiting = 0;

   int studentInDoorway = 0;

   condition student, firefighter;

   Entry FirefighterEnterDoorway()

   {

     fireFighterWaiting++;

     while ( (studentInDoorway) || (firefighterInDoorway >= 2) )

         firefighter.wait();

      firefighterInDoorway++;

      firefighterWaiting--;

   }

   Entry FirefighterExitDoorway()

   {

     firefighterInDoorway--;

     if (firefighterWaiting())

         firefighter.signal();

     else

        if (!firefighterInDoorway)

             student.signal();

   }

   Entry StudentEnterDoorway()

   {

        while ( (studentInDoorway) || (firefighterInDoorway)

                (firefighterWaiting) )

             student.wait();

        studentInDoorway++;

   }

   Entry StudentExitDoorway()

   {

       studentInDoorway--;

       if (firefighterWaiting)

       {

          firefighter.signal();

          firefighter.signal();

       }

       else

         student.signal();

   }

}

**A Hoare Semantics Solution**

In order to convert the Mesa semantics solution into something that functions with Hoare semantics, we must look at two things:

1. We don't need to loop around the predicates (conditions) for which we are waiting -- no one can run between the time that we are signalled and the time that we run. The whiles can become ifs.
2. We need to check the code after ever time we signal and ask ourselves if it is still safe -- we need to remember that we could be preempted before it runs. Are all of the resources still available? In the state that we expect?

In this case, we can change the while's to ifs (whiles are still safe), and the code after signals is safe. The only time we do something after signalling is when we signal twice in a row. In that case the number of firefighters in the doorway could have changed from 0 to 1. But even if it is now 1, we're still safe -- it could not have gone to 2.

HOARE MONITOR doorway

{

   int firefighterInDoorway = 0;

   int firefighterWaiting = 0;

   int studentInDoorway = 0;

   condition student, firefighter;

   Entry FirefighterEnterDoorway()

   {

     firefighterWaiting++;

     if ( (studentInDoorway) || (firefighterInDoorway >= 2) )

         firefighter.wait();

      firefighterInDoorway++;

      firefighterWaiting--;

   }

   Entry FirefighterExitDoorway()

   {

     firefighterInDoorway--;

     if (firefighterWaiting())

         firefighter.signal();

     else

        if (!firefighterInDoorway)

             student.signal();

   }

   Entry StudentEnterDoorway()

   {

        if ( (studentInDoorway) || (firefighterInDoorway)

                (firefighterWaiting) )

             student.wait();

        studentInDoorway++;

   }

   Entry StudentExitDoorway()

   {

       studentInDoorway--;

       if (firefighterWaiting)

       {

          firefighter.signal();

          firefighter.signal();

       }

       else

         student.signal();

   }

}

**A Brinch Hanson Semantics Solution**

When we convert from Hoare sematnics to BH semantics, we have to convert ever signal() operation to a signalAndExit() operation. This means that any code after a signal has to be moved somewhere else.

In the case of the previous solution, this affects us in one critical place -- we can no longer signal twice in a row in order to let two firefighters into the doorway. Instead we convert the solution so that we signal the first firefighter. Firefighters must become more considerate -- if a firefighter enters the doorway and there is room for another, she/he signals another firefighter to enter. Of course, we need to be careful that this signal is the last thing that the firefighter does within the monitor -- as before, the operation is signalAndExit().

BH MONITOR doorway

{

   int firefighterInDoorway = 0;

   int firefighterWaiting = 0;

   int studentInDoorway = 0;

   condition student, firefighter;

   Entry FirefighterEnterDoorway()

   {

     firefighterWaiting++;

     if ( (studentInDoorway) || (firefighterInDoorway >= 2) )

         firefighter.wait();

     firefighterInDoorway++;

     firefighterWaiting--;

     if ((firefighterInDoorway < 2) && (firefighterWaiting) )

          firefighter.signalAndExit();

   }

   Entry FirefighterExitDoorway()

   {

     firefighterInDoorway--;

     if (firefighterWaiting())

         firefighter.signal();

     else

        if (!firefighterInDoorway)

             student.signalAndExit();

   }

   Entry StudentEnterDoorway()

   {

        if ( (studentInDoorway) || (firefighterInDoorway)

                (firefighterWaiting) )

             student.wait();

        studentInDoorway++;

   }

   Entry StudentExitDoorway()

   {

       studentInDoorway--;

       if (firefighterWaiting)

       {

          firefighter.signalAndExit();

       }

       else

         student.signalAndExit();

   }

}

**Monitors In Java**

The Java programming language provides support for monitors via *synchronized* methods within a class. We are assured that at most one synchronized method within a particular class can be active at any particular time, even in multi-threaded applications. Java does not require that all methods within a class be synchronized, so every method of the class is not necessarily part of the monitor -- only synchronized methods of a class are protected from concurrent execution. This is obviously an opportunity for a programmer to damage an appendage with a massive and rapidly moving projectile.

Java monitors are reasonably limited -- especially when contrasted with monitors using Hoare or Mesa semantics. In Java, there can only be one reason to wait (block) within the monitor, not multiple conditions. When a thread waits, it is made unrunnable. When it has been signaled to wake-up, it is made runnable -- it will next run whenever the scheduler happens to run it. Unlike BH monitors, a signal can occur anywhere in the code. Unlike Hoare semantics, the signaling thread doesn't immediately yield to the signaled thread. Unlike all three, there can only be one reason to wait/signal. In this way, they offer simplified Mesa sematics.

To wait for a condition, a Java thread invokes *wait()*. To signal a waiting thread to tell it that it can run (the condition upon which it is waiting is satisfied), a Java thread invokes *notify()*. Notify is actually a funny name -- normally this operation is called *signal*.

**Monitor Examples in Java**

In class we walked through these examples in Java from Concurrent Programming: The Java Language by Stephen Hartley and published by Oxford Univerity Press in 1998:

The first example is a solution to the Bounded Buffer problem, also known as the Producer-Consumer Problem. This solution supports one producer thread and one consumer thread.

Please notice that the producer signals a waiting consumer if it fills the first slot in the buffer -- this is because the consumer might have blocked because there were no full buffers. The consumer follows a similar practice if it takes the last item in the buffer -- there could be a producer blocked waiting for an available slot in a buffer.

class BoundedBuffer { // designed for a single producer thread

// and a single consumer thread

private int numSlots = 0;

private double[] buffer = null;

private int putIn = 0, takeOut = 0;

private int count = 0;

public BoundedBuffer(int numSlots) {

if (numSlots <= 0) throw new IllegalArgumentException("numSlots<=0");

this.numSlots = numSlots;

buffer = new double[numSlots];

System.out.println("BoundedBuffer alive, numSlots=" + numSlots);

}

public synchronized void deposit(double value) {

while (count == numSlots)

try {

wait();

} catch (InterruptedException e) {

System.err.println("interrupted out of wait");

}

buffer[putIn] = value;

putIn = (putIn + 1) % numSlots;

count++; // wake up the consumer

if (count == 1) notify(); // since it might be waiting

System.out.println(" after deposit, count=" + count

+ ", putIn=" + putIn);

}

public synchronized double fetch() {

double value;

while (count == 0)

try {

wait();

} catch (InterruptedException e) {

System.err.println("interrupted out of wait");

}

value = buffer[takeOut];

takeOut = (takeOut + 1) % numSlots;

count--; // wake up the producer

if (count == numSlots-1) notify(); // since it might be waiting

System.out.println(" after fetch, count=" + count

+ ", takeOut=" + takeOut);

return value;

}

}

**Fair Reader-Writer Solution**

What follows is a fair solution to the reader-writer problem -- it allows the starvation of neither the producer, nor the consumer. The only tricky part of this code is realizing that starvation of writers by readers is avoided by yielding to earlier requests.

class Database extends MyObject {

private int numReaders = 0;

private int numWriters = 0;

private int numWaitingReaders = 0;

private int numWaitingWriters = 0;

private boolean okToWrite = true;

private long startWaitingReadersTime = 0;

public Database() { super("rwDB"); }

public synchronized void startRead(int i) {

long readerArrivalTime = 0;

if (numWaitingWriters > 0 || numWriters > 0) {

numWaitingReaders++;

readerArrivalTime = age();

while (readerArrivalTime >= startWaitingReadersTime)

try {wait();} catch (InterruptedException e) {}

numWaitingReaders--;

}

numReaders++;

}

public synchronized void endRead(int i) {

numReaders--;

okToWrite = numReaders == 0;

if (okToWrite) notifyAll();

}

public synchronized void startWrite(int i) {

if (numReaders > 0 || numWriters > 0) {

numWaitingWriters++;

okToWrite = false;

while (!okToWrite)

try {wait();} catch (InterruptedException e) {}

numWaitingWriters--;

}

okToWrite = false;

numWriters++;

}

public synchronized void endWrite(int i) {

numWriters--; // ASSERT(numWriters==0)

okToWrite = numWaitingReaders == 0;

startWaitingReadersTime = age();

notifyAll();

}

}