Hoare Semantics for Condition Variables
(Ideas on implementing project 4, task 4)

Harry H. Porter III
Portland State University

May 9, 2006

This note sketches out some ideas on how to complete Project 4, task 4, which asks you to implement "Hoare Semantics" for condition variables.

Currently, "Mesa Semantics" are implemented. Recall that the current implementation of monitors works as follows:

(1) Each monitor is implemented by a class.

(2) There is a **Mutex** lock associated with each monitor class, which we can call the "monitor lock".

(3) Some methods are "entry methods". At the beginning of each entry method, the programmer must remember to execute:

```
monitorLock.Lock()
```

(4) Just before returning from an entry method, the programmer must remember to execute:

```
monitorLock.Unlock()
```

(5) The monitor may also have several "condition variables". There is a class called **Condition**, with the following methods: **Init**, **Signal**, **Wait** and **Broadcast**.

(6) The Condition class is implemented with a list called **waitingThreads**.

(7) The **Wait** operation is implemented by the following steps:

```
disable interrupts
monitorLock.Unlock ()
waitingThreads.AddToEnd (currentThread)
currentThread.Sleep ()
monitorLock.Lock ()
re-enable interrupts
```

(8) The **Signal** operation is implemented by the following steps:

```
disable interrupts
get the next thread from the "waitingThreads" list
put it on the ready list, thereby waking it up
re-enable interrupts
```

As an example, assume thread "**A**" executes a **Wait**. Later, thread "**B** " executes a **Signal**.

When thread **A** executes **Wait**, it will first release the monitor so that other threads can enter the "entry methods". Then **A** goes to sleep, waiting until a **Signal** operation is executed by some other thread. Once awake, thread **A** must re-acquire the monitor lock. Note that thread **A** will possibly have to go to sleep a second time, while waiting for the monitor lock. There may be other threads, such as "**C**", already waiting for the monitor lock. If so, then once the monitor becomes free, **C** will be awakened first and will enter the monitor before **A**.

Our goal is to implement Hoare Semantics, which will ensure that threads like **C** will be forced to wait and thread **A** will be effectively moved to the head of the line and will get back into the monitor before **C**.

To implement Hoare semantics, let us redefine how we implement monitors and condition variables. Each monitor will still be implemented with a class and there will be a class to implement condition variables. However, we will slightly modify the code that must be executed on exit from a entry method and we will redefine the **Condition** class.

(1) Each monitor is implemented by a class, as before.

(2) There is a **Mutex** lock associated with each monitor class, which we can call the "monitor lock". In addition, there is also an integer count, called "**nextCount**" and a semaphore called "**nextSem**". Both the count and the semaphore must be initialized to 0.

(3) As before, at the beginning of each entry method, the programmer must remember to execute:

```
monitorLock.Lock()
```

(4) Just before returning from an entry method, the programmer must remember to execute this code:

```
if nextCount > 0
  nextSem.Signal()
else
  monitorLock.Unlock()
endIf
```

At any time there may be normal threads (like **C**) waiting to enter the monitor. As before, these threads are waiting to acquire the **monitorLock**. But there may also be other threads which have higher priority and which must be let into the monitor sooner. These threads are waiting on **nextSem** and **nextCount** tells how many such high-priority threads there are waiting.

Whenever a thread leaves the monitor, it must wake up a high priority thread first, if there are any. That's what the above code does.

(5) Each "condition variable" is represented as an instance of a class, which we might call "**HoareCondition**" to prevent confusion with the existing **Condition** class. [Remember that you must not change any of the existing classes, or else some of the kernel code we are distributing may break. Since we need to change the implementation of **Signal** and **Wait**, we will therefore create a new class, which we will call **HoareCondition**.] This new class will have methods called **Signal** and **Wait**.

(6) The **HoareCondition** class will contain an integer variable called "**cnt**" and a semaphore called "**sem**". These must be initialized to 0.

(7) The "**Wait**" operation is implemented by the following steps:

```
cnt = cnt + 1
if nextCount > 0
  nextSem.Signal ()
else
  monitorLock.Unlock ()
endIf
sem.Wait ()
cnt = cnt - 1
```

(8) The "**Signal**" operation is implemented by the following steps:

```
if cnt > 0
  nextCount = nextCount + 1
  sem.Signal ()
  nextSem.Wait ()
  nextCount = nextCount - 1
endIf
```

In the implementation we gave you of **Condition**, the implementations of **Signal** and **Wait** explicitly disabled interrupts and manipulated the ready list. In the implementation we are proposing here, we are working at a slightly higher level, by using semaphores.

Let's look at an example. As before, assume thread **A** gets into the monitor first and executes a **Wait**. Then assume **B** gets into the monitor and executes a **Signal**. Here is the sequence of what happens:

A enters the monitor
    A acquires the "monitorLock" and proceeds
A executes a **Wait** on a **HoareCondition** variable called "**X**"
    X.cnt is incremented 1
    Since **nextCount** is zero, there is no high-priority thread waiting
    The **monitorLock** is released
    Thread **A** waits on **X.sem**
Now **B** enters the monitor
    B acquires the **monitorLock** (since it is free) and proceeds
B executes **Signal** on **X**, the **HoareCondition** upon which **A** is waiting
    X.cnt is 1 so the following statements are executed...
    nextCount is incremented to 1
X.sem is signalled, so now Thread **A** is ready and possibly running.
    [Perhaps **A** will run immediately, perhaps **B** will continue, or perhaps both **A** and **B** are running simultaneously on separate CPUs.]
    Thread **B** waits on **nextSem**
    [Since **B** goes to sleep immediately, it doesn't matter whether **A** or **B** actually gets the CPU first.]
A wakes up from waiting on **X.sem**
    X.cnt is decremented back to 0

Thread **A** proceeds.

Note that no intervening thread **C** could have snuck in, since the first thing **C** will do is try to lock the monitor. However, nowhere in the above sequence after **B** is in the monitor, is the **monitorLock** ever released. Also notice that **B** acquired the **monitorLock**, but at the end, **A** is executing in the monitor. Also notice that **B** is suspended on the monitor's **nextSem**; essentially **B** has become a high-priority thread which must be given precedence whenever the monitor becomes free.

Now let's assume that **A** is ready to leave the monitor. Upon returning from the monitor entry method, the following sequence will occur...

A will check the **count** variable and will find that it is 1.
A will signal the **nextSem** semaphore and wake up **B**.
A will then return, leaving the monitor.
B, which was waiting on **nextSem**, will wake up.
[Note that the **monitorLock** has not been released, so no other waiting threads, such as **C**, will be allowed to sneak in to the monitor. After **A** exits, **B** will resume execution.]
B will then decrement **nextCount** back to 0.

From a higher-level perspective, when thread **B** invoked the **Signal** operation, **B** went to sleep immediately, then **A** woke up and ran to completion, and finally **B** was re-awakened.

Later, when **B** is ready to leave the monitor, it will check **nextCount** and see that it is 0. Finally **B** will release the **monitorLock** and allow threads like **C** to have a chance.

To recap, the semantics implemented here will do the following:

(1) When thread **B** executes a **Signal**, it will go to sleep.
(2) Thread **A**, which has previously executed a **Wait**, will wake up and run.
(3) When thread **A** is done and leaves the monitor, thread **B** will wake up and resume.
(4) No other threads, like **C**, will be allowed in the monitor until **B** finally returns.

Let's look at another scenario. As before, imagine that **A** executes a **Wait**, then **B** enters the monitor. But this time, assume **B** also executes a **Wait** on **X**.

So now, **X.cnt** will be incremented to 2 and there will be two threads waiting on **X.sem**. The **monitorLock** will be released. Now assume thread **C** comes along and enters the monitor. Then thread **C** executes a **Signal**.

This will wake up **A**, which has been waiting the longest. **C** will then go to sleep, by waiting on **nextSem**. Let's look at these three cases, in turn:

Case 1: Thread **A** executes a **Signal**.

Case 2: Thread **A** exits; Thread **C** resumes and executes a **Signal**.

Case 3: Thread **A** exits; Thread **C** resumes and exits. Thread **B** continues to wait.

In all cases, we start with this scenario:

A executed a **Wait**.
B executed a **Wait**.
C executed a **Signal**.
C is waiting on **NextSem**.
B is waiting on **X.sem**.
The **monitorLock** is locked (by **C**).
A is running.

First look at case 1: Thread **A** executes a **Signal**. In this case, **A** will wake up **B**. Then **A** will wait on **nextSem**, after **C**. **B** will run to completion and, upon leaving the monitor, will wake up **C**. When **C** exits the monitor, it will wake up **A**, which will release the monitor lock.

With this scenario, we have this sequence:

A executes a **Wait**.
B executes a **Wait**.
C executes a **Signal** and sleeps.
A wakes up.
A executes a **Signal** and sleeps.
B wakes up and exits the monitor.
C wakes up and exits the monitor.
A wakes up and exits the monitor.

[Notice that the **monitorLock** will be released by a thread other than the one which acquired it. In this example, **C** acquired the monitor lock, but it was thread **A** that finally released it. The implementation we have provided for class **Mutex** considers this a bug. It will check for it and cause an error. Therefore, in order to make this scheme work, we will need to re-implement **Mutex** and take out this error checking. Alternately, we could simply use a **Semaphore** for the "**monitorLock**". We'd have to initialize it with one excess signal and then use **Wait** to lock the **monitorLock** at the top of every entry method.]

Also notice what happened from **C**'s point of view. Thread **C** signaled thread **A**. Since this is Hoare semantics, no threads intervened: upon the signal, **C** went to sleep immediately and thread **A** woke up and ran, with no intervening threads. However, after that, it wasn't so simple as "When **A** finishes, **C** will resume after its call to Signal." Instead, **A** decided to execute a **Signal**. This allowed **B** to run and ended up allowing **C** to wake up and resume execution before **A** had actually completed.

Next look at case 2: Thread **A** exits; thread **C** resumes and executes a **Signal**. In this case, we get the following sequence of events:

A executes a **Wait** and sleeps.
B executes a **Wait** and sleeps.
C executes a **Signal** and sleeps.
A wakes up.
A leaves the monitor.

C wakes up.
C executes another **Signal** and sleeps again.
B wakes up.
B leaves the monitor.
C wakes up.
C leaves the monitor.

Finally look at case 3: Thread **A** exits; thread **C** resumes and exits; thread **B** continues to wait. In this case, we get the following sequence:

A executes a **Wait** and sleeps.
B executes a **Wait** and sleeps.
C executes a **Signal** and sleeps.
A wakes up.
A leaves the monitor.
C wakes up.
C leaves the monitor.
B is left waiting.