D

Monitors (Deprecated)

Around the time concurrent programming was becoming a big deal, object-oriented programming was also gaining ground. Not surprisingly, people started to think about ways to merge synchronization into a more structured programming environment.

One such approach that emerged was the **monitor**. First described by Per Brinch Hansen [BH73] and later refined by Tony Hoare [H74], the idea behind a monitor is quite simple. Consider the following pretend monitor written in C++ notation:

```
monitor class account {
private:
    int balance = 0;

public:
    void deposit(int amount) {
        balance = balance + amount;
    }
    void withdraw(int amount) {
        balance = balance - amount;
    }
};
```

Figure D.1: **A Pretend Monitor Class**

Note: this is a "pretend" class because C++ does not support monitors, and hence the **monitor** keyword does not exist. However, Java does support monitors, with what are called **synchronized** methods. Below, we will examine both how to make something quite like a monitor in C/C++, as well as how to use Java synchronized methods.

In this example, you may notice we have our old friend the account and some routines to deposit and withdraw an amount from the balance. As you also may notice, these are **critical sections**; if they are called by multiple threads concurrently, you have a race condition and the potential for an incorrect outcome.

In a monitor class, you don't get into trouble, though, because the monitor guarantees that **only one thread can be active within the monitor at a time**. Thus, our above example is a perfectly safe and working

piece of code; multiple threads can call deposit() or withdraw() and know that mutual exclusion is preserved.

How does the monitor do this? Simple: with a lock. Whenever a thread tries to call a monitor routine, it implicitly tries to acquire the monitor lock. If it succeeds, then it will be able to call into the routine and run the method's code. If it does not, it will block until the thread that is in the monitor finishes what it is doing. Thus, if we wrote a C++ class that looked like the following, it would accomplish the exact same goal as the monitor class above:

```
class account {
private:
    int balance = 0;
    pthread_mutex_t monitor;

public:
    void deposit(int amount) {
        pthread_mutex_lock(&monitor);
        balance = balance + amount;
        pthread_mutex_unlock(&monitor);
    }
    void withdraw(int amount) {
        pthread_mutex_lock(&monitor);
        balance = balance - amount;
        pthread_mutex_unlock(&monitor);
    }
};
```

Figure D.2: **A C++ Class that acts like a Monitor**

Thus, as you can see from this example, the monitor isn't doing too much for you automatically. Basically, it is just acquiring a lock and releasing it. By doing so, we achieve what the monitor requires: only one thread will be active within deposit() or withdraw(), as desired.

D.1 Why Bother with Monitors?

You might wonder why monitors were invented at all, instead of just using explicit locking. At the time, object-oriented programming was just coming into fashion. Thus, the idea was to gracefully blend some of the key concepts in concurrent programming with some of the basic approaches of object orientation. Nothing more than that.

D.2 Do We Get More Than Automatic Locking?

Back to business. As we know from our discussion of semaphores, just having locks is not quite enough; for example, to implement the producer/consumer solution, we previously used semaphores to both put threads to sleep when waiting for a condition to change (e.g., a producer waiting for a buffer to be emptied), as well as to wake up a thread when a particular condition has changed (e.g., a consumer signaling that it has indeed emptied a buffer).

```
monitor class BoundedBuffer {
  private:
    int buffer[MAX];
    int fill, use;
    int fullEntries = 0;
    cond_t empty;
    cond_t full;

  public:
    void produce(int element) {
        if (fullEntries == MAX)     // line P0
            wait(&empty);           // line P1
        buffer[fill] = element;     // line P2
        fill = (fill + 1) % MAX;    // line P3
        fullEntries++;              // line P4
        signal(&full);              // line P5
    }

    int consume() {
        if (fullEntries == 0)       // line C0
            wait(&full);            // line C1
        int tmp = buffer[use];      // line C2
        use = (use + 1) % MAX;      // line C3
        fullEntries--;              // line C4
        signal(&empty);             // line C5
        return tmp;                 // line C6
    }
}
```

Figure D.3: **Producer/Consumer with Monitors and Hoare Semantics**

Monitors support such functionality through an explicit construct known as a **condition variable**. Let's take a look at the producer/consumer solution, here written with monitors and condition variables.

In this monitor class, we have two routines, produce() and consume(). A producer thread would repeatedly call produce() to put data into the bounded buffer, while a consumer() would repeatedly call consume(). The example is a modern paraphrase of Hoare's solution [H74].

You should notice some similarities between this code and the semaphore-based solution in the previous note. One major difference is how condition variables must be used in concert with an explicit **state variable**; in this case, the integer `fullEntries` determines whether a producer or consumer must wait, depending on its state. Semaphores, in contrast, have an internal numeric value which serves this same purpose. Thus, condition variables must be paired with some kind of external state value in order to achieve the same end.

The most important aspect of this code, however, is the use of the two condition variables, empty and full, and the respective `wait()` and `signal()` calls that employ them. These operations do exactly what you might think: wait() blocks the calling thread on a given condition; `signal()` wakes one waiting thread that is waiting on the condition.

However, there are some subtleties in how these calls operate; understanding the semantics of these calls is critically important to understand-

ing why this code works. In what researchers in operating systems call **Hoare semantics** (yes, a somewhat unfortunate name), the `signal()` immediately wakes one waiting thread and runs it; thus, the monitor lock, which is implicitly held by the running thread, immediately is transferred to the woken thread which then runs until it either blocks or exits the monitor. Note that there may be more than one thread waiting; `signal()` only wakes one waiting thread and runs it, while the others must wait for a subsequent signal.

A simple example will help us understand this code better. Imagine there are two threads, one a producer and the other a consumer. The consumer gets to run first, and calls `consume()`, only to find that fullEntries = 0 `(C0)`, as there is nothing in the buffer yet. Thus, it calls `wait(&full)` `(C1)`, and waits for a buffer to be filled. The producer then runs, finds it doesn't have to wait `(P0)`, puts an element into the buffer `(P2)`, increments the fill index `(P3)` and the fullEntries count `(P4)`, and calls `signal(&full)` `(P5)`. In Hoare semantics, the producer does not continue running after the signal; rather, the signal immediately transfers control to the waiting consumer, which returns from `wait()` `(C1)` and immediately consumes the element produced by the producer `(C2)` and so on. Only after the consumer returns will the producer get to run again and return from the produce() routine.

D.3 Where Theory Meets Practice

Tony Hoare, who wrote the solution above and came up with the exact semantics for `signal()` and `wait()`, was a theoretician. Clearly a smart guy, too; he came up with quicksort after all [H61]. However, the semantics of signaling and waiting, as it turns out, were not ideal for a real implementation. As the old saying goes, in theory, there is no difference between theory and practice, but in practice, there is.

OLD SAYING: THEORY VS. PRACTICE

The old saying is "in theory, there is no difference between theory and practice, but in practice, there is." Of course, only practitioners tell you this; a theory person could undoubtedly prove that it is not true.

A few years later, Butler Lampson and David Redell of Xerox PARC were building a concurrent language known as **Mesa**, and decided to use monitors as their basic concurrency primitive [LR80]. They were well-known systems researchers, and they soon found that Hoare semantics, while more amenable to proofs, were hard to realize in a real system (there are a lot of reasons for this, perhaps too many to go through here).

In particular, to build a working monitor implementation, Lampson and Redell decided to change the meaning of signal() in a subtle but critical way. The signal() routine now was just considered a **hint** [L83]; it

would move a single waiting thread from the blocked state to a runnable state, but it would not run it immediately. Rather, the signaling thread would retain control until it exited the monitor and was descheduled.

D.4 Oh Oh, A Race

Given these new **Mesa semantics**, let us again reexamine the code above. Imagine again a consumer (consumer 1) who enters the the monitor and finds the buffer empty and thus waits `(C1)`. Now the producer comes along and fills the buffer and signals that a buffer has been filled, moving the waiting consumer from blocked on the full condition variable to ready. The producer keeps running for a while, and eventually gives up the CPU.

But Houston, we have a problem. Can you see it? Imagine a different consumer (consumer 2) now calls into the consume() routine; it will find a full buffer, consume it, and return, setting fullEntries to 0 in the meanwhile. Can you see the problem yet? Well, here it comes. Our old friend consumer 1 now finally gets to run, and returns from wait(), expecting a buffer to be full `(C1...)`; unfortunately, this is no longer true, as consumer 2 snuck in and consumed the buffer before consumer 1 had a chance to consume it. Thus, the code doesn't work, because in the time between the signal() by the producer and the return from wait() by consumer 1, the condition has changed. This timeline illustrates the problem:

```
      Producer                Consumer1                Consumer2
                        C0 (fullEntries=0)
                        C1 (Consumer 1: blocked)
P0 (fullEntries=0)
P2
P3
P4 (fullEntries=1)
P5 (Consumer1: ready)
                                                 C0 (fullEntries=1)
                                                 C2
                                                 C3
                                                 C4 (fullEntries=0)
                                                 C5
                                                 C6
                        C2 (using a buffer,
                            fullEntries=0!)
```

Figure D.4: **Why the Code doesn't work with Hoare Semantics**

Fortunately, the switch from Hoare semantics to Mesa semantics requires only a small change by the programmer to realize a working solution. Specifically, when woken, a thread should *recheck* the condition it was waiting on; because signal() is only a hint, it is possible that the condition has changed (even multiple times) and thus may not be in the desired state when the waiting thread runs. In our example, two lines of code must change, lines P0 and C0:

```
public:
  void produce(int element) {
      while (fullEntries == MAX) // line P0 (CHANGED IF->WHILE)
          wait(&empty);          // line P1
      buffer[fill] = element;    // line P2
      fill = (fill + 1) % MAX;   // line P3
      fullEntries++;             // line P4
      signal(&full);             // line P5
  }

  int consume() {
      while (fullEntries == 0)   // line C0 (CHANGED IF->WHILE)
          wait(&full);           // line C1
      int tmp = buffer[use];     // line C2
      use = (use + 1) % MAX;     // line C3
      fullEntries--;             // line C4
      signal(&empty);            // line C5
      return tmp;                // line C6
  }
```

Figure D.5: **Producer/Consumer with Monitors and Mesa Semantics**

Not too hard after all. Because of the ease of this implementation, virtually any system today that uses condition variables with signaling and waiting uses Mesa semantics. Thus, if you remember nothing else at all from this class, you can just remember: **always recheck the condition after being woken!** Put in even simpler terms, **use while loops** and not if statements when checking conditions. Note that this is always correct, even if somehow you are running on a system with Hoare semantics; in that case, you would just needlessly retest the condition an extra time.

D.5 Peeking Under The Hood A Bit

To understand a bit better why Mesa semantics are easier to implement, let's understand a little more about the implementation of Mesa monitors. In their work [LR80], Lampson and Redell describe three different types of queues that a thread can be a part of at a given time: the **ready** queue, a **monitor lock** queue, and a **condition variable** queue. Note that a program might have multiple monitor classes and multiple condition variable instances; there is a queue per instance of said items.

With a single bounded buffer monitor, we thus have four queues to consider: the ready queue, a single monitor queue, and two condition variable queues (one for the full condition and one for the empty). To better understand how a thread library manages these queues, what we will do is show how a thread transitions through these queues in the producer/consumer example.

In this example, we walk through a case where a consumer might be woken up but find that there is nothing to consume. Let us consider the following timeline. On the left are two consumers (Con1 and Con2) and a producer (Prod) and which line of code they are executing; on the right is the state of each of the four queues we are following for this example:

```
 t | Con1 Con2 Prod | Mon | Empty | Full | FE | Comment
-------------------------------------------------------------------------
 0  C0                                      0
 1  C1                                Con1  0   Con1 waiting on full
 2 <Context switch>                   Con1  0   switch: Con1 to Prod
 3            P0                      Con1  0
 4            P2                      Con1  0   Prod doesn't wait (FE=0)
 5            P3                      Con1  0
 6            P4                      Con1  1   Prod updates fullEntries
 7            P5                            1   Prod signals: Con1 now ready
 8 <Context switch>                         1   switch: Prod to Con2
 9        C0                                1   switch to Con2
10        C2                                1   Con2 doesn't wait (FE=1)
11        C3                                1
12        C4                                0   Con2 changes fullEntries
13        C5                                0   Con2 signals empty (no waiter)
14        C6                                0   Con2 done
15 <Context switch>                         0   switch: Con2 to Con1
16   C0                                     0   recheck fullEntries: 0!
17   C1                              Con1   0   wait on full again
```

Figure D.6: **Tracing Queues during a Producer/Consumer Run**

the ready queue of runnable processes, the monitor lock queue called Monitor, and the empty and full condition variable queues. We also track time (t), the thread that is running (square brackets around the thread on the ready queue that is running), and the value of fullEntries (FE).

As you can see from the timeline, consumer 2 (Con2) sneaks in and consumes the available data (t=9..14) before consumer 1 (Con1), who was waiting on the full condition to be signaled (since t=1), gets a chance to do so. However, Con1 does get woken by the producer's signal (t=7), and thus runs again even though the buffer is empty by the time it does so. If Con1 didn't recheck the state variable fullEntries (t=16), it would have erroneously tried to consume data when no data was present to consume. Thus, this natural implementation is exactly what leads us to Mesa semantics (and not Hoare).

D.6 Other Uses Of Monitors

In their paper on Mesa, Lampson and Redell also point out a few places where a different kind of signaling is needed. For example, consider the following memory allocator (Figure D.7).

Many details are left out of this example, in order to allow us to focus on the conditions for waking and signaling. It turns out the signal/wait code above does not quite work; can you see why?

Imagine two threads call allocate. The first calls allocate(20) and the second allocate(10). No memory is available, and thus both threads call wait() and block. Some time later, a different thread comes along and calls free(p, 15), and thus frees up 15 bytes of memory. It then signals that it has done so. Unfortunately, it wakes the thread waiting for 20 bytes; that thread rechecks the condition, finds that only 15 bytes are available, and

```
monitor class allocator {
    int available; // how much memory is available?
    cond_t c;

    void *allocate(int size) {
        while (size > available)
            wait(&c);
        available -= size;
        // and then do whatever the allocator should do
        // and return a chunk of memory
    }

    void free(void *pointer, int size) {
        // free up some memory
        available += size;
        signal(&c);
    }
};
```

Figure D.7: **A Simple Memory Allocator**

calls wait() again. The thread that could have benefited from the free of 15 bytes, i.e., the thread that called allocate(10), is not woken.

Lampson and Redell suggest a simple solution to this problem. Instead of a signal() which wakes a single waiting thread, they employ a **broadcast()** which wakes *all* waiting threads. Thus, all threads are woken up, and in the example above, the thread waiting for 10 bytes will find 15 available and succeed in its allocation. In this way,

In Mesa semantics, using a broadcast() is *always* correct, as all threads should recheck the condition of interest upon waking anyhow. However, it may be a performance problem, and thus should only be used when needed. In this example, a broadcast() might wake hundreds of waiting threads, only to have one successfully continue while the rest immediately block again; this problem, sometimes known as a **thundering herd**, is costly, due to all the extra context switches that occur.

D.7 Using Monitors To Implement Semaphores

You can probably see a lot of similarities between monitors and semaphores. Not surprisingly, you can use one to implement the other. Here, we show how you might implement a semaphore class using a monitor (Figure D.8).

As you can see, wait() simply waits for the value of the semaphore to be greater than 0, and then decrements its value, whereas post() increments the value and wakes one waiting thread (if there is one). It's as simple as that.

To use this class as a binary semaphore (i.e., a lock), you just initialize the semaphore to 1, and then put wait()/post() pairs around critical sections. And thus we have shown that monitors can be used to implement semaphores.

```
monitor class Semaphore {
    int s; // value of the semaphore
    Semaphore(int value) {
        s = value;
    }
    void wait() {
        while (s <= 0)
            wait();
        s--;
    }
    void post() {
        s++;
        signal();
    }
};
```

Figure D.8: **Implementing a Semaphore with a Monitor**

D.8 Monitors in the Real World

We already mentioned above that we were using "pretend" monitors; C++ has no such concept. We now show how to make a monitor-like C++ class, and how Java uses synchronized methods to achieve a similar end.

A C++ Monitor of Sorts

Here is the producer/consumer code written in C++ with locks and condition variables (Figure D.9). You can see in this code example that there is little difference between the pretend monitor code and the working C++ class we have above. Of course, one obvious difference is the explicit use of a lock "monitor". More subtle is the switch to the POSIX standard `pthread_cond_signal()` and `pthread_cond_wait()` calls. In particular, notice that when calling `pthread_cond_wait()`, one also passes in the lock that is held at the time of waiting. The lock is needed inside `pthread_cond_wait()` because it must be released when this thread is put to sleep and re-acquired before it returns to the caller (the same behavior as within a monitor but again with explicit locks).

A Java Monitor

Interestingly, the designers of Java decided to use monitors as they thought they were a graceful way to add synchronization primitives into a language. To use them, you just use add the keyword **synchronized** to the method or set of methods that you wish to use as a monitor (here is an example from Sun's own documentation site [S12a,S12b]):

This code does exactly what you think it should: provide a counter that is thread safe. Because only one thread is allowed into the monitor at a time, only one thread can update the value of "c", and thus a race condition is averted.

```
class BoundedBuffer {
  private:
    int buffer[MAX];
    int fill, use;
    int fullEntries;
    pthread_mutex_t monitor; // monitor lock
    pthread_cond_t empty;
    pthread_cond_t full;

  public:
    BoundedBuffer() {
        use = fill = fullEntries = 0;
    }
    void produce(int element) {
        pthread_mutex_lock(&monitor);
        while (fullEntries == MAX)
            pthread_cond_wait(&empty, &monitor);
        buffer[fill] = element;
        fill = (fill + 1) % MAX;
        fullEntries++;
        pthread_cond_signal(&full);
        pthread_mutex_unlock(&monitor);
    }

    int consume() {
        pthread_mutex_lock(&monitor);
        while (fullEntries == 0)
            pthread_cond_wait(&full, &monitor);
        int tmp = buffer[use];
        use = (use + 1) % MAX;
        fullEntries--;
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&monitor);
        return tmp;
    }
}
```

Figure D.9: **C++ Producer/Consumer with a "Monitor"**

Java and the Single Condition Variable

In the original version of Java, a condition variable was also supplied with each synchronized class. To use it, you would call either **wait()** or **notify()** (sometimes the term notify is used instead of signal, but they mean the same thing). Oddly enough, in this original implementation, there was no way to have two (or more) condition variables. You may have noticed in the producer/consumer solution, we always use two: one for signaling a buffer has been emptied, and another for signaling that a buffer has been filled.

To understand the limitations of only providing a single condition variable, let's imagine the producer/consumer solution with only a single condition variable. Imagine two consumers run first, and both get stuck waiting. Then, a producer runs, fills a single buffer, wakes a single

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Figure D.10: **A Simple Java Class with Synchronized Methods**

consumer, and then tries to fill again but finds the buffer full (MAX=1). Thus, we have a producer waiting for an empty buffer, a consumer waiting for a full buffer, and a consumer who had been waiting about to run because it has been woken.

The consumer then runs and consumes the buffer. When it calls notify(), though, it wakes a single thread that is waiting on the condition. Because there is only a single condition variable, the consumer might wake the waiting **consumer,** instead of the waiting producer. Thus, the solution does not work.

To remedy this problem, one can again use the broadcast solution. In Java, one calls **notifyAll()** to wake all waiting threads. In this case, the consumer would wake a producer and a consumer, but the consumer would find that fullEntries is equal to 0 and go back to sleep, while the producer would continue. As usual, waking all waiters can lead to the thundering herd problem.

Because of this deficiency, Java later added an explicit Condition class, thus allowing for a more efficient solution to this and other similar concurrency problems.

D.9 Summary

We have seen the introduction of monitors, a structuring concept developed by Brinch Hansen and and subsequently Hoare in the early seventies. When running inside the monitor, a thread implicitly holds a monitor lock, and thus prevents other threads from entering the monitor, allowing the ready construction of mutual exclusion.

We also have seen the introduction of explicit condition variables, which allow threads to signal() and wait() much like we saw with semaphores in the previous note. The semantics of signal() and wait() are critical; because all modern systems implement **Mesa** semantics, a recheck of the condition that the thread went to sleep on is required for correct execution. Thus, signal() is just a **hint** that something has changed; it is the responsibility of the woken thread to make sure the conditions are right

for its continued execution.

Finally, because C++ has no monitor support, we saw how to emulate monitors with explicit pthread locks and condition variables. We also saw how Java supports monitors with its synchronized routines, and some of the limitations of only providing a single condition variable in such an environment.

References

[BH73] "Operating System Principles"
Per Brinch Hansen, Prentice-Hall, 1973
Available: http://portal.acm.org/citation.cfm?id=540365
One of the first books on operating systems; certainly ahead of its time. Introduced monitors as a concurrency primitive.

[H74] "Monitors: An Operating System Structuring Concept"
C.A.R. Hoare
CACM, Volume 17:10, pages 549–557, October 1974
An early reference to monitors; however, Brinch Hansen probably was the true inventor.

[H61] "Quicksort: Algorithm 64"
C.A.R. Hoare
CACM, Volume 4:7, July 1961
The famous quicksort algorithm.

[LR80] "Experience with Processes and Monitors in Mesa"
B.W. Lampson and D.R. Redell
CACM, Volume 23:2, pages 105–117, February 1980
An early and important paper highlighting the differences between theory and practice.

[L83] "Hints for Computer Systems Design"
Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
Lampson, a famous systems researcher, loved using hints in the design of computer systems. A hint is something that is often correct but can be wrong; in this use, a signal() is telling a waiting thread that it changed the condition that the waiter was waiting on, but not to trust that the condition will be in the desired state when the waiting thread wakes up. In this paper about hints for designing systems, one of Lampson's general hints is that you should use hints. It is not as confusing as it sounds.

[S12a] "Synchronized Methods"
Sun documentation
http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html

[S12b] "Condition Interface"
Sun documentation
http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/Condition.html