# Implementing Condition Variables with Semaphores

*Andrew D. Birrell*
*Microsoft Research—Silicon Valley*
*January 2003*

## Introduction

All of today's popular systems for programming with multiple threads use designs based around three data types:

- "Thread", with operations Fork and Join
- "Lock" with operations Acquire and Release
- "Condition Variable" with operations Wait, Signal and Broadcast

This is true of PThreads, Java, and C#. It's also true of their predecessors, Modula-3, Modula-2+ and Mesa.

In 1984 a group of us at DEC SRC were implementing a new multi-processor operating system: the system was Taos, the machine was Firefly and the language, which we created, was Modula-2+. As part of that effort we implemented these threads primitives. In doing so we observed that the semantics of Acquire and Release were identical to those of a binary semaphore*. Also, the semantics of Wait and Signal are tantalizingly similar to those of a binary semaphore. So we thought we could provide a single abstraction in the kernel, and present it as locks and condition variables in the language support layer. This paper is the tale of what happened then.

The system we were building used what would nowadays be called a micro-kernel architecture (the term hadn't been invented then). The lead programmer for the kernel was Roy Levin and I was doing the user-mode thread support code (and the RPC system). We were ably assisted in building the threads facility by a large and highly qualified cast of other SRC employees, consultants, and passers-by, including Butler Lampson, Paul Rovner, Roger Needham, Jerry Saltzer and Dave Clark.

## Ground Rules

I'm not going to give formal semantics for the threads operations here. You can read the ones we wrote for Modula-2+ [1], or you can read the reasonably good description in chapter 17 of the Java Language Specification [3] (ignoring the stuff about re-entrant mutexes). It's worth reading those specifications sometime, but the following summary should be enough for appreciating this paper.

* Modula-2+ did not support the notion of re-entrant mutexes. If a thread holding m tried to acquire m again, the thread would deadlock. This still seems like a good idea. Implementing locks with semaphores is messier if for some reason you want to allow re-entrant locking, but it's still not difficult

- A condition variable c is associated with a specific lock m. Calling c.Wait() enqueues the current thread on c (suspending its execution) and unlocks m, as a single atomic action. When this thread resumes execution it re-locks m.

- c.Signal() examines c, and if there is at least one thread enqueued on c then one such thread is dequeued and allowed to resume execution; this entire operation is a single atomic action.

- c.Broadcast() examines c and if there are any threads enqueued on c then all such threads are allowed to resume execution. Again, this entire operation is a single atomic action: the threads to be awoken are exactly those that had called c.Wait() before this call of c.Broadcast(). Of course, the awoken threads have to wait in line to acquire the lock m.

Note that these are the Mesa (and Modula, PThreads, Java and C#) semantics. Tony Hoare's original condition variable design [4] had the Signal operation transfer the lock to the thread being awoken and had no Broadcast.

See Dijkstra's 1967 paper [2] for a precise description of semaphore semantics. In summary:

- A semaphore sem has an integer state (sem.count) and two operations, "P" and "V".

- sem.P() suspends the current thread until sem.count > 0, then decrements sem.count and allows the thread to resume execution. The action of verifying that sem.count > 0 and decrementing it is atomic.

- sem.V() increments sem.count, atomically. For the special case of a binary semaphore the increment is omitted if sem.count is already 1.

It's quite easy to implement semaphores very efficiently using a hardware test-and-set instruction, or more modern interlocked memory accesses (such as the load-locked and store-conditional features of the MIPS and Alpha architectures, or the analogous features of modern Intel processors).

To give this historical tale a modern flavour, I'm going to use C# for the programming examples (Java would be almost identical). In reality the implementations for the Firefly were written in Modula-2+, and the actual data representation was somewhat different than given here. I'm also going to ignore exceptions completely, to avoid cluttering the code with try … finally statements.

## Getting Started

A semaphore is ideal for implementing a lock with the Modula or Mesa semantics. We represent the lock directly as a semaphore, with its integer restricted to the range [0..1], initially 1. The Acquire operation is exactly P and Release is exactly V:

```
class Lock {
        Semaphore sm;
        public Lock() {            // constructor
                sm = new Semaphore(); sm.count =1; sm.limit = 1;
        }
        public void Acquire() { sm.P(); }
        public void Release() { sm.V(); }
}
```

You can come quite close to implementing a condition variable in a similar way:

```
class CV {
        Semaphore s;
        Lock m;
        public CV(Lock m) {      // Constructor
                this.m = m;
                s = new Semaphore(); s.count = 0; s.limit = 1;
        }
        public void Wait() {       // Pre-condition: this thread holds "m"
                m.Release();
(1) —
                s.P();
                m.Acquire();
        }
        public void Signal() {
                s.V();
        }
}
```

Most of this is obvious. The condition variable is associated with a Lock m. Enqueueing a thread on a condition variable is implemented by the s.P() operation. The only issues occur in the area around (1). Recall that the semantics say that c.Wait should *atomically* release the lock and enqueue the thread on c, which this code blatantly doesn't do.

The critical case is where there is no thread currently enqueued on c, and some thread A has called c.Wait() and has reached (1), then thread B calls c.Signal(). This calls s.V() which sets s.count to 1. When thread A eventually gets around to calling s.P(), it finds that s.count is 1, and so decrements it and continues executing. This is the correct behaviour. The effect was christened the "wake-up waiting race" by Jerry Saltzer [5], and using a binary semaphore ensures that A will not get stranded enqueued incorrectly on s.

However, this does have a side-effect: if a thread calls c.Signal() when no thread is inside c.Wait(), then s.count will be left at 1. This mean that the next thread to call c.Wait() will just decrement s.count and drop through, which isn't really what the semantics said. Fortunately, we were experienced enough to notice this problem immediately. You can fix it by counting the calls of c.Wait() and the matching calls of c.Signal().The counter also gives us a plausible implementation of c.Broadcast.

Of course, you need a lock to protect this counter. For the purposes of the current description I'll use another semaphore x in each condition variable. In a real implementation you'd probably optimize to some form of spin-lock, perhaps combined with a private agreement with the thread scheduler. Java and C# avoid this extra lock by requiring that the caller of Signal or Broadcast holds c.m; we didn't want this restriction in Modula-2+.

```
class CV {
        Semaphore s, x;
        Lock m;
        int waiters = 0;
        public CV(Lock m) {      // Constructor
                this.m = m;
                s = new Semaphore(); s.count = 0; s.limit = 1;
                x = new Semaphore(); x.count = 1; x.limit = 1;
        }
        public void Wait() {      // Pre-condition: this thread holds "m"
                x.P(); {
                        waiters++;
                } x.V();
                m.Release();
(1) —
                s.P();
                m.Acquire();
        }
        public void Signal() {
                x.P(); {
                        if (waiters > 0) { waiters--; s.V(); }
                } x.V();
        }
        public void Broadcast() {
                x.P(); {
                        while (waiters > 0) { waiters--; s.V(); }
                } x.V();
        }
}
```

This looks pretty good and we were happy with it for several weeks. But really, it rates only as a "good try". It took us a while to notice.

## Fixing things up

The problem with the above implementation of condition variables again lies at position (1), and there are actually two bugs there.

The first one we noticed is that there might be arbitrarily many threads suspended inside c.Wait at (1). Although a call of c.Broadcast() would call s.V() the correct number of times, the fact that it's a binary semaphore means that s.count stops at 1. So all but one of the threads at (1) would end up stranded, enqueued on s. We noticed this one day when Dave Clark was visiting. The obvious fix is to declare that s is a general counting semaphore, with unbounded s.count. That ensures the correct number of threads will drop through in c.Wait

Unfortunately they might not be the correct threads. If 7 threads have called c.Wait and are all at (1) when c.Broadcast is called, we will call s.V() 7 times and bump s.count to 7. If the threads that are at (1) were to continue, all would be fine. But what if before that some other thread were to call c.Wait()? Then that thread would decrement s.count and drop through, and one of the 7 threads would end up enqueued on s. This most definitely violates the specified semantics. Notice that c.Signal has the same problem.

So our next attempt was to use some form of handshake to arrange that the correct threads drop through. We do this by introducing yet another semaphore h, a general counting semaphore. This lets the signaller block until the appropriate number of threads have got past the call of s.P() in Wait. The thread in c.Signal waits on h.P() until a thread has made a matching call of h.V() inside c.Wait().

```
        class CV {
                Semaphore s, x;
                Lock m;
                int waiters = 0;
                Semaphore h;
                public CV(Lock m) {      // Constructor
                        this.m = m;
                        s = new Semaphore(); s.count = 0; s.limit = 999999;
                        x = new Semaphore(); x.count = 1; x.limit = 1;
                        h = new Semaphore(); h.count = 0; h.limit = 999999;
                }
                public void Wait() {       // Pre-condition: this thread holds “m”
                        x.P(); {
                                waiters++;
                        } x.V();
                        m.Release();
        (1) —
                        s.P();
                        h.V();
                        m.Acquire();
                }
                public void Signal() {
                        x.P(); {
                                if (waiters > 0) { waiters--; s.V(); h.P(); }
                        } x.V();
                }
                public void Broadcast() {
                        x.P(); {
                                for (int i = 0; i < waiters; i++) s.V();
                                while (waiters > 0) { waiters--; h.P(); }
                        } x.V();
                }
        }
```

Now, by this time you're probably thinking that this implementation is getting a bit heavyweight. You're probably right. But it's worse than that.

I think that the above version of CV is formally correct, in that it implements the correct semantics. However, it has a fundamental performance problem: there are necessarily two context switches in each call of Signal, because the signalling thread must wait for the signalled thread to call h.V() before the signalling thread can continue. We noticed this, and worried about it. There are a lot of similar designs you can construct, but as far as we could tell in 1984 all of them either give the wrong answer or have unacceptable performance problems.

So eventually we gave up on the idea that we should build locks and condition variables out of semaphores. Roy took the semantics of condition variables and implemented them directly in the kernel. There it's not difficult to do them: we built the atomicity of Wait as part of the scheduler implementation, using the hardware test-and-set instructions to get atomicity with spin-locks, and building the requisite queues through the thread control blocks.

## The Sequel—NT and PThreads

Microsoft released Windows NT to the world in 1993. At SRC we observed that this was a high quality kernel running on widely available hardware, and we decided it would be good to port our Modula-3 development environment to NT. As part of this I volunteered to implement Modula-3 threads on top of the Win32 API provided by NT. On the face of it, this seemed like it should be easy. It turned out to be easy in the same way that building condition variables out of semaphores was easy.

Even in 1993 the Win32 API provided lots of potentially useful features for concurrent programming. There was a satisfactory design for multiple threads in an address space, and a lot of synchronization primitives (events, mutexes, semaphores and critical sections). Unfortunately, none of them was exactly what was needed for condition variables. In particular, there was no operation that atomically released some object and blocked on another one. I went through much the same sequence of bad solutions as we went through in 1984 (memories are short). In this case, though we couldn't give up and modify the kernel primitives. Fortunately, there is another solution, as follows.

You can indeed build condition variables out of semaphores, but the only way I know of that is correct and adequately efficient is to use an explicit queue. If I have an object for each thread, I can implement Wait by running a queue through the thread object, with the head being in the condition variable object. Here's an outline (to keep it simple, the queue in this outline is LIFO; it should of course be roughly FIFO, allowing for thread priorities).

```
class Thread {
        public static Semaphore x;      // Global lock; initially x.count = 1; x.limit = 1
        public Thread next = null;
        public Semaphore s = new Semaphore(); // Initially s.count = 0; s.limit = 1;
        public static Thread Self() { … }
}
```

```
class CV {
        Lock m;
        Thread waiters = null;
        public CV(Lock m) {      // Constructor
                this.m = m;
        }
        public void Wait() {      // Pre-condition: this thread holds “m”
                Thread self = Thread.Self();
                Thread.x.P(); {
                        self.next = waiters;
                        waiters = self;
                } Thread.x.V();
                m.Release();
                self.s.P();
(2) —           m.Acquire();
        }
        public void Signal() {
                Thread.x.P(); {
                        if (waiters != null) {
                                waiters.s.V();
                                waiters = waiters.next;
                        }
                } Thread.x.V();
        }
        public void Broadcast() {
                Thread.x.P(); {
                        while (waiters != null) {
                                waiters.s.V();
                                waiters = waiters.next;
                        }
                } Thread.x.V();
        }
}
```

Mike Burrows encountered this problem one more time when implementing Posix Threads (PThreads) for the DEC Tru64 operating system. Once again, the kernel primitives didn’t include a suitable operation to let him build condition variables in an obvious way, so once again he implemented them by running an explicit queue through per-thread control blocks.

## Optimising Signal and Broadcast

Since we’re considering this level of the threads implementation, I should point out one last performance problem, and what to do about it. If Signal is called with the lock m held, and if you’re running on a multi-processor, the newly awoken thread is quite likely to start running immediately. This will cause it to block again a few instructions later at (2) when it wants to lock m. If you want to avoid these extra reschedules, you need to arrange to transfer the thread directly from the condition variable queue to the queue of threads waiting for m. This is especially important in Java or C#, which both require that m is held when calling Signal or Broadcast.

## Conclusions

Well, history doesn't really have conclusions. But it does have a tendency to repeat. It will be nice if reading this anecdote prevents someone from repeating our mistakes, though I wouldn't bet on it.

Implementing condition variables out of a simple primitive like semaphores is surprisingly tricky. The tricky part arises because of the binary atomic operation in Wait, where the lock is released and the thread is enqueued on the condition variable. If you don't have a suitable binary operation available, and you attempt to construct one by clever use of something like a semaphore, you'll probably end up with an incorrect implementation. You should either do the queuing yourself, or lobby your kernel implementer to provide a suitable primitive.

Eager readers of the Win32 API will have noticed that NT version 4.0 and later provides such a binary operation (SignalObjectAndWait). This is probably sufficient to do a simple implementation of condition variables, but I'm not going to write it here. Using SignalObjectAndWait does have the down-side that the object being released has to be an NT kernel object, for example a kernel mutex or kernel semaphore. This makes it trickier to use if you want to implement locks with the more efficient Win32 "critical section" operations.

Finally, I should admit that this is an area where a small investment in formal methods would help. With a formal specification of the underlying primitives and a formal specification of the desired condition variable semantics, it should not be difficult to see at least the correctness flaws in the buggy designs. Current formal methods would do less well in detecting unacceptable performance penalties.

## References

1. BIRRELL, A., GUTTAG, J., HORNING, J. AND LEVIN, R. Synchronization primitives for a multiprocessor: a formal specification. In *Proceedings of the 11th Symposium on Operating System Principles* (Nov. 1987), 94-102.

2. DIJKSTRA, E.W. The Structure of the T.H.E. Multiprogramming System. *Commun. ACM 11,* 5 (May 1968), 341-346

3. GOSLING, G., JOY, B., STEELE, G. AND BRACHA, G. The Java Language Specification, Second Edition, Sun Microsystems (2000), 429-447.

4. HOARE, C.A.R. Monitors: An operating system structuring concept. *Commun. ACM 17,* 10 (Oct.1974), 549-557.

5. SALTZER, J. Traffic control in a multiplexed computer system. Th., MAC-TR-30, MIT, Cambridge, Mass. (July 1966).