Synchronization Primitives for a Multiprocessor: A Formal Specification

A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin

August 20, 1987

SRC Research Report 20

John Guttag's permanent address is MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, U.S.A.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, and by the National Science Foundation under grant DCR-8411639.

Abstract

Formal specifications of operating system interfaces can be a useful part of their documentation. We illustrate this by documenting the Threads synchronization primitives of the Taos operating system. We start with an informal description, present a way to formally specify interfaces in concurrent systems, and then give a formal specification of the synchronization primitives. We briefly discuss both the implementation and what we have learned from using the specification for more than a year. Our main conclusion is that programmers untrained in reading formal specifications have found this one helpful in getting their work done.

Introduction

The careful documentation of interfaces is an important step in the production of software upon which other software is to be built. If people are to use software without having to understand its implementation, documentation must convey semantic as well as syntactic information. When the software involves concurrency, adequate documentation is particularly hard to produce, since the range of possible behaviors is likely to be large and difficult to characterize [Jones 83].

We believe that operating system documentation containing formal specifications can be significantly better than documentation restricted to informal or semi-formal descriptions. It can be made more precise and complete, and is more likely to be interpreted consistently by various readers. Our experience in specifying and documenting the synchronization facilities of DEC/SRC's Threads package supports this view.

The Threads package implements concurrent threads of control* for the Taos operating system [McJones 87] and application programs running on our Firefly multiprocessor workstation [Thacker 87]. The package supports large numbers of concurrent threads, and permits multiple concurrent threads within an address space. The synchronization facilities we describe permit threads to cooperate in the use of shared memory.

We begin with an informal description of the Threads synchronization primitives. These are similar to those in many other systems, but their use on a multiprocessor raises questions about their precise semantics that are difficult to answer using informal descriptions.

We briefly describe the formal language we use to specify interfaces involving concurrency, and then present the specification itself. We intend to give the reader a complete and precise understanding of the properties that client programmers may rely on when using the Threads synchronization primitives. The specification should answer any questions about how our primitives differ from others with which the reader is familiar.

We next discuss the implementation of Threads. It is chiefly interesting for the way efficiency is obtained despite limited hardware support. The need for efficiency

* "Lightweight processes"; we avoid the word "process" because of its connotation of "address space" in some operating systems. Saltzer credits the term "thread" to V. Vyssotsky [Saltzer 66].

led to an implementation that has a rather different structure than the specification might suggest. This illustrates how specifications can protect clients from needing to know implementation details.

Finally, we discuss how the formal specification has been and is being used by programmers.

Informal Description

The Threads package implements a Modula-2+ [Rovner 86] interface for creating and controlling a virtually unlimited number of threads, which may or may not share memory. This paper is concerned only with its synchronization facilities. These are rather simple, and are derived from the concepts of *monitors* and *condition variables* first outlined by Hoare [Hoare 74]. Their semantics are similar to those provided by Mesa [Lampson 80]. The synchronization facilities use three main types: *Mutex, Condition,* and *Semaphore.*

As far as clients of Threads are concerned, all threads can execute concurrently. The Threads implementation is responsible for assigning threads to real processors. The way in which this assignment is made affects performance, but does not affect the semantics of the synchronization primitives. The programmer can reason as if there were as many processors as threads. The Threads package also includes facilities for affecting the assignment of threads to real processors (for example, a simple priority scheme), but our specification is independent of these facilities.

Mutual Exclusion: Acquire, Release

A *mutex* [Dijkstra 68] is the basic tool enabling threads to cooperate on access to shared variables. A mutex is normally used to achieve an effect similar to monitors, ensuring that a set of actions on a group of variables can be made *atomic* relative to any other thread's actions on these variables. Atomicity can be achieved by arranging that:

- All reads and writes of the shared variables occur within *critical sections* associated with these variables.
- Each critical section is executed from start to finish without any other thread entering a critical section associated with these variables.
- Each action that is to be atomic is entirely contained within a single critical section.

Such *mutual exclusion* completely serializes the critical sections, and hence the atomic actions they contain. It can be implemented using the procedures Acquire and Release. A mutex "m" is associated with the set of variables, and each critical section is bracketed by Acquire(m) and Release(m) actions. The semantics of Acquire and Release ensure that these bracketed sections are indeed critical sections.

Critical sections are so frequently useful that Modula-2+ includes syntactic sugar for them. The statement:

LOCK e DO statement-sequence END

is equivalent to:

LET m=e; Acquire(m); TRY statement-sequence FINALLY Release(m) END

This syntax encourages the use of correctly bracketed occurrences of Acquire and Release. It also makes it easy for the compiler to produce highly optimized code for such occurrences. (The TRY ... FINALLY construct ensures that Release will be called regardless of whether control leaves the statement-sequence because it has been completed or because an exception has been raised.) Other uses of Acquire and Release are generally discouraged.

Condition Variables: Wait, Signal, Broadcast

Condition variables make it possible for a thread to suspend its execution while awaiting an action by some other thread. For example, the consumer in a producer-consumer algorithm might need to wait for the producer. Or the implementation of a higher level locking scheme might require that some threads wait until a lock is available.

The normal paradigm for using condition variables is as follows. A condition variable "c" is always associated with some shared variables protected by a mutex "m" and a predicate based on those shared variables. A thread acquires m (i.e., enters a critical section) and evaluates the predicate to see if it should call Wait(m, c) to suspend its execution. This call atomically releases the mutex (i.e., ends the critical section) and suspends execution of that thread.

After any thread changes the shared variables so that c's predicate might be satisfied, it calls Signal(c) or Broadcast(c). (Although the changes may be made only within a critical section, the thread may exit the critical section before this call.) Signal and Broadcast allow blocked threads to resume execution and re-acquire the mutex. When a thread returns from Wait it is in a new critical section. It re-evaluates the predicate and determines whether to proceed or to call Wait again.

There are several subtleties in the semantics of these procedures, and it is difficult to express them correctly in an informal description. This is the area where we have found the formal specification most useful.

- Even if threads take care to call Signal only when the predicate is true, it may become false before a waiting thread resumes execution. Some other thread may enter a critical section first and invalidate the predicate. Therefore when a thread returns from a Wait it must re-evaluate its predicate and be prepared to call Wait again. Return from Wait is only a *hint* [Lampson 84] that must be confirmed. By contrast, with Hoare's condition variables threads are guaranteed that the predicate is true on return from Wait. Our looser specification reduces the obligations of the signalling thread and leads to a more efficient implementation on our multiprocessor.

- The two things that Wait(m, c) must do first—leave the critical section and suspend execution of the thread—must be in one atomic action relative to any call of Signal. The following sequence would be incorrect: one thread leaves its critical section; then another thread enters a critical section, modifies the shared variables, and calls Signal (which finds nothing to be unblocked); and then the first thread suspends execution. This is familiar to most operating system designers as a "wakeup-waiting race" [Saltzer 66]. Our semantics specify, and our implementation ensures, that no signals are lost between these two actions.

- There is a distinction between Signal and Broadcast. Signal is used to unblock one waiting thread; Broadcast is used to unblock all of them. Using Signal is preferable (for efficiency) when only one blocked thread can benefit from the change (for example, when freeing a buffer back into a pool). Broadcast is necessary (for correctness) if multiple threads should resume (for example, when releasing a "writer" lock on a file might permit all "readers" to resume).

- If Signal is used, all threads waiting on the same condition variable must be waiting for satisfaction of the same predicate. If Broadcast is always used instead, the predicates for different waiting threads need not all be the same.

Semaphores: P, V

The Threads package also provides binary semaphores with their traditional P and V operations. The implementation of semaphores is identical to mutexes, but they are used differently.* There is no notion of a thread "holding" a semaphore, and no precondition on executing V, so calls of P and V need not be textually linked.

We discourage programmers from using semaphores directly, since we prefer the additional structure that comes with the use of mutexes and condition variables. However they are required for synchronizing with interrupt routines. This is because an interrupt routine cannot protect shared data with a mutex—because the interrupt might have pre-empted a thread in a critical section protected by that mutex—and using Wait and Signal to synchronize requires use of an associated mutex. Instead, a thread waits for an interrupt routine action by calling P(sem), and the interrupt routine unblocks it by calling V(sem).

Alerting: Alert, TestAlert, AlertWait, AlertP

Alerting provides a polite form of interrupt, used in conjunction with both semaphores and condition variables, typically to implement things such as timeouts and aborts. It allows a thread to request that another thread desist from a computation. It is generally used in situations where the decision to make this request happens at an abstraction level higher than that in which the thread is blocked, so that the appropriate condition variable or semaphore is not readily accessible.

Calling Alert(t) is a request that the thread t raise the exception Alerted. The procedure TestAlert allows a thread to see if there is a pending request for it to raise Alerted. AlertWait is similar to Wait, except that AlertWait may raise Alerted rather than returning. The choice between AlertWait and Wait depends on whether or not the calling thread is to respond to an Alert at the point of the call. The procedure AlertP provides the analogous facility for semaphores.

---

* "We used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores." [Dijkstra 68]

Specification Approach

We use the Larch two-tiered approach to specification [Wing 83, Guttag 85a, b]. The *Larch Shared Language* tier [Guttag 86a] is algebraic, and defines mathematical abstractions that can be used in the *interface language* tier to specify program interfaces. As it happens, all the abstractions needed for the Threads specification are well known (e.g., booleans, integers, and sets) and appear in the Larch Shared Language Handbook [Guttag 86b], so we do not discuss them here.

The logical basis for our treatment of concurrency is very similar to the one discussed in [Lamport 83, 86]. However, our specification deals only with safety properties, such as partial correctness, not with liveness properties.

Our specifications of procedures for concurrent programs are similar to our specifications of procedures for sequential programs. In both cases, the specifications prescribe the observable effects of procedures, without saying how they are to be achieved. In a sequential program, the states between a procedure call and its return cannot be observed in the calling environment. Thus we can specify a procedure by giving a predicate relating just the state when the procedure is called and the state when it returns [Hoare 71]. Similarly, an *atomic action* in a concurrent program has no visible internal structure; its observable effects can also be specified by a predicate on just two states.

Our method is based on the observation that any behavior of a concurrent system can be described as the execution of a sequence of atomic actions. A key property of atomic actions is *serializability*, which means that each concurrent execution of a group of atomic actions has the same observable effects as some sequential execution of the same actions. Serializability allows us to ignore concurrency in reasoning about the effects of an atomic action. Each atomic action appears indivisible, both to the thread invoking it and to all other threads.

In specifying atomic actions, we don't specify how atomicity is to be achieved, only that it must be. In an implementation, atomic actions may proceed concurrently as long as the concurrency isn't observable. Atomicity is intimately related to abstraction; at each level of abstraction atomicity is ensured by using sequences of lower-level actions, some of which are known to be atomic relative to each other. For example, the atomicity of the Threads synchronization primitives is ensured by the atomicity of the underlying hardware's test-and-set instruction.

Atomicity requirements constrain both the thread executing the atomic action and all other threads that share variables with the action. For such a set of actions to be atomic relative to each other, their implementations must all adhere to some synchronization protocol. It is necessary to consider them all when verifying atomicity, just as it is necessary to consider all the operations of an abstract data type when verifying its implementation [Liskov 86].

Atomic procedures execute just one atomic action per call. Each can be specified in terms of just two states: the state immediately preceding and the state immediately following the action. They are particularly easy to specify and to understand, since they behave so much like procedures in a sequential environment. Thus we would prefer for most procedures to appear atomic to their callers. However, most concurrent programs contain a few procedures that do not; these present a more difficult specification challenge.

The observable effects of a *non-atomic procedure* cannot be described in terms of just two states. Its effects may span more states, and actions of other threads may be interleaved with its atomic actions. However, each execution of a non-atomic procedure can be viewed as a sequence of atomic actions. We specify a non-atomic procedure by giving a predicate that defines the allowable sequences of atomic actions (i.e., sequences of pre-post state pairs). Each execution of the procedure must be equivalent to such a sequence. Although it is sometimes necessary to specify constraints on the sequence as a whole, in our example it suffices to specify the atomic actions separately.

The Threads interface contains two non-atomic synchronization procedures (Wait and AlertWait), each executing two visible atomic actions per call. Such procedures are nearly as easy to specify as atomic procedures. We specify that (the visible effect of) executing the procedure must be equivalent to executing two named actions in order (possibly separated by actions of other threads), and then write separate predicates specifying the two actions.

To make these ideas a bit more concrete, consider the following specification from the Threads interface:

TYPE Mutex = Thread INITIALLY NIL

TYPE Condition = SET OF Thread INITIALLY {}

PROCEDURE Wait(VAR m: Mutex; VAR c: Condition)
  = COMPOSITION OF Enqueue; Resume END
 REQUIRES m = SELF
 MODIFIES AT MOST [m, c]
 ATOMIC ACTION Enqueue
  ENSURES (c_{post} = insert(c, SELF)) & (m_{post} = NIL)
 ATOMIC ACTION Resume
  WHEN (m = NIL) & ¬(SELF ∈ c)
   ENSURES m_{post} = SELF & UNCHANGED [c]

This specification is similar to a sequential Larch specification:

- A REQUIRES clause states a precondition that the implementation may rely on; it is the responsibility of the caller to ensure that the condition holds at the start of the procedure's first atomic action. The specification does not constrain the implementation to any particular behavior if the precondition is not satisfied. An omitted REQUIRES clause is equivalent to REQUIRES TRUE, that is, nothing is required.
- A MODIFIES AT MOST clause identifies the objects that the procedure is allowed to change.
- An ENSURES clause states a postcondition that the atomic action must establish.
- An unsubscripted argument formal in a predicate stands for its value in the *pre state*—the state in which the atomic action begins. A return formal or an argument formal subscripted by $_{post}$ stands for the value associated with the formal in the *post state*—the state at the conclusion of the atomic action.

To deal with concurrency, we extended sequential Larch in the following ways:

- A WHEN clause states a condition that must be satisfied for an atomic action to take place. It is not a precondition of the call, but the called procedure is obligated to make sure that the condition holds before taking any externally visible action. A WHEN clause may thus impose a delay until actions of other threads make its predicate true. An omitted WHEN clause is equivalent to WHEN TRUE, that is, no delay is required.
- ATOMIC preceding PROCEDURE or ACTION indicates that any execution of the procedure or action must be atomic relative to the other actions of the interface.

- A COMPOSITION OF clause indicates that any execution of the procedure must be equivalent to execution of the named actions in the given order, possibly interleaved with actions of other threads.
- An ATOMIC ACTION clause specifies a named action in much the same way as an ATOMIC PROCEDURE specification does. It is within the scope of the procedure header, and may refer to its formal parameters and results.
- The keyword SELF stands for the identity of the thread executing the specified action.

Formal Specification

Now we present the formal specification without much commentary. This specification is self-contained; none of the informal description of threads is needed to understand its precise semantics. However, it is intended to be used in conjunction with informal material. In the documentation used at SRC, informal material is interleaved with the specification, both to provide intuition and to say how the primitives are intended to be used. Our informal documentation is somewhat shorter than that given above, because more of the burden of communicating precise details has been shifted to the formal specification.

Mutex, Acquire, Release

TYPE Mutex = Thread INITIALLY NIL

ATOMIC PROCEDURE Acquire(VAR m: Mutex)
  MODIFIES AT MOST [m]
  WHEN m = NIL ENSURES m_{post} = SELF

ATOMIC PROCEDURE Release(VAR m: Mutex)
  REQUIRES m = SELF
  MODIFIES AT MOST [m]
  ENSURES m_{post} = NIL

If Release(m) is executed when there are several threads waiting to perform Acquire(m), the WHEN clause of each of them will be satisfied. Only one thread will hold m next, because—by atomicity of Acquire—it must appear that one of the Acquires is executed first; its ENSURES clause falsifies the WHEN clauses of all the others. Our specification does not say which of the blocked threads will be unblocked first, nor when this will happen.

Condition, Wait, Signal, Broadcast

TYPE Condition = SET OF Thread INITIALLY {}

PROCEDURE Wait(VAR m: Mutex; VAR c: Condition)
    = COMPOSITION OF Enqueue; Resume END
  REQUIRES m = SELF
  MODIFIES AT MOST [m, c]
  ATOMIC ACTION Enqueue
    ENSURES (c_{post} = insert(c, SELF)) & (m_{post} = NIL)
  ATOMIC ACTION Resume
    WHEN (m = NIL) & ¬(SELF ∈ c)
      ENSURES m_{post} = SELF & UNCHANGED [c]

ATOMIC PROCEDURE Signal(VAR c: Condition)
  MODIFIES AT MOST [c]
  ENSURES (c_{post} = {}) | ($c_{post} \subset$ c)

ATOMIC PROCEDURE Broadcast(VAR c: Condition)
  MODIFIES AT MOST [c]
  ENSURES c_{post} = {}

Any implementation that satisfies Broadcast's specification also satisfies Signal's. We cannot strengthen Signal's postcondition: although our implementation of Signal usually unblocks just one waiting thread, it may unblock more.

Semaphore, P, V

TYPE Semaphore = (available, unavailable) INITIALLY available

ATOMIC PROCEDURE P(VAR s: Semaphore)
  MODIFIES AT MOST [s]
  WHEN s = available ENSURES s_{post} = unavailable

ATOMIC PROCEDURE V(VAR s: Semaphore)
  MODIFIES AT MOST [s]
  ENSURES s_{post} = available

Alerts, Alerted, TestAlert, AlertP, AlertWait

VAR alerts: SET OF Thread INITIALLY {}
EXCEPTION Alerted

ATOMIC PROCEDURE Alert(t: Thread)
MODIFIES AT MOST [alerts]
ENSURES alerts_{post} = insert(alerts, t)

ATOMIC PROCEDURE TestAlert() RETURNS(b: bool)
MODIFIES AT MOST [alerts]
ENSURES (b = (SELF ∈ alerts)) & (alerts_{post} = delete(alerts, SELF))

ATOMIC PROCEDURE AlertP(VAR s: Semaphore) RAISES{Alerted}
MODIFIES AT MOST [s, alerts]
RETURNS WHEN s = available
ENSURES (s_{post} = unavailable) & UNCHANGED [alerts]
RAISES Alerted WHEN (SELF ∈ alerts)
ENSURES (alerts_{post} = delete(alerts, SELF)) & UNCHANGED [s]

PROCEDURE AlertWait(VAR m: Mutex; VAR c: Condition) RAISES{Alerted}
= COMPOSITION OF Enqueue; AlertResume END
REQUIRES m = SELF
MODIFIES AT MOST [m, c, alerts]
ATOMIC ACTION Enqueue
ENSURES (c_{post} = insert(c, SELF)) & (m_{post} = NIL) & UNCHANGED [alerts]
ATOMIC ACTION AlertResume
RETURNS WHEN (m = NIL) & ¬(SELF ∈ c)
ENSURES (m_{post} = SELF) & UNCHANGED[c, alerts]
RAISES Alerted WHEN (m = NIL) & (SELF ∈ alerts)
ENSURES (m_{post} = SELF) & (c_{post} = delete(c, SELF)) &
(alerts_{post} = delete(alerts, SELF))

In both AlertP and AlertWait, the RETURNS and RAISES clauses are not disjoint. This non-determinism will be discussed later.

Implementation

We have two implementations of the Threads package. One runs within any single process on a normal Unix system. It is implemented using a co-routine mechanism for blocking one thread and resuming another. We will not discuss that version further in this paper. Our other implementation runs on the Firefly, and uses multiple processors to provide true concurrency.

The Firefly is a symmetric multiprocessor; each processor is able to address the entire memory. In general a thread is not concerned about which processors it executes on, and the scheduler is free to move it from one processor to another.

The implementation of the synchronization primitives has two layers. The top layer (the *user code*) is executed in the thread's own address space; the lower layer (the *nub code*), in the common kernel (Nub) address space. The purpose of having code in the user space is to optimize most cases where the synchronization action will not cause the thread to block, nor cause another thread to resume—for example, executing the Acquire and Release for a LOCK clause when there is no contention for its mutex, or executing Signal or Broadcast when there is no thread blocked on the condition variable. The user code avoids the overhead of calling the Nub in these cases.

The Nub subroutines execute under the protection of a more primitive mutual exclusion mechanism, a *spin-lock* [Jones 80]. The spin-lock is represented by a globally shared bit: it is acquired by a processor busy-waiting in a test-and-set loop; it is released by clearing the bit. Nub subroutines acquire the spin-lock, perform their visible actions, and release the spin-lock.

The Nub maintains queues of threads that have been blocked by Acquire, Wait, or P actions. It also maintains a "ready pool" of threads that are available for execution. When a thread is blocked by Acquire, Wait, or P, the Nub looks in the ready pool to see if there is a thread to run on the processor now available. When threads are added to the ready pool by Release, Signal, Broadcast, or V, the Nub looks for a suitable processor for them to execute on. The Nub also implements a priority-based scheduling algorithm and a time-slicing algorithm.

Further complications (which we will not discuss here) arise because synchronization data might be allocated in non-resident memory.

Mutexes and semaphores

A mutex is represented by a pair (Lock-bit, Queue). The Lock-bit is 1 if a thread is in a critical section protected by the mutex, and is 0 otherwise. In terms of the formal specification, the Lock-bit is 0 iff the mutex is NIL. The Queue contains the threads that are blocked in Acquire (awaiting its WHEN condition).

The user code for Acquire and Release is designed for fast execution of a LOCK clause when there is no contention for its mutex. In this case an Acquire-Release pair executes a total of 5 instructions, taking 10 microseconds on a MicroVAX II. This code is compiled entirely in-line. Acquire consists of two sequential actions: test-and-set the Lock-bit (implemented atomically in the hardware); call a Nub subroutine if the bit was already set. The user code for Release is two sequential actions: clear the Lock-bit; call a Nub subroutine if the Queue is not empty.

The Nub subroutine for Acquire (after acquiring the spin-lock) first adds the calling thread to the Queue. Then it tests the Lock-bit again. If it is still 1, this thread is de-scheduled and the general scheduling algorithm is invoked to determine what to do with this processor. On the other hand, if the Lock-bit is now 0, the thread is removed from the Queue, the spin-lock is released, and the entire Acquire operation (beginning at the test-and-set) is retried.

The Nub subroutine for Release (after acquiring the spin-lock) checks to see if there are any threads in the Queue. If there are, it takes one, adds it to the ready pool, and invokes the general scheduling algorithm, which will assign the thread to a suitable processor if one is available.

The implementation of semaphores is the same as mutexes: P is the same as Acquire and V is the same as Release.

Condition variables

The semantics of Wait and Signal could be achieved by representing each condition variable as a semaphore, and implementing Wait(m, c) as

Release(m); P(c); Acquire(m)

and Signal(c) as V(c). The one bit in the semaphore c would cover the wakeup-waiting race. Unfortunately, this implementation does not generalize to Broadcast(c). The reason is that there might be arbitrarily many threads in the race (at the semicolon between Release(m) and P(c)), and the implementation of Broadcast would have no way of indicating that they should *all* resume execution.

Our implementation uses an *eventcount* [Reed 77] to resolve this problem. An eventcount is an atomically-readable, monotonically-increasing integer variable. The representation of a condition variable is a pair (Eventcount, Queue).

The user implementation of Wait(m, c) is as follows. First it reads c's Eventcount; say this value is "i". Second, it calls Release(m). Third, it calls a Nub subroutine Block(c, i). On return from Block it calls Acquire(m). The Nub subroutine Block first acquires the spin-lock. It then compares i with the current value of c's Eventcount. If they are equal, the current thread (SELF) is added to c's Queue and de-scheduled; if they are unequal (because there has been an intervening Signal or Broadcast), Block just returns.

The Nub implementations of Signal and Broadcast (after acquiring the spin-lock) increment c's Eventcount, then inspect c's Queue. If it is non-empty, Signal takes one of its threads and adds it to the ready pool; Broadcast does this for all the threads in Queue. The wakeup-waiting race is handled by c's Eventcount. A thread executing in Wait will simply return from Block if a Signal or Broadcast has intervened between the time it read Eventcount and the time it acquired the spin-lock. It is possible (though unlikely) that Signal will acquire the spin-lock while more than one thread is trying to acquire it in Wait; if so, Signal will unblock all such threads.

The user code for Wait, Signal, and Broadcast includes optimizations so that Signal and Broadcast avoid calling the Nub if there are no threads to unblock.

Discussion

A prose description of the Threads synchronization primitives was written when the interface was first designed [Rovner 85, 86]. While it gave an indication of how the primitives were intended to be used, it left too many questions about the guaranteed behavior of the interface unanswered.

To provide more precise information for programmers who were starting to use the interface, a semi-formal operational specification was written. This description was both precise and (for the most part) accurate. The main problems with it were that it was too subtle and that important information was rather widely distributed. For example, to discover that Signal might unblock more than one thread involved looking at several procedures and observing that a race condition existed. If one failed to notice this race condition—and most readers seemed to—one was misled about the behavior of Signal. This is not a criticism of the particular specification, but rather an indication that it is difficult to write straightforward operational specifications of concurrent programs. In our specification, the weakness of the guarantee is explicit in Signal's ENSURES clause.

The operational specification was the starting point for our formal specification, and served us well. The two of us who wrote the formal specification still had questions for the two involved in the implementation, but never resorted to studying the actual code.

Our specification has passed the coffee-stain test. A somewhat condensed version of the documentation presented here (formal and informal) is the reference of choice for programmers using the Threads interface and for those responsible for its implementation. They seem to be able to read our specification and understand its implications. Two incidents illustrate this; both relate to places where the version of the specification we first released did not conform to the implementation:

- The original specification of AlertWait did not contain "m = NIL &" in the RAISES clause of AlertResume. That this presented a problem was discovered in less than an hour by someone with no prior knowledge of either the interface or the specification technique.
- The second problem was more subtle. In the specification of AlertP and AlertWait, the WHEN clauses of the normal (RETURNS) and exceptional (RAISES) cases are not mutually exclusive; this gives their implementations the right to make arbitrary choices when both are satisfied. In the original specification, these procedures were constrained to raise the exception Alerted if possible. This was consistent with the operational specification. After our specification was released, a programmer pointed out that the implementation was non-deterministic: sometimes it raised the exception and sometimes it didn't. The implementor decided that the efficiency advantages gained by allowing non-determinism made a specification change desirable.

We must also report a more worrisome incident:

- An error in the specification that had not been noticed during more than a year of use was discovered* while this paper was being prepared for publication. The problem was again in the specification of AlertWait. The specification incorrectly required that when AlertWait raised the exception Alerted it left the value of c unchanged. Thus c could contain threads that were no longer blocked on the condition variable.

* by Greg Nelson, in the course of preparing the review included in this report.

We are vexed that it took so long for anyone to notice this error. We can think of several possible contributing factors:

- AlertWait is the most complicated primitive in this interface, and also the least familiar. The specifiers had less experience to fall back on.
- Semaphores and condition variables are similar in many ways. Perhaps readers, having studied the specification of AlertP (where UNCHANGED [s] is correct for the Alerted clause), failed to notice the consequences of the essential difference in the abstractions used to describe semaphores and condition variables.
- Even after the problem was discovered, it was difficult to convince ourselves (one at a time) that it was indeed a bug. The most convincing argument was operational: suppose a thread, t, raises Alerted, then a thread invokes Signal, which chooses to remove t from c, which means that no blocked thread is awakened by that Signal.
- Our specification does not deal with liveness properties; it cannot be used to prove that anything *must* happen. Since arguments about safety properties are unaffected by this particular error in the specification, no one was forced to confront it.

A more encouraging aspect of our experience is the role played by the specification in insulating clients from the implementation of the Threads package:

- As discussed in the section on implementation, mutexes are implemented using queues of blocked threads, without recording which thread currently holds the mutex; this is quite different from what one might guess after reading our specification. The client programmer, however, need not know this. The specification abstracts from details of the implementation to provide a simpler model. (In fact, some programmers have complained because the debugger doesn't provide a simple way to determine which thread holds an unavailable mutex.)
- Although the underlying implementation has been reworked several times, both to improve efficiency and to make it easy to collect statistics on contention, the specification of the synchronization primitives (other than AlertWait) has been unchanged for more than a year. Client programmers have not needed to respond to, or even know about, the implementation changes.

- Although semaphores and mutexes have identical implementations, the interface provides distinct types with different specifications. Mutexes have holders and semaphores don't; Release has a REQUIRES clause and V doesn't. The choice to have two types for the two different ways of using the underlying mechanism had already been made by the designers when the formal specification was started. Client programs that rely only on the specified properties of these types would continue to work even if their implementations were different.

Our experience with the Threads specification indicates that formal specifications of concurrent programs can be used productively by systems programmers, but it says little about the ease with which they can be produced. The specification was written by two of us who have many years of experience in writing formal specifications.

Single small examples can be illustrative, but it is unwise to generalize too much from them. We are applying these techniques to other system interfaces, both at MIT and at SRC, and expect to report our results in future papers.

Writing good specifications is time consuming. In our experience, the bulk of the time spent specifying a system goes first to understanding the object to be specified and then to choosing abstractions to help structure the presentation of that understanding. We spend relatively little time translating our understanding into the specification language.

Understanding systems with a high degree of concurrency is particularly difficult. When studying the designs of such systems, it is often hard to disentangle the behavior implied by a particular implementation from the behavior that all implementations should be required to exhibit. At the very least, specifiers must have ready access to the designers of the system to answer such questions.

In this paper we have stressed the utility of formal specifications in documenting interfaces. They can contribute structure and regularity. By enforcing precision they encourage accuracy and completeness. Formal specifications are not a replacement for careful prose documentation. However, they can lead to better informal documentation by making it unnecessary for the prose to exhibit such a high level of precision and completeness.

Acknowledgments

Leslie Lamport and Jeannette Wing were both involved in the discussions leading to this specification, and helped us to understand the issues involved.

References

[Dijkstra 68] Edsger W. Dijkstra, "The Structure of the 'THE'—Multiprogramming System," *Comm. ACM*, vol. 11, no. 5, 341–346.

[Guttag 85a] J. V. Guttag, J. J. Horning, and J. M. Wing, "Larch in Five Easy Pieces," Report 5, Digital Equipment Corporation Systems Research Center, Palo Alto; portions published separately in [Guttag 85b, 86a, b].

[Guttag 85b] John V. Guttag, James J. Horning, and Jeannette M. Wing, "The Larch Family of Specification Languages," *IEEE Software*, vol. 2, no. 5, 24–36.

[Guttag 86a] J. V. Guttag and J. J. Horning, "Report on the Larch Shared Language," *Science of Computer Programming*, vol. 6, 103–134.

[Guttag 86b] J. V. Guttag and J. J. Horning, "A Larch Shared Language Handbook," *Science of Computer Programming*, vol. 6, 135–156.

[Hoare 71] C. A. R. Hoare, "Procedures and Parameters: An Axiomatic Approach," *Symposium on Semantics of Algorithmic Languages*, Springer-Verlag, 102–116.

[Hoare 74] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. ACM*, vol. 17, no. 10, 549–557.

[Jones 80] Anita K. Jones and Peter Schwarz, "Experience Using Multiprocessor Systems—A Status Report," *Computing Surveys*, vol. 12, no. 2, 121–165.

[Jones 83] C. B. Jones, "Specification and Design of (Parallel) Programs," *Proc. IFIP Congress '83*.

[Lamport 83] Leslie Lamport, "Specifying Concurrent Program Modules," *ACM TOPLAS*, vol. 5, no. 2, 190–222.

[Lamport 86] Leslie Lamport, "A Simple Approach To Specifying Concurrent Systems," Report 15, Digital Equipment Corporation, Systems Research Center, Palo Alto.

[Lampson 80] B. W. Lampson and D. D. Redell, "Experiences with Processes and Monitors in Mesa," *Comm. ACM*, vol. 23, no. 2, 105–117.

[Lampson 84] Butler W. Lampson, "Hints for Computer System Design," *IEEE Software*, vol. 1, no. 1, 11–28.

[Liskov 86] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*, MIT Press/McGraw-Hill.

[McJones 87] Paul R. McJones and Garret F. Swart, "Evolving the UNIX System Interface to Support Multithreaded Programs," Report to appear, Digital Equipment Corporation, Systems Research Center, Palo Alto.

[Reed 77] David P. Reed and Rajendra K. Kanodia, "Synchronization with Eventcounts and Sequencers," (abstract only) *Proc. Sixth ACM Symposium on Operating Systems Principles*, 91.

[Rovner 85] Paul Rovner, Roy Levin, and John Wick, "On Extending Modula-2 for Building Large, Integrated Systems," Report 3, Digital Equipment Corporation, Systems Research Center, Palo Alto.

[Rovner 86] Paul Rovner, "Extending Modula-2 to Build Large, Integrated Systems," *IEEE Software*, vol. 3, no. 6, 46–57.

[Saltzer 66] Jerome Howard Saltzer, "Traffic Control in a Multiplexed Computer System," Technical Report MAC-TR-30 (Thesis), Massachusetts Institute of Technology, Cambridge.

[Thacker 87] Charles P. Thacker and Lawrence C. Stewart, "Firefly: a Multiprocessor Workstation," to be published in *Proc. Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, October 5-8, 1987.

[Wing 83] Jeannette Marie Wing, "A Two-Tiered Approach to Specifying Programs," Technical Report MIT/LCS/TR-299, Massachusetts Institute of Technology, Cambridge.