Reliable Synchronization Primitives for Java Threads *

Paolo A.G. Sivilotti, K. Mani Chandy
California Institute of Technology
Pasadena CA, 91125
{mani,paolo}@cs.caltech.edu

June 1, 1996

Abstract

Java is an architecture-independent, object-oriented language designed to facilitate code-sharing across the Internet in general, via the Web in particular. Java is multithreaded, providing thread creation and synchronization constructs based on generalized monitors. Although these primitives are appropriate for many windowing applications, they are not necessarily well-suited for the larger class of multithreaded programs that occur as part of distributed systems. We demonstrate how the Java primitives, in conjunction with the object-oriented aspects of the language, can be used to implement a collection of other traditional synchronization paradigms. These paradigms are formally specified, their implementations are rigorously verified, and their use is illustrated with several examples.

1 Introduction

The World Wide Web is a popular tool for information exchange and for "user-to-remote-server" applications. The level of client-server interaction supported by these applications varies from Web browsers that permit direct information retrieval, to Web forms that provide a simple way for users to submit data to programs at remote sites, to applets written in Java that allow instruction streams from remote sites to be interpreted safely on a user's local computer. This report describes a part of a project that extends the Web as a general-purpose distributed system [4]. The next few paragraphs describe some of the overall design issues for this larger project.

*This research was supported in part by NSF/PSE grant CCR-9527130, by AFOSR grant AFOSR-91-0070, and by an IBM Computer Science Fellowship.

Distributed Systems and the Web. We define a distributed computing system to be a set of processes that communicate with each other by means of messages. A distributed system has constructs for creating processes, for changing the state of a process, and for sending and receiving messages. A distributed system may also have constructs for manipulating threads of computation within a single address space. The specific form of the constructs is not important at this stage; our goal here is limited to identifying some of the functionalities associated with distributed systems. The Web is extremely versatile, but it does not as yet provide the full functionality of a peer-to-peer distributed system. Two good reasons for *not* providing this functionality are: (i) erroneous processes can destroy information in the address spaces within which they execute, and (ii) poorly designed processes can degrade performance of the network.

There are, however, strong reasons for providing the functionality of a distributed system, provided the system is robust. A distributed system allows for direct interaction between processes without human intervention. Such interaction can be helpful for a variety of activities ranging from commerce (*e.g.*, arbitrage), to collaboration (*e.g.*, calendar tools to arrange meetings for a geographically distributed interest group) to entertainment (*e.g.*, distributed games).

Characteristics of Web-Based Distributed Systems. Some of the characteristics that we consider important in a Web-based distributed system include:

1. **High confidence:** The system must be robust for two reasons. The first reason is to reduce the possibility that a faulty application will harm performance of the network infrastructure or destroy information. The second reason is that, because of the democratic nature of the Web, the users of a Web-based system are likely to come from diverse backgrounds, with varying levels of expertise. Therefore, the system has to support the development of reliable applications by programmers unfamiliar with the subtleties of distributed systems.

2. **Wide variation in communication delays and process lifetimes:** The system must deal with a wide range in communication delays because of the ranges in distance (across the room or across the globe) and congestion. Process lifetimes will also vary greatly, with some long lasting (or even virtually permanent) processes, and others with very short duration.

3. **Integration with the Web:** One of the strengths of the Web is its accessibility. The distributed system should be a small extension of existing Web technologies to facilitate implementation and widespread use by the many people who use the Web.

Design Directions. These characteristics motivated the following design directions.

1. **Java:** We are basing our implementation on the Java language for several reasons including: Java is supported by some Web browsers, it is designed to provide safe interpretation of programs on a variety of platforms, it supports multiple threads and user interfaces, and it is object-oriented.

2. **Asynchronous communication:** Our programming model supports asynchronous sending of messages or "one-way" RPCs as opposed to rendezvous-based communication to deal with wide ranges of communication delay.

3. **Process persistence:** We plan to use Web tools, while allowing processes to maintain state during arbitrarily long delays between communication events.

4. **Specification and correctness of processes:** Much of our work deals with methods for specifying systems and processes and for developing correct systems (*i.e.*, systems that do what they are specified to do). An underlying problem is that the Web is amorphous, democratic and even chaotic, whereas development of reliable systems is simpler with uniform methods of specification, development, proving and testing.

5. **Application patterns of communication:** We propose to specify and then develop reliable subsystems that support classes of applications with common patterns of interaction among processes. Our intent is to support the development of reliable applications within these classes.

6. **Verified mechanisms of synchronization:** We also propose to specify and develop reliable libraries of constructs for synchronization and communication. This synchronization occurs between threads that share address space as well as between distributed processes. Examples of such constructs in the case of thread synchronization might include (among others): single-assignment variables for data-flow computations, semaphores for traditional suspension and signaling protocols, rendezvous points for irregularly coupled processes, and generalized barriers for reduction-based operations.

2 Threads and Synchronization Libraries

One of the fundamental elements of our model of Web-based distributed computing is the interaction (synchronization and communication) of threads within a single address space. That is, the processes involved in a distributed computation will themselves be multithreaded. This is useful because of the kinds of applications we anticipate being executed, in which processes are concurrently interacting with an end user, local files, and several classes of remote processes.

The focus of this paper is on a robust library for high-confidence interaction and synchronization between Java threads. Java [6] provides a thread synchronization mechanism based on a generalized notion of monitors. The Java synchronization primitives are naturally suited to multithreaded windowing applications and user interfaces. These primitives alone, however, are not appropriate for the full range of coordination paradigms desired by distributed application programmers. We exploit the object-oriented aspects of Java to provide a library of alternate synchronization schemes, including single-assignment variables, reusable barriers, locks, semaphores, bounded semaphores, bounded buffers. The remainder of this report presents these libraries – their formal specification, rigorous verification at the level of the Java code itself, and illustration with several examples.

3 Single-Assignment Variables

3.1 Specification

Single-assignment variables have their history in data-flow programming [1] [9] [3] [8]. They succinctly express data-flow-based synchronization requirements in concurrent programs. A single-assignment variable is initially undefined, and it can be written to (or defined) at most once. A subsequent attempt to write to the variable is a run-time error. If a thread attempts to read a single-assignment variable that has not yet been defined, that thread suspends until the variable becomes defined.

For a given single-assignment variable, a write operation never suspends. We consider this operation to be atomic, so the number of writes initiated is equal to the number that have completed. Let `cWrite` be the number of write operations that have been performed on such a variable. Because the variable cannot be multiply defined, we have the following safety condition:

$$(\texttt{cWrite} = 0) \quad \vee \quad (\texttt{cWrite} = 1) \tag{1}$$

A read operation, on the other hand, can suspend. We therefore distinguish between initiation and completion of this action. Let `iRead` and `cRead` be the number of read operations that have been initiated and completed, respectively. A second requirement is that no read operations complete on a variable that has not been defined:

$$(\texttt{cWrite} = 0) \quad \Rightarrow \quad (\texttt{cRead} = 0) \tag{2}$$

Conversely, we require that all read operations complete on a variable that has been defined.

$$(\texttt{cWrite} = 1) \quad \Rightarrow \quad (\texttt{cRead} = \texttt{iRead}) \tag{3}$$

3.2 Design and use

The most fundamental single-assignment type is the single-assignment unary type. More general single-assignment types can be created by extending this basic type, or through the use of generics. For example, to implement a single-assignment integer type, this class could be extended, and two new functions, `int read()` and `write(int)` defined. The function `read()` begins with a call to the base class `read()` method. The function `write()` ends with a call to the base class `write()` method.

3.3 Implementation

Single Class

```
public class Single  {
  private boolean defined = false;

  public synchronized void read () throws InterruptedException {
    if (!defined)
      wait();
  }

  public synchronized void write () throws DefinedTwiceException {
    if (defined)
      throw new DefinedTwiceException();
    else  {
      defined = true;
      notifyAll();
    }
  }
}
```

3.4 Verification

3.4.1 State

The state of a single-assignment variable is defined by:

1. The number of write operations that have been performed. Because this is a single-assignment variable, this number is bounded by 1.

$$\texttt{cWrite} \leq 1 \tag{4}$$

2. Whether or not the single-assignment variable is defined.

$$\texttt{defined} \Leftrightarrow (\texttt{cWrite} > 0) \tag{5}$$

3. The number of suspended read operations. Once initiated, a read operation either completes or suspends. Let `qRead` be the number of suspended read operations.

$$\texttt{iRead} = \texttt{qRead} + \texttt{cRead} \tag{6}$$

3.4.2 Properties

Maximality of Progress. Either there are no suspended read operations, or the single-assignment variable is not defined.

$$(\texttt{qRead} = 0) \quad \vee \quad (\neg\texttt{defined}) \tag{7}$$

Constraint of Progress. Either there are no completed read operations, or the single-assignment variable is defined.

$$(\texttt{cRead} = 0) \quad \vee \quad (\texttt{defined}) \tag{8}$$

3.4.3 Correctness

By annotating the implementation with assertions and ghost variables, the assertions (4) - (8) can be shown to be invariantly true. Note that the assertions need not hold inside synchronize actions (which are defined to be atomic by Java semantics).

Annotated Program for Read

```
public synchronized void read () throws InterruptedException {

    //assert: (4) - (8)

    //iRead++

    //assert: iRead= qRead+ cRead+ 1
    //assert: (4), (5), (7), (8)

    if (!defined) {

        //assert: ¬defined
        //assert: iRead= qRead+ cRead+ 1
        //assert: (4), (5), (7), (8)

        //qRead++

        //assert: ¬defined
        //assert: (4) - (8)

        wait();

        //assert: defined
        //assert: (4) - (8)
    }
}
```

Annotated Program for Write

```
public synchronized void write () throws DefinedTwiceException

    //assert: (4) - (8)

    if (defined) {

        //assert: defined
        //assert: (4) - (8)

        throw new DefinedTwiceException ();

    else {

        //assert: ¬ defined
        //assert: cWrite= 0
        //assert: (4) - (8)

        defined = true;

        //assert: defined
        //assert: cWrite= 0
        //assert: (4), (6), (8)

        notifyAll();
        //qRead= 0
        //cRead= iR

        //assert: defined
        //assert: cWrite= 0
        //assert: (4), (6) - (8)
    }

    //cWrite++

    //assert: defined
    //assert: (4) - (8)
}
```

3.4.4 Proof of Specification

Now the conjunction of the specifications (1) - (3) follows from the conjunction of the invariants.

Proof of (1)

$$
\begin{aligned}
& (\texttt{cWrite} = 0) \vee (\texttt{cWrite} = 1) \\
\Leftrightarrow \quad & \quad \{ \text{ by } (4) \} \\
& \textbf{true} \quad \Box
\end{aligned}
$$

Proof of (2)

$$
\begin{aligned}
& \texttt{cWrite} = 0 \\
\Leftrightarrow \quad & \{ \text{ by } (5) \} \\
& \neg\texttt{defined} \\
\Rightarrow \quad & \{ \text{ by } (8) \} \\
& \texttt{cRead} = 0 \quad \Box
\end{aligned}
$$

Proof of (3)

$$
\begin{aligned}
& \texttt{cWrite} = 1 \\
\Rightarrow \quad & \{ \text{ by } (5) \} \\
& \texttt{defined} \\
\Rightarrow \quad & \{ \text{ by } (7) \} \\
& \texttt{qRead} = 0 \\
\Leftrightarrow \quad & \{ \text{ by } (6) \} \\
& \texttt{iRead} = \texttt{cRead} \quad \Box
\end{aligned}
$$

3.5 Example: Calculating the Powers of 2

The powers of 2 can be calculated in parallel by exploiting the observation that:

$$
2^i = \begin{cases} 2^{i/2} * 2^{i/2} & \text{if } i \text{ is even} \\ 2^{(i-1)/2} * 2^{(i-1)/2} * 2 & \text{if } i \text{ is odd} \end{cases} \tag{9}
$$

The following program calculates an array of the first N powers of 2. A separate thread of control is spawned for each element of the array. This thread suspends until the $(i/2)^{th}$ element is defined. When the first element is defined, the computation proceeds with increasing opportunities for parallelism. This example is not meant to illustrate an efficient parallel implementation of this calculation, only to demonstrate the mechanism of synchronization provided by single-assignment types.

```
class Pow2Thread extends Thread  {
  private int id;
  private Single[] Flags;
  private int[] Values;

  Pow2Thread (int i, Single[] flags, int[] values)  {
    id = i;
    Flags = flags;
    Values = values;
  }

  public void run() {
```

```
    Flags[id/2].read();
    if (even(id))  Values[id] = Values[id/2] * Values[id/2];
    else           Values[id] = Values[id/2] * Values[id/2] * 2;
    Flags[id].write();
  }
} //end Pow2Thread

class Pow2  {
  static int[] Calculate (int N) throws DefinedTwiceException,
                                        InterruptedException  {
    int[] P = new int[N];
    Single[] F = new Single[N];
    for (int i=0; i<N; i++) F[i] = new Single();
    for (int i=1; i<N; i++) (new Pow2Thread (i,F,P)).start();
    P[0] = 1;
    F[0].write();
    for (int i=0; i<N; i++) F[i].read();
    return P;
  }
} //end Pow2
```

The parallelism and synchronization constraints expressed in this program are illustrated in Figure 1. Solid arrows indicate flow of control dictated by sequential composition of statements. Dotted arrows indicate a data dependency between concurrent threads of control which enforces a synchronization point via a single-assignment variable.

4 Reusable Barriers

4.1 Specification

A barrier provides rendezvous synchronization for a group of concurrent threads. Java provides a simple mechanism for pairwise barrier synchronization: One thread can execute a `join()` operation on another thread. This has the effect of suspending the former thread until the latter completes. Because a Java thread cannot be restarted after it has completed, a collection of tasks that require repeated barrier synchronization must be repeatedly reallocated, reinitialized, and restarted. To avoid this costly and cumbersome overhead, a reusable barrier can be used instead.

A reusable barrier object has a single fundamental operation, meet. This operation suspends until all participating threads have called meet. The number of participating threads is called the size of the barrier. Because the meet operation can suspend, we distinguish between the number of times it has been initiated (`iMeet`) and the number of times it has completed (`cMeet`). The specification for a barrier is therefore:

$$\texttt{cMeet} \;=\; (\texttt{iMeet} \operatorname{div} \texttt{size}) \times \texttt{size} \tag{10}$$

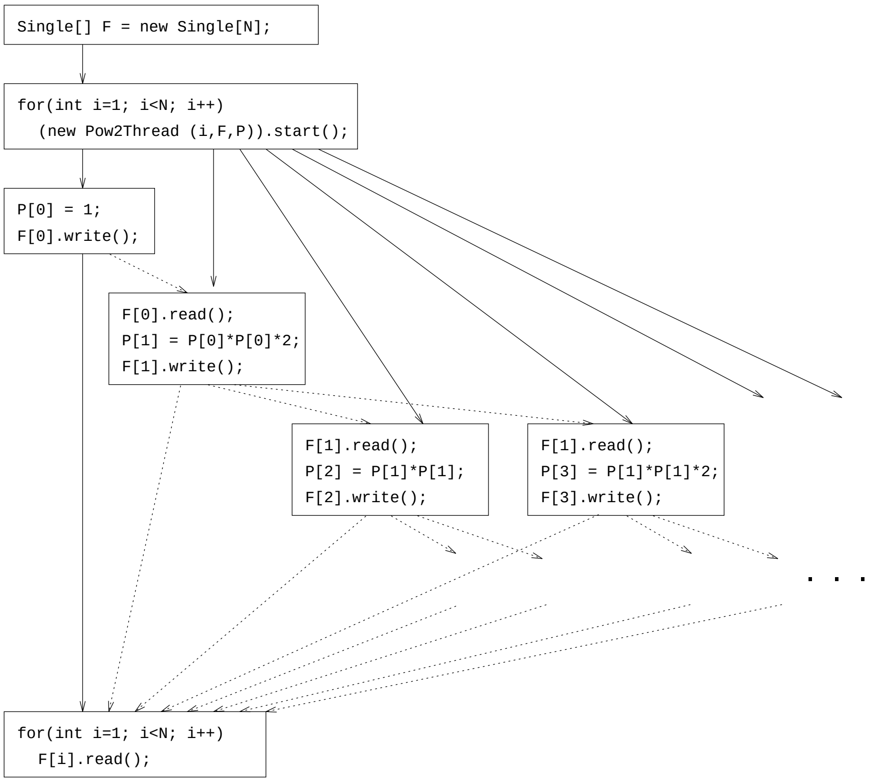

Figure 1: Effect of Single-Assignment Variables on Flow of Control

4.2 Implementation

Barrier Class

```
public class Barrier  {
  private int size;
  private int initiated = 0;

  public Barrier (int s)  {
    if (s >= 1) size = s;
    else        size = 1;
  }

  public synchronized void meet () throws InterruptedException  {
    initiated++;
    if (initiated == size)  {
      initiated = 0;
      notifyAll();
```

```
    }
    else wait();
  }
}
```

4.3 Verification

4.3.1 State

The state of a barrier is defined by the number of suspended meet operations. Once initiated, a meet operation either completes or suspends. Let `qMeet` be the number of suspended meet operations.

$$\texttt{iMeet} = \texttt{cMeet} + \texttt{qMeet} \tag{11}$$

4.3.2 Properties

Maximality of Progress. There cannot be more than $\texttt{size} - 1$ suspended meet operations.

$$\texttt{qMeet} < \texttt{size} \tag{12}$$

Constraint of Progress. A meet operation cannot complete unless all `size` threads have issued a meet to this barrier.

$$\texttt{cMeet} \bmod \texttt{size} = 0 \tag{13}$$

4.3.3 Correctness

By annotating the implementation with assertions and ghost variables, the assertions (11) - (13) can be shown to be invariantly true. Note that the assertions need not hold inside synchronize actions (which are defined to be atomic by Java semantics).

Annotated Program for Constructor

public Barrier (int s) {

  //**assert:** $\texttt{iMeet} = \texttt{cMeet} = \texttt{qMeet} = 0$
  //**assert:** (11)

  if (s >= 1) {

    //**assert:** $\texttt{iMeet} = \texttt{cMeet} = \texttt{qMeet} = 0$
    //**assert:** s >= 1
    //**assert:** (11)

    size = s;

```
        //assert: iMeet = cMeet = qMeet = 0
        //assert: size >= 1
        //assert: (11) - (13)
    }
    else {

        //assert: iMeet = cMeet = qMeet = 0
        //assert: (11)

        size = 1;

        //assert: iMeet = cMeet = qMeet = 0
        //assert: size = 1
        //assert: (11) - (13)
    }

    //assert: iMeet = cMeet = qMeet = 0
    //assert: size >= 1
    //assert: (11) - (13)
}
```

Annotated Program for Meet

```
public synchronized meet () throws InterruptedException {

    //assert: (11) - (13)

    //iMeet++

    //assert: iMeet= cMeet+ qMeet+1
    //assert: (12) - (13)

    initiated++
    //qMeet++

    //assert: qMeet<= size
    //assert: (11), (13)

    if (initiated == size) {

        //assert: qMeet= size
        //assert: (11), (13)

        initiated = 0;

        //assert: qMeet= size
        //assert: iMeet= cMeet+ size
        //assert: (11), (13)

        notifyAll();
        //qMeet= 0
        //cMeet= iMeet

        //assert: qMeet= 0
        //assert: cMeet= iMeet
```

```
        //assert: (11) - (13)
    }
    else {

        //assert: qMeet ≠ size
        //assert: (11) - (13)

        wait()

        //assert: (11) - (13)
    }

    //assert: (11) - (13)
}
```

4.3.4 Proof of Specification

Now the specification (10) follows from the conjunction of the invariants.

Proof of (10)

$$
\begin{aligned}
& \texttt{cMeet} = (\texttt{iMeet} \text{ div } \texttt{size}) \times \texttt{size} \\
\Leftrightarrow \quad & \quad \{ \text{ property of mod } \} \\
& \texttt{iMeet} - \texttt{cMeet} = \texttt{iMeet} \bmod \texttt{size} \\
\Leftrightarrow \quad & \quad \{ \text{ by (11) } \} \\
& \texttt{qMeet} = (\texttt{qMeet} + \texttt{cMeet}) \bmod \texttt{size} \\
\Leftrightarrow \quad & \quad \{ \text{ by (12) } \} \\
& \texttt{qMeet} \bmod \texttt{size} = (\texttt{cMeet} + \texttt{qMeet}) \bmod \texttt{size} \\
\Leftrightarrow \quad & \quad \{ \text{ by (13) } \} \\
& \textbf{true} \quad \square
\end{aligned}
$$

4.4 Example: Even-Odd Transposition Sort

A list of numbers can be sorted by repeatedly comparing all adjacent pairs. In the first phase, the i^{th} and $(i+1)^{th}$ elements are compared for all even i. In the second phase, the i^{th} and $(i+1)^{th}$ elements are compared for all odd i. If there are N numbers to be sorted, $N/2$ iterations of the two phases are required. In each phase, the comparisons are independent and can be carried out concurrently.

```
class EvenOddThread extends Thread {
  private int[] prob;
  private int id;
  private Barrier barrier, done;

  public EvenOddThread (int[] unsorted, int i, Barrier b, Barrier d)  {
```

```
    prob = unsorted;
    id = i;
    barrier = b;
    done = d;
  }

  public void run ()  {
    for (int i=0; i<prob.length/2; i++)  {
      if ((id > 0) && (prob[id-1] > prob[id])) swap (id-1, id);
      barrier.meet();
      if ((id < prob.length-1) && (prob[id] > prob[id+1])) swap (id, id+1);
      barrier.meet();
    }
    done.meet();
  }
} //end EvenOddThread

class EvenOdd  {
  static void Sort (int[] list)  {
    Barrier barrier = new Barrier((list.length+1)/2);
    Barrier finish = new Barrier((list.length+1)/2 + 1);
    for (int i=0; i<(list.length+1)/2; i++)
      (new EvenOddThread(list, i*2, barrier, finish)).start();
    finish.meet();
  }
} //end EvenOdd
```

5 Locks

5.1 Specification

A lock is an indivisible unit which, at any point in a computation, can be held by at most one thread of control. Java provides lock-based mutual exclusion through the `synchronized` key-word. The scope of the possession of these locks, therefore, is defined by the scope of the `synchronized` block. It is not possible, for example, to acquire a Java object lock in one method and release it in another. Nor is it possible to conditionally release a lock. We implement a generic lock class whose possession is not restricted to the scope of a block of code.

There are two operations on a lock: acquire and release. Let `cAcquire` (`cRelease`) be the number of completed acquire (release) operations. Because a lock must be acquired before it can be released, but can only be acquired by a single thread of control at a time, we have the following safety property:

$$\mathtt{cRelease} \leq \mathtt{cAcquire} \leq \mathtt{cRelease} + 1 \qquad (14)$$

Because an acquire operation can suspend, we distinguish between the number of such operations that have been initiated (`iAcquire`) and the number that have been completed (`cAcquire`). When a lock is available (*i.e.* it has been released as many times as it has been acquired), all initiated acquire operations must have completed:

$$(\texttt{cAcquire} = \texttt{cRelease}) \quad \Rightarrow \quad (\texttt{cAcquire} = \texttt{iAcquire}) \tag{15}$$

A final requirement on locks is that they be secure. That is, only the thread which has acquired the lock most recently is allowed to release it. We define `Acq` (`Rel`) to be the thread which has acquired (released) the lock most recently. When a lock is available, the last thread to release the lock must be the same as the last thread to acquire it.

$$(\texttt{cAcquire} = \texttt{cRelease}) \quad \Rightarrow \quad (\texttt{Acq} = \texttt{Rel}) \tag{16}$$

5.2 Implementation

Lock Class

```
public class Lock  {
  private Thread owner;
  private boolean free = true;
  private int suspended = 0;

  public synchronized void acquire () throws InterruptedException {
    if (!free)  {
      suspended++;
      wait();
      owner = Thread.currentThread();
    }
    else  {
      free = false;
      owner = Thread.currentThread();
    }
  }

  public synchronized void release () throws OwnerException {
    if ((Thread.currentThread() != owner) || (free))
      throw new OwnerException();
    if (suspended > 0)  {
      suspended--;
      notify();
      owner = NULL;
    }
    else
      free = true;
  }
}
```

5.3 Verification

5.3.1 State

The state of a lock is defined by:

1. Whether or not a lock is free. A lock is free exactly when it has been released as many times as it has been acquired.

$$\mathtt{free} \quad \Leftrightarrow \quad (\mathtt{cRelease} = \mathtt{cAcquire}) \tag{17}$$

2. The identity of last owner of the lock (`owner`). The last owner of the lock is the last thread which completed an acquire operation.

$$\mathtt{owner} \quad = \quad \mathtt{Acq} \tag{18}$$

3. The number of suspended acquire operations. Once initiated, an acquire operation either completes or suspends. Let `qAcquire` be the number of suspended acquire operations.

$$\mathtt{iAcquire} \quad = \quad \mathtt{qAcquire} + \mathtt{cAcquire} \tag{19}$$

Initially, the lock is free and there are no initiated acquire or release operations.

5.3.2 Properties

Maximality of Progress. There can be no pending acquires for a lock that is available. That is, either a lock is not available, or there are no pending acquires.

$$\neg\mathtt{free} \quad \vee \quad (\mathtt{qAcquire} = 0) \tag{20}$$

Security of Ownership. Because only the owner can release a lock, if the lock is free, the last thread to release the lock was also the last thread to own the lock.

$$\mathtt{free} \quad \Rightarrow \quad \mathtt{Rel} = \mathtt{owner} \tag{21}$$

5.3.3 Correctness

By annotating the implementation with assertions and ghost variables, the assertions (17) - (21) and the specification (14) can be shown to be invariantly true. Note that the assertions need not hold inside synchronize actions (which are defined to be atomic by Java semantics).

Annotated Program for Acquire

```
public synchronized void acquire () throws InterruptedException {
    //assert: (14),(17)-(21)

    //iA++
```

```
//assert: iAcquire = qAcquire + cAcquire + 1
//assert: (14), (17), (18), (20), (21)

if (!free) {

    //assert: ¬free
    //assert: iAcquire = qAcquire + cAcquire + 1
    //assert: (14), (17), (18), (20), (21)

    suspended++;
    //qA++

    //assert: ¬free
    //assert: (14), (17)-(21)

    wait();

    //assert: (14), (17)-(21)

    owner = Thread.currentThread();
    //Acq= Thread.currentThread()

    //assert: (14), (17)-(21)

}
else {

    //assert: free
    //assert: iAcquire = qAcquire + cAcquire + 1
    //assert: cRelease = cAcquire
    //assert: (14), (17), (18), (20), (21)

    free = false;

    //assert: ¬free
    //assert: iAcquire = qAcquire + cAcquire + 1
    //assert: cRelease = cAcquire
    //assert: (14), (18), (20), (21)

    owner = Thread.currentThread();
    //Acq= Thread.currentThread()

    //assert: ¬free
    //assert: iAcquire = qAcquire + cAcquire + 1
    //assert: cRelease = cAcquire
    //assert: (14), (18), (20), (21)

    //cA++

    //assert: ¬free
    //assert: cRelease + 1 = cAcquire
    //assert: (14), (17)-(21)
}
```

```
    //assert: (14), (17)-(21)
}
```

Annotated Program for Release

```
public synchronized void release () throws OwnerException {

    //assert: (14), (17)-(21)

    //Rel = Thread.currentThread()

    //assert: (14), (17)-(20)

    if ((Thread.currentThread() != owner) || (free)) {

        //assert: (Rel ≠ owner) ∨ (free)
        //assert: (14), (17)-(20)

        throw new OwnerException();

    }

    //assert: (Rel = owner) ∧ (¬free)
    //assert: (14), (17)-(21)

    if (suspended > 0) {

        //assert: qAcquire > 0
        //assert: (Rel = owner) ∧ (¬free)
        //assert: cAcquire = cRelease + 1
        //assert: (14), (17)-(21)

        suspended--;
        notify();
        //qAcquire--
        //cAcquire++

        //assert: (Rel = owner) ∧ (¬free)
        //assert: cAcquire = cRelease + 2
        //assert: (17)-(21)

        owner = NULL;
        //Acq= NULL

        //assert: ¬free
        //assert: cAcquire = cRelease + 2
        //assert: (17)-(21)

        //cRelease++

        //assert: ¬free
        //assert: cAcquire = cRelease + 1
        //assert: (14),(17)-(21)

    }
```

```
else {

    //assert: qAcquire = 0
    //assert: (Rel = owner) ∧ (¬free)
    //assert: cAcquire = cRelease + 1
    //assert: (14), (17)-(21)

    free = true;

    //assert: qAcquire = 0
    //assert: (Rel = owner) ∧ free
    //assert: cAcquire = cRelease + 1
    //assert: (14), (18)-(21)

    //cR++

    //assert: qAcquire = 0
    //assert: (Rel = owner) ∧ free
    //assert: cAcquire = cRelease
    //assert: (14), (17)-(21)
  }
}
```

5.3.4 Proof of Specification

Now the specification (14) follows directly from the annotation of the program, while the specifications (15) and (16) follow from the conjunction of the invariants.

Proof of (15)

$$
\begin{array}{ll}
 & (\mathtt{cAcquire} = \mathtt{cRelease}) \Rightarrow (\mathtt{cAcquire} = \mathtt{iAcquire}) \\
\Leftrightarrow & \quad \{ \text{ by } (17) \ \} \\
 & \mathtt{free} \Rightarrow (\mathtt{cAcquire} = \mathtt{iAcquire}) \\
\Leftrightarrow & \quad \{ \text{ by } (19) \ \} \\
 & \mathtt{free} \Rightarrow (\mathtt{qAcquire} = 0) \\
\Leftrightarrow & \quad \{ \text{ predicate logic } \} \\
 & \neg\mathtt{free} \vee (\mathtt{qAcquire} = 0) \\
\Leftrightarrow & \quad \{ \text{ by } (20) \ \} \\
 & \mathbf{true} \quad \Box
\end{array}
$$

Proof of (16)

$$
\begin{array}{ll}
 & (\mathtt{cAcquire} = \mathtt{cRelease}) \Rightarrow (\mathtt{Acq} = \mathtt{Rel}) \\
\Leftrightarrow & \quad \{ \text{ by } (17) \ \} \\
 & \mathtt{free} \Rightarrow (\mathtt{Acq} = \mathtt{Rel})
\end{array}
$$

$\Leftrightarrow \qquad \{ \text{ by } (18) \}$

$\quad \texttt{free} \Rightarrow (\texttt{owner} = \texttt{Rel})$

$\Leftrightarrow \qquad \{ \text{ by } (21) \}$

$\quad \textbf{true} \quad \Box$

5.4 Example: Finding the Minimum Element of an Array

The minimum element of an array can be found by comparing each element in turn with the smallest value seen to date. Regardless of the order in which these comparisons are performed, the result will be the same (min is associative). If the smallest value seen to date is protected by a lock to guarantee mutually exclusive access, then the threads responsible for comparing and modifying this value can be executed concurrently.

```
class FindMinThread extends Thread  {
  private int[] A;
  private int low;
  private int hi;
  private MinVal global_min;
  private Lock mutex;

  public FindMinThread (int[] Problem, int lower, int upper, MinVal m,
                        Lock guard)  {
    A = Problem;
    low = lower;
    hi = upper;
    global_min = m;
    mutex = guard;
  }

  public void run ()  {
    int my_min = A[low];
    for (int i=low+1; i<=hi; i++) {
      if (A[i] < my_min)
        my_min = A[i];
    }
    try  { mutex.acquire(); }
    catch (InterruptedException e)  { System.exit(1); }
    if (my_min < global_min.value)
      global_min.value = my_min;
    try  { mutex.release(); }
    catch (OwnerException e)  { System.exit(1); }
  }

}

class FindMin  {
  static int BlockSize = 4;

  static int Solve (int[] Problem)  {
    MinVal solution = new MinVal (Problem[0]);
```

```
    FindMinThread[] Tasks =
      new FindMinThread[((Problem.length-1) / BlockSize) + 1];
    Lock mutex = new Lock();

    int i;
    int taskid = 0;
    for (i=BlockSize-1; i<Problem.length; i=i+BlockSize)  {
      Tasks[taskid] = new FindMinThread (Problem, i-BlockSize+1, i,
                                         solution, mutex);
      Tasks[taskid].start();
      taskid++;
    }
    if (i-BlockSize < Problem.length-1)
      Tasks[Tasks.length-1] = new FindMinThread (Problem, i-BlockSize+1,
                                         Problem.length-1, solution, mutex);
    for (i=0; i<Tasks.length; i++)  {
      try  { Tasks[i].join(); }
      catch (InterruptedException e)  { System.exit(1); }
    }
    return solution.value;
  }
}
```

6 Semaphores

6.1 Specification

A semaphore has an integer value which is invariantly non-negative. Two operations are defined for a semaphore: P, and V. The V operation atomically increments this value. It never suspends. The P operation attempts to decrement the semaphore's value. If the value of the semaphore is greater than 0, then the decrement is performed and the P operation terminates. If, on the other hand, the value of the semaphore is 0 (and therefore a decrement would violate the invariant that the value be non-negative), the P operation suspends until the decrement may be safely performed. Semaphores were introduced in [5] and are further discussed in [7] and [2, Chapter 4].

Since the P operation is not atomic, we distinguish between the initiation and the termination of this operation. Let `iP` be the number of P operations that have been initiated, and let `cP` be the number of P operations that have completed. Since the V operation is atomic, no such distinction is required and we define `cV` to be the number of completed V operations.

The initial value of the semaphore is defined to be s_0 where $s_0 \geq 0$. The safety condition is that as many P operations as possible will complete, subject to the constraints of the number of initiated P operations and the sum of the initial value of the semaphore and the number of completed V operations.

$$\mathtt{cP} \quad = \quad \min(\mathtt{iP}, s_0 + \mathtt{cV}) \tag{22}$$

6.2 Implementation

Semaphore Class

```
package semaphore;
import semaphore.NegativeSemException;

public class Semaphore {
  private int value;
  private int suspended;

  public Semaphore ()  {
    value = 0;
    suspended = 0;
  }

  public Semaphore (int v) throws NegativeSemException  {
    if (v < 0)
      throw new NegativeSemException("Requested initial semaphore value: "+v);
    value = v;
    suspended = 0;
  }

  public synchronized void V ()  {
    if (suspended == 0)
      value++;
    else  {
      suspended--;
      notify();
    }
  }

  public synchronized void P() throws InterruptedException {
    if (value > 0)
      value--;
    else  {
      suspended++;
      wait();
    }
  }
}
```

NegativeSemException Class

```
package semaphore;

public
class NegativeSemException extends Exception {
  public NegativeSemException()  {
    super();
  }

  public NegativeSemException(String s)  {
    super(s);
  }
}
```

6.3 Verification

6.3.1 State

The state of a semaphore is defined by:

1. The value of the semaphore, s. Since each completed P operation decrements this value and each completed V operation increments this value, we have:

$$s = s_0 - \mathtt{cP} + \mathtt{cV} \tag{23}$$

2. The queue of suspended P operations. The number of suspended P operations, $\mathtt{qP}$, is the number of P operations which have been initiated, but have not yet completed.

$$\mathtt{iP} = \mathtt{qP} + \mathtt{cP} \tag{24}$$

6.3.2 Properties

Boundedness Requirement. The number of P operations that can complete is bounded by the initial value of the semaphore and the number of V operations that have completed.

$$\mathtt{cP} \leq s_0 + \mathtt{cV} \tag{25}$$

The Set of Suspended Processes is Minimal. A process can be suspended only if its completion would violate the safety condition given by (22).

$$(\mathtt{qP} = 0) \quad \vee \quad (s = 0) \tag{26}$$

6.3.3 Correctness

By annotating the implementation with assertions and ghost variables, the assertions (23) - (26) can be shown to be invariantly true. Note that the assertions need not hold inside synchronize actions (which are defined to be atomic by Java semantics).

Annotated Program for V

```
public synchronized void V () {

    //assert: (23) - (26)

    if (suspended == 0) {

        //assert: qP = 0
        //assert: (23) - (26)
```

```
        value++;
        //s++

        //assert: qP = 0
        //assert: s = s0 - cP + cV + 1
        //assert: (24) - (26)

    }
    else {

        //assert: qP > 0
        //assert: (23) - (26)

        suspended--;
        notify();
        //qP--
        //cP++

        //assert: s = s0 - cP + cV + 1
        //assert: cP <= s0 + cV + 1
        //assert: (24), (26)

    }

    //cV++

    //assert: (23) - (26)
}
```

Annotated Program for P

```
public synchronized void P() throws InterruptedException {

    //assert (23) - (26)

    //iP++

    //assert iP = qP + cP + 1
    //assert (23), (25), (26)

    if (value > 0) {

        //assert s > 0
        //assert qP = 0
        //assert iP = qP + cP + 1
        //assert (23), (25), (26)

        value--;
        //s--

        //assert qP = 0
        //assert s = s0 - cP + cV - 1
        //assert iP = qP + cP + 1
        //assert (25), (26)
```

```
        //cP++

        //assert qP = 0
        //assert (23) - (26)

    }
    else {

        //assert s = 0
        //assert iP = qP + cP + 1
        //assert (23), (25), (26)

        suspended++;
        //qP++

        //assert s = 0
        //assert (23) - (26)

        wait();

        //assert (23) - (26)

    }
}
```

6.3.4 Proof of Specification

Now the specification (22) follows from the conjunction of the invariants.

Proof of (22)

true

$\Leftrightarrow$ { by (26) }

$(\mathtt{qP} = 0) \vee (s = 0)$

$\Leftrightarrow$ { by (26) }

$(\mathtt{cP} = \mathtt{iP}) \vee (\mathtt{cP} = s_0 + \mathtt{cV})$

$\Leftrightarrow$ { by (23), (24) }

$((\mathtt{cP} = \mathtt{iP}) \vee (\mathtt{cP} = s_0 + \mathtt{cV})) \wedge (\mathtt{cP} \leq s_0 + \mathtt{cV})$

$\Leftrightarrow$ { property of min }

$\mathtt{cP} = \min(\mathtt{iP}, s_0 + \mathtt{cV})$ □

6.4 Example: Finding the Minimum Element of an Array

```
class Database  {
  private int nr;             //number of readers
  private Semaphore mutexR;   //mutual exclusion between readers
  private Semaphore rw;       //mutual exclusion for accessing database
```

```
  public Database ()  {    //mutexR and rw implicitly initialized to 1
   nr = 0;
   try  {
     mutexR = new Semaphore (1);
     rw = new Semaphore (1);
   } catch (NegativeSemException e) { System.exit(1); }
  }

  public void reader ()  {
    try { mutexR.P(); }
    catch (InterruptedException e) { System.exit(1); }
      nr += 1;
      if (nr == 1)
        try { rw.P(); }
        catch (InterruptedException e) { System.exit(1); }
    mutexR.V();
    //read the database
    try { mutexR.P(); }
    catch (InterruptedException e) { System.exit(1); }
      nr -= 1;
      if (nr == 0) rw.V();
    mutexR.V();
  }

  public void writer ()  {
    try { rw.P(); }
    catch (InterruptedException e) { System.exit(1); }
    //write the database
    rw.V();
  }

} //end Database class
```

7 Conclusions

We have specified, implemented, verified, and illustrated a library implementation of two traditional synchronization primitives. Widespread use of distributed systems requires a reliable collection of communication and synchronization paradigms that can be tailored to different applications. Although the formal methods required to guarantee this reliability are expensive, this cost is amortized over the large number of distributed applications which will be layered above these paradigms. We have demonstrated the viability of this approach in the context of single-address space concurrency. These results form one component of a larger model for a general-purpose distributed system, which is currently under investigation.

References

[1] W. B. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, February 1982.

[2] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, CA, 1991.

[3] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming language. Technical Report CS-TR-92-01, Computer Science Department, California Institute of Technology, 1992.

[4] K.M. Chandy, A. Rifkin, P.A.G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A world-wide distributed system using java and the internet. In *Proceedings of the Fifth Workshop on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[5] E. W. Dijkstra. Cooperating sequential processes. Technical report, Technological University, Eindhoven, 1965.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[7] A. J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.

[8] P. A. G. Sivilotti and P. A. Carlin. A tutorial for CC++. Technical Report CS-TR-94-02, Computer Science Department, California Institute of Technology, 1994.

[9] John W. Thornley. *A Parallel Programming Model With Sequential Semantics*. PhD thesis, California Institute of Technology, Pasadena, CA, May 1996.