```
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
```

4 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] Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, February 1982.

[2] Gosling, Joy, and Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[3] Sivilotti and Chandy. Reliable synchronization primitives for Java threads. Technical Report CS-TR-96-11, Computer Science Dept., California Institute of Technology, 1996.

3.2 Implementation

```
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();
  }
}
```

3.3 Verification

By annotating the implementation with assertions and ghost variables, the following assertions can be shown to be invariantly true. Let `qMeet` be the number of suspended meet operations.

$$\texttt{cMeet} \bmod \texttt{size} = 0 \tag{11}$$

$$\texttt{iMeet} = \texttt{cMeet} + \texttt{qMeet} \tag{12}$$

$$\texttt{qMeet} < \texttt{size} \tag{13}$$

Proof of (10)

$\texttt{cMeet} = (\texttt{iMeet} \text{ div } \texttt{size}) \times \texttt{size}$

$\Leftrightarrow$ { property of mod }

$\texttt{iMeet} - \texttt{cMeet} = \texttt{iMeet} \bmod \texttt{size}$

$\Leftrightarrow$ { by (12) }

$\texttt{qMeet} = (\texttt{qMeet} + \texttt{cMeet}) \bmod \texttt{size}$

$\Leftrightarrow$ { by (13) }

$\texttt{qMeet} \bmod \texttt{size} = (\texttt{qMeet} + \texttt{cMeet}) \bmod \texttt{size}$

$\Leftrightarrow$ { by (11) }

true □

3.4 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.

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
```

3 Reusable Barriers

3.1 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:

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

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

2.3 Verification

By annotating the implementation with assertions and ghost variables, the following assertions can be shown to be invariantly true. Let `qRead` be the number of suspended read operations, and let `defined` be true exactly when the variable is defined.

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

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

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

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

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

Note that these invariants need not hold inside synchronize actions (which are defined to be atomic by Java semantics). The conjunction of the specifications (1) - (3) follows from the conjunction of the invariants (4) - (8). For example, (3) follows from (5), (6), and (7):

Proof of (3)

$\texttt{cWrite} = 1$

$\Rightarrow$ { by (5) }

$\texttt{defined}$

$\Rightarrow$ { by (7) }

$\texttt{qRead} = 0$

$\Leftrightarrow$ { by (6) }

$\texttt{cRead} = \texttt{iRead}$ □

2.4 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

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, semaphores, bounded semaphores, and bounded buffers. These libraries have been formally specified and verified [3].

To illustrate our methodology, this paper presents two of these libraries. Section 2 describes a library for single-assignment variables — its specification, formal verification, and an example of its use; Section 3 contains a similar description of a reusable barrier library; and Section 4 concludes and summarizes the results.

2 Single-Assignment Variables

2.1 Specification

Single-assignment variables have their history in data-flow programming [1]. 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. 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, while conversely, all read operations complete on a variable that has been defined:

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

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

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.

2.2 Implementation

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

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

Toward High-Confidence Distributed Programming with Java: Reliable Thread Libraries*

Paolo A.G. Sivilotti and K. Mani Chandy

California Institute of Technology, MS 256-80, Pasadena, CA 91125

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 paper describes a part of a project that extends the Web as a general-purpose distributed system.

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 [2] 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

*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.