ThreadMentor: A Pedagogical Tool for Multithreaded Programming

STEVE CARR, JEAN MAYO, and CHING-KUANG SHENE
Michigan Technological University

ThreadMentor is a multi-platform pedagogical tool designed to ease the difficulty in teaching and learning multithreaded programming. It consists of a C++ class library and a visualization system. The class library supports many thread management functions and synchronization primitives in an object-oriented way, and the visualization system is activated automatically by a user program and shows the inner working of every thread and every synchronization primitive on-the-fly. Events can also be saved for playback. In this way, students will be able to visualize the dynamic behavior of a threaded program and the interaction among threads and synchronization primitives.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*

General Terms: threads, synchronization, visualization

Additional Key Words and Phrases: multithreaded programming, synchronization primitives

1. INTRODUCTION

The capability of multithreaded programming was first commercially available in the late 60's when IBM added the task feature and completion event variables into its PL/I F compiler and made the same available in all IBM VS (*i.e.*, virtual storage) operating systems as supervisor calls. Threading became popular in the Unix community in the early 80's. Today, virtually all operating systems have multithreaded capability and the POSIX Pthreads standard is also popular. Many well-known operating systems textbooks added sections on threads in their newest editions [Silberschatz et al. 2002; Stallings 2001; Tanenbaum 2001] and numerous books about threads were published in recent years.

To ensure students can lead the trend in computer science toward multithreaded programming in the foreseeable future, we have been teaching this concept in an operating systems course for seven years [Shene 1998; 2002]. Our experience shows that the paradigm shift from sequential to multithreaded causes students significant problems [Shene and Carr 1998], such as **(1)** multithreaded programming requires a new mindset, **(2)** the behavior of a multithreaded program is dynamic, making

Authors' address: Department of Computer Science, Michigan Technological University, 1400 Townsend Drive, Houghton, MI 49931, USA. E-mail: `{carr,jmayo,shene}@mtu.edu`

debugging very difficult, **(3)** proper synchronization is more difficult than anticipated, and **(4)** programming interfaces are usually more complex than necessary, causing students to spend time in learning the system details rather than the fundamentals. Items (2) and (3) have proven to be more troublesome because students have difficulties in reconstructing the events from the output of the execution of a threaded program. Hence, it is very challenging for students to pinpoint mistakes and potential problems (e.g., race conditions) in their programs. Moreover, since the output of a threaded program is serialized by the screen output, the dynamic behavior of the execution of a threaded program is lost. ThreadMentor has been developed to bring back and let students see the dynamic behavior of threaded programs, and to help students understand synchronization primitives.

ThreadMentor is a multi-platform pedagogical tool designed to ease the difficulty in teaching and learning multithreaded programming. It runs on multiple platforms (e.g., Windows, Linux and Sun Solaris) and consists of two components: a class library and a visualization system. These two components are integrated together seamlessly and a user does not have to instrument his/her program to obtain the visualization. The visualization can be real-time or post-mortem and supports all thread management (e.g., creation, termination, joining, yielding, suspension and resume) and almost all commonly used synchronization primitives (e.g., mutex locks, semaphores, Hoare and Mesa monitors, barriers, reader-priority and writers-priority readers-writers locks, and synchronous and asynchronous one-to-one, many-to-one and many-to-many channels). Additionally, a topology editor allows students to design the topology of communication links among threads and generate the necessary program code for channel construction. With the topology editor, students can concentrate on problem solving rather than the usually messy and tedious channel construction task. Moreover, ThreadMentor also includes a portable user-level kernel mtuThread that supports non-preemptive thread scheduling for students to learn the internals of thread dispatching and thread management. ThreadMentor's emphasis is on the low-level events rather than algorithm animation because we believe that these low-level activities are the most fundamental parts that students have difficulty comprehending and are the most important factors needing visualization. Therefore, ThreadMentor provides students with a coherent and unified platform for students to learn multithreaded programming in operating systems and/or concurrent programming courses.

In this paper, we shall provide an overview of the design philosophy and some important features of ThreadMentor. In the following, Section 2 contains a review of related work; Section 3 discusses some design issues, and advantages and disadvantages of our approach; and Section 4 presents a general system overview. Section 5 to Section 7 contain a detailed discussion of ThreadMentor's support of semaphores, monitors and channels. Although ThreadMentor supports more synchronization primitives, only semaphores, monitors and synchronous one-to-one channels will be discussed. The user-level kernel mtuThread and the topology editor are discussed in [Bedy et al. 1999] and [Carr et al. 2002], respectively, and will not be repeated here. Some preliminary results were also reported in [Bedy et al. 2000; Carr and Shene 2000] and a tutorial is available in [Shene 2001]. Finally, Section 8 has our conclusions.

2. PREVIOUS WORK

Many papers have been published in the SIGCSE Technical Symposium and ITiCSE Conference on multithreaded, multiprocess, parallel and distributed computing. These works include: socket-based message-passing libraries [Arnow 1995; Toll 1995b], various modifications to Ben-Ari's well-known Pascal compiler [Ben-Ari 1982; Bynum and Camp 1996; Kurtz et al. 1998; Persky and Ben-Ari 1998], PVM-based parallel computing [Jin and Yang 1995a; McDonald and Kazemi 1997], a data-parallel C++ library [Kotz 1995], Java-based concurrent programming [Hartley 1998; 1999; 2000; 2001], synchronization [Choi and Lewis 2000; Reek 2002; Robbins 2000; 2001; 2002], multithreaded programming [Berk 1996], parallel algorithms [Jin and Yang 1995b; McDonald and Kazemi 2000; Luque et al. 1996; Naps and Chan 1999], parallel computing labs [Elenbogen 1996; Harlan 1995] and other education related issues [Adams et al. 2000; Ben-Ari 1996; Burkhart 1997; Cunha and ao Lourenço 1998; Kolikant et al. 2000; Kurtz and Alsabbagh 1998; Kwiatkowski et al. 1996; Pollock and Jochen 2001; Toll 1995a]. There are other earlier studies also. Higginbotham and Morelli [1991] used the Unix IPC interface. Zimmermann et al. [1988] discussed their system for Portal, a language close to Modula 2. Unfortunately, a special hardware TMK (Test Machine Kernel) was used, making this system not portable. Hartley used a software technique [Hartley 1992; 1994]. The user program, written in SR [Andrews and Olson 1992], saves the activities of a program to a file and uses XTANGO [Stasko 1990] to playback *after* the program completes. Price and Baecker [1991] also discussed a framework for concurrent program animation. These works address some aspects of concurrent computing rather than providing a coherent and unified environment for students to learn multithreaded programming.

There are very few pedagogical tools for teaching threaded programming. Most tools are variations of Ben-Ari's Pascal compiler (see also Burns and Davies [1993]), focusing mainly on running threads or processes under the control of an interpreter with various kinds of synchronization primitives. Lester's Multi-Pascal [Lester 1993] is also an interpreter with limited parallel computing capability. There are some Java-based tools for distributed algorithms [Ben-Ari 1997; 2001; Schreiner 2002].

Visualizing parallel programs can be *online* or *offline*. The former generates visuals on-the-fly, while the latter saves the events and plays back with another system (i.e., post-mortem). The advantage of an offline system is that every event related to the execution of a program is saved and can be replayed at any time. However, its main disadvantages are: **(1)** it could be too late for a student to catch any bug, since it only shows one instance of the program execution; **(2)** a large volume of output may be generated that could be incomplete or even corrupted if the program ends abnormally; **(3)** the program being visualized must be "instrumented" by adding extra statements and/or directives, which could directly interfere with the behavior of the program; and **(4)** the offline system must also synchronize its own file writing activities, adding an extra level of complexity that may affect the program's original behavior. The most well-known system of this type is PARADE [Stasko 1995], which is based on POLKA. Other details can be found in [Flinn and Cowan 1990; Kraemer 1998; Stasko and Kraemer 1993a; 1993b]. Along this line of research, Zhao and Stasko [1995] designed an environment for visualizing Pthreads programs using

POLKA, and Cai et al. [1993] discussed a system for visualizing programs written in OCCAM. Other useful information may be found in [Bemmerl and Braun 1993; Chao and Liu 2000; Miller 1993; Pancake 1996; Roman et al. 1992; Zhang et al. 1999].

However, none of the above mentioned systems support multithreaded programming and its visualization tightly, and provides students with an environment for developing threaded programs and visualizing program execution and synchronization activities. Moreover, except for Zhao and Stasko [1995], none of the systems is able to reveal the low-level synchronization related information. In fact, most of the visualization systems are for performance and/or debugging rather than designed as pedagogical platforms to be used by beginners and students. Consequently, ThreadMentor is perhaps the only comprehensive pedagogical system available for teaching and learning multithreaded programming.

3. DESIGN ISSUES

The main design goal of ThreadMentor is to build a multi-platform system for helping students learn and visualize the fundamentals of multithreaded programming in concurrent programming and operating systems courses as described in unit OS3 Concurrency of CC2001 [ACM/IEEE 2001]. ThreadMentor consists of a class library and a visualization system. Both components support thread management (e.g., thread creation, termination and join) and popular synchronization primitives (e.g., mutex locks, semaphores, monitors, barriers, and synchronous and asynchronous channels). The class library uses textbook syntax so that students do not have to memorize many different parameters, and hides as much system details as possible from its users. Hence, threaded programs in ThreadMentor look very similar to textbook examples. Moreover, a user program and the visualization system run in separate address spaces, and, as a result, problems in one program will not interfere and propagate to the other.

To make ThreadMentor a multi-platform system, the class library translates many thread management and synchronization primitive calls to the corresponding system supported thread library calls, and the visualization system uses the multi-platform toolkit GTK [Mattis et al. 2002] for GUI development. As a result, ThreadMentor only supports user-level threads for maximum portability. A message queue is used to support the communication between the class library and the visualization system. In this way, ThreadMentor can support Windows, Linux and other Unix variants, and Sun Solaris using Win32 Threads, Pthreads, and Solaris threads, respectively. It also includes a small portable kernel mtuThread that supports user-level multithreaded programming [Bedy et al. 1999]. Hence, ThreadMentor can run on most popular platforms (Figure 1). Due to the multi-platform requirement and the use of GTK and other thread libraries, C/C++ is used. Java is not chosen because of its insecure parallelism [Brinch Hansen 1999]. The class library has two versions, one version with visualization enabled and the other without visualization. A student may use the version with visualization for learning and debugging, and the version without visualization for efficiency.

Although this approach provides maximum flexibility, some compromises must be made. **First**, since the class library is part of the user program that runs in

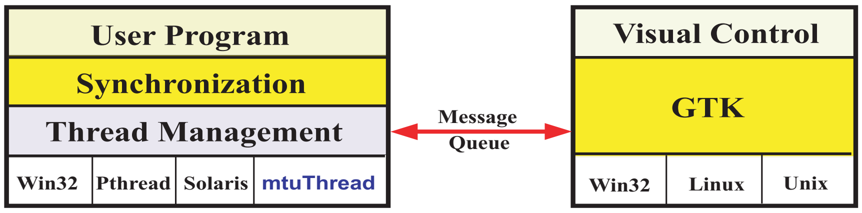


Fig. 1. ThreadMentor system design

user address space, ThreadMentor does not have access to system internal data. As a result, between the time when ThreadMentor detects a synchronization primitive activity and sends a message to the visualization system and the time the underlying system actually performs the desired operation, there is a brief delay. **Second**, since the message queue between the class library and the visualization system is an asynchronous channel with a finite capability, it is possible that the visualization display may also lag behind for another brief moment due to queued messages. Moreover, since most systems buffer screen output and since ThreadMentor has no control over system buffering, it is possible that a user program prints much faster than the activities shown in ThreadMentor's windows. **Third**, the information for a synchronization primitive maintained by ThreadMentor may not be identical to the information that is maintained by the underlying system for a very brief moment. For example, it is possible that the semaphore value recorded by ThreadMentor may be different from the one recorded by the underlying system due to system call delay. **Fourth**, after receiving a message from the class library, the visualization system takes time to update *all* of its windows. These four factors may cause the program output and the content of each ThreadMentor's window to be not fully synchronized. Fortunately, these have not been serious problems on faster machines; however, students must be informed of this fact.

4. SYSTEM OVERVIEW

The most important class in the class library is class `Thread`. A student defines a thread as a derived class of `Thread` and supplies a method `ThreadFunc()` as the thread body. Class `Thread` includes methods `Begin()` for executing a created thread, `Exit()` for terminating a running thread, `Join()` for joining with another thread, `Yield()` for relinquishing the execution to another thread, `Suspend()` for suspending a thread, and `Resume()` for resuming the execution of a suspended thread. A created thread does not run until its `Begin()` is called. Method `Delay()`, which is implemented by calling method `Yield()` to execute a random number of context switches, may be used for delaying its caller. A user may use a constructor for other initialization tasks (e.g., giving the thread a symbolic name). Figure 2, which implements the quicksort, provides a general feeling of how class `Thread` is used. After partitioning, each quicksort thread creates two child threads with `new`, runs them with `Begin()`, waits for their completion with `Join()`, and exits with `Exit()`.

Since all visualization activities are monitored and controlled within the class library, a user does not have to alter his/her program in order to generate visu-

```
#include "ThreadClass.h"

class QuickSortThread : public Thread
{
     public:
          QuickSortThread(int Lower, int Upper, int Input[]);
     private:
          void ThreadFunc();  // thread body
          int  lower, upper;  // lower & upper bounds
          int  *a;            // pointing to array to be sorted
};

void QuickSortThread::ThreadFunc()
{
     Thread::ThreadFunc();    // required!
     QuickSortThread  *leftThread, *rightThread;
     int   pivotIndex;
     ..........
     if (lower >= upper)
          Exit();             // nothing to be sorted
     pivotIndex = Partition(lower, upper, a); // partition
     Swap(&a[pivotIndex], &a[lower]); // swap
     leftThread = new QuickSortThread(lower, pivotIndex-1, a);
     leftThread->Begin();     // sort the left portion
     rightThread = new QuickSortThread(pivotIndex+1, upper, a);
     rightThread->Begin();    // sort the right portion
     leftThread->Join();      // wait for the child threads
     rightThread->Join();
     Exit();                  // done and exit
}

void main(int argc, char *argv[])
{
     QuickSortThread *quicksortthread;
     int             n, *a;
     // read the input of n integers into a[]
     quicksortthread = new QuickSortThread(0, n-1, a);
     quicksortthread->Begin();  // start the thread
     quicksortthread->Join();   // wait for its completion
     // print the sorted array
     Exit();                    // done and exit.
}
```

Fig. 2. A typical way of using class Thread

alization. If a program is linked with the visualization enabled class library, the visualization system will be brought up automatically. The visualization system is basically an event-driven system. Methods of the class library know what events are important and send the events to the visualization system through a message queue. The visualization system receives the events from the message queue and displays them in various windows on-the-fly. However, a user may also save all events for post-mortem display, and, in this case, the visualization system works as a stand-alone post-mortem system.

When the visualization system is activated by a user program, it displays the Main Window as shown in Figure 3, which is the entry-point of all features supported by ThreadMentor. The large white area, called the *display area*, is used for showing the first-level information of the selected primitive. The display area in Figure 3 shows `History Graph` and `Thread Hierarchy`, meaning the History Graph Window and the Thread Hierarchy Window are selected. The buttons on the right side of the Main Window permit a user to select synchronization primitives. For example, if `Semaphore` is selected, the display area will show *all* semaphores created so far in the running program. A click on the name of a semaphore brings up the second-level, detailed information of the selected semaphore. Currently, the supported synchronization primitives include mutex locks, semaphores, monitors, readers-writers locks, barriers, and synchronous and asynchronous channels. The lower-right corner of the Main Window has a speed-bar and buttons for a user to control the execution speed, pause and resume the execution of the running program, and step through the thread management and synchronization activities. Since ThreadMentor does not have access to compiler generated data due to its multi-platform nature, it cannot step through statements.

The lower-left corner of the Main Window has three thread management related buttons. The Thread Hierarchy Window displays the parent-child family tree and the current state of every thread. Figure 4 is a snapshot of the Thread Hierarchy Window while the quicksort program is running. A thread in this window has a symbolic name, assigned by the user, of form `Sorting(a:b)`, meaning this sorting thread sorts the array portion with lower bound `a` and upper bound `b`. From this figure, we learn the following (1) the main program receives eight integers to be sorted by `Sorting(0:7)`, (2) the pivot element is the seventh and two partitions are being sorted by `Sorting(0:5)` and `Sorting(7:7)`, (3) thread `Sorting(0:5)` finds the pivot element being the fifth and the two partitions are being sorted by threads `Sorting(0:3)` and `Sorting(5:5)`, (4) thread `Sorting(0:3)` creates threads `Sorting(0:1)` and `Sorting(3:3)`, and (5) threads `Sorting(7:7)` and `Sorting(5:5)` have terminated.

The History Graph Window shows the execution history of all threads (Figure 5). In this window, each created thread has a *history bar* running from left to right. Each history bar is color coded with green, blue and red for running, joining and blocked by a synchronization primitive, respectively. Each history bar is also tagged by two-letter tags representing synchronization events, which are chronologically ordered from left-to-right. There is only one type of tag in Figure 5, the join tag `JN`, because no other synchronization primitives are being used in the quicksort program. Since a thread must be running in order to execute a join and since after the execution of a join the thread is in the joining state, to the left (resp., right) of a `JN` tag, the history bar is colored in green (resp., blue). When a tag is clicked, a Source Window appears in which the source program is displayed with the line that contains the corresponding synchronization primitive highlighted. Consequently, the History Graph Window is perhaps the most commonly used ThreadMentor window

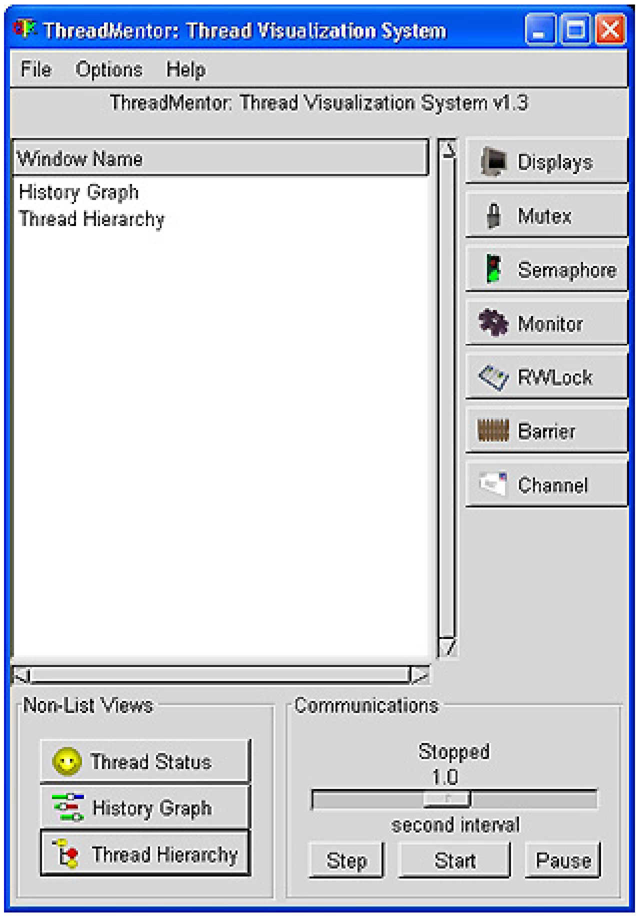


Fig. 3. ThreadMentor's Main Window

because, for every thread, it provides the state, the relative time of execution and description of a synchronization event with history bar tags, and a link between a tag and the corresponding source statement. With this window, one can easily reconstruct the events of the execution of a threaded program.

In addition to showing the events of a threaded program, these windows may also provide solid evidence of the dynamic behavior of a threaded program, since the change of states and relative execution speed on different platforms and on the same platform but with a different load mix can easily be visualized with ThreadMentor.

5. SYNCHRONIZATION PRIMITIVE: SEMAPHORES

ThreadMentor supports semaphores with class `Semaphore` and methods `Wait()` and `Signal()`. Methods that provide access to the internal data of a semaphore (e.g., retrieving the semaphore value and the number of waiting threads) are left out intentionally. In our experience, students can easily mis-use these features. For example, a student may retrieve the value of a semaphore, test it for zero, and

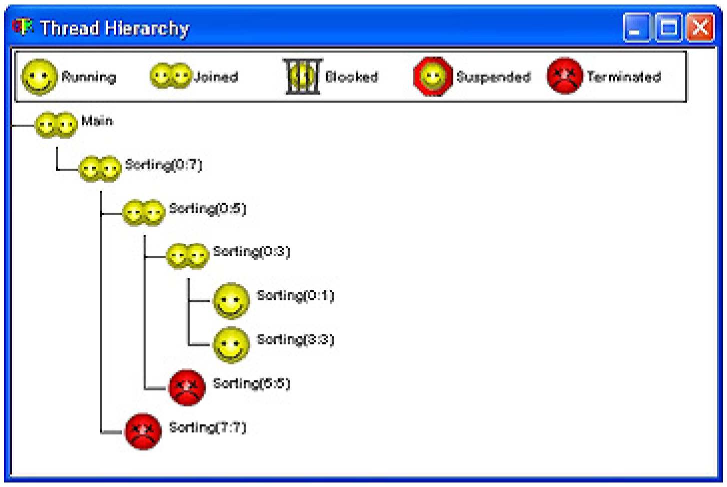


Fig. 4. ThreadMentor's Thread Hierarchy Window

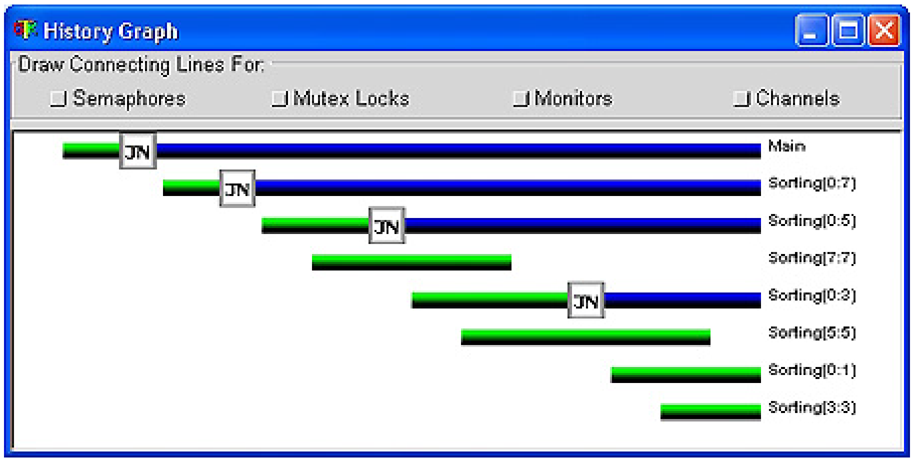


Fig. 5. ThreadMentor's History Graph Window

access a protected shared resource if the semaphore value is positive (i.e., no waiting threads). However, this can cause race conditions because after the testing of the semaphore value and before the access to the protected shared resource, other threads may call the `Wait()` method and decrease the semaphore value to zero. As a result, two or more threads may modify a shared data item at the same time. To force a proper use of semaphores, ThreadMentor removes these potential problems entirely.

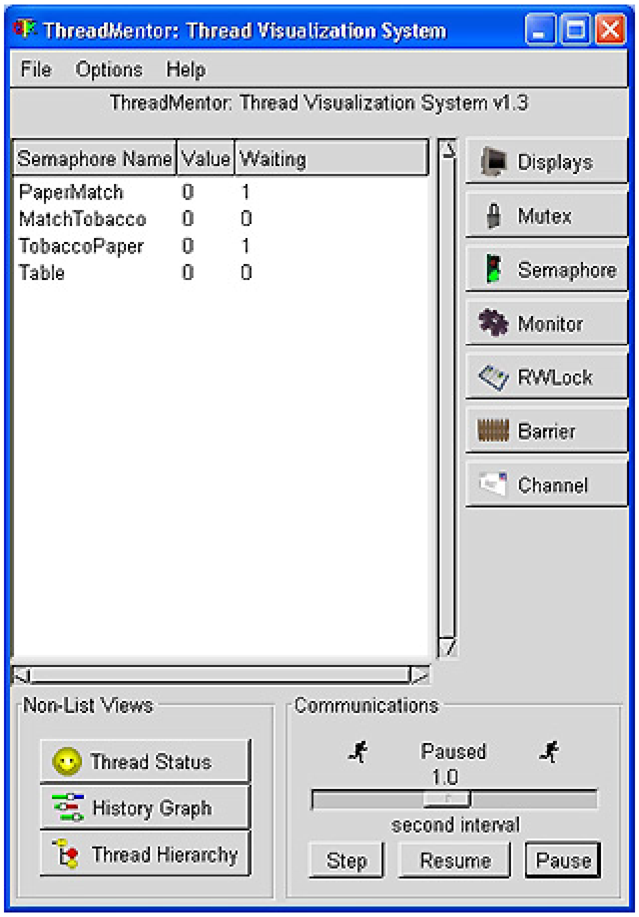
Figure 6 shows a portion of a program that solves the well-known smokers problem. Smokers who need paper and match, match and tobacco, and tobacco and paper are blocked by semaphores `PaperMatch`, `MatchTobacco` and `TobaccoPaper`, respectively. Each smoker thread waits on its semaphore for the needed ingredients, takes the ingredients when they are available, clears the table by signaling the agent, and smokes for a while simulated with method `Delay()`. The agent takes a rest, randomly generates the ingredients, signals the corresponding smoker to indicate ingredients are available, and waits for the table to be cleared. All four semaphores are initialized to zero with symbolic names.

```
#include "ThreadClass.h"

static Semaphore PaperMatch("PaperMatch", 0);
static Semaphore MatchTobacco("MatchTobacco", 0);
static Semaphore TobaccoPaper("TobaccoPaper", 0);
static Semaphore *Sem[3] = {&PaperMatch, &MatchTobacco, &TobaccoPaper};
static Semaphore Table("Table", 0);

class SmokerThread : public Thread {   ..........   }

class AgentThread  : public Thread {   ..........   }

void  SmokerThread::ThreadFunc()
{
     SmokerThread::ThreadFunc();
     for (.....) {
          Sem[ID]->Wait();  // wait for ingredients
          Table.Signal();   // clear the table
          Delay();          // smoke for a while
     }
}

void  AgentThread::ThreadFunc()
{
     AgentThread::ThreadFunc();
     for (.....) {
          Delay();         // take a rest
          i = random integer in [0..2];
          Sem[i]->Signal();// let the smoker know
          Table.Wait();    // wait for the table
     }
}
```

Fig. 6. The smokers problem

Figure 7 is a snapshot of the Main Window while the smokers program is running. The display area shows the semaphore value and number of waiting threads of each semaphore. For example, semaphore `PaperMatch` has a value of 0 and one waiting thread, while semaphore `Table` has a value of 0 but no waiting thread.

The History Graph Window in Figure 8 provides the execution history of the smoker program. There are a few more tags associated with semaphores. Tags `SS` and `SW` indicate a thread executed a `Signal()` and a `Wait()`, respectively. The event that occurs when a thread is released from a semaphore due to a `Signal()` has



Semaphore Name	Value	Waiting
PaperMatch	0	1
MatchTobacco	0	0
TobaccoPaper	0	1
Table	0	0

Fig. 7. Main Window when running the smokers program

a SE tag. Since a thread must be blocked by a Wait() before it can be released by a Signal(), the history bar of this thread before (resp., after) a SE tag is colored red (resp., green). A click on the Semaphore button in the upper-left corner of the History Graph Window will display the *signal-release links* that connect a signal event with its release event. With the History Graph Window, we learn that (1) the main program created all threads and is in the joining state (i.e., the JN tag); (2) the Agent thread signals the Smoker(Match) thread indicating the ingredients (i.e., paper and tobacco) are available, and then waits for the table (i.e., its first SW tag) to be cleared; (3) initially all smoker threads are blocked (i.e., the first SW tag on each history bar); (4) thread Smoker(Match) is released by Agent's signal (i.e., the SS-SE link), signals Agent to indicate the table is free, smokes for a while (i.e., the green portion of its history bar), and finally goes back for the next round (i.e., the second SW tag); (5) Agent is released by Smoker(Match)'s signal, generates the next pair of ingredients, signals Smoker(Paper), and waits for the table; and (6) Smoker(Paper) is released by Agent's signal, takes the ingredients, signals Agent, and smokes. Thus, currently, there are two running threads: `Agent` is preparing for the third pair of ingredients, and `Smoker(Paper)` is smoking. The other two smoker threads, `Smoker(Tobacco)` and `Smoker(Match)`, are blocked, waiting for their ingredients. Again, a click on an event tag brings up the Source Window to display the source program with the line containing the call to the corresponding synchronization event highlighted.

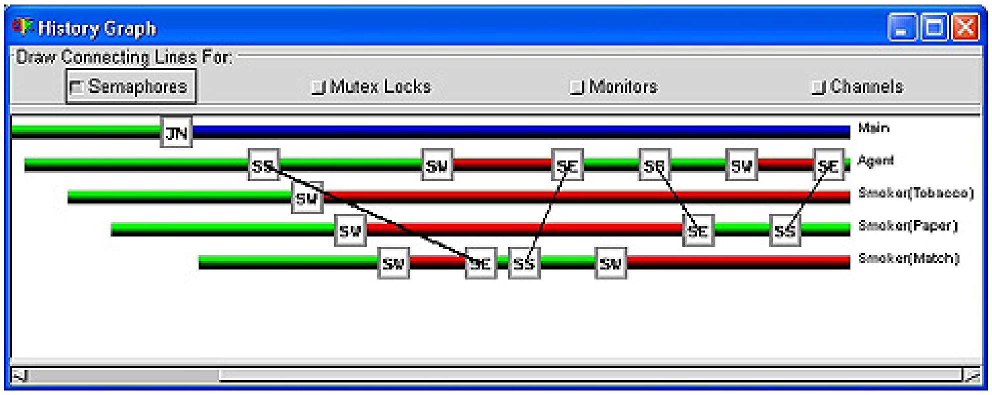


Fig. 8. History Graph Window when running the smokers program

A click on the name of a semaphore displayed in the display area of the Main Window brings up its Semaphore Window. Figure 9 shows all four Semaphore Windows. Each window has the name of the semaphore as the window name, and displays the current semaphore value and all waiting threads. For example, threads `Smoker(Tobacco)` and `Smoker(Match)` are waiting on semaphores `PaperMatch` and `TobaccoPaper`, respectively. The content of each semaphore (i.e., semaphore value and waiting threads) is updated on-the-fly. Combined with the History Graph Window, a user will be able to easily keep track of all semaphore related activities.

6. SYNCHRONIZATION PRIMITIVE: MONITORS

In ThreadMentor, a monitor is declared as a derived class of class `Monitor`. Constructors are used for initializing the local data of a monitor and methods are monitor procedures. Public monitor procedures must begin with a call to `MonitorBegin()` to establish monitor mutual exclusion, and must end with a call to `MonitorEnd()` to release the monitor lock. Without doing so, two threads that make calls to public monitor procedures may be in the monitor at the same time. However, private monitor procedures that can only be accessed within a monitor should not use `MonitorBegin()` and `MonitorEnd()` because the monitor has already been locked when a thread is executing in a monitor.

Condition variables are declared using class `Condition`, a private class in `Monitor`. As a result, no condition variable can be used outside of a monitor. Class `Condition` has two methods `Wait()` and `Signal()` for waiting on and signaling a condition variable, respectively. If a `Signal()` call causes the release of a thread that is waiting on the signaled condition variable, only one of the released thread and the

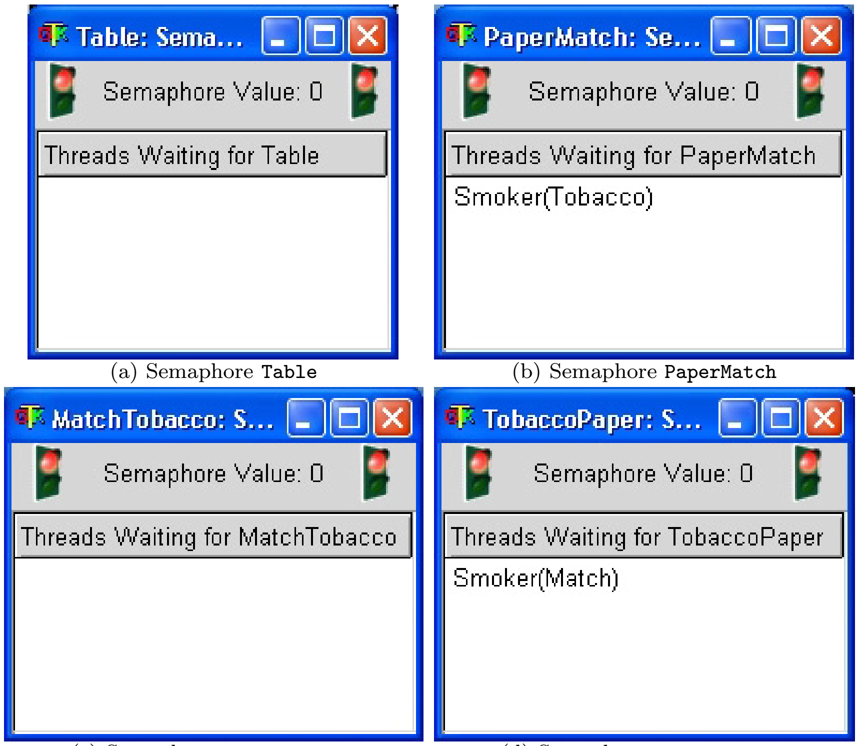


Table: Sema...
Semaphore Value: 0
Threads Waiting for Table

(a) Semaphore `Table`

PaperMatch: Se...
Semaphore Value: 0
Threads Waiting for PaperMatch
Smoker(Tobacco)

(b) Semaphore `PaperMatch`

MatchTobacco: S...
Semaphore Value: 0
Threads Waiting for MatchTobacco

(c) Semaphore `MatchTobacco`

TobaccoPaper: S...
Semaphore Value: 0
Threads Waiting for TobaccoPaper
Smoker(Match)

(d) Semaphore `TobaccoPaper`

Fig. 9. Semaphore Windows

signaling thread can continue. In a monitor of Hoare type, the signaling thread yields the monitor to the released thread, while in a monitor of Mesa type, the signaling thread continues [Buhr et al. 1995]. A user can specify the type of a monitor in a constructor.

Figure 10 shows a monitor of Hoare type for the bridge-crossing problem. Consider a very narrow bridge that can only allow three vehicles in the same direction to cross at the same time. If there are three vehicles on the bridge, any incoming vehicle must wait. When a vehicle exits the bridge, if there are other vehicles on the bridge, one waiting vehicle in the same direction should be allowed to proceed. Otherwise, we have two possibilities: (1) if there are vehicles waiting in the opposite direction, one of them should be allowed to proceed to make the scheduling fair, and (2) if there is no vehicle waiting in the opposite direction, let one waiting vehicle in the same direction proceed. Monitor `BridgeMonitor` has two monitor procedures `ArriveBridge()` and `ExitBridge()` for a vehicle to make a request to get on and off the bridge, respectively. Private monitor procedure `isSafe()` tests if a vehicle can be on the bridge safely. Monitor `BridgeMonitor` also includes some private variables: `WaitingLine[]` is an array of two condition variables for blocking incoming vehicles; `CurrentDirection` records the crossing direction of vehicles; `VehicleCount` is the number of vehicles on the bridge; and `Waiting[]` is an array of two integers for counting the number of waiting vehicles. For convenience, the moving directions of vehicles are east-to-west and west-to-east.

```
class BridgeMonitor: public Monitor
{
     public:
          BridgeMonitor(char* Name); // constructor
          void  ArriveBridge(int Direction);
          void  ExitBridge(int Direction);

     private:
          int  isSafe(int Direction);
          Condition *WaitingLine[2]; // blocks vehicles
          int  CurrentDirection;     // current direction of cars
          int  VehicleCount;         // # of vehicle on the bridge
          int  Waiting[2];           // # of east/west bound waiting
          char *Names[2];
};
```

Fig. 10. The bridge-crossing problem using monitor: class definition

Figure 11 shows a constructor and `isSafe()`. The constructor clears the counters and sets the monitor to the Hoare type. Procedure `isSafe()` receives the direction of a vehicle, and returns `TRUE` if the bridge has no vehicle or the vehicles on the bridge have the same direction as that of the requesting vehicle *and* the number of vehicles on bridge is not the maximum count. Otherwise, it returns `FALSE`.

```
BridgeMonitor::BridgeMonitor(char* Name): Monitor(Name, HOARE)
{
     VehicleCount = 0;             // no. vehicle on bridge
     Waiting[0] = Waiting[1] = 0;  // no. vehicle waiting
     Names[0] = "EastWest";
     Names[1] = "WestEast";
     WaitingLine[0] = new Condition(Names[0]); // E->W waiting line
     WaitingLine[1] = new Condition(Names[1]); // W->E waiting line
}

int  BridgeMonitor::isSafe(int Direction)
{
     if (VehicleCount == 0)        // if no vehicle on bridge
          return  TRUE;            //    safe to cross
     else if ((VehicleCount<MAX_VEHICLE) && (CurrentDirection==Direction))
          return  TRUE;            // if < max in the same direction
     else
          return  FALSE;           // otherwise, do not proceed
}
```

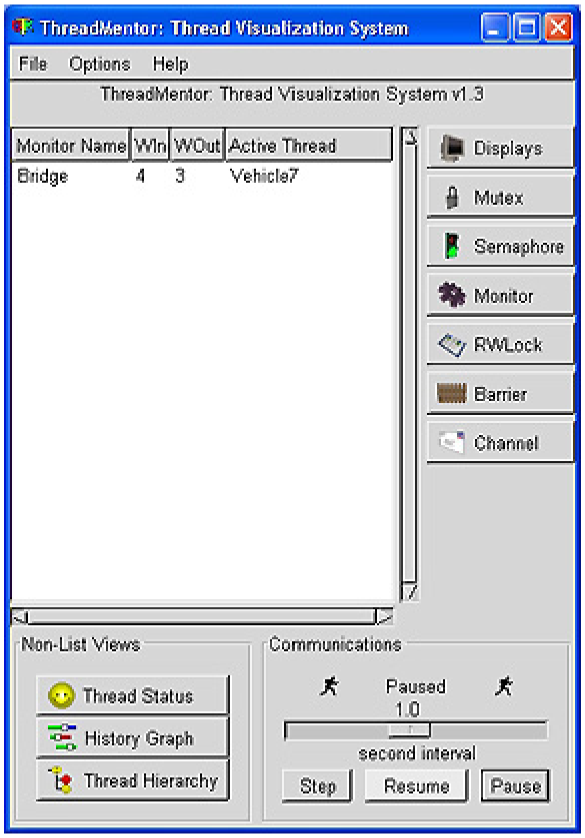
Fig. 11. The bridge-crossing problem using monitor: constructor and `isSafe()`

Figure 12 has the procedures `ArriveBridge()` and `ExitBridge()`. Procedure `ArriveBridge()` receives the direction of a vehicle and determines if that vehicle can be on the bridge. If not, the calling vehicle is blocked. It first checks if it is safe to allow the calling vehicle to be on the bridge using `isSafe()`. If it is safe, the on-bridge-vehicle counter is increased, the direction is set, and the calling vehicle has the permission to cross the bridge. Otherwise, before blocking the calling vehicle on a condition variable, the waiting vehicle count in the given direction is increased, and, after the release of a vehicle from the condition variable, the count is decreased. `ExitBridge()` takes the direction of an exiting vehicle and decreases the vehicle count. If the result is still positive, there still are vehicles on the bridge and a vehicle waiting in the same direction will be released. If the vehicle count reduces to zero, `ExitBridge()` allows a vehicle in the opposite direction to proceed if there are vehicles waiting in the opposite direction; otherwise, a vehicle in the same direction is allowed to proceed.

```
void  BridgeMonitor::ArriveBridge(int Direction)
{
     MonitorBegin();
     if (!isSafe(Direction)) {     // is it safe to be on the bridge
          Waiting[Direction]++;    // no, wait at the bridge
          WaitingLine[Direction]->Wait(); // block this vehicle
          Waiting[Direction]--;    // released
     }
     VehicleCount++;               // go on bridge
     CurrentDirection = Direction; // set direction
     MonitorEnd();                 // release monitor
}

void  BridgeMonitor::ExitBridge(int Direction)
{
     MonitorBegin();                    // lock the monitor
     VehicleCount--;                    // one vehicle exits
     if (VehicleCount > 0)
          WaitingLine[Direction]->Signal(); // release the same direction
     else {                             // no vehicle on bridge
          if (Waiting[1-Direction] != 0)// opposite direction non-empty?
               WaitingLine[1-Direction]->Signal(); // release one of them
          else                          // release the same direction
               WaitingLine[Direction]->Signal();
     }
     MonitorEnd();
}
```

Fig. 12. The bridge-crossing problem using monitor: monitor procedures

A thread involved in a monitor procedure call may be in one of the four states: (1) *Active* – the thread is executing in the monitor and holding the monitor lock, (2) *Waiting* – the thread is blocked on a condition variable, (3) *Entering* – the thread has called `MonitorBegin()` and is waiting to enter, and (4) *Re-Entering* – the thread has yielded the monitor and not yet regained the control. For example, in a Hoare type monitor, since the signaling thread must yield the monitor to the released thread, once the `Signal()` call completes, the signaling thread and released thread are in the Re-Entering state and Active state, respectively. On the other hand, in a Mesa type monitor, after the execution of a `Signal()`, the signaling thread and the released thread are in the Active state and Re-Entering state, respectively. In ThreadMentor, priority will be given to the threads in the Re-Entering state. More specifically, when a thread exits or yields the monitor due to a condition variable wait, ThreadMentor will bring a thread in the Re-Entering state to the Active state. If there is no thread in the Re-Entering state, ThreadMentor will bring a thread in the Entering state to the Active state. Those Re-Entering threads and threads blocked on condition variables are said to be *in* the monitor and those Entering threads are *outside* of the monitor. This information is shown in the Main Window when the `Monitor` button is selected. Figure 13 shows a snapshot of running the bridge-crossing program: thread `Vehicle7` is executing in the monitor (i.e., active), three vehicle threads are waiting to enter, and four vehicle threads are blocked on condition variables or re-entering.


ThreadMentor: Thread Visualization System v1.3

Monitor Name	WIn	WOut	Active Thread
Bridge	4	3	Vehicle7

Fig. 13. Main Window when running the bridge-crossing program

A click on the monitor name in the display area of the Main Window brings up the Monitor Window of the selected monitor. The Monitor Window displays the name of the active thread in the top and the type of the monitor (i.e., Hoare or Mesa) in the bottom, and has two rows of subwindows. The top row always has two subwindows, the left and right ones showing the names of all entering and re-entering threads, respectively. Each condition variable has a subwindow on the bottom row to show the threads that are blocked on the condition variable. The Monitor Window in Figure 14 provides the following information: (1) the active thread is `Vehicle7`; (2) three vehicles are in the Entering state, `Vehicle5`, `Vehicle4`, and `Vehicle3`; (3) one vehicle is in the Re-Entering state, `Vehicle2`; (4) two vehicles are blocked on condition variable `EastWest`, `Vehicle1` and `Vehicle6`; and (5) one vehicle is blocked on condition variable `WestEast`, `Vehicle8`.

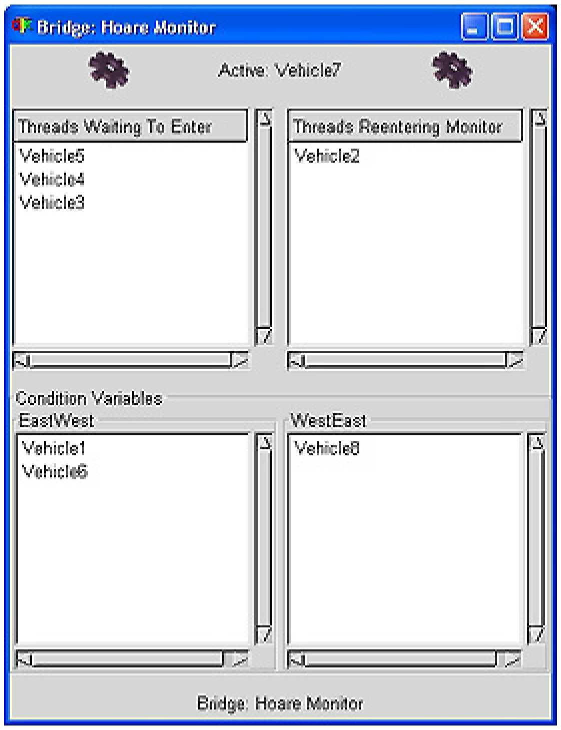


Fig. 14. Monitor Window when running the bridge-crossing program

The History Graph Window has seven tags that are related to a Hoare type monitor. Tags `MB` and `ME` indicate monitor calls `MonitorBegin()` and `MonitorEnd()`, respectively. Since a thread that calls `MonitorBegin()` may not be granted the permission to enter the monitor, after tag `MB` the history bar of the caller is in red. Later, when the monitor becomes empty, another thread will be allowed to enter the monitor, and, as a result, this thread will have a `MA` (monitor active) tag. Tag `CW` indicates a thread issued a condition variable `Wait()`. Thus, following a `CW` tag, the history bar is in red. Since this is a Hoare monitor, the signaling thread yields the monitor to the released thread, and tags CY (i.e., condition yield) and CA (i.e., condition active) are shown on the history bars of the signaling and released threads, respectively. A thread having a CY tag means it is in the Re-Entering state. A thread in the Re-Entering state will be allowed to run when the monitor has no running thread, and a MR (monitor reactive) tag will follow a CY tag.

A click on the Monitor button in the History Graph Window displays the *wait-active* and *signal-active* links. Since tags ME and CW indicate that the monitor is released by the executing thread, another thread can enter or re-enter the monitor. If this thread is in the Entering state (i.e., tagged with a MB on its history bar), the line segment between a ME and a MA or between a CW and a MA is referred to as a *wait-active link*. If the selected thread is in the Re-Entering state (i.e., tagged with a CY), the line segment between a CY and a CA is referred to as a *signal-active link*. In this way, all monitor related synchronization events and the relationship between two corresponding events are clearly shown by the tags and links.

Figure 15 is a snapshot of the History Graph Window of the bridge-crossing program. From this snapshot, we learn the following: (1) Vehicle1 is released by the exit of Vehicle6 as shown by a wait-active link; (2) Vehicle1 waits to get on the bridge and yields the monitor to Vehicle2 as shown by a wait-active link; (3) Vehicle2 exits the monitor, which causes Vehicle5 to become active as shown by a wait-active link, crosses the bridge, and comes back by calling ExitBridge() as shown by a MB tag; (4) Vehicle5 becomes active, exits the monitor, which causes Vehicle3 to become active, crosses the bridge, and comes back to call ExitBridge(); (5) Vehicle3 becomes active, calls Signal(), which causes Vehicle4 to become active as shown by a signal-active link, and yields the monitor to Vehicle4; (6) Vehicle4 becomes active, exits the monitor, which causes Vehicle3 to become active immediately as shown by a wait-active link, crosses the bridge, and comes back to call ExitBridge(); and (7) Vehicle3 regains the monitor (i.e., the ME tag and MR tag), exits (i.e., off the bridge), and comes back to cross the bridge again. The other events are similar. Again, a click on an event tag brings up the Source Window to display the source program with the line that contains the corresponding synchronization event highlighted.

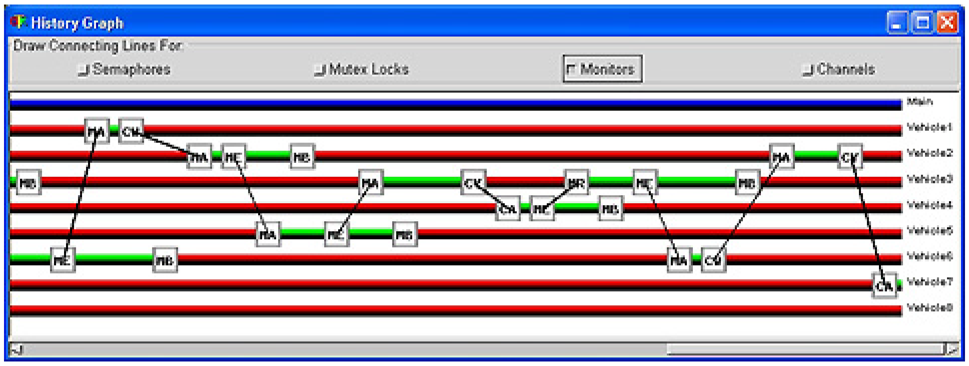


Fig. 15. History Graph Window when running the bridge-crossing program

For a Mesa type monitor, since the condition at the time a thread being released from a condition variable and the time of its execution may have been modified by other threads, re-testing the condition becomes necessary. Figure 16 is the procedure `ArriveBridge()` for the Mesa type. When a thread is released from a condition variable, it must go back to make sure it is safe to get on the bridge.

```
void  BridgeMonitor::ArriveBridge(int Direction)
{
    MonitorBegin();
    if (!isSafe(Direction)) {
        Waiting[Direction]++;           // not safe, wait
        while (!isSafe(Direction))      // always re-check
            WaitingLine[Direction]->Wait();
        Waiting[Direction]--;
    }
    VehicleCount++;
    CurrentDirection = Direction;
    MonitorEnd();
}
```

Fig. 16. The bridge-crossing problem using monitor: Mesa version

There are two more tags for a Mesa type monitor. Tag `SC` means a signal has been made to a condition variable, and tag `SR` means a signal has been received by a condition variable. Due to the semantics of a Mesa type monitor, the history bar portions following `SC` and `SR` are colored in green and red, respectively, because the signaling thread continuous to run and the released thread is re-entering. Figure 17 is a snapshot of the History Graph Window when running the Mesa version of the bridge-crossing program. From this figure, we learn that (1) `Vehicle8` executes a `Wait()` and yields the monitor to `Vehicle1`; (2) `Vehicle1` becomes active, executes a `Signal()` (i.e., tag `SC`) which causes `Vehicle4` to be released (i.e., tag `SR`), exits the bridge which causes `Vehicle2` to become active, and comes back to call `ArriveBridge()`; and (3) later, the exit of `Vehicle3` causes `Vehicle4` to become active. It is clear that the signal-release pattern of this Mesa monitor is much simpler than that of the Hoare monitor.

In addition to the above standard monitor features, `Wait()` can take a priority value as its only argument, method `Empty()` tests if there are waiting threads on a condition variable, and method `Broadcast()` (Mesa monitor only) releases *all* threads waiting on a condition variable.

7. SYNCHRONIZATION PRIMITIVE: CHANNELS

In ThreadMentor, a channel is a *bi-directional* communication link for threads to send messages to and receive messages from another thread. The *capacity* of a channel is its buffer size. If the capacity is zero, no message can be waiting in a channel, and a sender must wait until a receiver receives the message. The sender and receiver are synchronized for a message transfer to occur. Consequently, channels with zero capacity are usually referred to as *synchronous* channels (i.e., blocking send and blocking receive), and the synchronization is a *rendezvous*. If the

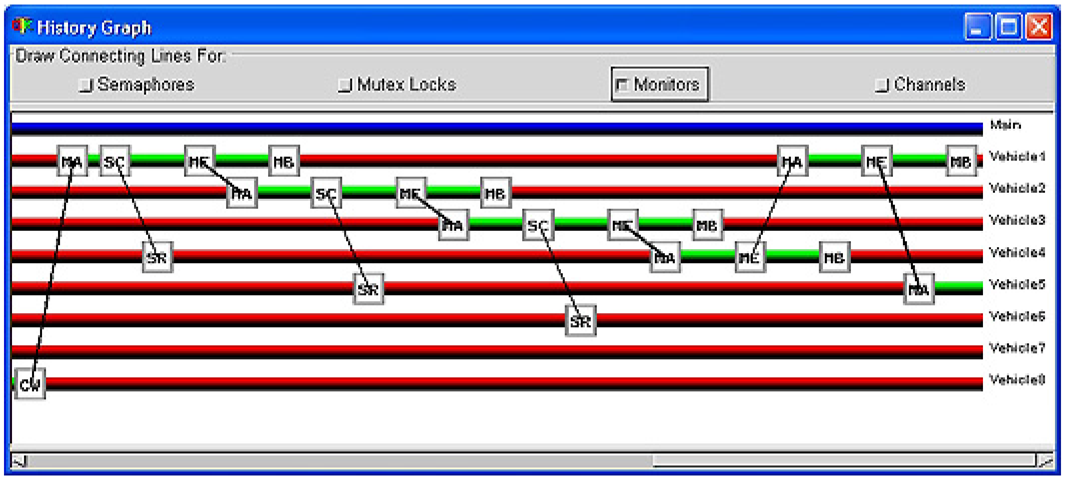


Fig. 17. History Graph Window when running the bridge-crossing program: Mesa version

capacity is non-zero, messages may wait in the buffer of a channel. A sender (resp., receiver) waits if the buffer is full (resp., empty), and the channel is *asynchronous*. However, if the buffer size is *unbounded*, a sender never waits. Messages that wait in a channel may not be in the first-in-first-out order; however, ThreadMentor always collects the messages in the incoming order (i.e., FIFO).

Channels are divided into three types, one-to-one, many-to-one and many-to-many. With a one-to-one channel, the relationship among the channel, the sender, and the receiver is fixed throughout the execution of the program, and only the specified threads can send messages to and receive messages from this channel. With a many-to-one channel, only the receiver end is fixed and every thread can send messages to the receiver. Any thread can send messages to and receive messages from a many-to-many channel. If the capacity of a many-to-many channel is finite but non-zero, this channel is simply a variation of a bounded-buffer. While one-to-one channels are commonly used in parallel and distributed computing [Lynch 1996], many-to-one channels are also being used frequently [Andrews 1991; 2000]. A Unix message queue is a finite but non-zero capacity many-to-many channel. ThreadMentor supports all channel types mentioned above. The default capacity of an asynchronous channel is unbounded; however, a user may set the capability when a channel is constructed.

In ThreadMentor, synchronous and asynchronous one-to-one channels are declared using classes `SynOneToOneChannel` and `AsynOneToOneChannel`, respectively. Three arguments are required to declare a one-to-one channel: a symbolic name and two user-defined IDs each of which identifies a thread at one end of the channel. A user defined ID is a unique non-negative integer chosen by the user for thread identification purpose. Each channel has three methods: `Send()` for sending a message into the channel, `Receive()` for receiving a message from the channel, and `Empty()` for testing if there are messages in the channel. Methods `Send()` and `Receive()` require two arguments. The first is a pointer to a data item and the second is message length.

Consider the sieve problem. Initially, there are two threads, the pump thread and sieve thread 2, connected with a synchronous one-to-one channel. The pump thread keeps sending integers 3, 4, 5, . . . to sieve thread 2 which memorizes the first prime number 2. When a sieve thread receives a number from its incoming channel, if the received number is a multiple of the memorized one, the received number is ignored and the sieve thread retrieves the next number from its incoming channel. If the received number is not a multiple of the memorized one, this sieve thread passes the received number to the next sieve thread through its outgoing channel. Otherwise, this sieve thread creates a new sieve thread, which builds an incoming channel from this thread and memorizes the first number. Repeating this process, eventually every sieve thread memorizes a prime number. Figure 18 shows three sieve threads that memorize prime numbers 2, 3 and 5. The pump thread just sends 10 to sieve thread 2. At the same time, sieve thread 2 sends 9 to sieve thread 3, because 9 is not a multiple of 2. Similarly, sieve thread 3 sends 7 to sieve thread 5. In the next round, sieve thread 5 creates sieve thread 7 which memorizes 7, sieve thread 3 ignores the incoming number 9, sieve thread 2 ignores the incoming number 10, and the pump thread sends out the next number 11.

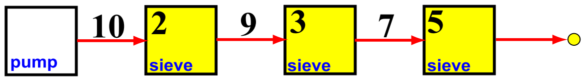


Fig. 18. The sieve problem with synchronous channels

Figure 19 shows the class definitions of the pump thread and the sieve thread. Private variable `prime` is the prime number memorized by a sieve thread. Figure 20 has the constructors. In the construction of a channel between two sieve threads, the user defined IDs are numbered sequentially. The pump thread receives 0, sieve thread 2 receives 1, and subsequent sieve threads receive 2, 3, 4, and so on. When a sieve thread is created, it saves the assigned thread ID into private variable `UserDefinedThreadID` for identification purpose, sets `nextSieve` to indicate that this is the last sieve thread in chain, memorizes the prime number, and creates a channel between this sieve thread and its creator (i.e., predecessor). The pump thread also saves the assigned user thread ID into `UserDefinedThreadID`. For every send and receive operation, ThreadMentor uses `UserDefinedThreadID` to verify if the sender and the receiver are the ones specified when the channel was created.

Figure 21 has the thread bodies of the pump thread and sieve thread, where `STOP` is a global integer with a negative value, and `sieve` is a global variable pointing to the channel, constructed by the main program, between the pump thread and sieve thread 2. A sieve thread keeps looping until it receives `STOP`, indicating that the prime finding process ends. Then, it passes the `STOP` mark to its successor if it has one, and terminates. If the received number is a multiple of the memorized one, it is ignored and the thread loops back to receive the next number. There are two cases to consider if the received number is not a multiple of the memorized one. If this sieve thread is not the last in the chain, the received number is sent to the successor. Otherwise, this sieve thread creates a new sieve thread, passes

```
class SieveThread : public Thread
{
    public:
        SieveThread(int prime, int threadID);
        SynOneToOneChannel *channel;
    private:
        void ThreadFunc();
        int prime;
        SieveThread *nextSieve; // next Sieve thread
};

class PumperThread : public Thread
{
    public:
        PumperThread(int Limit, int threadID);
    private:
        void ThreadFunc();
        int limit;
};
```

Fig. 19. Class definition of the sieve program

```
SieveThread::SieveThread(int prime, int threadID)
{
    UserDefinedThreadID = threadID;
    nextSieve = NULL;
    this->prime = prime;
    channel = new SynOneToOneChannel(name, threadID - 1, threadID);
}

PumperThread::PumperThread(int Limit, int threadID)
{
    limit = Limit;
    UserDefinedThreadID = threadID;
}
```

Fig. 20. The constructors of the sieve program

the incoming number to it, and runs the newly created sieve thread. Therefore, all threads are in a linear list with the head being the pump thread.

A click on the `Channel` button in the Main Window displays all channels created so far (Figure 22). For each channel, its name, type (i.e., synchronous or asynchronous), and state (i.e., `Sending`, `Received`, `Acknowledged` and `Empty`) are shown. In the figure, channel `Channel(0)(1)` between the pump thread and sieve thread 2 has a waiting message; channel `Channel(1)(2)` between sieve thread 2 and sieve thread 3 has received a message and acknowledged by sieve thread 3; and channel `channel(2)(3)` between sieve thread 3 and sieve thread 5 has been created and is empty.

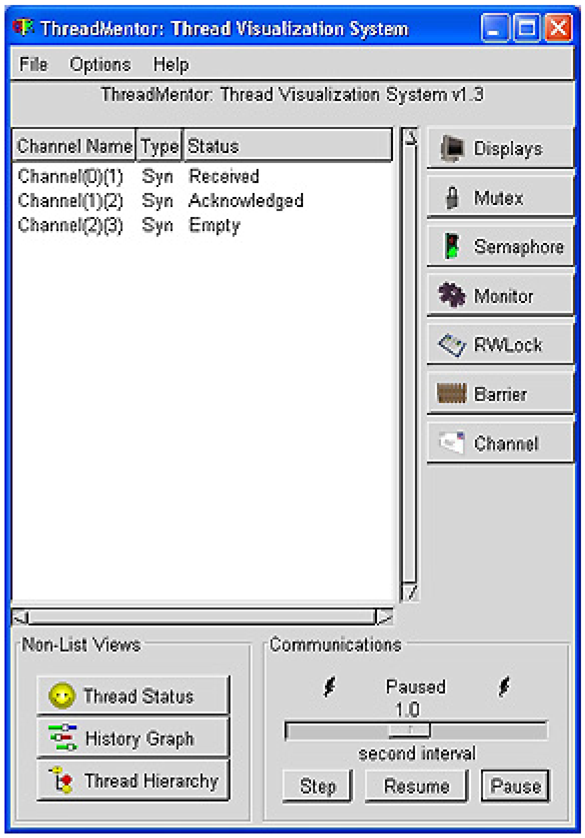
The History Graph Window has three tags for channels. Tags `CS`, `CR` and `CK` indi-

```
void SieveThread::ThreadFunc()
{
    Thread::ThreadFunc();
    int     number;

    while (true) {
        channel->Receive(&number, sizeof(int)); // get a number
        if (number == STOP)                     // is it a STOP?
            break;                              // yes, bail out
        if (number % prime != 0) {              // a composite?
            if (nextSieve != NULL)              // successor?
                nextSieve->channel->Send(&number, sizeof(int));
            else {                              // no, create one
                nextSieve = new SieveThread(number, UserDefinedThreadID + 1);
                nextSieve->Begin();             // run the succ
            }
        }
    }
    if (nextSieve != NULL) {                    // am I the last?
        nextSieve->channel->Send(&number, sizeof(int));
        nextSieve->Join();                      // no, pass STOP
    }
    Exit();
}

void PumperThread::ThreadFunc()
{
    Thread::ThreadFunc();
    int     i;

    for(i = 3; i <= limit; i++)
        sieve->channel->Send(&i, sizeof(int));
    sieve->channel->Send(&STOP, sizeof(int));
}
```

Fig. 21. The sieve and pump threads

cate a message is sent, received and acknowledged, respectively. With a synchronous channel, a sender waits until its message is received by the indicated receiver. As a result, the sender's history bar between tags CS and CK is in red, meaning the sender is blocked. A click on the Channels button in the upper-right corner of the History Graph Window displays all *send-receive links* and *receive-acknowledge links*. The former consist of all line segments between a tag CS and its corresponding CR tag, and the latter are line segments between a tag CR and its corresponding CK tag. In this way, the behavior of synchronous channels is clearly shown.

Figure 23 is a snapshot of the History Graph Window when running the sieve program. Since the main program creates sieve thread 2 first, followed by the pump thread, the second and third history bars are for sieve thread 2 and pump thread, respectively. Then, sieve thread 2 creates sieve thread 3 (i.e., the fourth



Channel Name	Type	Status
Channel(0)(1)	Syn	Received
Channel(1)(2)	Syn	Acknowledged
Channel(2)(3)	Syn	Empty

Fig. 22. Main Window when running the sieve program

history bar), which, in turn, creates sieve thread 5 (i.e., the fifth history bar). From this figure, we learn the following: (1) the pump thread sends 3 to sieve thread 2 (i.e., the first CS-CR-CK triplet); (2) sieve thread 2 receives 3 and creates sieve thread 3; (3) the pump thread sends 4 to sieve thread 2 (i.e., the second CS-CR-CK triplet); (4) sieve thread 2 ignores the received number 4; (5) the pump thread sends 5 to sieve 2 (i.e., the third CS-CR-CK triplet); (6) sieve thread 2 receives 5 and sends 5 to sieve thread 3 (i.e., the CS-CR-CK triplet between sieve thread 2 and sieve thread 3); (7) the pump thread sends 6 to sieve thread 2; and (8) sieve thread 3 creates sieve thread 5. Thus, with these tags, the relative timing of message passing activities are shown vividly.

A click on a channel name in the display area of the Main Window brings up the Channel Window of the selected channel (Figure 24). The top portion of the Channel Window displays the state of the channel. The status of a channel may be `Sending Message`, `Message Received` or `Last Message Acknowledged`. The left Message on Channel subwindow shows the incoming messages, and the right Received Message History subwindow has the message received history. The bottom part of the Channel Window displays the type of the channel and the lower-right

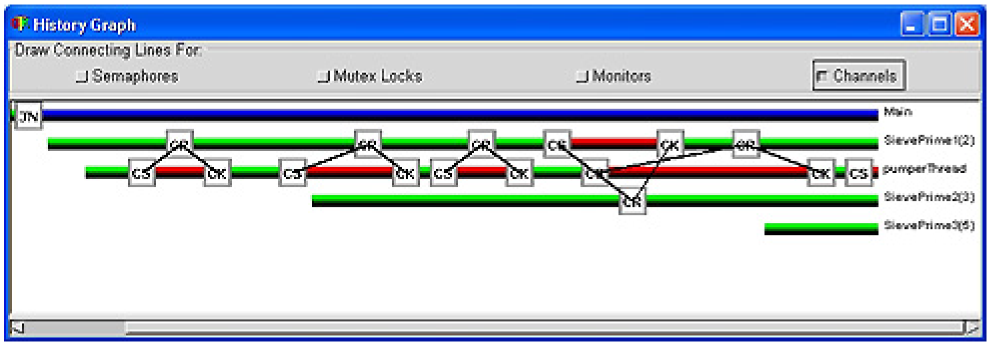
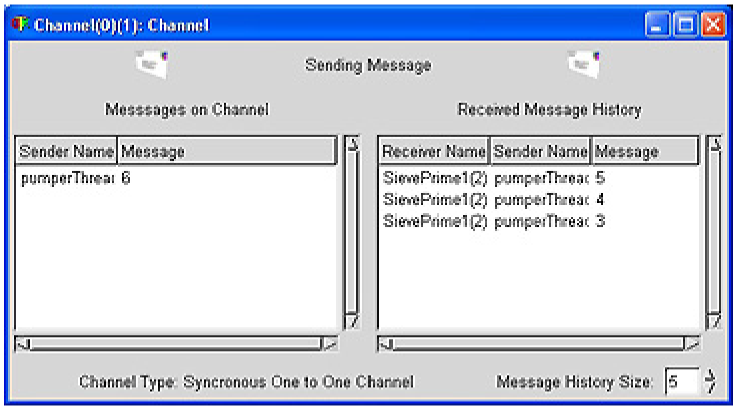


Fig. 23. History Graph Window when running the sieve program

corner has buttons for selecting the number of messages to be kept in the Message Received History subwindow. Figure 24 is the Channel Window for the channel between the pump thread and sieve thread 2. The Messages on Channel subwindow shows that the pump thread has sent 6 into the channel, and the Received Messages History subwindow shows that this channel has received three messages 3, 4 and 5, in this order, and that the sender and receiver are the pump thread and sieve thread 2, respectively. When the send operation completes, which means the channel has successfully received and delivered the message to the receiver, this message is moved to the Received Messages History subwindow and the channel state changes to `Message Received`. At this point, a send-receive link will be shown between the corresponding `CS` and `CR` tags. Because this is a synchronous channel, the sender can continue only if the receiver has received the message. Thus, once the receiver receives the message completely, the status will be changed to `Last Message Acknowledged`, both the sender and receiver continue, and a receive-acknowledge link appears between the corresponding `CR` and `CK` tags.


Channel1(0)(1): Channel

Sending Message

Messsages on Channel

Sender Name	Message
pumperThread	6

Received Message History

Receiver Name	Sender Name	Message
SievePrime1(2)	pumperThread	5
SievePrime1(2)	pumperThread	4
SievePrime1(2)	pumperThread	3

Channel Type: Syncronous One to One Channel

Message History Size: 5

Fig. 24. Channel Window when running the sieve program

Since asynchronous channels use non-blocking send, the `CK` tag does not appear in the History Graph Window, the `Last Message Acknowledged` state never appears in the Channel Window, and more than one message may be waiting in the Message on Channel subwindow. Consequently, a program that uses only asynchronous channels will have no red portion on any history bar of its History Graph Window.

8. CONCLUSIONS

We have presented a detailed overview of the class library and visualization of ThreadMentor. ThreadMentor was used twice in the programming track of our **Introduction to Operating Systems** course [Shene 1998; 2002] to replace Sun Solaris threads, and at three workshops [Carr et al. 2001; 2002; 2003]. It was also site-tested at a number of schools. Reactions from site testers and participants of our workshops were very positive and encouraging. Typical comments include "the visualization is excellent" and "[ThreadMentor] is a useful tool for OS classes." In the attitude surveys that were conducted at the end of the above mentioned course, students indicated overwhelmingly that the visualization system of ThreadMentor "helps pinpoint errors quickly" and "helps to see what is happening with the threads." The complete system is "wonderful," "ease of use and straightforward," "a good learning tool and very handy" and "[taking] a lot of trouble out of using threads," and has "easy semantics and calling convention" and "a common interface between Linux and Solaris [and Windows]." Two students indicated that they never used the visualization because it is not their programming style. There are only a few negative comments, most of which are due to unfamiliarity with the system. For example, a handful of students indicated that ThreadMentor does not perform properly on a remote machine (e.g., remote login). Since the GUI of the visualization system must transmit a large amount of graphical information, running ThreadMentor on a remote machine and displaying the windows locally cannot be very efficient. Thus, ThreadMentor is designed to run on a local machine, although it is possible to execute ThreadMentor on a remote machine. A few students mentioned that ThreadMentor's behavior could be different on different machines and on different operating systems. This is normal because the behavior of a threaded program cannot be identical across platforms. It is also interesting to mention that two students, one per year, criticized ThreadMentor for being proprietary and not used in the "real world." They prefer Pthreads or Sun Solaris threads over ThreadMentor. Since the number of negative comments is very few, we believe that ThreadMentor is currently a reasonably mature system. A detailed analysis of the effectiveness of using ThreadMentor in the above mentioned course with pre- and post- tests will be published elsewhere.

Based on the experience we gained from site testers, workshop participants and our students, some improvements can be made to the current version of ThreadMentor. The most important task is to make the system more stable across platforms, including possibly Mac OS X, and support more compilers (e.g., Microsoft and Borland). The channel component can also be improved to include more primitives (e.g., channel broadcasting and empty testing), better visualization support (e.g., buffer content and sender-receiver pair of many-to-one and many-to-many channels), and a tighter topology editor support [Carr et al. 2002]. Moreover, we are developing a sister system ConcurrentMentor, based on channels, for distributed programming with a similar visualization support [Carr et al. 2002; 2003]. We hope to keep the channel interfaces and visualization and topology editor components of both systems consistent. Currently, ThreadMentor is being ported to Java and IBM's Eclipse platform. A more ambitious plan is to extend ThreadMentor and ConcurrentMentor with two more layers: a protocol visualization layer and a distributed algorithm animation layer. The former helps students visualize various protocols (e.g., leader election, distributed mutual exclusion and distributed deadlock detection) and the latter, which is built on top of the former, provides students with an environment for distributed algorithm animation. In this way, our systems will support multithreaded and distributed programming as well as distributed algorithms studies.

The interested readers may find more about our work, software availability, course materials, a ThreadMentor tutorial, and future announcements at the following site:

`http://www.cs.mtu.edu/~shene/NSF-3`

The URLs of the home pages of ThreadMentor and ConcurrentMentor are

`http://www.cs.mtu.edu/ThreadMentor`
`http://www.cs.mtu.edu/ConcurrentMentor`

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation under grants DUE-9752244 and DUE-9952509. The second and third authors are also supported by an NSF CAREER grant CCR-9984682 and an IBM Eclipse Innovation Award, respectively. We also thank the anonymous referees for their constructive comments.

REFERENCES

ACM/IEEE. 2001. *Computing Curricula 2001.* `http://www.acm.org/sig/sigcse/ccs001/`.

Adams, J., Nevison, C., and Schaller, N. C. 2000. Parallel Computing to Start the Millennium. In *ACM 31st SIGCSE Technical Symposium.* 65–69.

Andrews, G. R. 1991. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings.

Andrews, G. R. 2000. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley.

Andrews, G. R. and Olson, R. A. 1992. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings.

Arnow, D. M. 1995. A Simple Library for Teaching a Distributed Programming Module. In *ACM 26th SIGCSE Technical Symposium.* 82–86.

Bedy, M., Carr, S., Huang, X., and Shene, C.-K. 1999. The Design and Construction of a User-Level Kernel for Teaching Multithreaded Programming. In *29th ASEE/IEEE Frontiers in Education.* Vol. II. (13a3–24)–(13a3–29).

Bedy, M., Carr, S., Huang, X., and Shene, C.-K. 2000. A Visualization System for Multithreaded Programming. In *ACM 31st SIGCSE Technical Symposium.* 1–5.

Bemmerl, T. and Braun, P. 1993. Visualization of Message Passing Parallel Programs with the TOPSYS Parallel Programming Environment. *Journal of Parallel and Distributed Computing 18*, 118–128.

Ben-Ari, M. 1982. *Principles of Concurrent Programming.* Prentice Hall.

Ben-Ari, M. 1996. Using Inheritance to Implement Concurrency. In *ACM 27th SIGCSE Technical Symposium.* 180–184.

Ben-Ari, M. 1997. Distributed Algorithms in Java. In *ACM 2nd ITiCSE Conference.* 62–64.

Ben-Ari, M. 2001. Interactive Execution of Distributed Algorithms. *ACM Journal of Education Resources in Computing 1,* 2 (Summer). article #2.

Berk, T. S. 1996. A Simple Student Environment for Lightweight Process Concurrent Programming under SunOS. In *ACM 27th SIGCSE Technical Symposium.* 165–169.

Brinch Hansen, P. 1999. Java's insecure parallelism. *ACM SIGPLAN Notices 34,* 4, 38–45.

Buhr, P. A., Fortier, M., and Coffin, M. H. 1995. Monitor Classification. *ACM Computing Surveys 27,* 1 (March), 63–107.

Burkhart, H. 1997. Parallel Programming Using Public Domain Software. In *ACM 28th SIGCSE Technical Symposium.* 224–228.

Burns, A. and Davies, G. 1993. *Concurrent Programming.* Addison-Wesley.

Bynum, B. and Camp, T. 1996. After You, Alfonse: A Mutual Exclusion Toolkit. In *ACM 27th SIGCSE Technical Symposium.* 170–174.

Cai, W., Milne, W. J., and Turner, S. J. 1993. Graphical Views of the Behavior of Parallel Programs. *Journal of Parallel and Distributed Computing 18*, 223–230.

Carr, S., Chen, P., Jozwowski, T. R., Mayo, J., and Shene, C.-K. 2002. Channels, Visualization, and Topology Editor. In *ACM 7th ITiCSE 2002 Conference.* 106–110.

Carr, S., Fang, C., Jozwowski, T., Mayo, J., and Shene, C.-K. 2002. A Communication Library to Support Concurrent Programming Courses. In *ACM 33rd SIGCSE Technical Symposium.* 360–364.

Carr, S., Fang, C., Jozwowski, T. R., Mayo, J., and Shene, C.-K. 2003. ConcurrentMentor: A Visualization System for Distributed Programming Education. In *The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications.* 1676–1682.

Carr, S., Mayo, J., and Shene, C.-K. 2001. Teaching Multithreaded Programming Made Easy (workshop). *The Journal of Computing in Small Colleges 17,* 1 (October), 156–157.

Carr, S., Mayo, J., and Shene, C.-K. 2002. Teaching Multithreaded Programming Made Easy (workshop). In *ACM 33rd SIGCSE Technical Symposium.* 420.

Carr, S., Mayo, J., and Shene, C.-K. 2003. Teaching Multithreaded Programming Made Easy Using ThreadMentor (workshop). In *ACM 34th SIGCSE Technical Symposium.* 409.

Carr, S. and Shene, C.-K. 2000. A Portable Class Library for Teaching Multithreaded Programming. In *ACM 5th ITiCSE 2000 Conference.* 124–127.

Chao, C. S. and Liu, A. C. 2000. A visualization modeling framework for a csp-based system. *Journal of Visual Languages and Computing 11*, 83–103.

Choi, S.-E. and Lewis, E. C. 2000. A Study of Common Pitfalls in Simple Multi-Threaded Programs. In *ACM 31st SIGCSE Technical Symposium.* 325–329.

Cunha, J. C. and ao Lourenço, J. 1998. An Integrated Course on Parallel and Distributed Processing. In *ACM 29th SIGCSE Technical Symposium.* 217–221.

Elenbogen, B. S. 1996. Parallel and Distributed Algorithms: Laboratory Assignments in Joyce/Linda. In *ACM 27th SIGCSE Technical Symposium.* 14–18.

Flinn, S. and Cowan, W. 1990. Visualizing the Execution of Multiprocessor Real-Time Programs. In *Graphics Interface 1990.* 293–302.

Harlan, R. M. 1995. Parallel Threads: Parallel Computation Labs for CS 3 and CS 4. In *ACM 26th SIGCSE Technical Symposium.* 141–145.

Hartley, S. J. 1992. An Operating System Laboratory Based on the SR (Synchronizing Resources) Programming Language. *Computer Science Education 3*, 251–276.

Hartley, S. J. 1994. Animating Operating Systems Algorithms with XTANGO. In *ACM 25th SIGCSE Technical Symposium.* 344–348.

Hartley, S. J. 1998. "Alfonse, Your Java Is Ready!". In *ACM 29th SIGCSE Technical Symposium.* 247–251.

Hartley, S. J. 1999. "Alfonse, Wait Here for My Signal!". In *ACM 30th SIGCSE Technical Symposium.* 58–62.

Hartley, S. J. 2000. "Alfonse, You Have a Message!". In *ACM 31st SIGCSE Technical Symposium.* 60–64.

HARTLEY, S. J. 2001. "Alfonse, Give Me a Call!". In *ACM 32nd SIGCSE Technical Symposium*. 229–232.

HIGGINBOTHAM, C. W. AND MORELLI, R. 1991. A System for Teaching Concurrent Programming. In *ACM 22nd SIGCSE Technical Symposium*. 309–316.

JIN, L. AND YANG, L. 1995a. A Laboratory for Teaching Parallel Computing on Parallel Structures. In *ACM 26th SIGCSE Technical Symposium*. 39–43.

JIN, L. AND YANG, L. 1995b. Integrating Parallel Algorithm Design with Parallel Machine Models. In *ACM 26th SIGCSE Technical Symposium*. 131–135.

KOLIKANT, Y. B.-D., BEN-ARI, M., AND POLLACK, S. 2000. The anthropology of Semaphores. In *ACM 5th ITiCSE Conference*. 21–24.

KOTZ, D. 1995. A Data-Parallel Programming Library for Education (DAPPLE). In *ACM 26th SIGCSE Technical Symposium*. 76–81.

KRAEMER, E. 1998. Visualizing Concurrent Programs. In *Software Visualization*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Eds. MIT Press, 237–256.

KURTZ, B. L. AND ALSABBAGH, J. 1998. Parallel Computing in the Undergraduate Curriculum. In *ACM 29th SIGCSE Technical Symposium*. 212–216.

KURTZ, B. L., CAI, H., PLOCK, C., AND CHEN, X. 1998. A Concurrency Simulator Designed for Sophomore-Level Instruction. In *ACM 29th SIGCSE Technical Symposium*. 237–241.

KWIATKOWSKI, J., ANDRUSZKIEWICZ, M., LUQUE, E., MARGALEF, T., CUNHA, J., LOURENCO, J., KRAWCZYK, H., AND SZEJKO, S. 1996. Teaching Parallel Processing: Development of Curriculum and Software Tools. In *ACM 1st ITiCSE Conference*. 159–161.

LESTER, B. P. 1993. *The Art of Parallel Programming*. Prentice-Hall.

LUQUE, E., SORRIBES, J., SUPPI, R., CESAR, E., FALGUERA, J. L., AND SERRANO, M. 1996. Parallel Systems Development in Education: A Guided Method. In *ACM 1st ITiCSE Conference*. 156–158.

LYNCH, N. A. 1996. *Distributed Algorithms*. Morgan Kaufmann.

MATTIS, P., KIMBALL, S., AND MACDONALD, J. 2002. GTK+ – The GIMP Toolkit. available at `http://www.gtk.org`.

MCDONALD, C. AND KAZEMI, K. 1997. Improving the PVM Teaching Environment. In *ACM 28th SIGCSE Technical Symposium*. 219–223.

MCDONALD, C. AND KAZEMI, K. 2000. Teaching Parallel Algorithms with Process Topology. In *ACM 31st SIGCSE Technical Symposium*. 70–74.

MILLER, B. P. 1993. What to Draw? When to Draw? An Essay on Parallel Program Visualization. *Journal of Parallel and Distributed Computing 18*, 265–269.

NAPS, T. L. AND CHAN, E. E. 1999. Using Visualization to Teach Parallel Algorithms. In *ACM 30th SIGCSE Technical Symposium*. 232–236.

PANCAKE, C. M. 1996. Visualization Techniques for Parallel Debugging and Performance-Tuning Tools. In *Parallel Computing: Paradigm and Applications*, A. Y. Zomay, Ed. International Thomson Computer Press, 376–393.

PERSKY, Y. AND BEN-ARI, M. 1998. Re-engineering a Concurrency Simulator. In *ACM 3rd ITiCSE Conference*. 185–188.

POLLOCK, L. AND JOCHEN, M. 2001. Making Parallel Programming Accessible to Inexperienced Programmers through Cooperative Learning. In *ACM 32nd SIGCSE Technical Symposium*. 224–228.

PRICE, B. A. AND BAECKER, R. M. 1991. The Automatic Animation of Concurrent Programs. In *Proceedings of the First International Workshop on Computer-Human Interface*. 128–137.

REEK, K. A. 2002. The Well-Tempered Semaphore: Theme with Variations. In *ACM 33rd SIGCSE Technical Symposium*. 356–359.

ROBBINS, S. 2000. Experimentation with Bounded Buffer Synchronization. In *ACM 31st SIGCSE Technical Symposium*. 330–334.

ROBBINS, S. 2001. Starving Philosophers: Experimentation with Monitor Synchronization. In *ACM 32nd SIGCSE Technical Symposium*. 317–321.

ROBBINS, S. 2002. Exploration of Process Interaction in Operating Systems: A Pipe-Fork Simulator. In *ACM 33rd SIGCSE Technical Symposium.* 351–355.

ROMAN, G.-C., KENNETH C. COX, C. D. W., AND PLUN, J. Y. 1992. Pavane: a system for declarative visualization of concurrent computation. *Journal of Visual Languages and Computing 3*, 161–193.

SCHREINER, W. 2002. A Java Toolkit for Teaching Distributed Algorithms. In *ACM 7th ITiCSE Conference.* 111–115.

SHENE, C.-K. 1998. Multithreaded Programming in an Introduction to Operating Systems Course. In *ACM 29th SIGCSE Technical Symposium.* ACM, New York, 242 – 246.

SHENE, C.-K. 2001. Multithreaded Programming with ThreadMentor: A Tutorial. `http://www.cs.mtu.edu/~shene/NSF-3/e-Book/index.html`.

SHENE, C.-K. 2002. Multithreaded Programming Can Strengthen an Operating Systems Course. *Computer Science Education Journal 12,* 4 (December), 275–299.

SHENE, C.-K. AND CARR, S. 1998. The Design of a Multithreaded Programming Course and Its Accompanying Software Tools. *The Journal of Computing in Small Colleges 14,* 1, 12–24.

SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. 2002. *Operating System Concepts*, sixth ed. John Wiley & Sons.

STALLINGS, W. 2001. *Operating Systems*, fourth ed. Prentice Hall.

STASKO, J. T. 1990. TANGO: A Framework and System for Algorithm Animation. *Computer 23,* 9, 27–39.

STASKO, J. T. 1995. *The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report.* Tech. Rep. GIT-GVU-95-03, Georgia Institute of Technology, College of Computing.

STASKO, J. T. AND KRAEMER, E. 1993a. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing 18*, 258–264.

STASKO, J. T. AND KRAEMER, E. 1993b. The Visualization of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing 18*, 105–117.

TANENBAUM, A. S. 2001. *Modern Operating Systems*, second ed. Prentice Hall.

TOLL, W. E. 1995a. Decision Points in the Introduction of Parallel Processing into the Undergraduate Curriculum. In *ACM 26th SIGCSE Technical Symposium.* 136–140.

TOLL, W. E. 1995b. Socket Programming in the Data Communication Laboratory. In *ACM 26th SIGCSE Technical Symposium.* 39–43.

ZHANG, K., HINTZ, T., AND MA, X. 1999. The Role of Graphics in Parallel Program Development. *Journal of Visual Languages and Computing 10*, 215–243.

ZHAO, Q. A. AND STASKO, J. T. 1995. *Visualizing the Execution of Threads-based Parallel Programs.* Tech. Rep. GIT-GVU-95-01, George Institute of Technology, College of Computing. January.

ZIMMERMANN, M., PERRENOUD, F., AND SCHIPER, A. 1988. Understanding Concurrent Programming Through Program Animation. In *ACM 19th SIGCSE Technical Symposium.* 27–31.

...