A Visualization System for Multithreaded Programming*

Michael Bedy, Steve Carr, Xianglong Huang and Ching-Kuang Shene†
Department of Computer Science
Michigan Technological University
Houghton, MI 49931–1295
{mjbedy,carr,xlhuang,shene}@mtu.edu

1 Introduction

Almost all modern operating systems, from Windows to Unix, support multithreaded programming. To make sure our students can lead the trend of computer science in the foreseeable future, we need to introduce them to this important technology. However, we have found through experience in teaching multithreaded programming that the paradigm shift from sequential to multithreaded causes students significant problems, such as **(1)** multithreaded program development requires a new mindset, **(2)** multithreaded program behavior is dynamic, making debugging very difficult, and **(3)** proper synchronization is more difficult than anticipated [10, 11]. Moreover, detecting race conditions and deadlocks is more easily said than done.

To address these challenging problems, we have designed a visualization system that is capable of displaying vital information of a multithreaded program for students to understand the dynamic behavior, to grasp the inner working of popular synchronization primitives, and to help pinpoint race conditions and deadlocks. The subject of this paper is this system. In what follows, Section 2 reviews previous work related to ours; Section 3 provides a brief system overview; Section 4 presents the user interface; Section 5 discusses the thread status and history windows; Section 6 through Section 9 cover various supported synchronization primitives; and, finally, Section 10 has our conclusions.

*This work was partially supported by the National Science Foundation under grant DUE-9752244.

†Communicating author

2 Previous Work

There are two types of visualization systems: *algorithm animation* and *program behavior visualization*. The former animates the execution of an algorithm, while the latter displays the execution behavior of a program. The visualization part can be *real-time* or *post-mortem*. The real-time visualization generates visuals on-the-fly, while the post-mortem visualization saves the events and plays them back with another program.

There are no major differences between sequential and parallel algorithm animation, since the program saves the events in order, perhaps synchronized by a file writing protocol. Zimmermann, et al. [15] is perhaps the earliest paper on concurrent program animation, while Price and Baeker [9] suggest a framework. With Hartley's approach [3], the user program saves its activities to a file and then uses Stasko's XTANGO to playback *after* the program completes. Recently, Naps and Chan [7] incorporate parallel program animation into Multi-Pascal. One of the most well-known system for parallel program visualization is unquestionably Stasko's PARADE [12], which is based on POLKA. More details can be found in [4, 5, 13]. Along this line of research, Zhao and Stasko [14] present an environment for visualizing Pthreads programs using POLKA, and Cai [2] describes a system for OCCAM programs. Other useful information can be found in [6, 8].

Most freely available and well-known visualization systems are post-mortem, including XTANGO, POLKA and PARADE. The advantage of a post-mortem system is that everything relevant to the execution of a program has been saved and can be replayed at any time. However, its main disadvantages are: **(1)** only *one* instance of the program execution is available, making debugging difficult; **(2)** a large volume of output will be generated which could be incomplete or even corrupted if the program ends abnormally; and **(3)** the program must be "instrumented" explicitly by adding statements or directives, which introduces an extra level of complexity

and could interfere with the original behavior of the program.

3 System Overview

The goal of our system is to provide a tool for students to visualize thread execution and synchronization behavior. Since we hope this system will be used widely, it has to be highly portable. Moreover, to avoid the problems of post-mortem systems, we choose the real-time approach and use post-mortem as a backup. To achieve maximum portability, we use EGCS, the newest GNU compiler, for program development, and GTK for GUI programming. STL is also used extensively. As a result, we are able to support most popular platforms (*i.e.*, Windows 95/98/NT, Linux, and Sun Solaris).

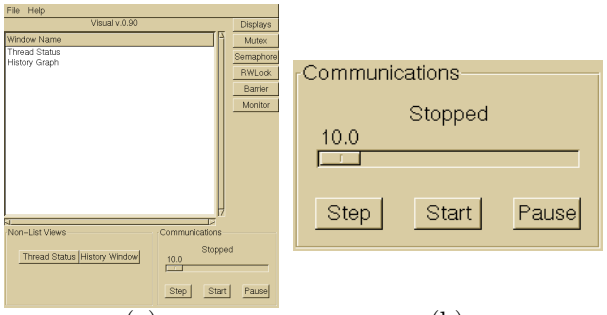
This system is a stand-alone program, and communicates with user programs by message passing. A student writes his/her multithreaded program in C/C++ and uses our class library for thread and synchronization operations [1]. All "instrumentations" are implicitly built into the class library and can be turned on or off. As a program runs, it activates the visualization system, and both are run in separate address spaces so that they will not interfere each other.

The visualization system supports thread creation, termination, and join. It also provides visual effects for popular synchronization primitives. These include mutex locks, semaphores, barriers, reader-priority and writer-priority reader-writer locks, and Hoare type and Mesa type monitors. Additionally, this system displays the status of a thread (*i.e.*, running, joining, waiting, and terminated), the running history of every thread, and source program with the line involved in a synchronization call highlighted. In this way, a student can easily grasp a global picture of his/her running threads.

4 The User Interface

When the system starts, it displays the Main Window (Figure 1(a)). The lower-left corner has buttons for activating the History Window and Thread Status Window. The upper-right corner has buttons for selecting the information to be displayed in the large *display area*.

The lower-right corner of the Main Window has a slide and three buttons: Step, Start and Pause (Figure 1(b)). Initially, the system suspends the execution of the running program. To run the program, click on Start; to step through the execution, click on Step; and to pause the execution, click on Pause. This system does not step through the execution of the instructions. Instead, it steps through the synchronization primitive calls. Thus, clicking on Step brings the execution to the next synchronization call. A user can use the slide to set the "delay" time for changing the "execution" speed.

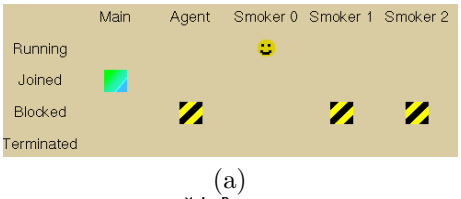


(a) (b)

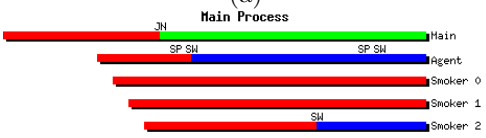
Figure 1: The Main Window and Slide

5 Thread Status and History

Clicking on the Thread Status button activates the Thread Status Window (Figure 2(a)) to display the status of each thread. The status of each thread is updated on-the-fly. Figure 2(a) shows a snapshot of the execution of the smokers problem. Thread `Agent` is waiting for the table; threads `Smoker 1` and `Smoker 2` are blocked, waiting for ingredients; `Smoker 0` is running (smoking); and the main thread `Main` is waiting for the completion of all four threads (joining).



(a)



(b)

Figure 2: Thread Status and History

The History Window displays the running history of each thread (Figure 2(b)). The "history" of a thread is represented by a color bar running from left to right. A different color indicates a different status of a thread. Tags such as `JN`, `SW` and `SP` shown on the top of each bar indicate what has happened to a particular thread. For example, after the agent and smoker threads are created, the main thread enters the *joining* status. Hence, a joining tag `JN` is displayed. Clicking on a tag activates the Source Window, which displays the program with the line containing the corresponding synchroniza-

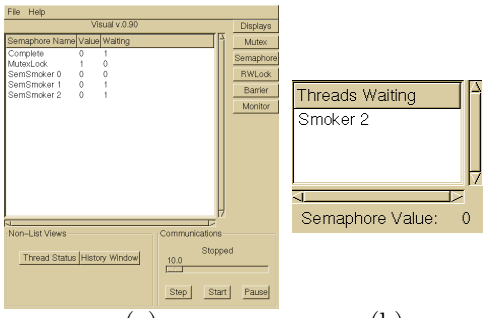
tion primitive call highlighted. Clicking on tag JN in Figure 2(b) yields Figure 3, where the line containing the join call is highlighted.

```
                Smokers[i]=new Smokerthread(name.str(), i);
                Smokers[i]->ThreadBegin();
    }

    Agent.Join();        // wait for agent to exit

    for (i=0; i<SMOKERNUM; i++)     // wait for smokers to exit
                Smokers[i]->Join();
}
```

Figure 3: The Source Window

6 Semaphores and Mutex Locks

Clicking on the Semaphore button displays semaphore information in the display area of the Main Window. Figure 4(a) shows all semaphores being used in the smokers program. Each semaphore occupies one line, and each line contains a semaphore name, its value, and the number of waiting threads. In the smoker program, thread `Agent` waits on semaphore `Complete` until the ingredients on the table are taken by a smoker, adds new ingredients on the table, and signals a smoker's semaphore (*i.e.*, `SemSmoker 0`, `SemSmoker 1` or `SemSmoker 2`) for the corresponding smoker to pick up the ingredients. In Figure 4(a), the counter of semaphore `SemSmoker 2` is zero with one thread waiting. Clicking on the line containing the semaphore activates a Semaphore Window (Figure 4(b)). In the figure the counter of semaphore `SemSmoker 2` is zero and thread `Smoker 2` is waiting.



(a) (b)

Figure 4: Semaphore Wait Windows

Associated with each semaphore, there are two tags in the History Window (Figure 2(b)): `SP` for semaphore post (or signal) and `SW` for semaphore wait. To look for the last call to semaphore wait, activate the History Window and click on the right-most `SW` tag of thread `Smoker 2`. The result is shown in Figure 5.

Since mutex lock is a binary semaphore, the content and format of a Mutex Lock Window are identical to

```
while (1) {
            ThrYield();
            SemSmoker[No]->Wait();  // smoker i wait at semaphore SemSmoker[i]
            MutexLock.Wait();
            if (Agentleft) {
                MutexLock.Signal();
                cout<<Space->str()<<ThreadName.str()<<" is quiting"<<endl;
                Exit();
            }
            MutexLock.Signal();
            cout<<Space->str()<<ThreadName.str()<<" is smoking"<<endl;
            Complete.Signal();  // finish smoking
}
```

Figure 5: The Source Window for Semaphore

those of a Semaphore Window. However, a lock/unlock icon replaces the semaphore value, and the owner of the lock is shown when the lock is locked. The two tags for mutex locks are `MW` for mutex wait and `MU` mutex unlock.

7 Monitors

Clicking on the Monitor button displays monitor information. Each monitor occupies one line, and each line displays the monitor's name, the number of threads waiting inside (*i.e.*, inactive) the monitor, the number of threads waiting to enter the monitor, and the running (*i.e.*, active) thread in the monitor. Clicking on a line brings up a Monitor Window which provides all of the detailed information of that monitor (Figure 6). This window has four components: (1) the top-row shows the active thread, (2) each box with black frame represents a condition variable and contains all threads that are currently blocked on it, (3) the box with green frame lists all inactive threads (*e.g.*, threads released from a condition variable but not yet running – re-entering), and (4) the left-side of the window lists all threads that are waiting to enter the monitor. This system supports two monitor types: the Hoare type – the signaler yields, and the Mesa type – the signaled threads yields. The type of the monitor is part of the window title.

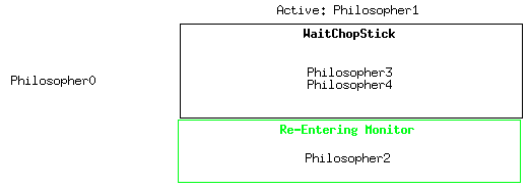
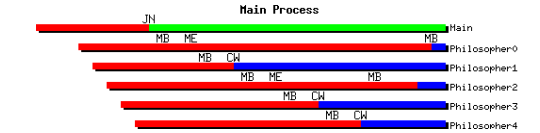


Figure 6: The Monitor Window

Tags for monitors include `MB` - monitor begins, `ME` - monitor ends, and `CW` - condition variable wait. Monitor begin and monitor end indicate that threads enter and leave the monitor, respectively. Figure 7(a) shows a snapshot of the execution of the dining philosophers problem. Clicking on a `CW` brings up the Source Window showing the line that contains the corresponding call to condition variable `WaitChopStick` (Figure 7(b)).



(a)

```
void PhilosopherMonitor::GetChopstick(int Number)
{
        Begin();
        while (!CanEat(Number)) WaitChopStick->Wait();
        ChopStick[Number]=0;                    // get two chopsticks
        ChopStick[(Number+1)%PHILOSOPHERNUM]=0;
        End();
}
```

(b)

Figure 7: Monitor History and Source Windows

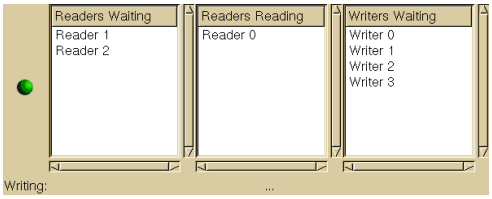
8 Reader-Writer Locks

A reader-writer lock, RW lock for short, is another way to prevent more than one thread from modifying shared data at the same time; however, unlike a mutex lock, it allows concurrent reading and exclusive writing. Clicking on the RWLock button displays RW lock information. Each RW lock uses one line, and each line contains the RW lock name, the numbers of waiting and reading readers, the number of waiting writers, and the active writer. Clicking on a line brings up the corresponding Reader-Writer Lock Window (Figure 8(a)). This system supports two types of RW lock: *reader-priority* and *writer-priority*. The former allows a writer to write only if there is no waiting readers, while the latter permits the writer to write after all *current* readers finish reading. The reader-priority type is simple; but, writers may starve. The type of the RW lock is shown as part of the window title. The Reader-Writer Lock Window has three sub-windows, showing the waiting and reading readers, and waiting writers. The thread name of the writing writer is displayed near the bottom if there is one; otherwise, "..." is shown. To the left of the window, a green sphere means no writer is writing, while a red sphere indicates the lock is owned by a writer.

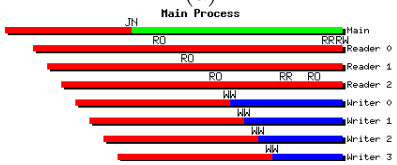
There are six RW lock tags: `RW` - reader wait, `RO` - reader own (reading), `RR` - reader release, `WW` - writer wait, `WO` - writer own (writing), and `WR` - writer release. Figure 8(b) is a snapshot of the History Window with various tags. With RW locks, solving both versions of the readers-writers problem becomes an obvious task.

9 Barriers

A barrier is a way to keep the members of a group of threads together. A barrier has a *capacity* value. Threads calling a barrier primitive are blocked until its capacity is reached. Then, all threads are released at the same time. This system also supports barriers. Clicking on the Barrier button displays barrier information.



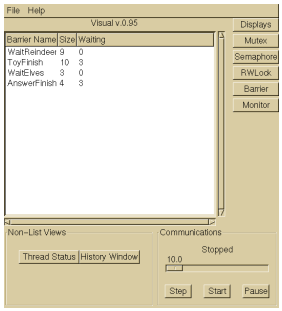
(a)



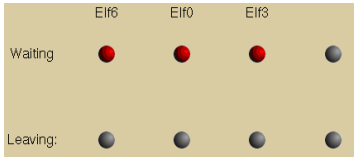
(b)

Figure 8: The Reader-Writer Lock and History Windows

Each barrier occupies one line in the display area, and each line contains a barrier name, its capacity, and the number of threads waiting in the barrier (Figure 9(a)).



(a)



(b)

Figure 9: The Main and Barrier Windows

Clicking on a line brings up the corresponding Barrier Window (Figure 9(b)). Each Barrier Window contains two rows of spheres, the *waiting* row and *leaving* row. The number of spheres on each row is equal to the capacity of the barrier. If a thread calls this barrier, its name is shown with a red sphere on the waiting row; when it is released, the sphere on the waiting (*resp.*, leaving) row becomes gray (*resp.*, green). Once a thread leaves the barrier, its corresponding sphere on the leaving row changes back to gray. In the figure. three threads `Elf6`, `Elf0` and `Elf3` are in the barrier `ToyFinish` (Figure 9(a)). The next thread calling this barrier will release all threads.

10 Conclusions

We have presented the important features of our thread visualization system. Combined with our class library [1], students will not only learn multithreaded

programming easily, but also vividly see the dynamic behavior of a running multithreaded program and the interaction of synchronization primitives. We plan to continue to expand this system to include deadlock detection and multiprocess programming, and support parallel and distributed programming visualization in the future. Our system will be available to the public after it becomes stabilized. The interested reader can find more about our work, including course material and future software announcements, at the following site:

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

References

[1] Bedy, M. J., Carr, S., Huang X and Shene, C.-K., A Class Library for Multithreaded Programming, 1999, submitted for publication.

[2] Cai, W., Milne, W. J. and Turner, S. J., Graphical Views of the Behavior of Parallel Programs, *Journal of Parallel and Distributed Computing*, Vol. 18 (1993), pp. 223–230.

[3] Hartley, S. J., Animating Operating Systems Algorithms with XTANGO, *ACM Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, 1994, pp. 344–348.

[4] Kraemer, E., Visualizing Concurrent Programs, in *Software Visualization*, edited by John Stasko, John Domingue, Marc H. Brown and Blaine A. Price, MIT Press, 1998, pp. 237–256.

[5] Kraemer E. and Stasko, J. T., The Visualization of Parallel Systems: An Overview, *Journal of Parallel and Distributed Computing*, Vol. 18 (1993), pp. 105–117.

[6] Miller, B. P., What to Draw? When to Draw? An Essay on Parallel Program Visualization, *Journal of Parallel and Distributed Computing*, Vol. 18 (1993), pp. 265–269.

[7] Naps, T. L. and Chan, E. E., Using Visualization to Teach Parallel Algorithms, *Thirtieth SIGCSE Technical Symposium on Computer Science Education*, 1999, pp. 232–236.

[8] Pancake, C. M., Visualization Techniques for Parallel Debugging and Performance-Tuning Tools, *Parallel Computing: Paradigm and Applications*, edited by A. Y. Zomay, International Thomson Computer Press, 1996, pp. 376–393.

[9] Price, B. M. and Baecker, R. M., The Automatic Animation of Concurrent Programs, *Proceedings of the First International Workshop on Computer-Human Interface*, 1991, pp. 128–137.

[10] Shene, C.-K., Multithreaded Programming in an Introduction to Operating Systems Course, *ACM Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, 1998, pp. 242–246.

[11] Shene, C.-K. and Carr, S., The Design of a Multithreaded Programming Course and Its Accompanying Software Tools, *The Journal of Computing in Small Colleges*, Vol. 14 (1998), No. 1, pp. 12–24.

[12] Stasko, J. T., The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report, Technical Report GIT-GVU-95-03, College of Computing, Georgia Institute of Technology, 1995.

[13] Stasko, J. T. and Kraemer, E., A Methodology for Building Application-Specific Visualizations of Parallel Programs, *Journal of Parallel and Distributed Computing*, Vol. 18 (1993), pp. 258–264.

[14] Zhao, Q. A. and Stasko, J. T., Visualizing the Execution of Threads-based Parallel Programs, Technical Report GIT-GVU-95-01, College of Computing, George Institute of Technology, January 1995.

[15] Zimmermann, M., Perrenoud, F. and Schiper, A. Understanding Concurrent Programming Through Program Animation, *ACM Nineteenth SIGCSE Technical Symposium on Computer Science Education*, 1988, pp. 27–31.