A Portable Class Library for Teaching Multithreaded Programming*

Steve Carr and Ching-Kuang Shene†
Department of Computer Science
Michigan Technological University
Houghton, MI 49931–1295, USA
Email: {carr|shene}@mtu.edu

1 Introduction

All modern operating systems support multithreaded programming (MTP). To ensure our students can lead the trend of computer science in the foreseeable future, we have been teaching MTP for four years [6]. Our experience shows that the paradigm shift from sequential to multithreaded causes students significant problems [7], such as **(1)** MTP requires a new mindset, **(2)** multithreaded program behavior is dynamic, making 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.

An ideal pedagogical system for teaching MTP is shown in Figure 1. A student program is first processed by a software metric system for the complexity measures of the program's level of parallelism and synchronization structures. Then, it is analyzed by a static analyzer for potential deadlocks and race conditions. At this stage, a student should have received sufficient information for improving his/her program. The program is compiled and run, and the visualization system is activated implicitly. A subsystem of the visualization system monitors if the running program has a deadlock and/or is in a safe state. The program's running activities can also be written to a file for post-mortem analysis.

The first step of realizing this system is to implement

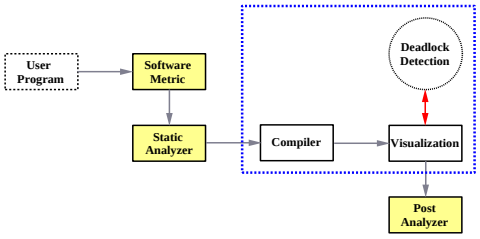

Figure 1: An Ideal Pedagogical System

the components shown in the dashed rectangle because they are the most fundamental parts that can directly affect student learning. This involves three components (Figure 2). The first is a class library that encapsulates frequently used thread functions and synchronization primitives into a number of easy-to-use classes. It will also provide a layer of abstraction for future extension to multiprocess, parallel and distributed programming. Thus, problem (4) above is addressed.

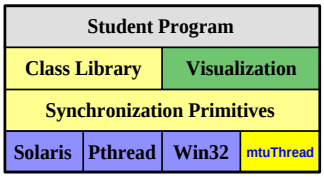

Figure 2: System Architecture

The visualization system provides an environment for visualizing the dynamic behavior of a threaded program and the involved synchronization primitives [1]. It runs as a separate process and communicates with the class library by passing messages that describe the execution behavior of a program. This is another advantage of using a class library since a student does not have to know the internal message communication protocol. The visualization system can display the status and execution history of each thread, and all relevant information for each synchronization primitive. For example, the visu-

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

alization system displays a list of mutex locks used in the user program and a particular lock can be selected to see its status (*i.e.*, locked/unlocked), its owner, and the contents of its waiting queue. This information is updated on-the-fly as the user program runs. A runtime deadlock detection subsystem is built into the visualization system. Hence, problem (2) and problem (3) mentioned earlier are also addressed. The unsolved and difficult part is race condition detection which is infeasible to be done in real time. One possible method is through static and/or post-mortem analysis.

The class library and visualization system are designed to sit on top of Solaris threads, Pthreads and Win32. We also designed a small user-level kernel mtuThread that supports MTP [2]. Currently, it runs on Solaris, SunOS, Linux, and Win32. By selecting an option, a student can compile his program using any one of the supported thread systems. When it is necessary, an instructor can use the source code of mtuThread to illustrate the implementation of multithreading.

There are commercial thread libraries available (*e.g.*, Microsoft MFC, Rogue Wave's `Threads.h++` and ObjectSpace's `Thread<ToolKit>`) for professional use with a steep learning curve. *Brown's Threads Package* [3] is a system similar to ours with the same design merit. However, this is a stand-alone class library rather than a component of a large system that can help students learn multithreaded, multiprocess, parallel and distributed programming. Java was not chosen due to its well-known shortcoming [4, 5].

The remaining of this paper presents a brief discussion of the features of our MTP class library (Section 2 and Section 3), and a summary of our experience (Section 4). Finally, Section 5 has our conclusions.

2 The Thread Class

Our class library provides an abstraction away from the system dependent details of MTP so that students can concentrate on designing and writing correct multithreaded programs. The most important class is `Thread`. A student defines a thread as a derived class of `Thread` and supplies a method `ThreadFunct()` to be run as a thread by calling method `Begin()`. A thread uses methods `Exit()`, `Join()`, `Yield()`, `Suspend()`, and `Continue()` to exit the system, to join with another thread, to relinquish the execution control, to suspend a thread, and to resume a suspended thread. Method `Delay()` delays its caller for a random number of context switches. Below is the main program of a solution to the smokers problem. It runs the agent thread `Agent` and three smoker threads `Smokers[0]`, `Smoker[1]` and `Smoker[2]`, and uses `Join()` to wait

```
class  SmokerThread: public Thread
{
     public:
          SmokerThread(...)  { ..... }
     private:
          void  ThreadFunct();
          .....
}

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

void  main(void)
{
     SmokerThread  *Smoker[3];
     AgentThread   Agent;
     .....
     Agent.Begin();
     for (i = 0; i < 2; i++) {
          Smoker[i] = new SmokerThread(...);
          Smoker[i]->Begin();
     }
     Agent.Join();
     for (i = 0; i < 2; i++)
          Smoker[i]->Join();
}
```

until all four threads complete.

3 Synchronization Primitives

Our class library supports mutex locks, semaphores, reader-writer locks, barriers, and monitors. Each synchronization primitive is also defined as a class. The following code implements the smoker thread and uses four semaphores. Semaphore `SemSmoker[i]` blocks smoker `i` and semaphore `Table` blocks the agent thread from adding ingredients on the *shared* table.

```
Semaphore  *SemSmoker[3], Table;

void  SmokerThread::ThreadFunct()
{
     for (...) {
          SemSmoker[ID]->Wait();
          Table.Signal();
          Delay();   // smoking
     }
}
```

Reader-writer locks provide a finer control over a shared resource than mutex locks, and mimic the readers-writers problem. A *reader* thread only reads the content of a shared resource, while a *writer* thread modifies that resource. Hence, readers can read simultaneously, but writers must write exclusively. There are two versions of reader-writer locks. The *reader-priority* version gives higher priority to readers and, as a result, writ-

ers may starve. The *writer-priority* version gives writers higher priority. More precisely, once a writer declares to write, all subsequent readers and writers must wait. The reader-writer lock class has four methods: `ReaderLock()`, `WriterLock()`, `ReaderUnlock()` and `WriterUnlock()`. A thread that calls `ReaderLock()` (*resp.*, `WriterLock()`) has non-exclusive (*resp.*, exclusive) access to the resource. A student must indicate the version of a reader-writer lock when it is created.

Our barrier class has three methods: (1) `Barrier()` constructs and initializes a barrier with a positive value, the barrier's *capacity*, (2) `~Barrier()`, and (3) `Wait()` blocks its caller until the number of blocked threads is equal to that barrier's capacity. Hence, barriers can be used to synchronize a group of threads.

The last primitive is the `Monitor` class that implements the classic monitor construct. The monitor class has two methods `MonitorBegin()` and `MonitorEnd()`, and a private class `Condition` that implements classic condition variables. Since C++ classes are *not* synchronized, to ensure that a monitor procedure is executed mutual exclusively, the first and last statements of a monitor procedure must be a call to `MonitorBegin()` and `MonitorEnd()`, respectively. Class `Condition` has methods `Signal()` and `Wait()` in addition to its constructor and destructor. A thread that calls a monitor procedure can be in one of the four possible states: (1) waiting in the entering queue, (2) waiting on a condition variable, (3) active (*i.e.*, running within a monitor), and (4) inactive (*e.g.*, released from a condition variable but not active). The constructor of `Monitor` accepts a *type* parameter for creating a monitor of Hoare or Mesa style. In the former, the signaler (*i.e.*, a thread that calls the `Signal()` method of a condition variable) becomes inactivate, while for the latter the signaler continues to be active and the one released from a condition variable becomes inactive. By supporting both styles, a student can experience and compare the differences between these two popular implementations.

```
class  ForkMonitor: public Monitor
{
     public:
          ForkMonitor(...)  { ..... };
          void  GetForks(int No), PutForks(int No);
     private:
          Condition  *Philos[5];
          int        Fork[5], CanEat(int No);
};
```

The above is a monitor class for the dining philosophers problem. A monitor must be declared as a derived class of `Monitor`. A philosopher can eat only if *both* forks are available. Each entry of condition variable `Philos[]` blocks the corresponding philosopher, and each entry of `Fork[]` holds either `USED` or `FREE`.

A private monitor procedure `CanEat()` tests if philosopher `No` can pick up *both* forks (see the program listing below). Monitor procedure `GetForks()` is called when a philosopher needs forks. If forks are not available, the calling philosopher blocks. When a philosopher finishes eating, he calls `PutForks()`, which releases both forks and wakes up all philosophers so that they can try again.

```
int  ForkMonitor::CanEat(int No)
{
     return (both forks are available) ? 1 : 0;
}

void  ForkMonitor::GetForks(int No)
{
     MonitorBegin();
          while (!CanEat(No))
               Philos[No]->Wait();
          set both forks to USED;
     MonitorEnd();
}

void  ForkMonitor::PutForks(int No)
{
     MonitorBegin();
          set both forks to FREE;
          for (i = 0; i < 5; i++)
               Philos[i]->Signal();
     MonitorEnd();
}
```

4 Experience

We have been teaching MTP in a junior-level introduction to operating system course eight times in the past four years. This course has two tracks, *theory* and *programming*. The programming track, which focuses on MTP, consumes about 30% of a 10-week quarter. Initially, we used Pascal-FC, and switched to the SunOS light-weight process library. Two years ago, due to a system upgrade, we switched to the Solaris thread library. A major problem of using a system-level interface is that students frequently struggle with the complex meaning of the parameters, although most can be default values. Moreover, the mapping between the system interface and the textbook discussion may not be one-to-one. This is why a class library that completely reflects the *simple* textbook type interface is helpful.

To build up a student's mindset for MTP, there are five programming assignments and one two-week mini-project. The first program is for warm-up only. We normally use matrix multiplication or quicksort; however, any problem that does not require synchronization will

fit our purpose. With this program, students learn how to partition a computation task into subtasks, each of which is handled by a thread. Only thread creation, join and exit are involved.

Problem 2 to problem 4 cover semaphores, monitors, and message passing. We have found that the semaphore assignment is usually the most challenging one for the following reasons: (1) semaphores and mutex locks are unstructured low-level primitives; (2) students are not used to properly coordinating tasks in different threads, especially for protecting shared data items; (3) students directly apply sequential programming techniques to their threaded programs and, as a result, race conditions occur frequently (*e.g.*, fail to protect a shared counter); (4) the counting mechanism built into a semaphore is in general not appreciated fully, causing students to use additional semaphores for protecting unnecessary counters; and (5) students create very large critical sections that in effect serialize the whole program. We have used many different programming problems in order to identify the causes of these problems. Unfortunately, except for the easiest ones (*e.g.*, the smokers problem), these problems persist. On the other hand, most students do not have a serious problem if the same assignment is redone using monitors. This is perhaps because monitors provide a well-structured mechanism and are easier to understand. Pipelined type problems (*e.g.*, parallel sieve, parallel insertion sort, and backward substitution) are used for students to write message passing programs. Surprisingly, few students encounter difficulty.

After these four programs, most students have been well-prepared for MTP, although the use of semaphores still proves to be a challenge. Then, we turn to the implementation of a small user-level kernel that supports MTP. The concept and implementation of coroutines using `setjmp()` and `longjmp()` are discussed, and a programming assignment is given. Finally, in a two-week period, students are given a stripped down version of `mtuThread` and are asked to complete the system with simple extensions. For example, they may be asked to implement thread join, suspend and continue, semaphores, and mailboxes. Or, they may be asked to extend the thread scheduler to handle priority scheduling and priority inversion. Since students have had more than eight weeks experience in MTP, most of them complete this mini-project successfully.

Due to time constraints, students do not use barriers and reader-writer locks. To make sure they are aware of these two useful primitives, the implementation and use of barriers is usually an exam problem. The reader-priority version of reader-writer locks is used to solve the readers-writers problem, while understanding the writer-priority version is part of a weekly reading assignment. Thus, the use and implementation of each primitive is covered to certain level of depth. We believe that this approach can help students understand the merit and skills of MTP and successfully adjust their mindset by the end of this course.

5 Conclusions

We have presented important features of our class library for MTP, and our classroom experience. Our class library combined with the visualization system will not only help students learn MTP more easily, but also allow them to vividly see the dynamic behavior of a running threaded program and the interaction of synchronization primitives. In the future, we will extend this class library to support multiprocess, parallel and distributed programming, and to complete the ideal system in Figure 1. Our system will be available to the public after it becomes stabilized. The interested readers 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 Visualization System for Multithreaded Programming, to appear in *31st SIGCSE Technical Symposium*, 2000.

[2] Bedy, M. J., Carr, S., Huang X. and Shene, C.-K., The Design and Construction of a User-Level Kernel for Teaching Multithreaded Programming, to appear in *Frontiers in Education*, 1999.

[3] Doeppner Jr., T. W., *The Brown C++ Threads Package*, Version 2.0, Dept. of Computer Science, Brown University, 1996.

[4] Brinch Hansen, P., Java's Insecure Parallelism, *ACM SIGPLAN Notices*, Vol. 34 (1999), No.4 (April), pp. 38–45.

[5] Hartley, S. J., "Alfonse, Wait Here for My Signal!" *30th SIGCSE Technical Symposium*, 1999, pp. 58–62.

[6] Shene, C.-K., Multithreaded Programming in an Introduction to Operating Systems Course, *29th SIGCSE Technical Symposium*, 1998, pp. 242–246.

[7] 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.