A Class Library for Multithreaded Programming*

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

1 Introduction

Almost all modern operating systems 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. We have found through experience in teaching multithreaded programming that the paradigm shift from sequential to multithreaded causes students significant problems [16, 17], such as **(1)** multithreaded program development 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.

To overcome these problems, we have developed a class library with a simplified and portable programming interface. Coupled with our visualization system [1], which displays vital information of a running threaded program on-the-fly, teaching and learning multithreaded programming will be easier than ever. This paper focuses on the design and features of our class library. In what follows, Section 2 reviews our and related work; Section 3 discusses system design issues; Section 4 presents the `Thread` class; Section 5 through Section 8 cover various supported synchronization primitives; and, finally, Section 9 has our conclusions.

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

2 Related Work

Although a typical operating systems textbook covers synchronization primitives, its discussion is usually too generic to be applied directly to multithreaded programming. There are publications addressing the need of multithreaded/multiprocess programming in undergraduate education. Berk [4] simplified SunOS lightweight process library calls; Higginbotham and Morelli [12] used the Unix IPC interface; Bynum and Camp [6] developed a language similar to Pascal-FC [5], which is based on Ben-Ari's well-known Pascal compiler [3]; and Kurtz, et al. employed essentially the same approach [13]. Hartley, initially, used SR [9], and later switched to Java [10, 11]. Doeppner Jr.'s Brown C++ threads package [7] has a similar merit to ours. However, it is a standard-alone package rather than designed as part of an integrated system (Figure 1).

Unfortunately, these studies only focus on designing or using a language for teaching multithreaded programming. As pointed out in Section 1 and [16, 17], this approach is insufficient to help students understand the dynamic behavior of multithreaded programs. Detecting race conditions and deadlocks is difficult for beginners, and writing well-structured and efficient multithreaded programs is even more challenging. To address these problems, a system must be able to **(1)** provide an easy to use interface, **(2)** help students visualize the dynamic behavior of a multithreaded program, **(3)** uncover potential race conditions and deadlocks, and **(4)** analyze a program and report the complexity of thread structure and synchronization primitives.

An ideal system is shown in Figure 1. A student's program is analyzed by a *software metric* subsystem for measuring the complexity and level of parallelism, followed by a *static analyzer* which reports potential race conditions and deadlocks. Then, the program is compiled and run. While the program is running, a visualization system displays the behavior of the program and its synchronization primitives on-the-fly and detects

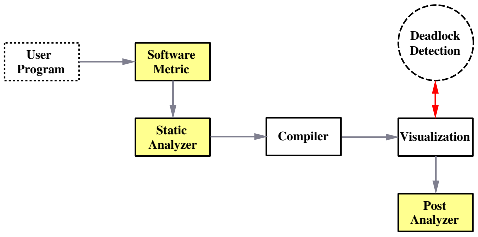

Figure 1: An Ideal System

deadlocks in real-time. Crucial data are saved to files for a *post analyzer* to perform post-mortem analysis. Note that statically detecting race condition (*resp.*, deadlock) is NP-complete [15] (*resp.*, NP-hard [14]), and heuristic algorithms are not very accurate. However, it is useful and helpful to students if these potential problems can be pinpointed *before* running a program.

As the first step toward an implementation of this system, we focus on building an easy-to-use interface and a visualization system. The latter is discussed in [1] and the subsequent sections concentrate on the former.

3 System Design

Our system consists of two major subsystems: a C++ class library and a visualization subsystem. The former hides as much system detail as possible and provides the user with a unified and portable programming interface, while the latter provides the user with a visualization environment to see the behavior of an executing program. The architecture of our system is shown in Figure 2. A user program creates threads with the class library; but it does not deal directly with the visualization system. Instead, the visualization system is activated by the class library implicitly and runs in a separate address space to minimize the interference among programs. The class library and the visualization system communicate by passing messages. Below the class library is a layer of synchronization primitives that includes mutex locks, semaphores, reader-writer locks, barriers, condition variables and monitors.

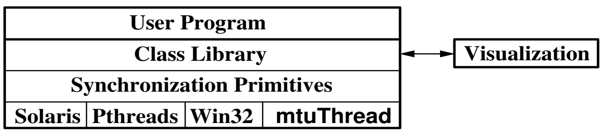

Figure 2: System Architecture

One of our goals is to design a system that can be widely used in a set of diverse environments. Hence, our system is designed to sit on top of Solaris threads, Pthreads and Win32. Since many instructors feel that a set of working code for a user-level thread system might help the students understand the concepts and implementation, we have also included a user-level kernel mtuThread. Details of the visualization system and mtuThread can be found in [1] and [2], respectively.

While Java is portable and has been used to teach concurrent programming [10], we use C++ because Java does not support multiprocess programming very well, making the implementation of running the user program and visualization system in separate address spaces difficult. Moreover, Java does have some serious problems in supporting the traditional concept of monitors [8, 11].

4 The Thread Class

Our class library provides an abstraction away from the system dependent details for its users to concentrate on designing and writing correct multithreaded programs. The most important class is `Thread` (Figure 3). A user supplies a function `ThreadFunc()` to be run as a thread and uses `Begin()` to run it. A thread uses `Exit()`, `Join()`, `Yield()`, `Suspend()`, and `Continue()` to exit the system, to join with another thread, to relinquish the execution control, to suspend the execution of a thread, and to resume the execution of a suspended thread. Function `Delay()` delays the thread for a random number of context switches.

Class `Thread`:

`Begin()`	**start running this thread**
`Exit()`	**terminate this thread**
`Join()`	**join with another thread**
`Yield()`	**yield execution control**
`Suspend()`	**suspend this thread**
`Continue()`	**resume a suspended thread**
`ThreadFunc()`	**function to be run as a thread**
`Thread()`	**constructor**
`~Thread()`	**destructor**
`Delay()`	**delay the execution**
`GetID()`	**retrieve the thread ID**

Figure 3: Class `Thread`

In fact, a user must define a thread as a derived class of `Thread`. Figure 4 is part of the main program in a solution to the smokers problem. It runs the agent thread `Agent` and three smoker threads `Smokers[0]`, `Smoker[1]` and `Smoker[2]`. After creating and running all four threads, the main program uses `Join()` to wait until all four threads complete.

5 The Mutex and Semaphore Classes

Class `Semaphore` implements counting semaphores (Figure 5). We only provide `Wait()` and `Signal()` to the students without other popular capabilities such as

```
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();
}
```

Figure 4: Thread Creation and Joining

TryWait() and GetCount(). Our experience shows that students tend to use TryWait() frequently, creating inefficient busy waiting. The use of GetCount() is even riskier, because students use it for testing if a specific number of threads are waiting rather than developing more concrete and efficient algorithms.

Class `Semaphore`

`Wait()`	**wait on this semaphore**
`Signal()`	**signal this semaphore**
`Semaphore()`	**constructor**
`~Semaphore()`	**destructor**

Class `Mutex`

`Lock()`	**lock this mutex lock**
`Unlock()`	**unlock this lock**
`Mutex()`	**constructor**
`~Mutex()`	**destructor**

Figure 5: Classes `Semaphore` and `Mutex`

Figure 6 shows a simplified version of the smoker thread. Each smoker waits on semaphore `SemSmoker[]` for the ingredients. The agent, after adding the ingredients on the table, signals the corresponding smoker. Then, that smoker takes the ingredients, signals the agent so that the agent can add ingredients on the table, smokes for a while simulated by function `Delay()`, and goes back waiting for ingredients.

Class `Mutex` implements mutex locks. The thread that locks the mutex lock successfully becomes the owner of the lock. Only the owner can unlock the lock; but

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

void  SmokerThread::ThreadFunc()
{
   while (1) {
      SemSmoker[ID]->Wait();
      Table.Signal();
      Delay();        // smoking
   }
}
```

Figure 6: The Use of Semaphores

cannot lock recursively.

6 Reader-Writer Locks

Although mutex locks implement mutual exclusion, it is not always the most efficient way because reading shared data does not have to be mutually exclusive. In fact, reading can happen concurrently, while writing must be done exclusively. This is the well-known readers-writers problem. In practice, it is usually referred to as the readers-writers lock, RW lock for short. A thread that reads the shared data is a *reader*, while a thread that writes information into the shared data is a *writer*. A reader (*resp.*, writer) uses reader lock and reader unlock (*resp.*, writer lock and writer unlock) to lock and unlock a RW lock. There are two types of RW locks: *reader priority* and *writer-priority*. A reader-priority RW lock blocks writers as long as there is a reader reading. As a result, writers may be starved. A writer-priority RW lock allows a waiting writer to write once all *current* readers finish reading. Figure 7 shows the class `RWLock`. An argument to the constructor specifies if the RW lock being created is of reader-priority or writer-priority type. With RW locks, it is easy to solve the readers-writers and similar problems.

Class `RWLock`

`ReaderLock()`	**a reader locks this lock**
`WriterLock()`	**a writer locks this lock**
`ReaderUnlock()`	**a reader unlocks this lock**
`WriterUnlock()`	**a writer unlocks this lock**
`RWLock()`	**constructor**
`~RWLock()`	**destructor**

Figure 7: Class `RWLock`

7 Barriers

The barrier is also an extension to the conventional mutex lock. A barrier is a way to keep the members of a group of threads together. Figure 8 shows the class `Barrier`. A user must supply the constructor a *capacity* value. Threads that call `Wait()` block until the number of waiting threads reaches the capacity value. At that moment, all threads that are in the barrier return. One of these threads, the last one that entered the barrier, returns 1, and all others return 0. As a result, the thread that receives a returned value 1 will have a chance to perform some cleanup tasks.

Class `Barrier`	
`Wait()`	**wait on this barrier**
`Barrier()`	**constructor**
`~Barrier()`	**desctrutor**

Figure 8: Class `Barrier`

Consider the matrix multiplication problem. Given a L×M matrix A and a M×N matrix B, their multiplication is a L×N matrix C. Each entry of C is associated with a thread for computing the "inner product" of the i-th row of A and the j-th column of B. After all threads complete, the main program prints C. One can solve this with thread join; but, it can also be solved with a barrier as shown in Figure 9. When a thread is created, the main program sends in the row number i and column number j for this thread to compute the "inner product" of A's row i and B's column j. After creating all threads, the main program waits on semaphore `MainProg` with initial value 0. `GetTogether` is a barrier of capacity L∗N. Each thread computes the result into `C[i][j]` and then waits on the barrier. The last returned thread signals the main program so that the result can be printed without creating a race condition.

```
Barrier   GetTogether;
Semaphore MainProg;

void  Multiply::ThreadFunc(..)
{
   for (int k = 0; k < M; k++)
      c[i][j] += a[i][k]*b[k][j];
   if (GetTogether.Wait() == 1)
      MainProg.Signal();
}
```

Figure 9: The Use of Barrier

8 Monitors

Class `Monitor` implements the monitor construct (Figure 10). It has a private class `Condition` for condition variables that can only be used *within* a monitor. Functions `MonitorBegin()` and `MonitorEnd()` must be the first and the last statements of a monitor procedure to ensure mutual exclusion. A user may supply a *type* to the constructor. Hoare's type forces the signaler to wait, while Mesa's type lets the signaler continue. Note that under Mesa's type, a reactivated thread must recheck the condition to determine if it can continue, since the condition that forced it to wait could be changed by other threads in the period between which it is signaled and run.

Class `Monitor`	
`MonitorBegin()`	**enter a monitor procedure**
`MonitorEnd()`	**exit a monitor pocedure**
`Monitor()`	**constructor**
`~Monitor()`	**destructor**
Class `Condition`	
`Wait()`	**wait on this cond. var.**
`Signal()`	**signal this cond. var.**
`Condition()`	**constructor**
`~Condition()`	**destructor**

Figure 10: Classes `Monitor` and `Condition`

Figure 11 is part of 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 he can pick up *both* forks at the same time. Each entry of condition variable `Philos[]` blocks the corresponding philosopher, and each entry of `Fork[]` holds either `USED` or `FREE`.

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

Figure 11: Monitor Class `ForkMonitor`

A private monitor procedure `CanEat()` tests if philosopher `No` can pick up *both* forks (Figure 12). Monitor procedure `Get()` is called when a philosopher needs forks. If forks are not available, the calling philosopher blocks. When this philosopher finishes eating, he calls `Put()`, which releases both forks, and wakes up all philosophers so that they can try again. Note that a signal to a condition variable without waiting threads is ignored and hence has no lasting effect.

9 Conclusions

We have presented the important features of our class library for multithreaded programming. Combined with our visualization system [1], students will not only learn multithreaded programming more easily, but also vividly see the dynamic behavior of a running program and the interaction of synchronization primitives. In the future, we plan to extend this class library to support

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

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

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

Figure 12: Monitor Procedures `Get()` and `Put()`

multiprocess programming, and 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] M. J. Bedy, S. Carr, X. Huang and C.-K. Shene, A Visualization System for Multithreaded Programming, 1999, submitted for publication.

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

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

[4] T. S. Berk, A Simple Student Environment for Lightweight Process Concurrent Programming under SunOS, *ACM Twenty-Seventh SIGCSE Technical Symposium*, 1996, pp. 165–169.

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

[6] B. Bynum and T. Camp, After You, Alfonse: A Mutual Exclusion Toolkit, *ACM Twenty-Seventh SIGCSE Technical Symposium*, 1996, pp. 170–174.

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

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

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

[10] S. J. Hartley, "Alfonse, Your Java Is Ready!" *ACM Twenty-Ninth SIGCSE Technical Symposium*, 1998, pp. 247–251.

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

[12] C. W. Higginbotham and R. Morelli, A System for Teaching Concurrent Programming, *ACM Twenty-Second SIGCSE Technical Symposium*, 1991, pp. 309–316.

[13] B. L. Kurtz, H. Cai, C. Plock and X. Chen, A Concurrency Simulator Designed for Sophomore-Level Instruction, *ACM Twenty-Ninth SIGCSE Technical Symposium*, 1998, pp. 237–241.

[14] S. P. Masticola and B. G. Ryder, Static Infinite Wait Anomaly Detection in Polynomial Time, LCSR-TR-114, Laboratory for Computer Science Research, Rutgers University, 1990.

[15] R. H. B. Netzer and B. P. Miller, On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions, *International Conference on Parallel Processing*, August 1990, pp. II93–II97.

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

[17] C.-K. Shene and S. Carr, 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.