The Design and Construction of a User-Level Kernel for Teaching Multithreaded Programming

Michael J. Bedy Steve Carr Xianglong Huang Ching-Kuang Shene
Department of Computer Science
Michigan Technological University
Houghton, MI 49931–1295, USA

***Abstract*— Multithreading is a powerful programming paradigm that has become very popular in recent years. The authors have developed a set of course materials and software tools for effectively teaching multithreaded programming (MTP). One important component of the authors' system is a very simple user-level kernel for instructors to teach MTP without getting into system details, and for the students to add extensions. This paper presents the design and implementation of this kernel as well as its use in the classroom. This minimal user-level kernel employs a £rst-come-£rst-served scheduling policy, and permits a user to create and join threads, and use mutex locks. With this kernel, students are able to implement semaphores, barriers, reader-writer locks, mail-boxes and condition variables. This approach has two advantages: (1) students can easily learn the basics and internal of a kernel that supports MTP, and (2) conventional debuggers can be used for debugging purposes, because the kernel is a user-level program.**

I. Introduction

Multithreading is a powerful programming paradigm that is becoming very popular. Most operating systems already support this capability and the POSIX standard includes a multithreaded extension called Pthreads. This powerful programming paradigm has been incorporated into our CS270 Introduction to Operating Systems course for three years [9]. In this course we have found that teaching multithreaded programming (MTP) is in general dif£cult because it requires a paradigm shift from sequential programming. Thread synchronization always causes problems. To address the problems associated with the paradigm shift to MTP, we are developing, with the support of the NSF, a set of course materials and software tools for effectively teaching MTP [10].

This paper focuses on the design of a portable user-level kernel that supports MTP. Section II and Section III present the system architecture and design objectives, respectively. Section IV discusses important system data structures, Section V focuses on the implementation of context switching as a set of coroutines, and Section VI describes the set of low-level functions of the kernel. Section VII covers supported synchronization primitives, and Section VIII presents some experience in teaching MTP with an older system and experience with the bene£t of using the new kernel. Finally, Section IX has our conclusions.

Please send correspondence to the fourth author.

II. System Architecture

Our system consists of two major subsystems: a set of classes that hides as much system detail as possible and a visual subsystem that provides the user with a visualization environment to see what is happening during user-program execution (Figure 1). A user program creates threads with the class wrappers; but it does not deal directly with the visual subsystem. Instead, the visual subsystem is activated by the class wrappers implicitly and runs in a separate address space to minimize the interference among programs. The class wrappers and visual subsystem communicate with each other by sending messages. Below the class wrappers and the visual subsystem is a layer of synchronization primitives that includes mutex locks, semaphores, mailboxes, reader-writer locks, barriers, condition variables and monitors.

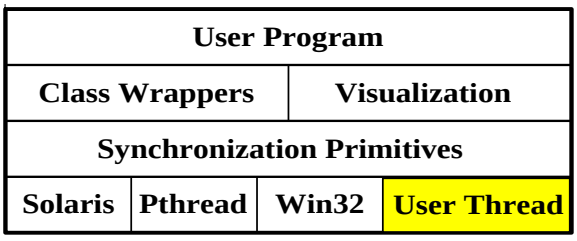

Fig. 1. System Architecture

One of our major goals is to design a portable system so that it can be widely distributed and used in a set of diverse environments. As a result, our system can sit on top of Solaris threads, Pthreads or Win32. Since many instructors feel that providing students with a set of working code for a user-level thread system helps the students understand the concepts and implementation, we have also implemented a user-level kernel that supports MTP. See the shaded part of Figure 1. The user-level kernel is the subject of this paper.

III. Design Objectives

Since the user-level kernel is built not only for supporting our system but also for providing a working example to students, we single out two important design factors: simplicity and no assembly language. Simplicity is in general easier than not using assembly language because a minimal kernel can be implemented in about 200 lines. Unfortunately, portability is extremely difficult due to the differences among compilers and operating systems (*e.g.*, run-time stack management). After examining a number of systems which only use C/C++ ([1], [5], [6], [7] and [8]), we have found that none of them is truly portable. Hence, we have decided to keep the use of assembly language to a minimum level, preferably to a level of only a few instructions for implementing the most critical part (*i.e.*, establishing environments). To further simplify this kernel, we have chosen a non-preemptive and first-come-first-served scheduling policy. Mutex locks are included as examples.

IV. Data Structures

Each thread is controlled by a thread control block which records the thread identifier, the status of the thread (*i.e.*, `RUNNING`, `READY`, `SUSPENDED`, `JOINING`, `WAITING` and `TERMINATED`), stack size and so on. There is only one thread in the `RUNNING` state at nay particular moment in time. Ready-to-run threads are in the *ready queue* (Figure 2). Suspended threads are in the *suspended queue*, while threads that are waiting on semaphores and other synchronization primitives are in one of the *waiting queues*. Threads waiting for the completion of a thread will be chained into that thread's *joining queue*. When the thread to be joined terminates, all threads in its joining queue are moved to the ready queue.

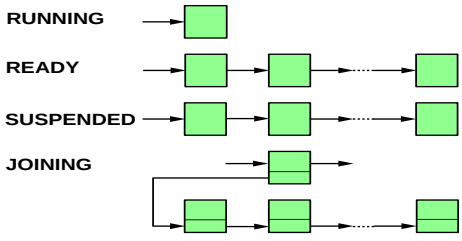

Fig. 2. System Data Structures

V. Context Switching

This section describes a possible way of simulating multitasking with coroutines, maintaining stack allocation, and performing context switching with two ANSI C library functions `setjmp()` and `longjmp()`.

A. Coroutines

Coroutines provide the execution model that we need to perform multithreading. When re-entering a coroutine, the execution starts at the instruction following the previous exit point. In Figure 3, when entering coroutine `B` from coroutine `A` the first time, the execution starts at the first instruction of `B` and exits at c. When re-entering `B`, execution starts at the instruction following c.

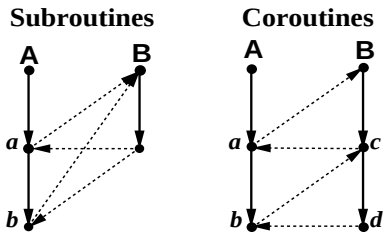

Fig. 3. Subroutines and Coroutines

All threads in this system form a big coroutine structure. The entry/exit points of threads are the places where rescheduling is necessary. More precisely, When a thread enters the waiting state, requests a join, executes a yield, or terminates, its execution is interrupted and the scheduler dispatches the control to another thread. When a thread becomes eligible to run again (*i.e.*, re-enter), the scheduler starts the execution at the next instruction of the interrupted (*i.e.*, exit) point.

B. Setjmp() and longjmp()

Switching between threads is done with a pair of ANSI C library functions `setjmp()` and `longjmp()`. Both access a *jump buffer* of type `jmp_buf`. Function `setjmp()` takes a jump buffer argument, saves the current environment (*i.e.*, context) into this jump buffer, and returns 0. Function `longjmp()` takes two arguments, a jump buffer and an integer, and transfers the execution back to the location recorded in the jump buffer, restores the environment, and forces the corresponding `setjmp()` to return the second argument of `longjmp()`.

```
jmp_buf  JBUF;
  ..........
if (setjmp(JBUF) == 0)
     Block 1
else
     Block 2
  ..........
longjmp(JBUF,1);
```

The call to `setjmp()` in the `if` statement saves the current environment into jump buffer `JBUF` and returns 0. After setting up `JBUF`, the program executes statements in Block 1, followed by other statements and perhaps function calls.

When the call to `longjmp()` is executed, as long as the environment saved in jump buffer `JBUF` is still valid, the execution is brought back to the call to `setjmp()` in the `if` statement and forces `setjmp()` to return the value of the second argument of `longjmp()`. In the above example, the execution ¤ow goes back to `setjmp()` and forces it to return 1. As a result, statements in Block 2 are executed. Note that `setjmp()` and `longjmp()` do not have to be in the same function and hence this type of transferring of control is usually referred to as a *non-local goto*.

C. Managing Contexts

Unfortunately, there is a problem. Executing `longjmp()` could unwind the stack pointer. More precisely, suppose function `A()` sets up a jump buffer `JBUF` with `setjmp()` and then calls function `B()`. Note that before `B()` is called, its stack frame does not exist and as a result `JBUF` does not include any information about `B()`'s stack frame. If `B()` executes a `longjmp()` sending the control back to where `JBUF` indicates, the stack frame of `B()` may be lost. Thus, executing a `longjmp()` back to `B()` could become impossible. To overcome this problem, we need to maintain the stack allocation information (*i.e.*, the stack and stack frame pointers), which can frequently be done with only a few assembly language instructions. Therefore, different platforms and even different operating systems on the same platform require a different set of instructions. Currently, this part is based on Steve Crane's REX system [4] and supports SunOS, Sun Solaris, SGI, Linux, and Windows 95/98/NT. In fact, this is the only non-portable part of our system and is used only in thread creation. Each platform/system has its own assembly language £le which contains less than 10 assembly instructions.

The best solution to the above mentioned problem is to allocate a separate stack for each thread. More precisely, when creating a thread, the user must indicate the stack size for running the thread and all functions called under this thread. This stack space should not be in the traditional £rst-in-last-out space, because it can be unwound in function call/return sequence. In this kernel, to create a thread, a stack space is allocated from the heap and the stack pointer is set to the beginning of this area. After setting up the stack pointer, the function that runs as a thread is called and as a result its stack space will be in the newly allocated area and all subsequent function calls will have their stack frames there. Since this space is allocated from the heap, it can only be freed explicitly. Therefore, as long as a function does not return, any jump buffer set up within that function will have a correct context. Since setting and saving the stack pointer cannot be performed directly with a high-level language, assembly language subroutines or inline assembly language instructions are required.

D. Thread Scheduler

Since this kernel uses a non-preemptive scheduling policy, its scheduler is quite simple. Before a thread relinquishes the control of CPU, it calls `setjmp()` to store the current environment into a jump buffer, which is part of its thread control block, and calls the thread scheduler:

```
/* before switching out */
if (setjmp(&(Running->Context)) == 0) {
     move control block to the ready Q;
     THREAD_SCHEDULER(); /* the scheduler */
}
/* switched back */
```

The running thread's control block is pointed by `Running`. The call to `setjmp()` saves the current environment into the `Context` member of the running thread's control block. This control block is moved to the ready queue followed by a call to the scheduler. The scheduler never returns. Instead, it uses `longjmp()` to send control to other threads. A simpli£ed scheduler is implemented as follows. It runs the £rst thread in the ready queue by executing a `longjmp()` to its `Context` member. Thus, the control ¤ow goes back to that thread's `setjmp()` call (see below), and executes the statement following the `if` statement.

```
move the first in ready Q to running;
longjmp(Running->Context, 1);
```

VI. Low-Level Functions

In addition to the scheduler discussed in Section V, there are queue manipulating functions. All of these functions are `static` so that they are invisible to the user. The most important user callable functions are the following four:

- `THREAD_SYS_INIT()`: This function must be called before calling any other thread related functions. It initializes the coroutine structure.
- `THREAD_CREATE()`: This function takes a pointer to a function and a pointer to a data item, allocates a stack area from the heap, and runs the function with the given argument. If creation is successful, it returns the identi£er of the newly created thread. The new thread will be put into the ready queue.
- `THREAD_EXIT()`: This function terminates the calling thread and cleans up stack frames on heap. If this thread has any joining threads, all of them will be moved to the ready queue.
- `THREAD_JOIN()`: This function takes a thread identi£er and puts the calling thread into the joining queue of the indicated thread. If the thread to be joined does not exist, this function returns.

There are three more low-level functions:

- `THREAD_YIELD()`: This function moves the calling thread to the ready queue and allows another thread to run. Thus, a context switching occurs.

- THREAD_SUSPEND(): This function takes a thread identifier and suspends the execution of the corresponding thread (*i.e.*, moves that thread to the suspended queue).
- THREAD_CONTINUE(): This function takes a thread identifier and moves the corresponding thread from the suspended queue to the ready queue.

With THREAD_SUSPEND() and THREAD_CONTINUE(), a user can build his/her own scheduler.

The following is a simple example that prints "Ping - Pong - Ping - Pong - ..." or "Pong - Ping - Pong - Ping - ..." depending on the first run thread. There are two functions Ping() and Pong() to be run as two threads. After printing "Ping -" or "Pong -", THREAD_YIELD() is called, yielding the CPU to the other thread. Thus, an alternating execution is implemented.

```
void  Ping(void *ptr)
{
     int i;
     for (i = 1; i <= 5; i++) {
          printf("Ping - ");
          THREAD_YIELD();
     }
     THREAD_EXIT();
}

void  Pong(void *ptr)
{
     int  i;
     for (i = 1; i <= 5; i++) {
          printf("Pong - ");
          THREAD_YIELD();
     }
     THREAD_EXIT();
}
```

To create these two threads, THREAD_CREATE() is called twice with the function names. It returns the identifier of the created thread so that a join can be performed. While the two threads are executing, the main program is blocked by the first join. The main program becomes ready-to-run when both threads terminate.

```
THREAD_ID  id1, id2;

THREAD_SYS_INIT();
id1 = THREAD_CREATE(Ping, NULL, ...);
id2 = THREAD_CREATE(Pong, NULL, ...);
THREAD_JOIN(id1);
THREAD_JOIN(id2);
```

Note that even though our kernel uses a non-preemptive scheduling policy, one should not assume that the created threads will run immediately and that the threads will run in the order of creation. This is why we should not assume whether the output is "Ping - Pong - ..." or "Pong - Ping - ..."

VII. Synchronization Primitives

In theory, all popular synchronization primitives (*i.e.*, semaphores, message queues and monitors) are equivalent to each other, and mutex locks can be used for implementing semaphores [11]. Because our system has an education slant, we choose to implement and build all other primitives on top of mutex locks and semaphores. This is certainly not the most efficient method, but it shows students how the textbook theory works.

A. Mutex Locks

Each mutex lock has an owner (*i.e.*, the identifier of a thread), and a waiting queue. There are three functions, MUTEX_INIT() for initialization, and MUTEX_LOCK() and MUTEX_UNLOCK() for locking and unlocking a mutex lock. Note that a thread cannot recursively acquire the same lock without first unlocking it.

```
int  MUTEX_LOCK(LOCK_t lock)
{
     if (lock has an owner) {  /* busy */
          if (the owner is this thread)
               return OWNER_ERROR;
          if (setjmp(&(Running->Context)) == 0) {
               move this thread to the waiting Q;
               THREAD_SCHEDULER();
          }
     }
     else
          set the owner to this thread;
}

int  MUTEX_UNLOCK(LOCK_t lock)
{
     if (not the owner)
          return  NOT_OWNER;
     set the owner to NOBODY;
     if (at least one thread waiting) {
          move one thread to the ready Q;
          set the lock owner to this thread;
     }
}
```

MUTEX_LOCK() receives a lock. If the lock is owned by some thread, the calling thread is moved to the waiting queue of the lock. Otherwise, the calling thread becomes the owner. MUTEX_UNLOCK() also receives a lock. If the calling thread is not the owner, the unlock request is rejected. Otherwise, the owner is set to NOBODY. Finally, if there is any thread in the waiting queue of this lock, one of them is released. The use of ownership is important because the lock can be protected from being unlocked accidently or intentionally by non-owners and because it is required in implementing condition variables.

B. Semaphores and Other Primitives

Although mutex locks are *binary* semaphores and can be used to implement general counting semaphores, this kernel

implements semaphores by replacing the lock ownership with a counter [11]. Once semaphores are ready, one can implement message queues, condition variables, monitors, reader-writer locks, and barriers. Most textbooks discuss monitor implementations and the reader priority version of the reader-writer lock [11]. The writer priority lock can be implemented using semaphores and is only a little more complicated ([2] and [3]). Barriers can easily be implemented using semaphores or condition variables.

A barrier has a £xed *capacity*, say n, and is associated with two functions: `BARRIER_INIT()` for initialization and `BARRIER_WAIT()` for waiting on a barrier. When a thread executes a `BARRIER_WAIT()`, if the number of threads waiting on that barrier is less than $n-1$, the calling thread is blocked on the barrier; otherwise, all waiting threads, including the calling thread, are released. This is a very useful primitive in multithreaded and parallel programming. The following is a simpli£ed implementation using semaphores:

```
int  BARRIER_WAIT(BARRIER_t  barrier)
{
     int  i;

     SEMAPHORE_WAIT(Enter_Barrier);
     Counter++;
     if (Count == maximum count) {
          for (i = 1; i < maximum count; i++)
               SEMAPHORE_SIGNAL(Exit_Barrier);
          Counter = 0;
          SEMAPHORE_SIGNAL(Enter_Barrier);
     ]
     else {
          SEMAPHORE_SIGNAL(Enter_Barrier);
          SEMAPHORE_WAIT(Exit_Barrier);
     }
}
```

A barrier has a counter `Counter` whose initial value is zero, and two semaphores `Enter_Barrier` and `Exit_Barrier` with initial values 1 and 0. Semaphore `Enter_Barrier` implements a critical section so that an entering thread can work on the internal data exclusively. If the counter is not full, the calling thread and other threads who called `BARRIER_WAIT()` earlier are blocked on semaphore `Exit_Barrier`. If the counter is full, the calling thread releases all waiting threads on semaphore `Exit_Barrier` and resets the counter.

VIII. Previous Experience

In the past three years, we used Sun's lightweight process library and then switched to Solaris threads in an introduction to operating systems course for juniors (sophomore 23.5%, junior 60.3%, senior 10.3% and other 5.9%). In a ten-week quarter, students learn all the basics of MTP using Solaris threads, which include thread creation and join, semaphores, condition variables, simulating monitors, mailboxes, signal handling, and coroutines, with one programming assignment for each topic. Then, in a two-week mini-project, students are asked to £ll in some important components (*e.g.*, stack space management, and scheduling policies such as priority and multilevel queues) of a user-level kernel and expand it with other synchronization primitives (*i.e.*, semaphores, barriers and mailboxes). Details can be found in [9]. A version of Kofoed's work [7] has been used quite successfully until we tried to port it to different platforms. The system is not completely portable. To overcome this problem and other dif£culties, we proposed to address all of these issues once for all under the support of an NSF grant [10]. The design discussed in previous sections will ful£ll our course need and £t well into our NSF project.

Replacing the original kernel with this new one will not introduce problems to the programming assignments and the two-week mini-project for the following reasons. **First**, the low-level functions mimic Solaris thread functions closely. Except for `THREAD_SYS_INIT()`, all the other six functions have their Solaris thread counterparts. In fact, `THREAD_SYS_INIT()` is unnecessary. It is included because the creation of a coroutine structure and initial stack space management can be separated from the other parts so that students can easily identify each important task. **Second**, the original Kofoed version is very tricky in stack frame allocation, although a complete understanding of its inner working is not necessary. The new system will simplify this step greatly. **Third**, in the previous version, stack space is allocated recursively and use a £rst-£t scheme to manage unused space. This is the major portability problem because it fails under a number of compilers and systems. With the new kernel, the solution to portability is more complex but would work on most platforms with minimal effort. Moreover, this kernel is a more structured version than that of Kofoed by including explicit thread yield, suspend, and continue function calls. **Fourth**, this new kernel is tightly integrated into our system. As a result, instructors can skip the low-level details and only use class wrappers. This would have an obvious bene£t as the visual system can be used to help cope with the paradigm shift smoothly. Or, after covering all the higher level constructs, instructors can dig into the low-level kernel and disclose the implementation details. This is what we have been doing for three years.

IX. Conclusions

We have brie¤y presented the design and critical elements of our user-level kernel that supports MTP. User level MTP has a clear advantage in that context switching costs less and is easy to implement, although all threads will be suspended if the containing process suspends because the underlying operating system only recognizes the containing process rather

than treating all threads as independent and schedulable entities. This kernel is only a small part of our overall design as shown in Figure 1. However, due to its simplicity, it can be used separately to show the details of context switching, scheduling, and the implementation of various synchronization primitives. Whether it is used to serve as the base of our class wrappers and visual system or used as an individual component, it will provide instructors and students with a simple and well-designed working example of a minimal MTP kernel.

We are working on extensions to our kernel, class wrappers, and visual system. We expect to support multiprocess programming, Unix System V type interprocess communications (*i.e.*, shared memory, semaphores and message queues), and sockets under a single uniform class-based interface. Extensions such as network and distributed programming and MPI programming are underway. The £nal product and its user manual will be free to the public. A base system that only supports threads, its class wrappers, and visual system will be released in the near future at the following URL:

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

Related course online notes and programming assignments and mini-project are available at

`http://www.csl.mtu.edu/cs270/www/Home.html`

ACKNOWLEDGMENTS

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

REFERENCES

[1] A. J. M. Beddow, Multi-Threaded C Functions, *The C Users Journal*, Vol. 9 (1991), No. 1 (January), pp. 57–60.

[2] R. Conradi, Some Comments on Concurrent Readers and Writers, *Acta Informatica*, Vol. 8 (1977), pp. 335–340.

[3] P. J. Courtois, F. Heymans, and D. J. Parnas, Concurrent Control with "Readers" and "Writers", *Communications of the ACM*, Vol. 14 (1971), No. 10, pp. 667–668.

[4] S. Crane, The REX Lightweight Process Library, Department of Computing, Imperial College of Science, Technology and Medicine, London, 1993.

[5] J. English, Multithreading in C++, *ACM SIGPLAN Notices*, Vol. 30 (1995), No. 4, pp. 21–28.

[6] J. Finger, Lightweight Tasks in C, *Dr. Dobb's Journal*, May, 1995, pp. 48–50, p. 102.

[7] S. Kofoed, Portable Multitasking in C++, *Dr. Dobb's Journal*, November, 1995, pp. 70–78.

[8] MIX Software, *Using Multi-C: A Portable Multithreaded C Programming Library*, Prentice Hall, 1994.

[9] C.-K. Shene, Multithreaded Programming in an Introduction to Operating Systems Course, *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, February 25 – March 1, 1998, Atlanta, Georgia, pp. 242–246.

[10] 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.

[11] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, £fth edition, Addison-Wesley, 1998.