A THREAD IMPLEMENTATION PROJECT SUPPORTING AN OPERATING SYSTEMS COURSE

Tom Bennet
Department of Computer Science
Mississippi College
Clinton, MS 39058
601-924-6622
bennet@mc.edu

ABSTRACT

This paper describes an assignment for an operating systems course in which students implement a simple threading support library. This project provides practical experience with scheduling and dispatch using a single, discrete project, without the complexity of writing or modifying a full OS scheduler. Students can solve this problem entirely in C or C++, so they do not need assembler language. We describe a version of such an assignment used at Mississippi College, and some possible variations.

INTRODUCTION

A course in operating systems is a standard part of the Computer Science curriculum [8]. A major purpose of any OS, and hence a major topic of any OS course, is the management and scheduling of processes [9]. While scheduler design will certainly be discussed in lecture, a project can be very helpful to reinforce the concepts. Some courses concentrate on the actual implementation of a small OS [11], which will include thorough study of its scheduler. A student project OS must run on a dedicated machine, real or virtual, with consequent complexities of running and debugging. It must either be part of a project to write the entire OS, or students will need to understand the balance well enough to operate on a vital component without killing the patient.

Not all instructors wish or need to pursue so lengthy and involved an exercise. This paper describes a less-extensive alternative: creation of a user-level threading library. Such a library is used by a program running in a traditional process. The operating system provides a single thread for this process, which the threading library multiplexes to support a threaded application. It exports primitives for thread creation, and for threads to exit or to terminate each other. Usually, there are synchronization primitives which allow threads to wait on one other and to communicate. An example of a full-scale threading interface, much more complex than our exercise, is the standard pthreads [1].

While threads are less complex than processes, a thread library has many duties in common with an OS: threads must be created, scheduled, dispatched and synchronized. Yet the library is used and tested in an ordinary process. Our simple thread library can be completed in a matter of a few weeks, and can be used to write non-trivial threaded programs.

SUPPORT LIBRARY

Students are provided with a small support library to manipulate CPU registers [2]. This allows them to solve the threading problem without using assembler language. The present version of the register library is written for the PC architecture using the NASM assembler [6]. This version runs on Linux. A port to Windows should be straightforward; porting to other hardware architectures may be possible as well. The register library is designed to be minimal and general. It exports the type name `regbuf_t` and just four operations.

Registers are saved and restored with the `regsave` and `regrest` operations, called like this:

```
int regsave(regbuf_t buffer);
int regrest(regbuf_t buffer, int retval);
```

The `regsave` operation copies the program-visible CPU registers into a memory region provided by the programmer, and `regrest` copies them back. They are very similar to the `setjmp` and `longjmp` operations which are part of standard C [4], and could probably be used just as well. Since the program counter is among the registers saved and restored, the `regrest` operation does not return; instead, the `regsave` call which saved the state being restored returns again. Generally, `regsave` returns once after it is called, then again each time `regrest` is called with the same save buffer.

When `regsave` does return, it is often necessary to distinguish between its initial return and a return caused by `regrest`. To accomplish this, `regsave` returns 0 in the first case, and otherwise in the second. The second parameter to `regrest` becomes the return value of the revitalized `regsave`. Most `regsave` calls will be either in an if test, or immediately followed by an if, to provide different behavior in those cases.

The `restack` operation relocates the stack into alternate space specified by the caller. It takes four arguments, like this:

```
int restack(void *newstack, int size, int nframe, int nparm);
```

This call accepts a block of space from the caller, and makes this the stack area of the current thread. It does this by copying a portion of the caller's original stack into the specified area, then directing the CPU stack pointer there. When `restack` returns, the calling thread is using the substitute stack. `Restack` copies `nframe` frames of the original stack, plus `nparm` additional integer-sized chunks. The latter accounts for parameters to the topmost copied frame.

Since `restack` does not copy the entire call stack, it is quite possible to return from the topmost copied frame, which is invalid. To deal with this, `restack` changes the return address of that last frame to cause a transfer to an error handler. The default error handler prints a message and terminates the program; `setcleanup` allows the client to specify an alternate cleanup function, like so:

```
cleanvec_t setcleanup(cleanvec_t cleanfunc);
```

This call sets the new handler, and returns the old one.

BUILDING ON THE FOUNDATION

Threading libraries with various interfaces may be built using the above register primitives. Here, we present the interface used for a assignment in the Operating Systems course at Mississippi College during Fall 2006. As will be apparent, the interface was modeled after some of the Unix process creation primitives [7].

Any reasonable solution must create a record for each thread, and manage collections of them. In the following, these are called Thread Control Blocks, or TCBs. Students must create TCBs for new threads, and manage a ready list for dispatching. Since one thread can wait for another to terminate, students must also manage a collection of blocked threads, and activate them when their awaited event occurs. We schedule threads using simple, non-preemptive first-come-first served, but other algorithms could be used just as well.

The remainder of this section summarizes our thread interface, sketches a possible implementation, and describes some alternatives. The function bodies here look somewhat C-like, but are very much pseudocode.

Initialization

The thread library must be initialized by a call to `th_init`. The client must do this before using any of the other facilities. Its primary job is to record the pre-existing thread so it may be scheduled, something like this:

```
th_init():
        ... initialize data structures ...
        current := new TCB;
```

A traditional process, as created by the operating system, has a single thread. The `th_init` function creates a TCB to represent it, and places it into the global variable `current`, which always points to the currently active thread. Some implementations will also add it to a collection, either a ready list or a list of all TCBs.

Thread Creation

Threads are created with `th_fork`. Like the Unix process-creation operation fork, `th_fork` returns twice, in the parent thread which called it, and again in the child thread which it creates. `Th_fork` returns the identifier of the new thread in the first case, and the symbol `ARE_CHILD` in the second. The function may be implemented along these lines:

```
th_fork():
        t := new TCB;
        ready.insert(t);
        if(regsave(tmpbuf) > 0) return t->identifier;
        restack(t->space,...);
        if(regsave(t->registers) > 0) return ARE_CHILD;
        regrest(tmpbuf,1);
```

The procedure creates a new TCB and inserts it into the ready list, then saves the registers for later. It fills in the new TCB by switching to stack space within the TCB, and saving the registers in the TCB also. The final `regrest` wraps up as follows:

- Restoring the registers restores the stack pointer, switching the thread back to original stack.
- The restore also transfers control to the first `regsave`, which returns 1.
- The if is satisfied, so `th_fork` finishes and returns the new thread identifier.

Notice how control jumps around. Each `regsave` returns twice, the second return causing immediate return from `th_fork`. For the first `regsave`, this return follows the concluding `regrest`, completing the initial `th_fork` call. The second `regrest` makes its second return later. When the newly-created thread is first dispatched, its TCB is removed from the ready list, and a `regrest` is executed on its register area, which brings control back here for the second return of `th_fork`.

Thread Dispatch

Since our thread scheduling is non-preemptive, threads must volunteer to surrender the CPU. They do this by calling `th_yield()`, which chooses another thread and dispatches it. An implementation might look like this:

```
th_yield():
        if(regsave(current->registers) > 0) return;
        ready.insert(current);
        current := ready.next();
        regrest(current->registers,1);
```

This suspends the current thread by saving its registers in its TCB, then placing it onto the ready list.

The function then chooses another thread to run, places it in current, and dispatches it by restoring its registers to the CPU. Notice that the `regrest` may return at the `regsave` in this function, or in another, depending on its history. If this is the first time the new thread has been run, the return will be from the second `regsave` in `th_fork`, which will then return to its newly-created thread.

The `ready.next()` call in the above encapsulates the selection of which ready thread to dispatch. Our classroom project used simple first-come, first-served scheduling, so `ready.next()` was simply removing the head of a queue. Any desired policy could be implemented.

Termination and Synchronization

A thread may terminate itself with `th_exit(n)`, or it may terminate another with `th_kill(id, n)`. In either case, the terminated thread is given the exit value specified by `n`, and that value can be recovered by `th_wait(id)`. The `th_wait(id)` call suspends the calling thread until the thread denoted by `id` terminates, then returns its exit value.

These synchronization operations require the students to manage a list of blocked (waiting) threads, and transfer exit values. A thread may terminate before or after another waits for it, so the code must handle both cases. If termination occurs first, the exit value must be retained until a wait occurs. Otherwise, the waiting thread must be suspended, and the exit value transferred when the awaited thread expires.

The Required Solution

Students were required to submit source code for a complete library in either plain C or C++, at the student's option. So each solution must contain a header file to define the interface, and an implementation file to provide the function bodies. Regardless of implementation language, the interface is plain-C style, with each operation a top-level function, rather than a class method. The functions are those described above: `th_init`, `th_fork`, `th_yield`, `th_exit`, `th_kill` and `th_wait`, along with utilities `th_me()` and `th_stksize(s)`. The former returns the identifier of the thread which calls it, and the later sets the size of thread stacks.

The interface file was also required to export a type name, `th_id_t`, which is the type of a thread identifier. It is the return type of `th_fork`, and the type of the `id` parameter sent to `th_wait` and `th_kill`. Accompanying these must be the symbolic constants `ARE_CHILD` and `DEF_STACK_SIZE`. The first is of type `th_id_t,` and is essentially a null thread identifier. The later is an integer giving the default size used for stacks when `th_stksize` has not set a value. The full specification is available from [2].

Students have access to a shared Linux system where the register utility is installed as a shared library. The submitted thread library must compile on this system. Any correctly-formed threaded client must also compile. Then the whole thing must successfully link, along with the register library, and run correctly.

Test Drivers

The students were given several example clients which show ways the threading interface can be used. Most of these are just test drivers, designed to exercise the library rather than attempt any meaningful calculation. The exception was a program which finds the shortest path in a graph. It reads a weighted graph followed by several node pairs, and finds the shortest acyclic path between each pair. The program executes a breadth-first search. When any node has multiple un-visited neighbors, the search thread visits one neighbor itself, and creates additional threads to visit each of the others. Copies of these programs are posted on line [2].

Alternative Interfaces

Since this interface is modeled after Unix process creation, it differs somewhat from most standard threading facilities. But such interfaces could be imitated instead. For instance, a thread creation primitive modeled after the pthreads interface [1] might use a creation primitive like the one below. It takes a function, along with an argument to send it, and runs these in the newly-created thread. It returns the id of the new thread to the caller through a parameter.

```
th_create(id, func, arg):
     t := new TCB;
     ready.insert(t);
     id = t->identifier;
     if(regsave(current->registers) > 0) return;

     restack(t->space,...);
     if(regsave(t->registers) > 0) {
          func(arg);
          delete current; // t == current

          current := ready.next();
          regrest(current->registers,1);
     }
     regrest(current->registers, 1);
```

The implementation of `th_create` is similar to `th_fork`. The main difference is that we must call the thread function in the new thread, then dispatch another thread when it finishes.

A Java-like interface [3] could also be created. It would consist of a class Thread containing an abstract method `run()` and a method `start()`. The client provides the code run in the thread by implementing `run()`, and starts the thread running it by calling `start()`. The `start` method could be implemented much like the `th_create` above, but would call the `run` method rather than a functional parameter. Any additional thread operations would be methods of the class. Structures such as the ready queue, which apply to all threads, would be static members.

So several threading interfaces might be built atop the register primitives. The implementations are similar, and require the creation an management of TCBs, along with blocking, unblocking and dispatching threads.

(JUST A LITTLE) EXPERIENCE

The author assigned the Unix-like threading problem described above in Fall 2006. (A somewhat different version was used the previous year.) Students had a month to solve it, including a one-week extension. Most students managed to create and dispatch threads, though some solutions worked only partially. Six used plain C and eleven used C++.

Most solutions contained three thread lists chosen from four possibilities: all threads, ready threads, terminated threads or waiting threads. Some used just two of those, and one had separate lists for exited and killed threads. The C++ solutions used STL container classes from the C++ libraries [5] for these lists, variously the list, queue, and map template classes. The C solutions generally used arrays of TCBs or of TCB pointers. Several declared fields to link TCBs into lists, but only one avoided arrays entirely. The use of efficient data structures was not emphasized, and the C solutions generally reflected this, frequently using linear search.

Because threads wait for others to terminate, the issues of termination and synchronization are mixed. From conversation with students, this seemed to be a particular problem. In the future, it may be better to separate these issues by eliminating the existing thread wait call, and introducing a mailbox object [10]. Threads would use a mailbox to transmit a single integer value. Executing a receive on an empty mailbox suspends the caller until data is available. Implementing mailboxes would still provide experience managing blocked threads, separately from termination, while actually providing a slightly more flexible synchronization facility.

CONCLUSION

This paper outlines the definition and solution of an exercise to create a threading library. This exercise supports the standard Operating Systems course as simpler alternative to writing or modifying a full OS. Students get experience in creating, managing, dispatching and synchronizing threads. They must manipulate the threads as an OS does, from the outside, without being part of any of them. Yet this is a single project which can be solved in a few weeks. It produces usable code which can be compiled, linked, tested and used on a conventional OS.

[1] Barney, B., POSIX Threads Programming, http://www.llnl.gov/computing/tutorials/pthreads/, last modified Aug 16, 2006.

[2] Bennet, T, The Tswitch Thread Support Library, http://sandbox.mc.edu/~bennet/thproj.html, last modified Dec 26, 2006.

[3] Class Thread, http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html, copyright 2004.

[4] Harbison, S., Steele, G., *C: A Reference Manual*, Upper Saddle River, NJ: Prentice Hall, 2002.

[5] Josuttis, N. M., *The C++ Standard Library*, Reading, MS: Addison-Wesley, 1999.

[6] NASM Home page, http://nasm.sourceforge.net/doc/html/nasmdoc0.html, retrieved Dec 2, 2006.

[7] Quarterman, J., Silberschatz, A., Peterson, J.,. 4.2BSD and 4.3BSD as examples of the UNIX system. *ACM Comput. Surv.* 17, (4), 379-418. 1985.

[8] Sharma, O., Enhancing operating system course using a comprehensive project: decades of experience outlined, *Journal of Computing Sciences in Colleges*, 22, (3), 206 - 213, 2007.

[9] Silberschatz, A., Galvin, P., Gagne, G., *Operating System Concepts*, Hoboken, NJ: John Wiley & Sons, 2005.

[10] Stallings, W., *Operating Systems: Internals and Design Principles*, Upper Saddle River, NJ: Prentice Hall, 2005.

[11] Tanenbaum, A., Woodhull, A., *Operating Systems: Design and Implementation*, Upper Saddle River, NJ: Prentice Hall, 2006.