Homework 4: Thread Implementation

Assigned: September 29, 2009
Due: October 9, 2009 (by midnight)

1 Overview

Your startup is in trouble! One of your star programmers has just left for greener pastures while in the middle of implementing My Threads (MT), a user-level threads package that is a key part of your product. Since all of your other developers are committed to other parts of the project, it's up to you (two) to finish the threads implementation.

There are a number of reasons that you are being asked to do this assignment. Among them are:

1. The structure of a user-level threads package is closely analogous to the structure of a simple operating system. Working on this code is a lot easier than working on a real OS and gives you much the same kind of experience.
2. As a programmer you commonly are forced, for one reason or another, to pick up and modify a big piece of code that someone else wrote. Although at around 1000 lines MT is not large, some of it is fairly complex.
3. This is the natural counterpart to Homework 3, where you work on a threaded program. By the time you are finished with this project, you will be able to run threaded programs on your own thread library, rather than relying on the Linux threads implementation.

2 Practicalities

For this project you will be working with a partner of your choice. You can find files that are referred to in this document here (in the CADE lab filesystem):

```
~cs5460/hw4
```

Please copy these files to one of your working directories.

This project will not require you to write a large amount of code. However, the code that you do write will be somewhat subtle and difficult to debug. It is therefore important that you:

- Read and understand this entire document before starting — try not to send questions to the class mailing list that are answered here.
- Develop incrementally: get something working, test it thoroughly, and then move on to the next feature.

- Think before you code. As with other projects in this class, once things start going wrong it will be difficult to figure out how to fix it. If you get stuck and things aren't working, avoid the temptation to start making random changes to the code until it works — this method almost always leads you farther from the right solution instead of closer. Rather, revert to the last version of your code where everything worked.
- Avoid making changes to parts of `mt.c` that you do not need to change. Part of the assignment is to understand enough of the structure of the code that you can make changes to it without breaking things.
- Avoid mixing your thread calls with any calls to native Linux threads or pthreads functionality. If you do this, the results will be confusing at best.
- To be on the safe side, protect every call into Linux or into a library by disabling SIGALRM for the duration of the call (see the `enter_MT_kernel()` function). Otherwise you may get highly unpredictable results.
- Make use, when appropriate, of the debugging information provided by the `DBGPrint()` calls scattered throughout the MT code. These calls are intended to get you started — you should add your own calls in order to understand the behavior of your code. Make sure that debugging printouts are turned **off** before you handin your code. Turn these printouts on and off by defining, or not, the preprocessor symbol `DEBUG_PRINT` (i.e., by supplying or not `-DDEBUG_PRINT` to `gcc`).

Finally, MT has been designed to be portable to versions of UNIX that provide the `set/getcontext()` calls. It works on recent versions of Linux and Mac OS X (if you adjust `Makefile`). However, if you choose to do your work on a platform other than Linux, remember that it is your responsibility to ensure that everything compiles and runs correctly on the CADE lab Linux machines.

3 The MT API

This section describes the application programming interface (API) that MT must provide. The functions that are not implemented yet are indicated — you must implement them precisely as they are specified because as part of the grading process we (Matthew and the TAs) will link our own test programs against your library.

int `MT_init` (void); (implemented)

This function initializes the threads library, and it must be called before any of the other functions are called. In effect, it turns a process into the first of possibly many threads. Return value is 0 on success and -1 on error.

int `MT_create` (thrd_main_t func, int arg);   (implemented)

This function creates a new thread running the specified function and passes it the specified argument. The return value is either the TID (thread identifier) of the new thread, a positive integer, or -1, indicating that an error occurred and the thread could not be created.

int `MT_join` (int tid, int *result);   (implemented)

The calling thread waits for the thread identified by the first argument to terminate. If the second argument is non-NULL the exit code of the thread is placed in the location that it points to. It is permissible for multiple threads to be waiting for a given thread to terminate. The return value is 0 on success and -1 on error (if the specified thread does not exist).

It is permissible to join a thread that has already terminated. However, after the first call to `MT_join` the terminated thread's resources will be deallocated and the thread will cease to exist.

void `MT_exit` (int status);   (implemented)

The call terminates the calling thread. It is equivalent to returning from the thread main function.

int `MT_gettid` (void);   (implemented)

Returns the TID of the calling thread.

int `MT_usleep` (int us);   (NOT implemented)

Puts the calling thread to sleep for the specified number of microseconds. This function must block only the calling thread — not the entire process. It should return 0 on success and -1 if given erroneous input (a negative argument).

void `MT_sem_init` (sema_t *sem, int init_count);   (NOT implemented)

Initialize a semaphore pointed to by the first argument with a count specified by the second argument. This function does no argument checking: it may assume that it is passed a valid pointer and all initial count values are acceptable.

void `MT_sem_wait` (sema_t *sem);   (NOT implemented)

The semaphore wait operation. It performs no argument checking.

void `MT_sem_signal` (sema_t *sem);   (NOT implemented)

The semaphore signal operation. It performs no argument checking.

int `MT_set_share` (int share); (NOT implemented)

Sets the scheduling share of the calling thread. Threads must be scheduled preemptively. Assuming no contention from other processes, each thread should receive a fraction of the CPU equal to its share divided by the total share over all threads. It returns 0 on success and -1 if passed an illegal argument (outside the range 1–10000).

Termination of a process using MT

A process using MT can terminate abnormally in the usual ways (core dump, etc.). There are only a few normal termination mechanisms:

- If `main()` returns then the entire process, with all threads, is immediately terminated.
- If any thread calls `exit()` then the process exits.
- If all threads are in the TERMINATED state, then the process exits.

On the other hand, the process does not exit just because there are no runnable threads. When there are no runnable threads the MT scheduler blocks the process until it has something to do.

When a process using MT receives SIGINT (which is delivered to a UNIX process when control-C is typed at the console) it prints a list of all existing threads and their states, and then exits.

Fatal vs. non-fatal errors

Usually, when an MT function encounters trouble it will return an error code. However, sometimes this just does not make sense. For example, if there is a problem with a context switch routine then there is no sensible way to recover and the MT function will just terminate the process.

4 The Assignment

Your assignment is straightforward: fill in the missing MT functions and write a few test programs to demonstrate that your functions work. The section describes the requirements that your implementation should fulfill and gives some implementation hints.

4.1 Timers

Timers are an operating system abstraction used to put a process or thread to sleep for a specified time. You cannot use native Linux timers to put an MT thread to sleep because Linux doesn't know anything about MT threads: it will just put the whole process to sleep.

To implement timers you should fill in the body of the function `MT_usleep()` and write some support code elsewhere in `mt.c`. You will need to:

- Figure out what time it is (and what time to wake up a thread) using the `gettimeofday()` system call.

- Add a "wakeup time" field to the thread control block struct.
- Add a new queue to MT that represents the list of threads currently blocked on timers.
- Add code to the `timer_handler` function that periodically looks for threads that need to be awakened.

An example program `timer_test.c` is provided that you can use to see if your timers support the basic functionality. However, if this test appears to be working, don't assume that you are finished testing your implementation — you should test more than the ten threads that it creates and you should test longer-range timers than its 30-second maximum delay.

Here are some requirements for your implementation:

- You should not store sleeping threads in a fixed-size data structure. Your timers, like the rest of MT, should be designed to support a number of threads bounded only by available memory.
- You cannot wake a thread up at precisely the time that it requests since the highest frequency of timer signals supported by Linux and Linux is one per 10,000 microseconds. Your implementation should never wake a thread up early. Rather, wake it up as soon as possible **after** its timer expires. It should be possible to essentially always wake a thread up within a hundredth of a second of its requested wakeup time.
- If you store times in units of microseconds in a 32-bit `int`, it might overflow. You should consider storing times using the `long long int` datatype, which is 64 bits.
- Don't use `floats` or `doubles` to support timers (or anywhere else in the MT code).

Expected lines of code: Completing this part of the assignment should add less than 20 lines of code to `MT_usleep()`, less than 20 lines to `timer_handler()`, and a few more lines elsewhere. If you write more code than this don't panic, but look carefully at what you are doing to make sure that it's not more complicated than it needs to be.

4.2 Semaphores

You should implement counting semaphores; you can find the algorithm in your textbook or in the lecture slides. To test your semaphore implementation, you can use the `philo-sema-mt.c` program that is provided, which also compiles with Linux pthreads. Run the program for at least 5 or 10 minutes to make sure that it does not crash or deadlock. (Compare the performance of the two implementations. Which performs better? What do you think accounts for the speed difference?)

Expected lines of code: Less than 20 lines per semaphore function. You shouldn't have to change any code outside of the semaphore functions.

4.3 A Better Scheduler

Unlike most user-level threads packages, MT has a preemptive scheduler. Threads are given time slices according to a round-robin discipline: each time a timer signal arrives, MT moves the currently running thread to the end of the run queue and dispatches the thread at the head of the run queue. In this part of the assignment you will implement a more sophisticated **proportional share** algorithm that permits prioritized allocation of CPU time.

The high-level specification of a proportional share scheduler is that over a period of time somewhat longer than the scheduling quantum, each thread receives a fraction of the CPU equal to its share divided by the sum of all threads' shares. So if there are three threads with shares 10, 10, and 20, the threads receive respectively 25%, 25%, and 50% of the CPU time allocated to the MT process, on average.

Here is the low-level, implementation-oriented specification:

- Each thread is characterized by a **share** and a **virtual clock**. The share of each thread is initialized to 10. The virtual clock of the initial thread is initialized to 0.
- Let **minclock** be the minimum virtual clock value among all threads that are in the READY and RUNNING states.
- When a thread unblocks, or when a thread (other than the initial thread) is created, its virtual clock value is set to the maximum of its previous virtual clock value (or 0 for a newly created thread) and minclock.
- To **update** the virtual clock of a thread, increment its virtual clock by a value that is computed by taking the amount of time it has spent in the RUNNING state since its virtual clock was last updated and dividing it by the thread's share.
- At every timer signal the virtual clock of the running thread (if any) is updated and the READY or RUNNING thread with the smallest virtual clock value is dispatched.
- When a thread blocks its virtual clock is updated and then the READY thread with the smallest virtual clock is dispatched.

The intuition behind this algorithm is that all CPU time allocated to a thread is "charged" to its virtual clock in inverse proportion to its share. In other words, threads with bigger shares are charged less. You must be careful to avoid security holes in your code that permit a thread to run without its virtual clock being charged, as threads could potentially exploit this to receive more than their share of CPU time. The reason that threads potentially have their virtual clocks increased when they wake up after being blocked is to prevent threads from building up too much credit while they are sleeping. For example, we do not want a thread to be created, spend an entire day sleeping, and then hog all CPU time because its virtual clock is much lower than other threads' clocks.

When you update the virtual clock of a thread you should not use the wall-clock timer `gettimeofday()` because your MT process could easily be sharing the machine with other processes (and hence not receiving one second of CPU time for each second of real time). Rather, use the `times()` call. If you add together the `tms_utime` and `tms_stime` fields of the struct returned by the `times()` call, you will get a reasonably accurate count of the total amount of CPU time allocated to the calling process.

You can use the program `sched_test.c` to see how your program is doing. Over a 10-second run the actual CPU allocation to each thread should always be within a percent of the predicted allocation, and usually within a few tenths of a percent. For example:

```
0: share 27, work 193230981 (expected 6.5%, got 6.6%)
1: share 44, work 315819861 (expected 10.6%, got 10.7%)
2: share 4, work 29913500 (expected 1.0%, got 1.0%)
3: share 24, work 172227684 (expected 5.8%, got 5.8%)
4: share 54, work 361174552 (expected 13.0%, got 12.3%)
5: share 70, work 504407118 (expected 16.9%, got 17.1%)
6: share 51, work 363835342 (expected 12.3%, got 12.4%)
7: share 38, work 272831098 (expected 9.2%, got 9.3%)
8: share 92, work 651167922 (expected 22.2%, got 22.1%)
9: share 11, work 80611075 (expected 2.7%, got 2.7%)

*** all threads have exited: normal termination ***
```

You should write extra test cases to ensure that your scheduler behaves correctly in the presence of blocking threads.

Expected lines of code: Fewer than 200 lines, but somewhat scattered around `mt.c`.

4.4 Extra Credit: Non-blocking socket calls

You can earn a small amount of extra credit (relative to the total weight of this project) by implementing non-blocking socket calls. Please do not start working on this until you have everything else working — overall you will do much better by turning in correct code without any extra credit, than by turning in flaky code and extra credit code.

Also, if you choose to earn some extra credit you are on your own as far as finding information goes. Use the web, the man pages, etc.

If an MT thread makes a blocking UNIX system call, for example reading from a socket, all threads in the process will be blocked. Obviously this is undesirable.

You can earn extra credit by implementing thread-safe versions of the `read()` and `write()` system calls. Here is how they should work:

1. A thread calls `safe_read (socket, buffer, size)`.
2. MT uses the `select()` or `poll()` system call to check if the read can be satisfied immediately. If so, the read is performed. If not, the thread is blocked, permitting MT to dispatch other threads.
3. MT periodically (i.e., in the timer interrupt) polls for completion of I/O for any blocked threads. When an I/O completes MT unblocks the thread and permits it to continue.

The correctness criteria for this approach are:

1. The MT process as a whole **never** blocks on a socket call — only individual threads may be blocked.
2. Blocking (or lack of blocking) is completely transparent to threads — they should not be able to tell which happened.

The catch: You must write a program to test and demonstrate your non-blocking socket calls. We can negotiate what this application is, but I suggest a multi-threaded N-way chat application.

If you write a chat application, it should take a command line listing the machines to connect to. So on machine A you would type `chat B C D`, on machine B you would type `chat A C D`, etc. Once connections are established between all parties, the application should divide the screen into N regions, each of which is managed by a single thread. The "curses" library makes it easy to do this kind of screen manipulation. Each region of the screen displays what is typed on one of the participating machines. So, each person using the chat application sees what she types in one window, and what each other participant types in a different window. Each thread, then, spends most of its time blocked on socket reads, waiting for input from the appropriate machine. The point of structuring this application using one thread per connection is that the code for each connection can be quite simple. This application can be implemented in about 400 lines of C.

Another reasonable demonstration application would be a simple multi-threaded web server. Here you would have one thread listening for new connections on a well-known port (8080 or whatever) and you would allocate a new thread to service each incoming connection. This architecture is good because: creating a thread is a lot cheaper than creating a process (like Apache does); it creates a convenient programming model — the code for the thread that services connections is pretty easy; it prevents a large file being served over a slow connection from delaying service of other connections; and, it permits a form of differentiated service where important connections are given a larger scheduling share than unimportant connections. If you go this route you will also have to write non-blocking versions of the `listen()` and `accept()` system calls.

5 Logistics and Grading

Grading for this project is primarily on correctness: your improved version of MT should implement the specified functionality and should not crash. However, we may award partial credit if you have taken a good approach but messed up some details. Good comments in your code are crucial for helping us understand your solution — partial credit will not be awarded when we cannot understand what you have done.

What to handin:

1. A file called `README` that provides the names and userids of all group members.
2. All source files required to build your improved version of MT.
3. A makefile that builds all programs you handin. This is **required** — we will not manually invoke the compiler while grading your project.
4. If you are trying for some extra credit, also include a file `extracredit.txt` outlining your approach and providing other information that might be useful to the graders.

Your project should include a `mt.h` that we can use for building our own programs (probably unmodified from the provided version), and your makefile should produce a `libmt.a` library that we can link to our own programs.

Although you may do your work wherever you want, your project will be graded on a CADE lab Linux machine, so **you must make sure that everything compiles and runs properly there**. If it doesn't you will not get credit for that part of the assignment.

This is a group assignment — you and your partner will hand in only one set of files and you will both receive the same grade. Either of you can handin the files (but not both, please).

Your handin command line for this project will have the form:

```
  handin cs5460 hw4 file1 file2 ...
```