Project Mthreads

Due: March 2, 2016

Contents

1 Introduction 1

2 Background 1

3 The Assignment 2

  3.1 Overview ................................................................. 3
  3.2 Assumptions ............................................................... 4
  3.3 Topics from Uthreads .................................................. 4
  3.4 The Reaper 2.0 ............................................................ 4

4 Compiling and Testing 5

  4.1 Debugging ................................................................. 5
  4.2 Test Code ................................................................. 5

5 Handing In 6

1 Introduction

In this project you will build on the TA solution for the uthreads assignment to make it multiprocessor-safe. In uthreads, you implemented a threading library with an N-to-1 (N threads multiplexed on 1 “kernel” thread) threading model. In mthreads, you will be implementing a library with an M-to-N (M threads multiplexed on N “kernel” threads) threading model. You can think of pthreads as kernel threads. Having completed this assignment, your threading library will support user-level threads running in parallel on many processors! You can copy the stencil code out of the /course/cs167/asgn/mthreads_stencil directory.

2 Background

M-to-N, also known as Hybrid Threading, is a threading model in which user-level application threads are multiplexed on a pool of kernel-level threads. Such kernel-level threads can be referred to as “virtual processors”, since these threads may or may not actually be running in parallel – N kernel threads may themselves share a single CPU.

In uthreads, each user-level thread had a context that the kernel thread, which was actually the main thread of the program, switched into when the associated thread was scheduled to run. In
mthreads, the story is basically the same only that instead of having just one main thread, we have a number of pthreads, each of which represents a kernel thread in an OS. These pthreads are referred to in the stencil as a lightweight-processes (LWP). LWPs execute one runnable uthread at a time.

M-to-N threading is considerably more complicated to implement than N-to-1 threading because, in addition to masking preemption in the right places, the model must synchronize kernel-thread access to data structures including, but not limited to, the run queue, user-level mutexes/condition variables, and user-level thread structures. In fact, this is most of the work you will be doing.

When implementing this assignment it is also important to keep in mind that a user-level thread may be scheduled on any number of kernel threads and processors during it’s lifetime. This means that in a routine like uthread_switch, we need to ensure that data that is necessary before and after a user thread is scheduled is not optimized away into a CPU register by the compiler.

Finally, you will find that there are several instances (documented in the stencil) where you will have to perform some operation on a user-thread member after the thread in question has switched off the processor. Reasons for doing this are described in the stencil, but you are strongly encouraged to read chapter 5 of the textbook, which discusses various multiprocessor issues like this, and refer to problem 3 on homework 2. Particular egregious errors include two processors executing the same thread, and a currently-executing thread’s stack being freed while some user-level thread is executing on that stack.

3 The Assignment

mthreads will make use of a lot of code that should already be familiar to you from the uthreads assignment. The logic for creating threads, joining and detaching threads, scheduling, and dealing with mutexes and condition variables is the same, only now you will find that the concerned structures have been augmented with a pthread_mutex_t, which should be used in the appropriate contexts to synchronize various LWPs contending for access.

You only need to worry about modifying functions with TODOs. Each of these also contains an call to our NOT_YET_IMPLEMENTED macro. To see a list of all the functions you have yet to modify/implement, run make nyi in the project working directory.

The majority the work in the assignment will go into implementing uthread_switch, which should be significantly different from the one you implemented in uthreads, and lwp_switch, which is where LWPs execute independent of any uthread context, waiting for a runnable uthread to become available. Here’s a list of functions you’ll need to touch while working on this assignment:

```c
// uthread.c
uthread_exit(int status);
uthread_join(uthread_id_t uid, int *return_value);
uthread_detach(uthread_id_t uid);
make_reapable(uthread_t *uth);

// uthread_sched.c
uthread_yield(void);
uthread_block(void);
```
uthread_wake(uthread_t *uth);
uthread_setprio(uthread_id_t id, int prio);
uthread_switch(utqueue_t *q, uthread_t *thr, pthread_mutex_t *m);
lwp_switch(void);
lwp_park(void);
uthread_runq_enqueue(uthread *thr);
uthread_runq_requeue(uthread *thr);

// uthread_mtx.c
uthread_mutex_lock(uthread_mtx_t *mtx);
uthread_mutex_trylock(uthread_mtx_t *mtx);
uthread_mtx_unlock(uthread_mtx_t *mtx);

// uthread_cond.c
uthread_cond_wait(uthread_cond_t *cond, uthread_mtx_t *mtx);
uthread_cond_broadcast(uthread_cond_t *cond);
uthread_cond_signal(uthread_cond_t *cond);

It may help to implement uthread_switch and lwp_switch before proceeding to the other functions even though they require more work because a) these routines interact with each other in a somewhat complicated way and b) they make the majority of the heavy-weight scheduling decisions that involve switching between the context of a uthread and the context of an LWP. The understanding you gain from implementing these two functions should make editing the remaining ones much easier.

3.1 Overview

Many of the functions mentioned above (particularly lwp_switch) have extensive comments in the source code which explain what is expected of you, but to save you some time, we will give you a brief summary of how the system works as a whole.

The first thing that any executable that uses your threads package should call is uthread_start. This should be called exactly once and is responsible for setting up global data structures such as the uthreads array (as it did in uthreads).

The major differences between the setup done by uthread_start and uthread_init (from uthreads) lies in create_first_thr. This function is responsible for setting up the main thread and the reaper thread, and placing them both on the run queue. It also sets up two additional LWPs by calling uthread_start_lwp twice. uthread_start_lwp spawns a pthread which runs lwp_start. It is important to note that, at this point, each of these two new LWPs has its own ut_curthr and curlwp variables. This can be seen from the declarations near the top of uthread.c:

// uthread.c
__thread uthread_t *ut_curthr = 0;
__thread lwp_t *curlwp;

The __thread storage class keyword is an extension to GCC that can be used alone or with the extern or static qualifiers for a global, file-scoped static, or function-scoped static variable to
ensure that each thread is allocated it’s own private copy of that variable. When used with `extern` or `static`, `ut_thread` must appear as the last storage class keyword. Addresses of such variables can be used by other threads, though you shouldn’t need to worry about that for this assignment.

What this means for us is that each LWP stores it’s own context in addition to the context of the `ut_thread` that it is currently running. Thus we can expect that `ut_curthdr` will change frequently throughout the LWP’s lifetime (in `lwp_switch`) as it multiplexes various user-level threads, whereas `curlwp` will remain constant so that various `ut_threads` can switch back into the context of their invoking LWPs if they need to block or yield. This is exactly what the interaction between `ut_thread_switch` and `lwp_switch` facilitates. A `ut_thread` invoking `ut_thread_switch` uses it’s thread-local `curlwp` to jump back into `lwp_switch`, whereas an LWP users the `ut_ctx` member of a runnable `ut_thread` to switch into it’s context and begin running.

By the end of `uthead_start` all 3 LWPs are ready to begin multiplexing `ut_threads`. The remainder of the assignment consists of ensuring that the `ut_thread` API (that you implemented in the previous assignment) is thread-safe and multi-processor safe. The `mthreads` library still supports user-level preemption, however your TAs have provided you with a solution to `ut_threads` that deals with this appropriately in most of the `ut_threads` API you will be working with.

### 3.2 Assumptions

In this assignment there will be multiple threads running at a time, and on multiple processors. Thus, make sure you must guard datastructures appropriately. Be sure to review `uthread.h`, which contains definitions for `uthread_t` and `lwp_t`, as well as other struct definitions for hints as to which data structure accesses need to be synchronized. Importantly, you cannot make the assumption that each `ut_thread` is scheduled on exactly one LWP at a time. This is something that you must ensure with proper synchronization!

The stencil as provided will not work because `ut_threads` aren’t synchronized, so running the stencil as given will not give you any information. You may, however, assume that any functions that do not have TODOs will work properly once you make the required functions (described in the Assignment section) thread-safe and multiprocessor-safe. You will not be held accountable for potential problem with the stencil, but you must modify your API to work with the stencil. If you write alternate implementation that doesn’t work with the given stencil, you will receive no credit.

### 3.3 Topics from Uthreads

Understanding the concepts from `ut_threads` will be very important in `mthreads`. You may want to reread the `ut_threads` assignment, including the sections Swapping Contexts, Time Slicing and Preemption, Dangers of Preemption, and the Reaper.

### 3.4 The Reaper 2.0

Conceptually, the reaper in `mthreads` serves the same purpose it did in `ut_threads`. However, because the reaper can now run in parallel with other `ut_threads`, we must be sure that it doesn’t clean up resources associated with a thread that has just called `ut_thread_exit` before that thread has switched away from it’s context and is no longer calling functions on that context’s stack. For this reason, before the reaper calls `ut_thread_destroy`, it must first attempt to lock the exited thread’s
ut_pmut mutex to be sure that it is no longer using it’s context. It is up to you to implement your uthread API so that the reaper can safely do it’s job.

4 Compiling and Testing

Currently, it is not possible to take any (already compiled) program and have it use mthreads instead of pthreads. In order to use mthreads, you will need to add all of mthreads’s files to its project, modify it to use the mthreads API functions and recompile it.

4.1 Debugging

As always, use of gdb will make your life much easier when trying to get this assignment up and running. However, gdb can sometimes get confused in multithreaded programs, and may have trouble printing stack traces. If that happens, don’t worry; it doesn’t mean your code is broken. Also, since uthreads is built as a library, gdb won’t find the symbols in it right away, so tell it to wait for the “future shared library to load”.

4.2 Test Code

We can’t stress enough how important test code is in an assignment like this. Without proper test code, finding bugs will be next to impossible. Make sure to test all sorts of situations with lots of threads at different priority levels. The Makefile included with the assignment will compile a simple test program which uses the uthreads functions, just to get you started (run ./test from the directory your uthreads library is in). If it runs and exits cleanly, most of your basic functionality is working, but be sure to test more complicated cases.

Judicious use of assert() will help you both understand your threads package and debug it. This is your first real systems-level coding project, and it is highly recommended that you assert a general sane state of the system whenever you enter a function. Thinking about what a “sane state” means should lead to a greater understanding of what is happening at any given time and what could go wrong. A caveat though: if you have an assert that fails in uthread_yield(), your program will enter an infinite loop due to assert() calling write() calling uthread_yield() and so on.

A final warning: printf() is NOT thread-safe. This means that while your program may appear to be executing incorrectly, it may just be that the data structures used for buffering are getting clobbered since printf() makes multiple calls to write(), and the TA code interpositions and thus yields control around each individual call to write(). If you are going to write a program to test your uthreads, consider using a combination of sprintf() and write() like is done in the test program the TAs provide for you. As described above, however, write() depends on uthread_yield and calling it may cause your program to fail if the scheduler isn’t yet fully functional. In such a situation you will need to use gdb to debug your program.
5 Handing In

To hand in your finished assignment, please run this command while in the directory containing your code: `~/course/cs167/bin/cs167_handin mthreads`. 