Project Uthreads

Due: February 8, 2017

This assignment must be completed by all 167 students (including those who are registered for 169).

Contents

1 Introduction

In this project you will develop and test your very own user-level threads package complete with thread creation/deletion/joining, mutexes, condition variables, and a priority-based scheduler. Having completed this assignment, you should be able to write multi-threaded applications using your package instead of, say, *pthreads*. You can copy the stencil code out of the `/course/cs167/asgn/uthreads_stencil` directory.

2 Background

Linux (and most other operating systems) have the notion of a "machine context" which contains the complete state (at least everything viewable from user-land) of the CPU at any given time.

Usually this includes a pointer to the stack, register contents, and other bookkeeping information. The idea is that since contexts contain a complete description of the CPU state, you should be able to save the current state and exchange it with another state (which has either been previously saved or constructed by hand). This enables the operating system to suspend and resume execution of programs at will, and also enables you to write a user-land threads package.

Let's try an example. Say you have threads T_A, T_B, and T_C with contexts C_A, C_B, and C_C of which T_A is currently running. Say T_A calls `swap(`C_A, C_B`)`, causing the current machine state to be saved as C_A and the thread T_B to begin executing. T_B then does a `swap(`C_B, C_C`)`, causing T_C to start running. Eventually T_C decides that, for whatever reason it wants to allow thread T_A to run again. When it calls `swap(`C_C, C_A`)`, the machine state that was previously saved as C_A is restored, and T_A starts running again. From T_A's perspective, the call to `swap()` simply did nothing except pause for some period of time.

It is important to note that context and threads are fundamentally different things. A context is merely a snapshot of the CPU as it is at a point in time/code. Threads are a kernel construct that contain a bunch of other information which is unimportant to the hardware (scheduling priority, thread state information, `errno`, signal mask, etc.). Threads contain a context, not the other way around.

The other little bit of magic that you will be (indirectly) employing is interpositioning, which Professor Doeppner touched on briefly in his lectures on Linkers and Loaders in CS033. When you run an executable that is using your *uthreads* package, some system calls will be interposed on by our wrapper code. This is done so that we can effect the rescheduling of a thread whenever it calls a system call that would normally block. In our syscall wrapper, we will call your `uthread_yield()` function which will cause another, appropriately prioritized thread to begin executing.

3 The Assignment

uthreads is a user-level threading package which has been loosely based on the familiar *pthreads* interface. It supports the creation of threads which can be joined with or detached, a simple priority based scheduler, mutexes and condition variables. You will be writing the majority of the *uthreads* code, but your generous TAs have provided you with some code for dealing with dead threads in addition to some wrappers around the Linux functions for creating and swapping contexts. The *uthreads* functions which we give you that you might have to call yourself are:

```
// uthread.c
char *alloc_stack();
void free_stack(char*stack);
void make_reapable(uthread_t *uth);

// uthread_ctx.c
void uthread_makecontext(uthread_ctx_t *ctx, char *stack,
                         int stacksz, void (*func)(),
                         long arg1, void *arg2);
void uthread_swapcontext(uthread_ctx_t *oldctx,
                         uthread_ctx_t *newctx);
```

as well as the functions in `uthread_queue.h` and the macros in `list.h`.

The *uthreads* API functions which you must implement are:

```
// uthread.c
void uthread_init();
int uthread_create(uthread_id_t *uidp, uthread_func_t func,
                   long arg1, void *arg2, int prio);
void uthread_exit(int status);
int uthread_join(uthread_id_t uid, int *return_value);
int uthread_detach(uthread_id_t uid);
uthread_id_t uthread_alloc();
void uthread_destroy (uthread_t *uth);

// uthread_sched.c
void uthread_yield();
void uthread_block();
void uthread_wake(uthread_t *uthr);
int uthread_setprio(uthread_id_t id, int prio);
void uthread_switch();
void uthread_sched_init();
void uthread_start_timer();
void clock_interrupt(int sig);
void uthread_nopreempt_reset();

// uthread_mtx.c
void uthread_mtx_init(uthread_mtx_t *mtx);
void uthread_mtx_lock(uthread_mtx_t *mtx);
int uthread_mtx_trylock(uthread_mtx_t *mtx);
void uthread_mtx_unlock(uthread_mtx_t *mtx);

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

This may seem like a lot of work, but most of these functions are short. As a case-in-point, we are providing you with 800+ lines of stencil code and the TA implementation is less than 1000 lines total. So, while this assignment contains some concepts that you might not have been exposed to, it is mercifully short.

These functions are marked by `NOT_YET_IMPLEMENTED` macros found in the code. To see a list of all the functions you still need to implement, type `make nyi` from the command line, while in the same directory as the makefile we have included.

3.1 Overview

Each of the functions mentioned above has 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_init()`. This should be called exactly once and is responsible for setting up all of your data structures such as the global `uthreads` array and `ut_curthr` (which you should make sure is always set to the thread that is currently executing). There is some special code provided for you in `uthread_init()` that will deal with making the currently executing context (from which the executable just called `uthread_init()`) a valid uthread and setting up `ut_curthr`. See the comments around `create_first_thr()` for more information, if you are interested. Once everything is initialized and `uthread_init` has completed, you can create threads using `uthread_create()`.

Once you can create threads, you need to schedule them. Whenever a thread needs to temporarily yield the processor to another thread (but still remain runnable), it should call `uthread_yield()`. You will need to call this in your interrupt handler when preemption is enabled. The support code also uses it as a hook to invoke your scheduler inside our syscall wrappers. Threads can be put to sleep indefinitely and woken up using `uthread_block()` and `uthread_wake()` respectively. Choosing another thread to run is done inside the `uthread_switch()` function. Your scheduler should take thread priorities into account (which are set by `uthread_setprio()`). To do this, we recommend using a table of separate round-robin queues, one queue for each priority level. This data structure has been provided for you as `runq_table` in `uthread_sched.c`.

In general, the *uthreads* assignment has been designed to behave like a system you are familiar with, *pthreads*. As such, it has functions to create, detach, and join threads. As with *pthreads*, if an undetached thread finishes executing, its cleanup should be deferred until a call to `uthread_join()` is made. The functions for dealing with mutexes and condition variables should be pretty straightforward as you are familiar with their expected functionality from CS033.

3.2 Assumptions

A note on some of the assumptions that you may make when writing this assignment: *uthreads* will never have more than one thread running at any one time. Handling multiple CPUs (the ability to run more than one thread concurrently) is beyond the scope of this assignment. This means that as long as preemption is disabled (which can be done with `uthread_nopreempt_on`) global data structures cannot be modified by other threads.

3.3 Swapping Contexts

In *uthreads*, you will use `uthread_makecontext()` to create a machine context for a thread. When you want to change which thread is currently executing (i.e. in your scheduler), you will need to call `uthread_swapcontext()`. This will cause the current CPU context to be saved and a new context to begin executing. The saved context will resume at a later time when `uthread_swapcontext()` is called with it as the `newctx` argument.

3.4 Time Slicing and Preemption

Previous versions of *uthreads* only allowed for threads to be de-scheduled if they explicitly `yield`ed the processor. Most interfaces for threads do not work this way[1]; instead the execution of various processes are interleaved implicitly by the scheduler.

A common way of implementing this is by time slicing: each thread is granted a certain amount of time that it can run continuously. After this time is up, the thread is "preempted" (i.e. descheduled) and another thread allowed to run. This is typically implemented by scheduling a periodic timer interrupt, and yielding the processor in an interrupt handler.

Since we are in user space, and don't have interrupts, you will use interval timers (see `setitimer`) and `SIGVTALRM` to simulate a timer interrupt; your signal handler will simulate an interrupt handler.

One thing to keep in mind when modifying a threads priority from a `uthread`'s context is that if the target thread (the thread whose priority is being modified) now has a higher priority than the calling thread, the calling thread should yield to the target thread immediately (see `uthread_setprio`), to ensure that threads with higher priorities run more frequently than ones with lower priority.

3.4.1 Dangers of Preemption

Unfortunately, adding preemption to your code does not "just work." Some routines must run to completion without yielding the processor - for example, what might happen if `uthread_mtx_lock` is preempted in the middle of execution? Use the `uthread_nopreempt_on/off` functions to maintain the correctness of key functions.

3.5 Dealing with Dead Threads: The Reaper

As discussed in lecture, the reaper thread is responsible for cleaning up the resources of unused threads. It is important to note that the reaper does not fully clean up non-detached threads which have finished but not yet been joined. Rather, it leaves that work for `uthread_join()`. We have given you a complete reaper as `reaper` in `uthread.c`. You should look at it to understand what this means.

3.6 Error Reporting

As a rule, one should use the standard error types defined in `errno.h` (although it should not be necessary to include this file to use the values) as mentioned in the *pthreads* `man` page. We will follow the same convention that the `pthread` functions use, which is to return the proper error code.

4 Compiling and Testing

Currently, it is not possible to take any (already compiled) program and have it use *uthreads* instead of *pthreads*. In order to use *uthreads*, you will need to add all of *uthreads*'s files to its project, modify

[1]And when they do, they often aren't called threads; more often, they are called coroutines

it to use the *uthreads* 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".

Also note the variables `clock_count` and `taken_clock_count`. The variables are just debugging/informational tools. `clock_count` should be incremented on every clock interrupt, while `taken_clock_count` should only be incremented if the clock interrupt actually results in a thread switch (i.e. when preemption is allowed).

Essentially `taken_clock_count` <= `clock_count` and they only fall out of sync when a clock interrupt is received but is "masked".

When debugging segfaults, sometimes you may want to run the program outside of GDB, perhaps because the program runs correctly under GDB, but not when run independently. In this case, it can sometimes be useful to enable core dumps. A *core dump* is a copy of the program's memory that is generated and written to disk (typically with a name like "core") in the current working directory when the program segfaults. Then, you can attach to the core dump with GDB with `gdb path/to/program/binary path/to/core`, and GDB will tell you where your program segfaulted, and you can examine the state of your program. To enable core dumps, just type `ulimit -c unlimited`. Note that core dumps take a lot of space, so you will want to delete them after you are done with them. Also, you cannot attach to a core dump generated from an old binary; that is, if you recompile, you will not be able to attach to core dumps from a previous compilation, unless you preserve the binary somewhere, but generally, you don't need to hold on to old cores or binaries.

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 uthreads`.