User-level Thread Library

Fall 2019 CSCi 5103 Project 1, Due at midnight Oct. 21

1. Overview

In this project, you are to implement a user-level thread library called ***uthread*** which mimics the interface of the *pthread* library and runs in user space. The purpose of this project is to deepen your understanding of the mechanism and design trade-offs behind threads in operating systems. Additionally, practitioners in industry are sometimes required to implement their own thread library for various reasons: (1) absence of a *pthread* library on some embedded systems; (2) need for fine-grained control over thread behavior; and (3) efficiency. Your solution **must run on** a Linux machine in CSE labs.

2. Project Details

2.1 User-Level Thread (ULT)

What is a user-level thread (**ULT**)? In this project, a ULT is an independent control flow that can be supported within a process, at the user-level. For example, thread A calls foo(), which calls bar(), thread B calls baz(), while the main program is waiting for threads A and B to finish. Each thread needs a stack of its own to keep track of its current execution state. Instead of relying on a system library or the operating system to manage the stack and to coordinate thread execution, ULT is managed at the user-level, including memory allocation and thread scheduling.

Your *uthread* library shall support the following interfaces:

- *pthread* equivalents. Each API provides the same functionality as its *pthread* counterpart, except that tid is represented as an integer. Otherwise, use return values to indicate success or failure.
    - **int uthread_create(void *(*start_routine)(void *), void *arg); // returns tid**
    - **int uthread_yield(void); // return value not meaningful**
    - **int uthread_self(void); // returns tid**
    - **int uthread_join(int tid, void **retval);**
- *uthread* control. This API allow application developers to have more fine-grained control of thread execution.
    - **int uthread_init(int time_slice);** (explained later)
    - **int uthread_terminate(int tid);**
    - **int uthread_suspend(int tid);**
    - **int uthread_resume(int tid);**
    - **int lock_init (lock_t*);**
    - **int acquire (lock_t*);**
    - **int release (lock_t*);**
- **The lock solution should not use any blocking system calls and is based on atomic instructions provided to you.**

- (Optional: no credit) add a notion of priority to a thread or other scheduling algorithms; add a stub for the start function of a thread.

Other than the pre-defined API(s), feel free to define additional ones you deem necessary or wish to explore. Describe your customized API(s) in a short document.

2.2 Context Switch

As discussed in class, a thread context is a subset of a thread's state, such as registers, stack environment, and signal mask, that must be saved/restored when switching contexts. Actually, the entire stack does not have to be saved in the thread context, only a pointer to the current top of the stack which is *sp* (think of why?). When a thread yields the processor (or is preempted), the *uthread* library must save the thread's context at that moment. When the thread is later scheduled to run, the library restores all of the saved registers that belong to the thread.

How is a new thread created? The answer is either by creating a fresh new context or making a copy of the currently running thread's context (remember `fork`?). In this project you should adopt the latter method since we want you to manipulate the values in the context directly. The C/C++ programming language provides two sets of library calls that: (1) retrieve the current context of the caller application, (2) store the context at a specified memory location, and (3) later allows the caller application's context to be set to a previously saved value. The C/C++ library calls are: `sigsetjmp` and `siglongjmp`. You may read the glibc manual to learn how they work (though we will supply basic demo code for you). To create a new thread, you need to change the saved value of *pc* to the thread entry function for the new thread, and *sp* to the specified memory location.

Per-thread information can be maintained in a Thread Control Block (TCB), which includes:

- Thread id (tid)
- Saved context (sigjmp_buf, or ucontext_t)
- Stack pointer (sp)
- Thread State // ready, running, blocked, terminated, etc
- Thread entry function
    - Entry function input parameters (arg)
    - Entry function's return value (which is a pointer to output data)

2.3 Yield, Preempt, and Scheduler

As you may have realized, all threads run within the same process, which runs on a single processor, meaning that only one thread can run at each time. To allow threads to run in turn, the *uthread* library uses two techniques: (1) relying on each thread's cooperation to yield processor from time to time, and (2) preempting a running thread and switch to another one after a time period. To begin with, you may first assume threads will call yield. Once you have implemented yield, preemption is very similar, for they both perform a context switch.

Your next task is to implement a round-robin scheduler using time slicing to achieve fair allocation of the processor to all active threads within a process. Since every thread is allotted the same time slice to run, this simply means that the scheduler is run once every <time slice> microseconds to choose the next thread to run. Once a running thread's time slice has expired, it is preempted, moved to the READY state, and placed at the end of the READY queue. If a running thread is suspended (discussed later) or yields voluntarily, its remaining time slice will be abandoned. Call uthread_init(time_slice) to set the length (in ms) of the time slice.

You can use setitimer for interval timers, and note that the timer should count down against the user-mode CPU time consumed by the thread (call setitimer with ITIMER_VIRTUAL instead of ITIMER_REAL). A caveat for the timer interrupt is that it may arrive during uthread_create, uthread_yield, and uthread_terminate when critical data such as the READY queue is being modified. You may need to disable timer interrupts after entering these API(s) and enable interrupts before leaving. Since the lab relies heavily on signals, you want to read up on: `sigprocmask`, `sigemptyset`, `sigaddset`, `sigaction`. You may not use Posix locks since these are blocking system calls.

2.4 Suspend, Resume

At this point, *uthread* library has a severe limitation: when a thread makes a system call that blocks, the whole process suspends. An efficient implementation would allow the thread to block "within the library" w/o knowledge of the OS, and to schedule other threads that are ready to run. To fix this issue, you first need to implement the suspend/resume API. Any thread can suspend/resume itself or any other thread with its tid. When a thread is suspended, if the thread is in RUNNING state, it is moved to the SUSPEND queue, a reschedule is triggered, and the time slice should be reset; if the thread is in READY state, remove it from the READY queue and place it in the SUSPEND queue.

3. Project Group

Students should work in groups of size 2 or 3. It is advised to form groups of 3, such that each member could lead the development of one subproblem (context switch, scheduler, etc). **Only one submission is allowed for each group. You may use C or C++. You may look on-line ONLY for the meaning of basic system calls relating to signals and other documentation, or to code to help with implementing queues and supporting needed data structures.**

4. Test Cases

You should also develop your own test cases and provide documentation that explains how to run each of your test cases, including the expected results and an explanation of why you think the results look as they do. For example, you could build a demo application which calls all functions on your *uthread* API(s). Sample test cases are provided including ones we will use to grade your project.

5. Deliverables

1. A **concise** document that describes your important design choices, including:
   a. What (additional) assumptions did you make?
   b. Describe your API, including input, output, return value.
   c. Did you add any customized APIs? What functionalities do they provide?
   d. How did your library pass input/output parameters to a thread entry function?
   e. How do different lengths of time slice affect performance?
   f. What are the critical sections your code has protected against interruptions and why?
   g. If you have implemented more advanced scheduling algorithm(s), describe them.
   h. Does your library allow each thread to have its own signal mask, or do all threads share the same signal mask? How did you guarantee this?
2. Source code, makefiles and/or a script to start the system, and executables (No object files).

6. Grading

1. (Basic Rule: only verified code earn credits) Project owners are responsible for proving that their code could work. How? By calling all the functions in user-defined test cases. In real-life, the ratio between test code and functional code is 8:1 (anecdotally). We don't require students' code to achieve production-level quality. But the minimum requirement is that every function should be called at least once. Uncalled functions do not count, since they are not verified.
2. (Rough distribution of scores; this may change somewhat)
   a. (+20) Correct implementation of uthread_init()
      i. Initialization of TCB READY queue
   b. (+20) Scheduler, time slicing
      i. Basic context switch
   c. (+30) Correct implementation of uthread_create(), uthread_terminate(), uthread_join()
   d. (+10) Correct implementation of lock/unlock
   e. (+20) Document, answering questions

7. Hints

1. If you do not use a stub, make sure all user thread functions call **uthread_terminate.**
2. Look into uthread_demo.c to see how to set up stacks for different threads and how to do a context switch, to compile the demo code, just run:
   gcc uthread_demo.c -o uthread_demo
3. To understand sigsetjmp and siglongjmp, check
   http://vmresu.me/blog/2016/02/09/lets-understand-setjmp-slash-longjmp/
4. For timer interrupts, look at setitimer() and ITIMER_VIRTUAL, SIGVTALRM.
5. For lock/unlock, check the TAS function in demo code.

6. Look at test1.cpp and test2.cpp, develop your own test cases accordingly. There should be enough evidence in the output for us to tell if your implementation works correctly or not.
7. You code should be able to execute on CSE lab machines, the hostname of these machines: csel-kh4250-[01|02|...|49].cselabs.umn.edu