User-level Thread Library

Fall 2017 CSCi 5103 Project 1, Due at midnight Oct. 23

1. Overview

In this project, you are asked to implement a user-level thread library called the ***uthread***. *uthread* mimics the interface of *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. Apart from an educational purpose :-), practitioners in industry are sometimes required to implement their own thread library for various reasons: (1) lacking of *pthread* library on some embedded systems; (2) needs for fine-grained control over thread behavior; and (3) efficiency. Your solution **must run** a Linux machine in the 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.

uthread API

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.
  - **int uthread_create(void *(*start_routine)(void *), void *arg);**
  - **int uthread_yield(void);**
  - **int uthread_self(void);**
  - **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);** (Will be explained in Sec. 3.1)
  - **int uthread_terminate(int tid);**
  - **int uthread_suspend(int tid);**
  - **int uthread_resume(int tid);**

- Asynchronous I/O. This provides the appearance of a “blocking” call to the user, but hides the blocking by using non-blocking I/O with polling and/or signals.
    - **ssize_t async_read(int fildes, void *buf, size_t nbytes);**
- (Optional) *uthread* synchronization. If you are feeling very ambitious, you can try to add locks or other kinds of synchronization. Not required though.

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.

Fig. 1 shows all states of a ULT and how *uthread* API(s) trigger ULT state transitions.

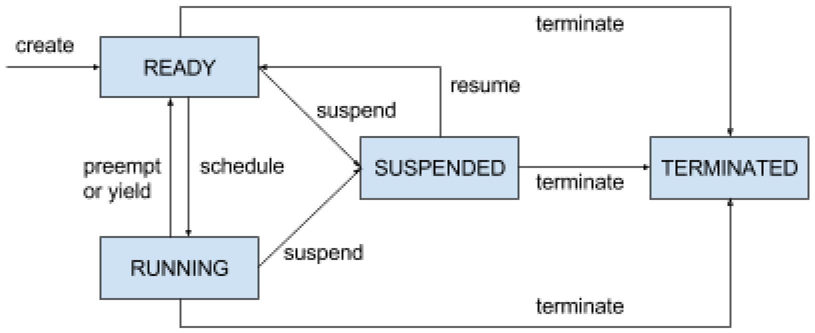

Fig. 1. ULT States and State Transitions

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 [1]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 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 library calls are: (1) `sigsetjmp` and `siglongjmp`, and (2) `getcontext` and `setcontext`. You may read glibc manuals to learn how either pair of calls work. After you learn the nuanced difference between `sigsetjmp` and `getcontext`, you are given the choice of selecting which one to use. To create a new thread, you need to change the saved value of *pc* to

[1] Unless otherwise specified, the term thread in the rest of this document refers to ULT.

the thread entry function for the new thread, and *sp* to the specified memory location. If you choose to use functions from the getcontext family, you may NOT use makecontext and swapcontext in your final code. We want you to manipulate the fields in the saved ucontext_t directly. E.g., push entry function's input parameter to the stack.

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
- 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 may consider 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`.

2.4 Suspend, Resume, and Asynchronous I/O

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.

Since the *uthread* library runs at user-level, how does it detect blocking system calls and suspend the calling thread? One solution is to provide wrapper or jacket around system calls. In this project, you are asked to implement the asynchronous read wrapper function. A simpler yet inefficient implementation of async_read would be to poll the status of the read fd, and call yield if it is not ready. You may use the aio functions to perform this, which is acceptable to us. Alternatively, you may try to implement a more-efficient solution: instead of polling, which wastes CPU cycles during context switches, the calling thread could be suspended and resumed by a SIGIO signal sent when the input fd is ready. In this implementation, you may need to maintain a mapping from fds to suspended tids. Note that in either case **async_read** still appears synchronous or "blocking" to the caller.

Note also that calling async_read or join are the only ways currently that your threads becomes suspended.

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, async io). **Only one submission is allowed for each group.**

4. Test Cases

You should also develop your own test cases for all the implementations, 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 your *uthread* API(s).

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 did you implement context switch? E.g. using *sigsetjmp* or *getcontext*?
f. How do different lengths of time slice affect performance?
g. What are the critical sections your code has protected against interruptions and why?
h. If you have implemented more advanced scheduling algorithm, describe them.
i. How did you implement asynchronous I/O? E.g. through polling, or asynchronous signals?
    i. If your asynchronous read is based on asynchronous signals, how did you implement it? Does it support concurrent signals, e.g. when multiple fds are ready at the same time?
j. 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).