User-level Thread Library

Spring 2025 CSci 5103 Project 1

Final Submission Due Feb. 26
Intermediate Submission Due Feb. 19

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 mechanisms 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** on 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 begins at foo(), which calls bar(), thread B begins at 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. 2.3)
    - **int uthread_exit(void *retval);**
    - **int uthread_suspend(int tid);**

- **int uthread_resume(int tid);**
- **int uthread_once(uthread_once_t *once_control, void (*init_routine)(void))**

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 an ULT and how *uthread* API(s) trigger ULT state transitions.

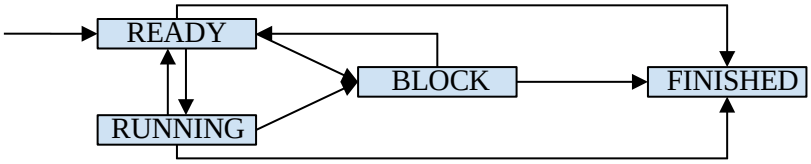

Fig. 1. ULT States and State Transitions

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. The C/C++ programming language provides the `getcontext` library call that: (1) retrieves the current context of the caller application, (2) stores the context at a specified memory location, and (3) later allows the caller application's context to be set to a previously saved value. 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. To do this you can use the `makecontext` library function. To switch to a configured thread context, use the `setcontext` library call. You should not use `swapcontext` (use `getcontext` and `setcontext` instead). Refer to the man page for details on how this family of functions work in detail. You can also refer to the provided demo code to see an example of switching between two threads. Why does return `getcontext` twice?

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

When a thread returns from its top-level function where does it return to? In order to have the thread return to a predictable location (and not just some garbage address on the stack) you can use a stub function. The stub function's job is to simply call the top-level thread function and then call *uthread_exit* after the top-level function has returned. The initial thread context should be set up to call the stub instead of the top-level thread function directly.

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

- Thread id (tid)
- Scheduling quantum count
- Thread state: READY, RUNNING, BLOCK, FINISHED
- Stack pointer (sp)
- Saved context (ucontext_t): provided by `getcontext`

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 the processor from time to time, and (2) preempting a running thread and switching 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 us) 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 other uthread library functions when critical data such as the READY queue is being modified. You may need to disable timer interrupts (SIGVTALRM) after entering these API(s) and enable interrupts before leaving (**user code must always run with interrupts enabled, i.e. when control leaves the UTL library/API and returns back to user code**). Since the lab relies heavily on signals, you want to read up on: `sigprocmask`, `sigemptyset`, `sigaddset`, `sigaction`.

2.3.1 Priority Scheduler (Not required/graded; just for extra fun)

Modern operating systems typically have scheduling algorithms beyond a simple FIFO ready queue. The Linux Completely Fair Scheduler (CFS), for example, uses a red-black tree to create a fair scheduling policy that aims to ensure fairness by rewarding interactive threads that do not hog the CPU. To do this,

extend your simple FIFO scheduling algorithm to a more complex algorithm and describe its design and design goals in the document deliverable.

2.4 Suspend and 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 any other thread with its tid. When a thread is suspended, if the thread is in RUNNING state, it is moved to the BLOCK 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 BLOCK queue.

2.5 Join

In order to synchronize thread completions, a thread can call *uthread_join* to block until the specified thread has terminated. Joining on a thread requires careful attention since it is possible that the thread is still running or that it has already terminated. If the thread has already terminated, *uthread_join* should not block. If the thread is still running, *uthread_join* should block and wait for termination. For the former case, it is important that the *uthread* library maintains some state about a thread after it has terminated so that a thread calling *uthread_join* knows whether or not it has already terminated. *uthread_join* can then clean up the terminated thread's state.

Additionally, threads can return an output parameter (*void**) that can be received by another thread using *uthread_join.* After a thread gracefully terminates, the output parameter pointer should be saved so that it can be returned by *uthread_join.*

Hint: Joining requires a fairly complex interplay between *uthread_join* and *uthread_exit*. You may want to maintain a join queue (for managing threads that are waiting for certain other threads to complete) and a finished queue (for holding threads that have already terminated that have yet to be joined). Example queue entry types for these two queues are provided in the skeleton code.

User code is expected to join on every thread created (*pthread* has the same requirement but additionally allows you to detach threads – which is not supported in our *uthread* library), so you should not free a thread ID until it has been joined.

2.6 uthread_once

The *uthread_once* function ensures that a specified initialization routine is executed only once, even if multiple threads attempt to invoke it concurrently. It follows the same concept as pthread_once, using a synchronization mechanism to prevent redundant executions. The function takes a control variable (uthread_once_t *once_control), which tracks whether the initialization routine has been executed. If the routine has already run, subsequent calls return immediately without executing it again. This functionality is particularly useful for initializing shared resources, such as global variables or data structures, that should only be set up once per program execution.

Internally, *uthread_once* should safely access the execution_status flag within the uthread_once_t structure. If

the flag is set to “not executed” you should execute the init_routine and set the flag to “executed”.

3. Project Group

Students should work in groups of 2, such that each member could lead the development of one subproblem (context switch, scheduler). **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).

Challenge:

Can you generate a test case which demonstrates the benefits of using self-implemented context switching (uthread_yield, uthread_suspend, uthread_resume) to decrease the execution time of your overall program? Think of a scenario where your whole process can avoid getting blocked (e.g. async I/O).

5. Deliverables

1. A **concise** document (writeup.pdf) that describes your important design choices, including:
    a. For Grader:
        i. How to compile
        ii. How to run
        iii. How to test
    b. What (additional) assumptions did you make?
    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? What must `makecontext`, do “under the hood” to invoke the new thread execution?
    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 a more advanced sched algorithm for extra credit, describe it.
2. Source code, test case files, and a Makefile

All files should be submitted in a single tarball (.tar.gz). The following command can be used to generate the file:

```
$ tar -cvzf submission.tar.gz project_folder
ables or left-over testing files.
```

The submitted tar file should NOT contain any compiled executables or left-over unnecessary files. Additionally, the Makefile should include a “clean” target which will allow the grader to quickly clean and re-test multiple times.

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. (Tentative grade breakdown)
    a. (+10) Intermediate submission (must be on-time, compile and meet requirements)
    b. (+25) Scheduler, context switching, time slicing
    c. (+40) Correct implementation of uthread library
    d. (+25) Document, answering questions

7. Intermediate Submission

1. Complete *uthread_init*, *uthread_create*, and *uthread_yield*.
   In order to get these functions working you will have to at least complete TCB.cpp as well as some thread queuing code.

2. Be able to setup a timer interrupt and be able to disable and enable timer interrupts
   You do not have to integrate interrupts into the library yet (this should be done after you have *uthread_yield* working).