COMP 321: Introduction to Computer Systems

Project 2: User-level Threads
Assigned: 2/12/25, **Due: 2/27/25, 11:55 PM**

Overview

You will write a subset of the `pthreads` library that is implemented using user-level threads.
The goals of this project are as follows:

- Understand threads and user-level context switching and scheduling.
- Become familiar with the concepts of I/O multiplexing and non-blocking I/O.
- Use library interpositioning.

General Overview of Pthreads

A Unix process starts with a single kernel-level thread that runs the `main` function of the program. A kernel-level thread is an execution context that is scheduled to run by the kernel. This initial kernel-level thread may spawn additional kernel-level threads within the same process. All threads within the same process share the process state (including memory, file descriptors, etc.). Each kernel-level thread, though, has its own context (including the program counter, other register state, a stack, a signal mask, etc.). In contrast, user-level threads implement their own thread scheduler that runs within a single kernel-level thread to provide the capability to run multiple thread contexts within that kernel-level thread.

In this project you will implement a user-level threading library. This is similar to the underlying infrastructure that Go uses to implement goroutines and Java uses to implement virtual threads, for example. Moreover, such modern user-level thread implementations also rely on I/O multiplexing and non-blocking I/O.

`Pthreads` is the most commonly used threading library in C, which is also used to implement threads in higher-level languages, such as Java. `Pthreads` is currently implemented using kernel-level threads, but was originally implemented using user-level threads.

You will implement a user-level thread library, `uthreads`, that contains equivalent implementations for the following functions from the `pthreads` library:

- `pthread_create`
- `pthread_detach`
- `pthread_self`
- `pthread_exit`
- `pthread_join`
- `sched_yield`

`Pthreads` applications that only use these functions (without options, as you will not implement the `pthreads` options) should be able to use the normal `pthreads` library or your library interchangeably.

A multi-threaded program creates new threads using `pthread_create`. This function creates a new thread and runs a specified function within that thread. As threads share the address space, the new thread has access to exactly the same memory as the thread that spawned it, including global variables, the heap, etc. So, if there are global variables, for example, they can be accessed by either thread. This is both good (as it is convenient) and bad (as it can create data races if you write to those global variables).

A thread ends when the specified function returns or the thread calls `pthread_exit`.

A thread can either be "detached" or "joinable". A newly created thread starts out in the "joinable" state. This means that when it finishes, another thread must call `pthread_join` to clean up the resources of the thread and receive the return value (if desired). When a thread calls `pthread_join` it will wait for the thread it is trying to join with to complete (if it has not already done so).

A thread can be placed in the "detached" state by calling `pthread_detach` at any time. The resources of a detached thread are cleaned up automatically when the thread ends and the return value is lost.

The `pthread_self` function returns a thread identifier that can be used to identify the currently running thread in other pthread calls.

The `sched_yield` function yields control back to the thread scheduler, allowing other threads to run before the calling thread is run again.

Non-blocking Operations

Given that all `uthreads` within a process execute in the same kernel-level thread, if any thread "blocks" in a system call, then the entire process becomes blocked. This means that until that thread unblocks, no threads can execute, even if there are other threads that are ready to run.

Therefore, in `uthreads`, or any user-level thread library, it is important that all (or at least as many as possible) operations be non-blocking. You will therefore need to provide non-blocking implementations of the following common system calls that your threads can use:

- `socket(2)`
- `accept(2)`
- `read(2)`
- `write(2)`

Note that this does **not** mean that you will need to reimplement these functions! Rather, it means that you will need to provide wrappers that call the underlying functions in a non-blocking way and correctly handle the situations in which they would block (for example, by blocking the thread, switching to another thread, and not rescheduling this thread until the operation can complete).

Thread Preemption

Again, given that all `uthreads` within a process execute in the same kernel-level thread, if a thread only runs computations and never makes any blocking calls, then it could run within the process forever, preventing any other threads from running. Therefore, there needs to be a way to "preempt" running threads. This is typically done by allowing each thread to run for a maximum duration before it is preempted and another thread is allowed to run. The `setitimer(2)` function can be used to set a recurring timer that will deliver a `SIGPROF` signal at the specified interval. The timer accounts for time spent both in user-space and kernel-space, so it includes any time the thread spends in non-blocking system calls.

Whenever the `SIGPROF` signal is received, the library should switch into the scheduler to enable it to switch to running a different thread. However, if you switch threads at an "inopportune" time, it can be problematic. Therefore, you must block `SIGPROF` signals during any operations that modify shared state. This includes the scheduler, which will modify queues of threads, and several standard library functions that utilize shared state.

To help deal with this issue, we have provided protected implementations of the following standard library functions that your thread can use:

- `malloc(3)`
- `calloc(3)`
- `realloc(3)`
- `free(3)`

These functions are only thread-safe (meaning multiple threads can call them without issue) when using kernel-level threads. When they detect that there is only a single kernel-level thread within a process, they assume that they will only ever be called from a single thread, so they perform optimizations that make them not thread-safe. The provided wrappers ensure that they can only be called from a single user-level thread at a time. Otherwise, they could exhibit undefined behavior.

Thread Scheduler

A key component of the `uthreads` library is the thread scheduler. The scheduler will need to manage several thread queues. The scheduler will be responsible for making sure that it properly runs threads that are not blocked, that it unblocks threads when the resource upon which they were blocked becomes ready, and that zombie threads are reaped appropriately. The scheduler should do this in a fair way so that threads do not starve each other. This means that the runnable threads should get roughly equal shares of the CPU time. And when threads are blocked, no runnable thread should run twice before the scheduler checks if blocked threads can become runnable.

Each thread can be in one, and only one, of the following states at any given time:

- `FREE`: this thread context is currently not being used.
  - A `FREE` thread becomes `RUNNABLE` when it is allocated for use by `pthread_create`.
- `RUNNABLE`: this thread context is currently runnable, which means that it has not completed and it has not been blocked.
  - A `RUNNABLE` thread becomes `BLOCKED` when it tries to perform an I/O operation that would block.
  - A `RUNNABLE` thread becomes `JOINING` when it calls `pthread_join` and has to wait for the other thread.
  - A `RUNNABLE` thread becomes a `ZOMBIE` when it calls `pthread_exit` or returns.
- `JOINING`: this thread context is waiting to join with another thread context.

  - A `JOINING` thread becomes `RUNNABLE` after the thread context it is waiting to join with completes.

- `ZOMBIE`: this thread context has completed, but has not yet been reaped.
  - A detached `ZOMBIE` thread becomes `FREE` after it has been reaped.
  - A joinable `ZOMBIE` thread becomes `FREE` after it has been joined.
- `BLOCKED`: this thread context is blocked waiting for I/O.
  - A `BLOCKED` thread becomes `RUNNABLE` when the I/O upon which it is blocked is ready.

To manage the threads in these states, your scheduler should use the following four queues:

- The *run* queue should contain all of the threads in the `RUNNABLE` state in the order that they will be scheduled.
- The *blocked* queue should contain all of the threads in the `BLOCKED` state.
- The *reap* queue should contain all of the detached threads in the `ZOMBIE` state.
- The *free* queue should contain all of the threads in the `FREE` state.

Note that there is no need to hold the threads in the `JOINING` state in any queue, as they will be become `RUNNABLE` when the thread upon which they are waiting to join exits. Similarly, there is no need to hold the joinable threads in the `ZOMBIE` state in any queue, as the thread that joins with them will ultimately reap them.

The scheduler is implemented by the internal function `uthr_scheduler`, and executed by a dedicated `ucontext_t`, `sched_uctx`, that is initialized to run with `SIGPROF` signals blocked, so that the scheduler will not itself be preempted. `uthr_scheduler` consists of an infinite loop that performs the following actions on each iteration:

- If the run queue is not empty, the scheduler calls `setitimer(2)` to reset the preemption timer, sets the global variable `curr_uthr` to the first thread in the run queue, and runs that thread by resuming its `ucontext_t`. When the scheduler itself is resumed, after running this thread, it checks whether the thread is still in the `RUNNABLE` state (which implies either the thread called `sched_yield` or it was interrupted by `SIGPROF`; in either case it did not block on I/O). If the thread is still in the `RUNNABLE` state, the scheduler ...
  - calls the internal function `uthr_check_blocked`, passing `false`, to move any `BLOCKED` threads that can now run to the tail of the run queue, and ...
  - moves the previously running thread to the tail of the run queue after any threads that were just unblocked.
- If the run queue is empty, the scheduler calls `uthr_check_blocked`, passing `true`, to wait for a `BLOCKED` thread to become runnable.
- The scheduler frees all of the threads in the reap queue.

The following `uthreads` library functions invoke the scheduler by resuming `sched_uctx`:

- `pthread_exit`

- pthread_join, specifying a RUNNABLE thread as the first argument
- uthr_block_on_fd, which is an internal function of the library that blocks the calling thread until the specified I/O operation can complete.
- uthr_timer_handler, which is an internal function of the library that handles SIGPROF signals.

The internal functions uthr_block_on_fd and uthr_check_blocked collaborate on the implementation of blocking I/O operations through three shared data structures: the thread's struct uthr, the blocked queue, and fd_state. For example, when the application calls read, this call will be directed to the uthread library's wrapper for read. This wrapper will itself call the standard library's read function. If that function returns -1 and sets errno to EAGAIN, then the wrapper will call uthr_block_on_fd. uthr_block_on_fd will update calling thread's struct uthr to reflect the file descriptor and operation (read or write) on which the thread is blocking. Then, uthr_block_on_fd will move the thread to the blocked queue, updating its state accordingly. Finally, uthr_block_on_fd will update fd_state to reflect that a thread is now blocked reading or writing on the specified file descriptor. For example, suppose that the thread was attempting to read from file descriptor fd. If the fd_state's readers count for fd is 0, then uthr_block_on_fd will perform FD_SET(fd, &fd_state.read_set). Then, it will unconditionally increment fd's readers count.

When the scheduler calls uthr_check_blocked, it passes a Boolean that indicates whether or not uthr_check_blocked should wait for a thread to become runnable. As described above, if there are already runnable threads, then the scheduler will direct uthr_check_blocked to not wait. In either case, uthr_check_blocked will perform a select on a copy of the read and write fd_sets maintained within fd_state. Then, uthr_check_blocked will iterate over the threads in the blocked queue, checking to see if the fd_sets returned by select indicate that any of the I/O operations that blocked threads are waiting on can complete. For example, suppose that the returned fd_set for reads indicates that a thread that was waiting to read from file descriptor fd can be unblocked. Then, uthr_check_blocked will decrement fd_state's readers count for fd, and if the updated count is 0, perform FD_CLR(fd, &fd_state.read_set). Finally, uthr_check_blocked will move the thread to the run queue.

The internal function uthr_init initializes the shared state of the scheduler, such as the various queues, and the thread structure that represents the initial thread that is already running, in fact, executing uthr_init itself and eventually main. It also sets up the handler for SIG_PROF signals and starts the preemption timer. uthr_init is defined as a constructor so that it will execute before the main function of the application program.

Library Interpositioning

The uthread library relies on *library interpositioning* to intercept calls to various standard library functions from the application, and itself make calls to those standard library functions. Library interpositioning was discussed in the lectures on Linking in COMP 222 and is described in Chapter 7 of the Textbook.

To support the implementation of the non-blocking I/O functions, the internal function uthr_lookup_symbol should use the functions dlsym and dlerror to look up and return the specified symbol's address. See the manual page for dlopen on CLEAR9 for a description of how to correctly use these functions.

Note that unlike the uthreads wrappers for functions like malloc and read, which are permitted to call their counterparts in the standard C library, the implementations of the pthreads functions, including sched_yield, within the uthreads library may **not** call their counterparts. More generally, no function within the uthreads library may call **any** pthreads functions from the standard C library (or any other library). Only applications should call `pthreads` functions, which will then call your `uthread` implementations of those functions (either by directly linking with `uthreads` or by using library interpositioning).

The "Checking Your Work" section below provides examples of how to use interpositioning in order to use your `uthreads` library in place of the `pthreads` library.

Getting Started

Please visit the web page at `https://classroom.github.com/a/5L0crBb9`. (If you are copying this URL, do not include the period at the end of the sentence, as it is not part of the URL.) This page should say "RICE-COMP321-S25-Classroom" and "Accept the assignment — Threads". Moreover, it should have a green button labeled "Accept this assignment"[1]. Please accept the assignment.

Upon accepting the assignment, you will be redirected to another web page. This page will confirm that you have accepted the assignment, and it will eventually (after you click refresh) provide you with a link to your personal repository for the assignment. Click this link to go to your personal repository.

The web page for your personal repository has a green button labeled "Code". Click this button. You should now see a text field with a URL. Copy or remember this URL.

Login to the `CLEAR9` system if you have not already done so. Type the following:

```
git clone [Copy the URL for your repo here]
```

You will be prompted for your `github` username and password.

Once the clone operation is complete, you will have a directory named

```
threads-[YOUR github ID]
```

Please `cd` into this directory, and do the following:

- Type a header comment at the top of `uthread.c` with your name and NetID.
- Type the command `make` to compile and link a skeleton thread library and some simple programs you can run to test it.

The provided code includes the following source files for the `uthreads` library:

1. `uthread.c`
2. `uthread_internal.h`
3. `unix.c`
4. `stdlib.c`

The `uthread.c` file is the heart of the `uthreads` library. It implements the `pthreads` functions, the thread scheduler, and `uthr_lookup_symbol` for implementing library interpositioning. You will need to complete this file to create a functioning `uthreads` library.

The `uthread_internal.h` file declares the utility functions of the `uthreads` library that are defined within `uthread.c` and can be called by functions in `unix.c` or `stdlib.c`. These functions are for internal use by the `uthreads` library and will not be called by the applications. You should not modify this file.

The `unix.c` and `stdlib.c` files contain the wrappers for the Unix system-level I/O and standard C library functions, respectively. You will only need to modify `unix.c` to complete these wrappers, not `stdlib.c`. These wrappers can call functions declared in `uthread_internal.h`.

[1]You may have to login to GitHub to see this page. If so, you will be prompted for your GitHub username and password.

Checking Your Work

We have provided some test programs to help you check your work (all of which will be built when running make):

- test_uthread: a simple test program that tests the edge cases of your pthread_create, pthread_detach, pthread_exit, and pthread_join functions.
- spin: a simple program that creates the number of threads specified on the command line (or 5 if nothing is specified) that each spin in a loop calling malloc(3) and then free(3), printing the thread id every 5000 times through the loop. The point of the spin program is to allow you to test preemption. You should see the threads preempt each other as it runs.
- tiny: a multithreaded implementation of the tiny web server from the text book. The point of the web server is to test blocking I/O.

The rio.c and rio.h files are solely for the use of these test programs (specifically the tiny web server). No rio functions should be called from your uthreads library.

To run the test program, run ./test_uthread after running make. This will run a sequence of tests and print messages to tell you whether your implementation passes or fails each test. You should get these tests working before trying the other test programs.

To run the spin program, you can run ./spin_pthread or ./spin_uthread. As the name suggests, the former runs the spin program using the system's pthreads library, while the latter runs the spin program using your uthreads library. As you run the two versions of the spin program, you will immediately see the difference between using kernel threads and user-level threads. Do not expect the output from the two programs to match! Rather, you should be able to successfully run the program using your library without crashing or locking up and never finishing. You should also be able to make spin_pthread use your uthreads library, if you have implemented the pthreads API correctly. In order to do so, you should run it as follows:

```
UNIX% env LD_PRELOAD=./libuthread.so ./spin_pthread
```

This should behave similarly to spin_uthread and demonstrates the use of library interpositioning to allow your library to be used instead of the default pthreads library (compare the output to running spin_pthread without the LD_PRELOAD).

Finally, to run the tiny web server, you can run ./tiny [-v] <port>, where <port> must be a number between 18000 and 18200 and -v turns on "verbose" mode, which you may find helpful in debugging your thread library. However, the web server needs files to serve. You can make it serve the COMP 222 website by doing the following:

```
UNIX% cd ~comp222/public_html
UNIX% ~/comp321/assignments/threads/tiny -v <port>
```

This assumes your assignment is stored in:

```
~/comp321/assignments/threads/
```

If it is not, you should adjust the second command accordingly. This will start the tiny web server in the course web page directory, so it will serve the pages of the COMP 222 web page. You can browse those pages by going to http://<machine>.clear9.rice.edu:<port>/html/, where <machine>

is replaced with the name of the machine you ran the tiny web server on above and `<port>` is replaced with the port number you selected when you ran it.

You can also use the `ab` program to concurrently access the server with an arbitrary number of connections. To do so, after starting the tiny web server in the COMP 222 web page directory, you should run the following command from a **separate** terminal window:

```
UNIX% ab -c 16 -n 1000 http://<machine>.clear9.rice.edu:<port>/html/pptx/12-memhier.pptx
```

This will retrieve the specified file 1000 times using 16 concurrent connections. You can change these parameters if you like, but keep in mind that your `uthreads` library supports a maximum of 64 threads, so you won't be able to serve more than 64 connections at a time.

The output should look something like the following. In particular, observe the difference between the two "Time per request" lines. This shows that there is concurrency between the requests, as each request takes roughly 17ms to complete, but 16 of them operate concurrently, so when you average the time per request, it only appears to take 1ms per request.

```
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking pyrite.clear9.rice.edu (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests


Server Software:        Tiny
Server Hostname:        pyrite.clear9.rice.edu
Server Port:            18111

Document Path:          /html/pptx/12-memhier.pptx
Document Length:        1264162 bytes

Concurrency Level:      16
Time taken for tests:   1.103 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      1264257000 bytes
HTML transferred:       1264162000 bytes
Requests per second:    906.37 [#/sec] (mean)
```

```
Time per request:       17.653 [ms] (mean)
Time per request:       1.103 [ms] (mean, across all concurrent requests)
Transfer rate:          1119029.15 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.2      0       3
Processing:     3   17  10.6     20      57
Waiting:        0    1   1.4      1      13
Total:          3   18  10.6     21      57

Percentage of the requests served within a certain time (ms)
  50%     21
  66%     23
  75%     25
  80%     26
  90%     29
  95%     37
  98%     42
  99%     43
 100%     57 (longest request)
```

Notes

- The definition of `pthread_create` includes an argument `attr` of type `pthread_attr_t`, which is used to override the default attributes of a thread, such as its stack size. However, the `uthread` library does not support this feature. If the `attr` passed to `pthread_create` is not `NULL`, then `pthread_create` fails, returning `ENOTSUP`.
- `pthread_create` can also fail, returning `EAGAIN`, if it is unable to allocate either a `struct uthr` from the free queue or the memory for the thread's stack from `uthr_intern_malloc`. The size of every `uthread`'s stack is 2MB.
- Given the relatively large size of a `uthread`'s stack, only active threads possess stacks. In other words, a `uthread`'s stack is allocated when it first becomes runnable and deallocated when it transitions from the `ZOMBIE` state to the `FREE` state.
- `pthread_detach` and `pthread_join` fail, returning `ESRCH`, if the specified thread is free.
- `pthread_detach` and `pthread_join` fail, returning `EINVAL`, if the specified thread is already detached. `pthread_join` also fails, returning `EINVAL`, if the specified thread already has another thread joining with it.
- `pthread_join` fails, returning `EDEADLK`, if the specified thread is the calling thread or the specified thread has previously called `pthread_join`, specifying the calling thread. In other words, the calling thread is attempting to join itself or two threads are attempting to join each other.
- `pthread_exit` cannot free the calling thread's stack because it is running on that stack. An exiting thread's stack can only be freed after the `uthread` library has resumed a different `ucontext_t`, such as the scheduler or that of a joining thread.

- `sched_yield` should always return 0. Its implementation within the `uthreads` library should never fail and return -1.
- The various queues must be implemented as circular doubly-linked lists as described in COMP 222, so that adding and removing threads are $O(1)$ operations.
- The call to `sigaction` by `uthr_init` to set up the `SIGPROF` handler must specify `SA_RESTART`.
- Even when `SA_RESTART` is specified to `sigaction`, `select` behaves differently from most system calls. It will not automatically restart after a signal is received. Instead, it will return without completing what it was requested to do. Consequently, `uthr_check_blocked` must be prepared for `select` to return -1 and set `errno` to `EAGAIN`. As the `errno` value suggests, the call to `select` should simply be retried.
- Just as you did within functions that manipulated or accessed the jobs array in the Shell assignment (such as `add_job`) you need to block the appropriate signals when working with shared state in the `uthreads` library.
- Any calls to the functions `getcontext`, `makecontext`, `setcontext`, and `swapcontext` within the `uthreads` library must be performed with `SIGPROF` signals blocked, with the exception of calls within `uthr_init`. For example, if a call to `swapcontext` were preempted, and another call to `swapcontext` with the same `ucontext_t` as the first argument were to occur, the results could be catastrophic.

Writeup

Your `uthreads` library implements "preemptive multithreading", as the scheduler runs on a timer and can prempt running threads. Alternatively, running threads are never preempted in "cooperative multithreading". Instead, the scheduler would only run when a thread blocks (on I/O, for example), yields (by calling `sched_yield`, for example), or completes.

In your writeup (in `writeup.txt`), reflect upon how your `uthreads` implementation would change if you were to implement "cooperative multithreading" instead of "preemptive multithreading". Be specific about what would be different in the implementation.

Turning in Your Assignment

To turn in your code and writeup, you ***must*** use `git push` to copy your work to the `github` remote repository. We will *only* look at the last version that you `push`ed before the deadline. As a precaution against accidental loss of your code, we encourage you to `push` periodically. Please note, the *only* files that you need to turn in are `unix.c`, `uthread.c`, and `writeup.txt`. You should not modify or add any other files in your repository. In other words, these are the only three files on which you should ever perform `git add`. **Do not ignore this instruction.**

For grading your submission, we will use the `Makefile` that was originally provided to compile your code. Therefore, your code should not rely on any modifications to the `Makefile` for correct compilation (and you shouldn't be modifying that file anyway, as discussed in the previous paragraph).

As a sanity check, you should use your web browser to visit your assignment repo. Make sure that what you see in the browser is consistent what you think you have pushed.

Evaluation

The project will be graded as follows:

- Style: 15%
- Writeup: 5%
- Correctness: 80%

Your solution will be tested for correctness on a CLEAR9 machine.