Lab 3: Wacky User Threads `1.1.0`

ECE 353: Systems Software
Jonathan Eyolfson
February 13 **Due: March 6, 2023 @ 11:59 PM ET**

In this lab you'll create a small library called *Wacky User Threads* (`wut`) that implements threads in user-space. Your implementation should use the concepts learned during the lectures, along with some new system calls. You'll create library called `libwut` this lab, not an executable. You'll be using Git to submit your work and save your progress.

This lab has a early testing component to make sure you start early and understand what output you should expect. It is worth 5% of the lab and is due **February 27, 2023 @ 11.59 PM ET**.

Lab setup. Ensure you're in the repository (`cd ~/ece353-labs`) directory. Make sure you have the latest skeleton code from us by running: `git pull upstream main`.

This will create a merge, which you should be able to do cleanly. If you don't know how to do this read Pro Git. **Be sure to read chapter 3.1 and 3.2 fully.** This is how software developers coordinate and work together in large projects. For this course, you should always merge to maintain proper history between yourself and the provided repository. **You should never rebase in this course, and in general you should never rebase unless you have a good reason to.** It will be important to keep your code up-to-date during this lab as the test cases may change with your help.

You can finally run: `cd wut` to begin the lab.

Your task. You're going to create a threading library similar to `pthread` in some respects. Unlike `pthreads`, you will be creating user-level cooperative threads. This means a thread will continue to execute until it exits, or yields. You must run your threads in FIFO order (re-queuing them if they yield). You may use all functions in `ucontext.h` to do the heavy lifting of initializing and swapping threads in your context switching. Your version is going to be a C library with the following API:

```
void wut_init();
int wut_create(void (*run)(void));
int wut_id();
int wut_yield();
int wut_cancel(int id);
int wut_join(int id);
void wut_exit(int status);
```

The description of what each function should do is below:

```
void wut_init()
```

This will always be called once before a user makes any other call to your library. You set up the thread executing `wut_init` as thread 0. You should initialize or setup anything else you need here.

```
int wut_create(void (*run)(void));
```

You will create a new thread in this function, that new thread should be setup to start executing the function given by the `run` argument. You should return a unique `wut_id` that your library will use to refer to this created process. The IDs should be sequential and start with `0`, and should always use the lowest ID available.

You should allocate a stack for the new thread, set it's user context using `ucontext_t` (likely using `makecontext`). Each thread should have it's own thread control block (`tcb`) that you design. We've provided you a `new_stack` function that returns a pointer to a new stack of size `SIGSTKSZ`. You **must use** this function, mostly for you own sanity, it registers the stack with `valgrind`, so you don't get a lot of false positives. If you ever need to use a stack size, use `SIGSTKSZ`.

After initializing the thread control block, you should add it to a ready queue in FIFO order. You should not switch to this thread yet.

```
int wut_id()
```

You should return the `id` of the currently executing thread.

```
int wut_yield()
```

This function should yield to the next thread on the ready queue. The thread that called yield should be put at the end of the ready FIFO queue.

This function should return `0` to the caller if it successfully yielded. Otherwise, it should return `-1` to indicate an error. Some errors include: no available threads to switch to.

```
int wut_cancel(int id)
```

This function should cancel a thread specified by `id` and remove it from the ready queue, so it will not execute again. You should also free any memory associated with its stack and ucontext. The cancelled thread's status should be set to `128`, and you should maintain the information in the thread control block.

This function should return `0` to the caller if it successfully cancelled. Otherwise, it should return `-1` to indicate an error. Some errors include: invalid thread to cancel, cannot cancel self.

`int wut_join(int id)`

This function should cause the calling thread to wait on the thread specified by `id` to finish (either by exiting to getting cancelled). The waited upon thread should also free any memory associated with its stack and ucontext. A thread may only be waited on by one other thread. You should detect and report an error if two threads attempt to join on the same thread (only the first thread that calls join should succeed). The calling thread should be removed from the ready queue, and only re-added after the other thread finishes.

This function should return the status of the waited on thread to the caller. Also, the waited on thread should have its thread control block removed and its `id` should be available to new threads. Otherwise, it should return -1 to indicate an error. Some errors include: invalid thread to wait on, cannot wait on self.

1.1.0 Additions: You should only successfully block the calling thread if the thread specified by `id` is running or runnable (it's in the ready queue). You should not successfully join a thread that is waiting on another thread already. In this case, return -1 and continue execution.

If the thread specified by `id` is already terminated, then the thread calling `wut_join` must continue execution and not be re-added to the back of the ready queue. You must of course return the `status` of the terminated thread and clean up all its resources still.

`void wut_exit(int status)`

This causes the current thread to exit and set its status in the thread control block to the value given by `status`. Similar to cancel, this thread should be removed from the ready queue so it will not execute again. You would not want to free the stack at this point (why?).

If this is the final thread in the process, the process should exit with and exit code of `0`.

1.1.0 Addition: Internally the status should only be values between `0` and `255` inclusive. You must only store the lower byte of the `status` argument. If you are unfamiliar with lower level operations in C, you can use: `status &= 0xFF;` and afterwards `status` will be between `0` and `255`. (Note: this is what occurs when you `exit` from a process as well). This means that successful calls to `wut_join` will also return a value between `0` and `255`.

Errors. You need to check for and properly handle errors. For fatal errors, you should exit with the `errno` of the first fatal error. However, your implementation should not generate errors

Early testing (5% of the grade). The test cases assume you have a more-or-less complete working implementation. In order for you to test your code in development, and to suggest changes. You may modify the code in `test/main.c` to call and test your library. When you build your code the executable will be `build/test/wut`.

You should not modify check and code should call check *one or more times*. The purpose of check is to check the value of an integer that you'd like to know from the solution. You'll be provided the output of the check calls shortly after the due date.

You may create however many threads you wish, and do whatever calls you wish. However, your program must terminate, and cannot have an infinite loop. These checks may be turned into test cases and used as part of the suite.

Building. First, make sure you're in the `wut` directory if you're not already. After, run the following commands:

```
meson setup build
meson compile -C build
```

Whenever you make changes, you can run the compile command again. You should only need to run setup once.

Testing. You cannot execute your library directly, however you can run the test programs manually. Please find the files in `tests/*.c`. You should be able to read and understand what they're doing with your library. Find the executables in `build/tests/*`.

You may also choose to run the test suite provided with the command:

```
meson test --print-errorlogs -C build
```

The first 10 tests are arranged in the order you should do them.

Grading. Run the `./grade.py` script in the directory. This will rebuild your program, run the tests, and give you a grade out of 100 based on your test results. Note that these test cases may not be complete, more may be added before the due date, or there may be hidden test cases. These labs are new, so we may need to change.

Tips. You'll want to read the documentation on the `ucontext` family of C functions. Some header files you'll need to use are provided for you in the skeleton code. You may include additional parts of the standard library. It's highly recommended to at least use the following functions:

```
getcontext makecontext swapcontext exit
```

You may also find `sys/queue.h` helpful, especially the `TAILQ` family of functions that implement a useful linked list. There's some other headers and functions you may find useful during development provided for you.

Submission. Simply push your code using `git push origin main` (or simply `git push`) to submit it. *You need to create your own commits to push, you can use as many as you'd like.* You'll need to use the `git add` and `git commit` commands. Push as many commits as you want, your latest commit that modifies the lab files counts as your submission. For submission time we will *only* look at the timestamp on our server. We will never use your commit times (or file access times) as proof of submission, only when you push your code to the course Git server.