The George Washington University

Advanced Operating Systems

Programming Exercise - User-Level Threading

Due date: Monday, February 6th at midnight.
I *will* assign the next part of the project *before* this is due, so please complete this early. I will be unlikely to be very active on Piazza for the two days before the deadline to discourage procrastination.
You *may work in teams with two or three members*, but do not have to.

1 Objective

For this assignment, you will implement a cooperatively scheduled (non-preemptive) user-level threading library called `lwt`, or lightweight thread. For this assignment, you won't have any kernel programming, and hardly any kernel interaction. This is the *first part of this project*, so implement this well. You will add on to your implementation in future assignments.
Fair warning: each subsequent assignment will modify the requirements, which might require changing your current implementations. This will simulate changing requirements for a real software product. Spend time writing your code correctly, cleanly, and focus on rewriting and simplification. I highly suggest you read and follow the COMPOSITE style guide (linked on the webpage) at in the resources section below, especially if you aren't experience with C.

- `lwt_t lwt_create(lwt_fn_t fn, void *data)` where `typedef void *(*lwt_fn_t)(void *)`. As with `pthread_create`, this function calls the passed in `fn` with the argument passed in as `data`. We'll discuss the return value of `fn` in `lwt_die`. This function returns `lwt_t` which you can define in any way you please. I would highly recommend against making `lwt_t` a struct (though a pointer to a struct is fine). (Self-question: why?) The function that executes in a new lwt returns a `void *`. This leads to the next function.
- `void *lwt_join(lwt_t)` is the equivalent of `pthread_join`. It blocks waiting for the referenced lwt to terminate, and returns the `void *` that the thread itself returned from its `lwt_fn_t`, or that it passed to `lwt_die`. Note that, unlike with `pthreads`, *any* thread can `join` another. It is not only parents that can join on children.
- `void lwt_die(void *)` will attempt to kill the current thread. Note that this is the only way to terminate a thread. The argument passed into `lwt_die` will eventually be returned from `lwt_join`, when it is executed with this lwt as its argument. Note that `lwt_fn_t` returns a `void *` as well. When a thread returns from it's function, the library must treat it as if it called `lwt_die` (i.e. it must behave identically), passing the return value from that function as the argument to `lwt_die`. Note that this is, again, similar to the `pthread` interface.

Note that you *must* free the memory associated with this lwt at the appropriate times. One thing to contemplate is this: when do you free the stack of the lwt that calls `lwt_die`? You *cannot* free this memory within the thread calling die as it is executing on the stack. When do you free the TCB for the thread? This function will only return if the current thread has not joined all of the threads it has created. This is a little awkward, but enables a thread to catch some errors. For example, it might be common for a thread to do `lwt_die(ret); assert(0);` to report any errors due to not joining when it is required.

- `int lwt_yield(lwt_t)` will yield the currently executing lwt, and possibly switch to another lwt (as determined by the scheduling policy). The function's argument is either a reference to a lwt in the system, or a special "no lwt" value (the equivalent to `NULL`) called `LWT_NULL`. If the argument to this function is *not* `LWT_NULL`, then instead of allowing the scheduler to decide the next thread to execute, the library instead attempts to switch directly to the specified lwt. This functionality is commonly used to implement synchronous IPC to direct switches between communicating threads, and is called a "directed yield".
- `lwt_t lwt_current(void)` returns the currently active lwt (that is calling this function). It is highly suggested that you keep a global variable tracking the currently active threa.d
- `int lwt_id(lwt_t)` returns the unique identifier for the thread. I suggest you simply use a monotonically increasing global counter to keep track of the next thread id to assign. You can ignore integer overflow of such a counter for this assignment.
- `int lwt_info(lwt_info_t t)` is simply a debugging helper. `lwt_info_t` is an `enum` including `LWT_INFO_NTHD_RUNNABLE`, `LWT_INFO_NTHD_BLOCKED`, `LWT_INFO_NTHD_ZOMBIES`. Depending on which of these is passed in, `lwt_info` returns the number of threads that are either `runnable`, `blocked` (i.e. that are joining, on a lwt that hasn't died), or that have `died, but not been joined`, respectively.

Your implementation must target x86-32 processors. If you don't have an x86-32 machine handy, you can use the a virtual machine in VMWare, or QEmu.

The code to switch between threads (i.e. to *dispatch* between threads) is provided for you in the `lwt_dispatch.o` file. You can see the `__lwt_dispatch` function it provides with a prototype defined in `lwt_dispatch.h`. For the context switch code, I challenge you to provide your own (you'll need to write inline assembly code), however you cannot use `setcontext`, `setjmp`, any function in those families, nor signals. To implement thread creation, you'll have to also implement a trampoline function – a simple function that is where a newly created thread begins execution whose job is to call the new `lwt` thread's function, passing the data. You'll implement the following API.

I strongly suggest that you implement at least these four utility functions:

- `void __lwt_schedule(void)` which will possibly switch to another thread based on its scheduling

policy. `lwt_yield`, when taking no arguments, is a small wrapper around this function.

- `void __lwt_dispatch(lwt_t next, lwt_t current)` which implements the inline assembly for the context switch from the current thread to the next. The scheduling function uses this to dispatch to the thread chosen to execute next. (This function is provided for you.)
- `__lwt_trampoline` is the execution point where execution for a new thread begins. Make sure to set up the threads stack, and set up the registers appropriately so that when this function begins execution, it is successful. This function must essentially call the function passed to `lwt_create` (of type `lwt_fn_t`), and pass it the data also passed to `lwt_create`. You are not allowed to pass the function pointer and data via global variables. This function will also be returned to after a lwt function returns, and must `lwt_die` accordingly (passing to `lwt_die` the return value from the `fn`). One question is how `__lwt_start` can know the function and argument to start the thread. The easiest mechanism is to simply store the function and data argument in the TCB, and provide means to find that TCB (i.e. see `lwt_current`).
- `void *__lwt_stack_get(void)` which simply allocates a new stack for a new lwt. If you want the `lwt_create` function to be fast, you will have to implement a pool of threads so that previously dead threads can be reused quickly for new `lwt_create` calls.
- `void __lwt_stack_return(void *stk)` which simply recovers the memory for the lwt thread's stack. If you are pooling the lwt's stacks, then they can be pooled here.

Note that it is not uncommon for functions that are not meant to be used directly by "clients" to be prefixed with __ to indicate that they are not an external API.

Scheduling. The scheduling rules are relatively simple. Your scheduling policy can be very simple for the time being: "round robin" – this policy is FIFO. A number of rules:

- When a thread yields, it is moved to the end of the list of lwts so it is scheduled last.
- If it is yielded to, it is moved to the head of the list.
- When a thread is created, it is placed at the end of the runqueue.
- When a thread dies, it is removed from the runqueue.
- As discussed above, a yield will result in the thread calling yield being placed at the end of the runqueue.

Please use your discretion what should be implemented in a `*.c` file, and what should be in the header `*.h`. If you don't understand the use of `inline` or `static`, then you should probably read the Composite style guide linked in the resource section below.

1.1 What to Expect

Debugging this implementation will be difficult if not impossible at parts. Given this, you need to figure out ways to implement small pieces, get them working and tested, and build on top of them.

I'd imagine you can implement these functions in around 50 lines of code, so if you start producing a lot of code, you're doing something wrong. Attempt to avoid complexity at all turns.

2 Testing

You are expected to provide your own test regime that tests all edge cases in your implementation. There are a lot of edge cases. A not even close to complete list includes:

- For example, interactions between die and join are complicated. Does the dying thread call `lwt_die` before or after the join?
- Does your round-robin scheduling order threads correctly?
- Do nested thread creation and join work (i.e. can children create children, and join correctly)?
- Do you detect the error when a thread attempts to join itself?
- ...

You're responsible for writing the test cases for all of these cases. I'll provide a testing file a day or two before the deadline, which isn't enough time to test and debug, so you must test rather completely with your own implementation.

3 Sample Implementation Plan

I would strongly suggest you consider following the following implementation plan:

- Read the resources below.
- Implement thread creation. Create a thread stack. I would personally use that stack to pass any arguments you require (i.e. `fn` and `data`). Set the stack pointer for the new thread to the proper location in the stack, and use the context switch code to switch to it.
- Figure out how and when you can reclaim threads, and implement join.
- The rest is much simpler from there.

Throughout all of these steps, make sure that you clean up the mess you've made in previous steps, leaving only clean, understandable code. If you ask me for help, and have a ton of unintelligible code, it is unlikely I'll be able to help. Keep it as simple as possible. Add features after it all works.

4 Resources

- For help with inline assembly for your dispatching function (if you're writing your own), see "Brennan's Guide to Inline Assembly" at `http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html`, especially the section on extended inline assembly, and the examples. There are also other, more up to date, tutorials.

- For help with creating a thread, setting up its stack correctly, and writing the trampoline function, first, you want to understand how C and assembly interface at the function-level. For this, you'll want to understand the `cdecl` calling conventions (see section on `cdecl` at `http://en.wikipedia.org/wiki/X86_calling_conventions`). Also, compile some sample C code with function calls with debugging on (e.g. using the `-ggdb3` flag), and look at the assembly output using `objdump -Srhtl sample.o`.
- The initial thread that executes in `main` must be associated with a `lwt`. You might want to look into `gcc` support for constructors to execute some code *before* `main` to set up the necessary `lwt` structures for this default thread associated with the initial Linux thread. See `https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html` (search for "constructor") and a google search for "gcc constructor" will yield a lot of examples. You are allowed to seek out code on the Internet for constructors.

5 Next Steps

Subsequent assignments will add the following to this implementation:

- Synchronous IPC between threads
- Asynchronous IPC between threads
- M-to-N threading with multiple kernel threads
- ...