Deliverables

You should deliver a **compressed file** containing all the files originally handed in the skeleton code, completed according to the specifications below.

Pair Programming

You are required to work with your assigned partner on this assignment. You must observe the pair programming guidelines outlined in the course syllabus — failure to do so will be considered a violation of the Davidson Honor Code. *Collaboration across teams is prohibited and at no point should you be in possession of any work that was completed by a person other than you or your partner.*

1 Introduction

In this project, you will implement a user-level thread package in C, using the `setjmp()` and `longjmp()` library calls to save/restore context. The package includes thread creation/joining, as well as synchronization primitives (spinlocks and condition variables).

2 Setup

You can implement this homework in any UNIX-like system (including FreeBSD, OpenBSD, Linux, macOS). The easiest programming/debugging setup is using VSCode, but you can use any other preferred method.

3 (60 pts) Scheduler Implementation

In this part, you will implement the thread scheduler in your user-level threading package. Create a file called `threads.c` implementing all functions described in the corresponding `threads.h` header file.

Inspect the `threads.h` header file. You will note that each thread's information is located in a *thread control block* that contains:

1. The thread number.
2. The thread's `jmp_buf` containing its current state.
3. The thread's *initial* stack pointer. The *current* stack pointer is part of the current state, saved in `jmp_buf` every time you call `setjmp()`.
4. The thread's function, arguments to that function, and the return value of that function upon returning from it.
5. The state of the thread, which can be:
   (a) *Active*: considered for being scheduled whenever the active thread either (i) cooperatively calls `thread_yield()` to call your scheduler to switch control to another thread; or (ii) is preemptively interrupted and your scheduler switches control to another thread. *The code in `thread_yield()`, responsible for switching threads, is henceforth called the scheduler.*
   (b) *Blocked*: if your thread is waiting on a condition variable, its state should be changed to blocked. Blocked threads are not considered by your scheduler to be run on either preemptive or cooperative context switches.
   (c) *Finished*: when your thread finishes, you should change the state to finished. *An exiting thread should never clean up its own resources (i.e., entries in its `tcb_t`). It should only mark itself as finished and the thread scheduler will perform any necessary cleanup.*
   (d) *Invalid*: used to mark an empty `tcb_t` entry. All `tcb_t` entries are stored in the array called `thread_context` (which is prototyped in `threads.h` – please find it). If the state of an entry is Invalid, it means that it can be used for a new thread.

The `threads.h` also prescribes a variable called `current_thread_context`. This variable should point to the `tcb_t` of the current thread in execution.

You have complete flexibility to internally design your `threads.c` file, but you **cannot change any definition in `thread.h`.** Here are some specific requirements for your implementation:

1. `thread_init()`. Any application that uses your thread package will call `thread_init()` before using any of the functions. Please see an example in `producer_consumer1.c`. The application's `main()` function should be considered as one of the threads, because you are going to have to switch control between not only the threads you create later

on, but also give a chance for the `main()` method to execute from time to time. Hence, `thread_init()` should initialize the `thread_context` array (making all entries Invalid), and then create a first entry for *the main program*. For that particular entry, make the function, argument, and return value be NULL. It's assumed that the function of the main thread is the `main()` function. The state of this thread should be `STATE_ACTIVE`.

Also, `thread_init()` should setup the signal handlers necessary for thread creation, following the thread creation variant that we discussed in class (see `newthread.c` on Moodle).

Initially, ignore the preemption flag in `thread_init()`. Preemption will be implemented at a later step.

2. `thread_create()`. Whenever the main thread wants to create a new thread, it will call `thread_create()`. Please see an example in `producer_consumer1.c`. This function goes over the `thread_context` array, finds an empty entry and initializes it with the function and arguments provided as parameters. The new thread's state is set to Active. Each of these newly created threads has a separate stack. You should declare an array containing multiple chunks of memory of size 64K, each chunk being the stack for each newly created thread. After initializing the state, you should use the thread creation method that we used in class and effectively create the thread. The function returns the ID of the new thread.

3. `thread_yield()`. **This function is your scheduler.** When a running thread calls `thread_yield()`, you should look for the next available thread for running (i.e., a thread in the Active state), and switch context to it using `setjmp()/longjmp()`. If no other thread besides the current running one is active, you should return 0. Make sure to change the `current_thread_context` variable to the TCB of the new thread when you change context.

4. `thread_exit()`. When a running thread calls `thread_exit()`, its state is marked as Finished, and the `return_value` field of the TCB entry for the current thread is set to the provided returned value. If another thread has been waiting for this exiting thread's termination, its variable `joiner_thread_number` will be set with the ID of the thread that is waiting. In that case, immediately switch context to that (currently blocked) waiting thread, which should be reactivated upon restart (change its status back to Active). If the variable `joiner_thread_number` indicates that there is no joiner, call your scheduler in `thread_yield()` to find another thread to run.

5. `thread_join()`. When a running thread calls `thread_join()` (call it *waiting thread*), it should change its state to Blocked, and immediately switch context to the other thread the joining thread is waiting for (call it *waited thread*). It is assumed that the caller will pass the ID of an active thread as parameter. When the waiting thread restarts, it must be the case that the waited thread switched context to it; hence, mark the state of the now awaken waiting thread to be Active again.

Use `producer_consumer1.c` to test your code so far. That program is a 100% CPU variant of the producer/consumer example that will make use of your threading package.

This producer/consumer variant uses a lock defined in the "synchronization" module to obtain locks.

4 (25 pts) Condition Variables

The problem with the `producer_consumer1.c`, as noted, is that the threads are constantly acquiring/releasing the lock even though there is no element in the queue. Your task now is to implement condition variables in `synchronization.c`. Your implementation of condition variables should make use of the lock implementation in the same file (which you should read and understand!).

Note 1: All functions in this module return 0 **(zero)** on success and 1 **(one)** on failure, following the convention in Pthreads.

Note 2: A doubly-linked list implementation has been provided for you in case you want to use it.

Essentially, a condition variable contains an internal mutex (mutex stands for Mutual Exclusion object – a lock), and a linked list of waiters associated with the condition variable.

- The `thread_cond_init()` function initializes the internal mutex of the condition variable pointed out as parameter.
- The `thread_cond_wait()` function adds the current thread number to the list of waiters, and changes the state of the current thread to Blocked. The manipulation of the list of waiters associated with the condition variable passed as parameter should be done only after you acquire the internal mutex of that condition variable, otherwise multiple threads that call `thread_cond_wait()` could operate at the same time in that list and

corrupt it. After you safely added the current thread number to the list of waiters, call `thread_yield()` to yield control to another thread.

When a thread calls `thread_cond_init()`, it passes as a second argument a `mutex` associated with the first argument `condition_variable`. That argument `mutex` is the lock associated with the condition variable. Mapping to our examples in class, the first parameter could be *queueNotEmpty* while the second parameter is *queueLock*. Per specification of how condition variables work, you should release the `mutex` associated with the condition variable **before** setting the thread state to Blocked[1]. Also, you have to release the lock associated with the condition variable and set the state to Blocked **before** releasing the internal mutex of the condition variable[2].

- The `thread_cond_signal()` function removes the first waiter of the condition variable's waiter's list if there is one. After doing that, you set the corresponding thread's state to Active. All of those operations should be done holding the internal mutex of the condition variable, because no two threads can be operating on a condition variable object at the same time.
- The `thread_cond_broadcast()` function calls `thread_cond_signal` repeatedly for as many waiters are queued in a condition variable.

Use `producer_consumer2.c` to test your code so far. That program is a proper producer/-consumer example that will make use of your threading package.

5 (15 pts) Preemption

Implement preemption in your threading package by setting up a periodic alarm that fires every 100ms, calling your scheduler, `thread_yield()`. Use the `setitimer()` system call to setup this alarm. You also have to register a signal handler for the SIGALRM signal that your program will receive every 100ms.

You have to make sure that threads never call `thread_yield()` when you have preemption enabled. To test your code, comment the `thread_yield()` method in `producer_consumer2.c`, and substitute it with a call to `usleep(100)`. The reason we do this is that we want to increase the likelihood that your SIGALRM signal comes in a time where the producer thread

[1] If you did it after, there would be a window of time where a blocked thread holds a lock. With preemption, if that thread were to be preempted in that window of time, the lock would be held forever.

[2] If you did it after releasing the internal mutex, another thread could signal the condition variable and you could signal an active thread.

does not have the lock. If your SIGALRM arrived constantly when the producer had the lock, switching to the consumer would always cause it to sleep again, making your scheduler switch back to the producer.

Now, fix the problem mentioned in the paragraph above. Deactivate the alarm upon entering `thread_yield()` and reactivate it again after the **new thread** starts running. So, the "old thread" that is switching off deactivates the alarm, and the "new thread" that is starting to run activates it back again. Isn't that cool?

6 Style Deductions

I may deduct up to 15% of every grade item to account for bad style, which includes:

1. Poor indentation;
2. Cryptic variable names;
3. Poor error treatment;
4. Naming style inconsistencies (adopt one style with your partner and stick to it);
5. Overly-complicated code.

Good luck,
- Hammurabi