Project #2:
User Mode Thread Library

Objectives

This project will give you experience with threads and synchronization. You will produce a library that can be used by applications—much like the pthreads API—to create and manage threads in user mode (user threads). Your library will also provide programmers with mutex locks and condition variables for use in synchronizing data access.

1 Thread Management API

Your threading library will implement the API shown in Listing **??**. This API will be defined in a file called **mythreads.h** (I will put the header file in the git repository). You should use this header file, as is. **Do not change it, or we may not be able to test your program**. This API should look familiar as it is similar to the pthreads API that we have used in class (ours is just simpler). The expected functionality of each call is described below:

- **threadInit**—My test applications will call `threadInit` to initialize your library. You can safely assume that this will be the first call into your library that any application will make. This would be a good place to initialize any data structures.
- **threadCreate**—My test applications will call `threadCreate` any time I want to create a new thread. When an application calls this function, a new thread should be created. The new thread should have a stack of size `STACK_SIZE` bytes and should execute the specified function with the argument (`void *`) that was passed to `threadCreate`. If successful, this function should return an `int` that uniquely identifies the new thread, and will be used in calls to the `threadJoin` function. The newly created thread should start running immediately after it is created (before `threadCreate` returns).
- **threadYield**—Calls to `threadYield` cause the currently running thread to "yield" (give up) the processor to the next runnable thread. Your threading library will save the current thread's context and select the next thread

Listing 1: mythreads.h—the API for your threading library

```
#define STACK_SIZE (16*1024)

//the type of function used to run your threads
typedef void *(*thFuncPtr) (void *);
extern void threadInit();

//**Thread Management Functions **
extern int threadCreate(thFuncPtr funcPtr, void *argPtr);
extern void threadYield();
extern void threadJoin(int thread_id, void **result);
extern void threadExit(void *result);

//**Synchronization Types and Functions **
struct mutexlock; //opaque -- implement in your src file
typedef struct mutexlock mutexlock_t;

extern mutexlock_t * lockCreate(void);
extern void lockDestroy(mutexlock_t * lock);
extern void threadLock(mutexlock_t *lock);
extern void threadUnlock(mutexlock_t *lock);

struct condvar; //opaque -- implement in your src file
typedef struct condvar condvar_t;

extern condvar_t * condvarCreate(void);
extern void condvarDestroy(condvar_t * cv);
extern void threadWait(mutexlock_t* lock, condvar_t *cv);
extern void threadSignal(mutexlock_t* lock, condvar_t *cv);

//control atomicity
extern int interruptsAreDisabled; //defined in library
```

for execution. The call to `threadYield` will not return until the thread that called it is selected again to run. We will also call this function in order to preempt your threads during testing.

- **threadJoin**—My test applications will call this function to wait for a thread to complete. This function waits until the thread corresponding to `id` exits (either through a call to `threadExit` or by returning from the thread function). If the `result` argument is not `NULL`, then the thread function's return value (or the return value passed to `threadExit`) is stored at the address pointed to by `result`. Otherwise, the thread's return value is ignored. If the thread specified by `id` has either already exited, or does not exist, then the call should return immediately. You should store the results of all exited threads, at least until `threadJoin` has been called on that thread, so that

the result can be retrieved. Once `threadJoin` has been called on a thread, you may free its results.

- **threadExit**—This function causes the currently-running thread to exit. Calling this function in the main thread will cause the entire process to exit. The argument passed to `threadExit` becomes the thread's return value (equivalent to returning the same value from the thread function), which should be passed to any calls to `threadJoin` by other threads waiting on this thread.

2 Synchronization

Your library will also give programmers tools so they can synchronize their threads and protect shared data accesses. Specifically, your library will provide mutex locks and condition variables. The `mutexlock` and `condvar` structs are opaque to my tests, meaning you can put whatever you into them. Both structs must be defined in your library or the test programs will not compile. All locks must initially be in the unlocked state, and condition variables should start with no threads waiting.

The synchronization API functions are described below:

- **lockCreate**—This function allocates and initializes a new `mutexlock` struct and returns a pointer to it.
- **lockDestroy**—This function frees the memory associated with the specified `mutexlock` struct. This will never be called if a thread holds the lock or is waiting on the lock.
- **threadLock**—This function blocks (waits) until it is able to acquire the specified lock. Your library should allow other threads to continue to execute concurrently while one or more threads are waiting for locks.
- **threadUnlock**—This function releases the specified lock. I will never try to unlock a lock that is not held by the current thread (acquired by a call to `threadLock`).
- **condvarCreate**—This function allocates and initializes a new `condvar` struct and returns a pointer to it.
- **condvarDestroy**—This function frees the memory associated with the specified `condvar` struct. I will never call this function when there are threads waiting on the condition variable.

- **threadWait**—This function atomically unlocks the specified mutex lock and causes the current thread to block (wait) until the specified condition variable is signaled (by a call to `threadSignal`). The waiting thread unblocks only after another thread calls `threadSignal` with the same lock and condition variable. The specified lock must be held when calling `threadWait`, otherwise the behavior is undefined (your library should exit the process and print out an error message if this occurs). Before `threadWait` returns to the calling function, it re-acquires the lock.
- **threadSignal**—This function unblocks a single thread waiting on the specified condition variable. If no threads are waiting on the condition variable, the call has no effect.

You should not do any deadlock detection. If a thread calls `threadLock()` on a lock that it already holds, it should deadlock. Just let it lock up.

3 Preemption

When we test your threading library, control can pass from one thread to the next in one of two ways: 1) the current thread calls `threadYield` and gives up the processor willingly, or 2) the current thread is forced to yield the processor. In testing we will periodically force your threads to yield by calling `threadYield` when a timer fires (we will use `setitimer` to generate "interrupts"). This means that, just like hardware interrupts, `threadYield` can be called at anytime (including during a call to `threadYield`), which requires careful handling.

You are not allowed to use pthread locks, any other pthreads function, or any other locks to provide synchronization. Instead, your library can make operations atomic by temporarily disabling these "interrupts" by setting the `interruptsAreDisabled` variable to a value of 1. This variable is shown in Listing **??**. *Note that it is extern and must be defined in your library.* Setting this variable equal to 1 will disable interrupts and prevent the current running code from being interrupted. Setting this variable back to 0 will enable interrupts. Interrupts should only ever be disabled when executing in one of your library's functions. When the user's code is running, interrupts must be enabled. I strongly recommend using the two functions shown in Listing **??** to enable and disable interrupts. They are simple wrappers, but the assertions will likely help provide some sanity checking, and help you debug common issues.

Hint: you should get your library working in cooperative mode first (without preemption). Things are much much easier to debug in cooperative mode.

Listing 2: Methods for enabling/disabling of interrupts.

```
static void interruptDisable() {
    assert(!interruptsAreDisabled);
    interruptsAreDisabled = 1;
}

static void interruptEnable() {
    assert(interruptsAreDisabled);
    interruptsAreDisabled = 0;
}
```

4 How to compile your library?

After the last project, you have some experience writing a library, but this time your library will be a static library (called "libmythreads.a"), meaning that programs will link to it when they are compiled, and the library will become part of the compiled program. To compile a static library from a set of source files (say, a.c and b.c), you will do the following:

```
$ gcc -g -Wall -c a.c b.c
$ ar -cvrs libmythreads.a a.o b.o
```

Note, you could use `clang` rather than `gcc`, and you are welcome to use additional compile flags if you like.

You can have more source files, and you can, of course, name them whatever you want; however, it is important that your library is named `libmythreads.a`. Also, keep in mind that if a `libmythreads.a` already exists, **ar** will update that file, replacing the files that you specified. It will not recreate the library from scratch. This has caused students some confusion in the past, and I recommend having your makefile delete the old library and recreate it rather than updating it, each time.

When we test your library we will link our test programs with your library. If I wanted to compile a test, implemented in `threadtest1.c`, then I will compile it like so:

```
$ gcc -o executable_name threadtest1.c libmythreads.a
```

Your library must be written in C, and you will need to provide a Makefile that compiles your library. The example above used `gcc`, but you're welcome to use `clang` if you want.

Listing 3: Creating a new execution context.

```
ucontext_t newcontext;  //in your project, this should be on
    the heap, not on the stack.

getcontext(&newcontext); //save the current context

//allocate and initialize a new call stack
newcontext.uc_stack.ss_sp = malloc(STACK_SIZE);
newcontext.uc_stack.ss_size = STACK_SIZE;
newcontext.uc_stack.ss_flags = 0;

//modify the context, so that when activated
//func will be called with the 3 arguments, x, y, and z
//in your code, func will be replaced by whatever function you
    want
//to run in your new thread.
makecontext(&newcontext,
      (void (*)(void))func, 3, x, y, z);
```

5 How to implement threads?

Ok, so how do we do it? Your library will implement threads using the `getcontext`, `makecontext`, and `swapcontext` functions. These functions allow you to get the current execution context, make new execution contexts, and swap the currently-running context with one that was stored.

So, what is a context? Hopefully, you remember when we talked about *context switches*—when one process or thread is interrupted and control is given to another. It's called a context switch because you are changing the current execution context (the registers, the stack, and program counter) for another. We could do this from scratch, using inline assembly, but the `getcontext`, `makecontext`, and`swapcontext` functions make it much simpler.

For example, when `getcontext` is called, it saves the current execution context in a struct of type `ucontext_t`. The man page for `getcontext` describes the elements of this struct. Some of these elements are machine dependent and you don't have to worry about the majority of them. They may include the current state of CPU registers, a signal mask that defines which signals should be responded to, and of course, the call stack. The call stack is the one element of this struct that you will need to pay some attention to.

The steps for creating a new thread are shown in Listing ??. The current context should be saved first using `getcontext` (the new thread's context is based on the saved context). Space for a new stack must be allocated, and the size

recorded. Finally, `makecontext` modifies the saved context, so that when it is activated, it will call a specific function, with the specified arguments. The newly created context is then activated with a call to `setcontext` or `swapcontext`, as shown in Listing **??**. The former (`setcontext`) replaces the current context with a stored context. When successful, a call to `setcontext` does not return (it begins running in the new context). A call to `swapcontext` is similar to `setcontext`, except that it saves the context that was running (a useful thing to do, if you want to resume the old context later). A successful call to `swapcontext` also does not return immediately, but it may return later, when the thread that was saved is swapped back in. **Note that** `swapcontext` **is more useful for this project than** `setcontext`**.**

Be sure to read the man pages for these functions thoroughly. Also, note that the example shown in Listing **??** stores the new context on the current thread's call stack (local variable). In your code, you will want to store your thread contexts on the heap.

Important: On some versions of Linux, the `ucontext_t` struct contains internal pointers (pointers that point to something inside the struct). If you ever try to copy that struct to another location in memory, those pointers won't be updated correctly and your code will probably break. So, please don't copy ucontext_t structs. Store them (using `getcontext` or `swapcontext`) and then leave them where they are. Copying them can produce endless headaches.

For each thread you create, your library should assign that thread to a unique identifier (the number returned by `threadCreate`), in addition to its context. Thread IDs must be unique during the life of each thread; however, you may reuse a thread's ID, if you want, after that thread completes and threadJoin has been called on it. Recording additional information about the thread may be helpful, but is not necessarily required. You will probably also want some sort of data structure to keep track of the threads in the system, so when `threadYield` is called, you can efficiently find the next thread to activate.

Hint: When moving on to preemptive mode, interrupts must be disabled when you swap contexts. Leaving interrupts enabled when calling `swapcontext` will almost certainly invite problems (usually seg faults) in the preemptive tests.

One last hint: Students often wonder how they should tell when a thread completes. One way to handle this is by using the `uc_link` field in the context (see the `man` page for `makecontext`). **This is the hard way, and I do not recommend it.** Instead, think about how you would determine when a function in a sequential program is finished. Note that the function you pass to `makecontext` doesn't have to be the user's thread function—and your job will be much easier, if it is not.

Listing 4: Swapping two contexts.

```
//points to a saved context somewhere in memory
ucontext_t *savedcontext;

//points to the place in memory where
ucontext_t *where_to_save_currently_running_context;

//save the running context, and give the saved context the CPU
swapcontext(where_to_save_currently_running_context, saved
    context);

//note that swapcontext() will not return until the initial
    context is swapped back.
```

Other thoughts, hints, and warnings

Your program should be written in C and use good programming style. It should be `-Wall` clean[1], and most importantly it should be your own. Your library will be tested against a variety of programs that create threads, join threads, and use locks and condition variables. You should write a variety of your own test programs to test your library.

Make sure that your code compiles, runs and has been thoroughly tested on the lab machines. Resist the urge to get fancy. Create a simple implementation that works (meaning you have tested it thoroughly), first.

Remember that your main thread is a bit of a special case. You need to keep track of it, so that you can schedule it along with the rest of the threads, however you don't need to allocate a stack for it (it already has a stack—trying to replace its stack will end badly). You just need to save its context at the appropriate time.

Keep in mind that freeing a thread's resources when it's finished requires some care. Normally, when a thread completes, you need to deallocate (free) the finished thread's stack and any other memory you allocated for it. The trick is that the exiting thread can't free its own stack safely (that's like folding up the ladder you're standing on). Another thread will need to free its resources after it is finished. Also, you can't (and shouldn't try) to free the main thread's resources. Its stack is not allocated on the heap, and so trying to free it will probably not end well.

[1] **"-Wall clean"** = If compiled with the `-Wall` flag, it should not produce any warnings.

Submission Instructions

This project has two deadlines. Absolutely no late assignments will be accepted, and deadline extensions typically require a documented medical emergency or natural disaster.

Compile deadline: 4:00 PM, February 18th — This is a preliminary deliverable worth 10 points. For this, you simply need to submit a valid tgz file (more info below on this) with code and a Makefile that compiles without any errors or warnings and produces a static library, called **libmythreads.a**. This compiled library does not need to work correctly, but it must include all required functions, variables, and types or my tests will not compile with it. For this deliverable, submit to the **compile2** assignment on Canvas.

Functional deadline: 4:00 PM, March 10th — This is the final submission deadline worth the remaining 90 points. For this deliverable, submit to the **project2** assignment on Canvas.

For submission, you should achive your source materials, using the following command:

```
$ tar cvzf project2.tgz README Makefile <list of source files>
```

The **Makefile** should build your library (by running **make**). It should produce a single file (**libmythreads.a**).

The **README** file should include your name, a short description of your project, and any other comments you think are relevant to the grading. It should have two clearly labeled sections titled DESIGN and KNOWN PROBLEMS. The KNOWN PROBLEMS section should include a specific description of all known problems or program limitations. The DESIGN section should include a short description of how you designed your code (especially anything you thought was interesting or clever about your solution). You should also include references to any materials that you referred to while working on the project. Please do not include special instructions for compilation. The compilation and running of the tests will be automated (see details below) and your instructions will not be followed.

Please make sure you include only the source files, not the object files. Before you submit this single file, please check and make sure you have everything in the `project2.tgz` file, using the following script:

```
$ tar xvzf project2.tgz
$ make
```

This should put all of your source files in the current directory, and compile your library.

Submit your `project2.tgz` file via Canvas. You must name your archive **project2.tgz**,

and it must compile and run. We will compile your code using your **Makefile**, and link it with our test programs.

6 Grading

Your project will be graded based on the results of functional testing and the design of your code. We will run several tests to make sure it works properly and correctly handles various error conditions. We will test your programs against a variety of test programs that we have written. Your code should not crash. Your code should not leave interrupts disabled. Your code should not starve threads. If you have only two thread, and one yields, it should always run the other one, if it's runnable. Your locks and condition variables should behave correctly. You will receive 10% credit if your code successfully compiles, and 10% for code style and readability. The rest of your score will be determined by the number of tests that your library passes.

I will give up to 5% extra credit for any libraries that get full credit (pass all functional tests) and are faster than my implementation. **Fast, but buggy libraries will be scored the same as slow, buggy libraries**.

Your source materials should be readable, properly documented and use good coding practices (avoid magic numbers, use appropriately-named identifiers, etc). Your code should be `-Wall` clean (you should not have any warnings during compilation). **Our testing of your code will be thorough. Be sure you test your application thoroughly.**

7 Collaboration

You will work independently on this project. You must *not* discuss the problem or the solution with classmates, and all of your code must be your own code.

You may, of course, discuss the project with me, and you may discuss conceptual issues related to the project that we have already discussed in lecture, via Piazza (do not post any code or implementation details of the algorithms to Piazza or any other web site). **Collaborating with peers on this project, in any other manner will be treated as academic misconduct.**