Project: Thread Scheduling and Synchronization (100 points)

Due date: See Canvas.

1 Logistics

You can work alone, in pair, or in a group of three members. However, if you choose to work alone, you can do so for a 10% bonus (I will add 10% of what you earn to the total points). You can work on CSE or CSCE. However, OS-X does not support the four key methods needed to complete this work. Any clarification and/or revision to the assignment will be posted on canvas.

2 Prerequisite

I have provided three sample programs that should help you get ready to tackle the project. They are available as a zip file (*context_lab.zip*) under the project assignment page on Canvas. These three programs focus on different usage of the following four methods available in most Unix and Linux systems to support user-level context switching between multiple threads:

- `getcontext()`: This method stores the currently active context (various registers on the running CPU) to the structure pointed to by *ucp*.
- `setcontext()`: This method restores the context pointed to by *ucp*. The restored context should have been stored by a call to `getcontext()` or `makecontext()`.
- `makecontext()`: This method modifies the context pointed to by `ucp`, which must be obtained from a call to `getcontext()`. Before invoking this method, the caller must allocate a new stack for this context and assign its address to *ucp->uc_stack* and define a successor context and assign its address to *ucp->uc_link*.
- `swapcontext()`: This method saves the current context in the structure pointed to by *oucp* (current context), and then activates the context pointed to by *ucp* (next context).

We will go over these three programs in class. If you miss the lecture, please make sure that you get the lecture information from your classmates. The use of these four methods can be quite confusing. You can issue "man getcontext" and "man swapcontext" in CSE or CSCE to get more information about these four methods.

3 Implementation Details

There are two parts to this assignment. The completion of the first part will only earn you 70 points (out of 100 points). To be able to earn as many as 100 points, you will need to complete the second part, which involves extending the implementation in the first part to support synchronization operations via semaphores.

Note that while part 1 is worth considerably more than part 2, part 2 is significantly more difficult. Most of you can already do well in this class by completing only part 1 but for those who like to be challenged, part 2 may be just what you need.

3.1 Part I: Building a User-level Thread Scheduler (70 points)

Under the project assignment page on Canvas, I am providing a skeleton file and a Makefile for you to start your project (see Project_handout.zip). The skeleton file can be compiled and run but it doe not perform anything interesting. I've already defined some constants and that you will need. I've also declared some essential variables that you would also need. Furthermore, I have already created a stack space for `myCleanup` context. This space is on the stack of the main thread. I've also created an array of pointers for the user threads that you will create. You will need to use *malloc* to create a stack space for each thread (defined by STACKSIZE which is 8KB).

The integer status array can be used to record the status of each user's thread. The values representing a thread's status are 1 for "ready", 2 for "running", and 0 for "blocked or finished". The status of each thread should be kept in this array to ease the scheduling decision process. (e.g., the first element (`status[0]`) contains to the status of Thread 0. I've also declared the context areas for the main thread, clean up thread, and user threads.

In addition to the `main` method, there are four additional methods in the program. They are declared in *myThread.h* and implemented in *myThread.c*. Also note that you don't need to touch *mysem.h* and *mysem.c* for this part of the assignment.

- `void task1(int tid)`: This task will be threaded and concurrently executed. Half of the created threads will be mapped to this task. It takes thread ID (tid) as its sole argument.
- `void task2(int tid)`: This task will also be threaded and concurrently executed. Half of the created threads will be mapped to this task. It takes thread ID (tid) as its sole argument.
- `void cleanup(void)`: This method takes care of recording and managing status of *task1* or *task2* threads that have finished executing. Its main responsibility is to ensure the correct status of all user threads. When the last user thread is finished, this method would return to the main thread.
- `void signalHandler(int signal)`: This method is the alarm handler. When an alarm signal is received, it suspends the current thread and then schedule the next runnable thread. It should not try to schedule any thread that has already finished its task. Doing so would result in segmentation faults.

Your task is to implement the `cleanup` and `signalHandler` methods and complete the `main` method (e.g., create the first transition from the main thread to the first user thread, free the stack space created for each user thread). You should not need to modify `task1` and `task2` methods. Note that method `makecontext` may warn you about type incompatibility when we try to pass thread id to `task1` and `task2`. This is fine and you can ignore the warning messages by using `-w` compilation flag (already provided in the Makefile).

3.1.1 Hints

It is critical that you have a complete understanding of thread transitions. These transitions include.

1. The initial transition from the main thread to the first user thread.
2. The transition from one user thread to the next.
3. The transition from a finished thread to the cleanup thread.
4. The transition from the cleanup thread back to the main thread.

When these transitions occur correctly, you will be able to terminate the program normally. Otherwise, you may have a case where the program terminates as soon as the first `task2` thread is done. You may also have deadlocks or segmentation faults. Start slowly with the initial goal of having a transition from the main thread to the first user thread. Once this is working, try to switch among all user threads. I'm providing a binary program (*part1sol*) in the *Project_handout* directory that shows the expected output when run for this part of the assignment. Basically, you should see good threads interleaving between `task1` and `task2`. `sharedCounter` should be continuously incrementing or decrementing but the result may be incorrect due to data races. Each thread eventually completes its task. It should show the final value of `sharedCounter` which may be wrong (the correct value is 0 and the program may produce the correct output every once in a while). Finally, the program should terminate normally.

There are also conditional compilation flag, `DEBUG` that you should use to enclose debugging statements. To turn the debug feature on, simply change `#DEFINE DEBUG 0` to `#DEFINE DEBUG 1`. Note that the provided binary program has DEBUG sets to 1 to provide the output. Once your program can produce a similar output, you can decide if you want to tackle the second part, which is described next.

3.2 Building your own Semaphores (30 points)

In the *Project_handout* directory, there are also two additional files that you will need to complete part 2. Note that this part needs the correct implementation of the first part.

In *mysem.h*, I've provided the declaration of `mysem_t` which is your semaphore type and the prototypes of the supporting methods. Because a semaphore needs to be able to block threads, I also declare `signalHandler` as an external method so that you can call it from within these methods. You can implement methods that have been declared in *mysem.h* in *mysem.c*. I've also provided detailed comments to explain each user-defined types, variables, and methods. Below, I describe these semaphore supporting methods.

- `void atomic_swap(volatile long long int * lock)`: As mentioned in class, low-level data races can occur when we move data from memory to a register, update the register, and store the value back to memory. This is because the steps are not *atomic*. Intel provides us with the `xchg` instruction to allow atomic swapping of a value in a memory location and a value in a register. For example, if %r14 contains 0 and variable `lock` contains 1. An `xchg` operation between these two would cause an atomic swap that results in %r14 containing 1 and `lock` containing 0. You can use this operation to ensure that while you are executing in a semaphore related method, no other threads can come in and change the internal states of the same semaphore. Note that this method is provided and working correctly.

- `int mysem_init(mysem_t * mysem, int val)`: This method initializes a semaphore. Note that `lock` field is used by the aforementioned `atomic_swap` to prevent other threads from entering the semaphore methods. Each semaphore also contains a blocking queue (Q), which is an integer array with the same number of elements as the number of threads (defined by constant `THREADS`). It returns 1 when successful and 0 if the provided semaphore's address is invalid. Note that this method is provided and working correctly.

- `void mysem_wait(mysem_t * mysem)`: This method performs the *down* operation on the provided semaphore. It has the following requirements. If $mysem \rightarrow val$ is greater than 0, simply decrements $mysem \rightarrow val$ by 1. If $mysem \rightarrow val$ is equal to 0, suspends the current thread (use `signalHandler`) while keeping the value at 0. Note that there is an assertion statement requiring that $mysem \rightarrow val$ is never less than 0. You must keep the assert statement in your implementation.

- `void mysem_post(mysem_t * mysem)`: This method performs the *up* operation on the provided semaphore. It has the following requirements. If $mysem \rightarrow val$ is equal to 0 and there is at least one thread blocked on the semaphore, release one thread. It then increments $mysem \rightarrow val$ by 1. Note that for this assignment, the semaphore is used as a mutex lock so its value can never exceed 1. As such, there is an assertion statement requiring that $mysem \rightarrow val$ is less than or equal to 1. You must keep the assert statement in your implementation.

- `int mysem_value(mysem_t * mysem)`: This method simply returns $mysem \rightarrow val$. Note that this method is provided and working correctly.

Your task is to implement `mysem_wait` and `mysem_post` methods and interface them correctly with your thread scheduler from part 1. Again, you should not modify `task1`, `task2`, `atomic_swap`, `mysem_init`, and `mysem_value` methods. You cannot remove the provided assert statements. Any scheduling related variables and methods can be called in *mysem.c* by using the `extern` keyword to declare them in *mysem.h*. You can add variables, types, and methods as necessary.

3.2.1 Hints

To test your semaphore implementation, look at the top of *myThread.c* and change `#DEFINE MUTEX NONE` to `#DEFINE MUTEX MYSEM`, and recompile the program using "make" command. Also note that if you want to use the POSIX semaphore library, you can simply change to `#DEFINE MUTEX POSIX`. I'm also providing a binary program (*part2sol*) that produces the expected output for this part. Note that it runs significantly slower than *part1sol* due to synchronization. It also runs significantly slower than the same program that uses a POSIX semaphore.

Any failed assertion may indicate that you have data races within your semaphore related methods. Make sure that these methods are protected by `atomic_swap`. Also to set 1 to $mysem \rightarrow lock$, just use the following statement: $mysem \rightarrow lock$ = 1;

I encourage you to use `DEBUG` conditional compilation flag to enclose debugging statements. To turn the debug feature on, simply change `#DEFINE DEBUG 0` to `#DEFINE DEBUG 1`.

4 Submission

Create a zip file that has all your solutions and submit it through canvas. The step to create proper directory structure is as follows:

1. If you are working as a team, please sign up your team on Canvas. This can simplify grading of team projects significantly.
2. Create a directory called *lastname_project* (replace *lastname* with the submitter's lastname). If you are working with partner(s), only one submission is needed.
3. Place your solutions in the directory. Provide README.txt file.
   - Provide the name of every member on your team.
   - Document the amount of time you spent on each part.
   - Quantify the level of challenge from 0 to 5 (5 is most difficult) for each part.
4. Once the files are properly stored, zip the folder *lastname_project* and submit the zip file through canvas. Note that canvas will only take .zip extension. If you use other means to compress your file, the system will not accept it. You can use "zip -r" command in cse.unl.edu to zip your folder.