Overview

For this homework, you will be working with the "threads"' subsystem of NACHOS. This is the part of NACHOS that supports multiple concurrent activities within the kernel. In the exercises below, you will write some simple programs that create multiple threads, you will demonstrate the problems that arise when multiple threads perform unsynchronized access to shared data, and you will rectify these problems by introducing synchronization (in the form of semaphores) into the code. Then, you will implement the lock and condition synchronization primitives that are missing from NACHOS. Finally, you will simulate two real world synchronization problems by using the synchronization primitives.

All your work for this assignment will take place within the `threads` directory. Although you might find it helpful to *look* at files outside this directory, you should not make any changes to files other than those in the `threads` directory. In general, if a source code file has the following message at the beginning:

```
// DO NOT CHANGE -- part of the machine emulation
```

then you are not to change it. All other files are fair game. Sometimes, you may even wish to add additional source files. This is OK, but make sure to make appropriate changes to Makefile.common.

Important! Read This

In this assignment, and in all subsequent assignments in this course, you will be given very detailed specifications concerning particular functions or classes you are to implement. Though you will generally have considerable flexibility in exactly how you implement these functions, it is *extremely important* that the names of these functions and their behaviors are *exactly as specified*. If an assignment tells you to implement a function or class with a particular name and/or prototype, *do not implement something with a different name or prototype*.

In this course, we are going to perform a great deal of the building and testing of your software automatically. We will write our own driver functions which we will link with your code. If you adhere exactly to the specifications, everything should work properly. If not, your code will require manual intervention to test, and the large number of students in the class will probably make this impossible for us.

One other thing that will make manual intervention necessary is if your code produces extraneous printout when it runs. You should therefore see to it that *your code, as submitted, should produce no printout other than that originally present in Nachos, or that specifically indicated in the assignment*. In addition, if the assignment specifies that you are to produce printout, your printout must appear *exactly as specified*. Do not add extraneous newlines, extra characters, or change in any way the format of the messages you are told to produce.

You will probably find it useful to incorporate debugging printout into your code during development and testing. This is OK, but all such code should use the NACHOS `DEBUG` macro defined in

threads/utility.h, so that debugging printout is not produced unless a suitable command-line flag is provided. Examples of the use of this macro appear throughout the NACHOS code.
Every time you modify a NACHOS file, you should surround all of your changes with preprocessor commands that make it easy to compile the code without your changes.

```
#if defined(CHANGED) && defined(THREADS)
/* put your changed code here */
#else
/* the original code goes here */
#endif
```

Exercise 1: Simple Threads Programming - Weight: 25%

The purpose of this exercise is for you to get some experience using the threads primitives provided by NACHOS, and to demonstrate what happens if concurrently executing threads access shared variables without proper synchronization. Then you will use the semaphore synchronization primitives in NACHOS to achieve proper synchronization.

When the ``threads" version of NACHOS is started, it initially creates a single thread that begins executing the function `ThreadTest()` in the file threadtest.cc. This function creates a new thread that calls the function `SimpleThread(1)`, and the original thread calls the function `SimpleThread(0)`. The two threads each execute the loop in the `SimpleThread()` function, in which they yield control of the CPU back and forth five times.
Modify the function `ThreadTest()` to take a single integer argument `n`, so that its prototype becomes:

```
void
ThreadTest(int n);
```

Your new version of `ThreadTest()` should fork `n` new threads instead of just one. Test your function on values of `n` from zero to four. You will have to change the file `main.cc` to supply the required integer argument to `ThreadTest()`.
Next, modify function `SimpleThread()` to read *exactly* as follows:

```
int SharedVariable;
void SimpleThread(int which) {
 int num, val;
 for(num = 0; num < 5; num++) {
 val = SharedVariable;
 printf("*** thread %d sees value %d\n", which, val);
 currentThread->Yield();
```

```
 SharedVariable = val+1;
 currentThread->Yield();
 }
 val = SharedVariable;
 printf("Thread %d sees final value %d\n", which, val);
}
```

Try the modified version of SimpleThread with the argument to `ThreadTest()` set to 0, 1, 2, 3, and 4. Analyze and explain the results. Put your explanation in your project report.
The file synch.cc contains an implementation of the semaphore operations `Semaphore::P()` and `Semaphore::V()` .
Modify `SimpleThread()` by introducing calls to the semaphore operations, so that accesses to the shared variable are properly synchronized. Try your synchronized version of `SimpleThread()` with the argument to `ThreadTest()` set to 0, 1, 2, 3, and 4. Access to the shared variables are properly synchronized if each iteration of the loop increments the variable by exactly one and each thread sees the same final value. These observations should hold when random interrupts are turned on using the `-rs` option It may be necessary to implement a "barrier" in order to allow all threads to wait for the last to exit the loop.
Using the `#ifdef` / `#endif` preprocessor construct, bracket your synchronization code so that its compilation is controlled by the preprocessor symbol `HW1_SEMAPHORES` . That is, if the preprocessor symbol `HW1_SEMAPHORES` is defined, the synchronization code should be included, otherwise the synchronization code should not be included. Recompile NACHOS both ways to make sure our "conditionalized" code works properly.

Exercise 2: Implementing Locks - Weight: 20%

For this exercise, you are to implement the `Lock` operations that are missing from the file synch.cc.
Locks are used to ensure mutual exclusion between threads. At any time, a lock is either *free*, or else it is *held* by a thread. At most one thread at a time can hold a lock. If a thread tries to acquire a lock that is currently being held by another thread, the requesting thread will *block* (sleep) until the thread that holds the lock *releases* the lock.
There are four missing functions you must implement:

1. `:Lock::Lock(char* debugName)` alizes a lock object.
   The `debugName` argument is a string supplied by the caller, which should just be stored into the new `Lock` structure. Its purpose is simply to help distinguish various instances of locks in debugging printout.
2. `Lock::~Lock()` : This function deallocates a lock object, when it is no longer needed.
3. `void Lock::Acquire()` : This function waits for a lock to become free and then acquires the lock for the current thread.
4. `void Lock::Release()` : This function releases a lock that was previously acquired by the current thread, and wakes up *one* of the threads waiting for the lock.

Use the `Semaphore` code as a guide when writing the `Lock` code. Locks are very much like semaphores with initial value 1.
Test your code by replacing the `Semaphore` operations you added to `SimpleThread()` by corresponding `Lock` operations, and verifying that the demonstration still works properly.
Conditionalize your code so that its complication is controlled by the preprocessor symbol `HW1_LOCKS` . (Be sure that you can still compile your code with `HW1_SEMAPHORES` defined to get the semaphore demonstration.)

Exercise 3: Implementing Conditions - Weight: 25%

For this exercise, you are to implement the `Condition` operations that are missing from the file synch.cc. Conditions are used to ensure proper synchronization among threads. The specifications for this primitive appear in the `synch.h` module. There are five missing functions you must implement:

1. `Condition::Condition(char* debugName)`: initializes a condition object. argument is a string The `debugName` supplied by the caller, which should just be stored into the structure. Its new `Condition` purpose is simply to help distinguish various instances of conditions in debugging printout.
2. `Condition::~Condition()` : This function deallocates a condition object, when it is no longer needed.
3. `void Condition::Wait(Lock* conditionLock)` : This function waits for a condition to become free and then acquires the conditionLock for the current thread.
4. `void Condition::Signal(Lock* conditionLock)` : This function wakes up *one* of the threads that is waiting on the condition.
5. `void Condition::Broadcast(Lock* conditionLock)` : This function wakes up *all* threads that are waiting for the condition.

Use the `Semaphore` and `Lock` code as a guide when writing the `Condition` code.

Exercise 4: Elevator - Weight: 30%

You've been hired by the University to build a controller for the elevator in the ECS bulding (we all know how it can get sometimes!), using semaphores or condition variables. The elevator is represented as a thread; each student or faculty member is also represented by a thread. In addition to the elevator manager, you need to implement the routines called by the arriving student/faculty.
You are supposed to provide the following two functions that can be called from `main` in `main.cc` . *In your submission, please include these calls in main using appropriate preprocessor directives (explained further below) so we can test directly.*

- `Elevator(int numFloors)` . This call starts the elevator thread that serves `numFloor` floors. You can expect that the function is only called once and that the argument is greater than zero. Floors are numbered starting from 1 to the maximum number of floors as defined by numFloors.
- `ArrivingGoingFromTo(int atFloor, int toFloor)` . This should create a student/faculty thread. It wakes up the elevator (when it is not already active), tells it the current floor the person is on,

and waits until the elevator arrives. When the elevator arrives, it tells it which floor to go to. The elevator is amazingly fast, but it is not instantaneous - it takes 50 ticks to go from one floor to the next. This delay can be easily implemented by a `for` loop that counts from 0 to 50 (as integer addition accounts for one tick on the MIPS simulator). We assume that there's only one elevator and that it can fit up to five people at a time. The trivial solution of serving one person at a time and putting others on hold is not acceptable. Simply going up and down taking people to where they want to go and picking people up on the way is sufficient. You can expect that the `atFloor` and `toFloor` numbers are in the interval 1 to `numFloors`.

The following outputs are expected.

- Each person is identified by a unique number.
- When a person issues a request for the elevator, print...
  `Person x wants to go to floor y from floor z.`
- When a person gets into the elevator, print...
  `Person x got into the elevator.`
- When a person gets out of the elevator, print...
  `Person x got out of the elevator.`
- When the elevator reaches a floor, print...
  `Elevator arrives on floor x.`

`x`, `y` and `z` are placeholders for the appropriate integer values. You can put the code for this exercise in `threadtest.cc` or in a separate file. When you put the code into a separate file, make sure that it is included in the Makefile. For testing purposes, you can call the appropriate functions in `main.cc`, but make sure that these calls are surrounded by `HW1_ELEVATOR` preprocessor directives. That is, your elevator simulation should be only started (with parameters of your choice) when `HW1_ELEVATOR` is defined.

To start with you can use the following data structures to define the student/faculty and elevator threads.

```
struct PersonThread {
   int id;
   int atFloor;
   int toFloor;
}
struct ElevatorThread {
   int numFloors;
   int currentFloor;
   int numPeopleIn;
}
```

What to Submit

When you submit your assignment, make sure that all preprocessor directives (`HW1_SEMAPHORES`, `HW1_LOCKS`, `HW1_ELEVATOR`) are disabled. That is, your submission should simply output the Nachos default. We will automatically build your submission with different preprocessor directives. In order to do that, it is necessary that you leave the `$(DEFINES)` variable in the compiler invocation in `Makefile.common`. You can add directives to the compilation process as you like, but we will use the `DEFINES` variable to pass our defines to the compiler. Note that you do not need to do anything right away. Only when you substantially change the Makefile(s), it is necessary for you to make sure that the `DEFINES` variable can still be used to pass arguments to the compilation process.

You have to provide a writeup in a file `reports/project1.txt` in which you have to mention which exercises you have been able to complete. And for the ones that are incomplete, what is their current status. The description of status for completed exercises or those that you did not start should not be more than 1 line. However, when you submit code that is not working properly (partial solution), make sure that your documentation is *very* verbose so that the TAs can understand what you have achieved and which parts are missing. This is necessary to receive at least partial credit.

When you just submit something that does not work and give no explanations, expect to receive no credit. Make sure that the writeup file is in the correct code/reports/ directory. Also make sure you have names of both partners written in your report. Please turnin only one code per group.

- Archive and compress the compiled nachosdir (top-level) directory structure (This directory structure should also contain your report, project1.txt). Run the following command:
- tar c nachosdir | gzip > proj1.tgz
- Upload one copy per group of proj1.tgz. Note! The name of the file you will upload **must be** proj1.tgz
- All Done!