Project 4: Synchronization
CSCI 423 Fall 2010

Due: Sunday October 24, 11:59PM

Always follow the submission directions at the end the this document. Following directions is part of your grade. Any variance from the stated directions may result in zero credit for the project.

Students enrolled in CSCI 423 should complete this project with their partner. Students enrolled in CSCI 501 must complete this project independently. You are not permitted to search for, look at, read, or in any way use solutions to this problem that you did not think of on your own. Note that these projects are similar to but different from the ones used in previous semesters and the solutions described in class are different from the techniques used in the full version of Xinu. In general, the techniques discussed in previous versions of this course or found online will be too complicated for you to have developed yourself. If I catch anyone using a solution found online or from any other source he or she will fail this course. Its is not worth the risk. Don’t cheat. You have been warned.

For this project you may only use your book, notes, and general C programming sources. You may discuss solutions to the projects with other students as long the discussion does not include specific code. If I find that discussion between students crosses this line, this privilege may be revoked.

Objectives: After completing this assignment students will:

- understand IPC (semaphores, and atomic instructions)
- be able to implement semaphores using a test and set lock instruction in the Xinu operating system.

Preparation:

First, make a fresh copy of your work thus far.

```
cp -R xinu-proj3 xinu-proj4
```

Untar the new project files on top of this directory:

```
tar xvzf /home/ruth/pub/xinu-proj4.tgz
```
You should now see the new project files in with your old files. Be certain to make clean before compiling for the first time.

Atomic Actions:

In a system with preemption, we must now guard against being interrupted while in the middle of an atomic operation. There are many points in operating system code where an unfortunately timed interrupt could leave the system in an inconsistent state.

For example, in the \texttt{resched()} function, there are several lines of C code (which translate into many more lines of machine code,) between the point where the outgoing running process is put back into the queue of ready processes and the point where an incoming process is dequeued and set to running. If an interrupt were to take place in the midst of this transition, the interrupt handling code might see the system in an inconsistent state, in which there doesn't seem to be a currently running process. For this reason, we consider process rescheduling to be atomic with respect to the rest of the operating system. The easiest way to enforce this is to temporarily disable interrupts during a critical, atomic section of code, and then reenable them when done.

The new file \texttt{system/intr.S} contains functions \texttt{enable()}, \texttt{disable()}, and \texttt{restore()} for manipulating the master bit that enables processor interrupts. From this point forward, these functions may be used to guard atomic sections in the operating system.

Semaphores:

With this assignment you must learn about and understand classic semaphores before modifying them during the remainder of the assignment. An implementation of classic semaphores with waiting queues has been provided for you. Please examine and understand the implementation which can be found across several files including \texttt{include/semaphore.h}, \texttt{system/newsem.c}, \texttt{system/signal.c}, and \texttt{system/wait.c}.

Test and Set Lock in MIPS:

The semaphore code provided disables interrupts in order to perform the required atomic action on the count variable. Disabling all interrupts is an effective but heavy-handed approach for providing mutual exclusion. Multicore systems and complex real-time systems often cannot afford to disable interrupts, and rely more on hardware support for atomic updates.
The MIPS architecture does not provide the traditional test-and-set-lock hardware support for atomic actions (as described in your text). Instead, MIPS uses a Load-Linked and Store-Conditional instructions. A detailed description of these instructions will be provided in class.

Your project is to complete the functions in spinlock.c to provide hardware supported spin locks that use the assembly function in system/testandset.S. In addition, you will need to write the testandset.S file using the load-linked and store-conditional instructions.

Finally, you need to modify the semaphore code’s signal and wait functions to use your spinlocks instead of disabling interrupts.

Submission:

Before submitting your project please type make clean from your compile directory. This removes the compiled files leaving only the source files. This is necessary to save space on the server.

To submit your files ON-TIME use the following command:

    turnin -c csci423 -p proj4 xinu-proj4

To submit your files LATE use the following command:

    turnin -c csci423 -p proj4-late xinu-proj4

Any variance from the following requirements may result in zero credit for the project.

- Your project MUST be submitted using the turnin command on xinu.cs.olemiss.edu.
- You MUST include the entire directory xinu-proj4.
- Your project MUST compile and run on xinu.cs.olemiss.edu using all of the original files with the exception of your testandset.S, spinlock.c, signal.c, and wait.c files.