Nachos Assignment #1: Build a thread system

Adapted from Tom Anderson’s assignment

Disclaimer: Don’t Panic! This may look difficult and/or overwhelming at the beginning. You’ll get a lot of help getting started with this assignment. You will need to put in a lot of effort, but if you do, this assignment is easily doable. Start early, and make sure that you come and see us if you are stuck or need help.

Please play fair: copying from others or from the Internet is cheating and will result in an F in the class.

This is a group project; you can work in groups of 2, or 3. If you work in a group of 3, you have to do two extra problems. Note: All members of the group will be expected to know all parts of the assignment and may be asked about them in the group grading interview.

In this assignment, we give you part of a working thread system; your job is to complete it, and then to use it to solve several synchronization problems. The starting code is in the directory /home/nael/nachos/nachos-proj1/ on the classroom machines. To copy it over to your account type:

cp -r /home/nael/nachos/nachos-proj1/ .
in your top directory. NOTE: the “.” at the end of the cp command line is required. cd (change directory) to nachos-proj1/code/ and type make to compile the code. This should produce an executable called nachos in your directory nachos-proj1/code/threads/ – You can execute it by going to that directory and typing ./nachos

You can also download the linux patch (and instructions to install it) from the class website if you want to do the project on linux.

The first step is to read and understand the partial thread system provided to you. This thread system implements thread fork, thread completion, along with semaphores for synchronization. Run the program ‘nachos’ for a simple test of the code. Trace the execution path manually for the simple test case that is provide. The first deliverable (hand in) is a one to two page code “walk-through” to be submittedTue 3/11; the full project is due Tue 4/1.

When you trace the execution path, it is helpful to keep track of the state of each thread and which procedures are on each thread’s execution stack. You will notice that when one thread calls SWITCH, another thread starts running, and the first thing the new thread does is to return from SWITCH. This will seem cryptic to you at this point, but you will understand threads once you understand why the SWITCH that gets called is different from the SWITCH that returns. The switch implements in assembly language a save of the thread context of the current thread and a restore of the context of the thread you are switching to. (Note: because gdb does not understand threads, you will get bizarre results if you try to trace in gdb across a call to SWITCH.)

I will give you a tour of the code in class. The important files for this assignment are:

main.cc, threadtest.cc — a simple test of the thread routines.

thread.h, thread.cc — thread data structures and thread operations such as thread fork, thread sleep and thread finish.

scheduler.h, scheduler.cc — manages the list of threads that are ready to run.
Properly synchronized code should work no matter what order the scheduler chooses to run the threads on the ready list. In other words, we should be able to put a call to Thread::Yield (causing the scheduler to choose another thread to run) anywhere in your code where interrupts are enabled without changing the correctness of your code. You will be asked to write properly synchronized code in parts of this project, so understanding how to do this is crucial to being able to do the project.

**In the synchronization problems, you are to model each entity (or activity) as a thread.** The threads should use the synchronization mechanisms in a way that will realize the behavior that the synchronization problem asks for. The first synchronization problem (2) will elaborate on this. To test each problem, you should generate multiple threads to simulate the scenario. For example, in the whale problem, a good way to test it is to generate 5 threads each of type male, female and matchmaker, and show that correct behavior happens. For the bridge problem, generate 20 car threads, 10 going in each direction.

To aid you in this, code linked in with Nachos will cause Thread::Yield to be called on your behalf in a repeatable but unpredictable way. Nachos code is repeatable in that if you call it repeatedly with the same arguments, it will do exactly the same thing each time. However, if you invoke “nachos -rs #”, with a different number each time, calls to Thread::Yield will be inserted at different places in the code to simulate unpredictable scheduling.

Make sure to run various test cases against your solutions to these problems (to make sure you have the synchronization right). Use “nachos -rs #” with different seeds. Try to create many threads (if the problem allows it) to test that the operation is correct.

Warning: in the implementation of threads that you are given, each thread is assigned a small, fixed-size execution stack. This may cause bizarre problems (such as segmentation faults at strange lines of code) if you declare large data structures to be automatic variables (e.g., “int buf[1000];”). You will probably not notice this during the semester, but if you do, you may change the size of the stack by modifying the StackSize define in switch.h.
Although the solutions can be written as normal C routines, you will find organizing your code to be easier if you structure your code as C++ classes. Also, there should be no busy-waiting in any of your solutions to this assignment.

Each “problem” will ask you to code a change to the OS or a synchronization problem. The problem may also ask questions. The answers to these should be provided in a README file on your final floppy.

Implement parts 1 to 4 as well as three problems chosen from problems 5–9. If you work in a group of 3, you have to do all problems 5–9.

1. Implement (non-preemptive) priority scheduling. Modify the thread scheduler to always return the highest priority thread. (You will need to create a new constructor for Thread to take another parameter – the priority level of the thread; leave the old constructor alone since we’ll need it for backward compatibility.) You may assume that there are four priority levels, although you can do all the testing with just two priority levels.

Can changing the relative priorities of the producers and consumer threads have any affect on the output? For instance, what happens with two producers and one consumer, when one of the producers is higher priority than the other? What if the two producers are at the same priority, but the consumer is at high priority?

2. Implement locks and condition variables. You may either use semaphores as a building block, or you may use more primitive thread routines (such as Thread::Sleep). You are provided with the public interface to locks and condition variables in synch.h. You need to define the private data and implement the interface. Note that it should not take you very much code to implement either locks or condition variables.

3. Implement Thread::Join in Nachos. Add an argument to the thread constructor that says whether or not a Join will be called on this thread. Your solution should properly delete the thread control block whether or not Join is to be called, and whether or not the forked thread finishes before the Join is called.

4. Implement producer/consumer communication through a bounded buffer, using semaphores. So, the producer should be one nachos thread, and the consumer another. They must coordinate their activities such that the producer does not overrun the buffer, and the consumer waits when the buffer is empty. (You can use the solution from the book that we discussed in class).

The producer places characters from the string "Hello world" into the buffer one character at a time; it must wait if the buffer is full. The consumer pulls characters out of the buffer one at a time and prints them to the screen; it must wait if the buffer is empty. Test your solution with a multi-character buffer and with multiple producers and consumers. Of course, with multiple producers or consumers, the output display will be gobbledygook; the point is to illustrate

(b) Reimplement this problem using locks and condition variables instead of semaphores

5. A mini-bus shuttles between three stops around the university circle going in the same direction. Passengers wait at their stop until the bus arrives. The bus only has seats for 3 passengers. When the bus reaches a stop, the passengers going down leave the bus, and the new passengers
climb on the bus if there is room (assume they leave from different doors). The bus driver waits for all the passengers to climb, before moving to the next station. You may use semaphores, locks and condition locks. Write pseudocode to simulate this bus operation.

6. The local laundromat has just entered the computer age. As each customer enters, he or she puts coins into slots at one of two stations and types in the number of washing machines he/she will need. The stations are connected to a central computer that automatically assigns available machines and outputs tokens that identify the machines to be used. The customer puts laundry into the machines and inserts each token into the machine indicated on the token. When a machine finishes its cycle, it informs the computer that it is available again. The computer maintains an array `available[NMACHINES]` whose elements are non-zero if the corresponding machines are available (NMACHINES is a constant indicating how many machines there are in the laundromat), and a semaphore `nfree` that indicates how many machines are available.

The code to allocate and release machines is as follows:

```c
int allocate() /* Returns index of available machine.*/
{
    int i;

    P(nfree); /* Wait until a machine is available */
    for (i=0; i < NMACHINES; i++)
        if (available[i] != 0) {
            available[i] = 0;
            return i;
        }
}

release(int machine) /* Releases machine */
{
    available[machine] = 1;
    V(nfree);
}
```

The `available` array is initialized to all ones, and `nfree` is initialized to NMACHINES.

(a) It seems that if two people make requests at the two stations at the same time, they will occasionally be assigned the same machine. This has resulted in several brawls in the laundromat, and you have been called in by the owner to fix the problem. Assume that one thread handles each customer station. Explain how the same washing machine can be assigned to two different customers.

(b) Modify the code to eliminate the problem.

(c) Re-write the code to solve the synchronization problem using locks and condition variables instead of semaphores.
7. A particular river crossing is shared by both cannibals and missionaries. A boat is used to cross the river, but it only seats three people, and must always carry a full load. In order to guarantee the safety of the missionaries, you cannot put one missionary and two cannibals in the same boat (because the cannibals would gang up and eat the missionary), but all other combinations are legal. You are to write two procedures: MissionaryArrives and CannibalArrives, called by a missionary or cannibal when it arrives at the river bank. The procedures arrange the arriving missionaries and cannibals into safe boatloads; once the boat is full, one thread calls RowBoat and after the call to RowBoat, the three procedures then return. There should also be no undue waiting: missionaries and cannibals should not wait if there are enough of them for a safe boatload.

8. Simulate one of the BU’s fraternities to pair up men and women as they enter a Friday night mixer. Each man and each woman will be represented by one thread. When the man or woman enters the mixer, its thread will call one of two procedures, man or woman, depending on the sex of the thread. You must write code to implement these procedures. Each procedure takes a single parameter, name, which is just an integer name for the thread. The procedure must wait until there is an available thread of the opposite sex, and must then exchange names with this thread. Each procedure must return the integer name of the thread it paired up with. Men and women may enter the fraternity in any order, and many threads may call the man and woman procedures simultaneously. It doesn’t matter which man is paired up with which woman (BU frats aren’t very choosy), as long as each pair contains one man and one woman and each gets the other’s name. Use semaphores and shared variables to implement the two procedures. Be sure to give initial values for the semaphores and indicate which variables are shared between the threads. There must not be any busy-waiting in your solution.

9. You have been hired by the CS Dept. to write code to help synchronize a professor and his/her students during office hours. The professor, of course, wants to take a nap if no students are around to ask questions; if there are students who want to ask questions, they must synchronize with each other and with the professor so that (i) only one person is speaking at any one time, (ii) each student question is answered by the professor, and (iii) no student asks another question before the professor is done answering the previous one. You are to write four procedures: AnswerStart(), AnswerDone(), QuestionStart(), and QuestionDone(). The professor loops running the code: AnswerStart(); give answer; AnswerDone(). AnswerStart doesn’t return until a question has been asked. Each student loops running the code: QuestionStart(); ask question; QuestionDone(). QuestionStart() does not return until it is the student’s turn to ask a question. Since professors consider it rude for a student not to wait for an answer, QuestionEnd() should not return until the professor has finished answering the question.