Nachos Assignment #1: Build a thread system

Tom Anderson
Computer Science 162
Due date: Tuesday, Sept. 21, 5:00 p.m.

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 first step is to read and understand the partial thread system we have written for you. This thread system implements thread fork, thread completion, along with semaphores for synchronization. Run the program 'nachos' for a simple test of our code. Trace the execution path (by hand) for the simple test case we provide.

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. We realize this comment 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. (Note: because gdb does not understand threads, you will get bizarre results if you try to trace in gdb across a call to SWITCH.)

The files for this assignment are:

main.cc, threadtest.cc — a simple test of our 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.

synch.h, synch.cc — synchronization routines: semaphores, locks, and condition variables.

list.h, list.cc — generic list management (LISP in C++).

synchlist.h, synchlist.cc — synchronized access to lists using locks and condition variables (useful as an example of the use of synchronization primitives).

system.h, system.cc — Nachos startup/shutdown routines.

utility.h, utility.cc — some useful definitions and debugging routines.

switch.h, switch.s — assembly language magic for starting up threads and context switching between them.

interrupt.h, interrupt.cc — manage enabling and disabling interrupts as part of the machine emulation.

timer.h, timer.cc — emulate a clock that periodically causes an interrupt to occur.

stats.h – collect interesting statistics.

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 as part of the later assignments, so understanding how to do this is crucial to being able to do the project.

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.

Make sure to run various test cases against your solutions to these problems; for instance, for part two, create multiple producers and consumers and demonstrate that the output can vary, within certain boundaries.

Warning: in our implementation of threads, 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.

The assignment is items 1, 2, 5, 6 and 7 listed below.

1. 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). We have provided 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.

2. Implement producer/consumer communication through a bounded buffer, using locks and condition variables. (A solution to the bounded buffer problem is described in Silberschatz, Peterson and Galvin, using semaphores.)

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

3. 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:

```
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.

4. Implement an "alarm clock" class. Threads call "Alarm::GoToSleepFor(int howLong)" to go to sleep for a period of time. The alarm clock can be implemented using the hardware Timer device (cf. timer.h). When the timer interrupt goes off, the Timer interrupt handler checks to see if any thread that had been asleep needs to wake up now. There is no requirement that threads start running immediately after waking up; just put them on the ready queue after they have waited for the approximately the right amount of time.

5. You've been hired by the University to build a controller for the elevator in Evans Hall, 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: "ArrivingGoingFromTo(int atFloor, int toFloor)". This should wake up the elevator, tell it which floor the person is on, and wait until the elevator arrives before telling it which floor to go to. The elevator is amazingly fast, but it is not instantaneous – it takes only 100 ticks to go from one floor to the next. You may find it useful to use your solution for part 4 here. For simplicity, you can assume there's only one elevator, and that it holds an arbitrary number of people. Of course, you should give priority to those threads going to or departing from the fifth floor :-).

6. You've just been hired by Mother Nature to help her out with the chemical reaction to form water, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom invokes a procedure *hReady* when it's ready to react, and each O atom invokes a procedure *oReady* when it's ready. For this problem, you are to write the code for *hReady* and *oReady*. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the procedures must call the procedure *makeWater* (which just prints out a debug message that water was made). After the *makeWater* call, two instances of *hReady* and one instance of *oReady* should return. Write the code for *hReady* and *oReady* using either semaphores or locks and condition variables for synchronization. Your solution must avoid starvation and busy-waiting.

7. You have been hired by Caltrans to synchronize traffic over a narrow light-duty bridge on a public highway. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. In this system, each car is represented by one thread, which executes the procedure *OneVehicle* when it arrives at the bridge:

```
OneVehicle(int direc)
{
  ArriveBridge(direc);
  CrossBridge(direc);
  ExitBridge(direc);
}
```

In the code above, *direc* is either 0 or 1; it gives the direction in which the vehicle will cross the bridge.

(a) Write the procedures *ArriveBridge* and *ExitBridge* (the *CrossBridge* procedure should just print out a debug message), using locks and condition variables. *ArriveBridge* must not return until it safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions or bridge collapses). *ExitBridge* is called to indicate that the caller has finished crossing the bridge; *ExitBridge* should take steps to let additional cars cross the bridge. This is a lightly-travelled rural bridge, so you do not need to guarantee fairness or freedom from starvation.

(b) In your solution, if a car arrives while traffic is currently moving in its direction of travel across the bridge, but there is another car already waiting to cross in the opposite direction, will the new arrival cross *before* the car waiting on the other side, *after* the car on the other side, or is it impossible to say? Explain briefly.

8. 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 a fixed, small number of priority levels – for this assignment, you'll only need two 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?

9. You have been hired to simulate one of the Cal fraternities. Your job is to write a computer program 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 C 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 (Cal 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.

10. Implement the synchronization for a "lockup-free" cache, using condition variables. A lockup-free cache is one that can continue to accept requests even while it is waiting for a response from memory (or equivalently, the disk). This is useful, for instance, if the processor can pre-fetch data into the cache before it is needed; this hides memory latency only if it does not interfere with normal cache accesses.

    The behavior of a lockup-free cache can be modelled with threads, where each thread can ask the cache to read or write the data at some physical memory location. For a read, if the data is cached, the data can be immediately returned. If the data is not cached, the cache must (i) kick something out of the cache to clear space (potentially having to write it back to physical memory if it is dirty), (ii) ask memory to fetch the item, and (iii) when the data returns, put it in the cache and return the data to the original caller. The cache stores data in one unit chunks, so a write request need not read the location in before over-writing. While memory is being queried, the cache can accept requests from other threads. Of course, the cache is fixed size, so it is possible (although unlikely) that all items in the cache may have been kicked out by earlier requests.

    You are to implement the routines *CacheRead(addr)* and *CacheWrite(addr, val)*; these routines call *DiskRead(addr)* and *DiskWrite(addr, val) on a cache miss* – you can assume these disk operations are already implemented.

11. You have been hired by the CS Division 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.

12. You have been hired by Greenpeace to help the environment. Because unscrupulous commercial interests have dangerously lowered the whale population, whales are having synchronization problems in finding a mate. The trick is that in order to have children, *three* whales are needed, one male, one female, and one to play matchmaker – literally, to push the other two whales together (I'm not making this up!). Your job is to write the three procedures *Male(), Female(), and Matchmaker()*. Each whale is represented by a separate thread. A male whale calls Male(), which waits until there is a waiting female and matchmaker; similarly, a female whale must wait until a male whale and a matchmaker are present. Once all three are present, all three return.

13. 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.

14. You have been hired by your local bank to solve the *safety deposit box* synchronization problem. In order to open a safety deposit box, you need *two* keys to be inserted simultaneously, one from the customer and one from the bank manager. (For those of you who saw Terminator 2, something like this protected access to the vault containing the Terminator's

CPU chip.) The customer and the bank manager are threads. You are to write two procedures: *CustomerArrives()* and *BankManagerArrives()*. The procedures must delay until both are present; the customer then calls *OpenDepositBox()*. In addition, the bank manager must wait around until the customer finishes, to lock up the bank vault – in other words, the bank manager cannot return from *BankManagerArrives()* until after the customer has returned from *OpenDepositBox()*. The bank manager can then take a coffee break, while the customer goes off to spend the contents of the safety deposit box.