CS 4/53201 # Project #2 **Operating Systems**

Due via email by 11:59pm on Friday 3 November 2000

Introduction

In this assignment, you are given a working thread system, which is part of Nachos; your job is to explore the use of threads, semaphores, and locks, and to use them to solve a couple of synchronization problems. This project will comprise 15% of your final course grade.

Reading the Nachos Source Code

The first step in this project is to prepare for the assigned problems by reading some of the Nachos source code. Most of this code should be familiar to you from Project 1, while only a small amount of code is new. As in Project 1, I suggest that you read the files in the order described below, and as you do so, read the corresponding sections in Thomas Narten's "A Road Map Through Nachos" and Archna Kalra's "Salsa — An Operating Systems Tutorial".

For Project 1, you read through the following code. For this project, you might want to read through it once again, while thinking primarily about how it handles threads.

- **threads/main.cc**, **threads/threadtest.cc** — a simple test of the thread routines.
- **threads/system.h**, **threads/system.cc** — Nachos startup/shutdown routines.
- **threads/thread.h**, **threads/thread.cc** — thread data structures and thread operations such as thread fork, thread sleep and thread finish.
- **threads/scheduler.h**, **threads/scheduler.cc** — manages the list of threads that are ready to run.
- **threads/switch.h**, **threads/switch.s** — assembly language magic for starting up threads and context switching between them. *Don't worry if you don't understand these two files — you are not responsible for understanding them.*
- **threads/list.h**, **threads/list.cc** — generic list management.
- **threads/utility.h**, **threads/utility.cc** — some useful definitions and debugging routines.

After re-familiarizing yourself with the code above, read through the following files to see how Nachos implements semaphores. Note that the structure for locks and condition variables is in place, but the code to implement them is missing. You might also want to review the material in Lectures 13 and 14 describing how semaphores and locks are implemented.

- **threads/synch.h**, **threads/synch.cc** — synchronization routines: semaphores, locks, and condition variables.

Finally, read through the following files to see how Nachos puts semaphores to a practical use.

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

Tracing Through and Debugging Nachos Source Code

The first step in this project is to read and understand the thread system in Nachos, which implements thread fork, thread completion, and semaphores for synchronization. To copy the necessary files and compile Nachos, see the Project 1 instructions on the course home page entitled "How to get an early start." Run the resulting program "**nachos**" in the **thread** directory for a simple test of the code. Trace the execution path (as in the Project 1 assigment) for the simple test case provided. As you do this project, don't forget about the **gdb** debugger and the Nachos *DEBUG* function (particularly the "*t*", "*i*", and "*m*" flags).

Writing Properly Synchronized Code

Part of the goal of this project is to learn how to write code that synchronizes multiple threads, which may be important in later projects since all threads share a common address space in Nachos. Note that properly synchronized code should work no matter what order the scheduler chooses to run the threads on the ready list. In

other words, the TA or I 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.

Identifying Your Changes

So that the TA and I can easily identify which code you have changed or added, surround all changes and additions in your code by comments in the following form:

```
// PROJECT 2 CHANGES START HERE
<your changed code goes here>
// PROJECT 2 CHANGES END HERE
```

Use your own judgment about how much code to surround in a single comment.

The Problems

1. (50 points) In the file **threads/threadtest.cc**, write a function *TestList* with the functionality described below. Modify **threads/main.cc** so that if Nachos is called with the "*–tt*" command line option, *ThreadTest* is called, while if Nachos is called with the "*–tl*" command line option, *TestList* is called.

   Your function *TestList* should fork two threads named *Bee* and *Buzz*. *Bee* should loop 10 times, printing out a single character 'b' each time using a *printf* statement. Similarly, *Buzz* should loop 10 times, printing out a single character 'B' each time using a *printf* statement.

   After writing this code, in a file called **proj2.prob1**, do the following:

   a. Include a sample run to demonstrate that everything you implemented works.

   b. Run Nachos using the "*t*", "*i*", and "*m*" debugging flags, and explain briefly what is happening (don't include the program output as it'll be too long—just explain what's happening).

   c. If you insert calls to *Thread::Yield* inside *Bee* so that it yields after printing each 'b', and add similar calls inside *Buzz*, does the output change? If so, show the new output, and explain why it changes (you might want to use the debugging flags again to figure this out). If not, explain why it doesn't change. (Note — after you add these *Yield*s, comment them out when you email the code to the TA.)

   d. If you did not complete this problem, clearly describe what you have done, what is not working, and how you would go about finishing the problem if you had more time.

2. (20 points) Implement locks and condition variables using the Nachos semaphores and other high-level functions. More specifically, the public interface to locks and condition variables has been provided in **threads/synch.h** (read that file, including the comments). What you need to do is to define any necessary private data and implement the interface in **threads/synch.cc**.

   After writing your code, compare your implementation to the one in **~walker/pub/synch.cc** and **~walker/pub/synch.h**. Note that this part of the assignment is not for credit, it is just suggested that you implement locks and condition variables on your own to test your understanding of them, but if you choose not to, then you can simply copy the working version from **~walker/pub**.

   After implementing locks, modify your solution to problem 1 above so that the loops in *Bee* and *Buzz* are protected by locks (i.e., so that the loops are mutually-exclusive critical sections).

   After writing this code, in a file called **proj2.prob2**, do the following:

   a. Include a sample run to demonstrate that everything you implemented works.

   b. Run Nachos using the "*t*", "*i*", and "*m*" debugging flags, and explain briefly what is happening (don't include the program output as it'll be too long—just explain what's happening).

   c. If you insert calls to *Thread::Yield* inside *Bee* so that it yields after printing each 'b', and add similar calls inside *Buzz*, does the output change? If so, show the new output, and explain why it changes (you might want to use the debugging flags again to figure this out). If not, explain why it doesn't change. (Note — after you add these *Yield*s, comment them out when you email the code to the TA.)

   d. If you did not complete this problem, clearly describe what you have done, what is not working, and how you would go about finishing the problem if you had more time.

3. (30 points) Implement the Dining Philosophers problem using locks and condition variables as described at the end of Lecture 14. In the file **threads/threadtest.cc**, write a function *Philosophers* with this functionality, and then modify **threads/main.cc** so that if Nachos is called with the "*–tp*" command line option, *Philosophers* is called. Note that you may have to add some additional code to that provided in the lecture to initially set some philosophers to a particular state, to produce interesting output, etc.

   After writing this code, in a file called **proj2.prob3**, do the following:

   a. Include a sample run to demonstrate that everything you implemented works.

   b. Add calls to *Thread::Yield* inside any critical sections, and include a sample run to demonstrate that your solution still works. (Note — after you add these *Yield*s, comment them out when you email the code to the TA.)

   c. If you did not complete this problem, clearly describe what you have done, what is not working, and how you would go about finishing the problem if you had more time.

Where to Get Help

Help is available from Prof. Walker and from the TA (Mr. Yihua He):

- For questions on what the assignment is asking, please contact Prof. Walker.
- For questions on Nachos, please contact either the TA or Prof. Walker.
- For help with your code or debugging, please contact the TA.

Our office hours are on the course syllabus, and may be extended if necessary as the project deadline approaches; see the course web page for any announcements of extended office hours.

Also, if there are corrections or amplifications to this project, or if someone asks a question and we feel the answer may be relevant to other people, that information will be posted on the course home page under the project assignment. Thus, you might want to check the course home page periodically until the project due date to avoid getting bogged down in some problem to which a solution has been announced.

Cooperation versus Cheating

See the class syllabus, and contact me if you have any questions. For this project, you are allowed to *study the Nachos source code* with your friends, but you are ***not*** allowed to work with anyone else to actually *solve the problems*, and you are certainly ***not*** allowed to *copy anyone else's solution.*

Submitting Your Project

When you finish, submit the files **proj2.prob1**, **proj2.prob2**, **proj2.prob3**, **main.cc**, **threadtest.cc**, **synch.cc**, **synch.h**, and any other files that your modified to the TA for grading by typing the following commands in the threads directory:

```
shar  proj2.prob1  proj2.prob2  proj2.prob3  main.cc
     threadtest.cc  synch.cc  synch.h  >shar.out          (type all this on one line)
elm  -s  "Project  2  for  Your  Name  Here"  yihe@mcs.kent.edu  <shar.out
rm  shar.out
```

The first line puts your files into a single file called **shar.out**, and the second line emails that file to the TA (replace "Your Name Here" with your own name).

Important warning — once you submit your files, **DON'T TOUCH THEM AGAIN** — if your email didn't reach the TA, or something happens, the TA may need to ask you to resubmit your files. However, before he lets you do so, he will ask you to log on in her presence, and he will check the modification dates on your files to make sure that they haven't been modified after the due date (if they have been, you will be assessed the appropriate late penalties).

The project is due at 11:59pm on Friday 3 November 2000. For a discussion of my late policy, see the course syllabus.