CS 6/73201 # Project #1 **Advanced OS**

Due via email by 11:59pm on Friday 12 March 1999

Introduction

In this assignment, you are given a working thread system, semaphores for synchronization, and some low-level network communication facilities, all of which are part of Nachos. Your job is to implement locks and condition variables, and use the low-level network communication facilities (which need the locks and condition variables) to implement a client-server system. This project will comprise 10% of your final course grade.

Getting Started

To begin, review the material presented in Lecture 10 on Monday 22 February, and then read the Nachos "Overview paper", available on the class web page.

Second, read the file "Project 1 — Getting Started" on the class web page. Decide whether you will use aegis for the project, or some other platform, and copy the necessary files to your account. Compile Nachos to produce an executable program "**nachos**" in the **threads** directory, and make sure it runs and produces the expected output.

Reading the Nachos Source Code

Now start reading the Nachos source code. 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".

To begin, read through the following files. When you compiled Nachos into the **threads** directory, the Makefile turned on the THREAD switch. Notice what code in these files is included when they are compiled using the THREAD switch, and what code is omitted when they are not compiled with the USER_PROGRAM, FILESYS, and NETWORK switch. Notice what command line arguments you can give to Nachos, and what global data structures are created.

- **threads/main.cc**, **threads/threadtest.cc** — a simple test of the thread routines.
- **threads/system.h**, **threads/system.cc** — Nachos startup/shutdown routines.

Then read through the following files, and see how Nachos implements and schedules threads. Study the thread class, its private data, and its public member functions. Study the Scheduler class, and how it dispatches threads. Glance at the code for context switching, but don't read it in detail.

- **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.*

Next, read through the following files to see how Nachos implements semaphores, and how it puts them to a practical use (in the SynchList class). Note that the structure for locks and condition variables is in place, but the code to implement them is missing.

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

Next, skim through the following files, so you will recognize the functions when you encounter them elsewhere. After reading about DEBUG statements, go back through the files above, and see which debugging options may be useful when working with threads.

- **threads/list.h**, **threads/list.cc** — generic list management.
- **threads/utility.h**, **threads/utility.cc** — some useful definitions and debugging routines.

Now that you've gotten an overview of the Nachos operating system, it's time to look at the emulated machine underneath (all of these files are in the **machine** directory). For now, just skim through the files that emulate the machine. *Warning: since these files represent the "simulated" hardware underneath Nachos, you are not allowed to change them — that would be equivalent to modifying the hardware, which an OS designer usually isn't at liberty to do!*

- **machine/machine.h**, **machine/machine.cc** — emulates the part of the machine that executes user programs: main memory, processor registers, etc.
- **machine/mipssim.cc** — emulates the integer instruction set of a MIPS R2/3000 CPU.
- **machine/interrupt.h**, **machine/interrupt.cc** — manage enabling and disabling interrupts as part of the machine emulation.
- **machine/timer.h**, **machine/timer.cc** — emulate a clock that periodically causes an interrupt to occur.
- **machine/network.h**, **machine/network.cc** — emulation of the physical network hardware. The network interface is similar to that of the console, except that the transmission unit is a packet rather than a character. The network provides ordered, unreliable transmission of limited size packets between nodes. All routing issues (how the message gets from node to node) are taken care of by the network.
- **machine/stats.h** — collect interesting statistics.

Finally, one last set of files (in the **network** directory) provides a "Post Office" protocol that allows multiple copies of Nachos to communicate over the network by sending messages between "mailboxes". There is no description of this code in "A Road Map Through Nachos"; however, it is described in Archna Kalra's "Salsa — An Operating Systems Tutorial", and a detailed discussion of the Nachos networking facility (along with some general comments on networking) is given in "Nachos Networking Background" (from Berkeley, also available on the class web page). Read that material, and read through these files.

- **network**/**nettest.cc** — network test routines.
- **network**/**post.h**, **network**/**post.cc** — a post office abstraction, built in software on top of the network. This provides synchronized delivery and receipt of messages to/from specific mailboxes; there may be multiple mailboxes per machine.

Tracing Through and Debugging Nachos Source Code

To trace through code in Nachos, there are three main approaches: (1) using the **gdb** debugger, (2) using *printf*, and (3) using the *DEBUG* function provided by Nachos. The debugger **gdb** usually works, and is often the best alternative, although tracking across a call to *switch* can be confusing. Adding calls to *printf* often works, but sometimes fails since printf does not always flush the stdout buffer as expected.

The final debugging option, which is particularly useful when working with threads, is to use the Nachos *DEBUG* function, which is declared in **threads/utility.h**. The command line options to Nachos are specified in **threads/main.cc** and **threads/system.cc**; if you look at those files you will see that the command line option for debugging is "–d", which should be followed by a flag to tell Nachos which type of debugging messages to print (these flags are defined in **threads/utility.h**). To look at the various debugging statements that are included in the thread system in Nachos, execute the command "*grep DEBUG *h *cc*" in the **threads** directory — as you can see, all of the those debugging statements have the "t" flag. In the **machine** directory, the debugging statements have "i" and "m" flags. Putting all this together, you might want to run Nachos as "*nachos -d t*", "*nachos -d i*", or "*nachos -d ti*" to see what your code is doing while working with threads. If you need more information, add more debugging statements (add your own debugging flag), or use the Nachos *ASSERT* function.

Writing Properly Synchronized Code

In your solution to this project, you may need to write code that synchronizes multiple threads. 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.

Installing and Testing a Version of Nachos That Supports Networking

When you compiled Nachos in the **threads** directory as described in "Project 1 — Getting Started", only part of Nachos — just the thread system and the emulated machine — was compiled. For this project, you will need to compile a larger, more complete version of Nachos. You'll have to copy over some more Nachos files, so before you do so, you might want to clean up your account a bit. First, "*cd*" to the directory containing your **Makefile**

and **threads** directory, and type "*make clean*" to remove all your old **.o** files in the **threads** directory. While you're at it, you might want to go into the **threads** directory and remove any old editor backup files and core files as well.

When you got started, you copied all of the files in the **threads** and **machine** directory from **~walker/pub/nachos–3.4–hp/code** (assuming you used aegis) to your account For this project, you must copy ***all*** of the files in the **walker/pub/nachos–3.4–hp/code** to your account. When you finish, edit the **Makefile** to comment out the 2 lines that compile into the **threads** directory, and uncomment the 2 lines that compile into the **network** directory. Now type "*make*", and you'll get a new executable **nachos** file — but this time, it will be placed in the **network** directory, rather than the **threads** directory.

Some General Comments on Working with Nachos and Networking

The second step in this project is to read and understand the networking system in Nachos. There is a simple test of the code described below, but this test will not work unless you have correctly implemented locks and condition variables as described in Problem 1.

Assuming you have locks and condition variables correctly implemented, if you run two copies of **network/nachos**, each copy (running as a separate a UNIX process) will act as a node in the network, and a simulated network (implemented via UNIX sockets) will provide the communication medium between these nodes. You can test the basic network functionality by running "*nachos –m 0 –o 1*" and "*nachos –m 1 –o 0*" simultaneously (preferably in different windows so you can more easily see what is happening. (Warning — did I mention yet that you need to have locks and condition variables implemented correctly for this test to work???)

The post office abstraction provided by Nachos is more convenient than working with the raw network. You could continue this layering process — at each level, removing one physical constraint and replacing it with an abstraction. Thus, reliable messages can be built on top of an unreliable service, large messages can be built on top of fixed-length messages, etc. For a detailed discussion of the Nachos networking facility, see "Nachos Networking Background", available on the class web page (this file is an 8-page PostScript document).

To test the higher-level abstractions, the low-level Nachos network emulation can be told to randomly drop packets. The Nachos documentation says that you do this by using the command line option "–n #", where the number, between 0 and 1, reflects the likelihood that a packet will be successfully delivered. However, the documentation is wrong; the actual command line option (defined in **threads/system.cc**) is "-l #" (this is a lower-case "L", not a number one). To simplify matters, you may assume that packet delivery is "fail-safe"; packets may be dropped, but if a packet is delivered, its contents have not been corrupted (although in practice, a hardware or software checksum would be needed to detect this kind of error).

Adding New Files to Nachos

If you need to add a new file to Nachos, it's very easy to do so. However, the Nachos **Makefile** structure is pretty complicated, so if may take you a while to figure out how to do this. For example, suppose you want to add the files **network/blat.h** and **network/blat.cc**. To do this, edit the file **Makefile.common**, and update the definitions of NETWORK_H, NETWORK_C, and NETWORK_O. That's all you have to do! Then, the next time you run "*make*", the **Makefile** in the **network** directory will be updated automatically, and your new files will be compiled and lined into Nachos.

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 1 CHANGES START HERE

<your changed code goes here>

// PROJECT 1 CHANGES END HERE
```

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

The Problems

1. (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**. Your code should also use the passed *conditionLock* to ensure that the member functions only have an effect if the thread calling the member function

is also the holder of the *conditionLock*. It should not take you very much code to implement either locks or condition variables, and there are some helpful hints in the section "6.3 Synchronization" in "A Road Map Through Nachos" (available on the class web page).

To test your code, you might want to use the *SynchList* class (defined in **threads/synchlist.h** and **threads/synchlist.cc**), especially since that class is used by the networking code.

After writing and testing your code, you might want to make an appointment with the TA to have him "OK" your code before you move on to question 2.

2. (80 points) Use the Nachos PostOffice to implement a client-server system, using a separate copy of Nachos for each client and for the (single) server. More specifically, write a server that performs a task of your choice, that is structured using multiple threads and the dispatcher-worker model, and that can accommodate an arbitrary number of client requests. Write code to call this server, such that multiple clients can be communicating with the client simultaneously. You can assume that you only need to send small messages to the server — stay within a single network packet — and that the network is reliable. Since Nachos does not support a preemptive CPU scheduler, you should include a large number of calles to *currentThread->yield* throughout your server code so that all threads get a chance to run.

   In a file called **p1.overview**, write an overview of what you implemented and any design decisions that you made, and supply test code and instructions to demonstrate that your implementation works.

   Do a good job on the coding (good style, comments, etc.), and on the test cases and overview — you will definitely be graded on these items as well as the actual code itself!

3. (Extra Credit) Learn about Nachos interrupts and timers, and do as much as possible to recover if the client or server crashes (test this by killing a copy of Nachos) or if a message gets lost (test this by increasing the likelihood that Nachos will drop packets. In **p1.overview**, write a brief overview of what you implemented and any design decisions that you made, and supply test code and instructions to demonstrate that your implementation works.

Where to Get Help

Help is available from Prof. Walker and from Mr. Kun Qiu (who will act as the course TA for programming projects, but ***not*** for helping with homework assignments, exam preparation, etc.):

- For questions on what the assignment is asking, please contact Prof. Walker.
- For questions on Nachos, please contact either Prof. Walker or Mr. Qiu.
- For help with your code or debugging, please contact Mr. Qiu (***not*** Prof. Walker)

Our office hours are on the class web page, and may be extended if necessary as the project deadline approaches; see the class 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 class web page under the project assignment. Thus, you might want to check the class web 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. You are allowed to discuss the problems with your friends, and to study the Nachos internals with your friends, but you are not allowed to write pseudo-code to solve the problems with your friends, and you are certainly not allowed to copy anyone else's code.

Submitting Your Project

When you finish, submit the project overview file and all files that you modified to the TA. Assuming you have the overview file **proj1.overview**, and you've modified the **Makefile**, **threads/synch.cc**, and **network/nettest.cc**, and you've and added the file **network/rpc.cc**, you can submit those files to the TA for grading by typing the following commands in the directory that contains the **Makefile** and **threads** and **network** directories:

```
shar proj1.overview Makefile threads/synch.cc
network/nettest.cc network/rpc.cc >shar.out
```
(type all this on one line)

```
elm -s "Project 1 for Your Name Here" kqiu@mcs.kent.edu
<shar.out
```
(type all this on one line)

```
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 his 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 12 March 1999. For a discussion of my late policy, see the class syllabus. However, you should probably plan on starting early, ending on time, and then spending the weekend resting or working on something else, instead of trying to perfect a late project.