Nachos Project Guide

CPS 110: Introduction to Operating Systems
Spring 1999

Jeff Chase

Duke University
Department of Computer Science

Nachos Project Guide

Nachos Project Guide

1 Preface

This document describes the Nachos programming projects used for CPS 110, the one-semester undergraduate operating systems course at Duke University. It includes a full description of all of the assignments, and some related material useful for orienting students and guiding them through the exercises and around some common pitfalls. It is intended to supplement other materials about Nachos, including A Road Map Through Nachos and Darrell Anderson’s Nachos Resource Page on the CPS 110 course web.

Nachos is an instructional operating system conceived and implemented by Dr. Tom Anderson and his associates at the University of California at Berkeley. I have had the privilege of knowing Tom as a colleague and as a fellow graduate student at the University of Washington in Seattle, where he has now returned as a Professor. This is not the first time that I have found myself trying to improve in some small way on something that he has done, nor do I believe it will be the last.

Almost all of the content in these projects derives directly from the Berkeley projects included with the Nachos distribution, and more recent versions available on the Web. The Berkeley projects were first used at Duke in Fall 1994 by Dr. Thomas Narten, the author of A Road Map Through Nachos. His materials were adapted by Dr. Carla Ellis and her teaching assistants for her offerings of CPS 110 in the 1995-1997 academic years. I have reorganized, modified, and extended her materials based on my experiences teaching the course in the Fall of 1997.

The current CPS 110 projects are generally easier for students than the Berkeley projects. At this time we do not use the assignments pertaining to file system internals or networking, leaving projects in those areas to follow-on courses. We have made an effort to divide the projects into evenly-sized chunks that start early and build functionality regularly through the semester, and to provide enough step-by-step guidance to draw students into Nachos slowly without overwhelming them. The more guidance we offer, the more our students can accomplish with a given amount of effort, leading to a more satisfying experience for everyone.

Note to students. While we have made an effort to simplify these projects for you, most Duke undergraduates find these projects sufficiently difficult to dominate their lives during the one-semester course. A common misconception from earlier semesters is that we are sadistic individuals who enjoy seeing students suffer. Actually, this is not the case. We enjoy seeing students who are proud of what they have accomplished and excited by the power that flows from a relatively small set of simple abstractions in an operating system, even a toy one like Nachos.

Our goal in developing and refining these projects is to minimize the amount of busy work and orientation time for you, while maximizing the learning value of the projects. Even so, the projects demand that you invest a large amount of effort to learn the internals of an operating system that you will never use again in your life. However, we are confident that in doing so you will learn more about all operating systems and software systems in general, and not just Nachos.

Even so, we are committed to continuing to refine these projects. Your responsibility is to do the best job that you can with them, maintain a positive attitude, and take the time to constructively suggest ways we can make life easier and more productive for students in subsequent offerings of the course. You can tell us in person, send e-mail, or fill out the anonymous suggestion form on the course web. We will not lower the bar, but we will do what we can to help you over it.

2 Nachos Project Policies and Mechanisms

In this course we will have a Nachos assignment due every two weeks, starting two weeks after the first day of classes. This section gives an overview of the course procedures and policies for doing the Nachos projects and assigning grades.

The details of each assignment are covered in Section 4. You will find the information in Section 3 valuable for some or all of the assignments. The system call definitions in Section 5 will be important for Labs 4 and 5. By the end of the semester you will have read everything in this document.

2.1 Project Teams

The projects are done in teams of 2-4 students. Form your project groups amongst yourselves before the first assignment is due. You should try to join a group of three or four; groups of two are acceptable but they should be rare, since they cause more work for everybody. You may reorganize your groups at any time during the semester by mutual consent.

From our perspective, the ideal teams would feature an even distribution of the strongest students and the less strong students. If you are not confident of your abilities, try to find somebody good to help pull you through. If you are a strong student, please consider it your personal mission to share your abilities to help make the semester more fun and more productive for everybody. Although projects are graded on a team basis, we have plenty of opportunities to observe what is happening, and to account for team imbalances in the final grading.

If you feel that a team member is not pulling their weight, you should feel free to let us know, using the anonymous comment box if you wish. We may or may not choose to take steps to deal with the problem, but in any case your anonymity will be protected.

2.2 What to Hand In

Please e-mail a 1-2 page writeup for each assignment to the instructor before class on the due date, with the string **nachos project writeup *N*** in the subject line, where *N* is the assignment number. The first line of your e-mail should be a list of the login names of each project member, separated by commas. The second line of your e-mail should be the full pathname of your team's source code directory. The rest of the e-mail should give an overview of your approach to the assignment, and a summary of the status, i.e., what works and what does not. Your writeups should be specific, clear, and terse. It is not necessary to repeat the issues for each assignment, only a summary of your solutions.

You should expect that your source code will be automatically archived on the due date. This allows us to keep a snapshot of the state of your code at the time each assignment is due, and to keep a record of Nachos project work from semester to semester. You should not count on us for backup copies of your source code, but we may be able to help you if you get into trouble with lost data or version control disasters.

2.3 Demo Sessions

Most of our evaluation of your work will occur in demo/grilling sessions for each assignment during a scheduled slot shortly after the due date. Typically you will schedule your demo either with the instructor or one of the teaching assistants. Please try to spread your demos evenly among the instructor and teaching assistants so that we all get to know each other.

We expect that all members of each team will participate equally in the demos. As a general rule, every demo will be attended by every team member, but we will allow a few exceptions if schedules cannot be reconciled. We expect that any subset of team members can carry out a demo; absence of your strongest team member is not a valid excuse to postpone a demo. At the start of each demo the instructor or TA will designate a "primary spokesperson" for the demo. The primary spokesperson will sit at a workstation and lead the conversation about your project. However, any team member should feel free to add comments during the demo, and we may choose to address our questions to any team member.

Some students are nervous about the demo sessions early in the semester. There would seem to be good reason for this, since we will try to uncover bugs in your code, and we will put you on the spot to tell us about your work. If you are nervous at first, you will get used to it and relax after a few weeks. We are friendly people, and our goal is to give everyone credit for the work they have done, while giving the most credit to the teams who do the best work on each assignment. We also look for opportunitities to use the issues raised by the assignment to help you learn more about operating systems and software in general. Most students find that once they start talking they have plenty to say, and that we are happy to reward students who work hard on the assignments. If you do a good job on each assignment, the demos will be fun.

2.4 Grading

Each project is graded on a 100-point scale, with less than half-credit for projects that do not build and/or execute. Your grade will depend partly on your writeup and your explanation of your work during the demo/grilling, during which we will review your source code and ask you to show us that your code works. Your implementation will be graded on completeness, correctness, programming style, thoroughness of testing, and documentation. Bugs that were not uncovered by your testing cost more than bugs you are aware of and tell us about in advance.

You can check your scores for the Nachos projects on the Web. See the TA at your earliest convenience to set up a password. In general, we will release the grades at most a few days after all demos are complete.

2.5 Extensions, Late Work, Partial Credit, and Solutions

Extensions. We have never granted extensions for individual teams on the Nachos projects. In general, if one member of your group gets sick, has a death in the family, spends an extra day on the beach during spring break, has four midterms on the day before the project is due, or just flakes out, then the other team members must pick up the slack. Get started early and schedule your time carefully.

Late work. You are permitted to continue working on each assignment after the due date, and between the due date and the demo. Of course, you must tell us during your demo about any changes you may have made after the due date. Generally we give about half credit for work done after the due date, but we reserve the right to assign any degree of partial credit that seems appropriate.

Availability of Solutions. In previous semesters all but a few groups have produced code that was solid enough to build on for subsequent assignments, and it has rarely been necessary to hand out solutions. We would rather have you expend the effort to fix your own bugs for each assignment, rather than heaving your first attempt and using our canned solution instead. If the take the time to fix up mistakes, we will give you some points back during the demo for the next assignment.

2.6 What Parts of Nachos Should We Modify?

Some students find the nature of the Nachos projects confusing, because there is a significant amount of code already written and it is not always clear what to change or where to make additions. Nachos is designed so that the changes for each assignment are reasonably localized, and we will tell you which areas are important for each assignment. In most cases, you will be adding new code to the existing framework, mostly in new procedures or classes that you define and create. In a few cases you will be extending or “filling in” classes or methods that are already defined. Very rarely will it be necessary to delete or rewrite code that already exists (this does happen in Lab 4), or to add code in areas outside of the main focus of each assignment.

In general, we do not prohibit you from modifying any part of Nachos that you feel is necessary to modify. However, we are telling you now that the simplest and most direct solutions for each assignment do not require you to modify code outside of the primary area of focus for each assignment. Also, under no circumstances should you modify the behavior of the "hardware" as defined by the machine simulation software in the **machine** subdirectory (see Section 3.4). It is acceptable to change *#define* directives that determine the machine or system parameters (e.g., size of physical memory or size of the default stack), but any code that implements the machine itself is strictly off limits. If you are unsure about this, then please ask.

3 Working With Nachos

This section contains general information that will help you understand Nachos and complete the projects. You should browse through this at the beginning of the semester, and then return to each of the subsections as they become relevant during the semester. A great deal of additional information about Nachos is available through the Nachos resource page.

Before you do any Nachos work, you should be familiar with Section 2, which gives an overview of the course procedures for all of the Nachos assignments. Section 4 gives specific instructions for each assignment.

3.1 Installing and Building Nachos

You will develop, test, and demo your code on Solaris SPARC machines (Sun workstations) in the acpub domain. The Nachos distribution we will use this semester is on the acpub machines at **/afs/acpub.duke.edu/project/cps/pkgs/courseware/cps110/nachos3.4**. Other versions of Nachos are lying around, but this is the only one we will use. The Nachos resource page on the course web includes an HTML source code browser and instructions for a full installation.

Install your Nachos copy into a directory of your choice on acpub. Wherever it goes, *please* give us read access to it with the following AFS incantation: **fs setacl mydirectory chase:cps110 rl**. Do this first! The Nachos resource page has links to complete instructions on AFS access control. AFS is flexible enough to allow you to give write access to the members of your group, read access to us, and no access to anyone else. We must have access to your code in order to give you credit for each assignment.

The Nachos code directory includes several subdirectories with source code for different pieces of the Nachos system. The subdirectories include Makefiles that allow you to automatically build the right components for specific assignments using the **gmake** command. The relevant subdirectories are **threads** for Labs 1-3, **userprog** for Labs 4-5, and **vm** for Labs 6-7. If you type **gmake** in one of these directories, it will execute a sequence of commands to compile and link Nachos, yielding an executable program called

nachos in that directory. All of your testing will be done by running these **nachos** executables built from your modified Nachos code.

You should study the Makefiles to understand how dependency identification and recompilation work. The dependency information determines which **.cc** files are rebuilt when a given **.h** file changes. The dependency lines in each subdirectory Makefile (e.g., **nachos/threads/Makefile)** are created automatically using the **make depend** facility. For example, if you type **cd threads; gmake depend**, this will regenerate the dependency information in the **threads/Makefile**. It is extremely important to keep your dependency information up to date.

A few simple guidelines wil help you avoid build problems, which can consume hours of frustrating debugging time tracking bugs that do not really exist. First, always be sure that you run **gmake depend** any time the header file dependencies change (e.g., you introduce a new **.cc** file or include new header files in an existing **.cc** file), or any time the location of the source code changes (e.g., you **cp** or **mv** the source tree from one directory to another). Second, always make sure you are running the right copy of the **nachos** executable, presumably one you just built. If in doubt, change directories to the correct directory, and execute Nachos with **./nachos**.

3.2 Tracing and Debugging Nachos Programs

There are at least three ways to trace execution: (1) add *printf* statements to the code, (2) use the *gdb* debugger or another debugger of your choosing, and (3) insert calls to the *DEBUG* function that Nachos provides.

Many people debug with *printfs* because any idiot can do it, whereas even smart people need to spend a few hours learning to use a debugger. However, investing those few hours will save you many more hours of debugging time that could be better spent watching TV or doing just about anything else. *Printfs* can be useful, but be aware that they do not always work right, because data is not always printed synchronously with the call to *printf*. Rather, *printf* buffers (“saves up”) printed characters in memory, and writes the output only when it has accumulated enough to justify the cost of invoking the *write* system call. If your program crashes while characters are still in the buffer, then you may never see those messages print. If you use *printf*, it is good practice to follow every *printf* with a call to *fflush(stdout)* to avoid this problem.

3.2.1 The *Debug* Primitive

If you want to debug with print statements, the nachos *DEBUG* function (declared in **threads/utility.h**) is your best bet. In fact, the Nachos code is already peppered with calls to the *DEBUG* function. You can see some of them by doing an **fgrep DEBUG *h *cc** in the **threads** subdirectory. These are basically print statements that keep quiet unless you want to hear what they have to say. By default, these statements have no effect at runtime. To see what is happening, you need to invoke **nachos** with a special command-line argument that activates the *DEBUG* statements you want to see.

See **main.cc** for a specification of the flags to the **nachos** command. The relevant one for *DEBUG* is **-d**. The **-d** flag followed by a space and a series of debug flags cause the *DEBUG* statements in nachos with those debug flags to be printed when they are executed. For example, the **t** debug flag activates the *DEBUG* statements in the threads directory. The machine subdirectory has some *DEBUG* statements with the **i** and **m** debug flags. See **threads/utility.h** for a description of the meanings of the current debug flags.

For a quick peek at what's going on, run **nachos -d ti** to activate the *DEBUG* statements in **threads** and **machine**. If you want to know more, add some some more *DEBUG* statements. You are encouraged to sprinkle your code liberally with *DEBUG* statements, and to add new debug flag values of your own.

3.2.2 Miscellaneous Debugging Tips

The *ASSERT* function, also declared in **threads/utility.h**, is extremely useful in debugging, particularly for concurrent code. Use *ASSERT* to indicate that certain conditions should be true at runtime. If the condition is not true (i.e., the expression evaluates to 0), then your program will print a message and crash right there before things get messed up further. *ASSERT* early and often! *ASSERTs* help to document your code as well as exposing bugs early.

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 is because of the way context switches work in Nachos. Because **gdb** and other debuggers are not aware of the Nachos thread library, tracing across a call to *SWITCH* might be confusing sometimes.

Warning: Each Nachos thread is assigned a small, fixed-size execution stack (4K bytes by default). This may cause bizarre problems (such as segmentation faults at strange lines of code) if you declare large data structures (e.g., *int buf[1000]*) to be automatic variables (local variables or procedure arguments). You will probably not notice this during the semester, but if you do, you may change the size of the stack by modifying the *#define* in **threads/thread.h**.

3.2.3 Defining New Command-Line Flags for Nachos

In addition to defining new debug flags as described in Section 3.2.1, it is easy to add your own command-line flags to Nachos. This allows you to initialize the value of a global variable of your choosing from the command line, in order to control the program's behavior at runtime. *Directions for doing this will be posted on the course newsgroup.*

3.3 Controlling the Order of Execution in Nachos

Many bugs in concurrent code are dependent on the order in which threads happen to execute at runtime. Sometimes the program will run fine; other times it will crash out of the starting gate. A program that works one time may fail the next time because the system happened to run the threads in a different order. The exact interleaving may depend on all sorts of factors beyond your control, such as the OS scheduling policies, the exact timing of external events, and the phases of the moon. The Nachos labs require you to write a lot of properly synchronized code, so it is important to understand how to test your code and make sure that it is solid.

3.3.1 Context Switches

On a multiprocessor, the executions of threads running on different processors may be arbitrarily interleaved, and proper synchronization is even more important. In Nachos, which is uniprocessor-based, interleavings are determined by the timing of *context switches* from one thread to another. On a uniprocessor, properly synchronized code should work no matter when and in what order the scheduler chooses to run the threads on the ready list. The best way to find out if your code is “properly synchronized” is to see if it breaks when you run it repeatedly in a way that exhaustively forces all possible interleavings to occur. To experiment with different interleavings, you must somehow control when the executing program makes context switches.

Context switches can be either voluntary or involuntary. *Voluntary* context switches occur when the thread that is running explicitly calls *Thread::Yield* or some other routine to causes the scheduler to switch to another thread. Note that the thread must be running within the Nachos kernel in order to make a voluntary context switch. A thread running in the kernel might initiate a voluntary switch for any of a number of reasons, e.g., perhaps as part of an implementation of some higher level facility, or maybe the programmer was just being nice.

In contrast, *involuntary* context switches occur when the inner Nachos modules (*Machine* and *Thread*) decide to switch to another thread all by themselves. In a real system, this might happen when a timer interrupt signals that the current thread is hogging the CPU. Nachos does involuntary context switches by taking an interrupt from a simulated timer, and calling (you guessed it) *Thread::Yield* when the timer interrupt handler returns.

3.3.2 Voluntary Context Switches with Thread::Yield

One way to test concurrent code is to pepper it with voluntary context switches by explicitly calling *Thread::Yield* at various interesting points in the execution. These voluntary context switches emulate what would happen if the system just happened to do an involuntary context switch via a timer interrupt at that exact point.

Properly synchronized concurrent code should run correctly no matter where the yields happen to occur. At the lowest levels of the system, there is some code that absolutely cannot tolerate an unplanned context switch, e.g., the context switch code itself. This code protects itself by calling a low-level primitive to disable timer interrupts. However, you should be able to put an explicit call to *Thread::Yield* anywhere that interrupts are enabled, without causing your code to fail in any way.

3.3.3 Involuntary Context Switches with the -rs Flag

To aid in testing, Nachos has a facility that causes involuntary context switches to occur in a repeatable but unpredictable way. The **-rs** command line flag causes Nachos to call *Thread::Yield* on your behalf at semi-random times. The exact interleaving of threads in a given nachos program is determined by the value of the “seed” passed to **-rs**. You can force different interleavings to occur by using different seed values, but any behavior you see will be repeated if you run the program again with the same seed value. Using **-rs** with various argument values is an effective way to force different orderings to occur *deterministically.*

In theory, the **-rs** flag causes Nachos to decide whether or not to do a context switch after each and every instruction executes. The truth is that **-rs** won’t help much, if at all, for the first few assignments. The problem is that Nachos only makes these choices for instructions executing on the *simulated* machine, i.e., “user-mode” code in later assignments. In the synchronization assignments, all of the code is executing within the Nachos “kernel”. Nachos may still interrupt kernel-mode threads “randomly” if **-rs** is used, but these interrupts can only occur at well-defined times: as it turns out, they can happen only when the code calls a routine to re-enable interrupts on the simulated machine. Thus **-rs** may change behavior slightly, but many possibly damaging interleavings will unfortunately never be tested with **-rs**. If we suspect that your code has a concurrency race during the demo, we may ask you to run test programs with new strategically placed calls to *Thread::Yield*.

3.4 The Nachos MIPS Simulator

As discussed in class, true support for user programs and a protected kernel requires that we give each of you your own machine. This is because your kernel will need complete control over how memory is managed and how interrupts and exceptions (including system calls) are handled. Since we cannot afford to give you a real machine, we will give you a *simulated* machine that models a MIPS CPU. You will use the simulated MIPS machine in Labs 4-7 to execute test programs, as discussed in Section 3.5. Your **nachos** executable will contain a MIPS simulator that reads the test program executables as data and *interprets* them, simulating their execution on a real MIPS machine booted with your Nachos kernel.

3.4.1 The MIPS CPU Simulator

The simulated MIPS machine is really just a big procedure that is part of the Nachos distribution. This procedure understands the format of MIPS instructions and the expected behavior of those instructions as defined by the MIPS architecture. When the MIPS simulator is executing a ‘‘user program’’ it simulates the behavior of a real MIPS CPU by executing a tight loop, fetching MIPS instructions from a simulated machine memory and ‘‘executing’’ them by transforming the state of the simulated memory and simulated machine registers according to the defined meaning of the instructions in the MIPS architecture specification. The simulated machine's physical memory and registers are data structures in your **nachos** program.

3.4.2 Interactions Between the Kernel and the Machine

Your Nachos kernel can control the simulated machine in the same way that a real kernel controls a real machine. Like a real kernel on a real machine, your kernel can direct the simulated machine to begin executing code in user mode at a specific memory address. The machine will return control to the kernel (by calling the Nachos kernel procedure *ExceptionHandler*) if the user program executes a system call trap instruction, or if an interrupt or other machine exception occurs.

Your Nachos kernel will need to examine and modify the machine state in order to service exceptions and run user programs. Your kernel may also find it useful to examine and modify the *page table* data structure that is used by the simulated machine to translate virtual addresses in the current user program, or to switch the page table in effect, e.g., before switching control to a different user process. All of the machine state — registers, memory, and page tables — are simply arrays in your Nachos kernel address space, accessible through the *Machine* object. See the definitions in **machine/machine.h**.

3.4.3 I/O Devices and Interrupts

The Nachos distribution extends the MIPS CPU simulator to simulate some devices, e.g., disks, a timer, and a console. Nachos maintains a queue of interrupts that are scheduled to occur (e.g., completion of a pending disk operation), and simulates delivery of these interrupts by calling kernel interrupt handler procedures at the appropriate times.

Why does Nachos hang when I enable the console? The current version of Nachos has an annoying ''feature'' that has caused problems for some groups. The Nachos kernel will not shut down if there are pending I/O operations, even if there are no threads or processes ready to run. This is the behavior you would expect, but Nachos simulates a console by using the interrupt queue to repeatedly poll for characters typed on the console — thus there is always a pending I/O operation on the console. This means that if you create a *Console* object as required for the later assignments, then Nachos will never shut down when it is done running your test programs. Instead it will idle just like a real kernel, waiting for console input. Feel free to kill it with **ctrl-C**. It is bad manners to leave idle Nachos processes running, since they chew up a lot of CPU time.

3.5 Creating Test Programs for Nachos Kernels

In later assignments you will need to create test programs to test your Nachos kernel. The test programs for a Nachos kernel are C programs that compile into executables for the MIPS R2000 architecture. These executable programs run on a simulated MIPS machine using the SPIM machine simulator linked with your **nachos** executable, as described in Section 3.4.

Because the user programs are compiled for the MIPS architecture, they will not run directly on the SPARC CPU that you run Nachos on. In fact, since they use Nachos system calls rather than Unix system calls, they cannot even execute correctly on a real MIPS CPU running an operating system such as IRIX or DEC Ultrix. They are built specifically to execute under Nachos. The bizarre nature of these executables introduces some special considerations for building them.

The **Makefile** in the **test** directory takes care of all the details of producing the Nachos user program executables. The user programs are compiled using a **gcc** cross-compiler that runs on Solaris/SPARC but generates code for the MIPS processor. The compiled code is then linked with the MIPS assembly language routines in **start.s**. (Look at these routines and be sure your understand what they do.) Finally, the programs are converted into a MIPS executable file format called NOFF, using the supplied program **coff2noff**.

To run the test programs, you must first build a Nachos kernel that supports user-mode programs. The raw Nachos release has skeletal support for running a single user program at a time; you will extend Nachos to support multiprogramming, virtual memory, and file system calls during the course of the semester. To build a Nachos kernel that can run user programs, edit your Nachos makefile to uncomment the ‘‘cd userprog’’ lines, then run **gmake** to build a new Nachos executable within the **userprog** directory. You may then use this kernel to execute a user program using the **nachos -x** option. The argument to **-x** is the name of the test program executable, produced as described above.

The Nachos distribution includes several sample test programs. For example, look at **test/halt.c**, which simply asks the operating system to shut the ‘‘machine’’ down using the Nachos *Halt* system call. Run the **halt** program with the command **nachos -x halt**, or **nachos -x ../test/halt**. It may be useful to trace the execution of the **halt** program using the debug flag (see Section 3.2). The **test** directory includes other simple user programs to test your kernels in the later assignments. For these exercises we also expect you to extend these tests and add some of your own.

3.5.1 Troubleshooting Test Programs

Some students have difficulty building new test programs. The following guidelines will help you avoid trouble.

1. Don’t screw around with the build process. Place your source code in the **test** directory, and extend the existing **Makefile** to build them. Do not remove any of the files in the distributed version of the **test** directory. In particular, do not remove the file called **script** or any other files used by the build process.
2. Recognize that your Nachos test programs are extremely limited in what they can do. In particular, their only means of interacting with the outside world is to request services from your Nachos kernel. For example, your test programs cannot call *printf*, *malloc* or any other C library routine. If you attempt to

call these routines, your program will fail to link. If you somehow succeed in linking library routines into your executable image, the resulting program will not execute because these routines will use Unix system calls, which are not recognized by your Nachos kernel.

3. The Nachos distribution includes a warning that global variables may not work correctly. It is safest to avoid the use of global variables in your Nachos test programs.

4. In the past, some students have excused their difficulties with Nachos test programs by informing us that they ‘‘do not know C’’. It should come as no surprise that we are not impressed by this excuse. At this point in your career, you have written a large amount of code in C++. We are certain that you are smart enough to adapt quickly to C, which is merely a restricted subset of C++. Just don’t try anything fancy: C does not have classes, operator overloading, streams, *new*, or *delete*. In addition, a C compiler may not permit you to declare items in the middle of a procedure, which is poor programming practice anyway. If you follow these guidelines and use the existing test programs as a starting point, then you should be OK. If you find that you are having problems because you don’t know C, then learn it.

4 Nachos Lab Assignments

This section covers the details of the Nachos assignments. Before starting any assignment you should be familiar with the material in Section 2, which presents the policies and procedures that apply to all of the Nachos assignments. You will find the information in Section 3 valuable for some or all of the assignments. The system call definitions in Section 5 are important for Labs 4 and 5.

4.1 Lab 1: The Trouble with Concurrent Programming

In this assignment, you are to become familiar with Nachos and the code of a working (but incomplete) thread system. In subsequent assignments you will this thread system, but for now you will merely use what is supplied and experience some of the ‘‘joys’’ of concurrent programming.

The first step is to understand how the partial thread system is used within Nachos. In later assignments, all use of the Nachos thread primitives will be internal to your Nachos operating system kernel; in fact, these primitives are quite similar to internal primitives used for managing processes in real operating system kernels. For now, you are using these internal Nachos primitives to run simple concurrent programs at the application level under Unix. If you find this confusing at this point, don’t worry about it.

Build a **nachos** executable using the **gmake** command Run **gmake** (with no arguments) in the **code** directory; the **nachos** executable will be deposited in the **threads** subdirectory. Once you are in the **threads** subdirectory with a **nachos** executable, you can run a simple test of Nachos, by entering the command **nachos**. If you examine **threads/main.cc**, you will see that you are executing the *ThreadTest* function in **threadtest.cc**. Your first goal is to understand the thread primitives used by this program, and to do some experiments to help you understand what happens at runtime. This gives you an opportunity to warm up with some of the tools you will use to debug your Nachos code. To understand the execution path, trace through the code for the simple test case. See the notes in Section 3.2 for some tips on how to do this.

ThreadTest is a simple example of a concurrent program. In this case there are two independent threads of control executing “at the same time” and accessing the same data in a process. The basic goal of this assignment is to give you some experience with the many ways that concurrent code like this can break given a nondeterministic ordering of thread executions at runtime. In your demos, you will show us some of the difficulties caused by specific execution interleavings that could occur at runtime. The goal here is to expose you to some of the pitfalls up front, when you expect them, so that they are less likely to bite you unexpectedly when you think your code is correct later in the semester. To do this, you will modify the code to allow you to “force” specific interleavings to occur deterministically, and show how the program fails as a result of those interleavings. See the notes in Section 3.3 for a more complete discussion of interleavings and some ways of controlling them.

The assignment is to create your own thread test that starts *T* threads accessing a shared data structure, an unsynchronized list used as a sorted priority queue. By *unsynchronized* we mean that *ThreadTest* and the list implementation do not use semaphores, mutexes, interrupt disable, or other synchronization mechanisms that you will learn about later in the semester. The purpose of these mechanisms is to prevent the problems that you will illustrate and experience in this assignment.

Each of your threads thread will generate *N* items with random keys (or to aid debugging, you might control the input sequence with a more carefully selected key order) and insert into the shared list in sorted order of the key. After inserting its *N* items, each thread should remove *N* items from the front of the list. Since the list is sorted, the thread will expect the removed items to come off in sorted order, eventhough they won’t necessarily be the same items that thread put into the list, if $T > 1$. The expected behavior is that each item inserted into the list will be returned by the remove primitive exactly once, that every remove call will return a valid item, and that the list will be empty when the last thread has finished.

Your job is to identify and illustrate all the kinds of incorrect or unexpected behaviors that can occur in this simple scenario. The programming challenge is to build a test program with options and outputs that show us the buggy behaviors. Create a driver file analogous to the file **threadtest.cc** that makes calls on your list class. Make the changes to **threads/main.cc** so that an execution of the **nachos** command causes the function in the new driver file to be executed instead of the function *ThreadTest* in the file **threadtest.cc**. Find as many different behaviors as you can with some explanation of the scenario giving rise to the observed behavior. You should be able to demonstrate each scenario during the demo, using command line flags (Section 3.2.3), without recompiling your test program.

Use the *List* class in **threads/list.h** and **threads/list.cc** as the starting point for your unsynchronized list. Be aware that this class is used by the Nachos thread system itself. This means that you will have to make a copy of it (with a different class name) and leave the original unmodified, or else you will experience bizarre bugs that you do not understand. There will be plenty of time for that later in the semester.

Warning: the Nachos *List* class will return a null pointer if you attempt to remove an item from an empty list. Be careful not to confuse this with a real item whose value happens to be null. In short, don’t ever insert a null item on a Nachos list, and always use *ASSERT* (see Section 3.2) to ensure that a retrieved item is non-null.

4.2 Lab 2: Threads and Synchronization

In this assignment, you will complete the Nachos thread system by adding support for locks (mutexes) and condition variables. We will use these locks and condition variables later to support monitor-like synchronization in other parts of the system. For example, they are used by the *SynchList* class (see **threads/synchlist.cc** and **threads/synchlist.h**) to avoid the concurrency problems illustrated in Section 4.1.

The public interface to mutexes and condition variables is defined in **synch.h**, which includes useful comments on the semantics of the primitives. Your first mission is to define private data for these classes in **synch.h** and implement the interfaces in **synch.cc**. Look at *SynchList* to see how the synchronization primitives for mutexes and condition variables are used. You are to write two different implementations of these synchronization primitives in two different versions of the **synch.h** and **synch.cc** files. You should be able to switch from one version to the other by (at worst) moving these files around and recompiling Nachos.

Here are the specific steps and requirements for Lab 2:

1. Implement your locks and condition variables using the sleep/wakeup primitives (the *Thread::Sleep* and *Scheduler::ReadyToRun* primitives). It will be necessary to disable interrupts temporarily at strategic points, to eliminate the possibility of an ill-timed interrupt or involuntary context switch. In fact, *Thread::Sleep* requires you to disable interrupts before you call it, to avoid the *missed wakeup* problem discussed in class. However, it is an error to hold interrupts disabled when it is not necessary to do so. Disabling interrupts is a blunt instrument and should be avoided unless absolutely necessary.
2. Implement your locks and condition variables using semaphores as the only synchronization primitive. This time it is not necessary to disable interrupts in your code for mutexes or condition variables: the semaphore primitives disable interrupts as necessary to implement the semaphore abstraction, which you now have at your disposal as a sufficient “toehold” for synchronization.

   Warning: this part of the assignment seems easy but it is actually the most difficult. In particular, the first solution that occurs to you is likely to be one of several obvious solutions that turn out to be wrong in subtle ways. For example, it violates the semantics of condition variables to apply a *Signal* to a thread that is not yet waiting on the condition variable at the time the *Signal* call is made.

Your implementations of locks and condition variables should use *ASSERT* checks (see Section 3.2) to enforce any usage constraints necessary for correct behavior. For example, every call to *Signal* and *Wait* passes an associated mutex; what could go wrong if a given condition variable is used with more than one mutex? What will happen if a lock holder attempts to acquire a held lock a second time? What if a thread tries to release a lock that it does not hold? These *ASSERT* checks are worth points on this assignment, and they will save you headaches in later assignments.

You will also need to consider other usage issues. For example, what should your implementation do if the caller tries to delete a mutex or condition variable object while there are threads blocked on it?

You should be able to explain why your implementation is correct (e.g., what will happen if we put a yield between lines X and Y), and to comment on its behavior (fairness, starvation, etc.) under various usage scenarios. Use the *SynchList* class (in **threads/synchlist.h** and **threads/synchlist.cc**) to demonstrate that your code works. As in Lab 1, do this by creating a driver file analogous to the file **threadtest.cc** that makes calls on the *SynchList* class. Make the changes to **threads/main.cc** so that an execution of the **nachos** command causes the function in the new driver file to be executed instead of the function *ThreadTest* in the file **threadtest.cc**.

Warning: *SynchList* differs from *List* in that placing a null item (a zero) on a *SynchList* will trigger an *ASSERT* when the item is removed. Again, make sure your test programs do not place null items on lists.

Warning: The Nachos condition variable interface is ugly in that it passes the associated mutex on every call to *Wait* or *Signal*, rather than just binding the mutex once in the condition variable constructor. This means you must add code to remember the mutex on the first call to *Wait* or *Signal*, so that you can verify correct usage in subsequent calls. But make no mistake: each condition variable is used with exactly one mutex, as stated in **synch.h**. Be sure you understand why this is so important.

Warning: The definition of semaphores does not guarantee that a thread awakened in *V* will get a chance to run before another thread calls *P*. In particular, the Nachos implementation of semaphores does not guarantee this behavior. That is, *V* increments the count and wakes up a blocked thread if the count transitioned from zero to one, but it is the responsibility of the awakened thread to decrement the count again after it wakes up in *P*. If another thread calls *P* first, then it may consume the count that was “meant for” the awakened thread, which will cause the awakened thread to go back to sleep and wait for another *V*.

Note: for debugging, you may use the **-s** debug flag. However, there are no current *DEBUG* statements with the **s** debug flag, so you will need to add some to your code. See Section 3.2.

4.3 Lab 3: Programming with Threads

Your mission in this assignment is to use the Nachos thread system to solve several synchronization problems. The objective is to improve your skill in writing correct concurrent programs and dealing with the now-familiar pitfalls: race conditions, deadlock, starvation, and so on.

You will use the same files as for the previous programming assignments, and introduce one or more files of your own. As you create any new files, you will need to add them in the proper macro definition in the makefile, and update the makefile dependencies. Note that for this assignment, you will be using the synchronization facilities you implemented for the previous assignment. If your implementation of locks and conditions was broken, then you will need to fix it for this assignment.

Your new classes will be based on header files provided in the course directory (subdirectory **aux**). These header files contain initial definitions of the classes, with the signatures of some methods. You should copy these header files into your source pool and extend them. Feel free to add your own methods, definitions, and classes as needed. However, do not modify the interfaces which are already defined.

Problem 1: Multithreaded Table

Implement a thread-safe *Table* class, which stores a collection of untyped object pointers indexed by integers in the range [*0..size-1*]. You may use *Table* in later labs to implement internal operating system tables of processes, threads, memory page frames, open files, etc. *Table* has the following methods, defined in the header file **Table.h** which is provided in the course directory (subdirectory **aux**):

*Table(**int** size) -- Create a table to hold at most **size** entries.*
int** Alloc (**void object) -- Allocate a table slot for **object**, returning **index** of the allocated entry.*
void Get (**int** index) -- Retrieve the object from table slot at **index**, or NULL if not allocated.*
***void** Release (**int** index) -- Free the table slot at **index**.*

You should be careful that your *Table* class is free from deadlock if used correctly.

Problem 2: Bounded Buffer

This is a classical synchronization problem called bounded producer/consumer. Implement a thread-safe *BoundedBuffer* class, based on the definitions in ***/aux/BoundedBuffer.h**.

*BoundedBuffer(**int** maxsize) -- Create a bounded buffer to hold at most **maxsize** bytes.*

void** Read (**void data, **int** size) -- Read **size** bytes from the buffer, blocking as necessary until enough bytes are available to completely satisfy the request. Copy the bytes into memory at address **data**.*

void** Write (**void data, **int** size) -- Write **size** bytes into the buffer, blocking as necessary until enough space is available to completely satisfy the request. Copy the bytes from memory at address **data**.*

BoundedBuffer will be used in Lab 5 to implement pipes, an inter-process communication (IPC) mechanism fundamental to Unix systems. The basic idea is that the pipe or *BoundedBuffer* passes data from a producer thread (which calls *Write*) to a consumer thread (which calls *Read*). The consumer receives the bytes placed in the buffer with *Write*, in the same order as those bytes were written by the producer. If the producer generates data too fast (i.e., the buffer overflows with more than *maxsize* bytes) then *Write* puts the producer to sleep until the consumer can catch up and read some data from the buffer, freeing up space. If the consumer reads data too fast (i.e., the buffer empties), then *Read* puts the consumer to sleep until the producer can catch up and generate some more bytes.

Note that there is no restriction on which threads call *Read* and which call *Write*. Your implementation should not assume that it is used by only two threads, or that the calling threads play fixed roles as producer and consumer. If a given *BoundedBuffer* is used by multiple threads, then you should take care to preserve the atomicity of *Read* and *Write* requests. That is, data written by a given *Write* should never be delivered to a reader interleaved with data from other *Write* operations, even if writers and/or readers are forced to block because the buffer fills up or drains.

Hint: You may find it useful to base your implementation on the Nachos *SynchList* class. Your implementation will also need to manage some dynamically allocated arrays to store the bytes in the buffer.

Problem 3: Alarm Clocks

Implement an *AlarmClock* class. Threads call *Alarm::Pause(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 in **system.cc** should check 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 at least the right amount of time.

Warning: Do not change the behavior of the timer hardware to implement alarms. You may modify the timer interrupt handler, but do not modify the timer class itself.

Warning: Nachos will exit if there are no runnable threads, even if there is a thread waiting for an alarm. You must find a way to prevent that from happening. This is one rare case in which the ‘‘right’’ solution is to modify code in **machine**. You are welcome to modify the machine in this case only, but it is not required. If you do not modify the machine, you will need to devise a hack to prevent Nachos from exiting when there are threads waiting for an alarm. A cheap if ugly fix is to fork a thread that yields in a tight loop iff there is at least one thread waiting for an alarm.

Warning: It is never correct for an interrupt handler to sleep. Think about the effect of this constraint on your scheme for synchronization.

Warning: Boxed Nachos does not deliver timer interrupts unless the **-rs** option is used. You may use the **-rs** option, but it is better to tweak the initialization code so that it has the correct behavior with or without **-rs**.

Problem 4: Elevator

Implement an elevator controller for a building with *F* floors, using threads and a synchronization scheme of your choosing, involving mutexes, condition variables, and/or semaphores.

The elevator controller is split between two classes, *Elevator* and *Building*. We have provided skeletal definitions of these classes in **elevator.h**. The definitions include two sets of methods: one set for internal use and another set that defines the external interface to the riders. You will need to add new internal methods to these classes to allow communication between the *Elevator* and *Building* classes, but do not modify the signatures of the methods that have already been defined.

A program that uses elevators will first create a single instance of *Building*, which will in turn create one *Elevator* object for each elevator in the building. Each elevator runs as a separate thread. Each elevator thread should enter an endless loop in an *Elevator* method, then use the internal elevator control interfaces (*OpenDoors*, *CloseDoors*, *VisitFloor* and the *Building* methods) to serve rider requests in an orderly fashion.

Each rider also runs as a separate thread. The riders use the elevator rider interface (*CallUp*, *CallDown*, *AwaitUp*, *AwaitDown*, *Enter*, *Exit*, *RequestFloor*) to request services from the elevator and its controller. These rider methods synchronize with the controller methods using shared state in the *Elevator* class and the *Building* class. You will implement these classes. **Note**: there is no thread for the *Building*; its code executes only when one of its methods is called by an elevator or a rider thread.

Part 1. Implement the elevator controller for a single elevator, including the *Building* and *Elevator* methods defined in **elevator.h**, and any new methods needed by your implementation. Your solution should avoid rider starvation as well as all obvious races, e.g., doors open while the elevator is in transit, riders entering and exiting while the doors are closed, etc. You may assume that the elevator is of unbounded size, i.e., it can hold all arriving riders. When your elevator stops at a floor, it should wait until all exiting riders have exited and all boarding riders have boarded.

Include a rider test program that demonstrates the operation of your elevator for multiple riders. Your rider program should be ‘‘well-behaved’’ in the following sense: any rider that calls the elevator for the up or down direction should then immediately wait until an elevator traveling in the requested direction arrives, then get on it, then request a floor in the correct direction, then get off at the correct floor.

Part 2. Extend your solution in Part 1 to handle elevators that can hold only *N* riders at a time, where *N* is a compile-time constant.

Part 3 (extra credit). Extend your solution to handle *E* elevators. The elevators should coordinate through the *Building* class to serve requests efficiently. For example, at most one elevator should go to pick up each set of passengers, and it should be carefully chosen. This is a difficult resource scheduling problem.

4.4 Lab 4: Multiprogrammed Kernel

Up until now, all of your test programs have executed entirely within the Nachos kernel. In a real operating system, the kernel not only uses its procedures internally, but allows *user programs* to access some of its routines via system calls. In Lab 4 you will modify Nachos to support execution of multiple concurrent user programs, using system calls to request services from the kernel.

To test your kernel, you will create some simple user programs as described in Section 3.5. These test programs will execute on the simulated MIPS machine within Nachos as discussed in Section 3.4. Your goal is to allow multiple processes to be created through the *Exec* system call and active at the same time. When you "boot" nachos and tell it to run a test program with **nachos -x** it will start running a test program using *StartProcess* in **progtest.cc**, and this test program will then be able to create new processes using *Exec*. Similarly, those processes will be able to create new processes and so on.

Your kernel will use process page tables and your memory management code to allow multiple processes to reside in the simulated machine memory concurrently. It will use preemptive scheduling to share the simulated CPU fairly among the active user processes, and it will allow processes to synchronize their execution using the *Join* system call.

Since your kernel does not trust user programs to execute safely, the kernel and the (simulated) hardware will work together to protect the system from damage by malicious or buggy user programs. Virtual addressing prevents user processes from accessing kernel data structures, and all attempts by a user program to enter the kernel funnel through your *ExceptionHandler* routine. However, your kernel must be very careful about passing system call arguments and results between the user program and the kernel. In particular, your system call code must convert user-space addresses to Nachos machine addresses or kernel addresses before they can be dereferenced. Your system call scheme must ensure that the kernel is ''bulletproof'', i.e., buggy or malicious user programs cannot cause the kernel to crash or behave inappropriately.

This assignment may sound difficult, but most of the basic infrastructure is already in place. In particular: (1) the thread system and timer device already support preemptive timeslicing of multiple user threads, (2) the thread context switch code already saves and restores MIPS machine registers and the process page table, and (3) the Nachos distribution includes skeletal code to set up a new user process context, load it from an executable file, and start a thread running in it. As with all the programming assignments this semester, you can complete Lab 4 with at most a few hundred lines of code. The hard part is figuring out how to build on the Nachos code you already have — and debugging the result if you don't get it right the first time.

Most of the new files that you will be using are in the **nachos/code/userprog** directory. You will build your **nachos** executables in **userprog** instead of **threads**: be sure you build and run the "right" **nachos**. Even before you make any changes, you can build a Nachos executable that allows you to run a single user program, however, there is only skeletal support for exceptions or system calls. Most of the heavy lifting for Labs 4 and 5 is rooted in **userprog/exception.cc** and **addrspace.cc**, and there is some useful code to start with in **progtest.cc**.

For Lab 4 you will implement the process-related system calls defined in **userprog/syscall.h** (also see Section 5), including the *Exec*, *Exit* and *Join* system calls. Since your kernel will allow multiple processes to run at once, you will need appropriate synchronization in your kernel code that handles processes and system calls.

First you will need basic facilities to load processes into the memory of the simulated machine, and some sort of process table to keep track of active processes. Spend some time studying the *AddrSpace* class, and look at how the *StartProcess* procedure uses the *AddrSpace* class methods to create a new process, initialize its memory from an executable file, and start the calling thread running user code in the new process context. The current code for *AddrSpace* and *StartProcess* assumes that there is only one program/process running at a time (started with *StartProcess* from main by the **nachos -x** option), and that all of the machine's memory is available to that process. Your first job is to generalize the code for *StartProcess* and *AddrSpace*, and use it to implement the *Exec* system call.

1. Implement a memory manager module to allow your kernel to allocate page frames of the simulated machine's memory for specific processes, and to keep track of which frames are free and which are in use. You may find your *Table* class from Lab 3 useful here.
2. Modify *AddrSpace* to allow multiple processes to be resident in the machine memory at the same time. The default *AddrSpace* constructor code assumes that all of the machine memory is free, and it loads the new process contiguously starting at page frame 0. You must modify this scheme to load the process into page frames allocated for the process using your memory manager.

   Your code should always succeed in loading the process if there is enough free memory for it, thus you must allow address spaces to be backed by noncontiguous frames of physical memory. For now it is acceptable to fail if there is not enough free machine memory to load the executable file. You will need to move the *AddrSpace* loading code out of the constructor and into a new *AddrSpace* method to allow you to report a failure. It is poor programming practice to put code that can fail into a class constructor, as the Nachos designers have done in this release.
3. If necessary, update *StartProcess* to use the new *AddrSpace* interface so that you do not break the **nachos -x** option.
4. Modify *AddrSpace* to call the memory manager to release the pages allocated to a process when the process is destroyed. Make sure your *AddrSpace* code also releases any frames allocated to the process in the case where it discovers that it does not have enough memory to load the entire process.

Next, use these new facilities to implement the *Exec* and *Exit* system calls. If an executing user processes requests a system call (by executing a trap instruction) the machine will transfer control to your kernel by calling *ExceptionHandler.* Kernel code must decode the system call identifier and the arguments, and call internal procedures that implement the system call. Here are a couple of issues you need to attend to:

1. When an *Exec* call returns, your kernel should have created a new process and started a new thread executing within it to run the specified program. However, you do not need to concern yourself with setting up *OpenFileIds* until the next assignment. For now, you will be able to run user programs, but they will not be able to read any input or write any output.
2. For *Exec*, you must copy the filename argument from user memory into kernel memory safely, so that a malicious or buggy user process cannot crash your kernel or violate security. In particular, you must handle the case where the filename string crosses user page boundaries and resides in noncontiguous physical memory. You must also detect an illegal string address or a string that runs off the end of the user's address space without a terminating null character, and handle these cases by returning an error (*SpaceId* 0) from the *Exec* system call.

You may impose a reasonable limit on the maximum size of a file name. Also, use of *Machine:ReadMem* and *Machine:WriteMem* is not forbidden as the comment in **machine.h** implies.

3. *Exec* must return a unique process identifier (*SpaceId*), which can be used as an argument to Join, as discussed in Section 5.1. Your kernel will need to keep a table of active processes. You may find your *Table* class from Lab 3 useful here. Be sure your *Join* system call validates any *SpaceId* passed to it by a user process.

4. Implement the Nachos kernel code to handle user program exceptions that are not system calls. Exceptions, like address protection and the kernel/user mode bit, are important hardware features that allow the hardware and OS kernel to cooperate to ''bullet-proof'' the operating system from user program errors. The machine (or simulator) raises an exception whenever a user program attempts to execute an instruction that cannot be completed, e.g., because of an attempt to reference an illegal address, a privileged or illegal instruction or operand, or an arithmetic underflow or overflow condition. The kernel's role is to handle these exceptions in a reasonable way, i.e., by printing an error message and killing the process rather than crashing the machine. No, an *ASSERT* that crashes Nachos does not qualify as a reasonable way to handle a user program exception.

   You may find it convenient to implement process exit as an internal kernel procedure called by the *Exit* system call handler, rather than calling the lower-level procedures directly from *ExceptionHandler.* This will make it easy to “force” a process to exit from inside of the kernel, by calling the internal exit primitive from another kernel procedure in the process' context. In general, this kind of careful internal decomposition will save you from reinventing and redebugging wheels, and it is always good practice.

5. Synchronization between *Join* and *Exit* is tricky. Be sure you handle the case where the joinee exits before the joiner executes the *Join*. Your kernel should also clean up any unneeded process state if *Join* is never called on a given exiting process.

6. To implement *Join* correctly and efficiently, you will need to keep a list of all children of each process. This list should be maintained in temporal order, so that you can always determine the most recently created child process. This will be necessary when you implement pipes in Lab 5.

7. **Extra credit**. Implement the *Yield* and *Fork* system calls to allow Nachos user programs to use multiple threads, as defined in Section 5.4.

Finally, you should test your code by exercising the new system calls from user processes.

1. Write a test program(s) to create a tree of processes *M* levels deep by calling *Exec N* times from each parent process, and joining on all children before exiting. You may hard-code *M* and *N* in your test programs as constants, since *Exec* as defined provides no way to pass arguments into the new program.

2. Create a modified version of the tree test program in which each parent process exits without joining on its children. The purpose of this program is to (1) test with larger numbers of processes, and (2) test your kernel's code for cleaning up process state in the no-join case.

3. Run your tree test programs with timeslicing enabled (using the Nachos **-rs** option) to increase the likelihood of exposing any synchronization flaws.

4.5 Lab 5: I/O

For Lab 5, you will extend your multiprogrammed kernel with support for I/O, by implementing the system calls for reading and writing to files, pipes, and the console: *Create*, *Open*, *Close*, *Read*, and *Write*. You will exercise your kernel by implementing a simple command interpreter (shell) that can run multiple test programs concurrently.

Use the *FileSystem* and *OpenFile* classes as a basis for your file system implementation. For now, your implementations of the file system calls will use the default ‘‘stub’’ file system in Nachos since *FILESYSTEM_STUB* is defined. The stub file system implements files by directly calling the underlying host (e.g., Solaris) file system calls; thus you can access files in the host file system from within Nachos using their ordinary Unix file names.

Lab 5 includes the following requirements and options.

1. Implement, test, and debug the *Create*, *Open* and *Close* system calls. Be sure to correctly handle the case where a user program passes an illegal string to *Open*/*Create* or an illegal *OpenFileId* to *Close* (recall the discussion of similar concerns for *Exec* and *Join* in Section 4.4).

   The *Open* system calls return an *OpenFileId* to identify the newly opened file in subsequent *Read* and *Write* calls. Note that it is not acceptable to use an *OpenFile** or other internal kernel pointer as an *OpenFileId*, because a user program could cause the kernel to follow an illegal pointer. Instead, you will need to implement a per-process (per-*AddrSpace*) open file table to assign integer *OpenFileIds* and to map them to *OpenFile* object pointers by indexing into the protected kernel table. Your *Table* class from Lab 3 may be useful here. Of course, your process must properly clean up the file table along with other process state when a process exits.

2. Implement, test and debug the *Read* and *Write* system calls for open files. Again, you must be careful about moving data between user programs and the kernel. In particular, you must ensure that the entire user buffer for *Read* or *Write* is valid. However, your scheme for moving bulk data in or out of the kernel for *Read* and *Write* must not arbitrarily limit the number of bytes that that the user program can read or write.

   We will supply a kernel primitive to validate user addresses and translate them to kernel addresses one page at a time.

3. Modify *Exec* to initialize each new process with *OpenFileIds* 0 and 1 bound by default to the console. As defined in **syscall.h**, *OpenFileIds* 0 and 1 always denote the console device, i.e., a program may read from *OpenFileId* 0 or write to *OpenFileId* 1 without ever calling *Open*. To support reading and writing the console device, you should implement a *SynchConsole* class that supports synchronized access to the console. Use the code in **progtest.cc** as a starting point. Your *SynchConsole* should preserve the atomicity of *Read* and *Write* system calls on the console. For example, the output from two concurrent *Write* calls should never appear interleaved.

   Warning: Failure to carefully manage the order of initialization is often a source of bugs. To use the console, you must create a Nachos *Console* object. Create this object with *new* late in *Initialize* in **system.cc**, after the interrupt mechanism is initialized.

Warning: Once you create a *Console* object, you may be annoyed that nachos no longer shuts down when there is nothing to do, as discussed in Section 3.4. For the rest of the semester, get in the habit of killing nachos with **ctrl-c** when each run is complete. You may even start to enjoy it.

4. Implement pipes using the interface and semantics defined in Section 5.3. Your *BoundedBuffer* class from Lab 3 may be useful here.
5. **Extra credit**. Modify your file-related system calls to use the Nachos file system discussed in class, rather than the stub file system. To do this, you will need to undefine *FILESYSTEM_STUB,* then run nachos using the options to create a disk and filesystem and perhaps install some files in it. You must also modify the Nachos file system code to synchronize file accesses so that directory operations and read/write operations execute atomically. For example, data written by concurrent *Write* calls should never be interleaved, and a *Read* should never return partial results from a concurrent *Write*. The host Unix system will provide these guarantees for you if the stub file system is used. (However, this does not mean that your system call code does not need to synchronize when the stub file system is used.)

Spend some time devising test programs to exercise your kernel:

1. Write a user program that exercises the system calls and triggers exceptions of different kinds. You do not need to exhaustively test all the types of exceptions; that's boring.
2. Write a shell using the sample in **test/shell.c** as a starting point. The shell is a user program that loops reading commands from the console and executing them. Each command is simply the file name of another user program. The shell runs each command in a child process using the *Exec* system call, and waits for it to complete with *Join*. If multiple commands are entered on the same line (e.g., separated by a semicolon), the shell executes all of them concurrently and waits for them to complete before accepting the next command.

   You may define your own shell command syntax and semantics. Real shells (e.g., the Unix **tcsh**) include sophisticated features for redirecting program input and output, passing arguments to programs, controlling jobs, and stringing multiple processes together using pipes. Your shell should demonstrate use of pipes, but the Nachos kernel interface is not rich enough to support most of the other interesting features.

   We will supply some string-parsing functions to simplify your shell.
3. Test your kernel by using your shell to execute some test programs as concurrent processes. Adequate testing is a critical aspect of any software project, and designing and building creative test programs is an important part of this assignment.

   One useful utility program is **cp**, which copies the contents of a file to a destination file. If the destination file does not exist, then **cp** creates it. Your implementation of **cp** will exercise command line arguments using the following syntax: **cp source-file destination-file**).

   You may find **cat** useful to demonstrate pipes: **cat filename** copies the file named by **filename** to its standard output (*OpenFileId* 1); **cat** with no arguments simply copies its standard input (*OpenFileId* 0) to its standard output.

4.6 Labs 6-7: Virtual Memory

In these assignments you will modify Nachos to support virtual memory. The new functionality gives processes the illusion of a virtual memory that may be larger than the available machine memory.

In Lab 6, you will implement page faults and dynamically load your program's pages on demand, rather than initializing page frames for the process at *Exec* time. As in Labs 4-5, you should allocate all page frames for a newly executed process at *Exec* time, and return an error from the *Exec* system call if there are not enough free page frames for the new address space. However, once you have allocated the frames, you will load the frames with their contents *dynamically* in response to page faults, rather than initializing the contents of all frames in advance at *Exec* time.

In Lab 7, you will implement page replacement. This allows you to simultaneously execute more processes than would fit in machine memory at any one time. In this case, you will have to evict a page in order to satisfy a page fault. **Warning**: Lab 7 is much harder than Lab 6. Don't procrastinate. Think about the issues for Lab 7 (e.g., allocating swap space as described below) while you're implementing Lab 6.

Page tables were used in Labs 4 and 5 to simplify memory allocation and to prevent failures in one process from affecting other processes. Page tables serve another purpose in Labs 6 and 7. Your kernel will work together with the machine (MMU) mechanisms to implement virtual memory, communicating through special bits in each page table entry (PTE).

- Your kernel uses the *valid* bit in each PTE to tell the machine which virtual pages are resident in memory (a valid translation) and which are not resident (an invalid translation). If a user process references an address for which the PTE is marked invalid, then the machine raises a *page fault* exception and transfers control to your kernel's exception handler.
- The machine uses two other bits to pass your kernel information about how pages are being used by user programs as they execute. The machine sets the *use* bit (reference bit) in the PTE whenever the corresponding page is referenced, and it sets the the *dirty* bit in the PTE whenever the page is modified. (These bits are used in Lab 7 but not Lab 6.)

4.6.1 Details for Lab 6

For Lab 6, you should allocate memory for each newly created process at *Exec* time and fill the page frame numbers into the page table, just as in Lab 4. However, unlike Lab 4, you will initialize all the PTEs as *invalid*, and then prepare the page frames on demand as the process references its pages, triggering page fault exceptions.

Your kernel will then satisfy each fault in an appropriate way. For example, a fault on a text page should read the text from the executable file, and a fault on a stack or unitialized data frame should zero-fill the frame. However you initialize the frame, handle the exception by marking the PTE as valid, then resuming execution of the user program at the faulting instruction.

See the notes on testing in Section 4.6.3.

4.6.2 Details for Lab 7

To implement virtual memory with paging, your operating system will have to use main memory as a cache for the disk. In Lab 6, frames are preallocated to pages, so it is always easy to find a frame to satisfy each page fault. In Lab 7, frames will not be reserved in advance in *Exec*; instead, they should be allocated on demand by your page fault exception handler. It may be necessary to free up a frame by evicting some other page from memory, using one of the page replacement policies discussed in class. Your policy may examine and reset the *use* bit in each PTE in order to gather information to drive the policy.

As with any caching system, performance depends on the policy used to decide which pages are kept in memory and which to evict. When your kernel evicts a page, it must take care to invalidate any page table entries (possibly in the page tables of other processes) that point to the frame containing the victim page. If the evicted page is referenced again shortly after eviction, then this will trigger another page fault, forcing the kernel to choose yet another victim. Choose your policy carefully based on your group's design criteria (e.g. simplicity, performance, simplicity, correctness) and be able to explain your choices.

The simulated MIPS machine provides enough functionality to implement a fully functional virtual memory system. Do not modify the "hardware". In particular, the simulator procedure *Machine::Translate* is off limits.

If your kernel decides to evict a page that is dirty, it must write the page contents out to disk before replacing it. Be sure to clear the dirty bit when you mark the PTE of the victim page as invalid. You will need routines to move a page from memory to disk (for pageout) You will need to use the Nachos file system (or stub file system) as backing store, and you will need routines to push a page from memory to backing store (for pageout) and from backing store to memory (for pagein).

Because of the delays and synchronization issues inherent in evicting and writing back dirty pages on demand, it is preferable to use a special system thread (a *paging daemon*) to periodically select eviction candidates, clean their frames if necessary, and invalidate their PTEs. In this way, a clean, unmapped frame will always be available to grab if a page fault occurs. Think about how to put the paging daemon to sleep when there is plenty of free memory, and when to wake it up.

4.6.3 Testing Your VM System

Labs 6 and 7 build directly on Labs 4 and 5, so you will mostly be working with the same set of files. In addition, there is a test case available in **test/matmult.c**. This program exercises the virtual memory system by multiplying two matrices. Of course, you may not increase the size of main memory in the Nachos machine emulation, although for debugging, you may want to decrease this number. You may find it useful to devise your own test cases - they can be simpler patterns of access to a large array that might let you test and debug more effectively.

Devising useful test programs is crucial in all of these assignments. Give thought to how you plan to debug and demo your kernels. Initial testing with only a single user process may be a good idea so that you can trace what is going on in a simpler environment. In later testing, you want to stress things more and run multiple programs.

Lab 6 implies that pages are brought into memory only if they are actually referenced by the user program. To show this, one could devise test cases for various scenarios, such as (1) the whole program is, in fact, referenced during the lifetime of the program; (2) only a small subset of the pages are referenced. Accessing an array selectively (e.g. all rows, some rows) can give different page reference behavior. We don't particularly care if the test program does anything useful (like multiplying matrices), but it should generate memory reference patterns of interest.

In Lab 7, the key issues are your page replacement policy and correct handling of dirty pages (e.g., allocating backing storage and preserving the correct contents for each page on backing store). Again the test cases can generate different kinds of locality (good and bad) to see how the paging system reacts. Can you get the system to *thrash*, for example? Have you built in the kinds of monitoring and debugging that lets you see if it is thrashing? Try reducing the amount of physical memory to ensure more page replacement activity.

Consider what tracing information might be useful for debugging and showing off what your kernel can do. You might print information about pagein and pageout events, which processes are affected (whose pages are victimized) or responsible (who is the faulting process) for each event, and how much of the virtual address space is actually loaded at any given time. It is useful to see both virtual and physical addresses. It may also be useful to print the name of the executable file that each process is running, as a way to identify the process.

5 Nachos System Call Interface

5.1 Process Management

There are three system calls for executing programs and operating on processes.

*SpaceId Exec (char *executable, int pipectrl)*

> Creates a user process by creating a new address space, reading the *executable* file into it, and creating a new internal thread (via *Thread::Fork*) to run it. To start execution of the child process, the kernel sets up the CPU state for the new process and then calls *Machine::Run* to start the machine simulator executing the specified program's instructions in the context of the newly created child process. Note that Nachos *Exec* combines the Unix *fork* and *exec* system calls: *Exec* both creates a new process (like

Unix *fork*) and executes a specified program within its context (like Unix *exec*).

Exec returns a unique *SpaceId* identifying the child user process. The *SpaceId* can be passed as an argument to other system calls (e.g., *Join*) to identify the process, thus it must be unique among all currently existing processes. However, your kernel should be able to recycle *SpaceId* values so that it does not run out of them. By convention, the *SpaceId* 0 will be used to indicate an error.

The *pipectrl* parameter is discussed in Section 5.3. User programs not using pipes should always pass zero in *pipectrl.*

void Exit (int status)

A user process calls *Exit* to indicate that it is finished executing. The user program may call *Exit* explicitly, or it may simply return from *main*, since the common runtime library routine that calls *main* to start the program also calls *Exit* when *main* returns. The kernel handles an *Exit* system call by destroying the process data structures and thread(s), reclaiming any memory assigned to the process, and arranging to return the exit *status* value as the result of the *Join* on this process, if any. Note that other processes are not affected: do not confuse *Exit* with *Halt*.

int Join (SpaceId joineeId)

This is called by a process (the *joiner*) to wait for the termination of the process (the *joinee*) whose *SpaceId* is given by the *joineeId* argument. If the joinee is still active, then *Join* blocks until the joinee exits. When the joinee has exited, *Join* returns the joinee's exit status to the joiner. To simplify the implementation, impose the following limitations on *Join*: the joiner must be the parent of the joinee, and each joinee may be joined on at most once. Nachos *Join* is basically equivalent to the Unix *wait* system call.

5.2 Files and I/O

The file system calls are similar to the Unix calls of the same name, with a few differences.

*void Create (char *filename)*

Create an empty file named **filename**. Note this differs from the corresponding Unix call, which would also open the file for writing. User programs must issue a separate *Open* call to open the newly created file for writing.

*OpenFileId Open (char *filename)*

Open the file named *filename* and return an *OpenFileId* to be used as a handle for the file in subsequent *Read* or *Write* calls. Each process is to have a set of *OpenFileIds* associated with its state and the necessary bookkeeping to map them into the file system's internal way of identifying open files. This call differs from Unix in that it does not specify any access mode (open for writing, open for reading, etc.)

*void Write (char *buffer, int size, OpenFileId id)*

Write *size* bytes of the data in the buffer to the open file identified by *id*.

*int Read (char *buffer, int size, OpenFileId id)*

Try to read *size* bytes into the user *buffer.* Return the number of bytes *actually* read, which may be less than the number of bytes requested, e.g., if there are fewer than *size* bytes available.

void Close (OpenFileId id)

Clean up the “bookkeeping” data structures representing the open file.

5.3 Pipes

The *Exec* system call includes a *pipectrl* argument as defined in Section 5.1. This argument is used to direct the optional binding of *OpenFileIds* 0 (stdin) and 1 (stdout) to pipes rather than the console. This allows a process to create strings of child processes joined by a pipeline. A *pipeline* is a sequence of pipes, each with one reader and one writer. The first process in the pipeline has stdin bound to the console and stdout bound to the pipeline input. Processes in the middle of the pipeline have both stdin and stdout bound to pipes. The process at the end of the pipe writes its stdout to the console.

The Nachos interface for creating pipes is much simpler and less flexible than Unix. A parent process can use nonzero values of the *pipectrl* argument to direct that its children are to be strung out in a pipeline in the order in which they are created. A *pipectrl* value of 1 indicates that the process is the first process in the pipeline. A *pipectrl* value of 2 indicates a process in the middle of the pipeline; stdin is bound to the output of the preceding child, and stdout is bound to the input of the next child process to be created. A *pipectrl* value of 3 indicates that the process is at the end of the pipeline.

To handle these *pipectrl* values, the kernel must keep a list of all children of each process in the order that they are created.

Pipes are implemented as producer/consumer bounded buffers with a maximum buffer size of *N* bytes. If a process writes to a pipe that is full, the *Write* call blocks until the pipe has drained sufficiently to allow the write to continue. If a process reads from a pipe that is empty, the *Read* call blocks until the sending process exits or writes data into the pipe. If a process at either end of a pipe exits, the pipe is said to be *broken*: reads from a broken pipe drain the pipe and then stop returning data, and writes to a broken pipe silently discard the data written.

5.4 Threads

Thread support may be implemented for extra credit in Labs 4-5. Nachos systems with thread support should define the semantics of *Exit* in the following way. *Exit* indicates that the calling thread is finished; if the calling thread is the last thread in the process then it causes the entire process to exit (e.g., wake up joiner if any). The status value reported by the last thread in the process to call *Exit* shall be deemed the exit status of the process, to be returned as the result from *Join*.

*void Fork (void(*function)())*

Creates a new thread of control executing in the calling user process address space. The *function* argument is a procedure pointer, a user-space virtual address: the new thread will begin executing at this address in the same address space as the thread calling *Fork*. The new thread must have its own user stack in the user address space, separate from all other threads. It must be able to make system calls and block independently of other threads in the same process.

void Yield()

Called within a user program to yield the CPU. *Yield* triggers a *Thread::Yield* within Nachos, allowing some other ready thread to run.