Nachos Project Guide

CPS 110: Introduction to Operating Systems
Spring 2001

Jeff Chase

Duke University
Department of Computer Science

Nachos Project Guide

1 Preface

2 Nachos Project Policies and Mechanisms

2.1 Project Teams 4
2.2 Demo Interviews 4
2.3 What to Hand In 5
2.4 Grading 6
2.5 Extensions, Late Work, Partial Credit, and Solutions 6
2.6 Working With Your Team 7
2.7 What Parts of Nachos Should We Modify? 8

3 Working With Nachos

3.1 Installing and Building Nachos 9
3.2 Systems Programming 9
3.3 Tracing and Debugging Nachos Programs 10
3.3.1 The *Debug* Primitive 11
3.3.2 Miscellaneous Debugging Tips 11
3.3.3 Defining New Command-Line Flags for Nachos 11
3.4 Controlling the Order of Execution in Nachos 12
3.4.1 Context Switches 12
3.4.2 Voluntary Context Switches with Thread::Yield 12
3.4.3 Involuntary Context Switches with the -rs Flag 13
3.5 The Nachos MIPS Simulator 13
3.5.1 The MIPS CPU Simulator (SPIM) 14
3.5.2 Interactions Between the Kernel and the Machine 14
3.5.3 I/O Devices, Interrupts, and the Console 14
3.6 Creating Test Programs for Nachos Kernels 15
3.6.1 Troubleshooting Test Programs 16

4 Nachos Lab Assignments

4.1 Lab 1: The Trouble with Concurrent Programming 17
4.2 Lab 2: Threads and Synchronization 18
4.3 Lab 3: Programming with Threads 22
4.4 Lab 4: Multiprogrammed Kernel 25
4.5 Lab 5: I/O 29
4.6 Lab 6: Virtual Memory 31

5 Nachos System Call Interface

5.1 Process Management 35
5.2 Files and I/O 36
5.3 Pipes 37
5.4 Threads 37

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 and in subsequent semesters.

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. 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 lab due every two weeks, starting two weeks after the first day of classes. This section defines the course procedures and policies for doing the Nachos labs and assigning grades. Please read it carefully.

The specific 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 2-5 students. Form your project groups amongst yourselves before the first assignment is due. You should try to form a group of four.

From our perspective, the ideal teams would feature an even distribution of the strongest students and the less strong students. If your preparation for the course is weak, then try to team up with some students who are better prepared and more confident. If you are a more capable student, please consider it your personal mission to share your talent with other students 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.

2.2 Demo Interviews

Most of our evaluation of your labs will occur in demo/interview 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. Instructions for scheduling demos will be given in class and on the course

web page. Please try to spread your demos evenly among the instructor and the various 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 team member will attend every demo, but we will allow a few exceptions if you cannot reconcile your schedules. 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 (or stand at a whiteboard) and lead the conversation about your project. Any team member should feel free to add comments during the demo, and we may choose to address our questions to any team member.

Many students are nervous about the demos early in the semester. There would seem to be good reason for this: we will review your code, ask you to show us that the code works, attempt to uncover bugs, and generally put you on the spot to explain what you did and why. 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, then the demos will be fun.

2.3 What to Hand In

Please e-mail a short writeup for each assignment to the instructor before the deadline, with the string **nachos project writeup *N*** in the subject line, where *N* is the assignment number. The message should cc: all project team members. The first line of your message should be the full pathname of your team's source code directory. You should also make every effort to sign up for your demos before the deadline; if possible, please cc: your writeup to the TA with whom you will demo the lab, and state the TA's name and the date and time of the scheduled demo in the second line of the message. The rest of the message should give an overview of your approach to the assignment, and a summary of the status, i.e., what works and what does not.

Writeups should be short and sweet. Do not spend too much effort on your writeups: the course is time-consuming enough as it is. The purpose of the writeup is to give you an opportunity to "come clean" about any problems with your work, and to add information that may be useful in grading. If you had specific problems or issues, approaches you tried that didn't work, or concepts that were not fully implemented, then an explanation in your writeup may help us to assign partial credit. If your code looks good and your demo interview goes well, then we might not even read your writeup.

Expect that your source code will be automatically archived at deadline time. This allows us to keep a snapshot of the state of your code at deadline, 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 lose something. You are strongly encouraged to use version control tools such as CVS, but *please* leave clean and up-to-date versions of your **.c** and **.h** files in the directory for us to look at. Try to make it obvious which files contain the code you want us to grade you on.

2.4 Grading

Our goal is to give every team credit for the work they have done, while giving the most credit to the teams that do the best work on each assignment. Your grade is based partly on your explanation of your work during the demo. Your implementation is graded on completeness, correctness, programming style, and thoroughness of testing. Bugs that were not uncovered by your testing cost more than bugs you are aware of and tell us about in advance. We typically give less than half-credit for projects that do not build or do not run.

Note that in this course we do NOT grade your code by how fast it executes. Performance concerns are critical in many software systems, but correctness ALWAYS comes first. As an unbending rule, you should strive for solutions that are simple, direct, elegant, structurally sound, easy to extend, and obviously correct. Simple solutions can save you endless nights of debugging time. These solutions are also the easiest to grade, and they demonstrate that you understand the principles and know what is "right".

Each project is graded on a 100-point scale. At Duke it is typical for one or more teams to receive perfect scores on any given lab. You should be pleased with any grade above an 80. Grades below 70 indicate trouble. If you are in trouble, feel free to stop by during office hours to talk about it with the instructor (or send e-mail).

We award extra credit points for optional work as described in the lab descriptions, or as negotiated with the instructor. Extra credit points are reported separately from the lab score and are considered separately in all grading. They are used to "bump up" final grades at the end of the semester, after the curve is established. Your final grade cannot be harmed by a choice to skip the optional extra credit work.

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 for each lab at most a few days after all demos for that lab are complete.

2.5 Extensions, Late Work, Partial Credit, and Solutions

Extensions. We have never granted penalty-free extensions for individual teams on the Nachos projects. In general, if one member of your group is 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, is stuck in K-ville, has a job interview or a hangover, or just flakes out, then the other team members are expected to 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 deadline, and between the deadline and the demo. If you change code after the deadline, please send e-mail to the instructor and the demo TA before the interview, telling us about any changes you have made after the deadline. Generally

we give about half credit for work done after the deadline, 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 is solid enough to build on for subsequent labs. It has never been necessary to hand out solutions. We would rather have you expend the effort to fix your own bugs for each lab, rather than heaving your first attempt and using our canned solution instead. If you take the time to fix up mistakes, we will give you some points back during the demo for the next lab.

2.6 Working With Your Team

One purpose of college is to learn about life. Most of the work that you do in your life, and in this course, will be in teams. Conflicts among team members are a fact of life. Good strategies for avoiding and coping with these conflicts are crucial to your success in this course, and in life.

Team strategies. With the Nachos projects, some strategies are better than others for coordinating the activities of your team. A common error is to attempt a superficial division of the lab requirements among the team members, with each member responsible for fulfilling specific requirements. This might work in the first few labs, but teams often suffer painful losses of points for the mistakes of individual team members. In the later labs, the various requirements are so intertwined that you cannot reasonably divide the work until you understand all of the pieces of your system and how they fit together. You should meet as a group to work out a solid high-level design, before you partition the work and before you start coding. If you take the time to do this phase well, then your group will have a more effective and equitable division of labor, you will spend much less time (re)writing and (re)debugging code, you will learn more, and we will all be happier with the result.

Slackers, dictators, and irreconcilable differences. We run the course on the premise that you are all adults who are capable of coordinating your efforts effectively without our intervention. In practice this occasionally turns out not to be the case. If you feel that a team member is not pulling their weight or is otherwise behaving unreasonably, you are free to voice your concerns to the instructor. Use 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 comments and identity will be held in confidence. If your team develops "irreconcilable differences", then a "divorce" may be the only solution.

Divorces and other team reorganizations. You may reorganize your groups at any time during the semester if there is mutual consent of all parties involved. In extreme cases, a team may eject a "slacker" without their consent by unanimous vote of the surviving members. If you choose to reorganize, please state and justify your intention in an e-mail message to the instructor and the TA, with a cc: to all team members involved. We reserve the right to deny any request to reorganize teams.

Teams and grades. Conflicts among team members are often amplified by stress about grades. It is true that the success of your Nachos team determines a large component of your final grade in this course, even if mistakes made by your team are "not your fault". This is life, and life is not always fair. However, in managing this course we try very hard to understand what is happening in each team and to assign a fair grade to each individual at the end of the semester. In practice, problems with teams tend to matter less in the final grading than most students expect.

2.7 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" C++ 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 3.5). 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. Much of it will not make sense to you at first. 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 defines 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 resource page on the course web includes an HTML source code browser and instructions for a full installation of the Nachos release we will use this semester.

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 **gmake 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 Systems Programming

In this course you will be using C++ to do "systems programming" which is different in character from most of programming work in introductory courses. Systems programming requires you to be more directly in touch with the representation of data structures in the machine memory. For example, systems programmers often need to set up data structures interpreted directly by hardware, which expect everything

to be just exactly perfect. Examples that will come up during the semester include thread stack frames, page tables, and saved register contexts. Also, to understand concurrency behaviors (e.g., in Lab 1) you will need to understand how lists and other data structures are represented in memory.

One risk of learning about nice clean object-oriented programming abstractions in introductory courses is that it is easy to lose sight of how data structures are really laid out in machine memory. We have found that students often make time-consuming errors in this course as a result. In particular, be sure that you understand the relationship between character strings, pointers, and arrays in "raw" C and C++, and that you understand how memory for your data structures is allocated. We often see students use array indexing and/or string functions to address storage that they did not allocate properly. This always leads to unexpected errors that are difficult to trace back to the source.

In general, we discourage the use of prepackaged classes and C++ features not already present in the nachos release. You can do everything you need to do in this course cleanly and directly using simple constructs such as arrays together with the *List* class already included in nachos. Experience has shown that use of fancier features rarely if ever results in cleaner code, occasionally leads to build problems, and very often leads to tedious debugging marathons. Templates, vectors, and overuse of inheritance have proven to be particularly dangerous. We advise you to try to write the cleanest and simplest code that you can, using basic C constructs wrapped in a C++ veneer.

3.3 Tracing and Debugging Nachos Programs

There are at least three ways to trace execution: (1) add *printf* (or *fprintf*) 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 operating system's *write* system call to put the output on your screen. 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* to avoid this problem.

One of the first concepts you will learn in this course is the idea of a *thread*. Your Nachos programs will execute as multiple independent threads, each with a separate execution stack. When you trace the execution path of your program, it is helpful to keep track of the state of each thread and which procedures are on each thread's stack. You will notice that when one thread calls *SWITCH*, another thread starts running (this is called a *context switch*), and the first thing the new thread does is to return from *SWITCH*. Because **gdb**

and other debuggers are not aware of the Nachos thread library, tracing across a call to *SWITCH* might be confusing sometimes.

3.3.1 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 **threads/main.cc** for a specification of the flags arguments for **nachos**. The relevant one for *DEBUG* is **-d**. The **-d** flag followed by a space and a list of single-character debug flags (e.g., **nachos -d ti**), enabling the nachos *DEBUG* statements matching any of the specified flags. For example, the **t** debug flag activates *DEBUG* statements relating to thread events. 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 for thread and "interrupt" events. If you want to know more, add 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.

3.3.2 Miscellaneous Debugging Tips

The *ASSERT* macro, 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.

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**.

3.3.3 Defining New Command-Line Flags for Nachos

In addition to defining new debug flags as described in Section 3.3.1, 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 are available on the course web site.

3.4 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 runs fine; other times it crashes out of the starting gate. A program that works once may fail on the next run because the system happened to run the threads in a different order. In principle, 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.

3.4.1 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 system chooses to run the threads. 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 a *yield* or *sleep* primitive (*Thread::Yield* or *Thread::Sleep*) to cause the system to switch to another thread. A thread running in Nachos might initiate a voluntary switch for any of a number of reasons, perhaps in the implementation of a synchronization facility (e.g., locks in Lab 2).

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.

3.4.2 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.

3.4.3 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 used in the second half of the semester (see below). In the synchronization assignments, all of the code is executing within the Nachos "kernel" running as an application program on the underlying host system (e.g., a SPARC/Solaris workstation where you log in). 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 nachos kernel 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*.

3.5 The Nachos MIPS Simulator

In these labs you will implement key pieces of an operating system supporting user programs and a protected kernel. In principle, this requires that we give each of you your own machine 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. (MIPS processors were used in some workstations when nachos was first written; they are still around, but now they are mostly used for embedded systems applications, such as inside network switches.) You will use the simulated MIPS machine in Labs 4-6 to execute test programs above your operating system, as discussed in Section 3.6. Your **nachos** executable will contain a MIPS simulator (called SPIM) that reads the test program executables as data and *interprets* them, simulating their execution on a real MIPS machine booted with your Nachos kernel. "It's like a ship in a bottle."

3.5.1 The MIPS CPU Simulator (SPIM)

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.

3.5.2 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.

Like a real kernel, your Nachos kernel must examine and modify the machine registers and other machine state in order to service exceptions and run user programs. For example, system call arguments and results are passed between user programs and the kernel through the machine's registers. Your kernel will also examine and modify *page table* data structures that are used by the simulated machine to translate virtual addresses in the current user program. From the perspective of the Nachos kernel, all of the machine state — registers, memory, and page tables — are simply arrays in the kernel address space, accessible through the *Machine* object. See the definitions in **machine/machine.h**.

Note: in these projects, the page table format is defined by the machine architecture, not by the operating system. When your kernel examines or modifies page tables — or any other machine state — you should be certain to “give the machine what it wants”, carefully adhering to the defined interface between the machine and the operating system. *Do not attempt to change the architecture!* In the real world, the operating system and the hardware architecture are often defined by different companies (e.g., Microsoft and Intel respectively); in this case, the software designers cannot dictate the hardware architecture, and they must respect the defined hardware interface or else their software *will not work.*

3.5.3 I/O Devices, Interrupts, and the Console

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.

3.6 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 3.5.

Because the user programs are compiled for the MIPS architecture, they will not run directly on the host 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 (old) 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 other features 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 exe-

cution of the **halt** program using the debug flag (see Section 3.3). The **test** directory includes other simple user programs to test your kernels in the later assignments. For later labs we also expect you to extend these tests and add some of your own.

3.6.1 Troubleshooting Test Programs

Some students have difficulty building and running new test programs, and may even spend lots of good sleeping time trying to track down "Nachos bugs" that were actually bugs in their test programs. The following guidelines will help you to 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 may not execute because these routines may 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. We have not had any problem with this, although there is a known MIPS simulator bug (and fix) for programs that initialize global strings as arrays (e.g., char[] = "foo"). Even so, it is safest to keep your test program code to "least common denominator" C. If you suspect a bug in the cross-compiler or simulator, try to reproduce it in the smallest possible program. Please report any problems. Finding bugs is extra credit.
4. In the past, some students have excused their difficulties with Nachos test programs by informing us that they ''do not know C''. At this point in your career, you have written a large amount of code in C++. 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, dynamic strings, operator overloading, streams, *new*, or *delete*. In addition, a C compiler may not permit you to declare items in the middle of a procedure. If you write simple code ("least common denominator") and use the existing test programs as a starting point, then things will turn out OK. If you find that you are having problems because you don't know C, then learn it. In particular, be sure that you understand the relationships between strings, arrays, and pointers in C (and C++).

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 labs. 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 behavior of a working (but incomplete) thread system. In subsequent assignments you will extend this thread system, but for now you will merely use it "out of the box" to experience the joys of concurrent programming.

The first step is to understand how threads are 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 routines are quite similar to internal routines used for managing processes in real operating system kernels. For now, you are using these internal Nachos primitives to create simple concurrent (multi-threaded) programs as applications under Unix (e.g., Solaris). 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 is deposited in the **threads** subdirectory. Once you are in the **threads** subdirectory with a **nachos** executable, you can try a simple test by running this program, i.e., by typing **nachos** or **./nachos**).

If you examine **threads/main.cc**, you will see that the program is executing the *ThreadTest* function in **threadtest.cc**. *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. Your first goal is to understand the thread primitives used by this program, and to do some experiments to help you understand what really happens with multiple threads at runtime. To understand the execution path, trace through the code for the simple test case. See the notes in Section 3.3 for some tips on how to do this.

Your next goal is to show the many ways that concurrent code like this can break given a nondeterministic ordering of thread executions at runtime. This will 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. The assignment is to create your own variant of *ThreadTest* that starts *T* threads operating on a specific shared data structure: an unsynchronized list used as a sorted priority queue. All of the code for lists/queues and sorting is available in the Nachos release in the *List* class; make a copy of this class and call it *TestList*. By *unsynchronized* we mean that your *ThreadTest* and your *TestList* 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 are supposed to illustrate and experience in this assignment.)

Each of the *T* test threads in your new *ThreadTest* will generate *N* items with arbitrary keys and insert each item into a shared *TestList* in sorted order of the key. *T* and *N* should be settable from the command line (directions for doing this are available on the course web site). After inserting its *N* items, each thread should call a *TestList* remove primitive *N* times to remove *N* items from the front of the list. The "correct" or "expected" behavior is that each item inserted into the list is returned by the remove primitive exactly once, that every remove call returns a valid item, and that the list is empty when the last thread has fin-

ished. Since the list is sorted, each thread expects the items that it removes to come off in sorted order, eventhough they won't necessarily be the same items that thread put into the list (assuming *T > 1)*.

Once you have created your test program, the next step is to identify and illustrate all the kinds of incorrect or unexpected behaviors that can occur in this simple scenario. In your demos and writeups you will show us some of the difficulties caused by specific execution interleavings that could occur at runtime. The programming challenge is to instrument the test program with options and outputs that demonstrate the buggy behaviors. 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 some of those interleavings.

Here is a more detailed list of steps for Lab 1:

1. Create an unsynchronized *TestList* list class based on the *List* class in **threads/list.h** and **threads/list.cc**. Be aware that *List* is used by the Nachos thread system itself. This is why you must make a copy of it and leave the original unmodified. If you modify *List* directly then you may experience bizarre bugs that you do not understand. There is plenty of time for that later in the semester.
2. Create a test procedure analogous to the file **threadtest.cc** that makes calls on your list class. Make the changes to **threads/main.cc** so that it calls your new *ThreadTest* instead of the original one in **threadtest.cc**.
3. Modify your list class and your *ThreadTest* to force interleavings that illustrate interesting incorrect or unexpected behaviors. See the notes in Section 3.4 for a more complete discussion of interleavings and some ways of controlling them. You should be able to enumerate and demonstrate each scenario during the demo, using command line flags (Section 3.3.3), without recompiling your test program.
4. Think about the buggy behaviors you can demonstrate by changing the interleavings, and divide them into categories as you see fit. Your writeup should describe each category of bug, outline the interleavings that can cause it to occur, and explain how the interleaving caused the resulting behavior. Your treatment of the bugs should focus on the interleavings that are most "interesting", and avoid spending time describing or demonstrating behaviors that are substantially similar. The goal here is to show that you thoroughly understand the concurrent behaviors and can demonstrate them and explain them to us. We leave it to you to decide how best to achieve that goal.

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 3.3) to ensure that a retrieved item is non-null.

4.2 Lab 2: Threads and Synchronization

In this assignment, you extend the Nachos thread system by adding support for locks (mutexes), condition variables, and a useful coordination primitive called *EventBarrier* that combines elements of condition variables and semaphores. We will use the 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 Lab 1. *EventBarrier* is useful for high-level thread coordination in the later labs.

You are expected to understand the basic behaviors of interrupt enable/disable, locks/mutexes, condition variables, and semaphores from class and from the readings. If you don't understand them, study up on the lecture notes, then bug the instructor if it is not clear. You need to understand it before you start this lab.

Implementing Mutexes and Condition Variables (70 Points)

The public interface to mutexes and condition variables is defined in **synch.h**, which includes important comments on the semantics of the primitives. The condition variable interface is clunky in some respects, but please just accept it as defined by **synch.h**. Look at *SynchList* to see how the synchronization primitives for mutexes and condition variables are used.

Your first mission is to define private data for these classes in **synch.h** and implement the interfaces in **synch.cc**. 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. Each implementation is worth 35 points in this lab. Here are the specific steps in more detail:

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, you must disable interrupts before calling *Thread::Sleep*, to avoid the *missed wakeup* problem discussed in class. However, you may lose points for holding interrupts disabled when it is not necessary to do so. Disabling interrupts is a blunt instrument, and it should be avoided unless absolutely necessary. Be sure to understand why it is sometimes necessary to disable interrupts, and what the cost is.
2. Implement your locks and condition variables again using semaphores as the only synchronization primitive. This time it is not necessary (or permitted) to disable interrupts in your code: 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 subtle and difficult. In particular, your solution should guarantee that a *Signal* cannot affect a subsequent call to *Wait* as required by standard condition variable semantics.
3. Test both implementations of your locks and condition variables, and figure out how to show that they are correct during the demo. We may provide a test program to make your life easier. Check the lab notes on the course web.

Implementing *EventBarrier* (30 Points)

Your next mission is to use the Nachos synchronization primitives to create an *EventBarrier* class that allows a group of threads to wait for an event and respond to it in a synchronized fashion. Unlike locks,

condition variables, and semaphores, *EventBarrier* is a new primitive that we have defined just for this class, incorporating useful properties from both condition variables and semaphores. The semantics of EventBarrier are specified below. You are to implement these semantics correctly. If the specification is incomplete in some way then you are free to resolve the ambiguity as you see fit.

Threads may use an *EventBarrier* to notify one another of events or conditions. Threads may *Wait* for an event or *Signal* that an event has occurred. For example, weary traveling minstrels may *Wait* outside the gate of the castle for the drawbridge to come down; the gatekeeper calls *Signal* to wake up the minstrels when the drawbridge is down; the minstrels respond to the event by entering the castle and informing the gatekeeper of their arrival by invoking a new method called *Complete*. EventBarrier enables this synchronization in a way that lets any number of minstrels cross the bridge while it is down, and prevents the gatekeeper from raising the bridge while minstrels are still on it.

Like a binary semaphore, *EventBarrier* requires no external synchronization, and it keeps internal state to "remember" a previous signal; a *Wait* returns immediately if the event is already signaled (e.g., the bridge is down). Unlike condition variables or semaphores, a signaled *EventBarrier* stays in the signaled state until all threads (minstrels) that called *Wait* have responded to the signal (crossed), then it reverts to the unsignaled state (the bridge is up). To ensure that all signaled threads (minstrels) have an opportunity to respond, it holds back the signaling thread (gatekeeper) until the signaled threads (minstrels) have called *Complete* to notify the *EventBarrier* that they have finished responding to the event (crossed the bridge). This behavior makes *EventBarrier* a powerful primitive for thread coordination.

Here is the interface for *EventBarrier*:

***void** Wait() -- Wait until the event is signaled. Return immediately if already in the signaled state.*
***void** Signal() -- Signal the event and block until all threads that wait for this event have responded. The EventBarrier reverts to the unsignaled state before Signal() returns.*
***void** Complete() -- Indicate that the calling thread has finished responding to a signaled event, and block until all other threads that wait for this event have also responded.*
***int** Waiters() -- Return a count of threads that are waiting for the event or that have not yet responded to it.*

Note that despite its name, *EventBarrier::Signal* is more like *Condition::Broadcast* than it is like *Condition::Signal*, since *EventBarrier::Signal* wakes up *all* threads waiting for the event.

You may implement *EventBarrier* using any combination of mutexes, condition variables, and/or semaphores, but do not stoop to disabling interrupts. Test your *EventBarrier* implementation in whatever way you think best. Be sure your implementation correctly handles threads that call *Wait* while the *EventBarrier* object is in the signaled state; all participating threads must wait until the late arrival has responded to the event and called *Complete*. Also, your implementation should handle the case where threads call *Wait* again immediately after returning from *Complete*; these threads should block in *Wait* until the *EventBarrier* is signaled again. Each thread should process a given event at most once.

Some Notes for Lab 2

Your implementations of all the synchronization primitives should use *ASSERT* checks (see Section 3.3) to enforce any usage constraints necessary for correct behavior. For condition variables, every call to *Signal* and *Wait* passes a mutex as an argument. This is a clunky aspect of the interface as defined in Nachos, and it makes it possible for the calling threads to attempt to use a single condition variable with more than one mutex, which would be a fundamental violation of the rules of condition variables. Be sure you understand why this is illegal, and that your implementation catches this case with an *ASSERT*. Also, 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? What if a thread signals an *EventBarrier* while it is already in the signaled state, or calls *Complete* while the *EventBarrier* is in the unsignaled state? You should also treat these as examples of incorrect usage, and catch them with *ASSERT* checks. These *ASSERT* checks are worth points on this assignment, and they will save you headaches in later assignments. Consider other uses of ASSERT that catch internal errors as well as usage errors.

The handling of other usage issues is less clear. 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 implementations are correct (e.g., what will happen if we put a *yield* between lines X and Y), and to comment on their behavior (fairness, starvation, etc.) under various usage scenarios.

Use the **-s** debug flag to show what is going on and to help with debugging. However, there are no current *DEBUG* statements for the **s** debug flag, so you will need to add some to your code. See Section 3.3.

Warning: If you use the nachos *SynchList* class, be aware that it 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 definition of semaphores does not guarantee that a thread awakened in *V* gets a chance to run before another thread calls *P*. In particular, the Nachos implementation of semaphores does not guarantee this behavior: if you look at the implementation, you will see that *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*. In some interleavings another thread may call *P* first, consuming the count that was "meant for" the awakened thread and causing the awakened thread to go back to sleep and wait for another *V*. Incomplete understanding of this possibility is a frequent cause of programming errors with semaphores.

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. For this lab you will be using the synchronization facilities you implemented for the previous assignment. If your implementations were incorrect in any way, then you will need to fix the bugs first so they don't cause problems in this lab.

For this lab you will write new classes based on header files provided in the course directory (subdirectory **aux**) and/or on the course web site. 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. You will also need to add adequate tests for debugging and for use during the demo.

Problem 1: Multithreaded Table (15 points)

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 for 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.*
Return an error (-1) if no free table slots are available.
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 sure that your *Table* class is free from deadlock if used correctly. You are free to define for yourself what is "correct" usage, but be clear about your assumptions.

Problem 2: Alarm Clocks (20 points)

Implement an *AlarmClock* class for your Nachos kernel. Threads will call your *AlarmClock::Pause(int howLong)* to go to sleep for a period of time. The alarm clock can be implemented using the simulated *Timer* device (cf. **timer.h**). When the timer interrupt goes off, the *Timer* interrupt handler in **system.cc** must wake up any thread sleeping in *AlarmClock::Pause* whose interval has expired. There is no requirement that an awakened thread starts running immediately after the interval expires; just wake them up after they have waited for at least the specified interval (*howLong*). We have not created a header file for *AlarmClock*, so you may define the rest of the class interface as you see fit. You may use any convenient unit for *howLong*.

Warning: Do not change the behavior of the nachos timer "hardware" to implement *AlarmClock*. You may modify the timer interrupt handler, but do not modify the *Timer* class itself.

Warning: Nachos exits if there are no runnable threads, even if there is a thread waiting for an alarm. This is a "feature", i.e., some people think it is unreasonable behavior and view it as a bug. We apologize, but one tedious "Nachos nit" is that you must devise a way to prevent it from happening in this case. One quick-and-dirty hack/kludge/workaround is to fork a thread that yields in a tight loop forever.

Warning: It is never correct for an interrupt handler to sleep. Think about the effect of this constraint on your scheme for synchronization. Also, you should design your code to minimize the amount of time executing with interrupts disabled. This is one instance in which we will consider performance in assigning points.

Warning: Boxed Nachos does not deliver timer interrupts unless the **-rs** option is used. This is another "feature". 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**. Do not spend time on this: we will not deduct points if your tests depend on use of **-rs** or even if they "break" the implementation of **-rs**.

Problem 3: Bounded Buffer (25 points)

This is a classical synchronization problem called *bounded producer/consumer.* Implement a thread-safe *BoundedBuffer* class, based on the definitions in ***/aux/BoundedBuffer.h**. The semantics are defined below and will be discussed in class. As always, if the specification is incomplete you are free to resolve the ambiguity as you see fit. *BoundedBuffer* will be used in Lab 5 to implement pipes, an inter-process communication (IPC) mechanism fundamental to Unix systems.

Each pipe or *BoundedBuffer* is a flow-controlled channel for passing an ordered stream of bytes from one or more *producer* threads to one or more *consumer* threads. The producers call *Write* to push bytes into the stream. The consumers call *Read*, which extracts bytes from the buffer and copies them to memory specified by the consumer. Each byte written is delivered at most once, and bytes are delivered to the consumers in the order they were written by the producers. The channel is flow-controlled in the following sense. If producers generate data faster than the consumers consume it, then the buffer fills up, and any call to *Write* blocks until some of the bytes in the buffer are consumed, freeing up space in the buffer. If the consumers read data faster than the producers can write it, then the buffer empties, and any call to *Read* blocks until the producers write more bytes. By limiting the space (*maxsize*) reserved for the buffer, the system can automatically synchronize the speed of the producers and consumers. Thus *BoundedBuffer* and pipes are unified mechanisms for synchronization as well as communication.

*BoundedBuffer(**int** maxsize) -- Create a bounded buffer to hold at most **maxsize** bytes. The buffer is a storage area to hold bytes placed in the buffer by Write until they are removed by Read. Use a FIFO (first-in-first-out) policy to manage the buffer, i.e., Read returns bytes in the order that they were written.*

int** 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 starting at address **data**. Return the number of bytes successfully read.*

int** 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 starting at address **data**. Return the number of bytes successfully written.*

***void** Close () -- Permanently close the BoundedBuffer object. This releases any blocked readers or writers; any subsequent attempts to Read or Write the buffer do nothing and return 0.*

Synchronization for *BoundedBuffer* is a bit tricky. In particular, you should take care to preserve the atomicity of *Read* and *Write* requests when multiple producers or consumers share the same *BoundedBuffer*. That is, data written by a given *Write* should never be delivered to a reader interleaved with data from other *Write* operations. This invariant should hold even if writers and/or readers are forced to block because the buffer fills up or drains. Note further that any producer may also act as a consumer, and vice versa. You should also take care that no producer sleeps while there is space in the buffer for it, and no consumer sleeps while there are bytes in the buffer for it.

Hint: You may find it useful to base your implementation on the Nachos *SynchList* class.

Problem 4: Elevator (40 points)

Many teams find this problem difficult. Start early. Think before you code. Consider more than one possible framework for a solution, then pick the best one. Strive for a simple and elegant solution, even if it is less than optimal in terms of performance.

Implement an elevator controller for a building with *F* floors, using threads and a synchronization scheme of your choosing, using any combination of mutexes, condition variables, and/or semaphores. *Hint*: you may find *EventBarrier* (from Lab 2) useful here.

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". The riders are threads that use elevators to move around the building by calling methods of your *Elevator* and *Building* classes. You will need to add new internal methods and data items to these classes, but do not modify the signatures of the methods that are already defined.

The test driver routine for elevators should first create a single instance of *Building*, which will in turn create and start one *Elevator* object for each elevator in the building. Each elevator runs as a separate thread looping within the *Elevator* class, calling internal elevator control methods (*OpenDoors*, *CloseDoors*, *VisitFloor* and the *Building* methods) to serve rider requests in an orderly fashion.

After creating the building, the test program creates one or more riders, each of which runs as another thread. The rider threads use the elevator rider interface (*CallUp*, *CallDown*, *AwaitUp*, *AwaitDown*, *Enter*, *Exit*, *RequestFloor*) to request services from the elevator. These rider methods synchronize with the elevator threads using shared state in the *Elevator* class and in the *Building* class.

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 opening while the elevator is in transit, riders entering and exiting while the doors are closed, etc. In part 1 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 test program that demonstrates the operation of your elevator for multiple riders. Your program should be ‘‘well-behaved’’ in the following sense: any rider that calls the elevator for the up or down direction (using *CallUp* or *CallDown*) should then immediately wait for the elevator (using *AwaitUp* or *AwaitDown*) , then get on it (*Enter*), request a floor in the correct direction (*RequestFloor*), then get off (*Exit*) when *RequestFloor* returns, signifying arrival 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. In this case, your *Enter* primitive should return a failure status if a rider attempts to get on while the elevator is full.

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. For more extra credit, consider how your elevators might handle badly behaved riders (e.g., practical jokers who call the elevator and don’t wait for it, who get on but do not request a floor, or who do not exit at the requested floor).

4.4 Lab 4: Multiprogrammed Kernel

All of the code from Labs 1-3 has executed entirely “within the Nachos kernel”. In a real operating system, the kernel not only uses its procedures internally, but also executes *user programs* and allows them to invoke kernel facilities via *system calls*. An executing user program is a *process*. In this lab you will modify Nachos to support multiple processes, using system calls to request services from the kernel.

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. To this end, you will implement simple versions of key mechanisms found in real operating system kernels: virtual addressing, protected system calls, kernel exception handling, and preemptive timeslicing. Virtual addressing prevents user processes from accessing kernel data structures or the memory of other programs; your kernel will use process page tables to safely allow multiple processes to reside in memory at the same time. By now you should understand these concepts from class discussions.

With protected system calls and exceptions, all entries into the kernel funnel through a single kernel routine, *ExceptionHandler*. You will ''bullet-proof'' this routine so that buggy or malicious user programs cannot cause the kernel to crash or behave inappropriately. Your kernel will use preemptive scheduling to share the simulated CPU fairly among the active user processes, so that no process can take over the system. All of these protection mechanisms require cooperation between the hardware and the operating system kernel software. Your implementation will be based on "hardware" support in the Nachos MIPS simulator, which resembles a real MIPS processor. One aspect of this Lab is to understand how control transfers between the kernel and user programs running as processes on the machine simulator.

The key to Lab 4 is to implement the basic system calls for process management: the *Exec*, *Exit* and *Join* system calls described in Section 5. The kernel is responsible for managing the thread(s), virtual address space, and other resources for each process. New processes are created with *Exec*: once running as a process, a user program can invoke the *Exec* system call to create a new "child" process executing another user program — or a new instantiation of the same program. When the program is finished, the program may destroy its containing process by calling *Exit*. A parent process may call *Join* to wait for a child process to complete (e.g., to *Exit*).

If all processes are created by other processes, then who creates the first user process? The operating system kernel creates this process itself as part of its initialization sequence. This is *bootstrapping*. You can "boot" the Nachos kernel by running **nachos** with the **-x** option (**x** for "execute"), giving the name of a user program to run in the initial process. The Nachos release implements the **-x** option by calling *StartProcess* in **progtest.c** to handcraft the initial process and execute the initial program within it. The initial process may then create other processes, which may create other processes...and so on.

How to Get Started

This assignment is 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 threads, (2) the thread context switch code already saves and restores MIPS machine registers and the process page table, and (3) the Nachos distribution (*StartProcess*) 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 a few hundred lines of code. The difficult 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 to look at are in the **nachos/code/userprog** directory. Starting with this lab, you will build your **nachos** executables in the **userprog** subdirectory instead of in **threads**: be sure that you build and run the "right" **nachos**. Most of the heavy lifting for Labs 4 and 5 is rooted in **userprog/exception.cc** and **addrspace.cc**. The Nachos system call interface is defined in Section 5 and in **userprog/syscall.h**. The *StartProcess* code in **progtest.cc** is useful as a starting point for implementing *Exec*. Also, be sure to read Section 3.5 and the header file **machine/machine.h** that defines your kernel's interface to the simulated machine.

First, you will need basic facilities to load processes into the memory of the simulated machine. Spend a few minutes 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* works OK, but it assumes that there is only one program/process running at a time (started with *StartProcess* from *main* via the **nachos -x** option), and that all of the machine's memory is allocated to that process. Your first job is to generalize this code to implement the *Exec* system call for the general case in which multiple processes are active simultaneously:

1. Implement a memory manager module to allow your kernel to allocate page frames of the simulated machine's physical 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 "out of the box" *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 use your memory manager to allocate page frames for the new process, and load the process code and data into those allocated page frames, which often are not contiguous. Since this is a common source of bugs, check the lab notes on the course web site for a *LoadPage* primitive to make your life easier. This code is not guaranteed to work for you, but looking at it may help you to avoid some common errors which can be quite difficult to debug.
3. Set up each process page table correctly so that the contiguous process virtual address space maps to the noncontiguous physical memory.
4. If necessary, update *StartProcess* to use your new *AddrSpace* interface so that you do not break the **nachos -x** option.
5. Modify *AddrSpace* to call the memory manager to release the pages allocated to a process when the process is destroyed.

Note: What should your kernel do if there are not enough free page frames to load the entire process on an *Exec*? In Lab 6 you will add support for "juggling" to allocate physical memory dynamically, but for Lab 4 it is acceptable to fail the *Exec* and return an error (0). Make sure that your *AddrSpace* code releases any frames allocated to the process in this case. To cleanly handle these failures, you will need to move the

AddrSpace loading code out of the constructor and into a new *AddrSpace* method that can return an error. It is poor programming practice to put code that can fail into a class constructor, as the Nachos designers have done in this release.

Next, use these new facilities to implement the *Exec* and *Exit* system calls as defined in Section 5 and in **userprog/syscall.h**. If an executing user process requests a system call (by executing a trap instruction) the machine will transfer control to your kernel by calling *ExceptionHandler* in **exception.cc**. Your kernel code must extract the system call identifier and the arguments from the machine registers, decode them, and call internal procedures that implement the system call. This should be clear from class discussions. Here are some issues to attend to for implementing system calls in Nachos:

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. The filename string address (**char***) passed into the kernel as an argument is a process virtual address; in order for the kernel to access the filename it must locate the characters in the kernel address space (i.e., in the machine's physical "main memory" array) by examining the page table for the process. In particular, your kernel 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. Your kernel should 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.
4. 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 easier to kill a process from inside the kernel (e.g., if the process has some kind of fatal error), by calling the internal exit primitive from another kernel procedure (e.g., *ExceptionHandler*) in the target process context. In general, this kind of careful internal decomposition will save you from reinventing and redebugging wheels, and it is always good practice.

Next, implement the *Join* system call and other aspects of process management, extending your implementation of *Exec* and *Exit* as necessary.

1. Be sure you handle the argument and result of the *Join* system call correctly. The kernel *Join* primitive must validate any *SpaceId* passed to it by a user process. It must also validate that the calling process has privilege to *Join*; in this case, the caller must be the parent of the target process. Finally, your *Join* implementation must correctly return the exit status code of the target process.

2. 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.
3. 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 some process. Try to devise the simplest possible synchronization scheme for the code and data structures that manage process relationships and *Exit/Join*, even if your scheme is inefficient. One possibility might be to use broadcasts on a single condition variable shared by all processes in the system. Check the lab notes on the course web site for a precise definition of this interesting synchronization problem.

Last but not least, complete the bullet-proofing of your kernel. Implement the Nachos kernel code to handle user program exceptions that are not system calls. The machine (simulator) raises an exception whenever it is unable to execute the next user instruction, 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 whole system. **Note**: an *ASSERT* that crashes Nachos is a reasonable response to a bug within your Nachos kernel, but it not an acceptable response to a user program exception.

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

You should test your code by exercising the new system calls from user processes. To test your kernel, you will create some simple user programs as described in Section 3.6. Your Nachos kernel will execute user programs on a simulated MIPS machine as discussed in Section 3.5.

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 join on all children before exiting. Since *Exec* as defined provides no way to pass arguments into the new process, you may hard-code *M* and *N* into your test programs as constants, and you may use multiple versions of the programs with different constants encoded within them.
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. Since your kernel allows multiple processes to run at once, you have been careful to employ synchronization where needed inside your kernel. 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 call implementation. 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 calling the underlying host (e.g., Solaris) file system calls; thus your Nachos programs 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.

   *The **aux** subdirectory of the Duke Nachos repository includes 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 a common source of bugs. To use the console, you must modify the initialization code to create a *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 3.5. 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 user programs that exercise the system calls and trigger exceptions of different kinds. You do not need to exhaustively test all the types of exceptions.
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 all 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. **Extra credit**: extend your kernel to support more advanced shell features.
3. Test your kernel by using your shell to execute some test programs as concurrent processes. Designing and building creative test programs is an important part of this assignment. This aspect of Lab 5, and most elements of Lab 6, are deliberately left more vague than the earlier labs in order to encourage you to push your system and exercise your creativity and skills.

   One useful utility program is **cp**, which copies the contents of a file to a destination file. Unfortunately you will need to hardcode the source and destination filenames into the **cp** program, unless you extend your kernel to handle simple string arguments to *Exec* (for extra credit).

   You may find **cat** variants useful to demonstrate pipes and file I/O. Again, your kernel cannot support a real **cat** without support for *Exec* arguments, but you can write simple variants that (1) read a file and write it to standard output (*OpenFileId* 1), (2) read standard input (*OpenFileId* 0) and write it to a file, or (3) read standard input and write it to standard output. These may be concatenated into process pipelines of arbitrary length.

4.6 Lab 6: Virtual Memory

In this lab you will extend 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.

Virtual memory should be implemented and debugged in two steps. First, you will implement *demand paging* using page faults to dynamically load process virtual pages on demand, rather than initializing page frames for each process in advance at *Exec* time as you did in Labs 4 and 5. Next, you will implement *page replacement*, enabling your kernel to evict any virtual page from memory in order to free up a physical page frame to satisfy a page fault. Demand paging and page replacement together allow your kernel to "overbook" memory by executing more processes than would fit in machine memory at any one time, using page faults to "juggle" the available physical page frames among the larger number of process virtual pages. If it is implemented correctly, virtual memory is undetectable to user programs unless they monitor their own performance.

The operating system kernel works together with the machine's memory management unit (MMU) to support virtual memory. Coordination between the hardware and software centers on the page table structure for each process. You used page tables in Labs 4 and 5 to allow your kernel to assign any free page frame to any process page, while preserving the illusion of a contiguous memory for the process. The indirect memory addressing through page tables also isolates each process from bugs in other processes that are running concurrently. In Lab 6, you will extend your kernel's handling of the page tables to use three special bits in each page table entry (PTE):

- The kernel sets or clears 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 sets the *use* bit (reference bit) in the PTE to pass information to the kernel about page access patterns. If a virtual page is referenced by a process, the machine sets the corresponding PTE reference bit to inform the kernel that the page is active. Once set, the reference bit remains set until the kernel clears it.
- The machine sets the *dirty* bit in the PTE whenever a process executes a store (write) to the corresponding virtual page. This informs the kernel that the page is dirty; if the kernel evicts the page from memory, then it must first "clean" the page by preserving its contents on disk. Once set, the dirty bit remains set until the kernel clears it.

Implementing Demand Paging

In the first phase, you should preallocate a page frame for each virtual page of each newly created process at *Exec* time, just as in Labs 4 and 5. As before, return an error from the *Exec* system call if there are not enough free page frames to hold the process new address space. But now, *Exec* should initialize all the PTEs as *invalid*.

When the process references an invalid page, the machine will raise a page fault exception. Modify your exception handler to catch this exception and handle it by preparing the requested page on demand. This will likely require a restructuring of your *AddrSpace* initialization code. Faults on different address space segments are handled in different ways. 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, clear the exception by marking the PTE as valid, then restarting execution of the user program at the faulting instruction. When you return from the exception, be sure to leave the PC in a state that reexecutes the faulting instruction. If you set up the page and page table correctly, then the instruction will execute correctly and the process will continue on its way, none the wiser.

Test your demand paging implementation before moving on to page replacement. See the notes on testing below.

Implementing Page Replacement

In the second phase, your kernel delays allocation of physical page frames until a process actually references a virtual page that is not already loaded in memory.

First, complete the gutting of your code to create an address space: remove the code to allocate page frames and preinstall virtual-physical translations when setting up the page table. Instead, merely mark all the PTEs as invalid.

Next, extend your page fault exception handler to allocate a page frame on-the-fly when a page fault occurs. If memory is full, it will be necessary to free up a frame by selecting a victim page to evict from memory. To evict a page, the kernel marks the corresponding PTE(s) invalid, then frees the page frame and/or reallocates it to another virtual page. The system must be able to recreate the victim page contents if the victim page is referenced at a later time; if the page is dirty, the system must save the page contents in *backing store* or *swap space* on local disk or out on the network. An important part of this lab is to use the Nachos file system interface to allocate and manage the backing store. You will need routines to allocate space on backing store, locate pages on backing store, push pages from memory to backing store (for page-out), and pull from backing store to memory (for pagein). Be sure to clear the dirty bit when you mark a PTE for the victim page as invalid.

In this way, your operating system will use main memory as a cache over a slower and cheaper backing store. As with any caching system performance depends largely on the policy used to decide which pages are kept in memory and which to evict. As you know, we are not especially concerned about the performance of your Nachos implementations (simplicity and correctness are paramount), but in this case we want you to experiment with one of the page replacement policies discussed in class. Use FIFO or random replacement if you are short on time, but we will be more impressed if you implement an LRU approximation, examining and clearing the *use* bit in each PTE in order to gather information to drive the policy. We would like to see you implement a policy with an asynchronous paging daemon that responds to memory conditions, but we recognize that time may be short and we will not penalize you heavily if you do not. In any case, choose your policy carefully based on your group's design criteria (e.g. simplicity, performance, simplicity, correctness, fun) and be able to explain your choices.

Another important design question is the handling of dirty pages. Some rather poor kernels allow processes to fill memory with dirty pages. Consider what can happen when memory is full of dirty pages. If a page fault occurs, the kernel must find a victim frame to hold the incoming page, but every potential victim must be written to disk before its frame may be seized. Cleaning pages is an expensive operation, and it could even lead to deadlock in extreme cases, if for example a page is needed for buffering during the disk write. For these reasons, "real" operating systems retain a reserve of clean pages that can be grabbed quickly in a pinch. Maintaining such a reserve generally requires a paging daemon to aggressively clean dirty pages by pushing them to disk if the reserve begins to drain down. This makes correct management of the dirty bits more interesting. Again, implement the best scheme you can in the available time. This will make the experience (and the demos) more interesting and rewarding.

Warning. 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.

Testing and Demoing Your VM System

Lab 6 builds directly on Labs 4 and 5, so you will mostly be working with the same set of files. A useful test case is available in **test/matmult.c**. This program exercises the virtual memory system by multiplying two matrices. Of course, it is cheating to increase the size of main memory in the Nachos machine emulation, although you may want to decrease this number to stress your system for debugging (this counts as "reconfiguring" rather than "modifying" the hardware). 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 programs do anything useful (like multiplying matrices), but they should generate memory reference patterns of interest.

Your test programs should also demonstrate the 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). 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

This section defines the system call interface for your Nachos kernel.

Note on returning errors from system calls: One of the broken things about Nachos is that it does not provide a clean way to return system call errors to a user process. For example, Unix kernels return system call error codes in a designated register, and the system call stubs (e.g., in the standard C library or in **start.s**) move them into a program variable, e.g., the global variable *errno* for C programs. We are not bothering with this in Nachos. What is important is that you detect the error and reject the request with no bad side effects and without crashing the kernel. I recommend that you report errors by returning a 0 or -1 value where possible, instead of returning a value that could be interpreted as a valid result. If there is no clean way to notify the user process of a system call error it is acceptable to simply return from the call and just let the user process struggle forward.

5.1 Process Management

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

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

> Create a user process by creating a new address space, initializing it from the *executable* program file, and starting a new thread (via *Thread::Fork*) to run within 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 address space. 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*. Don't worry about this parameter for Lab 4; save it for Lab 5.

void Exit (int status)

A user process calls *Exit* to indicate that it is finished and ready to terminate. The user program may call *Exit* explicitly, or it may simply return from *main*, since the common runtime library routine (in **start.s**) 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*.

Note: if you are implementing threads (Section 5.4), then *Exit* destroys the calling thread rather than the entire process/address space. The process and its address space are destroyed only when the last thread calls *Exit*, or if one of its threads generates a fatal exception.

Warning: the *Exit* system call should never return; it should always destroy the calling thread. Returning from *Exit* may cause your Nachos system to mysteriously shut down. To see why, look at the startup instruction sequence in **test/start.s**.

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.