# Nachos Project Guide

G22.2250-001: Operating Systems
Fall 2007

Vijay Karamcheti

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

# Preface

This document describes the Nachos programming projects used for G22.2250, the one-semester M.S. Operating Systems course at New York University. Nachos is an instructional operating system, within which students can explore the key concepts that underlie most modern operating systems. Note that this is an introductory course, targeted towards students who may not have taken an operating systems course in their studies so far.

The document has its roots in the project guide created by Jeff Chase (chase@cs.duke.edu) for use in the introductory Operating Systems course at Duke University. I have made several modifications to the original document over the years, in some of the assignment descriptions, in formatting, and in making the policies, assignments, and document references suitable for use at New York University.

Vijay Karamcheti
August 15, 2007
New York

# 1 Nachos Project Policies and Mechanisms

In this course, we will have a Nachos lab project due every two weeks, starting two weeks after the first day of classes. The one exception is Lab 6, which is due three weeks after it is handed out. 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 3 (See Nachos Lab Assignments). You will find the information in Section 2 (See Working With Nachos) valuable for some or all of the assignments. The system call definitions in Section 4 (See Nachos System Call Interface) will be important for Labs 4 and 5. By the end of the semester you will have read everything in this document.

## 1.1 What to Hand In

Each of you is expected to turn in a completed lab, which represents work that you have performed on your own. To make this clearer, you can discuss with others the high-level approaches that you might take to solve a particular lab problem, but the actual coding and evaluation of this approach should be your own.

To turn in a lab:

1. Place the source code files for the lab in a *separate* sub-directory under your home directory on the department machines. Include only the files that have been modified from the original distribution provided to you.
2. Please e-mail a short writeup (1-2 pages) for each lab to the instructor before the deadline with the string **nachos project writeup N** in the subject line, where **N** is the lab number. The message should **cc:** the TA (if one exists). The first line of your message should be the full pathname of your source code directory. 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.

Since there will very likely be a delay from the time you turn-in a lab to the time we get a chance to examine your work, please make sure that you do not (accidentally or intentionally) update the source files in the turn-in directory.

For some of the labs, we may additionally require that you “demo” the work in front of the TA or instructor. These demo sessions will be announced in class, and held during office hours.

## 1.2 Grading

Your implementation will be graded on completeness, correctness, programming style, and thoroughness of testing. Bugs that were not uncovered by your testing will likely impact your grade more than bugs you are aware of and tell us about in advance. You should expect less than half-credit for projects that do not compile/build or do not run. Please make sure that your writeup documents whatever command-line options one may need to supply to run your program.

Note that in this course, labs will NOT be graded on the basis of how fast they execute. 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 will be graded on a 100-point scale. You should be pleased with any grade above an 80. Grades below 70 indicate that your effort is failing to meet expectation. If you are in trouble, please ask for help from the TA and/or the instructor, or stop by during office hours to talk about it with the instructor. The instructor or TA will send out e-mail to you with your grade at most a week after the turn-in date.

## 1.3 Late Work and Solutions

**Late work.** To ensure that students do not unduly put off working on the labs, a 20% grade penalty will be imposed for each week the lab is delayed after the due date. Get started early and schedule your time carefully.

**Availability of solutions.** In previous semesters that similar assignments have been used (at NYU or at other universities), 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 because ultimately that will help you learn the material better. Note that the labs build upon each other, so you do not really have the luxury to skip out on any of the labs without implementing at least some minimal functionality. We may revisit this policy mid-semester if we find that students are unable to make progress because of earlier labs.

## 1.4 What Parts of Nachos to 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 the guide attempts to identify 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 focus of each assignment.

In general, you should feel free to modify any part of Nachos that you feel is necessary to modify. However, 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, take extreme care should you choose to modify the behavior of the "hardware" as defined by the machine simulation software in the **machine** subdirectory described in Section 2.4 (See The Nachos MIPS Simulator). While 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), incorrect modifications to any code that implements the machine itself is likely to result in very long debugging cycles.

# 2 Working With Nachos

This section contains general information that will help you understand Nachos and complete the projects. A great deal of additional information about Nachos is also available through the *Nachos resource page*.

Before you do any Nachos work, you should be familiar with Section 1 (See Nachos Project Policies and Mechanisms), which defines the course procedures for all of the Nachos assignments. Section 3 (See Nachos Lab Assignments) gives specific instructions for each assignment.

## 2.1 Installing and Building Nachos

You will develop, test, and turn in your code on a department Solaris SPARC machine. The *Nachos resource page* on the course web includes an HTML source code browser and updated instructions for a full installation of the Nachos release we will use this semester. This web page also contains an appropriately modified version of Nachos that can be installed and run on a Linux/x86 machine. While you might find the latter convenient for development purposes, note that you will be expected to submit your lab on the department machines, so you should make sure that your code runs on the latter as well.

Install your Nachos copy into a directory of your choice under your home directory on the Solaris machine. Note that this directory is different from the sub-directory you use to hand in your labs. Make sure that you give us read access to the latter using the **chmod** command. 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 **make** command (make sure that you are using GNU make). The relevant subdirectories are **threads** for Labs 1-3, **userprog** for Labs 4-5, and **vm** for Lab 6. If you type **make** 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's Makefile (e.g., **nachos/threads/Makefile**) are created automatically using the **make depend** facility. For example, if you type **cd threads; make 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 will 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 **make 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** .

## 2.2 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.

While the first option may appear easy to use, spending the time learning how to use a debugger would be very worthwhile. Be aware that **printfs** may 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.

### 2.2.1 The DEBUG Primitive

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

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

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

### 2.2.2 Miscellaneous Debugging Tips

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

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 call 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 execution 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.

**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** .

### 2.2.3 Defining New Command-Line Flags for Nachos

In addition to defining new debug flags as described in Section 2.2.1 (See The Debug Primitive), 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.

## 2.3 Controlling the Order of Execution in Nachos

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

### 2.3.1 Context Switches

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

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

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

### 2.3.2 Voluntary Context Switches with Thread::Yield

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

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

### 2.3.3 Involuntary Context Switches with the -rs Flag

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

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

## 2.4 The Nachos MIPS Simulator

As discussed in class, true support for user programs and a protected kernel requires that we give each of you your own machine. This is because your kernel will need complete control over how memory is managed and how interrupts and exceptions (including system calls) are handled. Since we cannot afford to give you a real machine, we will give you a **simulated** machine that models a MIPS CPU. You will use the simulated MIPS machine in Labs 4-6 to execute test programs, as discussed in Section 2.5 (See Creating Test Programs for Nachos Kernels). Your nachos executable will contain a MIPS simulator that reads the test program executables as data and **interprets** them, simulating their execution on a real MIPS machine booted with your Nachos kernel.

### 2.4.1 The MIPS CPU Simulator

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

### 2.4.2 Interactions Between the Kernel and the Machine

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

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

### 2.4.3 I/O Devices and Interrupts

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

**Why does Nachos hang when I enable the console?** The current version of Nachos has an annoying ``feature'' that has caused problems in past versions of this course. 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 form to leave idle Nachos processes running, since they chew up a lot of CPU time.

## 2.5 Creating Test Programs for Nachos Kernels

In later assignments you will need to create (user-level) 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 2.4 (See The Nachos MIPS Simulator).

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 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 or Linux/x86 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 you understand what they do.) Finally, the programs are converted into a MIPS executable file format called NOFF, using the supplied program **coff2noff** .

To run the test programs, you must first build a Nachos kernel that supports user-mode programs. The raw Nachos release has skeletal support for running a single user program at a time; you will extend Nachos to support multiprogramming, virtual memory, and file system calls during the course of the semester. To build a Nachos kernel that can run user programs, edit your Nachos makefile to uncomment the ``cd userprog'' lines (if commented), then run **make** 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.

Nachos user programs invoke kernel-supplied functionality (e.g., to read files, do I/O, display something on the console, or read input) using Nachos system calls. You will provide the implementation for a number of these calls as part of the labs.

The Nachos distribution includes several sample test programs. For example, look at **test/halt.c** , which simply asks the operating system to shut the ``machine'' down using the Nachos **Halt** system call. Run the **halt** program with the command **nachos -x halt** , or **nachos -x ../test/halt** . It may be useful to trace the execution of the **halt** program using the debug flag described in Section 2.2 (See Tracing and Debugging Nachos Programs). The **test** directory includes other simple user programs to test your kernels in the later assignments. For these exercises we also expect you to extend these tests and add some of your own.

### 2.5.1 Troubleshooting Test Programs

The following guidelines will help you avoid trouble when building new test programs:

1. Don't mess around with the build process. Place your source code in the **test** directory, and extend the existing **Makefile** to build them.
2. Recognize that your Nachos test programs are extremely limited in what they can do. In particular, their only means of interacting with the outside world is to request services from your Nachos kernel. For example, your test programs cannot call **printf, malloc** or any other C library routine. If you attempt to call these routines, your program will fail to link. If you somehow succeed in linking library routines into your executable image, the resulting program will not execute because these routines will use Unix system calls, which are not recognized by your Nachos kernel.
3. The Nachos distribution includes a warning that global variables may not work correctly. It is safest to avoid the use of global variables in your Nachos test programs.
4. As graduate students in the Honors section, we expect you to pick up on your own the small amount of C and C++ required by this course. Just don't try anything fancy: unlike C++, C does not have classes, operator overloading, streams, **new** , or **delete** . In addition, a C compiler may not permit you to declare items in the middle of a procedure, which is poor programming practice anyway. If you follow these guidelines and use the existing test programs as a starting point, then you should be OK.

# 3 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 1 (See Nachos Project Policies and Mechanisms), which presents the policies and procedures that apply to all of the Nachos assignments. You will find the information in Section 2 (See Working With Nachos) valuable for some or all of the assignments. The system call definitions in Section 4 (See Nachos System Call Interface) are important for Labs 4 and 5.

## 3.1 Lab 1: The Trouble with Concurrent Programming

This assignment consists of two parts. In the first part, you will write, compile, and run a simple C++ program to create, and add and remove elements from a priority-sorted doubly linked list. In the second part, you are to use this program to become familiar with Nachos and the code of a working (but incomplete) thread system. In subsequent assignments, you will this thread system, but for now, you will merely use what is supplied to experience the joys of concurrent programming.

### 3.1.1 Priority Sorted Doubly Linked List (20 points)

Write a C++ program to implement a doubly-linked list based on the following definitions:

```
class DLLElement {
public:
  DLLElement( void *itemPtr, int sortKey );     // initialize a list element

  DLLElement *next;           // next element on list
                              // NULL if this is the last
  DLLElement *prev;           // previous element on list
                              // NULL if this is the first

  int key;                    // priority, for a sorted list
  void *item;                 // pointer to item on the list
};

class DLList {
public:
  DLList();                   // initialize the list
  ~DLList();                  // de-allocate the list

  void Prepend(void *item);   // add to head of list (set key = min_key-1)
  void Append(void *item);    // add to tail of list (set key = max_key+1)
  void *Remove(int *keyPtr);  // remove from head of list
                              // set *keyPtr to key of the removed item
                              // return item (or NULL if list is empty)

  bool IsEmpty();             // return true if list has elements

  // routines to put/get items on/off list in order (sorted by key)
  void SortedInsert(void *item, int sortKey);
  void *SortedRemove(int sortKey);  // remove first item with key==sortKey
                                    // return NULL if no such item exists

private:
  DLLElement *first;          // head of the list, NULL if empty
  DLLElement *last;           // last element of the list, NULL if empty
};
```

The doubly linked list should keep items sorted according to an integer key (for insert operations that don't take a key argument, assign key values that are consistent for the operation). You should write your program so that its code is contained in three files: **dllist.h, dllist.cc, dllist-driver.cc.** The first two files should provide the definitions and implementations of the two classes above, and the third file should contain two functions, one of which generates **N** items with random keys (or to aid debugging, you might control the input sequence with a more carefully selected key order) and inserts them into a doubly-linked list, and the other which removes **N** items starting from the head of the list and prints out the removed items to the console. Both functions should take as arguments the integer **N** and a pointer to the list.

To verify that you have indeed implemented the classes and the driver functions correctly, create a separate file containing the main function of your program. In this function, first allocate the list, and then make calls to the driver functions above, passing in appropriate values for the arguments. The behavior you need to demonstrate is that your remove function removes exactly the items you have inserted in sorted order. You should also do other tests to verify that your implementation does implement a doubly linked list.

Although you can write, compile, and run the above program on any platform, I would prefer that you do your development either on the Solaris SPARC platform or the Linux/x86 platform using **gcc** to make sure that your program can interact with Nachos in the second part of this assignment.

### 3.1.2 Nachos Familiarity and Understanding its Thread System (80 points)

In this part of the assignment, you will understand how the Nachos thread system (whose functionality you will extend in subsequent labs) works. In general, Nachos thread primitives are only used internal to the Nachos operating system kernel, never directly by user programs; in fact, these primitives are quite similar to internal primitives used for managing processes in real operating system kernels. However, to understand how they work, in this one lab (and in Lab 3), we will use the thread primitives directly to run simple concurrent programs as applications under Unix (Solaris). If you find this confusing at this point, do not worry about it.

Build a **nachos** executable using the **make** command. Run **make** (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 run a simple test of Nachos, by entering the command **nachos** (if that doesn't work, try **./nachos** ).

If you examine **threads/main.cc** , you will see that you are 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 in a process. 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 2.2 (See Tracing and Debugging Nachos Programs) 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 non-deterministic ordering of thread executions at runtime. By exposing you to some of the pitfalls up front, when you expect them, 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 accessing a specific shared data structure: the unsynchronized priority-sorted doubly linked list you implemented in the first part of the assignment. By **unsynchronized** we mean that your **ThreadTest** and your list implementation do not use semaphores, mutexes, interrupt disable, or other synchronization mechanisms that you will learn about later in the semester. The purpose of these mechanisms is to prevent the problems that you are supposed to illustrate and experience in this assignment.

Each of your test threads will call the two driver functions in **dllist-driver.cc** (first inserting the items and then removing the items). Note that you should have created the list in your variant of the **ThreadTest** function before starting the threads. Both **T** and **N** should be settable from the command line (directions for doing this are available on the course web site). 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 finished. Since the list is sorted, each thread expects the removed items to come off in sorted order, even though they won't necessarily be the same items that thread put into the list if **T > 1**.

Once you have written your test program, your job is to identify and illustrate all the kinds of incorrect or unexpected behaviors that can occur in this simple scenario. In your 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 and your doubly linked list implementation 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 because of some of those interleavings. See the notes in Section 2.3 (See Controlling the Order of Execution in Nachos) for a more complete discussion of interleavings and some ways of controlling them.

Specifically, this part of the assignment requires you to perform the following steps:

1. Copy your dllist.h, dllist.cc, and dllist-driver.cc files into the threads subdirectory. Modify the definitions of **THREAD_H** and **THREAD_C** in **Makefile.common** in the root **nachos** directory to include these files and update the makefile dependencies. This ensures that your files are also compiled and linked against the **nachos** distribution.
2. Create a driver file analogous to the file **threadtest.cc** that makes calls on your **DLL**ist class using the functions in **dllist-driver.cc** Make the changes to **threads/main.cc** so that an execution of the **nachos** command causes the function in the new driver file to be executed instead of the function **ThreadTest** in the file **threadtest.cc.**
3. Modify your **DLList** class and your **ThreadTest** to force interleavings that illustrate interesting incorrect or unexpected behaviors. You should be able to enumerate and demonstrate each scenario during the demo, using command line flags as described in Section 2.2.3 ( See Defining New Command-Line Flags for Nachos), without recompiling your test program.
4. Think about the buggy behaviors you can demonstrate, and divide them into categories. 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. Note that you may see more interesting behaviors if different threads exhibit different interleaving behaviors.

   Your treatment of the bugs should be thorough but it need not be exhaustive. Try to 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 understand the concurrent behaviors, not to create a lot of busy work for you or for us.

## 3.2 Lab 2: Threads and Synchronization

In this assignment, you will complete the Nachos thread system by adding support for locks (mutexes), condition variables, and build some synchronized data structures for use in the later labs.

As with Lab 1, this lab also requires you to introduce one or more files of your own. As before, you will need to add these files to proper macro definitions in the makefile, and update the makefile dependencies. Note that the latter half of this assignment requires you to use the synchronization facilities you will be implementing in the first part. So, please try and make sure that your implementation of locks and condition variables is thoroughly debugged before you move on to the rest of the assignment.

Your new classes will be 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.

### 3.2.1 Implementing Mutexes and Condition Variables (60 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** . Your first mission is to define private data for these classes in **synch.h** and implement the interfaces in **synch.cc** . Look at **SynchList** to see how the synchronization primitives for mutexes and condition variables are used. You are to write two different implementations of these synchronization primitives in two different versions of the **synch.h** and **synch.cc** files. You should be able to switch from one version to the other by (at worst) moving these files around and recompiling Nachos. Each implementation is worth 30 points in this lab.

Here are the specific steps and requirements 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 particular, **Thread::Sleep** requires you to disable interrupts before you call it. However, you may lose points for holding interrupts disabled when it is not necessary to do so. Disabling interrupts is a blunt instrument and should be avoided unless necessary.
2. Implement your locks and condition variables 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 for condition variables should guarantee that a **Signal** cannot affect a subsequent **Wait**.
3. Modify your DLList class from Lab 1 so that it uses synchronization primitives to ensure that the list is being updated consistently despite its use by multiple threads. You may want to take a look at the **SynchList** class (in **threads/synchlist.h** and **threads/synchlist.cc** ) to see how you might do this. To demonstrate that your code works, as in Lab 1, create a driver file analogous to the file **threadtest.cc** that makes calls on the synchronized version of the DLList class. Make the changes

to **threads/main.cc** so that an execution of the **nachos** command causes the function in the new driver file to be executed instead of the function **ThreadTest** in the file **threadtest.cc** . Use the synchronized version of **DLList** to test both versions of your locks and condition variables.

### 3.2.2 Implementing a Multithreaded Table (20 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 of processes, threads, memory page frames, open files, etc. Table has the following methods, defined in the header file **Table.h**, which is provided in the course directory (subdirectory **aux**):

*Table(int size) -- Create a table to hold at most size entries.*
*int Alloc (void* object) -- Allocate a table slot for object, returning index of the allocated entry.*
*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.*

### 3.2.3 Implementing a Bounded Buffer (20 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** .

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

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

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

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

### 3.2.4 Some Notes for Lab 2

Your implementations of locks and condition variables should use **ASSERT** checks as described in Section 2.2 (See Tracing and Debugging Nachos Programs) to enforce any usage constraints necessary for correct behavior. For example, every call to **Signal** and **Wait** passes an associated mutex; what could go wrong if a given condition variable is used with more than one mutex? What will happen if a lock holder attempts to acquire a held lock a second time? What if a thread tries to release a lock that it does not hold? These **ASSERT** checks are worth points on this assignment, and they will save you headaches in later assignments.

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

You should be able to explain why your implementation is correct (e.g., what will happen if we put a yield between lines X and Y), and to comment on its behavior (fairness, starvation, etc.) under various usage scenarios.

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

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

**Note**: for debugging, you may use the **-s** debug flag. However, there are no current **DEBUG** statements with the **s** debug flag, so you will need to add some to your code. See Section 2.2 (See Tracing and Debugging Nachos Programs).

## 3.3 Lab 3: Programming with Threads

Your mission in this assignment is to use the Nachos thread system to implement higher-level synchronization primitives, and to use these primitives to solve a synchronization problem. The objective of the assignment is to improve your skill in writing correct concurrent programs and dealing with the now-familiar pitfalls: race conditions, deadlock, starvation, and so on.

You will be using the synchronization facilities you implemented as part of the previous assignment, so before you can proceed, you will need to fix them if your implementation was broken.

### 3.3.1 Implementing an EventBarrier Primitive (25 points)

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. **EventBarrier** incorporates useful properties from both condition variables and semaphores. Like a condition variable, an **EventBarrier** is associated with an event (or condition); threads may wait for the event or signal that the event has occurred. 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. Unlike condition variables or semaphores, a signaled **EventBarrier** stays in the signaled state until all threads have responded to the signal, then it reverts to the unsignaled state. To ensure that all threads have an opportunity to respond, it creates a "barrier" that holds back the signaling thread and responding threads until all participating threads have finished responding to the event. 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 when 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.

### 3.3.2 Implementing an Alarm Clock Primitive (25 points)

Implement an **AlarmClock** class. Threads call **Alarm::Pause(int howLong)** to go to sleep for a period of time. The alarm clock can be implemented using the 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

**Alarm::Pause** whose interval has expired. There is no requirement that an awakened thread starts running immediately after the interval expires; just put them on the ready queue after they have waited for at least the specified interval ( **howLong** ). We have not created a header file for **Alarm** , 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 timer hardware to implement alarms. You may modify the timer interrupt handler, but do not modify the timer class itself. It is also not acceptable for you to create a new (simulated) hardware timer device for your implementation.

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

**Warning** : It is never correct for an interrupt handler to sleep. Think about the effect of this constraint on your scheme for synchronization. In addition, you should design your code to minimize the amount of work that must be done at interrupt time.

**Warning** : Nachos by default does not deliver timer interrupts unless the **-rs** option is used. You may want to tweak the initialization code to enable this functionality.

### 3.3.3 Synchronization Problem 1: Bridge (50 points)

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

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

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

Write the procedures **ArriveBridge** and **ExitBridge** (the **CrossBridge** procedure should just print out a debug message), using any synchronization scheme of your choosing involving mutexes, condition variables, semaphores, and/or event barriers. **ArriveBridge** must not return until it is safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions or a bridge collapse). **ExitBridge** is called to indicate that the caller has finished crossing the bridge and should take steps to let additional cars cross the bridge. To test your program, create some number of vehicles, which repeatedly use the bridge in different directions: use your **Alarm** clock implementation to model the non-bridge activity of a vehicle.

You should attempt to make your solution **fair** and **starvation free**. In order to understand what this involves, try answering the following question about your scheme: if a car arrives while traffic is currently moving in its direction of travel across the bridge, but there is another car already waiting to cross in the opposite direction, will the new arrival cross *before* the car waiting on the other side, *after* the car on the other side, or is it impossible to say?

## 3.4 Lab 4: Multiprogrammed Kernel

Up until now, all of your test programs have executed entirely within the Nachos kernel. In a real operating system, the kernel not only uses its procedures internally, but allows **user programs** to access some of its routines via system calls. An executing user program is a **process**. In this lab, you will modify Nachos to support execution of 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 and 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. With protected system calls and exceptions, all attempts by a user program to enter the kernel funnel through a single kernel routine, the **ExceptionHandler** routine; your kernel will “bullet-proof” this routine so that buggy or malicious user programs cannot cause the kernel to crash or behave inappropriately. Finally, 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.

The key to this lab is to implement the system calls for process management: the **Exec** , **Exit** and **Join** system calls. New processes are created with **Exec** : once running as a process, a user program can invoke the **Exec** system call to create new "child" processes executing other user programs -- or more instantiations of the same program. When the program is finished, it may destroy the containing process by calling **Exit** . A parent process may call **Join** to wait for a child process to complete.

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 an initial program to run as 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.

### 3.4.1 Details of Lab 4

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

Most of the new files that you will be using are in the **nachos/code/userprog** directory. You will build your **nachos** executables in **userprog** instead of **threads** : be sure you build and run the "right" **nachos** . Even before you make any changes, you can build a Nachos executable that allows you to run a single user program, however, there is only skeletal support for exceptions or system calls. Most of the heavy lifting for Labs 4 and 5 is rooted in **userprog/exception.cc** and **addrspace.cc.** The Nachos system call interface is defined in Section 4 (See Nachos System Call Interface) 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 The Nachos MIPS Simulator 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, and some sort of process table to keep track of active processes. Spend some time studying the **AddrSpace** class, and look at how the **StartProcess** procedure uses the **AddrSpace** class methods to create a new process, initialize its memory from an executable file, and start the calling thread running user code in the new process context. The current code for **AddrSpace** and **StartProcess** works OK, but it assumes that there is only one program/process running at a time (started with **StartProcess** from main by the **nachos -x** option), and that all of the machine's memory is available to that process. Your first job is to generalize the code for **StartProcess** and **AddrSpace**, and use it to implement the **Exec** system call.

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

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

Next, use these new facilities to implement the **Exec** and **Exit** system calls as defined in **userprog/syscall.h** (also see Section 4, See Nachos System Call Interface). If an executing user processes requests a system call (by executing a trap instruction) the machine will transfer control to your kernel by calling **ExceptionHandler** in **exception.cc**. Your kernel code must extract the system call identifier and the arguments, decode them, and call internal procedures that implement the system call. *In particular, your system call code must convert user-space addresses to Nachos machine addresses or kernel addresses before they can be dereferenced*. Here are a couple of issues you need to attend to:

1. When an **Exec** call returns, your kernel should have created a new process and started a new thread executing within it to run the specified program. However, you do not need to concern yourself with setting up **OpenFileIds** until the next assignment. For now, you will be able to run user programs, but they will not be able to read any input or write any output.

2. For **Exec** , you must copy the filename argument from user memory into kernel memory safely, so that a malicious or buggy user process cannot crash your kernel or violate security. 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, you must handle the case where the filename string crosses user page boundaries and resides in noncontiguous physical memory. You must also detect an illegal string address or a string that runs off the end of the user's address space without a terminating null character, and handle these cases by returning an error ( **SpaceId** 0) from the **Exec** system call.

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

3. **Exec** must return a unique process identifier ( **SpaceId** ), which can be used as an argument to Join, as discussed in Section 4.1 (See Process Management). 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 easy to "force" a process to exit from inside of 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 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 a given exiting process.

4. 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.

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

Finally, 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 2.5 (See Creating Test Programs for Nachos Kernels). These test programs will execute on the simulated MIPS machine within Nachos as discussed in Section 2.4 (See The Nachos MIPS Simulator).

1. Write a test program(s) to create a tree of processes **M** levels deep by calling **Exec** **N** times from each parent process, and joining on all children before exiting. You may hard-code **M** and **N** in your test programs as constants, since **Exec** as defined provides no way to pass arguments into the new program.
2. Create a modified version of the tree test program in which each parent process exits without joining on its children. The purpose of this program is to (1) test with larger numbers of processes, and (2) test your kernel's code for cleaning up process state in the no-join case.
3. 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.

## 3.5 Lab 5: I/O

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

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

Lab 5 includes the following requirements and options.

1. Implement, test, and debug the **Create** , **Open** and **Close** system calls. Be sure to correctly handle the case where a user program passes an illegal string to **Open** / **Create** or an illegal **OpenFileId** to **Close.** Recall the discussion of similar concerns for **Exec** and **Join** in Section 3.4 (See Lab 4: Multiprogrammed Kernel).

   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.

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

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

   **Warning**: Once you create a **Console** object, you may be annoyed that **nachos** no longer shuts down when there is nothing to do, as discussed in Section 2.4 (See The Nachos MIPS Simulator). 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 4.3 (See Pipes). Your **BoundedBuffer** class from Lab 3 may be useful here.

Spend some time devising test programs to exercise your kernel:

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

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

   Note that your shell implementation will be considerably simplified if you extend your **Exec** implementation to support command-line arguments. The strategy for supporting this is intentionally being left unspecified to see what kinds of creative solutions you folks can come up with. Note however that whatever your strategy, you will have to write your own string parsing and comparison utilities using primitive memory read/write operations (there is no libc for user programs that you can link against).
3. Test your kernel by using your shell to execute some test programs as concurrent processes. Adequate testing is a critical aspect of any software project, and designing and building creative test programs is an important part of this assignment.

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

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

## 3.6 Lab 6: Virtual Memory

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

The assignment consists of two parts. In Part 1, you will implement **demand paging** using page faults to dynamically load your program's pages on demand, rather than initializing page frames for the process at **Exec** time. 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.

### 3.6.1 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 uninitialized 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 in Section 3.6.3 (See Testing Your VM System).

### 3.6.2 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 pageout), 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.

**(Extra Credit)** 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.

### 3.6.3 Testing Your VM System

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

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

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

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). Again the test cases can generate different kinds of locality (good and bad) to see how the paging system reacts. Can you get the system to **thrash**, for example? Have you built in the kinds of monitoring and debugging that lets you see if it is thrashing? Try reducing the amount of physical memory to ensure more page replacement activity.

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

# 4 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 -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 let the user process struggle forward.

## 4.1 Process Management

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

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

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

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

The **pipectrl** parameter is discussed in Section 4.3 (See Pipes). User programs not using pipes should always pass zero in **pipectrl** .

***void Exit (int status)***

A user process calls **Exit** to indicate that it is finished executing. The user program may call **Exit** explicitly, or it may simply return from **main** , since the common runtime library routine (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 as discussed in Section 4.4 ( See Threads), 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 mysterious traps to the **Halt** system call. 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.

## 4.2 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.

## 4.3 Pipes

The **Exec** system call includes a **pipectrl** argument as defined in Section 4.1 (See Process Management). 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.