A Road Map Through Nachos

Thomas Narten
Department of Computer Sciences
Levine Science Research Center
Duke University
Box 90129
Durham, N.C. 27708-0129
narten@cs.duke.edu

January 4, 1995

Abstract

Nachos is instructional software that allows students to examine, modify and execute operating system software. Nachos provides a skeletal operating system that supports threads, user-level processes, virtual memory and interrupt-driven input output devices. Nachos is a complex piece of software and it is difficult for beginning students (and instructors) to easily gain on overall understanding of the various system pieces and how they fit together.

This document provides a road map to understanding the Nachos system. It gives a high-level overview of the source code, focusing on the big picture rather than on the details. It is not intended as a replacement for reading the source code. Rather, it is a companion that is intended to help students (and instructors) overcome the initial learning curve encountered when learning and using the system.

Contents

1 Introduction to Nachos **1**

2 Nachos Machine **1**

2.1 Machine Components . . . 2

2.2 Interrupt Management . . . 4

2.3 Real-Time Clock Interrupts . . . 5

2.4 Address Translation . . . 5

2.4.1 Linear Page Tables . . . 6

2.4.2 Software Managed TLB . . . 6

2.5 Console Device . . . 6

2.6 Disk Device . . . 7

3 Nachos Threads **9**

3.1 Mechanics of Thread Switching . . . 12

3.2 Threads & Scheduling . . . 13

3.3 Synchronization and Mutual Exclusion . . . 14

3.4 Special Notes . . . 14

4 User-Level Processes **15**

4.1 Process Creation . . . 15

4.2 Creating a Noff Binary . . . 16

4.3 System Calls and Exception Handling . . . 16

4.4 Execution Trace of User-Level Process . . . 17

5 Nachos Filesystem **19**

5.1 SynchDisk . . . 19

5.2 FileSystem Object . . . 21

5.3 OpenFile Object . . . 24

5.4 File System Physical Representation . . . 24

5.4.1 File Header . . . 24

5.4.2 Directories . . . 25

5.4.3 Putting It All Together . . . 26

6 Experience With Nachos Assignments **27**

6.1 General Tips 27
6.2 Synchronization 28
6.3 Multiprogramming 29
6.4 Virtual Memory 30
6.5 File System 34
6.6 Common Errors 34

7 MIPS Architecture **35**

1 Introduction to Nachos

Nachos is instructional software that allows students to study and modify a real operating system. The only difference between Nachos and a "real" operating system is that Nachos runs as a single Unix process, whereas real operating systems run on bare machines. However, Nachos simulates the general low-level facilities of typical machines, including interrupts, virtual memory and interrupt-driven device I/O.

The rest of this document attempts to provide a road map through Nachos. It is not intended to replace the need for reading the source code; rather, it this document attempts to speed up the learning process by describing "the big picture."

Section 2 provides an overview of the underlying machine that Nachos simulates and runs on top of. Section **??** describes Nachos threads and the mechanics of scheduling, synchronization and thread switching. Section 4 describes how user-level programs execute as separate processes within their own private address spaces. Section 5 provides an overview of the filesystem implementation. Section 6 reports on experience using Nachos to teach operating systems courses, and provides specific suggestions on individual assignments.

2 Nachos Machine

Nachos simulates a machine that roughly approximates the MIPS architecture. The machine has registers, memory and a cpu. In addition, an event-driven simulated clock provides a mechanism to schedule interrupts and execute them at a later time. The simulated MIPS machine can execute arbitrary programs. One simply loads instructions into the machine's memory, initializes registers (including the program counter PCReg) and then tells the machine to start executing instructions. The machine then fetches the instruction PCReg points at, decodes it, and executes it. The process is repeated indefinitely, until an illegal operation is performed or a hardware interrupt is generated. When a trap or interrupt takes place, execution of MIPS instructions is suspended, and a Nachos interrupt service routine is invoked to deal with the condition.

Conceptually, Nachos has two modes of execution, one of which is the MIPS simulator. Nachos executes user-level processes by loading them into the simulator's memory, initializing the simulator's registers and then running the simulator. User-programs can only access the memory associated with the simulated machine. The second mode corresponds to the Nachos "kernel." The kernel executes when Nachos first starts up, or when a user-program executes an instruction that causes a hardware trap (e.g., illegal instruction, page fault, system call, etc.). In "kernel mode", Nachos executes the way normal Unix processes execute. That is, the statements corresponding to the Nachos source code are executed, and the memory accessed corresponds to the memory assigned to Nachos variables.

Nachos does not have to execute user-level programs in order to perform useful things. Nachos supports kernel threads, allowing multiple threads to execute concurrently. In this context, Nachos behaves in a manner analogous to other thread packages. Indeed, user-level processes are executed by having a kernel thread invoke the simulator. Thus, multiprogramming makes use of multiple threads; each user-level process has a Nachos kernel thread associated with it to provide a context for executing the MIPS simulator.

2.1 Machine Components

The Nachos/MIPS machine is implemented by the *Machine* object, an instance of which is created when Nachos first starts up. The *Machine* object exports a number of operations and public variables that the Nachos kernel accesses directly. In the following, we describe some of the important variables of the *Machine* object; describing their role helps explain what the simulated hardware does.

The Nachos *Machine* object provides registers, physical memory, virtual memory support as well as operations to run the machine or examine its current state. When Nachos first starts up, it creates an instance of the *Machine* object and makes it available through the global variable *machine*. The following public variables are accessible to the Nachos kernel:

registers: An array of 40 registers, which include such special registers as a stack pointer, a double register for multiplication results, a program counter, a next program counter (for branch delays), a register target for delayed loads, a value to be loaded on a delayed load, and the bad virtual address after a translation fault. The registers are number 0–39; see the file *machine.h* for symbolic names for the registers having special meaning (e.g., *PCReg*).

Although registers can be accessed directly via *machine->registers[x]*, the *Machine* object provides special *ReadRegister()* and *WriteRegister()* routines for this purpose (described in more detail below).

mainMemory: Memory is byte-addressable and organized into 128-byte pages, the same size as disk sectors. Memory corresponding to physical address x can be accessed in Nachos at *machine->mainMemory[x]*. By default, the Nachos MIPS machine has 31 pages of physical memory. The actual number of pages used is controlled by the *NumPhysPages* variable in *machine.h*.

Virtual Memory Nachos supports VM through either a single linear page table or a software-managed TLB (though not simultaneously). The choice of which is in effect is controlled by initializing the *tlb* or *pageTable* variables of the *machine* class. When executing instructions, the *Machine* object uses whichever is defined, after verifying that they are not both set simultaneously.

At this point, we know enough about the *Machine* object to explain how it executes arbitrary user programs. First, we load the program's instructions into the machine's physical memory (e.g, the *machine->mainMemory* variable). Next, we initialize the machine's page tables and registers. Finally we invoke *machine->Run()*, which begins the fetch-execute cycle for the machine.

The *Machine* object provides the following operations:

Machine(bool debug) The *Machine* constructor takes a single argument *debug*. When *debug* is TRUE, the MIPS simulator executes instructions in single step mode, invoking the debugger after each instruction is executed. The debugger allows one to interactively examine machine state to verify (for instance) that registers or memory contain expected values.

By default, single-stepping is disabled. It is enabled by specifying the "-s" command line option when starting Nachos up.

ExceptionType Translate(int virtAddr, int* physAddr, int size, bool writing) converts virtual address *virtAddr* into its corresponding physical address *physAddr*. *Translate* examines the machine's translation tables (described in detail in Section 2.4) in order to perform the translation. When successful, *Translate* returns the corresponding physical address in *physAddr*. Otherwise, it returns a code indicating the reason for the failure (e.g., page fault, protection violation, etc.) Whenever a translation fails, the MIPS simulator invokes the Nachos routine *RaiseException* to deal with the problem. *RaiseException* is responsible for handling all hardware trap conditions. When *RaiseException* returns, the Nachos *Machine* assumes that the condition has been corrected an resumes its fetch-execute cycle.

Note that from a user-level process's perspective, traps take place in the same way as if the program were executing on a bare machine; a trap handler is invoked to deal with the problem. However, from the Nachos perspective, *RaiseException* is called via a normal procedure call by the MIPS simulator.

OneInstruction() does the actual work of executing an instruction. It fetches the current instruction address from the PC register, fetches it from memory, decodes it, and finally executes it. Any addresses referenced as part of the fetch/execute cycle (including the instruction address given by PCReg) are translated into physical addresses via the *Translate()* routine before physical memory is actually accessed.

Run() "turns on" the MIPS machine, initiating the fetch-execute cycle. This routine should only be called after machine registers and memory have been properly initialized. It simply enters an infinite fetch-execute loop.

The main loop in *Run* does three things: 1) it invokes *OneInstruction* to actually execute one instruction, 2) it invokes the debugger, if the user has requested single-step mode on the command line, and 3) it increments a simulated clock after each instruction. The clock, which is used to simulate interrupts, is discussed in the following section.

int ReadRegister(int num) fetches the value stored in register *num*.

void WriteRegister(int num, int value) places *value* into register *num*.

bool ReadMem(int addr, int size, int* value) Retrieves 1, 2, or 4 bytes of memory at virtual address *addr*. Note that *addr* is the virtual address of the **currently executing user-level program**; *ReadMem* invokes *Translate* before it accesses physical memory.

One point that should be noted is that *ReadMem* fails (returning FALSE), if the address translation fails (for whatever reason). Thus, if the page is not present in physical memory, *ReadMem* fails. *ReadMem* does **not** distinguish temporary failures (e.g., page not in memory) from hard errors (e.g., invalid virtual address)[1].

ReadMem is used (for instance) when dereferencing arguments to system calls.

bool WriteMem(int addr, int size, int value) writes 1, 2, or 4 bytes of *value* into memory at virtual address *addr*. The same warnings given for *ReadMem* apply here as well.

2.2 Interrupt Management

Nachos simulates interrupts by maintaining an event queue together with a simulated clock. As the clock ticks, the event queue is examined to find events scheduled to take place now. The clock is maintained entirely in software and ticks under the following conditions:

1. Every time interrupts are restored (and the restored interrupt mask has interrupts enabled), the clock advances one tick. Nachos code frequently disables and restores interrupts for mutual exclusion purposes by making explicit calls to *interrupt::SetLevel()*.
2. Whenever the MIPS simulator executes one instruction, the clock advances one tick.
3. Whenever the ready list is empty, the clock advances however many ticks are needed to fast-forward the current time to that of the next scheduled event.

Whenever the clock advances, the event queue is examined and any pending interrupt events are serviced by invoking the procedure associated with the timer event (e.g., the interrupt service routine). All interrupt service routines are run with interrupts disabled, and the interrupt service routine may not re-enable them.

Warning: in interrupt handler may **not** call any routines that lead to a context switch of the current thread (e.g., *scheduler::Run()* or *SWITCH()*). Doing so may lead to deadlock. This restriction is an artifact of the way interrupts are simulated under Nachos, and should not be taken as an indication of the way things are done on real machines. Specifically, consider the case where multiple events happen to be scheduled at *exactly* the same time. If the handler for the first event invokes *sleep* (which calls *SWITCH*), the others won't be serviced at the right time. In fact, the thread that called *sleep* may actually be waiting for one of the other events that is supposed to take place now, but is delayed because of the *SWITCH*. We now have a deadlock[2].

All routines related to interrupt management are provided by the *Interrupt* object. The main routines of interest include:

[1] See Section 1 for a description of one common problem students encounter due to this.

[2] To correctly implement preemption, the interrupt handler invoked when a running threads quantum expires needs to switch to another thread. This is handled by having the interrupt service routine invoke *Thread::YieldOnReturn()*, which delays the actual preemption until it is safe to do so.

void Schedule(VoidFunctionPtr handler, int arg, int when, IntType type) schedules a future event to take place at time *when*. When it is time for the scheduled event to take place, Nachos calls the routine *handler* with the single argument *arg*.

IntStatus SetLevel(IntStatus level) Change the interrupt mask to *level*, returning the previous value. This routine is used to temporarily disable and re-enable interrupts for mutual exclusion purposes. Only two interrupt levels are supported: *IntOn* and *IntOff*.

OneTick() advances the clock one tick and services any pending requests (by calling *CheckIfDue*). It is called from *machine::Run()* after each user-level instruction is executed, as well as by *SetLevel* when the interrupts are restored.

bool CheckIfDue(bool advanceClock) examines the event queue for events that need servicing now. If it finds any, it services them. It is invoked in such places as *OneTick*.

Idle() "advances" to the clock to the time of the next scheduled event. It is called by the scheduler (actually *Sleep()*) when there are no more threads on the ready list and we want to "fast-forward" the time.

2.3 Real-Time Clock Interrupts

Nachos provides a *Timer* object that simulates a real time clock, generating interrupts at regular intervals. It is implemented using the same event driven interrupt mechanism described above. *Timer* supports the following operations:

Timer(VoidFunctionPtr timerHandler, int callArg, bool doRandom) The *Timer* constructor creates a real-time clock that interrupts every *TimerTicks* (100) time units. When the timer goes off, the Nachos simulator invokes procedure *timerHandler*, passing it *callArg* as an argument.

To add a bit of non-determinism to the system, argument *doRandom* specifies that the time between interrupts should be taken from a uniform interval between 1 and $2 \times TimerTicks$.

The real-time clock can be used to provide preemption.

Note that starting Nachos with the "-rs" option creates a timer object that interrupts at random intervals and preempts the currently running thread.

2.4 Address Translation

Nachos supports two types of VM architectures: linear page tables, or a software managed TLB. While the former is simpler to program, the latter more closely corresponds to what current machines support. Nachos supports one or the other, but not both (simultaneously).

2.4.1 Linear Page Tables

With linear tables, the MMU splits a virtual address into page number and page offset components. The page number is used to index into an array of page table entries. The actual physical address is the concatenation of the page frame number in the page table entry and the page offset of the virtual address.

To use linear page tables, one simply initializes variable *machine->pageTable* to point to the page table used to perform translations. In general, each user process will have its own private page table. Thus, a process switch requires updating the *pageTable* variable. In a real machine, *pageTable* would correspond to a special register that would be saved and restored as part of the *SWITCH()* operation. The machine variable *pageTableSize* indicates the actual size of the page table.

Page table entries consist of the physical page frame number for the corresponding virtual page, a flag indicating whether the entry is currently valid (set by the OS, inspected by hardware), a flag indicating whether the page may be written (set by OS, inspected by hardware), a bit indicating whether the page has been referenced (set by the hardware, inspected and cleared by OS) and a dirty bit (set by hardware, inspected and cleared by OS).

The Nachos machine has *NumPhysPages* of physical memory starting at location *mainMemory*. Thus, page 0 starts at *machine->mainMemory*, while page N starts at $mainMemory + N \times PageSize$.

2.4.2 Software Managed TLB

[To be filled in.]

2.5 Console Device

Nachos provides a terminal console device and a single disk device. Nachos devices are accessed through low-level primitives that simply initiate an I/O operation. The operation itself is performed later, with an "operation complete" interrupt notifying Nachos when the operation has completed.

The *Console* class simulates the behavior of a character-oriented CRT device. Data can be written to the device one character at a time through the *PutChar()* routine. When a character has successfully been transmitted, a "transmit complete" interrupt takes place and the user-supplied handler is invoked. The interrupt handler presumably checks if more characters are waiting to be output, invoking *PutChar* again if appropriate.

Likewise, input characters arrive one-at-a-time. When a new character arrives, the console device generates an interrupt and the user-supplied input interrupt service routine is invoked to retrieve the character from the device and (presumably) place it into a buffer from which higher-level routines (e.g., *GetChar()*) can retrieve it later.

The *Console* object supports the following operations:

Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail, VoidFunctionPtr writeDone, int callArg) The constructor creates an instance of a terminal console. Argument *readFile* contains the Unix file name of where the data is to be read from; if NULL, standard input is assumed. Likewise, argument *writeFile* indicates where output written to the console is to go; if NULL, standard output is assumed. When a character becomes available for reading, *readAvail* is invoked with an argument of *callArg* to notify the Nachos that a character is available. The character itself is retrieved by calling *Console::GetChar()*. Upon return, it is assumed that the character has been retrieved and when the next one arrives, *readAvail* will be called again.

void PutChar(char ch) Writes character *ch* to the output device. Once output has started, it is an error to invoke *PutChar()* again *before* the corresponding I/O complete interrupt has taken place. Once the console device has written the character to the device, it invokes the user-supplied procedure *writeDone* , passing it *callArg* as an argument.

char GetChar() Retrieves a character from the console. *GetChar* returns EOF if no new data is available. Normally, the user would not invoke *GetChar* unless the availability of new data had first been signalled via the *readAvail()* interrupt service routine.

void CheckCharAvail() an internal procedure used to see if new data is available for reading.

When a console device is created by the constructor, the appropriate Unix files (or stdin/stdout) are opened and a timer event is scheduled to take place 100 time units in the future. When the timer expires, the routine *CheckCharAvail* is invoked to see if any data is present. If so, *CheckCharAvail* reads that character and invokes the user-supplied input interrupt handler *readAvail.* It then schedules a new timer event so that the process repeats every 100 time units. Thus, *CheckCharAvail* simply polls every 100 clock ticks for new data, calling the interrupt service routine whenever data is present for processing.

Device output is initiated by calling *PutChar*, giving it a single character to output. Once character output has been initiated, the device is made busy until the output complete interrupt takes place. *PutChar* simply outputs the one character, sets an internal flag to indicate that the device is busy, and then schedules a timer interrupt to take place 100 clock ticks later. When the timer expires, the state of the device is changed from busy to idle, and the user-supplied output interrupt complete routine is invoked. This routine would presumably invoke *PutChar* if additional output characters were queued awaiting output.

2.6 Disk Device

The *Disk* object simulates the behavior of a real disk. The disk has only a single platter, with multiple tracks containing individual sectors. Each track contains the same number of sectors, and blocks are uniquely identified by their sector number. As with a real disk, the OS initiates operations to read or write a specific sector, and a later interrupt indicates when the operation has actually completed. The Nachos disk allows only one pending operation at a time; the OS may initiate new operations only when the device is idle. Note that it is the responsibility of the OS to insure that new requests are not issued while the disk is busy servicing an earlier request.

In order to simulate typical delays in accessing a disk, the Nachos *Disk* object dynamically varies the time between the initiation of an I/O operation and its corresponding I/O complete interrupt. The actual delay depends on how long it takes to move the disk head from its previous location to the new track, as well as the rotational delay encountered waiting for the desired block to rotate under the read/write head.

The simulated disk contains *NumTracks* (32) tracks, each containing *SectorsPerTrack* (32) sectors. Individual sectors are *SectorSize* (128) bytes in size. In addition, *Disk* contains a "track buffer" cache. Immediately after seeking to a new track, the disk starts reading sectors, placing them in the track buffer. That way, a subsequent read request may find the data already in the cache reducing access latency.

The *Disk* object supports the following operations:

Disk(char *name, VoidFunctionPtr callWhenDone, int callArg): This constructor assumes that the simulated disk is kept in the Unix file called *name*. If the file does not already exist, Nachos creates it and writes a "magic number" of 0x456789ab into the initial four bytes. The presence of a magic number allows Nachos to distinguish a file containing a Nachos simulated disk from one containing something else. Finally, Nachos insures that the rest of the file contains NULL sectors. All Nachos disks have the same size, given by the formula $NumSectors \times SectorsPerTrack$.

If the file already exists, Nachos reads the first 4 bytes to verify that they contain the expected Nachos "magic number," terminating if the check fails. Note that by default the contents of a Nachos disk is preserved across multiple Nachos sessions, allowing users to create a Nachos file in one session, and read it in another. However, if the disk contains a file system, and the file system is left in a corrupted state by a previous Nachos session, subsequent Nachos invocations are likely run into problems if they don't first verify that the filesystem data structures are logically consistent.

The last two constructor arguments are used to provide an "I/O complete" interrupt mechanism. Specifically, the Nachos machine signals the completion of a Disk operation (e.g., read or write) by invoking the procedure *callWhenDone*, passing it an argument of *callArg*. As shown below, the *SynchDisk* object uses this routine to wake up a thread that has been suspended while waiting for I/O to complete.

ReadRequest(int sectorNumber, char *data): Is invoked to read the specified sector number into the buffer *data*. In Nachos, all sectors are the same size (*SectorSize*).

Note that this operations returns immediately, *before* the transfer actually takes place. *ReadRequest* schedules an interrupt to take place sometime in the future, after a time roughly dependent on the seek distance needed to complete the operation. Only after the interrupt takes place is it correct to start using the data.

WriteRequest(int sectorNumber, char *data): Similar to *ReadRequest*, except that it writes a single sector.

ComputeLatency(int newSector, bool writing): estimates the latency required to access the block *newSector* given the current position of the disk head. The routine is used in deciding when to schedule an I/O complete interrupt when servicing a read or write request.

3 Nachos Threads

In Nachos (and many systems) a *process* consists of:

1. An *address space*. The address space includes all the memory the process is allowed to reference. In some systems, two or more processes may share part of an address space, but in traditional systems the contents of an address space is private to that process. The address space is further broken down into 1) Executable code (e.g., the program's instructions), 2) Stack space for local variables and 3) Heap space for global variables and dynamically allocated memory (e.g., such as obtained by the Unix *malloc* or C++ *new* operator). In Unix, heap space is further broken down into BSS (contains variables initialized to 0) and DATA sections (initialized variables and other constants).

2. A single thread of control, e.g., the CPU executes instructions sequentially within the process.

3. Other objects, such as open file descriptors.

That is, a process consists of a program, its data and all the state information (memory, registers, program counter, open files, etc.) associated with it.

It is sometimes useful to allow multiple threads of control to execute concurrently within a single process. These individual threads of control are called *threads.* By default, processes have only a single thread associated with them, though it may be useful to have several. All the threads of a particular process share the same address space. In contrast, one generally thinks of processes as not sharing any of their address space with other processes. Specifically, threads (like processes) have code, memory and other resources associated with them. Although threads share many objects with other threads of that process, threads have their own *private* local stack[3].

One big difference between threads and processes is that global variables are shared among all threads. Because threads execute concurrently with other threads, they must worry about synchronization and mutual exclusion when accessing shared memory.

Nachos provides threads. Nachos threads execute and share the same code (the Nachos source code) and share the same global variables.

The Nachos scheduler maintains a data structure called a *ready list*, which keeps track of the threads that are ready to execute. Threads on the ready list are ready to execute and

[3]Actually, threads technically do share their stacks with other threads in the sense that a particular thread's stack will still be addressable by the other threads. In general, however, it is more useful to think of the stack as private, since each thread must have its own stack.

can be selected for executing by the scheduler at any time. Each thread has an associated *state* describing what the thread is currently doing. Nachos' threads are in one of four states:

READY: The thread is eligible to use the CPU (e.g, it's on the ready list), but another thread happens to be running. When the scheduler selects a thread for execution, it removes it from the ready list and changes its state from READY to RUNNING. Only threads in the READY state should be found on the ready list.

RUNNING: The thread is currently running. Only one thread can be in the RUNNING state at a time. In Nachos, the global variable *currentThread* always points to the currently running thread.

BLOCKED: The thread is blocked waiting for some external event; it cannot execute until that event takes place. Specifically, the thread has put itself to sleep via *Thread::Sleep()*. It may be waiting on a condition variable, semaphore, etc. By definition, a blocked thread does not reside on the ready list.

JUST_CREATED: The thread exists, but has no stack yet. This state is a temporary state used during thread creation. The *Thread* constructor creates a thread, whereas *Thread::Fork()* actually turns the thread into one that the CPU can execute (e.g., by placing it on the ready list).

In non-object oriented systems, operating systems maintain a data structure called a *process table*. Process (thread) table entries contain all the information associated with a process (e.g., saved register contents, current state, etc.). The process table information is frequently called a *context block*.

In contrast to other systems, Nachos does not maintain an explicit process table. Instead, information associated with thread is maintained as (usually) private data of a *Thread* object instance. Thus, where a conventional operating system keeps thread information centralized in a single table, Nachos scatters its "thread table entries" all around memory; to get at a specific thread's information, a pointer to the thread instance is needed.

The Nachos *Thread* object supports the following operations:

Thread *Thread(char *debugName) The *Thread* constructor does only minimal initialization. The thread's status is set to JUST_CREATED, its stack is initialized to NULL, its given the name *debugName*, etc.

Fork(VoidFunctionPtr func, int arg) does the interesting work of thread creation, turning a thread into one that the CPU can schedule and execute.

Argument *func* is the address of a procedure where execution is to begin when the thread starts executing. Argument *arg* is a an argument that should be passed to the new thread. (Of course, procedure *func* must expect a single argument to be passed to it if it is to access the supplied argument.)

Fork allocates stack space for the new thread, initializes the registers (by saving the initial value's in the thread's context block), etc.

One important detail must be considered. What should happen when procedure *func* returns? Since *func* was not called as a regular procedure call, there is no place for it to return to. Indeed, rather than returning, the thread running *func* should terminate. *Fork* takes care of this detail by building an initial activation record that makes this happen (described in detail below).

void StackAllocate(VoidFunctionPtr func, int arg) This routine does the dirty work of allocating the stack and creating an initial activation record that causes execution to appear to begin in *func*. The details are a somewhat complicated. Specifically, *StackAllocate* does the following:

1. Allocate memory for the stack. The default stack size is *StackSize* (4096) 4-byte integers.
2. Place a sentinel value at the top of the allocated stack. Whenever it switches to a new thread, the scheduler verifies that the sentinel value of the thread being switched out has not changed, as might happen if a thread overflows its stack during execution.
3. Initialize the program counter PC to point to the routine *ThreadRoot*. Instead of beginning execution at the user-supplied routine, execution actually begins in routine *ThreadRoot*. *ThreadRoot* does the following:
   (a) Calls an initialization routine that simply enables interrupts.
   (b) Calls the user-supplied function, passing it the supplied argument.
   (c) Calls *thread::Finish()*, to terminate the thread.

   Having thread execution begin in *ThreadRoot* rather than in the user-supplied routine makes it straightforward to terminate the thread when it finishes. The code for *ThreadRoot* is written in assembly language and is found in *switch.s* Note: *ThreadRoot* isn't run by the thread that calls *Fork()*. The newly created thread executes the instructions in *ThreadRoot* when it is scheduled and starts execution.

void Yield() Suspend the calling thread and select a new one for execution (by calling Scheduler::FindNextToRun()). If no other threads are ready to execute, continue running the current thread.

void Sleep() Suspend the current thread, change its state to BLOCKED, and remove it from the ready list. If the ready list is empty, invoke *interrupt->Idle()* to wait for the next interrupt. *Sleep* is called when the current thread needs to be blocked until some future event takes place. It is the responsibility of this "future event" to wake up the blocked thread and move it back on to the ready list.

void Finish() Terminate the currently running thread. In particular, return such data structures as the stack to the system, etc. Note that it is not physically possible for a currently running thread to terminate itself properly. As the current thread executes, it makes use of its stack. How can we free the stack while the thread is still using it? The solution is to have a *different* thread actually deallocate the data structures of a terminated thread. Thus, *Finish* sets the global variable *threadToBeDestroyed* to point to the current thread, but does not actually terminate it. *Finish* then calls *Sleep*, which effectively terminates the thread (e.g., it will never run again). Later, when the scheduler starts running another thread, the newly scheduled thread examines the *threadToBeDestroyed* variable and finishes the job.

3.1 Mechanics of Thread Switching

Switching the CPU from one thread to another involves suspending the current thread, saving its state (e.g., registers), and then restoring the state of the thread being switched to. The thread switch actually completes at the moment a new program counter is loaded into PC; at that point, the CPU is no longer executing the thread switching code, it is executing code associated with the new thread.

The routine *Switch(oldThread, nextThread)* actually performs a thread switch. *Switch* saves all of *oldThread*'s state (*oldThread* is the thread that is executing when *Switch* is called), so that it can resume executing the thread later, without the thread knowing it was suspended. *Switch* does the following:

1. Save all registers in *oldThread*'s *context block*.

2. What address should we save for the PC? That is, when we later resume running the just-suspended thread, where do we want it to continue execution? We want execution to resume as if the call to *Switch()* had returned via the normal procedure call mechanism. Specifically, we want to resume execution at the instruction immediately following the call to *Switch()*. Thus, instead of saving the current PC, we place the return address (found on the stack in the thread's activation record) in the thread's context block. When the thread is resumed later, the resuming address loaded into the PC will be the instruction immediately following the "call" instruction that invoked *Switch()* earlier.

   Note: It is crucial that *Switch()* appear to be a regular procedure call to whoever calls it. That way, threads may call *Switch()* whenever they want. The call will appear to return just like a normal procedure call *except* that the return does not take place right away. Only after the scheduler decides to resume execution of the switched thread will it run again.

3. Once the current thread's state has been saved, load new values into the registers from the context block of the next thread.

4. At what exact point has a context switch taken place? When the current PC is replaced by the saved PC found in the process table. Once the saved PC is loaded, *Switch()* is no longer executing; we are now executing instructions associated with the new thread, which should be the instruction immediately following the call to *Switch()*. As soon as the new PC is loaded, a context switch has taken place.

The routine *Switch()* is written in assembly language because it is a machine-depended routine. It has to manipulate registers, look into the thread's stack, etc.

Note: After returning from *Switch*, the previous thread is no longer running. Thread *nextThread* is running now. But because it also called *Switch()* previously, it will return to the "right place" (the instruction after the call to *Switch()*). Thus, it looks like *Switch()* is a "normal" procedure call, but in fact, a thread switch has taken place.

3.2 Threads & Scheduling

Threads that are ready to run are kept on the *ready list*. A process is in the READY state only if it has all the resources it needs, other than the CPU. Processes blocked waiting for I/O, memory, etc. are generally stored in a queue associated with the resource being waited on.

The *scheduler* decides which thread to run next. The scheduler is invoked whenever the current thread wishes to give up the CPU. For example, the current thread may have initiated an I/O operation and must wait for it to complete before executing further. Alternatively, Nachos may preempt the current thread in order to prevent one thread from monopolizing the CPU.

The Nachos scheduling policy is simple: threads reside on a single, unprioritized ready list, and threads are selected in a round-robin fashion. That is, threads are always appended to the end of the ready list, and the scheduler always selects the thread at the front of the list.

Scheduling is handled by routines in the *Scheduler* object:

void ReadyToRun(Thread *thread): Make *thread* ready to run and place it on the ready list. Note that *ReadyToRun* doesn't actually start running the thread; it simply changes its state to READY and places it on the ready list. The thread won't start executing until later, when the scheduler chooses it.

ReadyToRun is invoked, for example, by *Thread::Fork()* after a new thread has been created.

Thread *FindNextToRun(): Select a ready thread and return it). *FindNextToRun* simply returns the thread at the front of the ready list.

void Run(Thread *nextThread): Do the dirty work of suspending the current thread and switching to the new one. Note that it is the currently running thread that calls *Run()*. A thread calls this routine when it no longer wishes to execute.

Run() does the following:

1. Before actually switching to the new thread, check to see if the current thread overflowed its stack. This is done by placing a sentinel value at the top of the stack when

the thread is initially created. If the running thread ever overflows its stack, the sentinel value will be overwritten, changing its value. By checking for the sentinel value every time we switch threads, we can catch threads overflowing their stacks.

2. Change the state of newly selected thread to RUNNING. Nachos assumes that the calling routine (e.g. the current thread) has already changed its state to something else, (READY, BLOCKED, etc.) before calling *Run()*.

3. Actually switch to the next thread by invoking *Switch()*. After *Switch* returns, we are now executing as the new thread. Note, however, that because the thread being switched to previously called *Switch* from *Run()*, execution continues in *Run()* at the statement immediately following the call to *Switch*.

4. If the previous thread is terminating itself (as indicated by the *threadToBeDestroyed* variable), kill it now (after *Switch()*). As described in Section 3, threads cannot terminate themselves directly; another thread must do so. It is important to understand that it is actually another thread that physically terminates the one that called *Finish()*.

3.3 Synchronization and Mutual Exclusion

Low-level Nachos routines (including the ones discussed above) frequently disable and re-enable interrupts to achieve mutual exclusion (e.g., by calling *Interrupt::SetLevel()*).

Synchronization facilities are provided through semaphores. The *Semaphore* object provides the following operations:

Semaphore(char* debugName, int initialValue) The constructor creates a new counting semaphore having an initial value of *initialValue*. The string *debugName* is also associated with the semaphore to simplify debugging.

void P() Decrement the semaphore's count, blocking the caller if the count is zero.

void V() Increment the semaphore's count, releasing one thread if any are blocked waiting on the count.

3.4 Special Notes

When Nachos first begins executing, it is executing as a single Unix process. Nachos turns this single user process into a single Nachos thread. Thus, by default, Nachos executes a single thread. When the Nachos entry point routine *main* returns, that thread exits as well. However, if other threads have been created and continue to exist, the Unix process continues executing Nachos. Only after all threads have terminated does the Unix Nachos process exit.

4 User-Level Processes

Nachos runs user programs in their own private address space. Nachos can run any MIPS binary, assuming that it restricts itself to only making system calls that Nachos understands. In Unix, "a.out" files are stored in "coff" format. Nachos requires that executables be in the simpler "Noff" format. To convert binaries of one format to the other, use the *coff2noff* program. Consult the *Makefile* in the *test* directory for details.

Noff-format files consist of four parts. The first part, the Noff header, describes the contents of the rest of the file, giving information about the program's instructions, initialized variables and uninitialized variables.

The Noff header resides at the very start of the file and contains pointers to the remaining sections. Specifically, the Noff header contains:

noffMagic A reserved "magic" number that indicates that the file is in Noff format. The magic number is stored in the first four bytes of the file. Before attempting to execute a user-program, Nachos checks the magic number to be sure that the file about to be executed is actually a Nachos executable.

For each of the remaining sections, Nachos maintains the following information:

virtualAddr What virtual address that segment begins at (normally zero).

inFileAddr Pointer within the Noff file where that section actually begins (so that Nachos can read it into memory before execution begins).

size The size (in bytes) of that segment.

When executing a program, Nachos creates an address space and copies the contents of the instruction and initialized variable segments into the address space. Note that the uninitialized variable section does not need to be read from the file. Since it is defined to contain all zeros, Nachos simply allocates memory for it within the address space of the Nachos process and zeros it out.

4.1 Process Creation

Nachos processes are formed by creating an address space, allocating physical memory for the address space, loading the contents of the executable into physical memory, initializing registers and address translation tables, and then invoking *machine::Run()* to start execution. *Run()* simply "turns on" the simulated MIPS machine, having it enter an infinite loop that executes instructions one at a time).

Stock Nachos assumes that only a single user program exists at a given time. Thus, when an address space is created, Nachos assumes that no one else is using physical memory and simply zeros out all of physical memory (e.g., the *mainMemory* character array). Nachos then reads the binary into physical memory starting at location *mainMemory* and initializes the translation tables to do a one-to-one mapping between virtual and physical addresses (e.g., so that any virtual address N maps directly into the physical address N). Initialization of registers consists of zeroing them all out, setting *PCReg* and *NextPCReg* to 0 and 4 respectively, and setting the stackpointer to the largest virtual address of the process (the stack grows downward towards the heap and text). Nachos assumes that execution of user-programs begins at the first instruction in the text segment (e.g., virtual address 0).

When support for multiple user processes has been added, two other Nachos routines are necessary for process switching. Whenever the current processes is suspended (e.g., preempted or put to sleep), the scheduler invokes the routine *AddrSpace::SaveUserState()*, in order to properly save address-space related state that the low-level thread switching routines do not know about. This becomes necessary when using virtual memory; when switching from one process to another, a new set of address translation tables needs to be loaded. The Nachos scheduler calls *SaveUserState()* whenever it is about to preempt one thread and switch to another. Likewise, before switching to a new thread, the Nachos scheduler invokes *AddrSpace::RestoreUserState*. *RestoreUserState()* insures that the proper address translation tables are loaded before execution resumes.

4.2 Creating a Noff Binary

Nachos is capable of executing a program containing arbitrary MIPS instructions. For example, C programs in the *test* directory are compiled using *gcc* on a MIPS machine to create ".o" files. To create an *a.out* binary file, the loader prepends the instructions in *test/start.s* before the code of the user program. File *start.s* contains initialization code that needs to be executed before the user's main program. Specifically, the very first instruction in *start.s* calls the user-supplied *main* routine, whereas the second instruction invokes the Nachos *Exit* system call, insuring that user processes terminate properly when their main program returns. In addition, *start.s* contains stub modules for invoking system calls (described below).

4.3 System Calls and Exception Handling

User programs invoke system calls by executing the MIPS "syscall" instruction, which generates a hardware trap into the Nachos kernel. The Nachos/MIPS simulator implements traps by invoking the Routine *RaiseException()*, passing it a arguments indicating the exact cause of the trap. *RaiseException*, in turn, calls *ExceptionHandler* to take care of the specific problem. *ExceptionHandler* is passed a single argument indicating the precise cause of the trap.

The "syscall" instruction indicates a system call is requested, but doesn't indicate *which* system call to perform. By convention, user programs place the code indicating the particular system call desired in register *r2* before executing the "syscall" instruction. Additional arguments to the system call (when appropriate) can be found in registers *r4-r7*, following the standard C procedure call linkage conventions. Function (and system call) return values are expected to be in register *r2* on return.

Warning: When accessing user memory from within the exception handler (or within Nachos in general), user-level addresses cannot be referenced directly. Recall that user-level processes execute in their own private address spaces, which the kernel cannot reference

```
#include "syscall.h"
int
main()
{
    Halt();
    /* not reached */
}
```

Figure 1: Source for the program *halt.c*

directly. Attempts to dereference pointers passed as arguments to system calls will likely lead to problems (e.g., segmentation faults) if referenced directly. Use *ReadMem* and *WriteMem* to dereference pointers passed as arguments to system calls. Consult Section 2.4 for more details.

4.4 Execution Trace of User-Level Process

Consider the simplest Nachos program, *halt.c,* which invokes the "halt" system call and does nothing else. Its source code is shown in Figure 1

When compiled, *gcc -S* generates the assembly language code shown in figure 2 (line numbers and additional comments have been added for clarity). The only instruction that directly relates to invoking the "halt" system call is given in line 27. The *jal* instruction executes a jump that transfers control to the label "Halt". Note that there is no label called "Halt" in this program. The code for "Halt" can be found in the file *start.s*. Lines 1-21 are assembler directives that don't actually generate any instructions. Lines 22-25 perform the standard procedure call linkage operations that are performed as part of starting execution in a new subroutine (e.g., saving the return address and allocating space on the stack for local variables). Line 26 is a call to a gcc-supplied library routine that Unix needs to execute before your program normally begins execution (it is not needed in Nachos, so a dummy routine is provided that does nothing). Lines 29-33 provide the standard code for returning from a procedure call, effectively undoing the effects of lines 22-25 (e.g., fetch the saved return address and then jump back to it).

The actual instructions for making system calls are found in *start.s*. Figure 3 shows the subset of that file that is used by the halt program. Again, lines 1-7 are assembler directives rather than instructions. Line 8 is the first actual instruction, and simply calls the procedure *main*, e.g., the main program in *halt.c*. Lines 9-10 are executed whenever the call to *main* returns, in which case we want to tell Nachos we are done via the "exit" system call.

System calls are invoked at runtime by placing a code for the system call in register 2 and then executing the "syscall" machine instruction. The "syscall" instruction is a trap instruction, meaning that the next instruction to be executed will be the first instruction of the trap handler. In Nachos, this effectively means that execution continues now in the procedure *ExceptionHandler*. Consider lines 15-18, the steps involved in making an

```
	.file	1 "halt.c"
 # GNU C 2.4.5 [AL 1.1, MM 40] DECstation running ultrix compiled by GNU C
 # Cc1 defaults:
 # Cc1 arguments (-G value = 0, Cpu = default, ISA = 1):
 # -G -quiet -dumpbase -o
gcc2_compiled.:
__gnu_compiled_c:
	.text
	.align	2
	.globl	main
	.loc	1 16
	.ent	main
main:
	.frame	$fp,24,$31		# vars= 0, regs= 2/0, args = 16, extra= 0
	.mask	0xc0000000,-4
	.fmask	0x00000000,0
	subu	$sp,$sp,24		# SP <- SP-24 (allocate space on stack)
	sw	$31,20($sp)		# Save return address on stack
	sw	$fp,16($sp)		# Update FP
	move	$fp,$sp			# FP <- SP
	jal	__main
	jal	Halt
$L1:
	move	$sp,$fp			# sp not trusted here
	lw	$31,20($sp)
	lw	$fp,16($sp)
	addu	$sp,$sp,24
	j	$31
	.end	main
```

Figure 2: Source for the program *halt.c*

"exit" system call. Line 16 places a code for "exit" in register r2, and line 17 performs the actual system call. Line 18, the first instruction that will be executed after the system call simply returns to the caller. Note that the Exit system call normally won't return (if coded correctly!), but a return is provided anyway.

The actual steps in converting the *halt.c* source code into an executable program are shown in Figure 4. Line 1-3 generate object code from the *halt.c* and *start.s* source files. Line 5 creates an executable binary. Note that listing *start.o* before *halt.o* insures that the code in start.s resides before that of the main program. Finally, line 6 translates the executable into Noff format, making it ready to execute under Nachos.

The utility *disassemble* can be used to show the actual instructions in the halt binary. The result of running disassemble on *halt.coff* (NOT *halt*) is given in Figure 5. The instructions are the same as described above, but the code is somewhat harder to understand because the labels have been replaced with addresses.

5 Nachos Filesystem

There are two versions of the Nachos filesystem. A "stub" version is simply a front-end to the Unix filesystem, so that users can access files within Nachos without having to write their own file system. The second version allows students to implement a filesystem on top of a raw disk, which is in fact a regular Unix file that can only be accessed through a simulated disk. The simulated disk accesses the underlying Unix file exclusively through operations that read and write individual sectors. Both file systems provide the same service and interface.

Files are accessed through several layers of objects. At the lowest level, a *Disk* object provides a crude interface for initiating I/O operations for individual sectors. Above *Disk*, the *SynchDisk* object provides synchronized access to a *Disk*, blocking callers until a requested operation actually completes. In addition, *SynchDisk* allows multiple threads to safely invoke disk operations concurrently. The *FileSystem* object handles creating, deleting, opening and closing individual files. Sitting alongside the *FileSystem*, the *OpenFile* object handles the accessing of an individual file's contents, allowing seeks, reads, and writes. The *FileSystem::Open* routine returns a file descriptor (actually an *"OpenFile *"*), that is used to access the file's data. The raw *Disk* object was described in Section 2.6. The remaining objects are discussed in detail below.

5.1 SynchDisk

The *SynchDisk* object resides above the raw disk, providing a cleaner interface to its services. Specifically, it provides operations that block the calling thread until *after* the corresponding I/O complete interrupt takes place. In contrast, the *Disk* object provides the mechanism for initiating an I/O operation, but does not provide a convenient way of blocking the caller until the request completes. In addition, *SynchDisk* provides mutual exclusion, so that multiple threads can safely call the *SynchDisk* routines concurrently (recall that *Disk*

```
#include "syscall.h"
        .text
        .align  2

        .globl  __start
        .ent    __start
__start:
        jal     main            # Call the procedure ``main''
        move    $4,$0           # R4 <- 0
        jal     Exit            /* if we return from main, exit(0) */
        .end    __start

        .globl  Halt
        .ent    Halt
Halt:
        addiu   $2,$0,SC_Halt
        syscall
        j       $31
        .end    Halt

        .globl  Exit
        .ent    Exit
Exit:
        addiu   $2,$0,SC_Exit
        syscall
        j       $31
        .end  Exit
 /* dummy function to keep gcc happy */
        .globl  __main
        .ent    __main
__main:
        j       $31
        .end    __main
```

Figure 3: Instructions in *start.s* used by the program in *halt.c*

.

```
gcc -G 0 -c -I../userprog -I../threads -c halt.c
/lib/cpp -P -I../userprog -I../threads start.s > strt.s
as  -o start.o strt.s
rm strt.s
ld -N -T 0 start.o halt.o -o halt.coff
../bin/coff2noff halt.coff halt
numsections 2
Loading 2 sections:
        ".text", filepos 0xa0, mempos 0x0, size 0x100
         ".comment", filepos 0x200, mempos 0x0, size 0x24
```

Figure 4: The output from *make* when compiling *halt.c*.

cannot service more than one request at a time). Rather than using the *Disk* object directly, most applications will find it appropriate to use *SynchDisk* instead. *SynchDisk* provides the following operations:

SynchDisk(char *name): Constructor takes the name of Unix file that holds the disk's contents.

This routine is called once at system "boot time" (e.g., as part of Nachos initialization process in *system.cc*), and it uses the Unix file "DISK" as its backing store. The *SynchDisk* object can be accessed through the global variable *synchDisk*.

RequestDone(): Interrupt service routine the underlying *Disk* object calls when an I/O operation completes. Simply issues the semaphore *V* operation.

ReadSector(int sectorNumber, char *data): Acquire a mutual exclusion lock, invoke the underlying *Disk::ReadSector* operation and then wait on the semaphore that *RequestDone* signals when the I/O operation completes.

WriteSector(int sectorNumber, char *data): Acquire a mutual exclusion lock, invoke the underlying *Disk::WriteSector* operation and then wait on the semaphore that *RequestDone* signals when the I/O operation completes.

5.2 FileSystem Object

Nachos supports two different filesystems, a "stub" that simply maps Nachos file operations into ones the access Unix files in the current directory, and a Nachos file system that users can modify. Use of the stub filesystem makes it possible to implement paging and swapping before implementing a Nachos file system. The *-DFILESYS_STUB* compilation flag controls which version gets used.

The *FileSystem* object supports the following operations:

```
Address          Contents        Instruction Target
(hex)            (hex)
00000000: 0c000034          jal      000000d0
00000004: 00000000          nop
00000008: 0c000008          jal      00000020
0000000c: 00002021          addu     r4,0,0
00000010: 24020000          addiu    r2,0,0x0
00000014: 0000000c          syscall
00000018: 03e00008          jr       r31
0000001c: 00000000          nop
00000020: 24020001          addiu    r2,0,0x1
00000024: 0000000c          syscall
00000028: 03e00008          jr       r31
0000002c: 00000000          nop
 [ misc instructions deleted ]
000000c0: 03e00008          jr       r31
000000c4: 00000000          nop
000000c8: 00000000          nop
000000cc: 00000000          nop
000000d0: 27bdffe8          addiu    sp,sp,0xffffffe8
000000d4: afbf0014          sw       r31,0x14(sp)
000000d8: afbe0010          sw       r30,0x10(sp)
000000dc: 0c000030          jal      000000c0
000000e0: 03a0f021          addu     r30,sp,0
000000e4: 0c000004          jal      00000010
000000e8: 00000000          nop
000000ec: 03c0e821          addu     sp,r30,0
000000f0: 8fbf0014          lw       r31,0x14(sp)
000000f4: 8fbe0010          lw       r30,0x10(sp)
000000f8: 03e00008          jr       r31
000000fc: 27bd0018          addiu    sp,sp,0x18
```

Figure 5: The disassembled output of the executable for the *halt.c* program.

FileSystem(bool format) For the stub system, this constructor (and destructor) does nothing; the Unix file system is used instead. For the Nachos file system, *format* indicates whether *FileSystem* should be re-initialized or whether the previous contents of the file system (from an earlier Nachos session) should be retained.

The *FileSystem* constructor is called once at "boot time." It assumes that a *synchDisk* instance has already been created and uses it to store the file system.

bool Create(char *name, int initialSize) creates a zero-length file called *name*. If the file already exists, *Create* truncates its contents. Note that *Create* returns a boolean value, either success or failure, but does not actual open a file; it simply makes sure that it exists and has zero-length. To actually write to a file that has been created, a subsequent call to *Open* must be made.

Argument *initialSize* specifies the actual size of the file. (Note: *initialSize* is ignored in the stub filesystem; Unix doesn't require that a file's maximum size be specified at the time it is created.) For the regular filesystem, specifying the file's size at creation time simplifies implementation. Sufficient disk sectors can be allocated in advance to hold the entire (potential) file contents, and appending data to the file requires nothing more than accessing the appropriate blocks. Of course, one possible variation for the assignment is to implement extensible files whose actual size grows dynamically.

OpenFile *Open(char *name) opens file *name* for subsequent read or write access. In the stub file system, *Open* simply opens the Unix file for read/write access. Note that all files opened under Nachos are opened with both read and write access. Thus, the user opening the file must have write permission for a file even when the file will only be read. One consequence of this can cause confusion while implementing multiprogramming. Nachos is unable to *Exec* binary files that are not writable because it cannot open them.

Another thing to note is that *Open* does not truncate a file's contents (it can't since you might be reading the file). However, when writing a file, it is sometimes the case that the file's existing contents should be deleted before new data is written. In Nachos, invoking *Create* before *Open* insures that a file is zero-length.

bool Remove(char *name) deletes file *name* and frees the disk blocks associated with that file.

Note that in Unix, the disk sectors associated with a deleted file are not actually returned to the free list as long as there are any processes that still have the file open. Only after all open file descriptors for a file are closed does Unix deallocate the disk blocks associated with the file and return them to the free list. The Unix semantics prevent certain types of undesirable situations. For example, deleting a binary file cannot cause processes paging off of that file's text segment to terminate unexpectedly when the paging system attempts to load an instruction page from non-existent file.

One suggested Nachos assignment is to provide Unix-style file deletion.

5.3 OpenFile Object

The *OpenFile* object manages the details of accessing the contents of a specific file. Opening a file for access returns a pointer to an *OpenFile* object that can be used in subsequent read and write operations. Each *OpenFile* object has an associated read/write mark that keeps track of where the previous read or write operation ended. Supported operations include:

OpenFile(int sector) opens the file *sector*. Argument *sector* is the sector number containing the *FileHeader* for the file. A *FileHeader* (discussed in detail below) is similar to a Unix *inode* in that the low-level file system routines know nothing about file names; files are uniquely identified by there *FileHeader* number. *OpenFile* returns a pointer to an *OpenFile* object that is subsequently used to invoke any of remaining operations.

int ReadAt(char *into, int numBytes, int position) copies *numBytes* of data into the buffer *into*. Argument *position* specifies at what offset within the file reading is to start. ReadAt returns the number of bytes successfully read.

int Read(char *into, int numBytes) simply invokes *ReadAt*, passing it the current read/write mark. *Read* is used to read the file sequentially from start to finish, letting keeping track of which part of the file has already been read, and at what offset the next read operation is to continue. Read returns the number of bytes successfully read.

int WriteAt(char *from, int numBytes, int position) copies *numBytes* of data from the buffer *from* to the open file, starting *position* bytes from the start of the file. *WriteAt* returns the number of bytes actually written.

int Write(char *from, int numBytes) is used to write a file sequentially. The first *Write* begins at the start of the file, with subsequent writes beginning where the previous operation ended. Write returns the number of bytes actually written.

int Length() returns the actual size of the file.

5.4 File System Physical Representation

Nachos places a file system on top of an existing *SynchDisk* object. Before going into the details, it is helpful to consider how files themselves are stored.

5.4.1 File Header

Each Nachos file has an associated *FileHeader* structure. The *FileHeader* is similar to a Unix *inode* in that it contains all the essential information about a file, such as the file's current size and pointers to its physical disk blocks. Specifically, a Nachos' *FileHeader* contains the current size of the file in bytes, the number of sectors that have been allocated to the file and an array of sector numbers identifying the specific disk sector numbers where the file's data blocks are located. Recall that when a file is initially created, the caller specifies the size of the file. At file creation time, Nachos allocates enough sectors to match the requested size. Thus, as data is appended to the file, no sectors need be allocated. The current size field indicates how much of the file currently contains meaningful data.

Note that a *FileHeader* contains only "direct" pointers to the file's data blocks, limiting the maximum size of a Nachos file to just under 4K bytes.

The following *FileHeader* operations are supported:

bool Allocate(BitMap *bitMap, int fileSize): Find and allocate enough free sectors for *fileSize* bytes of data. Nachos uses a bit vector to keep track of which sectors are allocated and which are free. Argument *bitMap* is the bit map from which data blocks are to be allocated (e.g., the freelist).

void Deallocate(BitMap *bitMap): Return to the free list (e.g., *bitMap*) the blocks allocated to this file (header). Only the file's data blocks are deallocated; the sector containing the *FileHeader* itself must be freed separately. This operation is invoked when a file is deleted from the system.

void FetchFrom(int sectorNumber): Read the *FileHeader* stored in sector *sectorNumber* from the underlying disk.

void WriteBack(int sectorNumber) Write the *FileHeader* to sector number *sectorNumber*.

int FileLength(): Return the current size of the file.

int ByteToSector(int offset) Map the offset within a file into the actual sector number that contains that data byte.

Because a *FileHeader* fits into a single sector, the sector number containing a *FileHeader* uniquely identifies that file. We will use *fnode* (for FileheaderNODE) to refer to a sector that contains a *FileHeader*.

5.4.2 Directories

Nachos supports a single top-level directory, managed by the *Directory* object. When a file is created, it is added to the directory; likewise, deleting a file results in its removal from the directory. Directory entries themselves consist of (filename, fnode, free_flag) triplets, with the *free_flag* indicating whether that directory slot is currently allocated. The following directory operations are supported:

Directory(int size): This constructor creates an (in-memory) directory object capable of holding *size* entries.

void FetchFrom(OpenFile *file): Fetch the directory contents stored in file *file*.

void WriteBack(OpenFile *file): Flush the contents of the directory to the file *file*.

int Find(char *name): Search the directory for a file called *name*, returning its fnode number if the file exists.

bool Add(char *name, int newSector): Add the file *name* with fnode *newSector* to the directory. Note that this routine only updates the in-memory copy of the directory. To make the directory changes permanent, *WriteBack* must subsequently be invoked.

bool Remove(char *name): Remove file *name* from the directory. Note that the *Remove* operator simply updates the directory; the *FileHeader* and data sectors associated with *name* are deallocated separately. In addition, a subsequent call to *WriteBack* is needed to make the changes permanent.

List() Print out the directory contents (debugging).

Print() Print out contents of all files in the directory (debugging).

5.4.3 Putting It All Together

On disk, both the list of free sectors and the top-level directory are stored within regular Nachos files. The free list is stored as a bit map, one bit per sector (e.g., allocated or free), with the file itself stored in fnode 0. Thus, finding a free sector in the filesystem requires reading the file associated with fnode 0, using the bitmap functions to locate free sector(s), and then flushing the file changes back to disk (via the appropriate *WriteBack* routines) to make the changes permanent.

Likewise, the top-level directory is stored in a file associated with fnode 1. Updating the directory when creating a new file requires reading the file associated with fnode 1, finding an unused directory entry, initializing it, and then writing the directory back out to disk.

When a file system is created, the system initializes the freelist and top-level directory and then opens the files containing the two main data structures. Thus, the current contents of the free list and directory (as stored on disk) can be accessed via the *"OpenFile *"* variables *freeMapFile* and *directoryFile*.

When creating and modifying Nachos files, one must be careful to keep track of what data structures are on disk, and which ones are in memory. For example, when creating a file, a new *FileHeader* must be allocated, together with sectors to hold the data (free list file). The top-level directory must also be updated. However, these changes don't become permanent until the updated free list and directory files are flushed to disk.

If one traces through the code for *FileSystem::Create*, one sees that all the allocations and updates are first made in memory local to the calling thread, and only after all allocations have succeeded without error are the changes made permanent by committing them to disk. This approach greatly simplifies error recovery when (say) there are enough free blocks to create the file, but the directory has no room to hold the new file. By not flushing to disk the allocations that succeeded, they do not actually take effect, and cancelling the entire transaction is straightforward.

Note also that the supplied *FileSystem* code assumes that only a single thread accesses the filesystem at any one time. Specifically, consider what would happen if two threads attempted to create a file simultaneously. One scenario would have both threads fetching the current freemap, both updating their local copies, and then both writing them later. At best, only one set of changes will appear on disk. At worst, the disk becomes corrupted.

No attempt has been made to reduce disk latencies by clustering related blocks in adjacent sectors.

6 Experience With Nachos Assignments

The following subsections discuss in more detail issues related to the specific programming assignments. They are based on my experiences after having used Nachos in two courses. In my experience, students are easily overwhelmed by the projects. They frequently don't know where to start and spend a large amount of time unproductively. The result is frustrated students who don't complete projects. With a bit more guidance, these same students can complete the projects. Although it is valuable for students to tackle a large project given only a general problem description, this only benefits those who actually succeed. At the undergraduate level especially, students may lack adequate preparation.

6.1 General Tips

1. Use a debugger. Nachos programs simply cannot be debugged by looking at source code or inserting print statements. Unfortunately, that is what many students do (with obvious consequences). I found it necessary to force students to learn to use a debugger (e.g., no help in office hours otherwise).

2. Use of code browsing tools (e.g., the *emacs* "tags" facility) greatly simplifies the reading and understanding of source code.

3. Students don't always test programs well. If it works on one test case, they assume it works for all of them. For those studying operating systems for the first time, designing test cases is particularly difficult. When grading programs, it was not uncommon for student-supplied test programs to work, with all of my test cases failing. I found that I got much improved submissions by supplying concrete test programs that students had a chance to experiment with themselves.

4. The Nachos assignments included in the distribution are very general and leave out many details. I got much better results by providing more detail about what needs to be done and how to do it. Although students get a benefit greatly from figuring out things on there own, I found too many students were getting lost and frustrated. Students with weak backgrounds are especially vulnerable. Later subsections (and indeed, this entire document) provide examples of such additional details.

5. Students have a tendency to try to solve the entire problem at once, rather than one step at a time. This is particularly problematical with Nachos, because students often don't understand the big picture. I found it useful to break up an assignment into smaller standalone pieces, each building on the previous pieces, with a rough indication of the weight for each part. There are two benefits. First, solving one component of an assignment frequently leads to a better understanding of the issues necessary to solve subsequent components. Second, students know how much credit they will receive for the work done so far, and can better decide whether they should keep working on a project or use their time in other ways.

6.2 Synchronization

Testing the correctness of the synchronization facilities is difficult if only Nachos threads are used because Nachos preempts threads only at well defined points (e.g., when disabling or re-enabling interrupts). Thus, many incorrect solutions tend to work on the simple test cases students use. Fear not, later assignments test the synchronization facilities extensively and will uncover problems. Here are some suggestions for simplifying later debugging:

1. For the locking routines, have *Acquire* verify that the caller does not already hold the lock via an ASSERT statement. One common problem occurring in later assignments involves nested locking, where a thread already holding a lock attempts to acquire it again. Nachos can't always detect this deadlock condition itself (e.g, if the Console device is in use) and students with deadlocked programs spend a long time not knowing where to look before they realize they've deadlocked.

Another most common mistake students made while implementing locks is failing to understand their semantics (even having covered them in class). Specifically, students tended to have a flag or counter associated with condition variables. The following explanation will hopefully clarify this point and explain why a condition variable cannot have a value.

Semaphores have a count associated with them. This count is a "memory" that allows a semaphore to "remember" how many times the semaphore has been signalled (via P) relative to the number of times it has been signalled (via V). Semaphores are useful as gatekeepers that must limit the number of persons (threads) allowed to be in a room simultaneously. Threads enter the room enter through one door (the P operation) and leave through a second door (the V operation). The semaphore's value keeps track of the number of threads in the room at any given time. If the semaphore's initial value is 1, only one thread may be in the room at a time, whereas an initial value of 100 allows the room to hold 100 threads.

In contrast, a condition variable has *no* value associated with it. If a thread invokes *Wait*, it blocks *no matter what!* Specifically, it blocks regardless of how many previous *Signal* operations were performed. In contrast, the semaphore P operation blocks only when the count is non-positive, with the current value dependent on the initial counter and the number of V operations that have been performed. One uses a semaphore (and its counter) only when the number of P's and V's must balance each other. Note also that the condition

Signal operations wakes up one blocked thread (if one is waiting on the condition), but has no effect otherwise. If there are no threads waiting on a condition, executing *Wait* one time has the same effect as executing it 1,000 times (e.g., no effect). Compare this behavior with the semaphore V operation. Even when no threads are waiting on the semaphore, the V operation increments the semaphore's value, which influences P's behavior later.

There are times when a thread may wish to wait for some future event, but the conditions necessary that define the event can't be described with a simple count. For example, a thread may wish to wait for two resources to become available simultaneously, but does not wish to hold one while it is blocked waiting for the other. In this case, a thread might wait on a condition variable, with another thread issuing a *Signal* when the necessary condition (both resources available *simultaneously*) becomes true. Note that only the code that calls *Signal* and *Wait* know what the exact conditions are. Thus, the condition variable itself cannot hold a meaningful value.

Finally, note that Nachos intends condition variables to have "Mesa-style" semantics. In brief, when a *Signal* operation releases a thread, that thread can *not* assume that the condition that it waited for is now true. It must first reassert that the necessary condition still holds (e.g., it must test the condition in a while loop, calling *Wait* again whenever the condition is false). This is necessary because another thread may be scheduled and execute between the time of *Signal* and the running of the waiting thread. That intermediate thread may change the state of the system so that the waited-on condition no longer holds.

6.3 Multiprogramming

1. The Nachos *Fork* and *Exec* routines have completely different semantic from the Unix system calls. In particular, the Nachos system calls do the following:

   Exec Creates a new address space, reads a binary program into it, and then creates a new thread (via *Thread::Fork*) to run it. Note that in Unix separate *fork* and *exec* system calls are needed to achieve the same result.[4]

   Fork Creates a new thread of control executing in an existing address space. This is a non-trivial system call to implement (for reasons discussed later).

2. When a Nachos program issues an *Exec* system call, the parent process is executing the code associated with the *Exec* call. At some point during the *Exec* call, the parent will need to invoke *Thread::Fork* to create the new thread that executes the child process. Careful thought is required in deciding at what point the *Fork* operation should be done. Specifically, after *Fork*, the (new) child thread can no longer access variables associated with the parent process. For example, if the parent copies the name of the new file to be executed into a variable, then issues a *Fork*, expecting the child to open that file, a race condition exists. If the child runs *after* the parent, the parent may already have deleted (or changed) the variable holding that file name.

[4]Warning: when opening files, Nachos opens all files for both reading and writing. Thus, the binary being loaded off of disk must be writable.

The above race condition appears in a number of similar contexts where two processes exchange data. The problem is that the parent and child threads need to synchronize with each other, so that the parent doesn't reclaim or reuse memory before the child is finished accessing it.

3. When the MIPS simulator "traps" to Nachos via the *RaiseException* routine, the program counter (PCReg) points point to the instruction that caused the trap. That is, the PCReg will not have been updated to point to the next instruction. This is the correct behavior. When the fault is due to (say) a page missing, for example, the faulting instruction will need be re-executed after the missing page is brought into memory. If PCReg had already been updated, there would be no way of knowing which instruction to re-execute.

   On the other hand, when a user program invokes a system call via the "syscall" instruction, re-executing that instruction after returning from the system call leads to an infinite system call loop. Thus, as part of executing a system call, the exception handler code in Nachos needs to update PCReg to point to the instruction following the "syscall" instruction. Updating PCReg is not as trivial as first seems, however, due to the possibility of delayed branches. The following code properly updates the program counter. In particular, not properly updating NextPCReg can lead to improper execution.

```
pc = machine->ReadRegister(PCReg);
machine->WriteRegister(PrevPCReg, pc);
pc = machine->ReadRegister(NextPCReg);
machine->WriteRegister(PCReg, pc);
pc += 4;
machine->WriteRegister(NextPCReg, pc);
```

   Use of the following test programs greatly helped convince me that my implementation was (finally) correct:

   ConsoleA Simple program that simply loops, writing the character 'A' to the console. Likewise, *ConsoleB* loops, printing the character 'B'.

   shell The provided shell starts a process and then issues a "Join" to wait for it to terminate. This test multiprogramming in a very rudimentary fashion only, because the shell and the invoked program rarely execute concurrently in practice.

   Modify the shell so that it can run jobs in background. For example, if the first character of the file is an "&", start the process as normal, but don't invoke a "Join." Instead, prompt the user for the next command.

   With the modified shell, run the *ConsoleA* and *ConsoleB* programs concurrently.

6.4 Virtual Memory

1. It is easiest to test the VM algorithms by running only a single process and have it page against itself. It's a lot easier to see what is happening this way. Add debug print

statements using the *DEBUG()* routine; I used "p" for paging to turn it on. I found it immensely useful to add debugging statements at the following points:

- when a page is reclaimed (e.g., taken away from a process and made available to another one), indicate which process (address space) was forced to give up a page, which virtual page number it gave up, and the corresponding page frame number in physical memory.

  In order to print the name of the address space that was forced to give up a page, add a "char *name" field to the *AddrSpace* object and initialize it to the name of the Nachos binary when the space is initially created.

- Whenever a process has a page fault, print the address space that caused the fault, which virtual page number the fault was for, and the physical page frame that was used to satisfy the fault.

2. Reduce the amount of available physical memory to insure lots of page faults. I got good results with 25 pages of real memory. Also, the *matmult* program in the test subdirectory uses quite a lot of memory. It is a good program to run with the "-x ../test/matmult" option to nachos.

   Another good test is to use a modified shell (the one that allows you to run a process in background by putting a '&' character before the file name). Run the *matmult* program in background, and then also run the console1 and console2 programs at the same time. You should be seeing alternating A's and B's as before (if using the "-rs" Nachos option).

3. You will need to maintain a data structure called a *core map.* The core map is a table containing an entry for every physical page frame in the system. For each page frame, a core map entry keeps track of:

   - Is the page frame allocated or free?
   - Which address space is using this page?
   - which virtual page number within space?
   - Plus possibly other flags, such as whether the page is currently locked in memory for I/O purposes

   When the system needs to find a free page, but none are available, the memory manager inspects the core map to find good candidate replacement pages. Of course, the memory manager looks at the page table entry for that frame (via the "space" pointer) to determine if a page has been used recently, is modified, etc.

   Start with a simple page replacement policy. I simply reclaimed frames in sequential order starting with frame 0. That is, first reclaim 0, than 1, etc., wrapping around when you get to the end of the list. This is obviously NOT an ideal policy, but it works for the purposes of testing your implementation (in fact, by being a "bad" policy, it tests your code surprisingly well). Once the rest of your code is debugged, then worry about implementing a better policy.

4. You need to think *very* carefully about mutual exclusion. For example, the page replacement process may select frame 13, but find that it needs to write it to backing store before it can be reclaimed. This will take a long time (I/O is slow), and in the meantime the process that was using that frame may be selected by the scheduler and start running again. What happens when it tries to reference that frame? It will surely notice that the page is gone, and try to reload it from backing store. Will it get the proper copy? Will it read the page only *after* the page replacement object has written it? There are all sorts of race conditions like that can potentially mess things up. So be sure to use appropriate mutual exclusion in the paging routines.

5. You will need to maintain a "shadow" page table for each address space. The existing page table defined by Nachos has a specific format dictated by the address translation hardware. You will need to keep additional information about each page, thus the shadow table. The shadow table keeps track of such things as the current state of the page (e.g., loaded in memory, needs to be loaded from the text or swap file, etc.)

   Moreover, when you do away with the hardware page table and use the TLB, the shadow table contains everything the standard table contains (e.g., reference bit, dirty bit, etc.). That is, when a program traps due to a TLB miss, the page fault handler will look in the shadow table to find the appropriate mapping (loading it from backing store first, if necessary), update the TLB and then restart the faulting instruction.

   In order to simplify the migration to the TLB part of this assignment, you may want to have your paging algorithms access only the shadow page tables (rather than the hardware page tables) when deciding which pages to reclaim, etc. That way, when the hardware table goes away, none of the routines need to be changed. One consequence of this approach, however, is that some information (e.g., dirty and reference bits) will be duplicated in both tables at the same time. Since the hardware automatically updates its page table, but not the shadow table, you will need to synchronize the two tables at key points *before* inspecting the shadow table. For example, before checking the dirty bit for an entry in the shadow page table, make sure that the shadow table has the must recent value for that bit. You may find it useful to have a routine *SyncPageTableEntry()* that synchronizes a shadow entry with its corresponding hardware value. Note: you will probably need such a routine anyway later when doing the TLB part of the assignment, since the Nachos hardware only updates its TLB entries.

6. You will need to allocate backing store (swap space) for each address space. One approach is to create a file (e.g., swap.PID) when the address space is created, open it, and then use it while the process is running. The file should be closed and removed when the process terminates.

   When a page fault takes place, there are three scenarios:

   (a) The page belongs to the stack or heap section, and should be initialized to 0. This is the easiest case to handle, because you only have to find a free page and zero it out. There is no need for disk I/O.

(b) The page should be loaded from the swap file. This simply involves reading the page in the swap file at the offset corresponding to the page being accessed. For example, page number 7 would correspond to an offset of 7*PageSize = 7*128 = 896.

(c) The page contains instructions or initialized data and should be loaded from the original binary file.

This is the most difficult case. You will need to figure out what offset within the text file the desired page begins at. To simplify matters, assume that the text and initialized data segments are laid out contiguously in the Nachos binary file (they are). That way, you only need to keep track of where the text segment begins (it doesn't start at offset 0, the NOFF header resides there) and the combined length of the text and data segments are.

To handle the three above cases, you will need to keep two OpenFile objects with the *AddrSpace* object. One for the text file, the other for the swap file. In addition, for each page in an address space's shadow page table, maintain a state variable that indicates what state the page is in. For example, LOADFROMTEXT, LOADFROMSWAP and ZEROFILL. The state of a page is initialized when a process is created, and changes only if the memory manager reclaims the page, finds it modified, and writes it to backing store.

7. You will need routines that lock a range of addresses into memory (and unlock them later). For example, when a program does a system call to read data from a file, it provides a pointer to the buffer where it wants the file contents placed. The *Read* system call first reads the desired data into its own buffer, and then copies the data into the user buffer using the *machine::WriteMem()* operations. In order for this operation to succeed, the pages holding the buffer must be in memory. Thus, you will want to lock the pages associated with the buffer into memory before attempting to place data in the buffer. Once the data has been copied to the buffer, the pages can be unlocked again. For example, the routine *LockMemory()* might take a virtual address and length, insure that all pages corresponding to the specified addresses indicated are loaded and present in physical memory and lock them into place. While locked in memory, the memory manager would not reclaim them.

   What you must do for full credit. Please note that I will grade these in order. If part 1 doesn't work, I won't even look at 2 or 3. Don't waste your time working on any parts before completing the previous parts.

1. (40% of total grade) Get demand loading to work. That is, rather than load the program into memory during "exec", load individual pages as they are first referenced. If a page is never referenced, it is never loaded into memory. Assume that there is enough physical memory so that no pages ever need to be reclaimed.

2. (70% of total grade) In addition to 1), get full blown paging with backing store implemented. That is, when the system runs out of physical memory, grab any page, write it to backing store if necessary, and then reuse the page frame. Use a simple

policy (like the one described above). The focus is on getting the mechanics of page replacement/backing store to work.

3. (85% of total grade) In addition to the previous steps, implement a better page replacement policy, such as LRU.

4. (100% of total grade) In addition to the previous steps, use the TLB rather than hardware page tables. (E.g., compile with the -DUSE_TLB option).

6.5 File System

Locking is a major pain because *WriteAt* calls *ReadAt* (on occasion). One probably wants to lock for both reads and writes, but if one already has the lock (while writing) and then calls read, potential deadlock scenarios appear immediately. Here are some particularly tricky cases:

1. It is tempting to have a single lock per file, which one acquires as the first step in beginning a read or write. However, there are cases where *WriteAt* also calls *ReadAt*. Specifically, if *WriteAt* needs to write a partial sector, it must first read the sector and then update the subset of that sector the *WriteAt* refers to. However, if *ReadAt* attempts to acquire the same lock that *WriteAt* already holds, deadlock results.

2. Consider updating an directory (as part of create). To properly update the directory, it is read from disk, then modified in memory, then written back out to disk. For correctness, access to the directory must be locked for the duration of all operations (not just while reading or writing). Otherwise, two threads could attempt to update the directory simultaneously. If both threads read the directory, modified separate in-memory copies and then wrote out the changes, only one set of changes would make it out to disk. Note that it is insufficient to lock the directory only during the reads and writes; a lock must be held across both operations. Thus, acquiring locks within *ReadAt* or *WriteAt* is insufficient.

3. A similar scenario occurs when updating the freelist as part of removing a file. One must first read the bitmap, update it, and then flush the changes back out to disk. The bitmap routines *FetchFrom* and *WriteBack* call *ReadAt* and *WriteAt* respectively.

6.6 Common Errors

This section discusses common problems that students ran into over and over again (even after having been warned about them).

1. The VM assignment works mostly correctly, but running *matmult* in background prevents the shell from executing any more commands, even though it prompts the user and seems to read the command OK.

What is happening: The pages (in user space) holding the arguments to the system calls are being paged out before the system call routines copy the contents of those pages into local variables. For example, *Exec* needs the name of the binary file to run. Routine *machine::ReadMem* does indeed invoke *Translate* to perform the translation, and if the target page is not present in physical memory, generates a page fault (e.g., by invoking *RaiseException*). *However*, *ReadMem* does *not* retry the fetch operation; instead, it returns FALSE indicating that the operation failed. Thus, the first byte the user attempts to fetch will not be retrieved. If Nachos immediately restarts the operation, the page will now be present in memory (the probability is very high anyway). If *ReadMem* is called from a loop that fetches one character at a time, skipping the first character will likely cause the wrong thing to happen (it is frequently NULL in practice, terminating the string). The same problem occurs with *WriteMem.* and can occur with any system calls that take pointers as arguments.

One shortcut that students want to make is to simply retry the *ReadMem* operation when it fails. That is, they assume that the only reason *ReadMem* fails is because the target page was paged out. If the pointer is invalid (e.g., outside of the process's address space), however, retrying the operation results in the same failure. Indeed, Nachos goes into an infinite loop at this point.

2. Problem: value of variables (strings in particular) changes randomly.

   What is happening: Dangling references. Although this is a standard C programming problem, students repeatedly forgot to allocate space for strings. For example, as part of the *Exec* system call, it is useful for debugging purposes to store the name of the file with the address space. This requires allocating space to hold the string and copying that file name into that newly allocated space. In contrast, students often simply set a pointer to the file name argument passed to the routine that implements *Exec*. However, when that routine returns, the *Exec* system call completes, and the space that held the file name gets deallocated. Chance are good that the next system call to be executed will reuse the same space, placing different data in its contents.

7 MIPS Architecture

One thing you need to be aware of is that the MIPS architecture (like many RISC architectures) has a number of instructions that do not take effect right away. Instructions that fetch values from memory, for example, take so long to execute that another instruction (that doesn't access memory) can be executed before the memory access completes. Such instructions are called delayed loads; the loaded value doesn't become available right away. The MIPS architecture executes one instruction during such *load delay slots*.

In many cases, the compiler can rearrange the order of instructions in order to fill the delay slot with useful instruction. For example, a value could be loaded a few instructions earlier than it is actually needed, as long as the target register was not being used. When tracing programs in Nachos, you should be aware that instructions used in calling subroutines have a one-instruction delay. That is, the PC can be changed quickly, but the processor must go to memory to fetch the instruction at the new address before it can actually execute it. Thus, the code for calling the procedure *Exit* with an argument of 0 would be done as follows:

```
jal        Exit         # Jump to procedure ''Exit''
addu        r4,0,0      # r4 <= r0 + 0 (r0 always has a value of 0)
```

That is, the instruction that zeroes out register r4 appears *after* the "jal" instruction, rather than before as expected.

Table **??** describes some common instructions you may encounter.