The University of Southern Queensland
Faculty of Sciences

Study Book

CSC2404/66204: Operating Systems

Version 1.2
(April 1998, Revised April 2002)

CSC2404/66204: Operating Systems

Assoc. Prof. Peiyi Tang
(Revised by Dr. Ron Addie)
Department of Mathematics and Computing
Faculty of Sciences
University of Southern Queensland
Toowoomba QLD 4350
Australia

Contents

I Overview **1**

1 Introduction **3**

2 Computer-System Structures **5**

3 Operating-System Structures **7**

II Process Management **9**

4 Process and Thread **11**

5 Process Synchronization **32**

III Storage Management **49**

6 Memory Management **51**

7 Implementation of System Calls **66**

8 Virtual Memory **73**

9 File-System Interface **80**

10 I/O Systems and File-System Implementation **88**

Preface

This study book covers all modules of unit 66204–Operating Systems. The whole unit is divided into three parts with 11 modules altogether. They are:

Part One: Overview

- Introduction
- Computer-System Structures
- Operating-System Structures

Part Two: Process Management

- Process and Thread
- CPU Scheduling
- Process Synchronization

Part Three: Storage Management

- Memory Management
- Implementation of System Calls
- Virtual Memory
- File-System Interface
- I/O Systems and File-System Implementation

The textbook of this unit is as follows:

- Silberschatz, A., Galvin, P., and Gagne, G., ”Operating System Concepts”, 6th Edition., Addison-Wesley, Reading, Massachusetts, 2002, ISBN 0-471-41743-2.

We also use the source code of the Nachos operating system as the essential teaching material of this unit. The Nachos operating system is a small working operating system on MIPS architecture written by Prof. Tom Anderson from the University of California at Berkeley and used widely for teaching operating systems throughout the world. We strongly believe that the only way to teach operating systems concepts and their design and implementation effectively is to have students read and experiment with an operating system at the source code level. The Nachos operating system is well written and roughly half of its source code lines are comments. The source code listing of the Nachos operating system used by this unit is provided in the Selected Reading of this unit.

~~The source code of Nachos version 3.4 as well as gcc cross-compiler for MIPS are provided in the CD made by this department. This CD also contains Red Hat LINUX version 5. You need to first install the LINUX operating system from this CD on your home PC. You then need to install and compile Nachos in your home directory. You also need to install the gcc MIPS cross-compiler on your LINUX system. The detailed instruction of installing these software is provided in the Introductory Book of this unit.~~

In this study book, we cover both the operating systems concepts and the source code of the Nachos operating system. Since most of the concepts in design and implementation of operating systems are well covered in the text, we focus on the Nachos operating system. Our goal is to give you guidance not only to study the concepts in the text, but also to read and understand the source code of the Nachos operating system.

Understanding and experimenting with Nachos are extremely important. A great proportion of the assignments and examination questions are directly related to Nachos. The assignments normally contain a number of programming assignments on Nachos. This is because unless you complete these questions and programming tasks successfully, we cannot be sure that you really understand the concepts of operating systems and their design and implementation.

Part I

Overview

This is the introductory part of the unit. Three modules in this part give you an overview of what operating systems and computers systems are as well as the overall structure of operating systems.

Chapter 1

Introduction

This module is an introduction to the entire course. The material comes from the chapter 1 of the text. You need to read the whole chapter.

1.1 Concepts

This module first gives the fundamental concepts of operating systems. The primary question this module tries to answer is: “What is an operating system?”

This module reviews different types of operating systems such as

- Simple Batch Systems
- Multiprogrammed Batched Systems
- Time-Sharing Systems
- Personal Computer Systems
- Parallel Systems
- Distributed Systems
- Real-Time Systems

Almost all these operating systems now exist except simple batch systems. You need to understand the unique characteristics of each type of operating systems as well as differences between them.

1.2 Questions

We use some of the questions at the end of the chapter 1 of the text as your study and review questions:

1. Question 1.1
2. Question 1.3
3. Question 1.4
4. Question 1.6

You should be able to answer these questions after you study this module.

Chapter 2

Computer-System Structures

This module is a quick review of computer system structure that you need to know in order to study operating systems design and implementation. It focuses on computer I/O organization and introduces all the important concepts of I/O operations in operating systems.

The material of this module is the chapter 2 of the text. You need to read the whole chapter and understand all the contents.

2.1 Concepts

The important concepts of computer systems covered in this module include:

- hardware interrupt, interrupt handler and interrupt vector
- trap or exception
- system call
- synchronous and asynchronous I/O
- device-status table
- DMA
- memory-mapped I/O and I/O instructions
- magnetic disks and tapes
- cache
- dual-mode operation

- CPU, memory and I/O protections

More details about I/0 systems can be found in the chapter 13 of the text.

2.2 Questions

We use some of the questions at the end of the chapter 2 of the text as your study and review questions:

1. Question 2.2.
2. Question 2.3.
3. Question 2.5
4. Question 2.8.

You should be able to answer these questions after you study this module.

Chapter 3

Operating-System Structures

This module describes the structure of operating systems. You should get an overview of

- the services provided by operating systems
- the user interfaces of operating systems
- the components of operating systems and their interconnections

after the study of this module. This module is based on Chapter 3 of the text. You need to read the whole chapter and make sure that you understand the concepts discussed.

3.1 Concepts

The first three sections of the chapter are devoted to the following aspects of operating systems:

- System Components
- Operating System Services
- Systems calls (User Interfaces)

Section 3.4 of the text describes typical *system programs* provided by operating systems including editors, compilers, etc.

The last part of the chapter discusses the architecture and design methodology of operating systems:

- System Structure

- Virtual Machines
- System Design and Implementation
- System Generation

3.2 Questions

1. Study the user interface of your LINUX operating system by examining its system calls manual pages. LINUX is a UNIX-compatible operating system with a user interface similar to most other UNIX operating systems. System calls of a typical UNIX system include:

   - fork, exec, exit, dup, getpid, signal, kill, wait, nice
   - read, write, open, lseek, close
   - chdir, chmod, chown, link, unlink

   and many more. For each of the above system calls, type: "`man 2 sys-call`" on your LINUX system and you will see the manual page for *sys-call*.

2. Question 3.1.
3. Question 3.2.
4. Question 3.3.
5. Question 3.4.
6. Question 3.6.
7. Question 3.7.
8. Question 3.10.
9. Question 3.11.
10. Question 3.13.

You should be able to answer these questions after you study this module.

Part II

Process Management

In this part, we cover the process and thread management of operating systems. We start with the basic concepts of thread and process in Module 4. We address the process and thread synchronization in Module 5.

When possible, we use the source code of Nachos operating system to illustrate the implementation of thread and process, their scheduling and synchronization.

We do not discuss CPU scheduling and deadlocks in this unit.

Chapter 4

Process and Thread

4.1 Introduction

What are processes and threads? This is the question which this module answers. This module is based on Chapter 4 and Chapter 5 of the text excluding Sections 4.5, 4.6, and Sections 5.4 to 5.8) and the source code of the Nachos kernel compiled in directory `threads/`. (See Selected Reading for the complete listing of Nachos code.)

The following concepts related to processes and threads are covered in this module:

- Process
- Thread
- Thread and Process Control Block
- Thread and Process Scheduling
- Thread and Process State Transition
- Thread Creation
- Context Switch
- Thread Termination
- Operating Systems Kernel

In short, a process is a program in execution. To execute a program, many resources are needed: CPU cycles, memory, files and I/O devices. But the key

question is how multiple processes share the single CPU and other resources of the computer system. After a program is compiled and linked, its binary executable is usually stored in a file in the secondary storage. How does the process which executes this program start? How does the system switch from one process to another when it cannot proceed? How can a suspended process resume its execution? This module answers all these questions in detail.

The first thing you need to do is to read the whole Chapters 4 and 5 of the text except Sections 4.5, 4.6, and Sections 5.4 to 5.8. We do not expect you to fully understand all the concepts in the first round of reading. As a matter of fact, you will need to go back and forth between the text and the relevant Nachos source code many times before you can really grasp the concepts in this module.

In the following sections, we go through the major points in this module. In particular, we use the Nachos source code to illustrate the concepts. This is probably the only way to enable you to have a sound grasp of the concepts described in the text.

4.2 Process

The concept of process is described in the section 4.1 of the text.

Process is a program in execution. What does a program look like when it is executed. Figure 4.1[1] shows the component of a simple program in execution. The address space of a program in execution represents its storage in the memory and consists of

- Text: the code of the program loaded from the binary executable in the secondary storage,
- Data: the data area of the program for global and static variables. It is divided into the initialized data and uninitialized data sections. The initialized data section is loaded from the binary executable.
- Stack: the area to store local variables of functions. It grows and shrinks as functions are called and returned.
- Heap: the area for dynamically allocated variables by, for example, `malloc()` in C or `new` in C++. It grows and shrinks as dynamic variables or structures are allocated and deallocated.

Apart from the address space, a process also has

[1]The design of this figure and Figure 4.2 is borrowed from "Pthreads Programming" by B. Nichols, D Buttlar and J. P. Farrell, O'Reilly & Associates, Inc.

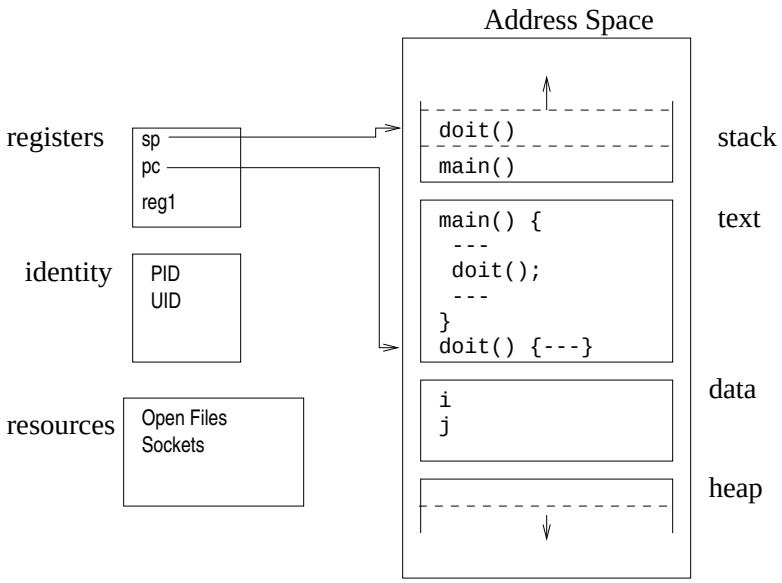

Figure 4.1: Components of a program in execution

- a set of registers. The most notable registers are
  - Program Counter (PC) which contains the address of the next instruction of the code to execute and
  - Stack Pointer (SP) which points to the current stack frame.

  Each CPU has only one set of registers and they are occupied by the *running* process at any particular time. Therefore, when a process is switched from the running state to any other states (blocked or ready), the contents of these registers need to be saved in the control block of the process; otherwise this process has no way to resume its execution. Saving and restoring the registers is part of the operation called *context switch*.

- identifying information associated with this process such as process identity (PID) – a unique integer number of the process within the operating system – and others.

- open and active resources such as open files which the process can read and write or connected active sockets to send and receive data through the network.

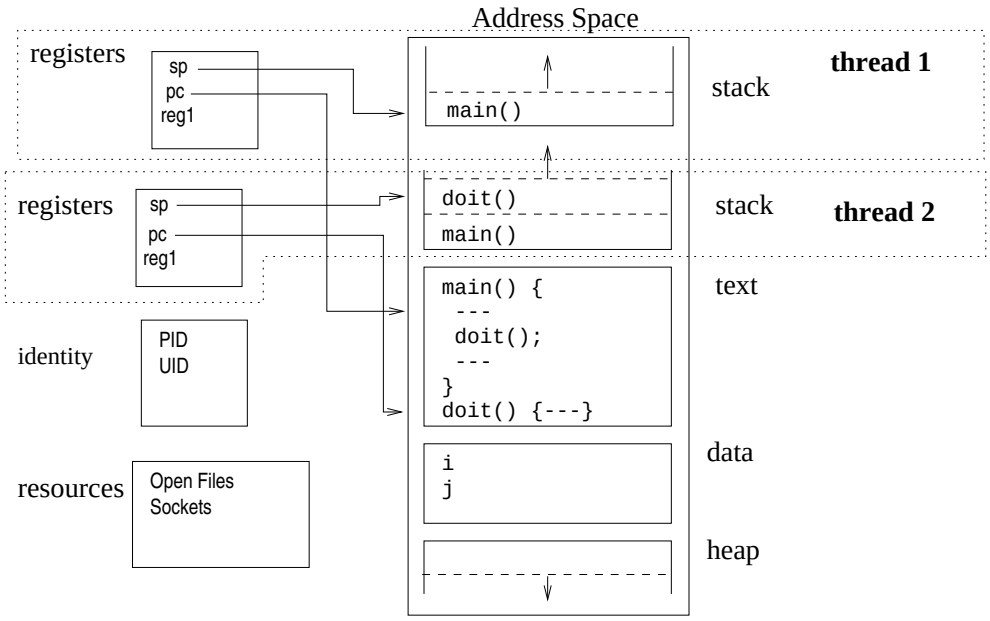

Figure 4.2: Threads of a process

4.3 Thread

What about thread? What are the differences between process and thread? The concept of thread is covered in Sections 5.1 to 5.3. Figure 4.2 shows two threads of a single process. They share the text and data sections, identity and resources of the process to which they belong. But, each thread has a separate register set and stack.

A thread is actually the abstraction of sequential execution of the program. Why does a thread needs a separate stack and register set? The stack and register set are the components that define the dynamic context of the program execution: the stack grows and shrinks as functions are called and returned and the contents of registers change after every instruction executed.

Now we have a hierarchical structure: an operating system can have many processes and each process can have multiple threads that share the common code, data, heap and process identity and resources.

Processes and threads share many concepts in common. They are:

- States and State Transition. Both threads and processes have states. Both processes and threads can make state transitions as shown in the Figure 4.1 of the text.

- Control Block. Both need to store critical information such as contents of registers. Of course, the actual fields in the control block differ. For example, memory management information is relevant only to process. A thread does not have its own address space.
- Context Switch. Both processes and threads need to support context switching.
- Scheduling. Both processes and threads need schedulers.

In the following discussion, we mainly concentrate on threads and their implementation in Nachos.

4.4 Thread and Process Control Block

A thread in Nachos is implemented as an object of class `Thread` in the kernel. The control block of a thread is implemented as part of data members of this class. The following lines of `threads/thread.h` shows the data members for the control block.

```
...

 77 class Thread {
 78   private:
 79     // NOTE: DO NOT CHANGE the order of these first two members.
 80     // THEY MUST be in this position for SWITCH to work.
 81     int* stackTop;                       // the current stack pointer
 82     _int machineState[MachineStateSize];  // all registers except for stackT
op
...

106   private:
107     // some of the private data for this class is listed above
108
109     int* stack;                         // Bottom of the stack
110                                         // NULL if this is the main thread
111                                         // (If NULL, don't deallocate stack)
112     ThreadStatus status;                // ready, running or blocked
113     char* name;

...
```

First of all, the state of a thread is stored in variable `status`. The type of `status`, `ThreadStatus`, is defined in the same file:

```
61 enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED };
```

A thread must be in one of these states. Here, state `BLOCKED` is the same as the waiting state in the text. As the state of the thread changes, the value of `status` changes accordingly.

A thread has its stack and registers. In Nachos, the stack of a thread is allocated when its state is changed from `JUST_CREATED` to `READY`. The constructor of class `Thread` (see `threads/thread.cc`) simply sets up the thread name (for debugging purposes) and sets `status` to `JUST_CREATED`. The variable `stack` is used to store the bottom of the stack (for stack overflow checking) and variable `stackTop` is the current stack pointer (SP). Other registers including program counter (PC) are all stored in the array `machineState[]`. The size of this array is `MachineStateSize` which is defined to be 18 in line 52 of `thread.h`, although some architectures such as Sparc and MIPS only need to save 10 registers.

We mentioned that a user process is derived from a kernel thread. In the same file `thread.h`, we see that two additional data members of class `Thread` are defined:

```
119 #ifdef USER_PROGRAM
120 // A thread running a user program actually has *two* sets of CPU registers --
121 // one for its state while executing user code, one for its state
122 // while executing kernel code.
123
124     int userRegisters[NumTotalRegs];    // user-level CPU register state
125
126   public:
127     void SaveUserState();               // save user-level register state
128     void RestoreUserState();            // restore user-level register state
129
130     AddrSpace *space;                   // User code this thread is running.
131 #endif
```

user registers: Array `int userRegisters[NumTotalRegs]` is used to save the contents of user registers. Modern computers have two sets of registers:

- system registers used for running the kernel
- user registers used for running user programs

In Nachos, the system registers are saved in array `machineState[]` mentioned above and the user registers in array `int userRegisters[]`.

address space of user program: This is the address space of the user program.

Now the relationship between kernel threads and the user processes becomes clear. Each user process in Nachos starts with a kernel thread. After having loaded the user program and formed the address space in the memory, the kernel thread becomes a user process.

4.5 Thread and Process Scheduling

A thread or process will go through many states during its life-time. Figure 4.1 of the text shows the state transition diagram of a thread or process. Figure 4.4 of the text shows the various queues to hold threads or processes. Among them, the ready queue is used to hold all the thread or processes in the ready state. Other queues are associated with I/O devices to hold the threads or processes in the block state, waiting for the corresponding I/O requests to complete. Threads or processes are moved between those queues by the job scheduler as shown in the Figures 4.5 and 4.6 of the text.

In Nachos, the job scheduler is implemented as an object of class `Scheduler`. Its code can be found in `threads/scheduler.h` and `threads/scheduler.cc`. The methods of this class provide all the functions to schedule threads or processes. When Nachos is started, an object of class `Scheduler` is created and referenced by a global variable `scheduler`. The class definition of `Scheduler` is as follows:

```
20 class Scheduler {
21   public:
22     Scheduler();                          // Initialize list of ready threads
23     ~Scheduler();                         // De-allocate ready list
24
25     void ReadyToRun(Thread* thread);      // Thread can be dispatched.
26     Thread* FindNextToRun();              // Dequeue first thread on the ready
27                                           // list, if any, and return thread.
28     void Run(Thread* nextThread);         // Cause nextThread to start running
29     void Print();                         // Print contents of ready list
30
31   private:
32     List *readyList;            // queue of threads that are ready to run,
33                                 // but not running
34 };
```

The only private data member of this class is the pointer to a `List` object (see `threads/list.h` and `threads/list.cc`). This list is the ready queue to hold all the threads in the `READY` state. Function `ReadyToRun(Thread* thread)` puts the thread at the end of the queue, while function `FindNextToRun()` returns the pointer to the thread removed from the queue (or NULL if the queue is empty).

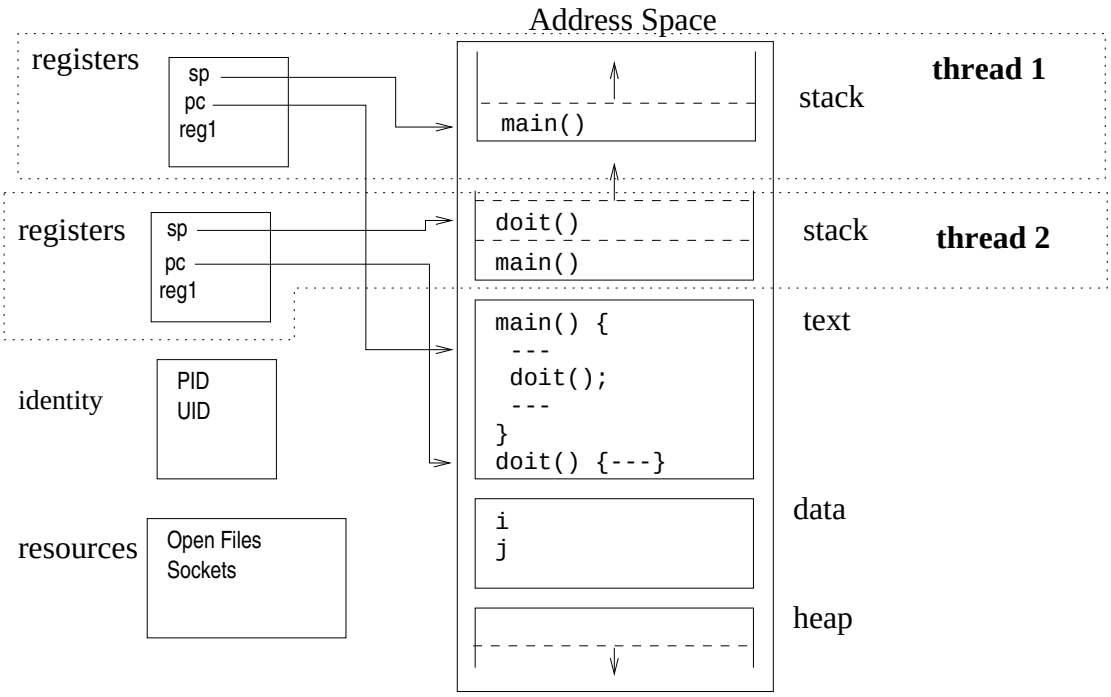

Figure 4.3: Class diagram of Nachos thread scheduling

Perhaps, the most interesting function in Nachos is function `Run(Thread*)` of this class. This function calls the assembly function `SWITCH(Thread*, Thread*)` for the context switch from the current thread (i.e. the thread which calls this function) to another thread pointed by the second argument.

These three functions of the scheduler are used by thread objects to make state transitions.

The relationship among classes `Thread`, `Scheduler` and `List` is shown in the diagram in Figure 4.3. An arrow from class A to class B represent the association: functions of class A will call functions of class B. An arrow with a diamond at its tail represents a whole-part relationship. In our case, a `Scheduler` object includes a `List` object, the ready queue, as its component. The rectangle boxes with a shaded triangles on their upper-right corners show the implementations of corresponding functions.

Read the source code of these functions of `Scheduler` in `threads/scheduler.cc` and make sure that you understand how class `Scheduler` works.

4.6 Thread and Process State Transition

Figure 4.1 of the text shows the state transition diagram for ordinary processes. We have seen that the thread or process in Nachos has similar states defined in `threads/thread.h`:

```
60 // Thread state
61 enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED };
```

Class `Thread` has four functions as follows:

```
93     void Fork(VoidFunctionPtr func, _int arg);  // Make thread run (*func)(arg)
94     void Yield();                               // Relinquish the CPU if any
95                                                 // other thread is runnable
96     void Sleep();                               // Put the thread to sleep and
97                                                 // relinquish the processor
98     void Finish();                              // The thread is done executing
99
```

Function `Fork(VoidFunctionPtr func, _int arg)` is used to make the transition from `JUST_CREATED` to `READY`.

Function `Yield()` is used to make the transition from state `RUNNING` to `READY` if the ready queue is not empty. It puts the current thread (calling thread) back to the end of the ready queue and makes a thread in the ready queue the running thread through context switch. If the ready queue is empty, it does not have effect and the current thread continues to run.

Function `Sleep()` is used to make the transition from `RUNNING` to `BLOCKED` and make a context switch to a thread from the ready queue. If the ready queue is empty, the CPU stays idle until there is a thread ready to run. This `Sleep()` function is usually called when the thread or process starts an I/O request or has to wait for an event. It cannot proceed until the I/0 is finished or the event occurs. Before calling this function, the thread usually puts itself into the queue associate with the corresponding I/O device or event as shown in the Figures 4.4, 4.5 and 4.6 of the text.

Function `Finish()` is used to terminate the thread.

Read the source code of functions `Yield()` and `Sleep()` in `threads/thread.cc` and make sure that you understand the state transition of threads in Nachos.

4.7 Thread Creation

Recall that the `Thread` object construction merely creates the data structure of the object and sets its state to `JUST_CREATED`. This thread is not ready to run yet,

because its stack has not been allocated. Its control block has not been initialized yet either. In particular, it does not have the initial value for PC (program counter) and, therefore, it does not know where to start if it is scheduled to run.

Allocation of stack and initialization of the control block are done by function `Fork(VoidFunctionPtr func, _int arg)` of `Thread` class. Argument `func` is the pointer to the function which the thread is to execute and `arg` is the argument for that function. `arg` could be a pointer and for 64-bit machines it is a 64-bit long integer. This is the reason that its type is `_int` which is translated to 32-bit or 64-bit integer depending on the architecture of the machine to run Nachos.

The code of function `Fork(VoidFunctionPtr func, _int arg)` is as follows:

```
 87 void
 88 Thread::Fork(VoidFunctionPtr func, _int arg)
 89 {
 90 #ifdef HOST_ALPHA
 91     DEBUG('t', "Forking thread \"%s\" with func = 0x%lx, arg = %ld\n",
 92           name, (long) func, arg);
 93 #else
 94     DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",
 95           name, (int) func, arg);
 96 #endif
 97
 98     StackAllocate(func, arg);
 99
100     IntStatus oldLevel = interrupt->SetLevel(IntOff);
101     scheduler->ReadyToRun(this);        // ReadyToRun assumes that interrupts
102                                         // are disabled!
103     (void) interrupt->SetLevel(oldLevel);
104 }
```

This function first calls function `StackAllocate (VoidFunctionPtr, _int )` to allocate the memory space for the stack and initialize the array `machineState[]`.

Let us have a look at `StackAllocate (VoidFunctionPtr, _int )` more closely. Its code is in `threads/thread.cc`. Nachos supports many architectures. In the subsequent discussion, we assume that the host running Nachos is a MIPS machine (although it is actually quite likely that you are running Nachos on a Linux system). The code of this function is as follows:

```
257 void
258 Thread::StackAllocate (VoidFunctionPtr func, _int arg)
259 {
260     stack = (int *) AllocBoundedArray(StackSize * sizeof(_int));
...
```

```
272     stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
...
284     machineState[PCState] = (_int) ThreadRoot;
285     machineState[StartupPCState] = (_int) InterruptEnable;
286     machineState[InitialPCState] = (_int) func;
287     machineState[InitialArgState] = arg;
288     machineState[WhenDonePCState] = (_int) ThreadFinish;
289 }
```

Here function `AllocBoundedArray(int)` (defined in `machine/sysdep.cc`) allocates a stack whose bottom is assigned to variable `stack`. Variable `stackTop` is set to the top of the stack.

For MIPS machines, five macros, `PCState`, `StartupPCState`, `InitialPCState`, `InitialArgState` and `WhenDonePCState`, are defined in `switch.h` and equivalent to `9`, `3`, `0` , `1` and `2`, respectively. `ThreadRoot` is the name of an assembly function defined in `switch.s`. `InterruptEnable` and `ThreadFinish` are two static functions defined in `thread.cc`. As a result, the pointers to these functions are stored in the corresponding places in `machineState[]`. The pointer to the function which the thread is to execute is stored in `machineState[0]` and its argument in `machineState[1]`.

When this thread is scheduled to run for the first time, the value in `machineState[PCState]` which is the reference to function `ThreadRoot`, is loaded into register `ra`. `ra` is called *return address register* and contains the address of the first instruction to execute after the assembly routine is complete. Therefore, the first routine executed by the new thread is `ThreadRoot`. The source code of `ThreadRoot` is as follows:

```
 69         .globl ThreadRoot
 70         .ent    ThreadRoot,0
 71 ThreadRoot:
 72         or      fp,z,z          # Clearing the frame pointer here
 73                                 # makes gdb backtraces of thread stacks
 74                                 # end here (I hope!)
 75
 76         jal     StartupPC       # call startup procedure
 77         move    a0, InitialArg
 78         jal     InitialPC       # call main procedure
 79         jal     WhenDonePC      # when we are done, call clean up procedure
 80
 81         # NEVER REACHED
 82         .end ThreadRoot
```

Macros `StartupPC`, `InitialArg`, `InitialPC` and `WhenDonePC` are MIPS reg-

isters `s3`, `s1`, `s0` and `s2`, respectively (see `switch.h`). The contents of these registers are loaded from

- `machineState[StartupPCState]`,
- `machineState[InitialArgState]`,
- `machineState[InitialPCState]` and
- `machineState[WhenDonePCState]`,

respectively. Therefore, register `StartupPC` contains the reference of static function `InterruptEnable`, register `WhenDonePC` the reference of static function `ThreadFinish`. The reference of the function we want the thread to execute for the real job is in register `InitialPC`, and its argument in register `InitialArg`.

MIPS Instruction

jal Reg

is called "jump and link". It saves the return address in hardware and jumps to the subroutine pointed by register *Reg*. When the subroutine is finished, the control goes back to the saved return address.

Function `ThreadRoot` is actually a wrapper which calls three subroutines: `InterruptEnable`, the function pointed by `InitialPC` and function `ThreadFinish`, in that order. The argument in `InitialArg` is loaded to register `a0` before the call starts.

What function `InterruptEnable` does is simply to enable interrupt. Function `ThreadFinish` calls `currentThread->Finish()` to terminate the thread.

Subroutine `ThreadRoot` defines the activities of the thread during its life time.

4.8 Context Switch

The concept of context switch is described in Section 4.2.3 of the text. Figure 4.3 of the text shows the steps involved in a context switch between two user processes. How is context switch implemented in real operating systems?

In Nachos, a context switch is started by calling function `Run (Thread *)` of class `Scheduler`. The source code of this function is as follows:

```
90 void
91 Scheduler::Run (Thread *nextThread)
92 {
93     Thread *oldThread = currentThread;
94
```

```
 95 #ifdef USER_PROGRAM                    // ignore until running user programs
 96     if (currentThread->space != NULL) { // if this thread is a user program,
 97         currentThread->SaveUserState(); // save the user's CPU registers
 98         currentThread->space->SaveState();
 99     }
100 #endif
101
102     oldThread->CheckOverflow();             // check if the old thread
103                                             // had an undetected stack overflow
104
105     currentThread = nextThread;             // switch to the next thread
106     currentThread->setStatus(RUNNING);      // nextThread is now running
107
108     DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",
109           oldThread->getName(), nextThread->getName());
110
111     // This is a machine-dependent assembly language routine defined
112     // in switch.s.  You may have to think
113     // a bit to figure out what happens after this, both from the point
114     // of view of the thread and from the perspective of the "outside world".
115
116     SWITCH(oldThread, nextThread);
117
118     DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
119
120     // If the old thread gave up the processor because it was finishing,
121     // we need to delete its carcass.  Note we cannot delete the thread
122     // before now (for example, in Thread::Finish()), because up to this
123     // point, we were still running on the old thread's stack!
124     if (threadToBeDestroyed != NULL) {
125         delete threadToBeDestroyed;
126         threadToBeDestroyed = NULL;
127     }
128
129 #ifdef USER_PROGRAM
130     if (currentThread->space != NULL) {         // if there is an address space
131         currentThread->RestoreUserState();     // to restore, do it.
132         currentThread->space->RestoreState();
133     }
134 #endif
135 }
```

Before we get into the detail of this function, we need to talk about a couple of important global variables in the Nachos kernel. There are at least six global variables defined in `threads/system.cc`:

```
11 // This defines *all* of the global data structures used by Nachos.
12 // These are all initialized and de-allocated by this file.
13
14 Thread *currentThread;                  // the thread we are running now
15 Thread *threadToBeDestroyed;            // the thread that just finished
16 Scheduler *scheduler;                   // the ready list
17 Interrupt *interrupt;                   // interrupt status
18 Statistics *stats;                      // performance metrics
19 Timer *timer;                           // the hardware timer device,
20                                         // for invoking context switches
21
22 #ifdef FILESYS_NEEDED
23 FileSystem  *fileSystem;
24 #endif
25
26 #ifdef FILESYS
27 SynchDisk   *synchDisk;
28 #endif
29
30 #ifdef USER_PROGRAM     // requires either FILESYS or FILESYS_STUB
31 Machine *machine;       // user program memory and registers
32 #endif
33
34 #ifdef NETWORK
35 PostOffice *postOffice;
36 #endif
```

Global variable `currentThread` is the pointer to the current running thread. Global variable `scheduler` is the pointer to the `Scheduler` object of the kernel which is responsible for scheduling and dispatching `READY` threads. This `Scheduler` object and others are created when the Nachos kernel is started. The routine to create these objects and initialize the system global variables is the function `Initialize(int argc, char **argv)` in `system.cc`. Other global variables will be clear when we talk about user process and file systems of Nachos.

Function `Scheduler::Run (Thread *nextThread)` first sets variable `oldThread` to the the current running thread (the thread which calls this function) and variable `currentThread` to the next thread. Then, it calls function `SWITCH(..)` at line 116. Note the comment from lines 111-114. Understanding what happens during `SWITCH(..)` call is essential to grasp the concept of context switch.

Function `SWITCH(..)` is implemented in files

```
../threads/switch.h
../threads/switch.s
```

Theses files contain the assembly codes for many hosts including Sun Sparc, MIPS, and Intel i386, etc. As we said, we assume that our Nachos is running on MIPS and we only look at the code for MIPS. In `switch.h`, the following constants are defined:

```
25 /* Registers that must be saved during a context switch.
26  * These are the offsets from the beginning of the Thread object,
27  * in bytes, used in switch.s
28  */
29 #define SP 0
30 #define S0 4
31 #define S1 8
32 #define S2 12
33 #define S3 16
34 #define S4 20
35 #define S5 24
36 #define S6 28
37 #define S7 32
38 #define FP 36
39 #define PC 40
```

In `switch.s`, the register names of MIPS are defined as follows:

```
50 /* Symbolic register names */
51 #define z       $0      /* zero register */
52 #define a0      $4      /* argument registers */
53 #define a1      $5
54 #define s0      $16     /* callee saved */
55 #define s1      $17
56 #define s2      $18
57 #define s3      $19
58 #define s4      $20
59 #define s5      $21
60 #define s6      $22
61 #define s7      $23
62 #define sp      $29     /* stack pointer */
63 #define fp      $30     /* frame pointer */
64 #define ra      $31     /* return address */
```

Here, `$16`, `$17`, $\cdots$ are the names of registers in MIPS. Function `SWITCH(..)` is defined as follows:

```
84          # a0 -- pointer to old Thread
85          # a1 -- pointer to new Thread
```

```
 86          .globl SWITCH
 87          .ent    SWITCH,0
 88 SWITCH:
 89          sw      sp, SP(a0)              # save new stack pointer
 90          sw      s0, S0(a0)              # save all the callee-save registers
 91          sw      s1, S1(a0)
 92          sw      s2, S2(a0)
 93          sw      s3, S3(a0)
 94          sw      s4, S4(a0)
 95          sw      s5, S5(a0)
 96          sw      s6, S6(a0)
 97          sw      s7, S7(a0)
 98          sw      fp, FP(a0)              # save frame pointer
 99          sw      ra, PC(a0)              # save return address
100
101          lw      sp, SP(a1)              # load the new stack pointer
102          lw      s0, S0(a1)              # load the callee-save registers
103          lw      s1, S1(a1)
104          lw      s2, S2(a1)
105          lw      s3, S3(a1)
106          lw      s4, S4(a1)
107          lw      s5, S5(a1)
108          lw      s6, S6(a1)
109          lw      s7, S7(a1)
110          lw      fp, FP(a1)
111          lw      ra, PC(a1)              # load the return address
112
113          j       ra
114          .end SWITCH
```

Here, `a0` and `a1` are the macro names for two argument registers, `$4` and `$5`. In MIPS, registers `$4` and `$5` are used to store the first and second arguments of a function call, respectively. In the case of `SWITCH(Thread*, Thread*)` register `a0` contains the first argument which is the reference of the current thread (see line 116 of `scheduler.cc`) and register `a1` the second argument which is the reference of the next thread. The machine instruction in MIPS

$$sw\ \ Rsrc,\ \ Const(Rindex)$$

stores the word in register *Rsrc* to the memory location whose address is the sum of *Const* and the contests of register *Rindex*. The instruction

$$lw\ \ Rdist,\ \ Const(Rindex)$$

does the reverse: loads the word from the memory location to register *Rdist*. Function `SWITCH(..)` first saves all the important registers of the current thread to the control block of of the current thread. Recall that the first private data member of `Thread` class is `stackTop` followed by `machineState[MachineStateSize]`. In other words, a reference to an object of `Thread` actually points to `stackTop`. Note that the constant offsets `SP, S0, S1` $\cdots$ have fixed values $0, 1, 2, \cdots$. Therefore, the position of `stackTop` and `machineState[]` is important as warned by the comment in lines 79-80 of `thread.h`.

Among the registers saved, register `ra` contains the the return address of the function call. In this case, `ra` contains the address of the instruction right after line 166 in `scheduler.cc`.

The current thread which yields the CPU will gain the CPU again by another context switch eventually. When it is switched back, all the register saved in its `stackTop` and `machineState[]` will be restored to the corresponding registers of the CPU, including the return address register `ra`. The instruction in line 113 makes the control jump to the address stored in `ra`, and the execution of the current thread resumes.

Note also the lines 95-100 and lines 129-134 in function `Run(Thread*)`. These codes are used to save and restore the user registers and the address space of user processes. The entire function `Run(Thread*)` is run by the kernel, because it belongs to the Nachos kernel. This corresponds to the operations drawn in the middle column in Figure 4.3 of the text.

Note the time of the return of function call `Run(Thread*)` to the current thread. It does not return immediately. It won't return until the calling thread is switched back and becomes the current thread again.

4.9 Thread Termination

We have seen from `ThreadRoot` that after the thread finishes its function, control goes to the cleanup function `ThreadFinish` defined in `threads/thread.cc`. This function simply calls function `Finish()` of the thread. Function `Finish()` sets the global variable `threadToBeDestroyed` to point to the current thread and then calls `Sleep()`. The termination of this thread is actually done by the next thread after the context switch. Note the following code in function `scheduler::Run(Thread *)` after the context switch call to `SWITCH(Thread *, Thread *)`:

```
124     if (threadToBeDestroyed != NULL) {
125         delete threadToBeDestroyed;
126         threadToBeDestroyed = NULL;
127     }
```

Whatever thread that resumes after the context switch will find that this thread should be terminated and delete it. The destructor of `Thread` class will deallocate the memory of the stack.

4.10 Operating Systems Kernel

We have covered all aspects of thread and process management of Nachos. We have seen how threads in the Nachos are created, run, and terminated. We also have examined how a context switch is done and how the scheduler of the Nachos system manages multiple runnable threads.

It is time now to have a look at the Nachos kernel itself. As with any other operating systems, the Nachos kernel is part of Nachos operating system that runs on the computer at all times. The smallest Nachos kernel containing thread management only can be compiled in directory `threads` by typing `make` on your LINUX machine. In any case, the kernel is a program itself and it must have a `main(..)` function.

The `main(..)` function of Nachos kernel can be found in `threads/main.cc`. This function does the following:

- call `(void) Initialize(argc, argv)`
- call `ThreadTest()`
- call `currentThread->Finish()`

Function `Initialize(argc, argv)` is defined in `threads/system.cc`. It initializes all the global variables by the creating corresponding objects in Nachos such as the job scheduler, the timer, etc.

`ThreadTest()` is a test function defined in `thread/threadtest.cc` as follows:

```
41 void
42 ThreadTest()
43 {
44     DEBUG('t', "Entering SimpleTest");
45
46     Thread *t = new Thread("forked thread");
47
48     t->Fork(SimpleThread, 1);
49     SimpleThread(0);
50 }
```

This function creates a new thread named “forked thread” to execute function `SimpleThread` with argument 1. The thread executing the main function executes the same function with argument 0 at line 49 above.

Function `SimpleThread` (defined in `threadtest.cc`) simply calls function `Yield()` five times:

```
24 void
25 SimpleThread(int which)
26 {
27     int num;
28
29     for (num = 0; num < 5; num++) {
30         printf("*** thread %d looped %d times\n", which, num);
31         currentThread->Yield();
32     }
33 }
```

Therefore, these two threads are making context switches back and forth until they are terminated. After we start the Nachos kernel as UNIX process on a UNIX machine, the screen output is as follows:

```
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 130, idle 0, system 130, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

The output shows that the two threads are running concurrently. They take turns to print out the messages in the loop.

The last function call of `main(..)` is `currentThread->Finish()`. The function call never returns, because this thread will be deleted by the next thread after the context switch. The Nachos kernel exits the UNIX system only when all the threads are terminated.

4.11 Questions

The following questions combine the concepts covered by the text with the Nachos system. Answer all these questions to test whether you understand the concepts in this module.

1. Check PC and SP in both Figures 4.1 and 4.2 to see if they point to the right positions of the memory. Why are they correct or wrong? Explain how the two threads in Figure 4.2 share the same "text" of the address space of the process.

2. Are Nachos threads kernel threads or user threads, if

   - Nachos runs on a raw hardware or
   - Nachos runs on a UNIX system?

3. Line 81 the code of `ThreadRoot` as follows comments that the control (or PC) will never reach this line. Why is this true?

```
69          .globl ThreadRoot
70          .ent    ThreadRoot,0
71  ThreadRoot:
72          or      fp,z,z          # Clearing the frame pointer here
73                                  # makes gdb backtraces of thread stacks
74                                  # end here (I hope!)
75
76          jal     StartupPC       # call startup procedure
77          move    a0, InitialArg
78          jal     InitialPC       # call main procedure
79          jal     WhenDonePC      # when we are done, call clean up procedure
80
81          # NEVER REACHED
82          .end ThreadRoot
```

4. Suppose that thread A calls function `Run(Thread *nextThread)` and `nextThread` points to thread B. Within the this function, the assembly function `SWITCH(..)` is called as follows:

```
115
116     SWITCH(oldThread, nextThread);
117
```

(a) What is unique about this function call?

(b) From the machine's point of view, what thread does this function call return to?

(c) From the viewpoint of thread A, when and how does this function call return?

Chapter 5

Process Synchronization

5.1 Introduction

Concurrent processes or threads need synchronization for cooperative work if they access shared data. This module addresses this important issue. The material used in this module is Chapter 7 of the text, except for Section 7.9, as well as the Nachos source code for various synchronization primitives. The material in Section 7.9 of the text (Atomic Transactions) is not used in this course.

There are basically three different synchronization mechanisms to control access to shared data by concurrent processes or threads:

1. Semaphore
2. Critical Regions
3. Monitor

Semaphore is a low-level synchronization mechanism and often used as building block for the creation of higher level mechanisms such as critical region and monitor. Monitor is widely used as a high-level synchronization mechanism. It has modular structure and easy-to-understand semantics. Critical regions are less popular and we do not discuss them in this course.

As a first step, you need to read the whole Chapter except Sections 7.6 and 7.9.

In the following sections, we will go through the major concepts in this module. We also discuss the implementation of semaphores, locks and condition variables in Nachos to further illustrate the concepts. Just as you did in Module 4, you need to go back and forth between the text and the Nachos source code several times before you can really understand how synchronization primitives are implemented.

5.2 Need for Synchronization

The section 7.1 of the text explains why concurrent processes need synchronization if they access shared data. You have probably become used to thinking "sequentially" when reasoning about program execution. This is all right if the process does not share data with other processes. But, when multiple concurrent processes modify shared data, race conditions may occur if the access of the shared data is not controlled properly by synchronization.

The example of a shared bounded buffer, introduced in Section 4.4 of the text, shows the need for synchronization.

5.3 The Critical-Section Problem

Section 7.2 of the text discusses a common model of accessing shared data among concurrent processes known as the *critical section*. The model is illustrated in Figure 7.1 of the text. The key problem is how to design the code in boxes the "entry section" and "exit section" properly so that the three requirements, namely *mutual exclusion*, *progress* and *bounded waiting* are satisfied. This section first concentrates on the two-process version of this problem showing three progressively more complex and effective algorithms. You may be surprised by the fact that even for the simple problem with only two processes the design which solves the critical section problem is not trivial. The algorithm in Section 7.2.2 of the text for multiple processes is quite complicated. Make sure you understand the algorithm and, more importantly, why it is correct.

5.4 Synchronization Hardware

Section 7.3 of the text provides solutions to the critical section problem through hardware support. You need to understand Test-and-Set and Swap instructions and how they are used to implement the mutual exclusion in the critical-region problem.

On system with just one processor, synchronization problems can be addressed by freezing interrupts during critical operations, however this approach cannot solve the problem of mutual execution of concurrent processes running on multiple processors. This is a major reason for making use of hardware solutions for synchronization.

The major problem of hardware solutions is that they rely on busy-waiting. Busy-waiting wastes CPU cycles in repeated testing in the while loop. For this reason, in multiprocess or systems, a combination of hardware and software ap-

proaches to synchronization is essential for achieving high levels of processor utilization.

5.5 Semaphore

Section 7.4 of text introduces probably the most important and the most widely-used software mechanism for process synchronization: the semaphore.

Operations *wait*(S) and *signal*(S) of a semaphore are usually denoted as $P(S)$ and $V(S)$, respectively, in other literature. It is probably the most notable problem in this text that the authors chose to use *wait*(S) and *signal*(S), instead of $P(S)$ and $V(S)$, for the names of the two operations of semaphore. This is because the name *wait* can be easily confused with the name, also *wait*, for an operation on condition variables. For this reason, we use $P(S)$ and $V(S)$ for the name of semaphore operations in this study book from now on.

Two implementations of semaphores are provided by the text:

- the spin-lock based on busy-waiting shown in page 201,
- the software implementation using a waiting queue inside the semaphore. The algorithm is shown on page 203.

It is important to realize that both implementations require that both the $P(S)$ and $V(S)$ operations be atomic. On uniprocess machines, this requirement is normally satisfied by disabling the hardware interrupt when the CPU is running these functions. Without this hardware support, it would be impossible to implement semaphores in software.

Nachos's implementation of semaphore uses a slightly different algorithm. It is very inspiring to see what is the difference and why both algorithms are correct. This difference is directly related to another question: what is the semantics of semaphore exactly. In the following discussion, we are going to answer this question by proving that both algorithms guarantee the invariants of semaphores — the assertions about semaphore in any circumstances. These assertions define the semantics of semaphore.

5.5.1 Semaphore Algorithm in The Text

The algorithm of semaphore presented in the text can be described by the following pseudo-code:

- `P()`

```
v = v − 1;
if (v < 0) {
    block the current process after putting it in the queue L;
}
```

- `V()`

  $v = v + 1$;
  if ($v \leq 0$) {
      remove a process from the queue L;
      and put it in the system READY queue;
  }

This algorithm is called Algorithm 1 in the following discussion.

5.5.2 Semaphore Algorithm in Nachos

Before we present the semaphore algorithm use by Nachos, let us have a look at the implementation of semaphores in Nachos.

A semaphore in Nachos is implemented as an object of class `Semaphore` whose definition can be found in `threads/synch.h` as follows:

```
40 class Semaphore {
41   public:
42     Semaphore(char* debugName, int initialValue);       // set initial value
43     ~Semaphore();                                       // de-allocate semaphore
44     char* getName() { return name;}                     // debugging assist
45
46     void P();    // these are the only operations on a semaphore
47     void V();    // they are both *atomic*
48
49   private:
50     char* name;        // useful for debugging
51     int value;         // semaphore value, always >= 0
52     List *queue;       // threads waiting in P() for the value to be > 0
53 };
```

The private variable `queue` is the pointer to the linked-list queue to hold all the threads blocked at the semaphore.

The implementation of `P()` can be found in `threads/synch.cc` as follows:

```
64 void
65 Semaphore::P()
66 {
67     IntStatus oldLevel = interrupt->SetLevel(IntOff);    // disable interrupts
68
69     while (value == 0) {                          // semaphore not available
```

```
70          queue->Append((void *)currentThread);   // so go to sleep
71          currentThread->Sleep();
72      }
73      value--;                                    // semaphore available,
74                                                  // consume its value
75
76      (void) interrupt->SetLevel(oldLevel);       // re-enable interrupts
77  }
```

Function `V()` is implemented as follows from the same file:

```
87  void
88  Semaphore::V()
89  {
90      Thread *thread;
91      IntStatus oldLevel = interrupt->SetLevel(IntOff);
92
93      thread = (Thread *)queue->Remove();
94      if (thread != NULL)    // make thread ready, consuming the V immediately
95          scheduler->ReadyToRun(thread);
96      value++;
97      (void) interrupt->SetLevel(oldLevel);
98  }
```

Notice that both functions are wrapped by `IntStatus oldLevel = interrupt->SetLevel(IntOff)` which disables the hardware interrupt and `(void) interrupt->SetLevel(oldLevel)` which restores the original interrupt level. Therefore, the atomicity of these functions is guaranteed.

Using the same format of pseudo-code as Algorithm 1, the semaphore algorithm used by Nachos can be described as follows:

- `P()`

  while $(v = 0)$ {
  block the calling process after putting it in the queue L;
  }
  $v = v - 1$;

- `V()`

if (the queue L is non-empty) {
    remove a process from the queue L;
    put it in the system READY queue;
}
$v = v + 1$;

This semaphore algorithm is referred to as Algorithm 2 in the further discussion.

5.5.3 Why Both Are Correct?

In both Algorithms 1 and 2, v is the integer variable of the semaphore and L the queue to hold the processes blocked at the semaphore. By "blocking the calling process" we mean that the process calling `P()` stops running and becomes a blocked process. The consequent context switch picks up a process from the system ready queue to run it. Both `P()` and `V()` operations are atomic, i.e. the hardware interrupt is turned off during `P()` or `V()` operations and instruction interleaving is not possible.

In Algorithm 1, `V()` always increments v and the calling process never blocks. `P()` decrements v first and the calling process blocks if the resulting value of v is negative. Let nV_c and nP_s denote the numbers of `V()` *completed* and `P()` operations *started*, respectively. Let I denote the initial value of v. Since both *V()* and *P()* are atomic, we have the following invariant about the value of v:

$$v = I + nV_c - nP_s \tag{5.1}$$

Let the numbers of `P()` operations *completed* and *blocked* are denoted by nP_c and nP_b, respectively. It is obvious that the following is always true:

$$nP_s = nP_b + nP_c \tag{5.2}$$

Note that processes for `P()` woken up by `V()` should be regarded as *completed* even when they are in the system ready queue.

Since a process calling `P()` blocks if it finds $v < 0$ after the decrement, the absolute value of the negative v is the number of processes blocked at the semaphore. We can have the following formula for nP_b:

$$nP_b = \begin{cases} -v & \text{if } nP_s - (I + nV_c) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

To get the abstract invariant, we need to eliminate v from the above formulas. By substituting (5.2) in (5.1), we will have:

$$I + nV_c - nP_c = v + nP_b$$

According to (5.3), $v + nP_b \geq 0$ holds. Hence, we reach the invariant about the relationship between nV_c and nP_c:

$$I + nV_c \geq nP_c \tag{5.4}$$

(5.3) can be re-written without referring to v as follows:

$$nP_b = \begin{cases} nP_s - (I + nV_c) & \text{if } nP_s - (I + nV_c) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

As a result, the implementation-independent invariants of semaphore are expressed by (5.4), (5.2) and (5.5).

Algorithm 2 is different from Algorithm 1 in the following accounts:

- Algorithm 1 decrements or increments integer variable v before testing for blocking or waking up; while Algorithm 2 changes v after the testing.
- Algorithm 1 uses the **if** statement for the testing in `P()`, while Algorithm 2 uses the **while** statement for the testing in `P()`.
- The test condition in `P()` of Algorithm 1 is $(v < 0)$, while the corresponding test condition in Algorithm 2 is $(v = 0)$

For Algorithm 2, since the value of v is changed in the last statement of both operations `P()` and `V()`, the following equation should be true:

$$v = I + nV_c - nP_c \tag{5.6}$$

The key aspect of Algorithm 2 is to keep $v \geq 0$, because it will translate (5.6) to the abstract invariant (5.4). To maintain $v \geq 0$, we must use the **while** statement for the testing in `P()`. If **if** statement were used in place of the **while** statement, some race conditions may cause $v < 0$. To see this possibility, assume that $I = 0$ and consider the following scenario of events:

1. Process A issues a `P()` and becomes blocked at the semaphore.
2. Process B issues a `V()` and wakes up and put process A into the system ready queue.
3. Process B increments v from 0 to 1.
4. Process C issues a `P()`, decrements v from 1 to 0 and exits the `P()`,
5. Process B is dispatched to run on the CPU. Because there is no re-testing as implied by the **if** statement, the resumed process B goes ahead to decrement v from 0 to -1.

With the **while** statement in `P()`, it is guaranteed that v is greater than 0 (recall the initial value of v is always non-negative) before v is decremented. Therefore $v \geq 0$ is always true and so is the invariant 5.4.

Note that the processes calling `P()` that are woken by `V()` and put in the system ready queue should still be regarded as *blocked*, because they cannot be counted as *completed* until they finish decrementing v. For instance, in the step 5 of the above scenario, process B finds $v = 0$ according to the **while** loop and blocks again. Therefore, the invariants (5.2) and (5.5) still hold for Algorithm 2.

We have shown that Algorithms 1 and 2 both maintain the abstract invariants in (5.4), (5.2) and (5.5) and therefore, both are correct.

5.5.4 Use of Semaphores

Semaphores can be used to implement critical sections in concurrent programs.

Semaphores can also be used for event posting-and-waiting synchronization. The details can be found in the section 7.4.1 of the text. The bounded buffer algorithms in the Figures 7.12 and 7.13 of the text use semaphores for this kind of synchronization.

Deadlock is possible in concurrent programs using semaphores. An example of deadlock caused by improper use of semaphores can be found in Section 7.4.3 of the text.

Starvation is also possible in the concurrent programs using semaphores. It depends on how the waiting queue of the semaphore is implemented.

5.6 Locks

Locks are another kind of low lever synchronization primitive which is similar (but different) to binary semaphore. A binary semaphore is a semaphore whose integer value cannot exceeds 1.

A lock L has two operations, *Acquire*() and *Release*(), which are similar to but different from $P()$ and $V()$ operations on binary semaphore S. A lock has only two states: *acquired* and *released*. The initial state must be *released*. The operations *Release*() on a *released* lock results in an error. The operation *Acquire*() on a *acquired* lock puts the calling process in the wait queue of the lock. Only the process that has acquired the lock can execute *Release*(); otherwise an error occurs. As with $P()$ and $V()$ of semaphore, both *Acquire*() and *Release*() should be atomic.

Clearly, a lock can be implemented by using a binary semaphore.

A lock in Nachos is implemented as an object of class `Lock` defined in `threads/synch.h`:

```
 67 class Lock {
 68   public:
 69     Lock(char* debugName);              // initialize lock to be FREE
 70     ~Lock();                            // deallocate lock
 71     char* getName() { return name; }    // debugging assist
 72
 73     void Acquire(); // these are the only operations on a lock
 74     void Release(); // they are both *atomic*
 75
 76     bool isHeldByCurrentThread();       // true if the current thread
 77                                         // holds this lock.  Useful for
 78                                         // checking in Release, and in
 79                                         // Condition variable ops below.
 80
 81   private:
 82     char* name;                         // for debugging
 83     Thread *owner;                      // remember who acquired the lock
 84     Semaphore *lock;                    // use semaphore for the actual lock
 85 };
```

As can be seen, the lock in Nachos is actually implemented by using a Nachos semaphore. The code for `Acquire()` and `Release()` is as follows (`threads/synch.cc`):

```
133 void Lock::Acquire()
134 {
135     IntStatus oldLevel = interrupt->SetLevel(IntOff);  // disable interrupts
136
137     lock->P();                              // procure the semaphore
138     owner = currentThread;                  // record the new owner of the lock
139     (void) interrupt->SetLevel(oldLevel); // re-enable interrupts
140 }
...
147 void Lock::Release()
148 {
149     IntStatus oldLevel = interrupt->SetLevel(IntOff);  // disable interrupts
150
151     // Ensure: a) lock is BUSY  b) this thread is the same one that acquired it.
152     ASSERT(currentThread == owner);
153     owner = NULL;                           // clear the owner
154     lock->V();                              // vanquish the semaphore
155     (void) interrupt->SetLevel(oldLevel);
156 }
```

Lines 138 and 152 are used make sure that the thread calling `Release()` is the

same thread which has acquired the lock. When the lock is in the *released* state, the variable "`owner`" is set to `NULL`. If any thread tries to call `Release()` of this *released* lock, error will occurs at line 152.

Of course, the initial value of the semaphore "`lock`" should be 1. As a matter of fact, this value will never exceeds 1. Therefore, this Nachos semaphore is made to behave like a binary semaphore.

5.7 Classical Synchronization Problems

Three classical synchronization problems are:

- the bounded-buffer problem,
- the readers and writers problem and
- the dining-philosophers problem.

They are discussed in the section 7.5 of the text. You need to understand these problems and how they can be solved by using semaphores.

It is important to understand why deadlocks are possible in the dining-philosophers program relying on semaphores only.

5.8 Monitors

A monitor is a high-level abstract data type which support mutual exclusion among its functions implicitly. In conjunction with condition variables, another low level synchronization primitives, monitors can solve many synchronization problems which cannot be solved by using low level synchronization primitives only. One example is the dining-philosopher problem for which monitors can offer a deadlock-free solution.

5.8.1 Mutual Exclusion

A monitor guarantees the mutual exclusive execution of its functions. It implies that the implementation of a monitor may have a lock or a binary semaphore to synchronize the concurrent processes invoking its functions. This part of the implementation is illustrated by Figure 7.20 of the text.

5.8.2 Condition Variables

Monitors would not be very useful without condition variables. Most synchronization problems require condition variables to be declared within a monitor. Condition variables provide a synchronization mechanism between the concurrent processes which invoke the functions of the monitor.

A condition variable represents the condition of a monitor for which several concurrent processes may be waiting. If the condition is found to be true by a process, this process may wake up one of the waiting processes. Obviously, a condition variable needs a waiting queue to hold these waiting processes.

Figure 7.21 of the text illustrates the relationship between a monitor and condition variables.

A condition variable has two operations: *wait* and *signal*. Depending on whether the process invoking *signal* continues to run, there are two styles of condition variables:

- Mesa Style: the invoking process continues to run and the woken process resumes after the process invoking *signal* blocks or leaves the monitor,
- Hoare Style: the woken process resumes immediately and the process invoking *signal* blocks until the woken process blocks or leaves the monitor.

Condition variables can be implemented by using semaphores. The algorithm for implementing Hoare style condition variables with semaphores can be found in Section 7.7 of the text. Make sure that you understand the algorithm.

Nachos provides an implementation of Mesa style condition variables as objects of class `Condition`. The definition of this class is as follows:

```
119 class Condition {
120   public:
121     Condition(char* debugName);         // initialize condition to
122                                         // "no one waiting"
123     ~Condition();                       // deallocate the condition
124     char* getName() { return (name); }
125
126     void Wait(Lock *conditionLock);     // these are the 3 operations on
127                                         // condition variables; releasing the
128                                         // lock and going to sleep are
129                                         // *atomic* in Wait()
130     void Signal(Lock *conditionLock);   // conditionLock must be held by
131     void Broadcast(Lock *conditionLock);// the currentThread for all of
132                                         // these operations
133
134   private:
```

```
135     char* name;
136     List* queue;  // threads waiting on the condition
137     Lock* lock;   // debugging aid:  used to check correctness of
138                   // arguments to Wait, Signal and Broacast
139 };
```

The argument `Lock *conditionLock` for both `Wait(..)` and `Signal(..)` is the lock used by the monitor for the mutual exclusion. As can be seen, Nachos uses a queue directly instead of a semaphore to hold the waiting processes.

The `Wait(Lock*)` function is as follows (see `threads/synch.cc`):

```
206 void Condition::Wait(Lock* conditionLock)
207 {
208     IntStatus oldLevel = interrupt->SetLevel(IntOff);
209
210     ASSERT(conditionLock->isHeldByCurrentThread());  // check pre-condition
211     if(queue->IsEmpty()) {
212         lock = conditionLock;  // helps to enforce pre-condition
213     }
214     ASSERT(lock == conditionLock); // another pre-condition
215     queue->Append(currentThread);  // add this thread to the waiting list
216     conditionLock->Release();      // release the lock
217     currentThread->Sleep();        // goto sleep
218     conditionLock->Acquire();      // awaken: re-acquire the lock
219     (void) interrupt->SetLevel(oldLevel);
220 }
```

An important pre-condition for this function is that the calling process must be the process which acquired the lock. This pre-condition is checked in line 210. Another pre-condition is that the lock passed to this function must be the same lock used by the monitor. Checking this pre-condition is done jointly by lines 211-214.

The implementation of `Signal(Lock *conditionLock)` is as follows:

```
229 void Condition::Signal(Lock* conditionLock)
230 {
231     Thread *nextThread;
232     IntStatus oldLevel = interrupt->SetLevel(IntOff);
233
234     ASSERT(conditionLock->isHeldByCurrentThread());
235     if(!queue->IsEmpty()) {
236         ASSERT(lock == conditionLock);
237         nextThread = (Thread *)queue->Remove();
238         scheduler->ReadyToRun(nextThread);      // wake up the thread
```

```
239     }
240     (void) interrupt->SetLevel(oldLevel);
241 }
```

Note that the argument `conditionLock` for this function is only for checking the pre-conditions which are the same as `Wait(..)`. The calling process just removes the woken process from the queue and puts it in the system ready queue. The calling process is expected to release the lock after it leaves the monitor or calls *Wait*(..) of another condition variable. The woken process tries to acquire the lock again as shown in line 218 of the above code.

As can been seen, the implementation of Mesa style condition variable is much simpler than Hoare style. It does not need a queue to hold the processes which are blocked after they call *signal*. That queue is implemented in the Hoare style algorithm in the text by using a semaphore called *next*.

5.8.3 Implementation of Monitor

If only Mesa style condition variables are used, the implementation of the monitor is simple. The monitor only needs a lock and its functions can be made mutually exclusive by calling *Acuquire* and *Release* of the lock at the beginning and end, respectively, of each function.

A monitor designed to use Hoare style condition variables needs an additional queue as part of its implementation. In the algorithm presented in the text, this queue is implemented by a semaphore called *next*. The code for each function is shown in Page 220 of the text. Be aware of the confusion caused by the authors of the text by using same names, *wait* and *signal*, for the two operations of both semaphores and condition variables. In all the code sections on page 220 and 221 of the text, all the *wait* and *signal* operations are *P* and *V* operations of the semaphores, respectively. These code sections are the algorithms to implement Hoare style monitor functions and the *wait* and *signal* operations of Hoare style condition variables.

5.8.4 Use of Monitors

The use of monitor is illustrated in the section 7.7 of the text. In particular, the dining-philosopher problem is recoded and solved by a monitor shown in Figure 7.22 of the text. This implementation of the dining-philosopher problem rules out the possibility of deadlock. Make sure that you understand this implementation and why deadlock is not possible anymore.

In the following, we show a synchronized linked list class in Nachos called `SynchList` to be accessed by multiple threads. This class is actually implemented as a monitor with Mesa style condition variables. As a matter of fact, this class *is* a Mesa style monitor.

The definition of `SynchList` is as follows (`threads/synchlist.h`):

```
24 class SynchList {
25   public:
26     SynchList();                // initialize a synchronized list
27     ~SynchList();               // de-allocate a synchronized list
28
29     void Append(void *item);    // append item to the end of the list,
30                                 // and wake up any thread waiting in remove
31     void *Remove();             // remove the first item from the front of
32                                 // the list, waiting if the list is empty
33                                 // apply function to every item in the list
34     void Mapcar(VoidFunctionPtr func);
35
36   private:
37     List *list;                 // the unsynchronized list
38     Lock *lock;                 // enforce mutual exclusive access to the list
39     Condition *listEmpty;       // wait in Remove if the list is empty
40 };
```

Note that it has a lock and a condition variable.

Function `Remove()` is implemented in `synchlist.cc` as follows:

```
70 void *
71 SynchList::Remove()
72 {
73     void *item;
74
75     lock->Acquire();                    // enforce mutual exclusion
76     while (list->IsEmpty())
77         listEmpty->Wait(lock);          // wait until list isn't empty
78     item = list->Remove();
79     ASSERT(item != NULL);
80     lock->Release();
81     return item;
82 }
```

Mutual exclusion is realized by using the lock. If the list is empty, the thread calls `Wait(lock)` on the condition variable `ListEmpty`. Note that the lock ac-

quired by this thread has to be released so that other threads blocked in the lock can continue. This is why we need to pass the lock pointer `lock` to function `Wait()`.

Symmetrically, function `Append()` is implemented as follows:

```
53 void
54 SynchList::Append(void *item)
55 {
56     lock->Acquire();            // enforce mutual exclusive access to the list
57     list->Append(item);
58     listEmpty->Signal(lock);    // wake up a waiter, if any
59     lock->Release();
60 }
```

Note both functions call `Acquire` and `Release` of the lock at the beginning and end, respectively, to ensure mutual exclusion.

5.9 Questions

1. Explain why starvation is possible if the waiting queue of semaphore is implemented by using the LIFO order.

2. Provide another example showing that incorrect results may occur when producer and consumer processes run the programs in page 190 of the text.

3. (Question 7.7 of the text) If the $P()$ and $V()$ operations of semaphore are not executed atomically, show how the mutual exclusion intended in the code in Figure 7.11 of the text may be violated.

4. Explain why the bounded buffer algorithms in Figures 7.12 and 7.13 of the text are correct.

5. If we change line 69 of function `P()` in the Nachos implementation of semaphore from `while (value == 0) {` to `if (value == 0) {` , is the semaphore implementation still correct? Why?

   If we change the second line of the algorithm of $P(S)$ (known as $voidwait(semaphoreS)$) of semaphore on page 203 of the text from **if** $(S.value < 0)$ to **while** $(S.value < 0)$, is the algorithm still correct? Why?

6. Implement Hoare-style condition variables in Nachos following the steps described below. Use name `ConditionH` for the class to avoid the clash with the existing Mesa style condition variable class. You can either use semaphore for the implementation using the algorithm in the text or implement them by using linked list queues directly.

(a) Figure out what private data members class `ConditionH` should have.

(b) Figure out the signatures of functions `Wait()` and `Signal()`.

(c) Write down the algorithms for functions `Wait()` and `Signal()`. (When implementing them using linked list queue directly).

(d) Code `Wait()` and `Signal()`.

(e) Design test programs and test your implementations.

7. Explain why the Hoare style condition variables degenerate to the Mesa style condition variables if if operation *Signal*() can only appear as the last statement in all functions of a monitor.

8. Implement the monitor for the dining-philosopher problem shown in Figure 7.22 of the text in Nachos. Use the Mesa style condition variables in Nachos first. Then use the Hoare style condition variables you implemented in the previous question.

9. Write a monitor for the bounded-buffer problem. Implement this monitor in Nachos using

   (a) the existing Mesa style condition variables

   (b) the Hoare style condition variables you implemented previously.

Part III

Storage Management

Starting from the next module, we discuss storage management of operating systems. There are two levels of storage managed by operating systems: memory and file systems.

We start with memory management in Module 6. The material is from the chapter 8 of the text. In Module 7, we discuss implementation of system calls. Module 8 covers virtual memory which is the topic of the chapter 9 of the text. File systems interfaces and Implementation are addressed in Modules 9 and 10, respectively. The material of Module 9 is from the chapter 10 of the text. The material of Module 10 is largely from the chapter 11 of the text. Part of the chapter 12 of the text is also used in Module 10.

As we did in the previous modules, we will continue to use the Nachos implementation to illustrate the concepts whenever possible in these modules.

Chapter 6

Memory Management

We said earlier that each user process has an address space. This address space (at least part of it) has to reside in the memory of the computer when the process is running. This module addresses how to implement the address spaces of multiple concurrent processes in a single physical memory.

This module also uses Nachos to illustrate how the address space is implemented in the MIPS "physical memory" simulated by the MIPS simulator.

The material used in this module is Chapter 9 of the text and the Nachos source code in directories `machine`, `bin` and `userprog`.

6.1 From Programs To Address Space

What should be done in order to run the programs of an application on a computer? Figure 9.1 of the text shows that there are three steps involved:

1. A *Compiler* translates the source code of each module to its `object` module,
2. A *linker* links all the object modules to form a single binary executable also known as `load module`,
3. A *loader* loads the load module in the memory of the computer.

Each object module has its own sections of text, initialized data, symbol table and relocation information. The task of the linker is to merge all the object modules into one load module with all cross references between the object modules and libraries resolved. This requires relocating the object modules by changing the address referenced within each object module as well as resolving external references between them.

The load module represents the logical address spaces of the program. The starting address of this space is 0 and all the addresses referenced in the module are relative to this zero starting address.

This logical address space needs to be translated to a physical address space before the program can run. This translation is usually done by the hardware of *memory management unit* as shown in Figure 9.5 of the text. As can be seen from this figure, the operating system can relocate the load module of the program to different regions of the memory by varying the value of its *relocation register*.

The concepts summarized above are described in greater detail in Sections 9.1 and 9.2 of the text. Make sure that you understand all of them.

6.2 Memory Management Schemes

From Section 9.3 to Section 9.6 of the text, four schemes of translating logical address space to physical address space are described:

1. Contiguous Allocation
2. Paging
3. Segmentation
4. Segmentation with Paging

Read all these sections and make sure that you understand each of the schemes as well as all the related concepts such as page, page frames, TLB, etc. In particular, the fundamental problem known as the *classical dynamic storage allocation problem* in any contiguous storage allocation system motivates all the advanced schemes such as paging and segmentation with paging.

6.3 Swapping

Swapping is necessary when the physical memory cannot accommodate all the address spaces of concurrent processes (if virtual memory is not supported). The concepts related to swapping and its impact on performance are described in Section 9.2 of the text. Read the section and make sure that you understand all the concepts.

In the following discussion, we describe the relevant part of Nachos to illustrate the concepts. We assume that you have read all the sections of the text mentioned above and understand the concepts covered.

Before we discuss address space and address translation in Nachos, we need to talk about the MIPS machine simulator in Nachos. This is because the user programs supported by Nachos at the moment are binary executables for MIPS machines. To enable experiments with MIPS user programs on a Nachos kernel which is not running on a MIPS machine (this is the case in your situation where an Intel x86 machine is used), the Nachos distribution comes with a MIPS simulator derived from J. Larus' SPIM simulator.

6.4 MIPS Simulator

6.4.1 Components of MIPS Machine

A MIPS machine simulator used to run user programs in Nachos is implemented as an object class `Machine` defined in lines 107-190 of `machine/machine.h`. The major components of the machine are:

- user registers: `int registers[NumTotalRegs];`
- main memory: `char *mainMemory;`
- page table for the current user process: `TranslationEntry *pageTable;`
- tlb (table look-ahead buffer): `TranslationEntry *tlb;`

The machine operations are simulated by various functions as follows:

- memory read and write:

```
129     bool ReadMem(int addr, int size, int* value);
130     bool WriteMem(int addr, int size, int value);
```

- register read and write

```
116     int ReadRegister(int num);  // read the contents of a CPU register
117
118     void WriteRegister(int num, int value);
```

- instruction interpretation

```
114     void Run();                 // Run a user program
...
124     void OneInstruction(Instruction *instr);
125                                 // Run one instruction of a user program.
```

- exception handling

```
142     void RaiseException(ExceptionType which, int badVAddr);
```

- address translation

```
135     ExceptionType Translate(int virtAddr, int* physAddr, int size,bool writing);
```

6.4.2 Instruction Interpretation

Function `Run()` in line 30-45 of `machine/mips.cc` sets the machine into user-mode and starts simulating CPU instruction fetch and execution in a infinite loop. An instruction fetch and execution cycle is simulated by function `OneInstruction(Instruction *)` shown as follows (see `machine/mips.cc`):

```
 93 void
 94 Machine::OneInstruction(Instruction *instr)
 95 {
 96     int raw;
 97     int nextLoadReg = 0;
 98     int nextLoadValue = 0;      // record delayed load operation, to apply
 99                                 // in the future
100
101     // Fetch instruction
102     if (!machine->ReadMem(registers[PCReg], 4, &raw))
103         return;                 // exception occurred
104     instr->value = raw;
105     instr->Decode();
106
...
122     // Execute the instruction (cf. Kane's book)
123     switch (instr->opCode) {
124
125       case OP_ADD:
...
547       case OP_UNIMP:
548         RaiseException(IllegalInstrException, 0);
549         return;
550
551       default:
552         ASSERT(FALSE);
553     }
554
```

```
555     // Now we have successfully executed the instruction.
556
557     // Do any delayed load operation
558     DelayedLoad(nextLoadReg, nextLoadValue);
559
560     // Advance program counters.
561     registers[PrevPCReg] = registers[PCReg];    // for debugging, in case we
562                                                 // are jumping into lala-land
563     registers[PCReg] = registers[NextPCReg];
564     registers[NextPCReg] = pcAfter;
565 }
```

After fetching (line 102) and decoding (line 105) the instruction, CPU proceeds to execute it in a big `switch` statement from line 123 to line 554. If the instruction is executed successfully, lines 558-564 advance the program counter of the CPU to make it ready for next instruction. MIPS is a RISC machine which supports delayed load. The details of this feature is not important for our purpose.

6.4.3 MIPS Instruction Set

The big `switch` statement above interprets most MIPS instructions. Details about the MIPS instruction set can be found in the article "SPIM S20: A MIPS R2000 Simulator" by James R. Larus in the Selected Readings of this course.

6.5 Nachos User Programs

Binary executables of user MIPS programs in Nachos are made in the following two steps:

- Use `gcc` MIPS cross compiler to produce the executable in the normal UNIX COFF format,
- Use program `coff2noff` made in directory `../bin/` to convert it to the Nachos NOFF format.

NOFF format is the format of binary executables used by Nachos. It is similar to COFF format. NOFF format is defined in `bin/noff.h` as follows:

```
1 /* noff.h
2  *     Data structures defining the Nachos Object Code Format
3  *
4  *     Basically, we only know about three types of segments:
5  *      code (read-only), initialized data, and unitialized data
```

```
 6  */
 7
 8 #define NOFFMAGIC       0xbadfad        /* magic number denoting Nachos
 9                                          * object code file
10                                          */
11
12 typedef struct segment {
13   int virtualAddr;              /* location of segment in virt addr space */
14   int inFileAddr;               /* location of segment in this file */
15   int size;                     /* size of segment */
16 } Segment;
17
18 typedef struct noffHeader {
19    int noffMagic;               /* should be NOFFMAGIC */
20    Segment code;                /* executable code segment */
21    Segment initData;            /* initialized data segment */
22    Segment uninitData;          /* uninitialized data segment --
23                                  * should be zero'ed before use
24                                  */
25 } NoffHeader;
```

In directory `bin/`, you can compile programs `coff2noff` and `coff2flat` by typing `make`. Ignore the current links of `coff2noff` and `coff2flat` in that directory. After you type `make`, the binary executables of these two programs will be in the appropriate directory (`arch/unknown-i386-linux/bin/` in your case) and new links will be made to them.

The source codes of test user programs are all in directory `test/` shown as follows:

```
decius : > pwd
/home/staff/ptang/units/204/98/linux/nachos-3.4/code/test
decius : > ls
Makefile       arch/          matmul.c      shell.c        start.s
Makefile.orig  halt.c         script        sort.c
decius : >
```

To make these programs, you need to install `gcc` MIPS cross compiler first. The binary `gcc` MIPS cross compiler is available in the directory `..../USQ/66204/linux/` on the CD you received from USQ. The file name is "`linux-gcc.tar.gz`". To install it, you need to become superuser (root) of your LINUX machine and then run the installation script "root_install" by simply typing `./root_install`.

After that, you can type `make` in directory `test/`. The `Makefile` of the directory shows that it will use `coff2noff` and `coff2flat` to convert the COFF MIPS

executables to NOFF and flat formats. The results should be as follows:

```
decius : > ls
Makefile       halt.flat@     matmult.noff@  shell.noff@    start.s
Makefile.orig  halt.noff@     script         sort.c
arch/          matmult.c      shell.c        sort.flat@
halt.c         matmult.flat@  shell.flat@    sort.noff@
decius : >
```

6.6 Address Space of User Process in Nachos

The address space of a user process is defined as an object of class `AddrSpace`. The definition of this class is in `userprog/addrspace.h`. An address space has a page table pointed by its data member `pageTable`. A page table in Nachos is an array of `TranslationEntry` which is defined in `machine/translation.h` as follows:

```
30 class TranslationEntry {
31   public:
32     int virtualPage;    // The page number in virtual memory.
33     int physicalPage;   // The page number in real memory (relative to the
34                         //  start of "mainMemory"
35     bool valid;         // If this bit is set, the translation is ignored.
36                         // (In other words, the entry hasn't been initialized.)
37     bool readOnly;      // If this bit is set, the user program is not allowed
38                         // to modify the contents of the page.
39     bool use;           // This bit is set by the hardware every time the
40                         // page is referenced or modified.
41     bool dirty;         // This bit is set by the hardware every time the
42                         // page is modified.
43 };
```

Nachos uses paging for its memory management of user processes.

An address space is formed by calling the constructor of `AddrSpace`. The argument of the constructor is an open file of the binary executable in the NOFF format. At the moment, the Nachos kernel compiled in `userprog/` does not have its own file system. Instead, it uses the a stub file system built on top of the UNIX system. As can be seen from the makefiles in `userprog/`, flag `FILESYS_STUB` is defined when the kernel is compiled. The following code shows how this stub file system is built:

```
41 #ifdef FILESYS_STUB             // Temporarily implement file system calls as
42                                 // calls to UNIX, until the real file system
```

```
43                                  // implementation is available
44 class FileSystem {
45   public:
46     FileSystem(bool format) {}
47
48     bool Create(char *name, int initialSize) {
49         int fileDescriptor = OpenForWrite(name);
50
51         if (fileDescriptor == -1) return FALSE;
52         Close(fileDescriptor);
53         return TRUE;
54         }
55
56     OpenFile* Open(char *name) {
57           int fileDescriptor = OpenForReadWrite(name, FALSE);
58
59           if (fileDescriptor == -1) return NULL;
60           return new OpenFile(fileDescriptor);
61       }
62
63     bool Remove(char *name) { return (bool)(Unlink(name) == 0); }
64
65 };
66
67 #else // FILESYS
```

Functions like `OpenForWrite(..)` or `OpenForReadWrite(..)` are wrappers of UNIX file system calls. They are defined in `machine/sysdep.cc`.

Similarly, the operations on open files such as read and write are defined by using UNIX file systems calls. See `filesys/openfile.h` and `machine/sysdep.cc` for the details.

Now we are ready to see how the address space of a user process is formed. The constructor of `AddrSpace` is as follows:

```
60 AddrSpace::AddrSpace(OpenFile *executable)
61 {
62     NoffHeader noffH;
63     unsigned int i, size;
64
65     executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
66     if ((noffH.noffMagic != NOFFMAGIC) &&
67                   (WordToHost(noffH.noffMagic) == NOFFMAGIC))
68         SwapHeader(&noffH);
```

```
69      ASSERT(noffH.noffMagic == NOFFMAGIC);
70
71  // how big is address space?
72      size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
73                             + UserStackSize;        // we need to increase the size
74                                                     // to leave room for the stack
75      numPages = divRoundUp(size, PageSize);
76      size = numPages * PageSize;
77
78      ASSERT(numPages <= NumPhysPages);              // check we're not trying
79                                                     // to run anything too big --
80                                                     // at least until we have
81                                                     // virtual memory
82
83      DEBUG('a', "Initializing address space, num pages %d, size %d\n",
84                                              numPages, size);
85  // first, set up the translation
86      pageTable = new TranslationEntry[numPages];
87      for (i = 0; i < numPages; i++) {
88          pageTable[i].virtualPage = i;   // for now, virtual page # = phys page #
89          pageTable[i].physicalPage = i;
90          pageTable[i].valid = TRUE;
91          pageTable[i].use = FALSE;
92          pageTable[i].dirty = FALSE;
93          pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
94                                          // a separate page, we could set its
95                                          // pages to be read-only
96      }
97
98  // zero out the entire address space, to zero the unitialized data segment
99  // and the stack segment
100     bzero(machine->mainMemory, size);
101
102 // then, copy in the code and data segments into memory
103     if (noffH.code.size > 0) {
104         DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
105                            noffH.code.virtualAddr, noffH.code.size);
106         executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
107                            noffH.code.size, noffH.code.inFileAddr);
108     }
109     if (noffH.initData.size > 0) {
110         DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
111                            noffH.initData.virtualAddr, noffH.initData.size);
112         executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),
113                            noffH.initData.size, noffH.initData.inFileAddr);
```

```
114     }
115
116 }
```

It first reads the NOFF header from the executable file (line 65). Then it calculates the size of the address space (lines 72-73). After figuring out how many pages the address space needs, the page table is allocated (line 86). This table is then initialized (lines 87-96) such that the page number and the frame number are identical. The physical memory simulated by `machine->mainMemory` is cleared (line 100). Note that `machine` is a global variable which is initialized to point to the simulated MIPS machine when the Nachos kernel is built. The code segment (text) and the data segment (initialized data) are loaded from the executable to the physical memory allocated to address space (lines 103-114). Uninitialized data do not need to be loaded because the corresponding segment already contains zeros.

6.7 Memory Management in Nachos

You have seen how the executable of a user programs is loaded to the physical memory of the simulated MIPS machine. The loaded image forms the address space of the user process. This address space is a logical address space.

Logical addresses need to be translated to physical addresses. In real machines, this is done by the hardware of memory management unit (MMU). Memory management unit also generates various kinds of exceptions if errors occur during the translation.

Nachos uses paging with translation lookahead buffer (TLB) for its memory management. The memory management unit is is simulated by function `Translate(..)` of class `Machine`. Both functions `ReadMem(..)` and `WriteMem(..)` call this function `Translate(..)` to translate the logical address to the physical address before accessing the physical memory. The code of these functions can be found in `machine/translate.cc`. Function `Translate(..)` returns an exception if there is an error with the translation. The code of `Translate(..)` is as follows:

```
186 ExceptionType
187 Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
188 {
189     int i;
190     unsigned int vpn, offset;
191     TranslationEntry *entry;
192     unsigned int pageFrame;
193
194     DEBUG('a', "\tTranslate 0x%x, %s: ", virtAddr, writing ? "write" : "read");
```

```
195
196 // check for alignment errors
197     if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr & 0x1))){
198         DEBUG('a', "alignment problem at %d, size %d!\n", virtAddr, size);
199          return AddressErrorException;
200     }
201
202     // we must have either a TLB or a page table, but not both!
203     ASSERT(tlb == NULL || pageTable == NULL);
204     ASSERT(tlb != NULL || pageTable != NULL);
205
206 // calculate the virtual page number, and offset within the page,
207 // from the virtual address
208     vpn = (unsigned) virtAddr / PageSize;
209     offset = (unsigned) virtAddr % PageSize;
210
211     if (tlb == NULL) {          // => page table => vpn is index into table
212         if (vpn >= pageTableSize) {
213             DEBUG('a', "virtual page # %d too large for page table size %d!\n",
214                         virtAddr, pageTableSize);
215             return AddressErrorException;
216         } else if (!pageTable[vpn].valid) {
217             DEBUG('a', "virtual page # %d too large for page table size %d!\n",
218                         virtAddr, pageTableSize);
219             return PageFaultException;
220         }
221         entry = &pageTable[vpn];
222     } else {
223         for (entry = NULL, i = 0; i < TLBSize; i++)
224             if (tlb[i].valid && ((unsigned int)tlb[i].virtualPage == vpn)) {
225                 entry = &tlb[i];                        // FOUND!
226                 break;
227             }
228         if (entry == NULL) {                            // not found
229             DEBUG('a', "*** no valid TLB entry found for this virtual page!\n");
230             return PageFaultException;          // really, this is a TLB fault,
231                                                 // the page may be in memory,
232                                                 // but not in the TLB
233         }
234     }
235
236     if (entry->readOnly && writing) {   // trying to write to a read-only page
237         DEBUG('a', "%d mapped read-only at %d in TLB!\n", virtAddr, i);
238         return ReadOnlyException;
239     }
```

```
240     pageFrame = entry->physicalPage;
241
242     // if the pageFrame is too big, there is something really wrong!
243     // An invalid translation was loaded into the page table or TLB.
244     if (pageFrame >= NumPhysPages) {
245         DEBUG('a', "*** frame %d > %d!\n", pageFrame, NumPhysPages);
246         return BusErrorException;
247     }
248     entry->use = TRUE;          // set the use, dirty bits
249     if (writing)
250         entry->dirty = TRUE;
251     *physAddr = pageFrame * PageSize + offset;
252     ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
253     DEBUG('a', "phys addr = 0x%x\n", *physAddr);
254     return NoException;
255 }
```

The current MIPS simulator should be able to simulate both paging and TLB address translation. This function is supposed to be used for both page table and TLB translations. However, the current implementation assumes the we have either page tables or a TLB, but not both. To simulate true paging with TLB, this function needs to be changed. As can be seen from the definition of class `Machine`, the machine has a pointer to the page table of the current user process and a pointer to the system TLB.

In the above code, the alignment of the logical address is checked in lines 197-200. An `AddressErrorException` exception is returned if the logical address is not aligned. The page number and the offset of the logical address are calculated in line 208-209. Translation for paging is done in lines 212-221. TLB translation is done in lines 223-233. During the translation for paging, an `AddressErrorException` is returned if the logical address goes beyond the page table size. If the corresponding page is invalid, a `PageFaultException` is returned. The TLB translation generates a `PageFaultException` if no match is found in the TLB.

The physical address is assembled at line 251 if everything is successful and `NoException` is returned.

6.8 From Thread to User Process

We said earlier that Nachos user processes are built on threads. By looking at the definition of class `Thread` in `threads/thread.h` again, you will find the following lines:

```
119 #ifdef USER_PROGRAM
```

```
120 // A thread running a user program actually has *two* sets of CPU registers --
121 // one for its state while executing user code, one for its state
122 // while executing kernel code.
123
124     int userRegisters[NumTotalRegs];    // user-level CPU register state
125
126   public:
127     void SaveUserState();               // save user-level register state
128     void RestoreUserState();            // restore user-level register state
129
130     AddrSpace *space;                   // User code this thread is running.
131 #endif
```

Flag `USER_PROGRAM` is defined when you compile the Nachos in directory `userprog/`. These lines show that an array for saving user registers (line 124) and a pointer to the address space (line 130) are added to the thread to make it a user process. Of course, the actual address space of the user process has to be constructed before the user process can run. This is done by function `StartProcess(char *)` defined in `userprog/progtest.cc` as follows:

```
 23 void
 24 StartProcess(char *filename)
 25 {
 26     OpenFile *executable = fileSystem->Open(filename);
 27     AddrSpace *space;
 28
 29     if (executable == NULL) {
 30         printf("Unable to open file %s\n", filename);
 31         return;
 32     }
 33     space = new AddrSpace(executable);
 34     currentThread->space = space;
 35
 36     delete executable;                  // close file
 37
 38     space->InitRegisters();             // set the initial register values
 39     space->RestoreState();              // load page table register
 40
 41     machine->Run();                     // jump to the user progam
 42     ASSERT(FALSE);                      // machine->Run never returns;
 43                                         // the address space exits
 44                                         // by doing the syscall "exit"
 45 }
```

The argument is the file name of the executable of a Nachos user program. After the thread constructs the address space (lines 33-34) and initializes the registers (lines 38-39), it becomes a user process running on the simulated MIPS machine by calling `machine->Run()` (line 41). If both the Nachos kernel and the user program were running on a real MIPS raw machine, a kernel thread becomes the user process by jumping to the first instruction of the code section of the address space at line 41 above (of course after changing the protection mode to user mode).

6.9 Questions

Review Questions

1. Question of 9.5 of the text.
2. Question of 9.7 of the text.
3. Question of 9.8 of the text.
4. Question of 9.10 of the text.
5. Question of 9.11 of the text.
6. Question of 9.16 of the text.

Questions about Nachos

1. The current MIPS simulator in Nachos only simulates pure paging as shown in Figure 9.6 of the text. However, the MIPS simulator does support a TLB (see lines 65-68 of `machine.cc`). To make the MIPS simulator simulate paging plus TLB as shown in Figure 9.10 of the text, describe all the changes you need to make in the Nachos code.
2. The current constructor of `AddrSpace` always uses the physical memory starting from address 0 when loading a user program. The page frame of each page is always the same as the page number. As a consequence, the current Nachos can run only one user program. In order to run multiple user programs, describe all the changes you need to make in the Nachos code.
3. Currently, you can start a user process by running the Nachos kernel with command `nachos -x` *`executable`*. Here *executable* is the file name of the Nachos user program for the user process. For example, you can run user program `matmult.noff` in `test/` by typing `nachos -x ../test/matmult.noff`

in directory `userprog/`. By tracing the code, you can find that this user process is constructed from the Nachos kernel thread running its `main` function.

(a) Can you run two concurrent user processes executing the same program, say `../test/matmult.noff`, in Nachos?

(b) Describe all the changes you need to make in the Nachos source code for the task above.

Chapter 7

Implementation of System Calls

In this module, we look at the system calls of Nachos and their implementation. System calls are the interface between user programs and the operating kernel. User programs get services from the kernel by invoking system call functions. When CPU control switches from the user program to the kernel, the CPU mode is changed from user mode to system mode. When the system call function is finished by the kernel, the CPU mode is changed back to user mode and control returns to the user program. The two different CPU modes provide the base for protection in the operating system.

We examined the Nachos code for thread management and synchronization. In the previous module, we examined how a user process is formed and run on the MIPS machine simulator in Nachos. Now it is the time to examine the system call implementation of Nachos in detail.

7.1 Construction of Nachos User Programs

How are user programs in Nachos compiled? Here are a few example user programs written in C. After compilation by the `gcc` MIPS cross compiler, each of them has to be linked with the object module of a MIPS assembly program called `start.s` in directory `test/` as follows:

```
1 /* Start.s
2  *      Assembly language assist for user programs running on top of Nachos.
3  *
4  *      Since we don't want to pull in the entire C library, we define
5  *      what we need for a user program here, namely Start and the system
6  *      calls.
7  */
8
```

```
 9 #define IN_ASM
10 #include "syscall.h"
11
12         .text
13         .align  2
14
15 /* ------------------------------------------------------------
16  * __start
17  *      Initialize running a C program, by calling "main".
18  *
19  *      NOTE: This has to be first, so that it gets loaded at location 0.
20  *      The Nachos kernel always starts a program by jumping to location 0.
21  * ------------------------------------------------------------
22  */
23
24         .globl __start
25         .ent    __start
26 __start:
27         jal     main
28         move    $4,$0
29         jal     Exit     /* if we return from main, exit(0) */
30         .end __start
31
...
44
45         .globl Halt
46         .ent    Halt
47 Halt:
48         addiu $2,$0,SC_Halt
49         syscall
50         j       $31
51         .end Halt
52
...
132
133 /* dummy function to keep gcc happy */
134         .globl  __main
135         .ent    __main
136 __main:
137         j       $31
138         .end    __main
139
```

This program provides a routine `__start` to invoke the C main function of the

user program. The first instruction of `start.s` is "`jal main`". After that, zero is moved to register $4 which is the argument to the next call of routine `Exit`. `Exit` is one of the Nachos systems calls.

This program also provides the assembly stubs for the Nachos systems calls. We will talk about the implementation of system calls in Nachos shortly.

The construction of a Nachos user program is illustrated in Figure 7.1. For

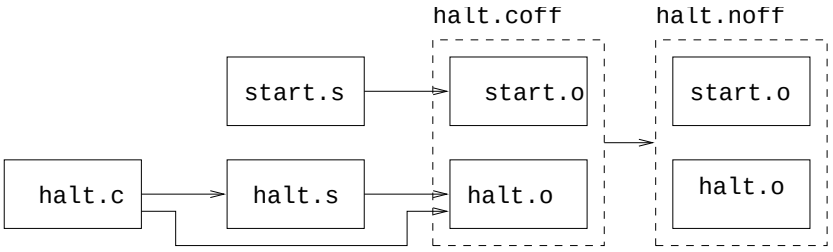

Figure 7.1: Construction of Nachos user program

example, program `halt.c` is compiled to object code `halt.o` by the MIPS cross compiler. `start.s` is translated to object code `start.o`. Then the two object codes are linked to form a COFF MIPS executable. This executable then is converted to a NOFF MIPS executable. The C program of `halt.c` is very simple as follows:

```
...
13 #include "syscall.h"
14
15 int
16 main()
17 {
18     Halt();
19     /* not reached */
20 }
```

The main program of `halt.c` simply invokes Nachos system call `Halt()`.

You can invoke the cross compiler with `-S` option and the corresponding MIPS assembly code is generated. The MIPS cross compiler which you installed in your LINUX machine is `/usr/local/nachos/bin/decstation-ultrix-gcc`. You can invoke this compiler manually in directly `test/` to generate the assembly code of `halt.c` as follows:

```
decius : > /usr/local/nachos/bin/decstation-ultrix-gcc -I../userprog -I../threads -S halt.c
decius : >
```

The compiler-generated `halt.s` is as follows:

```
 1          .file   1 "halt.c"
 2
 3  # GNU C 2.6.3 [AL 1.1, MM 40] DECstation running ultrix compiled by GNU C
 4
 5  # Cc1 defaults:
 6
 7  # Cc1 arguments (-G value = 8, Cpu = 3000, ISA = 1):
 8  # -quiet -dumpbase -o
 9
10 gcc2_compiled.:
11 __gnu_compiled_c:
12         .text
13         .align  2
14         .globl  main
15         .ent    main
16 main:
17         .frame  $fp,24,$31              # vars= 0, regs= 2/0, args= 16, extra= 0
18         .mask   0xc0000000,-4
19         .fmask  0x00000000,0
20         subu    $sp,$sp,24
21         sw      $31,20($sp)
22         sw      $fp,16($sp)
23         move    $fp,$sp
24         jal     __main
25         jal     Halt
26 $L1:
27         move    $sp,$fp                 # sp not trusted here
28         lw      $31,20($sp)
29         lw      $fp,16($sp)
30         addu    $sp,$sp,24
31         j       $31
32         .end    main
```

You can see that the first thing the main program does is to subtract stack frame pointer ($fp) to create new stack frame and save return address register ($31) and frame pointer register ($fp) (lines 20-23). The body of the main routine is simply to call the assembly routine `Halt` (line 25) which is defined in lines 47-54 of `start.s`. (Subroutine `__main` is a hook. `gcc` generates this hook (line 24) to provide an opportunity to do something before the body of the main function starts. At the moment, this hook is a dummy routine as shown in line 136-138 of `start.s`). If the main function exits normally, lines 27-31 removes the stack frame. This is the reverse of the actions done in lines 20-23. CPU control would return to the the routine which calls this main function by instruction "`j $31`" at line 31. In our

case, it would return to line 28 of `start.s` if the control reached this instruction.

7.2 System Call Interfaces

You can see that the above code (rightly) does not prepare any arguments for the system call `Halt`. How does the compiler know to do that? All the Nachos system call interfaces are defined in `userprog/syscall.h`. The compiler gets this information by including this file when compiling the user programs such as `halt.c`. (That is why you have to include the flag `-I../userprog` when you compile `halt.c` as shown above.)

Currently, Nachos supports 11 systems calls whose interfaces are defined in lines 45-125 of `userprog/syscall.h`. The assembly stubs for these system calls can be found in lines 45-131 of `test/start.s`. When a system call is issued by a user process, the corresponding stub (written in assembly language) is executed. The stub then raises an exception or trap by executing the system call instruction.

The codes of the stubs of the systems calls in `start.s` are the same:

1. set register $2 with the corresponding system call code, (see lines 21-31 of `userprog/syscall.h` for the definition of the codes of the 11 system calls.)
2. execute `syscall` instruction and
3. return to the user program.

7.3 Exception and Trap

The exceptions and traps handling of MIPS is simulated by function `RaiseException(ExceptionType , int)` of class `Machine`. Type `ExceptionType`, defined in `machine/machine.h`, includes the exceptions simulated:

```
39 enum ExceptionType { NoException,           // Everything ok!
40                      SyscallException,      // A program executed a system call.
41                      PageFaultException,    // No valid translation found
42                      ReadOnlyException,     // Write attempted to page marked
43                                             // "read-only"
44                      BusErrorException,     // Translation resulted in an
45                                             // invalid physical address
46                      AddressErrorException, // Unaligned reference or one that
47                                             // was beyond the end of the
48                                             // address space
49                      OverflowException,     // Integer overflow in add or sub.
```

```
50                      IllegalInstrException, // Unimplemented or reserved instr.
51
52                      NumExceptionTypes
53 };
```

System call exception is one of them. The MIPS "`syscall`" instruction is simulated by raising a system call exception as shown in line 534-436 of `machine/mipssim.cc`:

```
534       case OP_SYSCALL:
535         RaiseException(SyscallException, 0);
536         return;
537
```

It is very important to note that after the system call exception is handled, the next statement is `return` (line 536) instead of `break`. The program counter (PC) is not incremented and the same instruction will be re-started again.

The code of function `RaiseException(ExceptionType , int)` can be found in `machine/machine.cc`:

```
101 Machine::RaiseException(ExceptionType which, int badVAddr)
102 {
103     DEBUG('m', "Exception: %s\n", exceptionNames[which]);
104
105 //  ASSERT(interrupt->getStatus() == UserMode);
106     registers[BadVAddrReg] = badVAddr;
107     DelayedLoad(0, 0);                  // finish anything in progress
108     interrupt->setStatus(SystemMode);
109     ExceptionHandler(which);            // interrupts are enabled at this point
110     interrupt->setStatus(UserMode);
111 }
```

This function simulates the hardware actions to switch to the system mode and back to the user mode after the exception is handled. Lines 108 and 110 represent the boundary between Nachos kernel and user programs. Function call of `ExceptionHandler(which)` at line 110 simulates the hardware actions to dispatch the exception to the corresponding exception handler. This function is defined in `userprog/exception.cc`:

```
51 void
52 ExceptionHandler(ExceptionType which)
53 {
54     int type = machine->ReadRegister(2);
```

```
55
56      if ((which == SyscallException) && (type == SC_Halt)) {
57          DEBUG('a', "Shutdown, initiated by user program.\n");
58          interrupt->Halt();
59      } else {
60          printf("Unexpected user mode exception %d %d\n", which, type);
61          ASSERT(FALSE);
62      }
63  }
```

At the moment, Nachos can only handle `SyscallException` with system call code `SC_Halt`. Register $2 contains the system call code (if the exception is system call exception) and registers $4-$7 contain the first 4 arguments when the `SyscallException` handling starts. The result of the system call, if any, will be put back to $2. The exception handler for system call `Halt` is simply the `Halt()` function of the interrupt simulator pointed by `interrupt`.

7.4 Questions

Questions about Nachos

1. When linking `start.o` and the object module of a Nachos user program, say `halt.o`, why must we put `start.o` before `halt.o`?

2. Describe all the changes you need to make in the Nachos code in order to implement the remaining 10 systems calls as follows:

   (a) user process and thread control:

      i. `void Exit(int status)`
      ii. `SpaceId Exec(char *name)`
      iii. `int Join(SpaceId id)`
      iv. `void Fork(void (*func)())`
      v. `void Yield()`

   (b) file system calls:

      i. `void Create(char *name)`
      ii. `OpenFileId Open(char *name)`
      iii. `void Write(char *buffer, int size, OpenFileId id)`
      iv. `int Read(char *buffer, int size, OpenFileId id)`
      v. `void Close(OpenFileId id)`

Chapter 8

Virtual Memory

In Module 6, we discussed how multiple processes can share the memory of the system. We assumed that for a process to run, its entire address space has to be in the physical memory; otherwise, the program has to be kept in secondary storage until the physical memory it needs becomes available. This all-or-nothing memory management policy is too restrictive.

This module covers the technique called *virtual memory* that allows only part of the address space to be kept in the memory while the process is running.

The material used in this module is the Chapter 10 of the text and the Nachos source code.

8.1 Virtual memory

The basic idea of virtual memory is to allow the address space to reside partially in physical memory. This technique allows the logical address space of a process to be much larger than the physical memory. There are also other benefits of this technique. The details can be found in Section 10.1 of the text. Read the section and make sure you understand the concepts described.

8.2 Demand Paging

Virtual memory can be implemented by demand paging in systems with paging memory management. Demand paging means that a page is loaded to physical memory only when it is accessed. The details of the algorithm are described in Section 10.2 of the text. The key idea is to use *page fault exception* to trap into the kernel when the demanded page is not found in physical memory. The kernel handles this exception by loading the page from the secondary storage in a free

page frame (if no free frame is available, one of the pages in physical memory is chosen to be replaced). After the exception is handled, the same instruction which issues the memory access request is *re-started*. The algorithm is illustrated in Figure 10.4 of the text. Read the whole section and make sure you understand the concept and algorithm of demand paging.

Virtual memory has not been implemented in Nachos yet. The page fault exception is generated by function `translate(..)` of the MIPS simulator (See lines 219 and 230 of `machine/translate.cc`) in Nachos. This is function is called by functions `ReadMem(..)` and `WriteMem(..)` of the MIPS simulator. In the following discussion , we concentrate on `ReadMem(..)`. `WriteMem(..)` is similar and symmetrical. The code of `ReadMem(..)` is as follows:

```
 87 bool
 88 Machine::ReadMem(int addr, int size, int *value)
 89 {
 90     int data;
 91     ExceptionType exception;
 92     int physicalAddress;
 93
 94     DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);
 95
 96     exception = Translate(addr, &physicalAddress, size, FALSE);
 97     if (exception != NoException) {
 98         machine->RaiseException(exception, addr);
 99         return FALSE;
100     }
101     switch (size) {
102       case 1:
103         data = machine->mainMemory[physicalAddress];
104         *value = data;
105         break;
106
107       case 2:
108         data = *(unsigned short *) &machine->mainMemory[physicalAddress];
109         *value = ShortToHost(data);
110         break;
111
112       case 4:
113         data = *(unsigned int *) &machine->mainMemory[physicalAddress];
114         *value = WordToHost(data);
115         break;
116
117       default: ASSERT(FALSE);
118     }
119
```

```
120     DEBUG('a', "\tvalue read = %8.8x\n", *value);
121     return (TRUE);
122 }
```

The argument `addr` is a logical address issued by the CPU. The address translation is simulated by calling function `translate(..)` at line 96. If a page fault exception is generated, it is despatched and handled by the corresponding exception handler by calling function `RaiseException(..)` of the MIPS simulator. (We already introduced this function when we discussed system call implementation in the previous module.) If you are asked to implement virtual memory in Nachos, it is the place where you start. This corresponds to step 2 (trap) of the algorithm for demand paging described on page 294 and in Figure 10.4 of the text.

It is important to note that function `ReadMem(..)` returns `FALSE` (line 99) after the page fault exception is handled. This function is called when the MIPS simulator is fetching

- an instruction at lines 102 or
- an operand at lines 234, 252, 274, 288, 319, 484 and 514

of `machine/mips.cc`, all in function `OneInstruction(..)`. In all these cases, function `OneInstruction(..)` returns immediately without incrementing the user program counter if `ReadMem(..)` returns FALSE. When function `OneInstruction(..)` is called again in the `for` loop (lines 39-44, function `Run()`), *the same* instruction will be executed. Therefore, the MIPS simulator correctly simulates step 6 (restart instruction) of the demand paging algorithm.

8.3 Performance

Section 10.3 of the text discusses the issue of *process creation* and, in particular, the strategies for memory allocation which apply at this particular time. Read the whole section and make sure you understand the concepts and analysis of effective memory access time.

8.4 Page Replacement

Sections 10.4 of the text addresses the issue of page replacement. The question is how to select a page to be replaced when the physical memory is full and a new page needs to be brought in.

There are various kinds of page replacement algorithms discussed in detail in Section 10.4 of the text:

- FIFO Algorithm
- Optimal Algorithm
- LRU Algorithm

You also need to understand Belady's anomaly and stack algorithms.

Except for the optimal algorithm, you also need to understand how these algorithms can be implemented. Section 10.5.4 of the text discusses implementation of the algorithms. These implementations are often approximations and combination of FIFO and LRU algorithms for reasons of performance and cost. Section 10.5.5 of the text presents other replacement algorithms which can be implemented efficiently. Section 10.4.7 of the text discusses the technique of page buffering which is used to improve the performance.

You need to read the whole of Section 10.4 of the text and understand all the concepts, algorithms, and implementation techniques described.

Hardware support for page replacement in Nachos is reflected by the structure of page table entries defined in `machine/translate.h` as follows:

```
30 class TranslationEntry {
31   public:
32     int virtualPage;    // The page number in virtual memory.
33     int physicalPage;   // The page number in real memory (relative to the
34                         //  start of "mainMemory"
35     bool valid;         // If this bit is set, the translation is ignored.
36                         // (In other words, the entry hasn't been initialized.)
37     bool readOnly;      // If this bit is set, the user program is not allowed
38                         // to modify the contents of the page.
39     bool use;           // This bit is set by the hardware every time the
40                         // page is referenced or modified.
41     bool dirty;         // This bit is set by the hardware every time the
42                         // page is modified.
43 };
```

While variables `valid` and `readOnly`, corresponding to the valid and read-only bits in hardware, are used for memory management and protection, variables `use` and `dirty`, corresponding to the reference and dirty bits in hardware, are used for page replacement of the virtual memory system. Think about how you can take advantage of these two variables to implement page replacement for the virtual memory system of Nachos.

8.5 Allocation of Frames

Section 10.5 of the text addresses the issue of allocation of frames to user processes.

Section 10.5.1 discusses the minimum number of frames required by a process. The basic principle for finding this number is that a process must have enough frames to hold all the pages that any single instruction can reference either directly or indirectly. Consider the situation where we allocate only one frame to a process and an instruction which has one memory reference. Suppose the page to which this instruction belongs is in the memory and memory location referenced by this instruction is in a different page. Page fault exception will occur because that page is not in the memory. The virtual memory system will bring that page into the memory, replacing the page which includes the current instruction. When the CPU re-starts this instruction by fetching it again, another page fault will occur. Therefore, the virtual memory system will be in a infinite loop to serve page fault exceptions in the execution of this single instruction and no progress will be made.

Section 10.5.2 discusses different allocation algorithms.

Section 10.5.3 discusses global and local allocations of frames.

Read all of Section 10.5 of the text and make sure you understand all concepts and algorithms described.

Physical memory is simulated through a byte array in the MIPS simulator. The following constructor of class `Machine` shows that the size of the memory is `MemorySize` bytes or `NumPhysPages` frames. The current value of `NumPhysPages` is 32.

```
55 Machine::Machine(bool debug)
56 {
57     int i;
58
59     for (i = 0; i < NumTotalRegs; i++)
60         registers[i] = 0;
61     mainMemory = new char[MemorySize];
62     for (i = 0; i < MemorySize; i++)
63         mainMemory[i] = 0;
64 #ifdef USE_TLB
65     tlb = new TranslationEntry[TLBSize];
66     for (i = 0; i < TLBSize; i++)
67          tlb[i].valid = FALSE;
68     pageTable = NULL;
69 #else   // use linear page table
70     tlb = NULL;
71     pageTable = NULL;
72 #endif
73
```

```
74      singleStep = debug;
75      CheckEndian();
76  }
```

When implementing virtual memory for Nachos, you need to decide

- what is the minimum number of page frames for a process
- what allocation algorithm to use
- whether you apply the allocation algorithm globally or locally.

8.6 Thrashing

You must read all of Section 10.6 of the text and make sure that you understand all concepts, techniques and issues discussed.

8.7 Questions

We use some of the questions from the Chapter 10 of the text as the study and review questions:

1. Question 10.3 of the text.
2. Question 10.5 of the text.
3. Question 10.6 of the text.
4. Question 10.8 of the text.
5. Question 10.10 of the text.
6. Question 10.11 of the text.
7. Suppose that a memory reference instruction of a 32-bit machine can have at most two memory references. The instruction that has two memory references itself takes two 32-bit words. The machine allows at most 8 levels of indirection for each memory reference. What is the minimum number of frames that must be allocated to a process on this machine? Why?

Questions about Nachos

1. The MIPS simulator in Nachos provides boolean variables `use` and `dirty` in the page table entry. What page replacement algorithm can Nachos implement by taking full advantage of these variables. Describe this algorithm in detail using pseudo-code (informal C code for example).

2. We have seen that steps 1, 2 and 6 of the demand paging algorithm are correctly simulated by the MIPS simulator in Nachos. To implement a demand paging virtual memory system in Nachos, all you have to do is to implement the remaining steps: 3, 4 and 5. One of the things you have to do is management of free frames of physical memory. The concepts of free space management are described in detail in Section 12.5 of the text. Nachos provides a bitmap class called `BitMap` whose source code is in directory `userprog/`. You can use this class for free frames management.

   Describe in detail your designs to implement a demand paging virtual memory system in Nachos.

Chapter 9

File-System Interface

This module covers file system interface. The material used in this module is Chapter 11 of the text and the source code of the Nachos file system.

The concepts of a file system and its user interface are the themes of this module. *File* is an abstraction of a logical storage unit on a non-volatile storage device such as a hard disk or tape. The file system is a sub-system of the operating system to allow users to create, maintain, access, modify and delete files. A file system interface is a collection of system functions which user programs can invoke to create, maintain, access, modify and delete files.

These concepts are covered in detail in Chapter 11 of the text. You need to read the whole Chapter and understand all the concepts described.

In the following discussion, we highlight the interface and structure of the file system in Nachos to illustrate the concepts covered in this module.

9.1 Files and File Operations

A file is the abstraction of information in the form of sequence of bytes stored in a secondary storage. The operations defined on files include

1. create
2. open
3. close
4. delete

In Nachos, the operations on files can be found as functions of class `FileSystem`:

- `bool Create(char *name, int initialSize)`

- `OpenFile* Open(char *name)`
- `Remove(char *name)`

As we said earlier, Nachos has two implementations of its file system interface. One is the stub file system built on top of the underlying UNIX file system and the other the real Nachos file system of its own. However, the interface of the file system is still the same.

9.1.1 Files in the Stub File System

The stub file system in Nachos is implemented by using UNIX file system calls as shown by the following code from `filesys/filesys.h`:

```
40
41 #ifdef FILESYS_STUB             // Temporarily implement file system calls as
42                                 // calls to UNIX, until the real file system
43                                 // implementation is available
44 class FileSystem {
45   public:
46     FileSystem(bool format) {}
47
48     bool Create(char *name, int initialSize) {
49         int fileDescriptor = OpenForWrite(name);
50
51         if (fileDescriptor == -1) return FALSE;
52         Close(fileDescriptor);
53         return TRUE;
54         }
55
56     OpenFile* Open(char *name) {
57           int fileDescriptor = OpenForReadWrite(name, FALSE);
58
59           if (fileDescriptor == -1) return NULL;
60           return new OpenFile(fileDescriptor);
61       }
62
63     bool Remove(char *name) { return Unlink(name) == 0; }
64
65 };
66
67 #else // FILESYS
```

Note the flag `FILESYS_STUB` at line 41.

The three operations on files are implemented by calling functions `OpenForWrite(..)`, `OpenForReadWrite(..)`, `Close(..)`, and `Unlink(..)`, which are implemented in `machine/sysdep.cc`. The reason to use these functions instead of UNIX file system calls is to make the implementation of the stub file system system-independent.

`OpenForWrite(..)` is implemented in `sysdep.cc` as follows:

```
158 int
159 OpenForWrite(char *name)
160 {
161     int fd = open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);
162
163     ASSERT(fd >= 0);
164     return fd;
165 }
```

This function simply invokes the UNIX system call `open(name, O_RDWR|O_CREAT|O_TRUNC, 0666)` to create a UNIX file. To understand what these arguments mean and how to use this UNIX system call in general, you need to read its manual page by typing: *man 2 open* on your LINUX system. What this system call does is to open a file with the filename pointed by `name` for reading and writing. If the file does not exist, the kernel creates such a file with read and write access for the user, the group, and all others.

`OpenForReadWrite(..)` is implemented in the same file:

```
175 int
176 OpenForReadWrite(char *name, bool crashOnError)
177 {
178     int fd = open(name, O_RDWR, 0);
179
180     ASSERT(!crashOnError || fd >= 0);
181     return fd;
182 }
```

At this time, the call of function `open(..)` is used to open the file.

Function `Unlink(..)` calls the corresponding UNIX `unlink(..)` to remove the file.

9.1.2 Files in Nachos File System

The operations on files in Nachos file system are defined in `filesys/filesys.h`:

```
 68 class FileSystem {
 69   public:
```

```
...
77      bool Create(char *name, int initialSize);
78                                          // Create a file (UNIX creat)
79
80      OpenFile* Open(char *name);         // Open a file (UNIX open)
81
82      bool Remove(char *name);            // Delete a file (UNIX unlink)
83
84      void List();                        // List all the files in the file system
85
86      void Print();                       // List all the files and their contents
87
...
93  };
```

The implementation of these functions will be discussed in the next module.

9.2 Open Files

A file must be opened before it can be read and wrote. The operations on open files include:

- read
- write
- reposition with a file

Note that reading and writing a file as well as repositioning within a file are *not* operations on files. Instead, they are operations on *open files*. File and open file are two different concepts.

An open file is implemented as an object of class `OpenFile` in Nachos. The interfaces of open file for both the stub file system and Nachos file system are basically the same.

9.2.1 Open Files in Stub File System

The implementation of open file in the stub file system can be found in `filesys/openfile.h` as follows:

```
26  #ifdef FILESYS_STUB                 // Temporarily implement calls to
27                                      // Nachos file system as calls to UNIX!
28                                      // See definitions listed under #else
```

```
29 class OpenFile {
30   public:
31     OpenFile(int f) { file = f; currentOffset = 0; }    // open the file
32     ~OpenFile() { Close(file); }                        // close the file
33
34     int ReadAt(char *into, int numBytes, int position) {
35                 Lseek(file, position, 0);
36                 return ReadPartial(file, into, numBytes);
37                 }
38     int WriteAt(char *from, int numBytes, int position) {
39                 Lseek(file, position, 0);
40                 WriteFile(file, from, numBytes);
41                 return numBytes;
42                 }
43     int Read(char *into, int numBytes) {
44                 int numRead = ReadAt(into, numBytes, currentOffset);
45                 currentOffset += numRead;
46                 return numRead;
47                 }
48     int Write(char *from, int numBytes) {
49                 int numWritten = WriteAt(from, numBytes, currentOffset);
50                 currentOffset += numWritten;
51                 return numWritten;
52                 }
53
54     int Length() { Lseek(file, 0, 2); return Tell(file); }
55
56   private:
57     int file;
58     int currentOffset;
59 };
60
61 #else // FILESYS
```

Functions `WriteAt(..)` and `ReadAt(..)` are implemented by using system-independent functions `Lseek(..)`, `ReadPartial(..)` and `WriteFile(..)`. These functions are in turn implemented by invoking UNIX systems calls `lseek(..)`, `read(..)` and `write(..)`, respectively (See `machine/sysdep.cc`).

9.2.2 Open Files in Nachos File System

The interfaces of operations on open files in the Nachos file system are shown in lines 64-92 of `filesys/openfine.h`. Their implementation will be discussed in

the next module.

9.3 Directory

The concept of Directory is described in detail in Section 11.3 of the text. In short, a directory is a special file which contains a symbol table mapping file names to other information about the files. As an abstract data type, Directory can have the following operations:

- add a file to the directory
- remove a file from the directory
- rename a file in the directory
- search a file in the directory
- list all files in the directory

A directory is implemented as an object of class `Directory` in Nachos. The operations of directory are defined as follows:

```
51 class Directory {
52   public:
...
60
61     int Find(char *name);               // Find the sector number of the
62                                         // FileHeader for file: "name"
63
64     bool Add(char *name, int newSector);  // Add a file name into the directory
65
66     bool Remove(char *name);            // Remove a file from the directory
67
68     void List();                        // Print the names of all the files
69                                         //  in the directory
70     void Print();                       // Verbose print of the contents
71                                         //  of the directory -- all the file
72                                         //  names and their contents.
73
74   private:
75     int tableSize;                      // Number of directory entries
76     DirectoryEntry *table;              // Table of pairs:
77                                         // <file name, file header location>
...
81 };
```

You can also see that a Nachos directory is a table of `DirectoryEntry`. Each `DirectoryEntry` is a pair of a file name and the location in the disk of the file header (i-node) shown as follows. The file header or i-node is an object containing further information about the file.

```
32 class DirectoryEntry {
33   public:
34     bool inUse;                         // Is this directory entry in use?
35     int sector;                         // Location on disk to find the
36                                         //   FileHeader for this file
37     char name[FileNameMaxLen + 1];      // Text name for file, with +1 for
38                                         // the trailing '\0'
39 };
```

The implementation of Directory in Nachos will be discussed in the next module.

9.4 File System

The file system is the part of the operating system maintaining all files and directories. The top level concept in the file system is *partition*. The file system of an operating system may have many partitions. Each partition is a virtual secondary storage container with its own directory structure.

The Nachos file system has only one partition. Its directory structure allows only one directory (root directory) at the moment. The Nachos file system is implemented as an object of class `FileSystem`. Its data members are pointers to two open files: one is the bitmap file for the whole partition (disk) and the other is the file of the root directory. (See lines 89-91 of `filesys/filesys.h`.) These two files reside at special locations in the disk.

9.5 Questions

1. Explain the following concepts briefly:
   (a) file
   (b) open file
   (c) file pointer
   (d) file open count
   (e) open file table

(f) sequential access

(g) direct access

(h) directory

(i) directory entry

(j) partition

(k) tree-structured directories

(l) current directory

(m) relative path name

(n) absolute path name

(o) acyclic-graph directories

(p) symbolic link

(q) hard link

(r) reference count

(s) file protection

(t) access types

2. Why should files be opened before they are read and written? Give at least three reasons.

3. Among the four operations on Nachos open files, which are for sequential access and which for direct access? Use the implementation of the stub file system to justify your answer.

4. List all the UNIX file system calls used to implemented the stub file system in Nachos. For each of them, describe the syntax of the function and the meaning of each argument. You can consult the manual pages of these system calls by typing: "*man 2 xxx* on your LINUX system where *xxx* is the name of the function call.

5. Find all the shell commands related to the file system in your LINUX system. Describe the purpose and usage of each command briefly.

Chapter 10

I/O Systems and File-System Implementation

This module covers the concepts of I/O systems and the implementation of file systems. The material used in this module is Chapter 12 and part of the chapter 13 of the text, and the source code of the Nachos file system and I/O device simulation.

You need to read all of chapter 12 and part of chapter 13 and understand the issues and techniques of file system implementation.

The concepts and techniques of file system implementation will be illustrated by the implementation of the "real" Nachos file system.

You need to go back and forth between the material of the chapters 12-13 and the Nachos source code of file system implementation many times.

The topics addressed in this modules are:

- File System Structure
- Allocation Methods
- Free-Space Management
- Directory Implementation
- Efficiency and Performance
- Recovery

10.1 File System Organization

The layered structure of file systems is described in detail in Section 12.1 of the text. A typical file system includes 5 levels: logical file system, file organization

modules, basic file system and I/O control and device.

The Nachos file system implementation has seven modules. They are

- `FileSys`
- `Directory`
- `OpenFile`
- `FileHeader`
- `BitMap`
- `SynchDisk`
- `Disk`

The relationship among these modules is shown in Figure 10.1. The boxes represent these modules. The arrows in the figure represent associations between modules. For example, the arrows from module `Directory` to modules `Openfile` and `FileHeader` mean that the implementation of `Directory` uses functions from `Openfile` and `FileHeader`.

Figure 12.1 of the text shows the layered structure of a typical file system. The annotation in the left column of Figure 10.1 shows the corresponding layers in the Nachos file system.

10.2 I/O Control and Devices

As described in Section 13.2, there are two ways I/O can be completed by an I/O device controller and device driver: polling I/O and interrupt-driven I/O. The principle of interrupt-driven I/O is shown in Figure 13.3 of the text. The detail of the life cycle of an interrupt-driven I/O is shown in Figure 13.10 of the text.

As discussed by section 13.3 and illustrated in Figure 13.6 of the text, the details of the control and data transfer of the device are handled by the corresponding device controller. The device driver hides the differences among various device controllers and provides a uniform interface for I/O requests.

In the current Nachos file system, the interrupt-driven I/O of hard disk is simulated by two modules: `Disk` and `SynchDisk`. Module `Disk` is a simulator of the device controller and disk device itself, which we simply call hard disk. Module `SynchDisk` is part of the kernel I/O system (see Figure 13.10 of the text) which is responsible for queuing the blocked processes.

The source code of the hard disk simulation is in

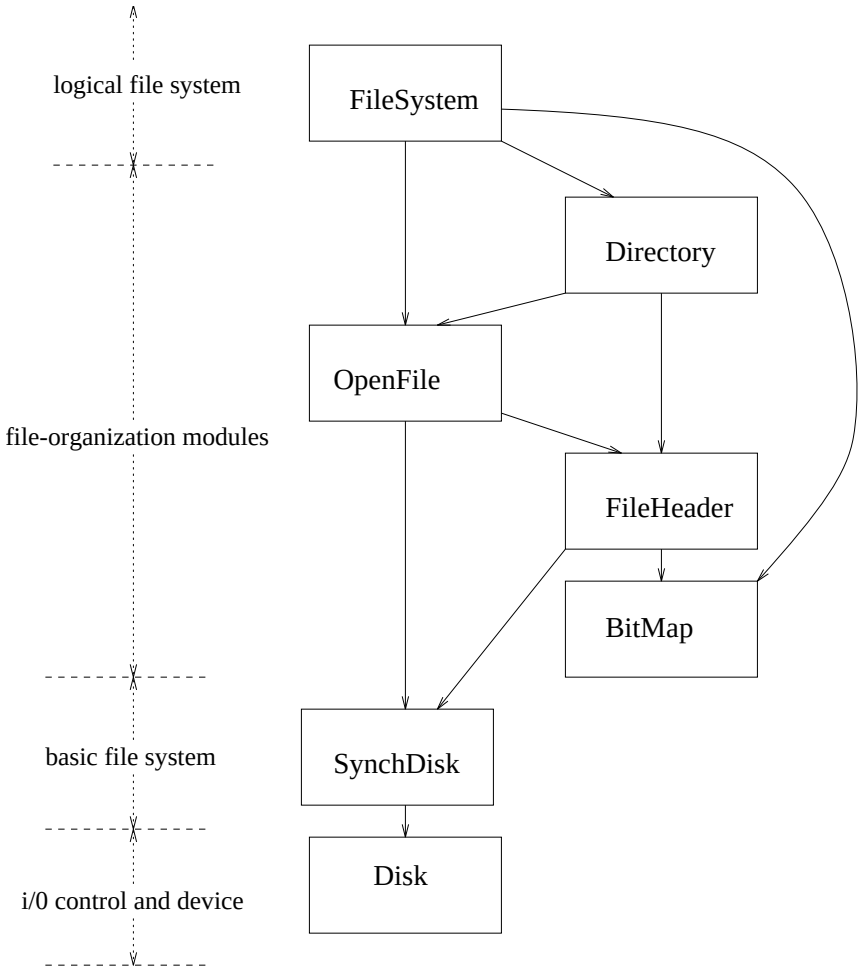

Figure 10.1: Structure of the Nachos File System

machine/disk.h
machine/disk.cc

Let us look at the structure the simulated hard disk. This hard disk has only one surface on a single platter. The structure of the disk is defined in lines 49-53 of `disk.h`:

```
49 #define SectorSize              128     // number of bytes per disk sector
50 #define SectorsPerTrack         32      // number of sectors per disk track
51 #define NumTracks               32      // number of tracks per disk
52 #define NumSectors              (SectorsPerTrack * NumTracks)
53                                         // total # of sectors per disk
```

This tells us that there are `NumTracks` tracks on the surface and `SectorsPerTrack` sectors on each track. The size of each sector is `SectorSize` bytes.

A hard disk is an object of class `Disk` defined in `machine/disk.h`:

```
55 class Disk {
56   public:
57     Disk(char* name, VoidFunctionPtr callWhenDone, int callArg);
58                                         // Create a simulated disk.
59                                         // Invoke (*callWhenDone)(callArg)
60                                         // every time a request completes.
61     ~Disk();                            // Deallocate the disk.
62
63     void ReadRequest(int sectorNumber, char* data);
64                                         // Read/write an single disk sector.
65                                         // These routines send a request to
66                                         // the disk and return immediately.
67                                          // Only one request allowed at a time!
68     void WriteRequest(int sectorNumber, char* data);
69
70     void HandleInterrupt();             // Interrupt handler, invoked when
71                                         // disk request finishes.
72
...
78   private:
79     int fileno;                         // UNIX file number for simulated disk
80     VoidFunctionPtr handler;            // Interrupt handler, to be invoked
81                                         // when any disk request finishes
82     int handlerArg;                     // Argument to interrupt handler
83     bool active;                        // Is a disk operation in progress?
84     int lastSector;                     // The previous disk request
85     int bufferInit;                     // When the track buffer started
```

```
 86                                          // being loaded
 ...
 91 };
```

When a hard disk is constructed by the constructor `Disk(char* name, VoidFunctionPtr callWhenDone, int callArg)`, the private data member `handler` is set to point to the handler `callWhenDone` and `callAgr` is assigned to another private data member `handlerArg`. The code for the constructor is in line 43-68 of `disk.cc`. Note how the function uses `OpenForReadWrite(..)` to open the named UNIX file (or `OpenForWrite(..)` to create it if it does not exist). This is the file used by the Nachos file system as a raw hard disk. The I/O functions provided by the hard disk are:

```
 63     void ReadRequest(int sectorNumber, char* data);
 ...
 68     void WriteRequest(int sectorNumber, char* data);
```

These two function simulate the commands issued by the device driver to the device controller of the hard disk (see Figure 13.10 of the text). It is important to understand how the disk I/O interrupt is simulated. Consider the implementation of `ReadRequest(..)` as shown in lines 115-133 of `disk.cc`:

```
115 void
116 Disk::ReadRequest(int sectorNumber, char* data)
117 {
118     int ticks = ComputeLatency(sectorNumber, FALSE);
119
120     ASSERT(!active);                            // only one request at a time
121     ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));
122
123     DEBUG('d', "Reading from sector %d\n", sectorNumber);
124     Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
125     Read(fileno, data, SectorSize);
126     if (DebugIsEnabled('d'))
127         PrintSector(FALSE, sectorNumber, data);
128
129     active = TRUE;
130     UpdateLast(sectorNumber);
131     stats->numDiskReads++;
132     interrupt->Schedule(DiskDone, (int) this, ticks, DiskInt);
133 }
```

After the specified data sector is read from the UNIX file (line 125), variable `active` is set to `TRUE` and the system interrupt simulator `interrupt` is called to

register a future event in `ticks` ticks time (line 132). This event represents the time when the read operation of the hard disk (not the UNIX file) is finished. When this event arrives, the interrupt hardware simulator (class `Interrupt`) will cause interrupt handler `DiskDone` to be executed with the argument provided, i.e. `(int) this`.

`DiskDone` is a static function define in line 19 of `disk.cc`:

```
28 // dummy procedure because we can't take a pointer of a member function
29 static void DiskDone(int arg) { ((Disk *)arg)->HandleInterrupt(); }
```

It simply uses the argument `arg` as a pointer to the object of `Disk` and calls its member function `HandleInterrupt()` which can be found in lines 161-166 of the same file:

```
161 void
162 Disk::HandleInterrupt ()
163 {
164     active = FALSE;
165     (*handler)(handlerArg);
166 }
```

This function sets variable `active` back to `FALSE` again signaling that the I/O operation of the disk is finished and then calls the interrupt handler in the kernel I/O subsystem.

10.3 Kernel I/O Subsystem

This layer is also called "basic file system" in Chapter 12 of the text. With regard to interrupt-driven I/O of the hard disk, we should provide the mechanism to block the processes issuing hard disk I/O here. The handler for disk I/O completion interrupts should also be provided. Another task is to synchronize concurrent disk accesses by multiple processes or threads. All these tasks are accomplished by class `SynchDisk` defined in `filesys/synchdisk.h`:

```
27 class SynchDisk {
28   public:
29     SynchDisk(char* name);              // Initialize a synchronous disk,
30                                         // by initializing the raw Disk.
31     ~SynchDisk();                       // De-allocate the synch disk data
32
33     void ReadSector(int sectorNumber, char* data);
34                                         // Read/write a disk sector, returning
35                                         // only once the data is actually read
```

```
36                                      // or written.  These call
37                                      // Disk::ReadRequest/WriteRequest and
38                                      // then wait until the request is done.
39      void WriteSector(int sectorNumber, char* data);
40
41      void RequestDone();             // Called by the disk device interrupt
42                                      // handler, to signal that the
43                                      // current disk operation is complete.
44
45    private:
46      Disk *disk;                     // Raw disk device
47      Semaphore *semaphore;           // To synchronize requesting thread
48                                      // with the interrupt handler
49      Lock *lock;                     // Only one read/write request
50                                      // can be sent to the disk at a time
51  };
```

The semaphore (data member `semaphore`) is obviously used to hold the blocked processes, while the lock (data member `lock`) is used to provide mutual exclusion of disk accesses. This class, of course, must have access to the hard disk (data member `disk`). Function `RequestDone()` is the interrupt handler for the interrupt delivered by the hard disk when the requested I/O is completed. As a matter of fact, this function is wrapped in a static function `DiskRequestDone(..)` (line 26-32) of `filesys/synchdisk.cc`), because C++ does not allow class member functions to be passed as function parameters.

The I/O operations:

```
33      void ReadSector(int sectorNumber, char* data);
...
39      void WriteSector(int sectorNumber, char* data);
```

are implemented by calling the corresponding I/O operations of the underlying raw hard disk with appropriate synchronization. The `ReadSector(..)` operation is implemented in lines 72-79 of `filesys/synchdisk.cc`:

```
72 void
73 SynchDisk::ReadSector(int sectorNumber, char* data)
74 {
75      lock->Acquire();                // only one disk I/O at a time
76      disk->ReadRequest(sectorNumber, data);
77      semaphore->P();                 // wait for interrupt
78      lock->Release();
79 }
```

The lock is used for mutual exclusion. The threads waiting for accessing the raw hard disk are held in the queue associate with the lock.

Recall that the function for the interrupt handler of the raw hard disk is `DiskRequestDone` and its argument is `(int) this` (see line 47). The function actually calls function `RequestDone()` of this synchronous disk (see line 26-32). Function `RequestDone()` then calls `V()` of the semaphore of the synchronous disk. This wakes up the thread waiting at the semaphore.

10.4 Free Space Management

The concepts and techniques of free space management for disk sectors are described in Section 12.5 of the text. The concepts and techniques described can also apply to free space management of physical memory with paging.

The raw hard disk consists of sectors. A sector is the smallest unit of storage as far as the disk I/O operation concerned. In the Nachos system, the free storage of sectors of the disk are managed by a data structure called `BitMap`. The definition of class `BitMap` can be found in `../userprog/bitmap.h` as follows:

```
34 class BitMap {
35   public:
36     BitMap(int nitems);          // Initialize a bitmap, with "nitems" bits
37                                  // initially, all bits are cleared.
38     ~BitMap();                   // De-allocate bitmap
39
40     void Mark(int which);        // Set the "nth" bit
41     void Clear(int which);       // Clear the "nth" bit
42     bool Test(int which);        // Is the "nth" bit set?
43     int Find();                  // Return the # of a clear bit, and as a side
44                                  // effect, set the bit.
45                                  // If no bits are clear, return -1.
46     int NumClear();              // Return the number of clear bits
47
48     void Print();                // Print contents of bitmap
49
50     // These aren't needed until FILESYS, when we will need to read and
51     // write the bitmap to a file
52     void FetchFrom(OpenFile *file);     // fetch contents from disk
53     void WriteBack(OpenFile *file);     // write contents to disk
54
55   private:
56     int numBits;                        // number of bits in the bitmap
57     int numWords;                       // number of words of bitmap storage
58                                         // (rounded up if numBits is not a
```

```
59                                      //  multiple of the number of bits in
60                                      //  a word)
61      unsigned int *map;              // bit storage
62  };
```

The comments in the code explain the meaning of each data member and function. The Nachos file system uses the bitmap structure to manage free sectors on the disk. If a sector is used, the corresponding bit is set. The bit of a free sector is cleared to 0. Note the side effect of function `find()`: it returns the index of the first clear bit and set it to 1 at the same time.

The bitmap for the hard disk needs to be saved on the disk as a file because memory is volatile. It is a special file managed by the kernel. Functions `FetchFrom(..)` and `WriteBack(..)` are used for this purpose.

The implementation of class of `BitMap` is in file `userprog/bitmap.cc`. You need to read it and make sure that you understand how the bitmap works.

10.5 File Header (I-Node)

File header, also known as i-node in UNIX operating systems, is an important data structure in file systems. Each file has a file header which stores the size of the file as well as the locations of the data sectors of the file. The structure of file headers depends on the method of allocation discussed section 12.4 of the text.

In the Nachos file system, a simple flat index allocation is used. The class `FileHeader` is defined in `filesys/filehdr.h`. The private data members of the class are:

```
60      int numBytes;                   // Number of bytes in the file
61      int numSectors;                 // Number of data sectors in the file
62      int dataSectors[NumDirect];     // Disk sector numbers for each data
63                                      // block in the file
```

where `dataSectors[]` is the index table which gives the disk sector number for each data block.

Two important constants are defined in lines 20-21:

```
20 #define NumDirect       ((SectorSize - 2 * sizeof(int)) / sizeof(int))
21 #define MaxFileSize     (NumDirect * SectorSize)
```

`NumDirect` is the size of the index table `dataSectors[]`. It is the maximum number of the data blocks a Nachos file can have. The reason for the definition

of NumDirect above is that the file header has the same size as a sector on the disk. Recall that a file header has to accommodate two variables, numBytes and numSectors, in addition to the index table.

The functions of class FileHeader are:

```
40     bool Allocate(BitMap *bitMap, int fileSize);// Initialize a file header,
41                                                 //  including allocating space
42                                                 //  on disk for the file data
43     void Deallocate(BitMap *bitMap);            // De-allocate this file's
44                                                 //  data blocks
45
46     void FetchFrom(int sectorNumber);   // Initialize file header from disk
47     void WriteBack(int sectorNumber);   // Write modifications to file header
48                                         //  back to disk
49
50     int ByteToSector(int offset);       // Convert a byte offset into the file
51                                         // to the disk sector containing
52                                         // the byte
53
54     int FileLength();                   // Return the length of the file
55                                         // in bytes
56
57     void Print();                       // Print the contents of the file.
```

The implementation of these functions can be found in filesys/filehdr.cc. For example, function Allocate(..) is as follows:

```
41 bool
42 FileHeader::Allocate(BitMap *freeMap, int fileSize)
43 {
44     numBytes = fileSize;
45     numSectors  = divRoundUp(fileSize, SectorSize);
46     if (freeMap->NumClear() < numSectors)
47         return FALSE;           // not enough space
48
49     for (int i = 0; i < numSectors; i++)
50         dataSectors[i] = freeMap->Find();
51     return TRUE;
52 }
```

This function is called when a new file is created. In the Nachos system, a file has a fixed size when it is created. After the number of sectors needed is found (stored in variable numSectors), the data allocation for the sectors is done by consulting the free map (freeMap) of the hard disk (see lines 49-50).

The implementation of other functions are straightforward. Function `Print()` is used to dump both the i-node and the data sectors of the file. The purpose of functions `FetchFrom(..)` and `WriteBack(..)` is to retrieve and store the i-node itself.

10.6 Open Files

The concept of open files is discussed in sections of 11.1.2 and 12.1 of the text, as part of the file system structure. In UNIX systems, each process has a open file table. The data structure of open file should provide:

- the current seek position of the open file
- a reference to the i-node of the open file
- functions to access the file such as read and write

In the Nachos file system, this data structure is provided through class `OpenFile` defined in `filesys/openfile.h`. As with class `FileSystem` discussed in Module 9, there are two implementations of open files: one for the UNIX files (when `FILESYS_STUB` is defined) and the other for the "real" Nachos file system. This is because open files, strictly speaking, are part of the user interface of file systems.

As shown in lines 90-91 of `openfile.h`, the integer `seekPosition` and the pointer to the file header (i-node) of the file are the data members of an open file. The implementation of the functions of the class can be found in `filesys/openfile.cc`.

The constructor of the class shown below loads the i-node of the file from the disk and set the seek position to 0.

```
27 OpenFile::OpenFile(int sector)
28 {
29     hdr = new FileHeader;
30     hdr->FetchFrom(sector);
31     seekPosition = 0;
32 }
```

The sector number for the i-node of the file is provided by the higher level module, `Directory`, which we will examine in the next section. Functions `Seek(..)` and `Length()` are straightforward. Functions `Read(..)` and `Write(..)` are implemented by using functions `ReadAt(..)` and `WriteAt(..)`, respectively.

There are a lot of details in the implementation of functions `ReadAt(..)` and `WriteAt(..)`. You must read them and make sure that you understand the mechanism and techniques used.

10.7 Directories

The concept and structure of a directory are described in Section 10.3 of the text. Section 11.4 of the text addresses the issues of its implementation.

The directory structure of the Nachos file system is very simple: there is one directory and all files belong to this directory.

The data structure of Nachos directory is defined as class `Directory` in `../filesys/directory.h`:

```
51 class Directory {
52   public:
53     Directory(int size);                // Initialize an empty directory
54                                         // with space for "size" files
55     ~Directory();                       // De-allocate the directory
56
57     void FetchFrom(OpenFile *file);     // Init directory contents from disk
58     void WriteBack(OpenFile *file);     // Write modifications to
59                                         // directory contents back to disk
60
61     int Find(char *name);               // Find the sector number of the
62                                         // FileHeader for file: "name"
63
64     bool Add(char *name, int newSector);  // Add a file name into the directory
65
66     bool Remove(char *name);            // Remove a file from the directory
67
68     void List();                        // Print the names of all the files
69                                         //  in the directory
70     void Print();                       // Verbose print of the contents
71                                         //  of the directory -- all the file
72                                         //  names and their contents.
73
74   private:
75     int tableSize;                      // Number of directory entries
76     DirectoryEntry *table;              // Table of pairs:
77                                         // <file name, file header location>
78
79     int FindIndex(char *name);          // Find the index into the directory
80                                         //  table corresponding to "name"
81 };
```

Private data member `table` is a pointer to a table of file-name and i-node sector number pairs called `DirectoryEntry`. `DirectoryEntry` is defined in lines 32-39:

```
32 class DirectoryEntry {
```

```
 33   public:
 34     bool inUse;                          // Is this directory entry in use?
 35     int sector;                          // Location on disk to find the
 36                                          //   FileHeader for this file
 37     char name[FileNameMaxLen + 1];       // Text name for file, with +1 for
 38                                          // the trailing '\0'
 39 };
```

The boolean variable `inUse` is used to indicate if this entry is in use.

A directory itself is a file. It needs to be stored and fetched from the disk. Functions `WriteBack(..)` and `FetchFrom(..)` are used for this purpose. Functions `Add(..)` and `Remove(..)` are used to add and remove a directory entry. The implementation of all these functions can be found in `../filesys/directory.cc`. The implementation of `Add(..)` is as follows:

```
129 bool
130 Directory::Add(char *name, int newSector)
131 {
132     if (FindIndex(name) != -1)
133           return FALSE;
134
135     for (int i = 0; i < tableSize; i++)
136         if (!table[i].inUse) {
137             table[i].inUse = TRUE;
138             strncpy(table[i].name, name, FileNameMaxLen);
139             table[i].sector = newSector;
140         return TRUE;
141         }
142     return FALSE;       // no space.  Fix when we have extensible files.
143 }
```

The size of a directory in the Nachos file system is fixed, because the file size of the directory is fixed. What this functions does is simply to find the first unused directory entry and put the name and sector number there.

Function `List()` is just like "ls" system call in the UNIX. It lists the names of all the files in the directory.

The implementations of the functions in class `Directory` are quite straightforward. You need to read them all and make sure that you understand them.

10.8 File System

We already discussed the user interface of the Nachos file system in Module 9. We also show the implementation of the stub file system.

As shown in lines 89-91 of `filesys/filesys.h`, there are two open file pointers associated with the file system: one for the free bit map file and the other for the directory file. These two files are always open while the file system exists.

You need to pay special attention to the constructor in lines 80-143 of `../filesys/filesys.cc` as follows:

```
 80 FileSystem::FileSystem(bool format)
 81 {
 82     DEBUG('f', "Initializing the file system.\n");
 83     if (format) {
 84         BitMap *freeMap = new BitMap(NumSectors);
 85         Directory *directory = new Directory(NumDirEntries);
 86         FileHeader *mapHdr = new FileHeader;
 87         FileHeader *dirHdr = new FileHeader;
 88
 89         DEBUG('f', "Formatting the file system.\n");
 90
 91     // First, allocate space for FileHeaders for the directory and bitmap
 92     // (make sure no one else grabs these!)
 93         freeMap->Mark(FreeMapSector);
 94         freeMap->Mark(DirectorySector);
 95
 96     // Second, allocate space for the data blocks containing the contents
 97     // of the directory and bitmap files.  There better be enough space!
 98
 99         ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
100         ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
101
102     // Flush the bitmap and directory FileHeaders back to disk
103     // We need to do this before we can "Open" the file, since open
104     // reads the file header off of disk (and currently the disk has garbage
105     // on it!).
106
107         DEBUG('f', "Writing headers back to disk.\n");
108         mapHdr->WriteBack(FreeMapSector);
109         dirHdr->WriteBack(DirectorySector);
110
111     // OK to open the bitmap and directory files now
112     // The file system operations assume these two files are left open
113     // while Nachos is running.
114
115         freeMapFile = new OpenFile(FreeMapSector);
116         directoryFile = new OpenFile(DirectorySector);
117
118     // Once we have the files "open", we can write the initial version
```

```
119     // of each file back to disk.  The directory at this point is completely
120     // empty; but the bitmap has been changed to reflect the fact that
121     // sectors on the disk have been allocated for the file headers and
122     // to hold the file data for the directory and bitmap.
123
124         DEBUG('f', "Writing bitmap and directory back to disk.\n");
125         freeMap->WriteBack(freeMapFile);         // flush changes to disk
126         directory->WriteBack(directoryFile);
127
128         if (DebugIsEnabled('f')) {
129             freeMap->Print();
130             directory->Print();
131
132         delete freeMap;
133          delete directory;
134         delete mapHdr;
135         delete dirHdr;
136         }
137     } else {
138     // if we are not formatting the disk, just open the files representing
139     // the bitmap and directory; these are left open while Nachos is running
140         freeMapFile = new OpenFile(FreeMapSector);
141         directoryFile = new OpenFile(DirectorySector);
142     }
143 }
```

The purpose of this constructor is to build up the Nachos file system.

Lines 63-65 of `filesys.cc` define the size of these two special files:

```
63 #define FreeMapFileSize         (NumSectors / BitsInByte)
64 #define NumDirEntries           10
65 #define DirectoryFileSize       (sizeof(DirectoryEntry) * NumDirEntries)
```

`NumDirEntries` defines the size of the directory entry table of a directory. Therefore, this is the maximum number of files that a directory can contain. `NumSectors` and `BitsInByte` are defined in `../machine/disk.h` and `../userprog/bitmap.h`, respectively. Both of them are included in this file.

The constructor first creates a bit map and a directory (not files yet) in lines 84-85. Lines 86-87 creates i-nodes for the bit map file and the directory file.

`FreeMapSector` and `DirectorySector` are defined as 0 and 1, respectively. Lines 93-94 mark sector 0 and sector 1 of the hard disk.

In lines 99-100, the disk space for the bit map file and the directory file is allocated. At this point, the i-nodes of the bit map file and the directory file are

valid. In lines 108-109, their i-nodes (in memory) are written back to sectors 0 and 1.

Then, these files are opened by lines 115-116. Now it is time to write the *contents* of the bit map file and the directory file to the data sectors allocated to them. This is done in line 115 and 116.

At this point, all information of these two files are saved in the disk. The i-nodes and the contents of each of them are consistent.

The data structures pointed by `freeMap`, `directory`, `mapHdr` and `dirHdr` are not used any more and, thus, deleted (lines 132-135) before the function finishes.

The implementations of other functions are straightforward. You must read them all and make sure that you understand them.

10.9 Questions

10.9.1 Review Questions

1. Question 12.1 of the text.
2. Question 12.6 of the text.
3. Question 12.7 of the text.
4. Question 12.8 of the text.

10.9.2 Questions about Nachos

1. Describe three tasks of class `SynchDisk` and how each of the tasks is completed.
2. The current Nachos file system allows only one root directory. To extend this file system to allow the tree-structure directories discussed in section 11.3 of the text, what changes do you need to make in the Nachos code?
3. To further extend the Nachos file system, we need to allow hard links in the directory tree as discussed in Section 11.3.4. What change do you need to make in the Nachos code?
4. The current Nachos file system uses simple index allocation of data sectors of a file. If you are asked to use linked allocation discussed in Section 12.4.2 of the text, what changes do you need to make in the Nachos code?

5. It the current Nachos file system, the size of a file is fixed and is determined when the file is created. To extend the Nachos file system so that the size of a file can be increased as more data are written to the file, what changes do you need to make in the Nachos code?

6. The current Nachos file system imposes a limit (32 sectors) on the size of a file. To implement the FAT allocation in Nachos so that its file can have any size, what changes do you nee to make in the Nachos code?