操作系统课程设计

张鸿烈 编著

山东大学计算机科学与技术学院

目录

Nachos Operating System Study Book

Preface

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 by this department. 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.

This study book is divided into three parts with 9 modules altogether. They are:

Part One: Process Management

Process and Thread

CPU Scheduling

Process Synchronization

Part Two: Memory Management

MIPS Simulator

Memory Management in Nachos

Implementation of System Calls

Virtual Memory

Part Three: File-System

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.

Part one

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 1 Process and Thread

1.1 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.

1.2 Process

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

Process is a 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

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.

1.3 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 1.1 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.

1.4 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.

1.5 Thread and Process Scheduling

A thread or process will go through many states during its life-time. Figure 1.1 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 };
```

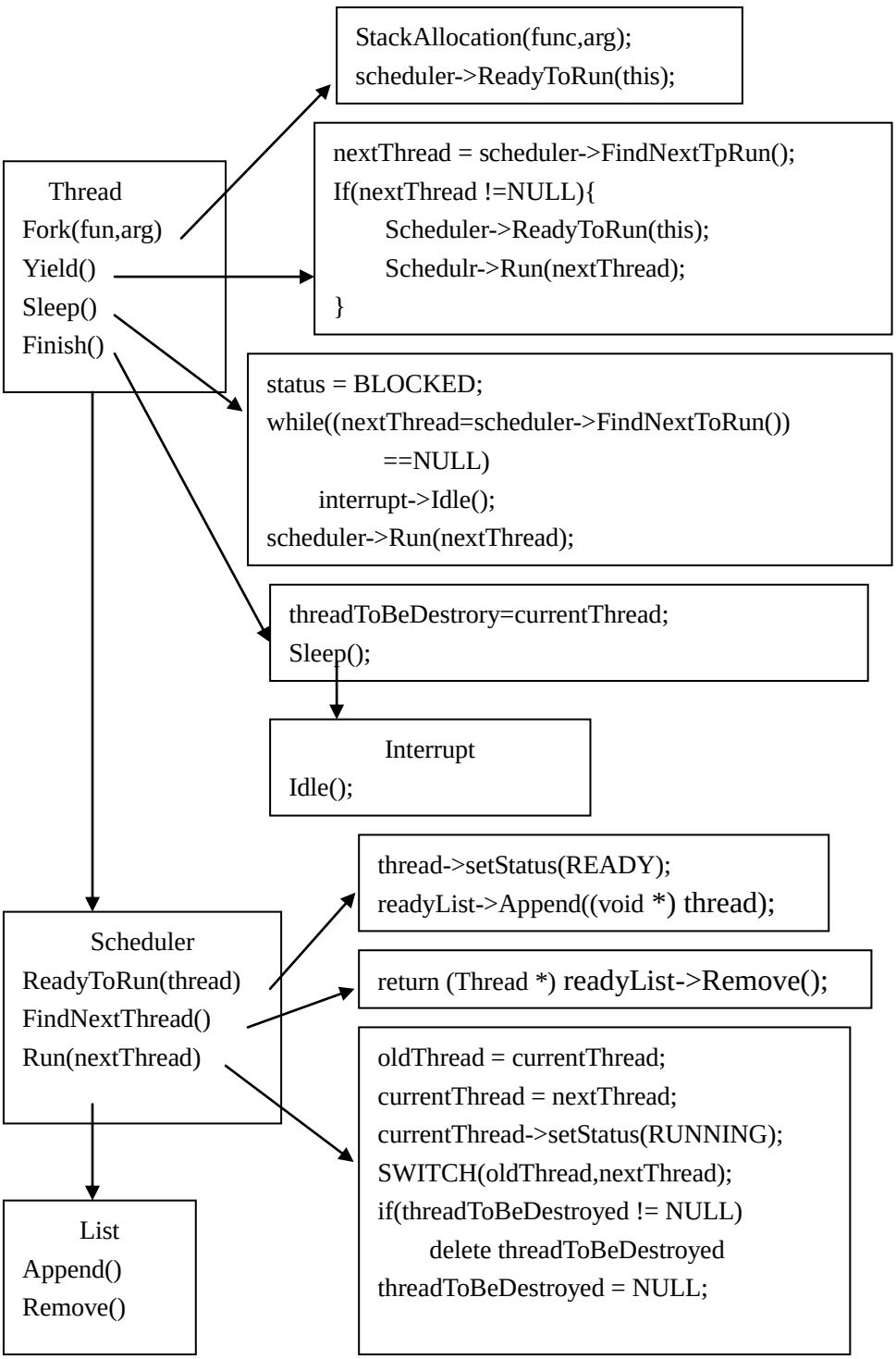


Figure 1.1 Class diagram of Nachos thread scheduling

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).

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 1.1. 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 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.

1.6 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.

1.7 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 registers 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 inregister 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.

1.8 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
22

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, · · ·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, have fixed values 0, 1, 2, … There S1 … fore, 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.

1.9 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.

1.10 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  }
28
```

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.

Chapter 2 Process Synchronization

2.1 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:

Semaphore

Critical Regions

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.

2.2 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.

2.3 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.

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

2.5 Semaphore

Section 7.4 of text introduces probably the most important and the most widelyused 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.

2.5.1 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) fblock the current process after putting it in the queue L;
}
V()
v = v + 1;
if (v ≤ 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.

2.5.2 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.

2.5.3 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 nVc and nPs 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:

$$v = I + nVc \,.\, nPs \qquad (2.1)$$

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

$$nPs = nPb + nPc \qquad (2.2)$$

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 nPb:

$$nPb = \begin{cases} -v & \text{if } nPs - (I + nVc) > 0 \quad (2.3) \\ 0 & \text{otherwise} \end{cases}$$

To get the abstract invariant, we need to eliminate v from the above formulas. By substituting (2.2) in (2.1), we will have:

$$I + nVc \,.\, nPc = v + nPb$$

According to (2.3), v + nPb $\geqslant$ 0 holds. Hence, we reach the invariant about the relationship between nVc and nPc :

$$I + nVc \geqslant nPc \qquad (2.4)$$

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

$$nPb = \begin{cases} nPs - (I + nVc) & \text{if } nPs - (I + nVc) > 0 \quad (2.5) \\ 0 & \text{otherwise} \end{cases}$$

As a result, the implementation-independent invariants of semaphore are expressed by (2.4), (2.2) and (2.5).

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:

$$v = I + nVc \,.\, nPc \qquad (2.6)$$

The key aspect of Algorithm 2 is to keep v $\geqslant$ 0, because it will translate (2.6) to the abstract invariant (2.4). To maintain v $\geqslant$ 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:

Process A issues a P() and becomes blocked at the semaphore.

Process B issues a V() and wakes up and put process A into the system ready queue.

Process B increments v from 0 to 1.

Process C issues a P(), decrements v from 1 to 0 and exits the P(),

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 $\geqslant$ 0 is always true and so is the invariant 2.4.

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 (2.2) and (2.5) still hold for Algorithm 2.

We have shown that Algorithms 1 and 2 both maintain the abstract invariants in (2.4), (2.2) and (2.5) and therefore, both are correct.

2.5.4 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.

2.6 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.

2.7 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.

2.8 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.

2.8.1 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.

2.8.2 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.

2.8.3 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.

2.8.4 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 acquired 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.

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 3 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.

3.1 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:

A Compiler translates the source code of each module to its object module,

A linker links all the object modules to form a single binary executable also known as load module,

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.

3.2 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:

Contiguous Allocation
Paging
Segmentation
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.

3.3 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.

3.4 MIPS Simulator

3.4.1 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);

3.4.2 Instruction Interpretation

Function Run() in line 30-45 of machine/mips.cc sets the machine into usermode 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.

3.4.3 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.

3.5 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 UNIXCOFF 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/ matmult.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. 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 : >

3.6 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
     #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);
58

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.

3.7 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.

3.8 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).

Chapter 4 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.

4.1 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.

For 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 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.

4.2 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:

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.)

execute syscall instruction and

return to the user program.

4.3 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.

Chapter 5 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.

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 Thrashing

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

Part Three
File-System

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.

Chapter 6 File-System Interface

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

6.1 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

create
open
close
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.

6.1.1 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.

6.1.2 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.

6.2 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.

6.2.1 Open Files in Stub File System

The implementation of open file in the stub file system can be found in filesys/open file.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).

6.2.2 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.

6.3 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 };
85
```

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.

6.4 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.

Chapter 7 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

7.1 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 7.1. 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 7.1 shows the corresponding layers in the Nachos file system.

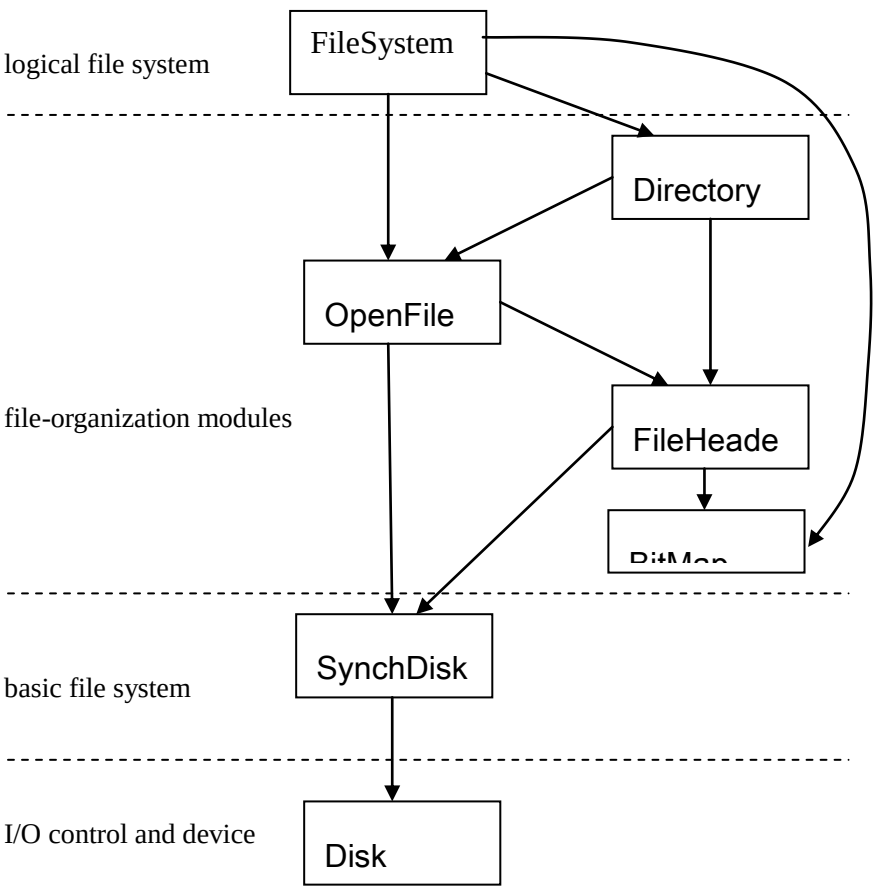


Figure 7.1: Structure of Nachos File

7.2 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

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);
...
    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.

7.3 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.

7.4 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.

7.5 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.

7.6 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 Module9, 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.

7.7 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.

7.8 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       (sizeof(DirectoryEntry) * NumDirEntries)
65 #define DirectoryFileSize   10
```

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 inodes 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.

Nachos Operating System

Introductory Book

Overview of the Course

1.1 Prerequisites

The prerequisites of this courses are

Computer Engineering and

Algorithms and Data Structures which has Advanced Procedural Programming and Discrete Mathematics for Computing as its prerequisites.

It is assumed that students taking this course already have the following knowledge and skills:

digital logic, binary number systems, computer organization, CPU, registers, ALU, bus, memory, machine instruction sets, addressing modes, I/O methods, interruption, and machine and assembly programming,

the programming language C,

advanced procedural programming skills in C, and

abstract data types and common data structures in computer science.

Those who do not have these knowledge and skills will find this course extremely difficult and are

strongly advised not to attempt it until ready.

We will use part of C++ as the languages for the programming assignment and lab sessions in this course. While we do not assume your knowledge and skills in C++, your knowledge and skills in C and abstract data types should be solid enough to enable you to pick up the C++ syntax and its classes as abstract data types in a short period of time, say two weeks. If you are not able to do so, you will find this course very difficult and are strongly advised not to attempt it until ready.

1.2 Assumption of UNIX skills

In order to be able to complete the laboratory work and programming assignments in this course, you need to have a minimum of skills and knowledge about UNIX. LINUX is a UNIX operating system. You need to know:

(a) basic UNIX commands and file systems

(b) make utility and Makefiles

(c) gdb debugger tool

The book "Running LINUX" by Matt Welsh (published by O'Reilly Associates Inc.) is an excellent introduction for everything you need to know about UNIX. I strongly recommend that every external student have an own copy of it.

We assume that you have these UNIX skills and knowledge when we describe the laboratory sessions in this book. Those who do not have these skills will find this course extremely difficult and are strongly advised not to attempt it until ready.

1.3 What This Course Is About

An operating system is the fundamental software between computer hardware and all other software in a computing system. It serves as a hardware resource manager which makes use of the hardware much easier and more efficient. It also provides an interface to support any other software including compilers, database systems, and application programs.

This course covers the design and implementation of three principle components of operating systems:

Process Management

Memory Management

File Systems Management

This course is not about how to use operating systems as in other introductory courses of computing. It concentrates on the concepts and techniques in design and implementation of operating systems.

In other words, this course is not a course about operating systems, but rather a course on the operating system software itself. The knowledge and skills gained from this course are essential for computer science professionals.

On completion of this course, you will

be able to demonstrate an understanding of the concepts and internal structures of operating systems.

understand the implementation of operating systems at source code level

have hands-on experience of design and implementation of a real operating system.

1.4 Teaching Material

The teaching material of this course consists of two parts: the textbook and the source code of a real operating system called NACHOS.

1.4.1 Textbook

The textbook used in this course is Silberschatz, A., Galvin, P., and Gagne, G., "Operating System Concepts", 6th Edition Addison-Wesley, Reading, Massachusetts, 2002,ISBN 0471417432.

The following chapters of the textbook are used:

(a) Part I: chapters 1,2,3

(b) Part II: chapters 4,5, 7

(c) Part III: chapter 9,10,11,12, 13.

1.4.2 Nachos Source Code

Nachos is a working operating system written by Prof. Tom Anderson from the University of California at Berkeley for teaching operating systems. In this course we use Nachos as the system for case study, laboratory exercises and programming assignments. While the textbook above gives

a thorough description of concepts of operating systems, the Nachos source code provides an illustration of implementation of these concepts. Reading the Nachos source code is essential to understand the concepts in the textbook. The Nachos source code is provided in the Selected Reading of this course.

The most effective way to learn the design and implementation of operating systems is to examine a real operating system at the source code level. The Nachos operating system has about 9,500 lines of C++ code with extensive comments. It is well designed and small enough to be used for teaching. It has been used by many universities in the world for teaching operating systems.

1.5 Course Structure and Modules

The course consists of three parts with 10 Modules and 8 laboratory sessions altogether.

In Part I, we provide the introductory material on operating systems. This part serves as an extended introduction to the subject. The following topics are covered in this part:

(a) Introduction (Module 1)

(b) Computer System Structures (Module 2)

(c) Operating System Structures (Module 3)

Part II and Part III are the core of the course, in which we examine the design and implementation of three principle operating systems components.

The topics covered in Part II are:

(a) Processes and Threads (Module 4)

(b) Process Synchronization (Module 5)

The topics covered in Part III are:

(a) Memory Management (Module 6)

(b) Implementation of System Calls (Module 7)

(c) Virtual Memory (Module 8)
(d) File System Interface (Module 9)
(e) File System Implementation (Module 10)

When studying Modules 4--10, you are required to read the relevant parts of the Nachos source code. An extensive guide for reading the Nachos source code is provided in the Study book of this course. It is important to read both the textbook and the Nachos source code when you study these Modules.

There are 8 laboratory sessions on Nachos. You need to complete these laboratory sessions on your LINUX/PC system. The details of the Nachos system and LINUX/PC home laboratory are provided in the following section.

1.5 The NACHOS System and Laboratory Sessions

An ancient Chinese proverb says: "I hear and I forgot, I see and I remember, I do and I understand". An operating system is a complicated artifact created by humans. The only way to study and learn effectively the concepts and techniques of design and implementation of operating systems is to examine and experiment with a real operating system at the source code level. However, most operating systems are large and not suitable for teaching purposes. Prof. Andrew Tanenbaum from Vrije University, the Netherlands, wrote a small UNIXlike operating system called MINIX for teaching. More recently, Prof. Tom Anderson from the University of California, Berkeley wrote the NACHOS operating system for the same purpose. This course uses the NACHOS operating system.

The NACHOS operating system is used in this course for the following purposes:

case study for operating systems design and implementation
laboratory exercises
programming tasks in assignments

We provide the source code of the Nachos system in two forms:

the source code listing in the Selected Reading of this course
the distribution of Nachos

You need to use the Nachos operating system to finish

8 laboratory sessions and
2 programming tasks.

on your LINUX/PC system.

The laboratory work and programming tasks are essential to this course. Eight laboratory sessions are designed to enable you to complete the programming tasks.

1.6 LINUX/PC Home Laboratory

You need to install Linux operating system on your home PC before experimenting with

the Nachos system.

1.7 Assignments and Examination

There are three assignments and one final examination in this course.

The modules covered by each assignment are as follows:

Assignment 1: Modules 1, 2, 3 and 4

Assignment 2: Module 5

Assignment 3: Modules 9 and 10

A 2hour closedbook examination at the end of semester covers all the Modules of the course.

The due dates of the assignments and examination are as follows:

Number	Marks	Due week	Description	Weight
1	15	5	Assignment 1	20
2	15	10	Assignment 2	20
3	15	15	Assignment 3	20
4	100	end of Semester	Final Exam (2 Hours)	40

1.8 Study Chart

Week	Modules	Laboratory	Assignment	Assessment
1	1,2,3	Lab1	Ass1	
2	4	Lab2		
3				
4	5	Lab3		Assignment 1 Due week 5
5			Ass2	
6	9	Lab4		
7	10	Lab5		
8				
9	6	Lab6		Assignment 2 Due week 10
10	break		Ass3	
11				
12	6	Lab7		
13	7			
14	8	Lab8		
15	8			Assignment 3 Due week 15
16	Review			

How to Submit Programming Assignments

2.1 How to Use Floppy Disk in LINUX

In this course, you need to use floppy disks to submit the source of your Nachos system for the assignments. In this section, we are going to describe the commands you need to mount and unmount floppy disk drive as well as to format a new floppy disk. If you are going to use a new floppy disk or a floppy of other file systems such as MSDOS, you need to format it before you can use it in LINUX. The command to format a floppy disk is:

```
# /sbin/mke2fs /dev/fd0
```

You need to become the superuser root to do that. Note that after you format an old disk, all the files on it will be wiped out. Make sure that you do not need the contents of the disk before you reformat it.

After you insert the floppy disk (and format it if it is a new floppy disk), you need to mount the floppy disk drive to your LINUX file system. You need to become the superuser root to mount and unmount the floppy drive.

mount: To mount the floppy disk drive, execute the following command as the superuser:

```
# mount /dev/fd0 /mnt/floppy (or mount /mnt/floppy)
```

Here, /mnt/floppy is the mount point for the floppy disk. After mounting, you can access the files on the floppy disk in directory /mnt/floppy/. Since directory /mnt/floppy/ is owned by the superuser root, only the superuser can write files in it. Ordinary users can only read readable files from this directory.

unmount: To unmount the floppy drive, execute the following commands (as the superuser root again):

```
# sync
# umount /mnt/floppy
```

Before unmounting the floppy, you have to make sure that no users (including root) are using /mnt/floppy/ as their current working directory; otherwise the system would show the error that the directory is busy and does not allow you to unmount it. After unmounting the floppy, you can remove the floppy disk from the drive.

2.2 How to Submit Your Nachos

You need to submit your Nachos system if you are required to complete a programming task in an assignment. You will submit your Nachos code through a floppy disk in the LINUX file format.

Follow the steps below to make a floppy disk of your Nachos code:

(a) move to the directory where your nachos3.4 directory is by using the cd command;

(b) make a tar file of your Nachos system, nachos.tar.gz, by typing the following command:

```
tar cvzf nachos.tar.gz    nachos-3.4
```

(c) insert a new floppy disk into the floppy drive;

(d) become the superuser root;

(e) format the new floppy by typing:

```
/sbin/mke2fs /dev/fd0
```

(f) mount the floppy to the file system by typing:

```
mount /dev/fd0 /mnt/floppy
```

(g) copy file nachos.tar.gz to the floppy disk as the superuser:

```
cp nachos.tar.gz    /mnt/floppy
```

(h) unmount the floppy by typing the following commands:

```
sync
umount /mnt/floppy
```

remove the floppy disk and include it with your paper work for the assignment.

Laboratory 1: Installation of Nachos System

3.1 Purpose

We assume that you already have installed the Red Hat LINUX 7.2. The purpose of this laboratory session is to enable you to:

install and compile the Nachos system,
understand the structure of Nachos, and
get familiar with C++ programming language.

3.2 Installation of Nachos and gcc CrossCompiler

The simplest way to install Nachos and gcc MIPS crosscompiler from the Computing CD is to use the command tar directly.

After you install LINUX from one of the CDs, mount another CD which contains courses software. Move to the directory for this course. In the following, we assume the path of this directory is /mnt/cdrom/ or similar. You will see the following files in the directory (or something similar):

nachos-3.4.tgz is the file of the Nachos system package.
Gcc-2.8.1-mips.tar.gz is the file of gcc MIPS crosscompiler.

Installation of Nachos

Follow the following steps to install Nachos:

(a) cd
(b) cp /mnt/cdrom/nachos-3.4.tgz .
(c) tar xzvf nachos-3.4-.tgz
(d) rm nachos-3.4.tgz

After these steps, you should have a directory named nachos-3.4 in your home directory should contain a directory named nachos-3.4 which is the Nachos package.

Installation of gcc MIPS crosscompiler

The gcc MIPS crosscompiler is to be installed in /usr/local directory. You need to become root (superuser) of your LINUX in order to install it.

Follow the following steps to install gcc crosscompiler:

(a) su
(b) cd /usr/local
(c) cp /mnt/cdrom/gcc-2.8.1-mips.tar.gz .

(d) tar xzvf gcc-2.8.1-mips.tar.gz

(e) rm gcc-2.8.1-mips.tar.gz

After these steps, there should be a directory named mips in /usr/local, which contains gcc mips crosscompiler.

3.3 Testing Nachos

Once you installed Nachos in your home directory, you can follow the steps below to test it:

(a) Move to directory /nachos-3.4/ and check its subdirectories:

c++example: This subdirectory contains examples of simple C++ programs and a short paper of programming language C++ (postscript file: c++.ps) written by Prof. Tom Anderson. This article is also included in the Selected Reading of this course. The purpose of these examples and the article is to provide a quick introduction to C++ for programming with Nachos.

code: This subdirectory contains the Nachos source code.

doc: This directory is empty at the moment.

(b) Move into subdirectory code and you will see the following files and directories in it:

Makefile.common ass2/ bin/ lab2/ lab5/ machine/ test/ userprog/
Makefile.dep ass3/ filesys/ lab3/ lab78/ network/ threads/ vm/

Here lab2/, lab3/, lab5/, lab78/ and ass2/ and ass3/ are your working directories for laboratory sessions and assignments, respectively. The remaining directories are the original Nachos directories.

The working directories for lab sessions 1, 4 and 6 are threads, filesys, and test and userprog, respectively.

(c) Move into directory threads/ and execute command make. You should see that the Nachos system is compiling. The last couple of lines of the output on the screen should be

....>>> Linking arch/unknowni386linux/bin/nachos <<<
g++ arch/unknowni386linux/objects/main.o .....
.....
.....
ln unknown-i386-linux/bin/nachos nachos

If you get this far, you have successfully compiled the smallest core of the Nachos system. You also should see symbolic link nachos in the current directly linked to arch/unknowni386linux/bin/nachos .

(d) Now you can test your Nachos system by executing command nachos in the

current directory. Note: it will be necessary to issue this command in the form ./nachos if your $PATH environmental variable does not include the current directory (.). The output on the screen should be 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...
```

3.4 The C++ Programming Language and the gdb Debugger

The Nachos source code is written in C++. This course does not require prior knowledge of C++. Of course, it is helpful if you have taken the course on objectoriented programming before. C++ is a complicated objectoriented language. We only use the part of C++ related to abstract data types and encapsulation. We do not use inheritance of C++ in Nachos. Dr. Tom Anderson wrote an article for quick introduction of C++ for the purpose of using Nachos. We included this article in the Selected Reading of this course. If you are not familiar with C++, read this article first. There are three C++ example programs in c++example directory discussed in this article. You only need to study program stack.cc. Read the code of stack.cc and make sure that you understand it.

If you are not familiar with GNU debugger gdb, you may want to learn how to use it. You can compile stack.cc by command make stack and run gdb for executable stack. You may need to use gdb when you need to debug your programs in this course.

Let us use the c++ program in directory c++example/ to show the steps to trace programs using gdb in emacs.

In the c++example directory, compile the stack program by typing make stack.

Run gdb from within emacs to debug this program as follows:

-- type emacs & to open an emacs window.

-- In emacs, type command Mx gdb .

-- in the command line of emacs, respond the full path name of program ``stack", nachos-3.4/c++example/stack after the gdb prompt. A gdb buffer will start. The initial prompt of the buffer is:

```
Current directory is /home/staff/ptang/units/204/98/linux/nachos3.4/c++example/
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386redhatlinux), Copyright 1996 Free Software Foundation, Inc...
(gdb)
```

-- At the prompt above, type gdb command list. The first 10 lines of the main program will be shown in the buffer as follows:

```
(gdb) list
120 }
121
122 //
123 // main
124 // Run the test code for the stack implementation.
125 //
126
127 int
128 main() {
129 Stack *stack = new Stack(10); // Constructor with an argument.
(gdb)
```

As you can see, the first statement of the main program is at line 129.

-- Set a break point at line 129 by typing gdb command break 129 and you will see:

```
(gdb) break 129
```

```
Breakpoint 1 at 0x80488b6: file stack.cc, line 129.
(gdb)
```

-- Then type gdb command run and the control will stop at the break point you just set up. You will see:

```
(gdb) run
Starting program: nachos3.4/c++example/stack
Breakpoint 1, main () at stack.cc:129
(gdb)
```

You will also see another buffer is opened in emacs as follows:

```
}
//
// main
// Run the test code for the stack implementation.
//
int
main() {
=> Stack *stack = new Stack(10); // Constructor with an argument.
stack>SelfTest();
delete stack; // always delete what you allocate
return 0;
}
```

The arrow => above is the break point where the execution stops currently.
-- In the original gdb buffer, continue to type gdb command next, (or just RETURN) and you will see the arrow is moved to the next statement.
-- In order to step into the method call stack>SelfTest() , type gdb command step.
You will see that the control steps into the method as follows:

```
void
Stack::SelfTest() {
=> int count = 17;
// Put a bunch of stuff in the stack...
while (!Full()) {
cout << "pushing " << count << "\n";
```

```
Push(count++);
}
```

-- you can print the value of any variable by gdb command print such as ``count'' or ``stack'' in the program. Try this command to watch the values of any variables that you think can demonstrate that the program is running correctly.
See the book ``Running Linux'' for more details of how to use emacs and gdb.

3.5 Clean Up

You need to clean up after you finish with Nachos in each directory. Simply execute make clean and all the object files, dependency files, binary file nachos as well as the symbolic link nachos in the current directory will be deleted. You can tell whether the directory is cleared by looking at whether the symbolic link nachos exists or not.

3.6 Questions

Answer these questions. They may be used as questions in assignments.

(a) What C++ files are used to build Nachos in threads/? Where are they located?

(b) What header files do these files depend on?

(c) Explain why INCPATH in threads/Makefile.local is defined as INCPATH += -I../threads -I../machine .

3.7 Things to do

We summarise the things to do in this lab session as follows:

install the LINUX operating system

install the Nachos system and gcc mips cross compiler (see Section 3.2)

compile and test the Nachos system installed (see Section 3.3)

exercise with gdb (see Section 3.4)

Laboratory 2: Makefiles of Nachos

4.1 Purpose

The purpose of this laboratory is

to understand the makefiles structure of Nachos system, and

to know how to set up a separate directory to develop a new version of Nachos system.

4.2 Makefiles Structure of Nachos

As we mentioned in Lab 1, the Nachos system can be compiled in a number of Nachos directories, ../threads/, ../filesys/, etc.

In each of these directories, there are two makefiles, Makefile and Makefile.local. In the parent directory ../code/, there are additional makefiles, Makefile.common and Makefile.dep, which are shared and invoked by the makefiles in all the Nachos directories.

Thus, the structure of the makefiles is as follows:

../code/Makefile.common
/Makefile.dep
|
|
/threads/Makefile
/Makefile.local
|
|
/filesys/Makefile
/Makefile.local
|
|

4.2.1 Makefile

This is the makefile used by make program, when you build a Nachos in any Nachos directory by typing command make or make all.

Examining this file reveals that it mainly includes two other makefiles:

```
include Makefile.local
include ../Makefile.common
```

4.2.2 Makefile.local

This makefile in each Nachos directory is to define a couple of important Macros:

CCFILES: the string to specify all the C++ files used to build the Nachos in this directory.

INCPATH: the string to define the include path for g++ to search head files (.h files) specified in the C++ programs.

DEFINES: the string for labels to be passed to g++.

Note that the assignment operator for INCPATH and DEFINES is +=, which means that the righthand side string is to be appended to the original contents of INCPATH and DEFINES.

4.2.3 Makefile.dep

This is the makefile to be included in Makefile.common. It defines a lot of systemdependent macros used by g++. The current Nachos distribution can be compiled on four different UNIX systems and all the object codes and binary executables as well as dependence files are to be placed in a particular directory under the directory arch in the Nachos directory shown as follows:

```
[threads]$ pwd
/home /nachos3.4/code/threads
[threads]$ ls arch
decalphaosf/ sunsparcsunos/
decmipsultrix/ unknowni386linux/
```

The systemdependent macros defined by Makefile.dep includes: HOST, arch, CPP, CPPFLAGS, GCCDIR, LDFLAGS and ASFLAGS. The definitions for the LINUX systems is:

```
# 386, 386BSD Unix, or NetBSD Unix (available via anon ftp
# from agate.berkeley.edu)
ifeq ($(uname),Linux)
HOST_LINUX=linux
HOST = -DHOST_i386 DHOST_LINUX
CPP=/lib/cpp
CPPFLAGS = $(INCDIR) -D HOST_i386 -D HOST_LINUX
arch = unknown-i386-linux
ifdef MAKEFILE_TEST
```

```
#GCCDIR = /usr/local/nachos/bin/decstation-ultrix
GCCDIR = /usr/local/mips/bin/decstation-ultrix
LDFLAGS = -T script -N
ASFLAGS = -mips2
endif
endif
```

Here, GCCDIR is the prefix for the gcc mips crosscompiler. Its definition shows why you need to install it in the /usr/local/ directory. This makefile also defines other macros dependent on the systemdependent macros above. They are:

```
arch_dir = arch/$(arch)
obj_dir = $(arch_dir)/objects
bin_dir = $(arch_dir)/bin
depends_dir = $(arch_dir)/depends
```

These macros show that in each of the systemdependent directories in arch directory, there are three directories to accommodate object codes, binary executable and dependence files, respec tively. For example, in the arch/unknown-i386-linux/ for your linux system, there are directories as follows:

```
[unknown-i386-linux]$ ls
bin/ depends/ objects/
```

4.2.4 Makefile.common

The file Makefile.common is the most complicated one and it defines all the rules for compiling a completed Nachos system.
It first includes the Makefile.dep. Then it defines the vpaths for various kinds of files as follows:

```
vpath %.cc ../network:../filesys:../vm:../userprog:../threads:../machine
vpath %.h ../network:../filesys:../vm:../userprog:../threads:../machine
vpath %.s ../network:../filesys:../vm:../userprog:../threads:../machine
```

It tells make where to find files if it cannot find them in the current directory. This is why you can build a Nachos in a new directory (other than ../threads/, ../filesys/) without copying the files which you do not need to modify.

This file then defines macros for object files (ofiles = $(cc_ofiles) $(c_ofiles) $(s_ ofiles)) , CFLAGS, and the ultimate target (program = $(bin_dir)/nachos). These definitions show that we are going to build the binary executable named nachos in the directory $(bin_dir) which is arch/unknown-i386-linux/bin/ in your linux system. The rule to build that target is defined in the following lines:

```
$(bin_dir)/% :
        @echo ">>> Linking" $@ "<<<"
        $(LD) $^ $(LDFLAGS) -o $@
        ln sf $@ $(notdir $@)
```

This rule is a static pattern rule. The % in the target can match any nonempty string in the target of other rules and these multiple rules will be combined to define the dependence. In our case, this rule is to be combined with rule:

```
$(program): $(ofiles)
```

In the command above, $@ represent the target which is
arch/unknown-i386-linux/bin/nachos in your linux system and $^ all the dependence files which are all object files defined by macro ofiles. The first command of this rule is simply to load these object files to form a binary executable. Note that LD is actually g++. The next command is to make a symbolic link to the binary executable. ln sf $@ $(notdir $@) actually expands to

```
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

The rule to make object codes from the C++ source codes is:

```
$(obj_dir)/%.o: %.cc
        @echo ">>> Compiling"    $<    "<<<"
        $(CC) $(CFLAGS) -c -o $@    $<
```

It is a static rule again. The % is to match any nonempty string. For example, this rule tells how to make arch/unknowni386linux/objects/main.o from main.cc. However, the object code should also depend on many head files (.h files) included by main.cc. This dependence relation for these head files is actually specified by another rule generated automatically.
First of all, we need to know which head files are included (directory and indirectly)

by the C++ source file during the compilation. g++ can do the search automatically for you. All you have to do is to use option MM . Let us do some experiments. In the ../threads/ directory, do the following (Use one line for the command. I split it here for clarity of presentation):

```
[threads]$ g++ MM -g Wall Wshadow -fwritablestrings
        -I../threads  -I../machine  -DTHREADS  -DHOST_i386
        -DHOST_LINUX -DCHANGED main.cc
```

You should see the results as follows:

```
main.o: main.cc copyright.h utility.h ../machine/sysdep.h \
../threads/copyright.h system.h thread.h scheduler.h list.h \
../machine/interrupt.h ../threads/list.h ../machine/stats.h \
../machine/timer.h ../threads/utility.h
```

This is the list of all the head files on which main.o depends. If any of these head files is updated, the main.cc should be recompiled to make a new main.o. You can also see that the output of the above command is actually a rule which can be included in the Makefile.common. This is exactly what is done by the remainder of the Makefile.common.
First of all, the makefile builds a dependence file in directory arch/unknown-i386-linux/depends/ for each source code file. The rule to do that is:

```
$(depends_dir)/%.d: %.cc
@echo ">>> Building dependency file for " $< "<<<"
@$(SHELL) ec '$(CC) MM $(CFLAGS) $< \
| sed '\"s@$*.o[ ]*:@$(depends_dir)/$(notdir $@) $(obj_dir)/&@g'\" > $
@'
```

Here CC is g++ and CFLAGS is the same flag which would be used for the real compiling. Note the MM option of g++. The rest of command is just to create a dependence file in directory arch/unknown-i386-linux/depends/ after appending the prefix arch/unknown-i386-linux/objects/ to the object file name.
For example, for main.cc, this rule will create a new file named main.d in directory arch/unknown-i3860-linux/depends/ whose contents are:

```
arch/unknown-i386-linux/depends/main.d
arch/unknown-i386-linux/objects/main.o:
```

```
main.cc copyright.h utility.h ../machine/sysdep.h \
../threads/copyright.h system.h thread.h scheduler.h list.h \
../machine/interrupt.h ../threads/list.h ../machine/stats.h \
../machine/timer.h ../threads/utility.h
```

You can check if these files exist, after you make the nachos.
Then, there is an include statement in Makefile.common:

```
include $(dfiles)
```

This means that Makefile.common includes all the dependence files it created. The contents of these files which are all makefile rules become part of this makefile. It is these rules that will be combined with the rule of compiling to make the object codes. As a result, we have a complete list of dependence files to make each object code.
Another important use of this technique is that we can see what head files are used in compiling a source code by examining the corresponding dependence file. This is very helpful when you are building a new version of Nachos in a separate directory which contains some modified source and head files and you want to be sure that these modified files are actually used in compilation.

4.3 Building a Modified Nachos in Another Directory

The current Nachos allows you to build different Nachos in directories ../threads/, ../filesys/ and ../userprog/. You will be required to extend or modify Nachos in the assignments and lab sessions. It is always a good idea to change only the relevant files in a separate directory and build the new Nachos there. You want to use the files which are not modified in their original directories.
Let us assume that you are required to build a new Nachos in a separate directory called ../lab2.
Suppose that you need to change class Scheduler. You do not want to change the original scheduler.h and scheduler.cc in directory ../threads/. What you can do is to copy these two files from ../threads/ to ../lab2/ and make changes to them in ../lab2/.
Suppose that you want to build the new Nachos in ../lab2/ using the new scheduler.h and scheduler.cc there. All the other files of the new Nachos should be the original ones from directories ../threads/ ../machine/, etc.
In order to do that, you need to copy the empty ../arch/ directory tree recursively and files Makefile and Makefile.local from ../threads/ to ../lab2.
The last task is to modify makefiles Makefile and Makefile.local so that you can build the new Nachos properly. Makefile in ../lab2 does not need changes, but you do need

to change Makefile.local in ../lab2/.

Makefile.local basically defines macro CCFILES and redefines the include path macro INCPATH. The definition of CCFILES does not need changes, because make will follow the vpaths to find the required source files if they are not in the current directory.

The redefinition of INCPATH needs changes.

In the following, I provide two solutions to this problem.

First Solution: You can change the redefinition of INCPATH as follows:

```
INCPATH += -I../lab2 -I../threads -I../machine
```

That is, add -I../lab2 before -I../threads so that C preprocessor (cpp) of g++ will search ../lab2/ first when it processes include macros in the source files. However, this simple change only does not solve all the problems. The current contents of ../lab2/ are as follows:

```
[ptang@zibal lab2]$ ls
Makefile Makefile.local arch/ scheduler.cc scheduler.h
```

We then type make to build the new Nachos as follows:

```
[ptang@zibal lab2]$ make
...
>>> Linking arch/unknown-i386-linux/bin/nachos <<<
g++ arch/unknown-i386-linux/objects/main.o .....
.....
ln -sf arch/unknowni386linux/bin/nachos nachos
[ptang@zibal lab2]$ ls
Makefile        Makefile.local~   nachos@      scheduler.h
Makefile.local  arch/             scheduler.cc
```

But, in this Nachos, only the new scheduler.cc uses the new scheduler.h. This can be shown by the following script:

```
[ptang@zibal lab2]$ touch scheduler.h
[ptang@zibal lab2]$ make
>>> Building dependency file for scheduler.cc <<<
>>> Compiling scheduler.cc <<<
```

```
g++ -g -Wall -Wshadow -fwritablestrings -I../lab2 -I../threads
-I../machine -DTHREADS -DHOST_i386 -DHOST_LINUX -DCHANGED
-c -o arch/unknown-i386-linux/objects/scheduler.o scheduler.cc
>>> Linking arch/unknown-i386-linux/bin/nachos <<<
g++ arch/unknown-i386-linux/objects/main.o .....
.....
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

Other classes which depend on scheduler.h use the old scheduler.h in ../threads/. This can be shown by the following script:

```
[ptang@zibal lab2]$ touch ../threads/scheduler.h
[ptang@zibal lab2]$ make
>>> Building dependency file for ../machine/timer.cc <<<
...
>>> Compiling ../threads/main.cc <<<
..
>>> Linking arch/unknowni386linux/bin/nachos <<<
g++ arch/unknown-i386-linux/objects/main.o .....
.....
ln -sf arch/unknown-i386-linux/bin/nachos nachos
[ptang@zibal lab2]$
```

This is because when g++ MM generates dependences, it looks for the .h files in the same directory as the .cc file. For example, ../threads/main.cc indirectly includes scheduler.h (through system.h). Therefore g++ MM looks for the scheduler.h there first and gener ates the dependence which includes string ../threads/scheduler.h (check the contents of main.d in ../lab2/arch/unknowni386linux/depends/).
In order to avoid this, you need to copy all the files in ../threads/ which directly and indirectly include scheduler.h there. To find the minimum set of these files, you can use grep command to search for the files which contain string scheduler.h as follows:

```
[ptang@zibal threads]$ grep scheduler.h *
grep: arch: Is a directory
scheduler.cc:#include "scheduler.h"
scheduler.h:// scheduler.h
system.h:#include "scheduler.h"
```

```
[ptang@zibal threads]$
```

We then search string system.h because file system.h includes scheduler.h.

```
[ptang@zibal threads]$ grep system.h *
grep: arch: Is a directory
main.cc:#include "system.h"
scheduler.cc:#include "system.h"
synch.cc:#include "system.h"
synchtest.cc:#include "system.h"
system.cc:#include "system.h"
system.h:// system.h
thread.cc:#include "system.h"
threadtest.cc:#include "system.h"
[ptang@zibal threads]$
```

This means that the minimum set of files we need to copy from ../threads/ to ../lab2/ are

```
system.h
main.cc
synch.cc
synchtest.cc
system.cc
thread.cc
threadtest.cc
```

Then we make the new Nachos and the contents of ../lab2/ should be as follows:

```
[ptang@zibal lab2]$ ls
Makefile        arch/       scheduler.cc   synchtest.cc   thread.cc
Makefile.local  main.cc     scheduler.h    system.cc      threadtest.cc
Makefile.local  nachos@     synch.cc       system.h
```

Now we can test that it works OK as follows:

i. We first change the timestamp of scheduler.h in ../lab2/ and then make Nachos again. The make command should cause recompiling of a lot of modules:

```
[ptang@zibal lab2]$ touch scheduler.h
```

```
[ptang@zibal lab2]$ make
>>> Building dependency file for ../machine/timer.cc <<<
...
>>> Compiling main.cc <<<
g++ -g -Wall -Wshadow -fwritablestrings -I../lab2 -I../threads
-I../machine -DTHREADS -DHOST_i386 -DHOST_LINUX -DCHANGED
-c -o arch/unknown-i386-linux/objects/main.o main.cc
...
>>> Linking arch/unknown-i386-linux/bin/nachos <<<
g++ arch/unknown-i386-linux/objects/main.o .......
.............
ln -sf arch/unknown-i386-linux/bin/nachos nachos
[ptang@zibal lab2]$
```

ii. We then change the timestamp of ../threads/scheduler.h and try the make Nachos again. This time, none of the modules should be recompiled and it should be shown that the existing Nachos is updated.

```
[ptang@zibal lab2]$ touch ../threads/scheduler.h
[ptang@zibal lab2]$ make
make: `arch/unknown-i386-linux/bin/nachos' is up to date.
[ptang@zibal lab2]$
```

Second Solution: The second solution is much simpler than the first one. It takes advantage of a feature of the preprocessor of g++ defined by I in the command. Here is the description of this include option of g++ (obtained through man gcc).

```
-I
....
In addition, the `-I' option inhibits the use of
the current directory (where the current input file
came from) as the first search directory for `#in
clude "file"'. There is no way to override this
effect of `-I'. With `-I.' you can specify
searching the directory which was current when the
compiler was invoked. That is not exactly the same
as what the preprocessor does by default, but it is
often satisfactory.
```

...

This means that -I prohibits including the .h files from the same directory of the .cc file processed. It therefore forces the preprocessor to look for .h files according to the path defined by I after the I . Therefore, we can use the redefinition of INCPATH in ../lab2/Makefile.local as follows:

```
INCPATH += -I- -I../lab2 -I../threads -I../machine
```

without copying any files from ../threads/ other than scheduler.cc and scheduler.h. The contents of ../lab2/ after Nachos is made is as follows now:

```
[ptang@zibal lab2]$ ls
Makefile        Makefile.local      nachos@             scheduler.h
Makefile.local  arch/               scheduler.cc
[ptang@zibal lab2]$
```

We can test that it works OK as follows:

```
[ptang@zibal lab2]$ touch ../threads/scheduler.h
[ptang@zibal lab2]$ make
make: `arch/unknowni386linux/bin/nachos' is up to date.
```

If we touch the scheduler.h in the current directory ../lab2, it will make Nachos recom piled as follows:

```
[ptang@zibal lab2]$ touch scheduler.h
[ptang@zibal lab2]$ make
>>> Building dependency file for ../machine/timer.cc <<<
...
>>> Compiling ../threads/main.cc <<<
g++ -g -Wall -Wshadow -fwritablestrings -I -I../lab2 -I../threads
-I../machine -DTHREADS -DHOST_i386 -DHOST_LINUX -DCHANGED
-c -o arch/unknown-i386-linux/objects/main.o ../threads/main.cc
...
>>> Linking arch/unknowni386linux/bin/nachos <<<
g++ arch/unknowni386linux/objects/main.o .....
.....
ln -sf arch/unknown-i386-linux/bin/nachos nachos
```

[ptang@zibal lab2]$

4.4 Things to Do

Your tasks in this lab session are as follows:

Read Section 4.2 and make sure you understand the makefile structure of Nachos.

Experiment with the two solutions to build a new Nachos in a separate directory described in

Section 4.3. Make sure you understand why both solutions are correct.

Laboratory 3: Synchronization Using Semaphores

5.1 Objectives

In this laboratory session, you are required to write a test program for the producer/consumer problem using semaphores for synchronization. After completing the session, you will

have a understanding in Nachos of
how semaphores are implemented, and
how the producer/consumer problem is implemented using semaphores
know how to create concurrent threads in Nachos, and
know how to test and debug programs in Nachos.

The work of this laboratory session will prepare you for the programming task in Assignment 2.

5.2 Background

5.2.1 Semaphores

Semaphores are one of the most commonly used synchronization schemes for concurrent processes or threads. Section 4.4 of the textbook gives a full description of the concept and implementation of semaphores. In Nachos, semaphores are implemented as class Semaphore. The implementation of Semaphore in Nachos is different from the textbook.
The implementation of semaphores in Nachos can be found in ../threads/synch.cc.

5.2.2 The Producer/Consumer Problem

The producer/consumer problem is one of the problems encountered frequently in operating systems design. Both producer and consumer threads access the same ring buffer in the shared memory. The producers produce items and put them in the ring buffer, while the consumers take and consume items from the buffer. A producer has to be blocked when the buffer is full and resumed when it becomes nonfull. Similarly, a consumer has to be blocked when the buffer is empty and resumed when it becomes nonempty. Consequently, producers and consumers need a mechanism for synchronization.

5.2.3 Nachos main Program

When you start nachos, the first program module executed is the main program. Every subdirectory of Nachos can have a main.cc. Take a look at the main.cc in ../threads. You need to study

how the command line of nachos is interpreted,

how the Nachos kernel is initialized, and

how the thread for the main program creates another thread executing function SimpleThread(int which). The source code of SimpleThread(int which) can be found in ../threads/threadtest.cc.

5.3 Things to Do

In this laboratory session, you are required to implement the producer/consumer algorithm in Nachos using semaphores for synchronization.

In your ../lab3/ directory, you can find files: main.cc, prodcons++.cc, ring.cc and ring.h.

Files ring.cc and ring.h define and implement a class Ring for the ring buffer used by producers and consumers. These two files are complete and you do not need to change any part of them. main.cc in this directory is modified from the version in ../threads/. It is complete and you do not need to change it.

In the new main.cc, function ProdCons() is called instead of ThreadTest(). Function ProdCons() is defined in file prodcons++.cc. This file is supposed to include the code to create producer and consumer threads as well as implement the producer/consumer algorithm described in the textbook.

However, this file is not complete yet. Your task in this laboratory session is to complete this file and make the producer/consumer algorithm work.

File prodcons++.cc contains all the data structures and the interfaces of the functions. There are detailed comments in the file about what needs to be done in each part of the code. Because all the interfaces of the functions are present, the file is compilable. You can execute make in the ../lab3/ now to make a new Nachos for producer/consumer problem. (But it won't work yet because prodcons++.cc is incomplete.)

Your tasks are:

Read ring.h and ring.cc and make sure that you understand everything in them.

Read main.cc.

Read prodcons++.cc and make sure that you understand

the structure of the program

the task to complete the program

Complete all programs in file prodcons++.cc.

Compile a new nachos by command make and test if your program is working or not. The output files of an example run of the problem with two consumers and two producers each of which produces four messages should be like this:

the contents of tmp 0:

producer id > 0; Message number > 0;
producer id > 0; Message number > 1;
producer id > 1; Message number > 3;

the contents of tmp 1:

producer id > 0; Message number > 2;
producer id > 0; Message number > 3;
producer id > 1; Message number > 0;
producer id > 1; Message number > 1;
producer id > 1; Message number > 2;

What is the criteria for testing this program? According to the concepts of producer/consumer, a correct implementation should guarantee the following:

all the messages produced by the producer threads are received and recorded in the output files, and

no messages are received and recorded more than once

messages that are from the same producer and received by the same consumer should be received in the increasing order.

You can run nachos with different random number seeds by nachos rs seednumber and check that all results satisfy the above criteria.

Laboratory 4: Nachos File System

6.1 Objectives

The purpose of this laboratory session is to study the functionality of the file system in Nachos. The file system in Nachos is designed to be small and simple so that you can read all its source code in a short period of time. Before starting to read the code, it is very useful to get an idea of what functionality the Nachos file system offers. In this laboratory session, you will run the commands of the Nachos file system and watch the effects on the simulated hard disk in Nachos.

On the completion of this laboratory session, you should know

what is the functionality of the Nachos file system, and

how to examine the contents of the simulated hard disk in Nachos.

6.2 Compiling the Nachos file system

It is very simple to compile Nachos with its file system. You simply move to the directory ../filesys and execute command make. A new version of Nachos with its file system included will be made in the directory. The Makefile in ../filesys/ includes both Makefile.local files from ../threads/ and ../filesys/. The Makefile.local in ../filesys/ is as follows:

```
ifndef MAKEFILE_FILESYS_LOCAL
define MAKEFILE_FILESYS_LOCAL
yes
endef

# Add new sourcefiles here.

CCFILES +=bitmap.cc\
          directory.cc\
          filehdr.cc\
          filesys.cc\
          fstest.cc\
          openfile.cc\
          synchdisk.cc\
```

```
        disk.cc

ifdef MAKEFILE_USERPROG_LOCAL
DEFINES := $(DEFINES:FILESYS_STUB=FILESYS)
else
INCPATH += I../userprog I../filesys
DEFINES += DFILESYS_NEEDED DFILESYS
endif

endif # MAKEFILE_FILESYS_LOCAL
```

This means that this version of Nachos uses C++ files listed above in addition to the files used to compile the Nachos in ../threads/. Most of these additional files exist in the current directory. Some of them are in other directories such as ../userprog/. The make utility program will find them automatically due to the VPATH defined in ../Makefile.common.

6.3 Usage of Nachos File System Commands

The Usage of Nachos commands is defined in ../threads/main.cc and ../threads/system.cc. In particular, the commands related to the file system are listed below. The optional flag d f is used to print all the debug information related to the file system.

nachos [-d f] -f . This is used to format the simulated hard disk named DISK before any other file system commands can start.

nachos [-d f] -cp unix filename nachos filename. This command copies a UNIX file named unix filename in your UNIX system to a Nachos file named nachos filename in the Nachos file system. This is currently the only way to create a file in the Nachos file system.

nachos [-d f] -p nachos filename. This command displays the contents of the nachos file named nachos filename (similar to UNIX command cat).

nachos [-d f] -r nachos filename. This command removes the nachos file named nachos filename (similar to UNIX command rm).

nachos [-d f] -l . This command lists the names of all the nachos files on the screen (similar to UNIX command ls).

nachos [-d f] -D . This command prints all the contents of the entire file system including the bitmap, the file headers, the directory and the files.

nachos [d f] -t . This command tests the performance of the file system. It is not work ing yet.

To understand how these commands work, you need to read ../threads/main.cc and ../filesys/fstest.cc.

6.4 Test Files

In the subdirectory test/ in ../filesys, there are three files to be used when testing the Nachos file system: small, medium and big. Take a look at the contents of them.

6.4.1 UNIX command od

You need to use the UNIX command od (Octal Dump) to examine the simulated hard disk when you debug the Nachos file system.

Read the manual page of od (by typing man od).

Execute command od c test/small . You should see

```
0000000 T  h  i  s     i  s     t  h  e        s  p  r  i
0000020 n  g        o  f     o  u  r        d  i  s  c  o  n
0000040 t  e  n  t  .     \n
0000046
```

on your screen. Each line displays 16 characters. The column on the left shows the offset in octal of the first character of each line. For example, the offset of the first character of the second line (``n'') is 0000020 in octal which is 16 in decimal.

6.5 Things to Do

6.5.1 Compiling Nachos File System

Follow the description in Section 6.2 to compile the Nachos with its file system in ../filesys/).

6.5.2 Testing Nachos File System

Execute the following commands and check the results as described:

Execute nachos -f . Nachos should have created the simulated hard disk called DISK in your current directory.

Execute nachos -D to dump the whole file system on the simulated hard disk DISK and you should have the following dump:

```
....
FileHeader contents. File size: 128. File blocks:
2
File contents:
\1f\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
31


\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Directory file header:
FileHeader contents. File size: 200. File blocks:
3 4
File contents:
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Bitmap set:
0, 1, 2, 3, 4,
Directory contents:
Directory contents:
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
Ticks: total 5500, idle 5030, system 470, user 0
Disk I/O: reads 10, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
Cleaning up...
```

We have deleted the output from function ThreadTest() in the above dump. You can get rid of them by making a copy of ../threads/main.cc in your current ../filesys directory and comment out the line of invoking function ThreadTest().

This dump shows that the Nachos file system has been created on DISK. There are no files at the moment in the only directory of the Nachos file system.

Execute od c DISK and you should have the following dump on the screen:

```
0000000 211 g E 200 \0 \0 \0 001 \0 \0 \0 002 \0 \0 \0
0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0000200 \0 \0 \0 \0 \0 \0 \0 002 \0 \0 \0 003 \0 \0 \0
0000220 004 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000240 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0000400 \0 \0 \0 \0 037 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000420 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

Execute nachos cp test/small small to copy file small into the Nachos file system. Use nachos l , nachos p and nachos D to make sure you do create a new file named small in the Nachos file system. Watch the dump of od c DISK again to see what has been changed.

Continue the experiment with other nachos commands such as nachos r on more files.

6.6 Questions

According to the result of the last command nachos D and the result of od c DISK , how many files are there on the hard disk DISK?

What are the sector numbers of data blocks for file big?

What is the sector number of the disk to store the file header for file big?

The sector size of the Nachos hard disk is 128 bytes. Could you check the result of od -c DISK to make sure that the data blocks and the file header of big are in the right places in the disk?

Laboratory 5: Extendable Files

7.1 Objectives

The purpose of this lab session is to help you start to work on extending the Nachos file system, the programming task of Assignment 3.
The work required in this laboratory session itself is part of Assignment 3.
The Nachos file system is a simple file system with many restrictions. One of them is that the size of the file is not extendable: once you specify the size of a file upon its creation, the size of the file is fixed throughout its lifetime. In this laboratory session, you are going to extend the Nachos file system to allow the size of files to be extended. In particular, we want the new Nachos file system to have the following features:

When a file is created, its initial size can be set to 0.
The size of a file can be increased if more data are written to the file.

For example, if the initial size of a file is 100 bytes and a write operation for 100 bytes data from the position 50 (the first byte is at position 0) will extend the size of the file to 150 bytes. The situation is illustrated in Figure 7.1, in which (a) represents the initial size (100 bytes) of the file. The light shadow represents the current contents of the file. (b) represents the new 100 bytes of data to be written from position 50. (c) shows the extended size of the file with dark shadow representing the new data.
The current Nachos file system does not allow the file size to be extended like this. Your task is to design and implement the extension of the Nachos file system to have these new features.

7.2 Analysis

The Nachos file system consists of the following modules:

class Disk
class SynchDisk
class BitMap
class FileHeader
class OpenFile
class Directory
class FileSystem

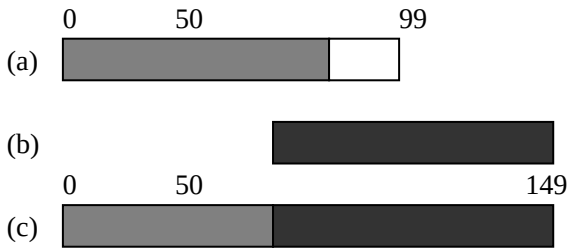


Figure 7.1: Extension of a file

The structure of the file system is shown in Figure 7.2.

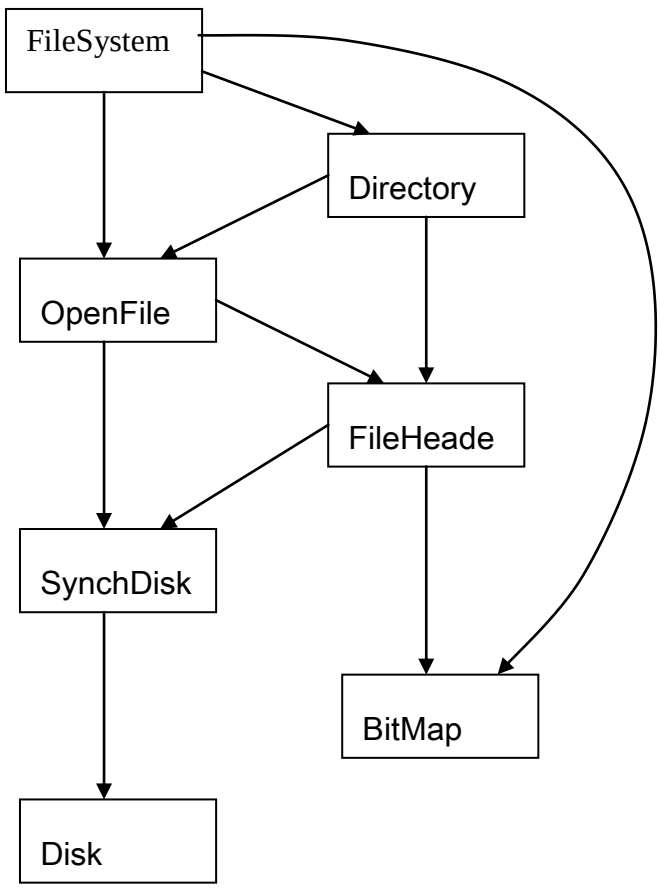


Figure 7.2: Structure of Nachos File

Before we get into the implementation, let us have a discussion on the design and analysis. We are not building a file system from scratch. Instead, we are extending an existing system. We want to make as less changes as possible to the original system.
The questions are:

What modules need to be changed and what module can be used without changes?

In those modules that need to be changed, which functions need to be changed and how?

In those modules that need to be changed, do you need to add new functions or variables?

Do you need to move some variables around in the modules to be changed, or across the modules?

You need to find answers to these questions before you do anything of implementation and coding.

7.3 Things to Do

7.3.1 Analysis

Answer the questions in Section 7.2. Write down your answers on paper. Verify your answers carefully by reading the original source code of the Nachos file system. The more time you spend in this step, the less troubles and difficulties you will have in later stages. If your answers are wrong, your file system will definitely not work.

7.3.2 Design and Implementation

After you finish the analysis, you can start to design and implement the changes to the existing Nachos file system. Your working directory is ../lab5/. There is one directory test in ../lab5 which contains the test files. You need to set up the arch subdirectory hierarchy and your new makefiles in ../lab5. You also have to make a decision as to which files in ../filesys/ need to be copied to ../lab5 in order to modify them.
The two files main.cc and fstest.cc in ../lab5/ are new and include many new file system commands to test the new features required. We will discuss these new commands in Section 7.4. main.cc is complete and you should not change it. fstest.cc is almost complete except that you need to uncomment four lines in it. In both functions Append(...) and NAppend(...), you can see the following three lines:

```
// Write the inode back to the disk, because we have changed it
// openFile>WriteBack();
// printf("inodes have been written back\n");
```

You need to uncomment the last two lines after you add Writeback() function to class OpenFile. Why do you need this function for OpenFile? Think about it.
Use the following guidelines for the implementation and coding.

Interface: In this stage, you need to build the new interface between the modules according to your plan of changing the file system. You only need to change class definitions in .h files and function heads in .cc files. You do not change the function bodies at this stage.
Make one change at a time and compile the system to make sure that you do not bring syntax and linking errors before making the next change.

Coding and Testing: Work out the strategies to change the functions bodies and test the resulting codes. You may divide the whole process into a number of smaller stages and each stage has a goal and the related test strategy.
Follow the order of stages specified and make sure the system is working before proceeding to the next stage.

7.4 Testing the New File System

We need commands to test the new features of the Nachos file system. These new commands have been implemented in main.cc and fstest.cc in ../lab5/ for you. They are:

nachos [-d f] -ap unix filename nachos filename. This command appends a UNIX file named unix filename in your UNIX system to the end of a Nachos file named nachos filename in the Nachos file system. It is used to test whether we can extend the file size as we append a file to the end of an existing Nachos file.

nachos [-d f] -hap unix filename nachos filename. This command overwrites the Nachos file (named nachos filename) from its middle with a UNIX file (named unix filename). If the length of the UNIX file exceeds the half of that of the Nachos file, the Nachos file size should be extended.

Read files main.cc and fstest.cc in ../lab5/ and make sure that you understand how these new commands are implemented.
When testing the new Nachos file system, start with a fresh DISK by removing the old DISK and executing nachos f . Then execute the following commands in the order:

```
nachos -cp test/small small
nachos -ap test/small small
nachos -cp test/empty empty
nachos -ap test/medium emtpy
```

Your file system dump with nachos D at this point should be like this:

```
Bit map file header:
FileHeader contents. File size: 128. File blocks:
2
File contents:
\ff\7\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Directory file header:
FileHeader contents. File size: 200. File blocks:
3 4
File contents:
\1\12\11@\5\0\0\0small\0\0\0\0G\5\8\1H\5\8\8\0\0\0empty\0\0\0\0\0\0\0G
\5\8\0\0\0\0\0\0\00\0\0\0\e0\12\11@\0\12\11@\10G\5\8X\13\0\0\d0\15\11@
\18\0\0\0\0\12\11@\c8\12\11@\80G\5\8\18\0\0\0`\0\0\0\0\0\0\0\0\0\0\2\0
\0\0\a0F\5\8\0\0\0\0\0\0\0\18\0\0\0
\0\0\0\0\1\0\0\0\a8D\5\8\0\11\0\0\c8\15\11@H\0\0\0\f8\12\11@\f8\12\11@\0F
\5\8\0\0\0\0\0\0\00\0\0\0\e0\12\11@\0\12\11@\90D\5\8\c8\15\11@HH\5\8\18
\0\0\0
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
Directory contents:
Name: small, Sector: 5
FileHeader contents. File size: 168. File blocks:
6 7
File contents:
small file small file small file\asmall file small file small
file\a***end of file***\asmall file small file small file\asm
all file small file small file\a***end of file***\a
Name: empty, Sector: 8
FileHeader contents. File size: 162. File blocks:
9 10
File contents:
medium file medium file medium file\amedium file medium file med
ium file\amedium file medium file medium file\amedium file medi
um file medium file\a***end of file***\a
```

```
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 8490, idle 8000, system 490, user 0
Disk I/O: reads 16, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

This dump shows that the bitmap file is of size 128 bytes (one sector) and is located in sector 2. It also shows the contents of the bitmap file. We know that the inode of the bitmap file is located in sector 0.
The dump also shows that the directory file has 200 bytes (two sectors) and the data blocks are located in sectors 3 and 4. The inode of the directory file is in section 1 (not shown in the dump) as we know.
The file named small is shown to have 168 bytes (two sectors) and its inode is located in sector 5. The data block of the file is in sectors 6 and 7.

You also need to test the cases where part of the file is overwritten and the file is being extended. Design your own test programs to make sure that your new file system works in all different kinds of situations.

Laboratory 6: Nachos User Programs and System Calls

8.1 Objectives

In this lab, you are required to

experiment with user programs in Nachos, and

get familiar with the code which you need to implement Nachos system calls.

The purpose is to enable you to gain a understanding of

how user processes are started

how user processes interact with an OS kernel through system calls

how system calls are implemented

8.2 Background

As we mentioned in the Study Book, Nachos uses a MIPS machine simulator to run user programs. The binary executable files of Nachos user programs are generated by gcc MIPS crosscompiler and then converted from the COFF format to the NOFF format by a program called coff2noff in the ../test directory.

To run a Nachos user program in ../userprog, you use command nachos x ../test/xxxx , where xxxx is one of the nachos executables in ../test/.

You need to compile the Nachos in ../userprog first, of course.

8.3 Things to Do

8.3.1 Nachos executables

In this task, you need to study the Makefiles in ../test/ to understand how Nachos user programs (executables) are generated. There are five user programs already compiled and there is a symbolic link to each of them in the directory.

change halt.c to become

```
#include "syscall.h"
int
main()
{
```

```
int i,j,k;
k = 3;
i = 2;
j = i1;
k = i
  j + k;
Halt();
/* not reached */
}
```

recompile it to have a new halt

To see the assembly code of this program, you can use the following to command to generate halt.s:

```
/usr/local/mips/bin/decstationultrixgcc -I../userprog
-I../threads S halt.c
```

Study halt.s. See how the stack frame for function main is created and deleted. See what instructions are generated by the compiler for the statements in this C program.

Move to ../userprog/ and

(a) compile the Nachos kernel there by typing ``make"

(b) run the new user program halt on Nachos. The command is nachos -x ../test/halt. You can add debug flag -d m to print every instruction which the MIPS simulator runs. You can also run this from within gdb for a slow trace. This is a way to gain an under standing of how user programs are started and executed in Nachos.

8.3.2 Page Table Dumping

You will be required to implement multiprogramming in Nachos. It is important to understand how a user process is created. As pointed out by the Study Book, a user process in Nachos is evolved from a thread by forming an address space. Nachos uses paging for its memory management. In your programming assignments, you may find the following page table dump function useful:

```
void AddrSpace::Print() {
printf("page table dump: %d pages in total\n", numPages);
printf("========================================\n");
printf("\tVirtPage, \tPhysPage\n");
for (int i=0; i < numPages; i++) {
```

```
printf("\t%d, \t\t%d\n", pageTable[i].virtualPage, pageTable[i].physicalPage);
}
printf("==========================================\n\n");
}
```

Add this function (or your own similar dump function) in your class AddrSpace and invoke it when you create a new address space.
You will see that the smallest Nachos user program halt.c takes 11 pages.

8.3.3 Making Address Space Larger

Sometimes, you may want to make a user program with a larger address space. One way to do that is to add static array in your program. For example, if you add an static integer array in user program halt.c as follows:

```
#include "syscall.h"
static int a[40];
int
main()
{
Halt();
/* not reached */
}
```

the size of the address space increases to 12 pages.

Laboratory 7: Extension of AddrSpace

9.1 Objectives

In this lab, you are required to

extend the current implementation of class AddrSpace so that Nachos can run multiple user programs.

complete the print function for AddrSpace as mentioned in Lab 6.

This lab will get you ready to implement the nachos system calls Exec() and Exit() in Lab 8.

9.2 Background

Suppose that we want Nachos to load and run a user program as follows:

```
#include "syscall.h"
int
main()
{
Exec("../test/exec.noff");
Halt()
}
```

and the C program for ../test/exec.noff is as follow:

```
#include "syscall.h"
int
main()
{
Halt()
}
```

This means that Nachos has to load the user program ../test/exec.noff while the user program ../test/bar.noff is running. The current implementation of Nachos does not allow this to happen, because it always uses the frames 0, 1, ... of the physical memory for any address space as shown by the following code in the constructor of AddrSpace:

```
..
// first, set up the translation
pageTable = new TranslationEntry[numPages];
for (i = 0; i < numPages; i++) {
pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
pageTable[i].physicalPage = i;
pageTable[i].valid = TRUE;
pageTable[i].use = FALSE;
pageTable[i].dirty = FALSE;
pageTable[i].readOnly = FALSE; // if the code segment was entirely on
// a separate page, we could set its
// pages to be readonly
}
..
```

9.3 Things to Do

You have two tasks in this lab:

Extend the Nachos system so that it can run multiple user programs.

Add the print function to AddrSpace class. You will need this class when you test the exten sion above and implement system calls Exec().

9.4 Bitmap Class

To complete the task above, you may find the bitmap class in ../userprog/ useful. Read the code of this class in ../userprog/bitmap.h and ../userprog/bitmap.cc and try to figure out how to use it in this lab task.

9.5 Analysis and Design

Again, before you program, you have to think about what changes you need to bring to the current Nachos. Don't start to program unless you finish the analysis and design of the task.

Laboratory 8: System Calls Exec() and Exit()

10.1 Objectives

In this lab, you are required to implement two Nachos systems calls: Exec(), Exit().

10.2 Working directory

The working directory of Labs 7 and 8 is ../lab78/ . You need to copy files from ../userprog/ and set up your own Makefiles and the arch subdirectory hierarchy in this new directory. If you cannot make it work, you can always use the original ../userprog/ and change files there.

10.3 How to make new Nachos user test programs

You will need to write a lot of new Nachos user programs for testing your work. To make new user programs is easy. Suppose that you have written a new test C program called ``exec.c'' to test your implementation of Exec() as follows:

```
#include "syscall.h"
int
main()
{
SpaceId pid;
pid = Exec("../test/halt.noff");
Halt();
/* not reached */
}
```

To make the NOFF user program for it, you

add exec in the list of targets in Makefile in ../test/ as follows:

```
...
# User programs. Add your own stuff here.
#
# Note: The convention is that there is exactly one .c file per target.
# The target is built by compiling the .c file and linking the
# corresponding .o with start.o. If you want to have more than
# one .c file per target, you will have to change stuff below.
```

```
targets = halt shell matmult sort exec
...
```
then type make to generate the exec.noff file.

You can also generate the assembly code of this C program by typing: make exec.s.

10.4 Design Issues

There are a couple of design issues you need to address before you can start to program. We list a few major ones in this section.

10.4.1 Openfile for the User program

To build the address space for a user program in a file, you simply open the file and then invoke the constructor of AddrSpace as shown in function StartProcess() as follows:

```
void
StartProcess(char *filename)
{
OpenFile *executable = fileSystem>Open(filename);
AddrSpace *space;
if (executable == NULL) {
printf("Unable to open file %s\n", filename);
return;
}
space = new AddrSpace(executable);
...
}
```

Note that the filename above is an object (string) in the Nachos kernel. The filename as the argument of Exec() is an object (string) in the user program. This can be shown by the following
assembly code of exec.c we mentioned above:

```
        .file 1 "exec.c"
gcc2_compiled.:
__gnu_compiled_c:
        .rdata
        .align 2
```

```
$LC0:
        .ascii "../test/halt.noff\000"
        .text
        .align 2
        .globl main
        .ent main
main:
        .frame $fp,32,$31           # vars= 8, regs= 2/0, args= 16, extra= 0
        .mask 0xc0000000,4
        .fmask 0x00000000,0
        subu $sp,$sp,32
        sw $31,28($sp)
        sw $fp,24($sp)
        move $fp,$sp
        jal __main
        la $4,$LC0
        jal Exec
        sw $2,16($fp)
        jal Halt
$L1:
        move $sp,$fp
        lw $31,28($sp)
        lw $fp,24($sp)
        addu $sp,$sp,32
        j $31
        .end main
```

The question is how to copy the string of filename from the user address space to the kernel. You have to address this problem in order to implement Exec() correctly.

10.4.2 Advance PC

You should have seen that the implementations of all the systems calls start with the corresponding assembly routines provided in start.s. For example, the routine for Exec() is

```
Exec:
    addiu $2,$0,SC_Exec
    syscall
```

```
    j $31
    .end Exec
```

The machine simulation of the execution of instruction syscall within the big switch statement in ../machine/mipssim.cc is as follows:

```
case OP_SYSCALL:
RaiseException(SyscallException, 0);
return;
```

Note return instead break above. This is because the machine has to restart the same instruction after exceptions are handled in general. But, the system call exception is a special case. It does not need to restart the same syscall instruction after the exception is serviced.

You can either change the code above or advance the PC in your system call exception handlers. If you chose the latter, the following function to advance the PC may be useful:

```
void AdvancePC() {
   machine>WriteRegister(PCReg, machine>ReadRegister(PCReg) + 4);
   machine>WriteRegister(NextPCReg, machine>ReadRegister(NextPCReg) + 4);
}
```

10.4.3 SpaceId

Exec() returns a value of type SpaceId. This value will be used as the argument of Join() to identify the newlycreated user process.

There are two questions about this value:

how this value (process id) is generated

how to record this value in the kernel so that Join() will be able find the corresponding thread by providing this value.

You have to address this issue in order to implement Exec() correctly.

10.4.4 Exit Status of Exit()

The argument of Exit()is an integer as the exit state of the user process. Join() needs to return this exit status. The user process may exit before the other (parent) user process calls Join(). This means that the exit status needs to be saved somewhere. But, how is this done?

You have to address this issue in order to implement Exit() correctly.

10.5 Things to Do

Obviously, there is one thing to do in this lab: implement Nachos system calls Exec() and Exit() based on your work in lab 7.

10.6 Report

Although the work of Labs 7 and 8 is not part of formal assessment in this course, I do encourage you to send me your report on it if you manage to complete it successfully.

Assignment 1: Overview and Processes

11.1 Introduction

In this assignment, you are required to answer a number of questions about the concepts of process and thread and how they are implemented in Nachos.

To complete this assignment properly, you must first

complete the study of Modules 1, 2, 3 and 4,

read Chapters 1, 2, 3, 4 and 5 of the textbook,

read all the Nachos source code in the directory ../threads/

complete the Labs 1 and 2.

11.2 Questions

Answer the following questions:

(**Process Concept**)

i. What are processes? What are the possible causes to make a process to change

A. from state **Running** to state **Waiting**?

B. from state **Running** to state **Ready**?

C. from state **Waiting** to state **Ready**?

D. from state **Ready** to state **Running**?

ii. In Nachos, when the current Thread invokes currentThread>Yield() , what will happen? Use the relevant Nachos source code to assist your answer. (Hint: Use gdb to trace the nachos main program in the ../threads/ directory which runs the thread test routine ThreadTest().)

(b) **(Process Scheduling**)

i. What is the scheduler? Why is it needed in operating systems?

ii. Describe how the shortterm scheduler is implemented in Nachos. Use the relevant Nachos source code to assist your answer.

(c) (**Process Creation and Termination**)

i. Describe how a new process is created in UNIX.

ii. Describe how a new thread is created in Nachos. Use the relevant Nachos source code to assist your answer.

iii. Describe how a thread is terminated in Nachos. Use the relevant Nachos source code to assist your answer.

(d) **(Context Switch**)

i. What must be accomplished to complete a context switch from process A to

process B?

ii. Describe how context switch between threads is implemented in Nachos. Use the rele vant Nachos source code to assist your answer.

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 proced ure
80
81      # NEVER REACHED
82      .end ThreadRoot
```

Assignment 2: Synchronization and Monitors

12.1 Introduction

In this assignment, you are required to complete a programming task and answer a few questions about algorithms for monitors.
To complete this assignment properly, you must have

completed the study of Modules 1, 2, 3, 4 and 5.
read Chapters 1, 2, 3, 4, 5 and 6 of the textbook,
read all the Nachos source code in the directory ../threads/
completed Labs 1, 2 and 3.

12.2 Questions

The textbook (pages 220221) provides the algorithm using semaphores for the Hoare style monitor with the additional semaphore next to hold the processes with higher priority than those waiting at the entry semaphore mutex. The algorithm has three components (1) the entry and exit code for each monitor function, (2) the code for Wait() function of condition variable and (3) the code for Signal() function of condition variable.

Write the algorithm using the same semaphores for the Mesa style monitor and explain why your algorithm is correct.

In the above algorithms, we use the additional semaphore next to distinguish the waiting processes which once entered the monitor from other processes waiting at mutex and give the former a higher priority to enter or resume its monitor function. Suppose we do not want to distinguish them and use only one semaphore mutex to hold all the waiting processes, i.e., we do not use next anymore. Write the algorithms for both Hoare style and Mesa style monitors in this context. Use C code or pseudocode to describe your algorithms. Explain why your algorithms are correct.

Implement Hoare style condition variables in Nachos as a new class called condition h. In your answer to this question you should simply quote the C code. In the next question you will need to make use of this new condition variables class.

12.3 Programming Task

After completion of this programming task, you need to write a report about it. You

need to submit both the report and the Nachos source code with your code. The details of submission will be explained later.

```
type Ring = monitor
    var count: integer;
        in: integer;
        out: integer;
        notfull: condition;
        notempty: condition;
    procedure Put(var message: slot)
    begin
        if count = N then notfull.wait();
        /* N is the size of the ring buffer. */
        buffer[in] := message;
        in := (in + 1) mod N;
        count := count + 1;
        notempty.signal();
    end
    procedure Get(var message: slot)
    begin
        if count = 0 then notempty.wait();
        message := buffer[out];
        out := (out + 1) mod N;
        count := count    1;
        notfull.signal();
    end
    begin
        in := 0;
        out := 0;
        count := 0;
end
```

Figure 12.1:Ring Buffer Monitor

In this programming assignment, you are required to implement the producer/consumer problem using monitors with Hoarestyle condition variables for synchronization.

Monitors use condition variables for interprocess synchronization. Currently, Nachos has only the Mesastyle condition variable implemented as class Condition in module synch.cc and synch.h. But, you are not allowed to use this condition variable class.

Instead, you need to implement the Hoarestyle condition variables and then use them to implement the producer/consumer problem with monitors.

12.3.1 Monitor Class Ring

The monitor type is a highlevel synchronization construct. Some programming languages such as Concurrent Pascal allow programmers to declare a monitor type using the syntax shown on page 221 of the textbook. This syntax reminds us that monitor is simply an abstract data type (ADT) with specific synchronization semantics. ObjectOriented languages such C++ support ADT through classes. In Lab 3, we provided a class Ring used in the producer/consumer problem. While C++ does not allow you to declare an arbitrary monitor type of class, you certainly can build a particular one by adding the synchronization mechanism into the class. This is exactly what you are required to do in this assignment.

The details of the implementation of the monitor mechanism are explained in section 7.7 of the textbook. The first thing you need to do is to modify the class Ring from Lab 3 to make it a monitor type of class so that producer and consumer threads can simply call the Put(..) and Get(..) member functions without invoking lowlevel synchronization primitives such as semaphores.

In particular, function Producer in prodcons++.cc should be like this:

```
void
Producer(_int which)
{
        int num;
        slot *message = new slot(0,0);
        for (num = 0; num < N_MESSG ; num++) {
        // the code to prepare the message goes here.
        // ..
        ring>Put(message);
    }
}
```

Likewise, function Consumer in prodcons++.cc should be as follows:

```
void
Consumer(_int which)
```

```
{
        char str[MAXLEN];
        char fname[LINELEN];
        int fd;
        slot *message = new slot(0,0);
        sprintf(fname, "tmp_%d", which);
        printf("file name is %s \n", fname);
        if ( (fd = creat(fname, 0600) ) == 1)
        {
                perror("creat: file create failed");
                exit(1);
        }
        for (; ; ) {
                ring>Get(message);
                sprintf(str,"producer id > %d; Message number > %d;\n",
                        message>thread_id,
                        message>value);
        if ( write(fd, str, strlen(str)) == 1 ) {
                perror("write: write failed");
                exit(1);
        }
    }
}
```

The semaphores declared in prodcons++.cc of lab session 3 are no longer needed, because the synchronization between the producer and consumer threads calling Put() and Get() functions is taken care of by the condition variables in your monitor type of class Ring.

In other words, you need to implement a monitor type of class Ring such that the producer/consumer problem with the Producer and Consumer functions as shown above is still working.

What is a monitor type of class Ring? The class Ring that you used in laboratory session 3 is not a monitor type class, because it does not have monitor synchronization mechanism. Figure 12.1 shows a monitor ring buffer in Concurrent Pascal.

One way to implement monitor type class Ring in our system is to add monitor synchronization control to the class. Here is the new definition of class Ring:

```
class Ring {
   public:
      Ring(int sz);              // Constructor: initialize variables, allocate space.
      Ring();                    // Destructor: deallocate space allocated above.
      void Put(slot *message); // Put a message the next empty slot.
      void Get(slot *message); // Get a message from the next full slot.
      int Full();                // Returns non0 if the ring is full, 0 otherwise.
      int Empty();               // Returns non0 if the ring is empty, 0 otherwise.
   private:
      int size;                  // The size of the ring buffer.
      int in, out;               // Index of Put and Get
      slot *buffer;              // A pointer to an array for the ring buffer.
      int current;               // the current number of full slots in the buffer
      Condition_H *notfull;    // condition variable to wait until not full
      Condition_H *notempty; // condition variable to wait until not empty
      Semaphore *mutex;        //semaphore for the mutual exclusion
      Semaphore *next;         //semaphore for "next" queue
      int next_count;            // the number of threads in "next" queue
};
```

Note that the private data members of this class now include two semaphores: mutex and next, as well as the integer next count to count the threads in the ``next" queue.
There are also two condition variables: notfull and notempty for synchronization as required by the algorithm shown in Figure 12.1.

12.3.2 Condition Variables

Monitors use condition variables for synchronization between the processes accessing the shared data in the monitors.
There are two styles of implementation of condition variables: Hoarestyle and Mesastyle. The Hoare style semantics is described as choice 1 in the discussion on page 218 of the textbook. The Mesastyle corresponds to the second choice in that discussion.
You are not allowed to use the Mesastyle condition variables implemented in Nachos. Instead, you are required to implement the class of condition variable in the Hoarestyle. The Hoarestyle algorithm for Wait(..) and Signal(..) member functions can be found in the textbook. To avoid the confusion between these two styles of condition variables, we use

another name, Condition H, for the Hoarestyle condition variable class. You need to add the definition and implementation of the new class Condition H into synch.h and synch.cc in your working directory for this assignment.

12.3.3 Working directory

The working directory of this assignment should be a separate directory called ass2/ in the code/directory. You need to use the knowledge and skills you learned from Labs 2 and 3 to create this working directory with appropriate makefiles, etc. You also need to copy relevant files from ../threads which you intend to modify. Of course, you need to add some files of your own.

12.4 Submission

You need to submit:

(a) The answers to the questions in Section 12.2.

The report about this programming assignment. The report should tell us everything that you think deserves the credit for your work. In general, it should include the analysis, design, implementation and testing of your programming task. It should be selfcontained and in clude all the necessary details to convince us that your design, implementation and testing are correct. There is a sample report for a programming task in Chapter 14 of this Intro ductory Book, but you do not have to follow the format of it. You may quote some of your source code in writing, but the source code listing itself is not acceptable as the report. Do not submit the source code listing as the report. The testing part should also include the evidence of the real testing processes. The best way is to copy and include the screen output of the testing. You can use the cutandpaste of the X window to do that. Another way is to use a shell buffer in Emacs. Also, there is a command script available which will save everything which takes place inside an xterm to a file. Report writing is very important. Poor reporting makes it difficult for us to assess your programming work. If you do not submit the report or submit a poor report, you may get zero marks even though the program you submit works correctly, because we cannot be convinced that the program submitted is your genuine work.

a floppy disk which contains the tar file of your complete Nachos source code including the code in your working directory ass2/ for this assignment. The details of how to submit a floppy disk for Nachos can be found in Chapter 4.

Assignment 3: File System Interface and Implementation

13.1 Introduction

This assignment comprises two parts: the work of Laboratory 5 and a programming task. In the first part, you need to report your work of Lab 5 by answering related questions given in Section 13.2 below. The last part is similar to the second part which focuses on a programming task described in Section 13.3. You need to submit your Nachos system with this assignment after you finish the work of Lab 5 and the programming task specified in 13.3.

13.2 Work of Lab 5

What modules of the Nachos file system need to be changed for the purpose of this laboratory work? Describe in detail (with relevant source code) the changes you made to the modules modified. State the reasons that you chose to make these changes.

Describe your test runs and show their results. Do you think that your test runs are complete to cover all the aspects of the new file system and if so, why?

Show the dump of the Nachos file system (by nachos D) after you execute the following command:

i. rm DISK
ii. nachos f
iii. nachos cp test/medium strange
iv. nachos hap test/small strange
v. nachos ap Makefile strange

Report other significant discoveries or tricks that helped you finish the work of this laboratory session, if any.

13.3 The Programming Task

The programming task of this assignment is based on your work in Lab 5. In Lab 5, you have extended the Nachos file system to allow extendable files.
You may need to use UNIX od command extensively when debugging the programs for this pro gramming task. Let us look at this command in more detail first.

13.3.1 UNIX od c Command

Let us move back to ../lab5/ and start with a fresh DISK by rm DISK and nachos f . Then

execute nachos cp test/small small .

Execute UNIX command od c DISK . You should have the following octal dump for file DISK:

```
0000000    211   g   E 200  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
0000020    020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0   H 005  \b
0000040      (  \0  \0  \0 234   I 005  \b  \b   G 021   @   G 005  \b  \0
0000060      H 005  \b   M 005  \b 020  \0  \0  \0   F 021   @
0000100      F 021   @ 200 001  \0  \0 020  \0  \0  \0  \0  \0  \0  \0
0000120     \0  \0  \0  \0   H 005  \b   8  \0  \0  \0 034   7 005  \b
0000140      T   I 005  \b  \0  \0  \0  \0   (   I 005  \b  \0   M 005  \b
0000160    020  \0  \0  \0   F 021   @   F 021   @   (  \0  \0  \0
0000200    020  \0  \0  \0  \0  \0  \0 002  \0  \0  \0 003  \0  \0  \0
0000220    004  \0  \0  \0 200   I 005  \b   8  \0  \0  \0 034   7 005  \b
0000240    030   G 021   @  \0  \0  \0  \0   (  \0  \0  \0  \b   G 021   @
0000260    004   J 005  \b   F 005  \b  \0 200   I 005  \b 214   M 005  \b
0000300    020  \0  \0  \0   F 021   @   F 021   @ 200 003  \0  \0
0000320    020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0   I 005  \b
0000340      (  \0  \0  \0 004   J 005  \b  \b   G 021   @   E 005  \b  \0
0000360      I 005  \b   M 005  \b 020  \0  \0  \0   F 021   @
0000400      F 021   @ 177  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000420     \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000600     \0  \0  \0  \0 001   F 021   @ 005  \0  \0  \0   s   m   a   l
0000620      l  \0  \0  \0  \0   G 005  \b  \0   H 005  \b 030  \0  \0  \0
0000640    030  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0   G 005  \b
0000660     \0  \0  \0  \0  \0  \0  \0  \0   0  \0  \0  \0 020   G 021   @
0000700     \0   G 021   @ 020   G 005  \b   X 023  \0  \0  \0   J 021   @
0000720    030  \0  \0  \0  \0   F 021   @   F 021   @ 200   G 005  \b
0000740    030  \0  \0  \0   `  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000760    002  \0  \0  \0 240   F 005  \b  \0  \0  \0  \0  \0  \0  \0  \0
0001000    030  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0   D 005  \b
0001020     \0 021  \0  \0   I 021   @   H  \0  \0  \0   (   G 021   @
0001040      (   G 021   @  \0   F 005  \b  \0  \0  \0  \0  \0  \0  \0  \0
0001060      0  \0  \0  \0 020   G 021   @  \0   G 021   @ 220   D 005  \b
0001100      I 021   @   H   H 005  \b 030  \0  \0  \0  \0  \0  \0  \0
0001120     \0  \0  \0  \0 210   2 005  \b   2 005  \b 220   2 005  \b
0001140      P  \0  \0  \0   0   G 021   @   0   G 021   @ 207
```

```
0001160     \a \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0001200     \0 \0 \0 \0 T \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
0001220     \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0001400     \0 \0 \0 \0 s m a l l f i l e s
0001420     m a l l f i l e s m a l l
0001440     f i l e \n s m a l l f i l e
0001460     s m a l l f i l e s m a l l
0001500     f i l e \n * * * e n d o f
0001520     f i l e * * * \n F 021 @ 200 G 005 \b
0001540     030 \0 \0 \0 ` \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0001560     002 \0 \0 \0 240 F 005 \b \0 \0 \0 \0 \0 \0 \0 \0
0001600     030 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0001620     \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0400000     \0 \0 \0 \0
0400004
```

We can verify the location of the data block of file small. According to the file system dump (by nachos -D), the data block (the only one) of the file starts at the beginning of sector 6. Each sector has 128 bytes. 128 equals to 200_8 in octal. Since $200_8 * 6_8 = 1400_8$, the starting address of sector 6 should be 1400_8 (in octal). Remember that we have put a magic integer number in the beginning of the hard disk, sector 0 actually starts from address 4_8. Therefore, sector 6 should start from offset 1404 8 . You can verify that the data block of file small does start in the right position of the hard disk. The size of the file is $84 = 124_8$. The address of the last byte of the file should be
$1404_8 + 124_8 - 1_8 = 1527_8$. We can see that the file does end at that offset and the bytes starting from 1530_8 of the sector are not used.

13.3.2 Objectives

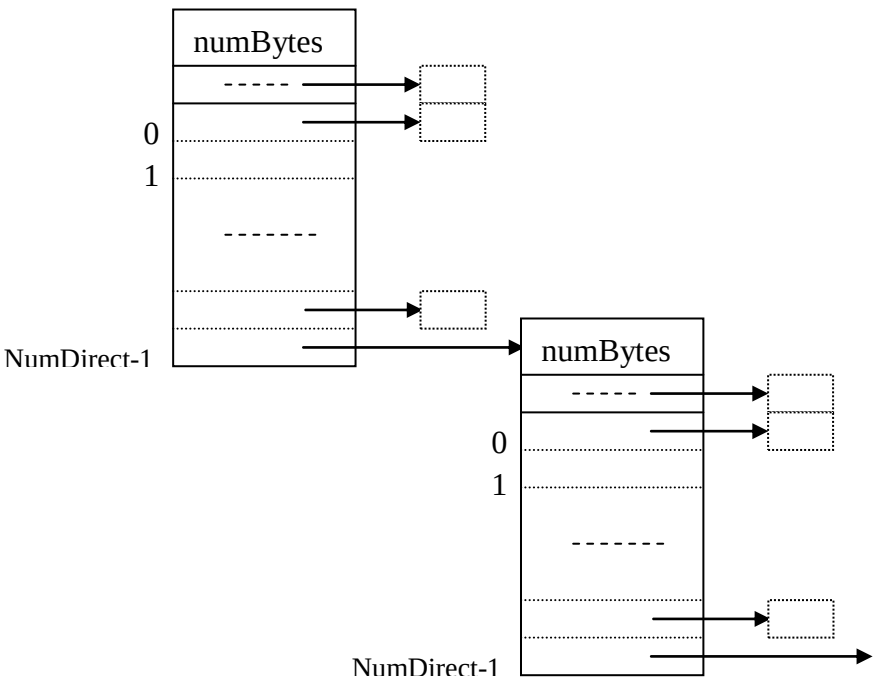
The Nachos file system uses single-level index allocation method for data allocation of files. In the current design, the file header (inode) of a file can point to only NumDirect (its value is 30) data sectors. Therefore, the maximum size of a file is NumDirect*SectorSize, which evaluates to 3720 bytes.

In Lab 5, you have made Nachos files extendable. That is, one can increase the size of a file by writing more data to it. But, the maximum size of a file is still the same. You simply cannot put more data than 3720 bytes to a Nachos file.

In this programming task, you are required to further change the Nachos file system to

increase the maximum file size with the twolevel index allocation method similar to UNIX file system.

In particular, you use the last entry of array dataSectors[] to store the sector number of the secondary indirect inode when necessary. When the file is large, the inode structure is as follows:



Note that the maximum file size becomes
(NumDirect-1)*SectorSize + NumDirect*SectorSize . In the original filehdr.h, the data members of class FileHeader are defined as follows:

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

where NumDirect is defined in the same file as:

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

The reason for this is that we want to fit a filehdr (inode) into exactly one data sector. Because there are two integer variables, numBytes and numSectors, in the file header, the above definition for numDirect is correct.
The new definition of the data members of FileHeader should be as follows:

```
private:
int numBytes;  // Number of bytes in the file
int numSectors; // Number of data sectors in the file
FileHeader *indirect; // pointer to the indirect header
int dataSectors[NumDirect]; // Disk sector numbers for each data
// block in the file
```

The reason is that we need a pointer to the indirect file header in the memory when processing the operation of class FileHeader. Therefore, the definition of numDirect should change to

```
#define NumDirect ((SectorSize - 3 * sizeof(int)) / sizeof(int))
```

accordingly.
The actual size of a file should be changed dynamically: if the size is small, the file system does not need to allocate the indirect inode. The indirect inode exists only if the size exceeds ((NumDirect-1) * SectorSize) .

13.3.3 Working Directory

The working directory of this programming task is ../ass3/. In this directory there are only Makefile and Makefile.local. Before you start, you need to copy all the .cc and .h files from your ../lab5/. That is, you start from your Nachos file system completed in Lab 5.
Your task is to further modify your Nachos file system to meet the requirement described above.

13.3.4 Encapsulation

The changes that you need to make in this programming are very isolated. You do not need to change any other classes except class FileHeader. In other words, all the changes you need are encapsulated within the functions of class FileHeader.

13.3.5 Testing

In this section, I explain the test results of my Nachos file system completed for this programming task on Sun Sparc platform. The purpose is help you understand how to test

your new system. You should have similar test results on your LINUX/PC platform. In your ../ass2/, you can find a test subdirectory which contain the files shown as follows:

```
ptang@titus: ls l test
total 16
rwrr   1    ptang   337     Oct 2 17:15      big
rwrr   1    ptang   3804    Oct 2 17:13      huge
rwrr   1    ptang   3573    Oct 2 17:12      huge1
rwrr   1    ptang   3678    Oct 2 17:13      huge2
rwrr   1    ptang   169     Oct 2 17:14      medium
rwrr   1    ptang   1       Sep 28 18:07     onebyte
rwrr   1    ptang   90      Oct 2 17:14      small
ptang@titus:
```

In the new Nachos file system, the total size of the data blocks pointed by the direct i-node is 28 sectors or 3584 bytes. If the size of the file exceeds 3584 bytes, it should start to use indirect i-node for the extra data blocks. In the following test, we first copy test/huge1 to Nachos file huge1.
Then we extend it by appending test/small and test/medium.

After nachos -f and nachos -cp test/huge1 huge1 , the following file system dump shows that Nachos file huge1 takes contiguous blocks from sector 6 through sector 33 as follows:

```
ptang@titus: nachos D
Bit map file header:
FileHeader contents. File size: 128. File blocks:
2
File contents:
\ff\ff\ff\ff\0\0\0\3\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Directory file header:
FileHeader contents. File size: 200. File blocks:
3 4
File contents:
\0\0\0\1\0\0\0\5huge1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
```

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
Directory contents:
Name: huge1, Sector: 5
FileHeader contents. File size: 3573. File blocks:
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
File contents:
This is a huge file.\aThis is a huge file.\aThis is a huge file.\aThis is a huge
file.\aThis is a huge file.\aThis is a huge file.\aTh
......

Execute nachos -ap test/small huge1 . The file system dump should show that the size of the file is increased to 3663 bytes and the last sector of the file is sector 35:

Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
Directory contents:
Name: huge1, Sector: 5
FileHeader contents. File size: 3663. File blocks:
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 35
File contents:
.....

Note that bitmap shows that sector 34 has been allocated. It is allocated for the indirect inode of the file.

Execute nachos -ap test/medium huge1 . The file system system dump should show that the file occupies 28+2=30 sectors now and the size is 3832 bytes. The following od dump from sector 33 (starting from 10204) to sector 36 (ending at 11203) shows that the file con tinues on sectors 35 and 36, skipping sector 34 which is used for the indirect inode.

0010200 a h u g e f i l e . \n T h i
0010220 s i s a h u g e f i l e
0010240 . \n T h i s i s a h u g e
0010260 f i l e . \n T h i s i s a

```
0010300 h u g e f i l e . \n T h i s
0010320 i s a h u g e f i l e .
0010340 \n * * * e n d o f h u g e 1
0010360 f i l e * * * \n s m a l l f
0010400 i l e \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0010420 \0 \0 \0 # \0 \0 \0 $ \0 \0 \0 \0 \0 \0 \0 \0
0010440 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0010600 \0 \0 \0 \0 s m a l l f i l e s
0010620 m a l l f i l e \n s m a l l
0010640 f i l e s m a l l f i l e
0010660 s m a l l f i l e \n * * * e n
0010700 d o f s m a l l f i l e *
0010720 * * \n m e d i u m f i l e m
0010740 e d i u m f i l e m e d i u
0010760 m f i l e \n m e d i u m f i
0011000 l e m e d i u m f i l e m
0011020 e d i u m f i l e \n m e d i u
0011040 m f i l e m e d i u m f i
0011060 l e m e d i u m f i l e \n m
0011100 e d i u m f i l e m e d i u
0011120 m f i l e m e d i u m f i
0011140 l e \n * * * e n d o f m e d
0011160 i u m f i l e * * * \n \0 \0 \0 \0
0011200 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
```

13.3.6 Change TransferSize

The macro TransferSize defined in fstest.cc determines the number of bytes to be appended at a time. Its current value is defined to be 10. Your program should also work correctly even when you change its value to 100, 250, 400, 1000, 2000.

13.3.7 Questions to Assist Your Report

The following questions can be used to assist you to write the report for this programming assignment:

In this programming task, you only need to change class FileHeader. Describe in detail what member functions are changed or what member functions are added. Describe in detail the algorithm to dynamically increase the size of a file.

Show the relevant source code for these changes.

Describe your testing with the corresponding result. Show why your tests are complete.

In the indirect in-ode, the last entry of array dataSectors is used for an ordinary data sector (last sector) of the file. We could use it to point to another indirect inode just as we did in the director inode. This way, we could allow the file size to grow without limit. Describe the further changes to the system and the appropriate algorithm for this new feature.

13.4 What to submit?

This is to summarize what you need to submit. You need to submit:

the paper work of the answers to the questions listed in Section 13.2.

the report of this programming assignment.

a floppy disk which contains the tar file nachos.tar.gz of your whole directory of nachos. This tar file is supposed to contain all your programs in ../lab5/ and ../ass3 subdirecto ries as well as the original Nachos system.