The University of Southern Queensland
Faculty of Sciences

Introductory Book
(2002)

CSC2404/66204: Operating Systems

Version 2.5

CSC2404/66204: Operating Systems

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

Contents

I Introduction **1**

1 Course Specification **2**

2 Overview of the Course **5**

3 Study Chart **14**

4 How to Submit Programming Assignments **15**

5 Sample Exam, with answers **17**

II Laboratory **9**

6 Laboratory 1: Installation of Nachos System **10**

7 Laboratory 2: Makefiles of Nachos **16**

8 Laboratory 3: Synchronization Using Semaphores **26**

9 Laboratory 4: Nachos File System **29**

10 Laboratory 5: Extendable Files **34**

11 Laboratory 6: Nachos User Programs and System Calls **40**

12 Laboratory 7: Extension of AddrSpace **43**

13 Laboratory 8: System Calls Exec() and Exit() **45**

III Assignments **49**

14 Sample of Assignment Submission **50**

15 Assignment 1: Overview and Processes **64**

16 Assignment 2: Synchronization and Monitors **66**

17 Assignment 3: File System Interface and Implementation **72**

Part I
Introduction

This part includes introductory information about this course. At the end of this part there is a sample exam, with answers and explanations of how marks will be awarded.

Chapter 1

Course Specification

Please refer to the Course web page, at http://www.sci.usq.edu.au/courses/CSC2404 or the USQ Online handbook at http://www.usq.edu.au/handbook.

Chapter 2

Overview of the Course

2.1 Prerequisites

The prerequisites of this courses are

- Computer Engineering I (ELE1301/70335) and
- Algorithms and Data Structures (CSC2401/66201) which has Advanced Procedural Programming (CSC2400/66121) and Discrete Mathematics for Computing (MAT1101/64611) 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**.

2.2 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**.

2.3 Order Departmental CDROM Now!

If you decide to go ahead to study this course, **you must order our departmental CDROM set from USQ Bookshop now**! The departmental CDROM comes in two sets, Set 1 containing course material and additional software for Linux and Windows, and Set 2 containing a distribution of Linux.

The CDROM Set 1 is essential for completing all the laboratory work and the two programming assignments of this course. Set 2 is only necessary if you need a distribution CD set for installing Linux on your computer.

2.4 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.

2.5 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.

2.5.1 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 0-471-41743-2.

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.

2.5.2 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.

2.6 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.

2.7 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 UNIX-like 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
  - in the CDROM set published by the Department of Mathematics and Computing of USQ. This CDROM set also includes the `gcc` MIPS cross-compiler used by the course.

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.

2.8 LINUX/PC Home Laboratory

You need to install Linux operating system on your home PC before experimenting with the Nachos system. You should purchase a CDROM set called "Mathematics and Computing CDROM" published by our department from the USQ Bookshop. One of the CDROMs contains the RedHat LINUX distribution 7.2 and the other all the teaching software of computing courses of this semester including CSC2404/66204.

To install LINUX operating system, follow the instructions in the booklet that comes with the CDROM set.

There are many documents in greater detail about Linux and Unix operating system in general. After you install Linux operating system, read the further documents in the CDROM set.

The Department provides limited support for Linux installation for external students. Any questions or problems related to Linux installation should be directed to

- phone: (07) 4631 1513 [(61-7) 4631 1513]
- email: linux@usq.edu.au
- Fax: (07) 4631 1775 [(61-7) 4631 1775]

The Departmental Internet home page for LINUX support is
http://www.sci.usq.edu.au/cdrom/.

2.9 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 3-hour closed-book 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	Description	Weight
1	15	16AUG02	Assignment 1	15
2	15	06SEP02	Assignment 2	15
3	15	18OCT02	Assignment 3	15
4	100	end of Semester	Final Exam (3 Hours)	55

2.10 Course Home Page and Contacting Lecturer

There is a web home page for this course which contains some last-minute information. The web address of the page is

http://www.sci.usq.edu.au/courses/CSC2404/

Visit this home page first if you have any questions about the course.

If the home page still does not solve your problems, you can contact USQ Outreach or directly email to lec66204@usq.edu.au or outreach@usq.edu.au.

2.11 Student Enquiries

You should carefully read the information provided in your Student Guide concerning contact details and support services.

If you have Internet access, USQAssist is the most efficient method for requesting support assistance. It is a web self service facility for all students to:

find answers to common questions;
ask a question; and/or
track the progress of a question.

By typing a keyword in the search field, you can find answers to many of the questions frequently asked by students, including course troubleshooters. To access USQAssist, go to http://usqconnect.usq.edu.au/usqassist or click on the 'Help' option at USQConnect.

You can also ask for support by telephone or facsimile.

For all Australian Citizens and all students enrolled through the Office of Continuing and Professional Education

All administrative queries should be directed to the Distance Education Centre (DEC) Outreach Services or your Regional Liaison Officer. Outreach Services can be contacted as follows:

Telephone:	07 46312285
Fax:	07 46361049
Web Form:	http://usqconnect.usq.edu.au/usqassist
Email	outreach@usq.edu.au

2.11.1 For International Students

International students should contact their Local Support Office for further assistance. If there is no Local Support Office in your country you should contact the International Office at USQ as follows:

Telephone:	61 7 46312362
Fax:	61 7 46362211
	61 7 46359225
Web Form:	http://usqconnect.usq.edu.au/usqassist
Email:	iosupport@usq.edu.au

2.11.2 USQConnect

USQConnect is a computer system which enables you to access information, services and program resources on the Internet environment - the World Wide Web (WWW). To find out more about this dynamic environment we recommend you look at the following URL regularly:

http://usqconnect.usq.edu.au

Services that are provided via USQConnect include:

- access to electronic course materials (where appropriate),
- access to up-to-date library catalogues, electronic journals and articles, and text databases,
- secure access to your enrolment details, course assignment and end of semester results,
- Faculty information on departments, programs, policies, and staff details,
- access to the USQAssist knowledge base - a list of common questions asked by students,
- Outreach Electronic Noticeboard for external students including Residential School and telephone tutorial timetables, learning circles and other information,
- communication facilities - email and discussion groups,
- opportunity to establish your home page.

2.12 Course Evaluation (External Students Only)

The University of Southern Queensland is committed to continuous improvement, and seeks your input to that process through your participation in our course evaluation process. Please complete and return the questionnaire 'Student Feedback on External Courses' included later in this introductory book.

Your response will be processed so that, unless you wish otherwise, the course Examiner will not be aware of your identity. Please help us to help our students by providing feedback on your experiences in this course.

When to Return the Questionnaire

Please return the questionnaire before the end of this semester's examination period.

Where to Send the Questionnaire

1. Insert the completed questionnaire in an envelope, seal and address envelope as follows:

The Course Evaluation Co-ordinator
Information Technology Services
University of Southern Queensland
Toowoomba 4350

2. The envelope may be posted directly to the above address

   OR

   attached to the outside of your last assignment for this course and then posted to DEC.

Chapter 3

Study Chart

Week	Modules	Laboratory	Assignment	Assessment
1	1, 2 and 3	Lab 1	Ass 1	
2	4	Lab 2		
3				
4	5	Lab 3		**Assignment 1**, Due 16 Aug 02
5			Ass 2	
6	9	Lab 4		
7	10	Lab 5		**Assignment 2**, Due 6 Sep 02
8			Ass 3	
9	6	Lab 6		
10	break			
11				
12	6	Lab 7	Ass 3	
13	7			**Assignment 3**, Due 18 Oct 02
14	8	Lab 8		
15	Review			

Chapter 4

How to Submit Programming Assignments

4.1 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 re-format 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.

4.2 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 **nachos-3.4** 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
```

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

Chapter 5

Sample Exam, with answers

UNIVERSITY OF SOUTHERN QUEENSLAND

FACULTY OF SCIENCES

Unit No: 66204 **Unit Name:** Operating Systems

Assessment No: 4 **Internal** [x] **External** [x]

This examination carries 55% of the total assessment for this unit.

Examiner: Ron Addie **Moderator:** Yanchun Zhang

Examination Period: November 2001

Time Allowed: **Perusal –** Ten (10) minutes
Working – Three (3) hours

Special Instructions:

This is an closed examination. Calculators may not be used. Programmable calculators are not allowed.
Students may keep this examination paper.

During perusal time, students are not permitted to write in the examination booklet or on any other material submitted for assessment.
No examination booklet, used or otherwise, is to be taken from the examination room. Obey the regulations on the Examination Booklet, and ensure that you place your name and student number in the places provided.

Students are to complete all 5 *five* questions.

Question 1.

A modern operating system usually includes the facility for several processes to be running "simultaneously", even when the computer has only one CPU. In this connection:

(a) List all the important situations in which control switches from one process to another.

(b) Describe all the processes which might be involved in the activity of transferring data from a disk to a user process.

(c) Describe in general terms how the operating system selects which process should gain access to the CPU next.

(d) Consider the code-fragments, one from the assembly language routine `switch` shown in Figure 5.1, and another from the method `Scheduler::Run`, shown in Figure 5.2. Where, precisely, in these fragments of code, does control switch between one thread and another?

(e) Explain the sequence of actions which leads to a thread being deleted.

Total: 11 marks

Answers to Question 1

Allocate 2 marks for each part of this question, together with one mark which is awarded if the student displays a satisfactory understanding of how processes work in an operating system.

(a) Full marks will be gained for listing at least 2 of the following, or equivalent:

1. When a process requests an operating system service which requires waiting, e.g. reading from a disk, writing to a network card, etc.;
2. When the system clock expires after the process with control has maintained control for the entire duration of the most recent clock interval;
3. when an interrupt occurs, indicating, for example, that some other process is ready to take control of the processor;
4. when a process invokes the semaphore wait (P()) method, and the resource is not available (the sempaphore value is zero or lower).

(b) Full marks will be gained for listing at least 2 of the following, or equivalent:

1. The user process, which initiated the transfer;
2. the system kernel, which receives the request from the user process;
3. the driver process, which manages passing requests to the hardware and from the hardware to the user process.

(c) The operating system maintains a list of processes which are "ready to run", generally known as the *ready queue*. Processes are removed from this queue in accordance with a discipline which in some sense optimizes the performance of the operating system. Factors which might be taken into account include: priority, expected duration of the next active period, memory usage, whether the process is in memory or not (in fact, if the process is not in memory, the kernel will have to swap it in as a stage before the process can be run), and so on.

```
#ifdef HOST_i386

...


/*  void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
*/
        .comm   _eax_save,4

        .globl  _SWITCH
_SWITCH:
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into eax
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack into ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage

        movl    8(%esp),%eax            # move pointer to t2 into eax

        movl    _EAX(%eax),%ebx         # get new value for eax into ebx
        movl    %ebx,_eax_save          # save it
        movl    _EBX(%eax),%ebx         # retore old registers
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp         # restore stack pointer
        movl    _PC(%eax),%eax          # restore return address into eax
        movl    %eax,4(%esp)            # copy over the ret address on the stack
        movl    _eax_save,%eax

        ret

#endif // HOST_i386
```

Figure 5.1: A Fragment from switch.s

```
//----------------------------------------------------------------------
// Scheduler::Run
//      Dispatch the CPU to nextThread.  Save the state of the old thread,
//      and load the state of the new thread, by calling the machine
//      dependent context switch routine, SWITCH.
//
//      Note: we assume the state of the previously running thread has
//      already been changed from running to blocked or ready (depending).
// Side effect:
//      The global variable currentThread becomes nextThread.
//
//      "nextThread" is the thread to be put into the CPU.
//----------------------------------------------------------------------

void
Scheduler::Run (Thread *nextThread)
{
    Thread *oldThread = currentThread;

#ifdef USER_PROGRAM                     // ignore until running user programs
    if (currentThread->space != NULL) {  // if this thread is a user program,
        currentThread->SaveUserState(); // save the user's CPU registers
        currentThread->space->SaveState();
    }
#endif

    oldThread->CheckOverflow();             // check if the old thread
                                            // had an undetected stack overflow
    currentThread = nextThread;             // switch to the next thread
    currentThread->setStatus(RUNNING);      // nextThread is now running

    DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",
          oldThread->getName(), nextThread->getName());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    SWITCH(oldThread, nextThread);

    DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());

    // If the old thread gave up the processor because it was finishing,
    // we need to delete its carcass.  Note we cannot delete the thread
    // before now (for example, in Thread::Finish()), because up to this
    // point, we were still running on the old thread's stack!
    if (threadToBeDestroyed != NULL)
        { delete threadToBeDestroyed;
        threadToBeDestroyed = NULL;
    }

#ifdef USER_PROGRAM
    if (currentThread->space != NULL) {         // if there is an address space
        currentThread->RestoreUserState();     // to restore, do it.
        currentThread->space->RestoreState();
```

```
}
```

(d) control is transferred when the routine SWITCH returns control to the procedure which called it. Any answer which suggests that control is transferred inside the routine SWITCH should get at least half marks.

(e) The thread to be deleted declares that it should be deleted, by setting the global variable, threadToBeDestroyed to point to itself. Next, this thread transfers control to another thread. Finally, after control has been transferred, the new thread removes the one to be deleted, inside the Run method.

Question 2.

(a) A fragment from the nachos code for a *disk* is shown in Figure 5.3. Explain the role of this particular class in the nachos system.

(b) What methods of the disk class are used by the rest of the nachos code in the normal operation of the disk?

(c) Explain the role (purpose) of an *inode* and how it is used in a file system to implement the file construct in an operating system.

(d) What methods could be used to keep track of the free space on a disk? You may choose to explain one method very carefully, or more than one method.

(e) What techniques are used to ensure that a file system is not permanently damaged when a software or hardware failure causes a computer to cease normal operation during a disk IO operation.

Total: 11 marks

Answer to Question 2

Allocate 2 marks for each of the following questions plus one mark if the student displays a reasonable understanding of how file systems work.

(a) this class emulates (simulates) the physical disk of tt nachos. This is one aspect of tt nachos which is not like a real operating system. In a real operating system, this bit would not be simulated – it would be hardware.

(b) ReadRequest and WriteRequest are used by other parts of the nachos system to simulate requests to the hardware to read, and to write data, and the routine HandleInterrupt is regularly used when reads or writes finish. The constructor is only called once, when the system starts, unbless there is more than one disk.

(c) An *inode* is used to record *where* on the disk the bytes of the actual file are stored. An inode is basically a table, with each entry corresponding to one block, of about 512 bytes, (although the precise size depends on the disk). The entry tells the user, and the computer, where each of these collection of bytes is stored.

(d) The basic method is a table of bits, one bit per block, each of which says whether the specific block, referred to (implicitly) by that bit is in use or not. Another method would be to have a linked list of free blocks, or contiguous sequences of blocks.

(e) Two basic techniques are used: duplication of critcal information; and secondly, in some situations, information is written to the disk as soon as possible, rather than caching this information for collection later.

Question 3.

(a) Explain how interrupts are implemented in nachos.

(b) How is it possible to simulate interrupts successfully in a program, such as `nachos` which runs as a user process on a Unix computer.

(c) Explain how the loading of a program is simulated in the nachos system.

Total: 11 marks

Answer to Question 3

Give 4 marks for the first two parts and 3 for the last part.

(a) Basically, instructions are simulated one-by-one, advancing the clock at the end of each instruction. Also, between each instruction, the list of pending interrupts is checked to see if an interrupt is due, and if it is, it is declared to *happen*. The *scheduling* of interrupts is handled by storing interrupts in this list of pending interrupts, and every time an interrupt is scheduled, it is placed in this list *in order*. Interrupts are scheduled to happen by nachos code such as the disk subsystem, which is simulating hardware, for example.

(b) Because the hardware is actually only simulated, the interrutpts are actually only simulated also, and so it does not take system privileges to execute hachos code.

(c) The program is read from disk and stored in memory, in an array which simulates the nachos main memory. When this occurs, the disk file is subdivided into memory blocks (that is not the correct term). In addition, certain instructions in the program are expanded into a whole collection of bits – for example, there are instructions which represent an array which is initialised to zero.

Question 4.

(a) Explain how system calls are typically implemented in an operating system.

(b) Give examples of system calls which return immediately and system calls which cause control to be switched to a different process. In each case, explain *why* control is transferred or not transferred, unless you feel that no explanation is necessary. Provide at least four examples and at least one example of each sort.

Total: 11 marks

```
// disk.h
//      Data structures to emulate a physical disk.  A physical disk
//      can accept (one at a time) requests to read/write a disk sector;
//      when the request is satisfied, the CPU gets an interrupt, and
//      the next request can be sent to the disk.
//
//      Disk contents are preserved across machine crashes, but if
//      a file system operation (eg, create a file) is in progress when the
//      system shuts down, the file system may be corrupted.
//
//  DO NOT CHANGE -- part of the machine emulation
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#ifndef DISK_H
#define DISK_H

#include "copyright.h"
#include "utility.h"

// The following class defines a physical disk I/O device.  The disk
...
#define SectorSize              128     // number of bytes per disk sector
#define SectorsPerTrack         32      // number of sectors per disk track
#define NumTracks               32      // number of tracks per disk
#define NumSectors              (SectorsPerTrack * NumTracks)
                                        // total # of sectors per disk

class Disk {
  public:
    Disk(char* name, VoidFunctionPtr callWhenDone, _int callArg);
                                        // Create a simulated disk.
                                        // Invoke (*callWhenDone)(callArg)
                                        // every time a request completes.
    ~Disk();                            // Deallocate the disk.

    void ReadRequest(int sectorNumber, char* data);
                                        // Read/write an single disk sector.
                                        // These routines send a request to
                                        // the disk and return immediately.
                                        // Only one request allowed at a time!
    void WriteRequest(int sectorNumber, char* data);

    void HandleInterrupt();             // Interrupt handler, invoked when
                                        // disk request finishes.

    int ComputeLatency(int newSector, bool writing);
                                        // Return how long a request to
                                        // newSector will take:
                                        // (seek + rotational delay + transfer)

  private:
...
};
```

Answer to Question 4

Give 6 marks for the first part and 5 for the last part.

(a) Normally there is an instruction of the computer which is used to call a system routine. If this is not the case, it might be that there is an instruction which causes an *exception*, or a *software interrupt*, which are much the same thing. In any of these cases, the interrupt to be executed is specified as the first parameter, ie first entry on the stack. This interrupt number is used to index into the interrupt vector, and in the case of system routines to index into the table of system routines. This system routine is then invoked. Any remaining parameters on the stack will be consumed.

(b) A system routine which requests a `read` operation, from a disk, will cause control to be transferred, until such time as the read is complete. The same is true of network read or write: control will be lost by the calling method. Another more benign exampe of a system routine is a routine to gnerate the system time, or to remember the `ID` of another user.

Question 5.

(a) Suppose that an unknown number of threads need to make use of and alter an operating system parameter, `freebobs`, denoting the number of free "bobs". If this variable is updated without any concern for inappropriate interaction between different processes, situations will arise where it is set to the wrong value. Explain how one or more semaphores could be used to overcome this difficulty.

(b) Describe the methods or procedures associated with a semaphore and the purpose of each method.

(c) Describe, in outline, how you might go about implementing a semaphore.

Answer to Question 5

Give 4 marks for the first two parts and 3 for the last part.

(a) We could allocate a semaphore, eg `mutex` such that this semaphore must be *held* before the number of `freebobs` can be changed. Thus, in any method or routine of the code, we would put `mutex.P()` before the code which accesses the freebobs quantity and we would put put `mutex.V()` afterwards.

(b) The methods, all of them, are `P()` and `V()`. The first of these will cause the calling process to wait, if the semaphore `value` is zero or less. The second will cause the semaphore variable to increase by one, and, if there is anyone wiating on the imlied list, the process at the head of this list will be woken up.

(c) Here is an outline. It is assumed that these routines will be used only in a single-processor machine.

```
void semaphore::P() {
    inhibit interrupts;
    if (the variable is > 0) {
        decrease this variable by one;
        open interrupts;
        return;
    } else {
        place this process in the semaphore queue
        and transfer control to another process
        by means of SWITCH;
        open interrupts;
        return;
    }
}
void semaphore::V() {
    inhibit interrupts
        if (the variable is < 1)
        {
            increase the variable by one;
            SWITCH control to the process
            at the head of the queue and
            open interrupts;
        } else {
            increase the variable by one;
        }
    open interrupts;
}
```

Variations of the above are, of course, acceptable, so long as they provide the same functionality. In particular, it doesn't greatly matter whether the counter keeps on decreasing or not as the number of waiting processes increases.

Total: 11 marks

Part II
Laboratory

This part includes the laboratory exercises you are required to complete in your LINUX system. The purpose is to prepare you for the programming tasks in the assignments and have a deep understanding of how real operating systems work.

Chapter 6

Laboratory 1: Installation of Nachos System

6.1 Purpose

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.

We assume that you already have installed the Red Hat LINUX 7.2 from our department CDROM set. If you have not done so, read Section 2.8 about how to install it.

6.2 Installation of Nachos and gcc Cross-Compiler

You need to install LINUX operating system on your PC before installing the Nachos system. The instructions to install LINUX operating system on PC are described in Section 2.8 of this introductory book.

The simplest way to install Nachos and gcc MIPS cross-compiler from the USQ Mathematics & 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/USQ/CSC2404/ or similar. You will see the following files in the directory (or something similar):

```
INSTALL                          gcc-2.8.1-mips.tar.gz
binutils-2.9.1.0.23-6.i386.rpm   index.html
binutils-2.9.5.0.22-6.i386.rpm   nachos-3.4-USQ01.tgz
```

nachos-3.4-USQ01.tgz is the file of the Nachos system package.
gcc-2.8.1-mips.tar.gz is the file of gcc MIPS cross-compiler.

Installation of Nachos

Follow the following steps to install Nachos:

(a) cd

(b) mkdir CSC2404

(c) cd CSC2404

(d) cp /cdrom/USQ/CSC2404/nachos-3.4-USQ01.tgz .

(e) tar xzvf nachos-3.4-USQ01.tgz

(f) rm nachos-3.4-USQ01.tgz

After these steps, you should have a directory named CSC2404 in your home directory. Directory CSC2404 should contain a directory named nachos-3.4 which is the Nachos package.

Installation of gcc MIPS cross-compiler

The gcc MIPS cross-compiler 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 cross-compiler:

(a) su -

(b) cd /usr/local

(c) cp /cdrom/USQ/CSC2404/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 cross-compiler.

6.3 Testing Nachos

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

(a) Move to directory ~/CSC2404/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/  lab7-8/   network/  threads/  vm/
```

Here **lab2/**, **lab3/**, **lab5/**, **lab7-8/** 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/unknown-i386-linux/bin/nachos <<<
g++ arch/unknown-i386-linux/objects/main.o ----------------------
----------------------------------------------------------------
----------------------------------------------------------------
ln -sf arch/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/unknown-i386-linux/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...
```

6.4 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 object-oriented programming (66203) before. C++ is a complicated object-oriented 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 `M-x gdb`.
  - in the command line of emacs, respond the full path name of program "stack", `~/CSC2404/nachos-3.4/c++` 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/nachos-3.4/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 (i386-redhat-linux), 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: /home/staff/ptang/units/204/98/linux/nachos-3.4/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.

6.5 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.

6.6 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**.

6.7 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 6.2)
- compile and test the Nachos system installed (see Section 6.3)
- exercise with **gdb** (see Section 6.4)

Chapter 7

Laboratory 2: Makefiles of Nachos & Context Switch in Nachos

1. Makefiles of Nachos

7.1 Purpose

The purpose of this laboratory is

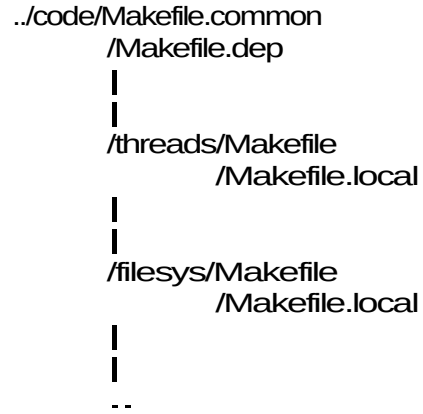
- to understand the makefiles structure of Nachos system, and
- to know how to set up a separate directory to develop a new version of Nachos system.

7.2 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:



7.2.1 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
```

7.2.2 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 right-hand side string is to be *appended* to the original contents of INCPATH and DEFINES.

7.2.3 Makefile.dep

This is the makefile to be included in Makefile.common. It defines a lot of system-dependent 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:

```
[ptang@probus threads]$ pwd
/home/www/staff/ptang/units/204/00/nachos-3.4/code/threads
[ptang@probus threads]$ ls arch
dec-alpha-osf/          sun-sparc-sunos/
dec-mips-ultrix/        unknown-i386-linux/
```

The system-dependent 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 cross-compiler. Its definition shows why you need to install it in the **/usr/local/** directory. This makefile also defines other macros dependent on the system-dependent 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 system-dependent directories in **arch** directory, there are three directories to accommodate object codes, binary executable and dependence files, respectively. For example, in the
arch/unknown-i386-linux/ for your linux system, there are directories as follows:

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

7.2.4 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 non-empty 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 non-empty string. For example, this rule tells how to make **arch/unknown-i386-linux/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):

```
ptang@probus threads]$ g++ -MM -g -Wall -Wshadow -fwritable-strings
                -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 re-compiled 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-i386-linux/depe** whose contents are:

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

7.3 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 re-defines the include path macro INCPATH. The definition of CCFILES does not need changes, because make will follow the **vpath**s to find the required source files if they are not in the current directory.

The re-definition of INCPATH needs changes.

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

(a) **First Solution:** You can change the re-definition 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/unknown-i386-linux/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 -fwritable-strings -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/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]$
```

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 generates the dependence which includes string **../threads/scheduler.h** (check the contents of **main.d** in **../lab2/arch/unknown-i386-linux/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 time-stamp of **scheduler.h** in **../lab2/** and then make Nachos again. The make command should cause re-compiling 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 -fwritable-strings -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 time-stamp of **../threads/scheduler.h** and try the make Nachos again. This time, none of the modules should be re-compiled 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]$
```

(b) **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 re-definition 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/unknown-i386-linux/bin/nachos' is up to date.
```

If we touch the **scheduler.h** in the current directory **../lab2**, it will make Nachos recompiled 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 -fwritable-strings -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/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]$
```

7.4 Things to Do

Your tasks in this lab session are as follows:

(a) Read Section 7.2 and make sure you understand the make-file structure of Nachos.

(b) Experiment with the two solutions to build a new Nachos in a separate directory described in Section 7.3. Make sure you understand why both solutions are correct.

2. Context Switch in Nachos

7.1' Purpose

The purpose of this laboratory is

- understand how context switch is realized in Nachos by tracing the Nachos test program in both C++ and the machine code levels.

7.2' Tasks

The main.cc program of Nachos in ../threads/ calls function ThreadTest() as follows:

```
void
ThreadTest()
{
    DEBUG('t', "Entering SimpleTest");

    Thread *t = new Thread("forked thread");

    t-
    >Fork(SimpleT
    hread, 1);
    SimpleThread(
    0);
}
```

The SimpleThread() function used above is as follows:

```
void
SimpleThread(_int which)
{
    int num;

    for (num = 0; num < 5; num++) {
        printf("*** thread %d looped %d times\n", (int) which, num);
        currentThread->Yield();
    }
}
```

Your tasks of this lab session is to

1. trace the execution of Nachos and observe the executions of

(a) context switch function SWITCH()

(b) function ThreadRoot()

using gdb and

2. answer the following questions:

(a) What are the addresses of the following functions in your Nachos:

i. InterruptEnable()

ii. SimpleThread()

iii. ThreadFinish()

iv. ThreadRoot()

and describe how did you find them.

(b) What are the addresses of the thread objects for

i. the main thread of the Nachos

ii. the forked thread created by the main thread

and describe how did you find them.

(c) When the main thread executes SWITCH() function for the first time, to what address the CPU returns when it executes the last instruction ret of SWITCH()? What location in the program that address is referred to?

(d) When the forked thread executes SWITCH() function for the first time, to what address the CPU returns when it executes the last instruction ret of SWITCH()? What location in the program that address is referred to?

Chapter 8

Laboratory 3: Synchronization Using Semaphores

8.1 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.

8.2 Background

8.2.1 Semaphores

Semaphores are one of the most commonly used synchronization schemes for concurrent processes or threads. Section 7.4 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`.

8.2.2 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 non-full. Similarly, a consumer has to be blocked when the buffer is empty and resumed when it becomes non-empty. Consequently, producers and consumers need a mechanism for synchronization.

8.2.3 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`.

8.3 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:

(a) Read **ring.h** and **ring.cc** and make sure that you understand everything in them.

(b) Read **main.cc**.

(c) Read **prodcons++.cc** and make sure that you understand

i. the structure of the program

ii. the task to complete the program

(d) Complete all programs in file **prodcons++.cc**.

(e) 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** *seed-number* and check that all results satisfy the above criteria.

Chapter 9

Laboratory 4: Nachos File System

9.1 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.

9.2 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**.

9.3 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 working yet.

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

9.4 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.

9.4.1 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.

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

(b) 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.

9.5 Things to Do

9.5.1 Compiling Nachos File System

Follow the description in Section 9.2 to compile the Nachos with its file system in **../filesys/**).

9.5.2 Testing Nachos File System

Execute the following commands and check the results as described:

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

(b) 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
```

```
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
Directory file header:
FileHeader contents.  File size: 200.  File blocks:
3 4
File contents:
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
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.

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

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

(d) 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.

(e) Continue the experiment with other nachos commands such as nachos -r on more files.

9.6 Questions

(a) 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`?

(b) What are the sector numbers of data blocks for file `big`?

(c) What is the sector number of the disk to store the file header for file `big`?

(d) 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?

Chapter 10

Laboratory 5: Extendable Files

10.1 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 10.1, 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.

10.2 Analysis

The Nachos file system consists of the following modules:

- class Disk

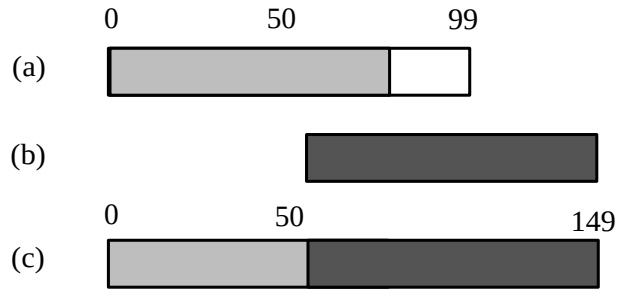


Figure 10.1: Extension of a file

- class SynchDisk
- class BitMap
- class FileHeader
- class OpenFile
- class Directory
- class FileSystem

The structure of the file system is shown in Figure 10.2.

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.

10.3 Things to Do

10.3.1 Analysis

Answer the questions in Section 10.2. 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.

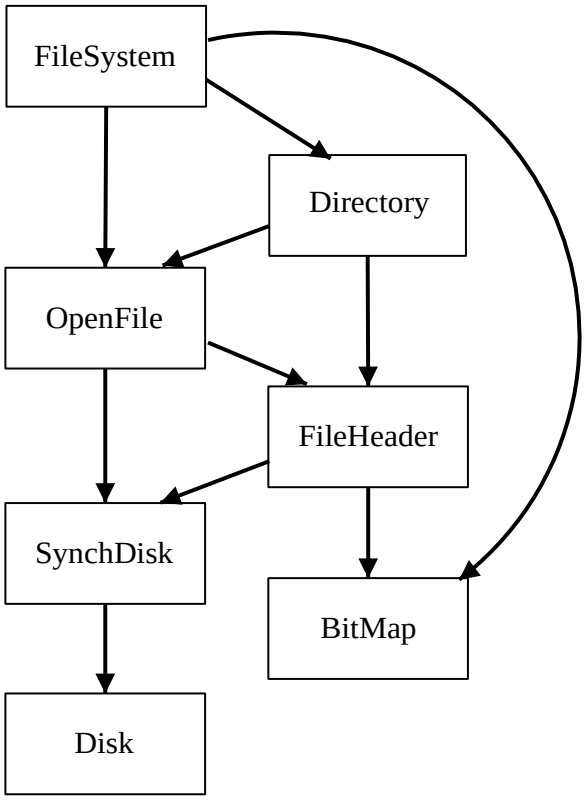


Figure 10.2: Structure of Nachos File System

10.3.2 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 10.4. **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.

10.4 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 over-writes 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\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O\O
```

```
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\0\0G
\5\8\0\0\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\0\2\0
\0\0\a0F\5\8\0\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\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 i-node 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 i-node 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 i-node 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.

Chapter 11

Laboratory 6: Nachos User Programs and System Calls

11.1 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

11.2 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 cross-compiler 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.

11.3 Things to Do

11.3.1 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 = i-1;
    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/decstation-ultrix-gcc -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 understanding of how user programs are started and executed in Nachos.

11.3.2 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.

11.3.3 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.

Chapter 12

Laboratory 7: Extension of AddrSpace

12.1 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.

12.2 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 read-only
    }
--
```

12.3 Things to Do

You have two tasks in this lab:

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

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

12.4 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.

12.5 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.

Chapter 13

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

13.1 Objectives

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

13.2 Working directory

The working directory of Labs 7 and 8 is **../lab7-8/**. 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.

13.3 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

(a) 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
---
```

(b) 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*.

13.4 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.

13.4.1 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.

13.4.2 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);
}
```

13.4.3 SpaceId

Exec() returns a value of type SpaceId. This value will be used as the argument of Join() to identify the newly-created 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.

13.4.4 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.

13.5 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.

13.6 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. I will read your report and give your feedback.

Part III
Assignments

This part includes the description of three assignments. To help you prepare the submission of these assignments, we provide the solution to a sample assignment in Chapter 14. You need to read this chapter before you prepare your submissions.

Chapter 14

Sample of Assignment Submission

14.1 Introduction

In this appendix, we provide a sample assignment and its sample submission. The purpose is to show the format of submission and what we expect from you for the questions and programming tasks in assignments of this course.

14.2 Question Types and Requirements

There are three kinds of questions and tasks in the assignments of this course:

Questions on concepts: These questions are mainly concerned with concepts, algorithms and designs of operating systems covered by the text. When answering these questions, you should demonstrate your *own understanding* using your *own* words. **Quoting or copying sentences from the text only is NOT acceptable.**

Questions on Nachos: These questions are designed to test your understanding of implementation of concepts in Nachos.

They can be the questions about the current Nachos implementation or questions about further extensions of Nachos. When answering these questions, you need to quote the relevant code sections of Nachos to demonstrate your understanding of the Nachos system as well as the concepts implemented. The purpose is to see if you know how to implement the concepts in an operating system.

Programming tasks with Nachos: These tasks are the ultimate tests of your understanding of the subject. They are also designed to make sure that you have the *ability* and *skills* to implement operating systems concepts in a real system.

You need to submit a report for each programming task. Your report should give details of how you accomplished the task. It should include three components as follows:

Design: You need to describe all the designs required by the tasks. They should include the designs of

(a) relationship among classes

(b) classes including their data members and functions

(c) interfaces and algorithms of functions

(d) others

Implementation: You need to present relevant code sections to demonstrate the implementation of your designs.

Testing: You need to describe in detail the entire testing process of your programs. In particular, you should

(a) describe the test strategy, test cases and programs

(b) report your test results

(c) analyze the results to show why your programs are correct or incorrect.

14.3 Sample Assignment

Questions

(a) What is the purpose of system calls? (Question 3.7 of the text)

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

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

i. What is unique about this function call?

ii. From the machine's point of view, what thread does this function call return to?

iii. From the viewpoint of thread A, when and how does this function call return?

Programming Tasks

(a) In this programming task, you are required to implement and experiment with the bounded-buffer algorithm using semaphores. The program structure for both producer and consumer processes is introduced in the section 7.1 of the text. The bounded-buffer is an array of a certain data type. The algorithms of producer and consumer processes using semaphores for synchronization can be found in the section 7.5.1 of the text.

Your program should work correctly according to the requirements of the bounded-buffer problem.

14.4 Sample Submission

Questions

(a) What is the purpose of system calls? (Question 3.7 of the text)
Answer: System calls are the interface between user programs and the operating system kernel. This interface is in the form of ordinary functions in system programming languages like C or assembly languages. For example, UNIX system calls have interfaces in C and they look like C library functions. However the purpose of system calls is completely different from that of C library functions.

User programs invoke systems calls to get the services from the operating system kernel for those tasks which cannot be done without the operations of the underlying operating system. Such tasks may be read and write on an open file or create another process to run a program concurrently. These tasks are completed by the kernel running in system mode. That way, the access to the critical shared hardware such as the memory, disks and interrupts can be protected from errant users because user programs can only run in the user mode.

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

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

i. What is unique about this function call?

ii. From the machine's point of view, what thread does this function call return to?

iii. From the viewpoint of thread A, when and how does this function call return?

Answer:

i. First of all, this is an invocation of the assembly function `SWITCH(..)`, rather than an ordinary C++ function. Secondly, the return of this functions call has different semantics depending on whether you look at it from the machine point of view or the view point of the calling thread. The details of the differences will be provided in the following two sub-questions. Most importantly, between the time of calling this function and the time of the return to the *calling thread*, the machine has at least two context switches: one to the another thread and another to switch back to this calling thread.

ii. To answer the question, we need to trace this function call in detail. The relevant part of the source code of function `Run(..)` which calls `SWITCH(..)` is as follows:

```
 90 void
 91 Scheduler::Run (Thread *nextThread)
 92 {
 93     Thread *oldThread = currentThread;
 94
---
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());
--
115
116     SWITCH(oldThread, nextThread);
117
118     DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
119
--
124     if (threadToBeDestroyed != NULL) {
125         delete threadToBeDestroyed;
126         threadToBeDestroyed = NULL;
127     }
128
135 }
```

When SWITCH(..) is called, oldThread points to thread A and nextThread to thread B. Assuming that Nachos is running in a MIPS machine, the code for SWITCH(..) is 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 registers a0 and a1 hold the references to threads A and B, respectively, when this subroutine is started. Lines 89-99 save all the registers of the CPU into the control block of thread A. In particular, the return address of this function SWITCH(..) call contained in register ra is the address of first instruction of line 118 in Run(). This address is saved at line 93.

After these registers are re-loaded with the contents saved previously in the control block of thread B at lines 101-111, the CPU has the new context of thread B. In particular, register ra now has the whatever return address was saved when thread B was switched off previously, or the initial PC value if this is the first time for thread B to run. In the former case, the new ra contains the same address of first instruction of line 118 in Run() (Remember that all threads share the same kernel and the scheduler is part of the kernel). Otherwise, ra should have the starting address of subroutine ThreadRoot.

The instruction at line 113 finishes this function and the CPU starts to execute the instruction pointed by ra. If ra points to address of first instruction of line 118 in Run(), this SWITCH(..) function appears to have returned, but this return is the return to thread B instead of thread A. (If thread B runs for the first time, this return causes thread B to execute subroutine ThreadRoot.)

In summary, from the view point of the machine which can see everything, this SWITCH(..) function call returns to thread B.

iii. However, from the view point of thread A which can only see its own context, this function call takes much time to complete. The return to thread A itself (i.e. the normal return as with any ordinary function calls) won't happen until the context of thread A including its saved ra are restored back into the CPU registers. This can only happen if another thread (not necessarily thread B) calls function SWITCH(..) again in a similar situation. Therefore, there exist at least two context switches in the machine between the call and the return of this function. Of course, thread A will not see these context switches and its only experience is that this function call seems to take a long time to complete.

Programming Task

In this programming task, you are required to implement and experiment with the bounded-buffer algorithm using semaphores. The program structure for both producer and consumer processes is introduced in the section 7.1 of the text. The bounded-buffer is an array of certain data type. The algorithms of producer and consumer processes using semaphores for synchronization can be found in the section 6.5.1 of the text.

Your program should work correctly according to the requirement of the bounded-buffer problem.

Report:

- **Objects and Data Structures Design.** To experiment with the bounded-buffer problem, we need to have
  - a bounded-buffer object,
  - a number of producer threads, and
  - a number of consumer threads.

Producer and consumer threads can be created by using the standard thread creation mechanism in Nachos. We need to provide separate functions for them.

The bounded-buffer object as shown in the section 7.1 of the text does not have synchronization control. To allow multiple producer and consumer threads to access the bounded buffer concurrently, we need to use three semaphores as follows:

- semaphore `mutex` for mutual exclusion,
- semaphore `nempty` to block consumer threads when the buffer is empty, and
- semaphore `nfull` to block producer threads when the buffer is full.

These semaphore objects can be created by using `Semaphore` class in Nachos.

To enable the testing, the data type of the messages to be passed through the bounded-buffer should contain

(a) the identity of the producer thread and

(b) the identity of the message.

A consumer then can record a line in its associate file in the following format:

```
producer id –> X; Message number –> Y;
```

for each of the messages it gets. We could use references of producer threads for their identities, but we chose to use integers for clarity. The message identity is just an integer.

- **Algorithms and Function Interfaces Design.** The bounded-buffer can be regarded as abstract data type with two functions:

(a) function `Put(slot*)` to deposit a message

(b) function `Get(slot*)` to remove a message

Here `slot` is the type of the message to be passed from producers to consumers through the bounded buffer.

The bounded buffer should have two private variables , `in` and `out`, to hold the indexes of next empty and full slots of the buffer, respectively. The algorithms of `Put(..)` and `Get(..)` are from the section 7.1 of the text, except that we do not need variable *counter*. The algorithm of `Put` is as follows:

```
buffer[in] = message;
in = in + 1 mod size
```

where “size” is the size of the buffer. The algorithm of `Put` is as follows:

```
message = buffer[out];
out = out + 1 mod size
return
```

Both functions **Put(..)** and **Get(..)** need a call-by-reference type of message argument to copy the value of the messages in and out of the bounded buffer. In particular, for function **Get(..)**, we cannot just return the reference of the message to be removed, because this message in the buffer may be overwritten immediately after **Get()** is finished.

We could have included the three semaphores above as a private data members of the bounded buffer. However, the algorithm from the text suggests to declare these semaphores outside of the bounded buffer and we follow this design here.

The algorithms for producer and consumer threads are from Figures 7.12 and 7.13 of the text, respectively, as follows:

– Producer

repeat
    ...
    produce an item *nextp*
    ...
    P(empty);
    P(mutex);
    ...
    add *nextp* to *buffer*
    ...
    V(mutex);
    V(full);
until *false*;

– Consumer

repeat
    P(full);
    P(mutex);
    ...
    remove an item from *buffer* to *nextc*
    ...
    V(mutex);
    V(empty);
    ...
    consume the item in *nextc*
    ...
until *false*;

We have to change the infinite loop in the producer thread to a finite loop; otherwise the record files of consumers will be infinitely long and our file system will be filled quickly. Our consumer function also needs to create and open the file for recording the messages received.

- **Implementation.** We start with the overall module structure of the implementation.

  (a) **Program Structure.** First of all, we do not want to mess up the files in **threads/** directory when implementing our program. Of course we need to use all the files for

compiling Nachos in **threads/** directory. We also need to add our own files. The best way is to create a separate directory and do the work in that new directory. In that directory (called **lab2/**), I have the following files:

```
sigma : > ls
Makefile        arch/           prodcons++.cc   ring.h
Makefile.local  main.cc         ring.cc
sigma : >
```

I copied the **main.cc** from **threads/** directory, because I need to change it[1].

New files **ring.h** and **ring.cc** are used to implement the bounded-buffer class. File **prodcons++.cc** is similar to **threadtest.cc** in **threads/** directory and includes all functions run by producer and consumer threads as well as a startup function called **ProdCons()**.

The makefiles are copied from **threads/**, but new files are added to the CC file list in **Makefile.local** as follows:

```
CCFILES = main.cc\
        list.cc\
        scheduler.cc\
        synch.cc\
        synchlist.cc\
        system.cc\
        thread.cc\
        utility.cc\
        threadtest.cc\
        synchtest.cc\
        interrupt.cc\
        sysdep.cc\
        stats.cc\
        timer.cc\
        prodcons++.cc\
        ring.cc
```

(b) **Bounded-Buffer.** The definition of **message** type is as follows:

```
class slot {
    public:
    slot(int id, int number);
    slot() { thread_id = 0; value = 0;};

    int thread_id;
    int value;
    };
```

A bounded-buffer is implemented as an object of class **Ring** whose definition is as follows:

```
class Ring {
  public:
    Ring(int sz);      // Constructor: initialize variables, allocate space.
```

[1]The basic rule here is that when you need to change a file from the original Nachos distribution, copy it to your own directory. The Nachos makefiles will look for the named file in the current directory first.

```
    ~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.


  private:
    int size;           // The size of the ring buffer.
    int in, out;        // Index of
    slot *buffer;       // A pointer to an array for the ring buffer.
};
```

The implementations of functions **Put(..)** and **Get(..)** are as follows:

```
void
Ring::Put(slot *message)
{
    buffer[in].thread_id = message->thread_id;
    buffer[in].value = message->value;
    in = (in + 1) % size;
}

void
Ring::Get(slot *message)
{
    message->thread_id = buffer[out].thread_id;
    message->value = buffer[out].value;
    out = (out + 1) % size;
}
```

(c) **Semaphores.** Three semaphores are declared and defined globally. The pointers to these semaphores are declared in file **prodcons++.cc** as follows:

```
Semaphore *nempty, *nfull; //two semaphores for empty and full slots
Semaphore *mutex;          //semaphore for the mutual exclusion
```

These semaphores are created and initialized at the beginning the test function **ProdCons()** as follows:

```
void
ProdCons()
{
    int i;
    DEBUG('t', "Entering ProdCons");

// create and initialize semaphores
    nempty = new Semaphore("nempty", BUFF_SIZE);
    nfull = new Semaphore("full", 0);
    mutex = new Semaphore("mutex", 1);
---
```

Here **BUFF_SIZE** is the macro for the size of the bounded buffer.

(d) **Consumer and Producer Threads.** To enable testing of different configurations of the problem, we define a couple of macros in prodcons++.cc as follows:

```
#define BUFF_SIZE 3   // the size of the round buffer
#define N_PROD    2   // the number of producers
#define N_CONS    2   // the number of consumers
#define N_MESSG   4   // the number of messages produced by each producer
```

The references to the producer and consume threads are kept in two arrays of Thread pointers as follows:

```
Thread *producers[N_PROD]; //array of pointers to the producer
Thread *consumers[N_CONS]; // and consumer threads;
```

The function for producer threads to run is coded in prodcons++.cc as follows:

```
void
Producer(_int which)
{
    int num;
    slot *message  = new slot(0,0);

    for (num = 0; num < N_MESSG ; num++) {
//      printf("*** producer %d produces %d-th message\n", which,  num);
        message->thread_id = which;
        message->value = num;
        nempty->P();
        mutex->P();
        ring->Put(message);
        mutex->V();
        nfull->V();
    }
}
```

Note that each procedure thread generates only N_MESSG messages. Argument which is the integer identity of the thread we mentioned earlier.
The function for consumer threads is in the same file as follows:

```
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 (; ; )
        { nfull-
        >P();
        mutex->P();
        ring->Get(message);
        mutex->V();
        nempty->V();

        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);
        }
    }
}
```

Note that the consumer thread first creates a UNIX file named "tmp_xxx", where xxx is its integer identity, as the file to record the messages it obtains from the bounded buffer. The way of consuming a message obtained is simply writing to the file a line in the format we designed earlier.

(e) **Start-up Function.** We need a start-up function to

- create the bounded buffer
- create and start producer and consumer threads.

This function will be called the initial Nachos kernel thread. This function is called ProdCons() and defined in prodcons++.cc as follows:

```
void
ProdCons()
{
    int i;
    DEBUG('t', "Entering ProdCons");

// create and initialize semaphores
    nempty = new Semaphore("nempty", BUFF_SIZE);
    nfull = new Semaphore("full", 0);
    mutex = new Semaphore("mutex", 1);

// create a ring buffer object with size BUFF_SIZE;

    ring = new Ring(BUFF_SIZE);

// create and fork threads for producers and consumers
    for (i=0; i < N_PROD; i++)
    {
        sprintf(prod_names[i], "producer_%d", i);
        producers[i] = new Thread(prod_names[i]);
        producers[i]->Fork(Producer, i);
    };

    for (i=0; i < N_CONS; i++)
    {
        sprintf(cons_names[i], "consumer_%d", i);
```

```
            consumers[i] = new Thread(cons_names[i]);
            consumers[i]->Fork(Consumer, i);
        };
    }
```

To create and start a producer or consumer thread, we follow the mechanism in Nachos to first construct a `Thread` object and then call its `Fork(..)` providing the function we want it to run. The integer identity is from the loop variable `i`.

- **Testing.** We start with the rules by which we can judge whether our programs behave correctly. According to the requirement of the bounded buffer problem, a correct implementation should guarantee the following:
  - all the messages produced by the producer threads are received by the consumer threads and recoded 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.

  We run tests on two different configurations. In the first configuration, we create 2 consumers and 2 producers each of which generates 4 messages. The buffer size is 3. We successfully compiled the Nachos kernel in our directory `lab2/` and run `nachos -rs 100` with random seed number 100 as follows:

```
sigma : > nachos -rs 100
file name is tmp_0
file name is tmp_1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 810, idle 20, system 790, 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...
sigma : > ls
Makefile        main.cc         ring.cc         tmp_1
Makefile.local  nachos@         ring.h
arch/           prodcons++.cc   tmp_0
sigma : >
```

We see that two record files are created by the two consumer threads. The contents of these files are:

```
sigma : > less tmp_0
producer id -> 1; Message number -> 0;
producer id -> 1; Message number -> 1;
producer id -> 1; Message number -> 2;
producer id -> 0; Message number -> 0;
```

```
producer id –> 0; Message number –> 1;
producer id –> 0; Message number –> 2;
producer id –> 0; Message number –> 3;
sigma : > less tmp_1
producer id –> 1; Message number –> 3;
sigma : >
```

The output is correct according to the rules described. By running nachos again with rs=99, we got the following results:

```
sigma : > less tmp_0
producer id –> 0; Message number –> 3;
producer id –> 1; Message number –> 2;
producer id –> 1; Message number –> 3;
sigma : > less tmp_1
producer id –> 0; Message number –> 0;
producer id –> 0; Message number –> 1;
producer id –> 0; Message number –> 2;
producer id –> 1; Message number –> 0;
producer id –> 1; Message number –> 1;
sigma : >
```

which are also correct.

Then we changed the configuration to as follows:

```
#define BUFF_SIZE 3  // the size of the round buffer
#define N_PROD    5  // the number of producers
#define N_CONS    2  // the number of consumers
#define N_MESSG   3  // the number of messages produced by each producer
```

We re-compiled the Nachos kernel and run it again with rs=99. The results are:

```
sigma : > less tmp_0
producer id –> 0; Message number –> 0;
producer id –> 0; Message number –> 1;
producer id –> 0; Message number –> 2;
producer id –> 3; Message number –> 0;
producer id –> 3; Message number –> 1;
producer id –> 4; Message number –> 0;
producer id –> 4; Message number –> 1;
producer id –> 3; Message number –> 2;
producer id –> 1; Message number –> 1;
producer id –> 4; Message number –> 2;
producer id –> 1; Message number –> 2;
sigma : > less tmp_1
producer id –> 1; Message number –> 0;
producer id –> 2; Message number –> 0;
producer id –> 2; Message number –> 1;
producer id –> 2; Message number –> 2;
sigma : >
```

Another run with rs=50 gives the following results:

```
sigma : > less tmp_0
producer id –> 0; Message number –> 0;
producer id –> 1; Message number –> 0;
producer id –> 3; Message number –> 0;
producer id –> 3; Message number –> 1;
producer id –> 1; Message number –> 1;
producer id –> 1; Message number –> 2;
producer id –> 4; Message number –> 2;
sigma : > less tmp_1
producer id –> 2; Message number –> 0;
producer id –> 2; Message number –> 1;
producer id –> 4; Message number –> 0;
producer id –> 0; Message number –> 1;
producer id –> 4; Message number –> 1;
producer id –> 2; Message number –> 2;
producer id –> 3; Message number –> 2;
producer id –> 0; Message number –> 2;
sigma : >
```

While it is impossible to do exhaustive tests for all configurations and all **rs** values, the test results above do show that our program appears to be correct.

Chapter 15

Assignment 1: Overview and Processes

Due: 16 August, 2002

Weight: 15%

15.1 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.

15.2 Questions

Answer the following questions:

(a) (**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 short-term 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 relevant Nachos source code to assist your answer.

(e) 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
```

Chapter 16

Assignment 2: Synchronization and Monitors

Due: 6 September, 2002

Weight: 15%

16.1 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.

16.2 Questions

The textbook (pages 220-221) 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.

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

(b) 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.

(c) 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.

16.3 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 16.1: Ring Buffer Monitor

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

Monitors use condition variables for interprocess synchronization. Currently, Nachos has only the Mesa-style 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 Hoare-style condition variables and then use them to implement the producer/consumer problem with monitors.

16.3.1 Monitor Class Ring

The monitor type is a high-level 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. Object-Oriented 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 low-level 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 16.1 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 non-0 if the ring is full, 0 otherwise.
    int Empty();       // Returns non-0 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 16.1.

16.3.2 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: Hoare-style and Mesa-style. The Hoare style semantics is described as choice 1 in the discussion on page 218 of the textbook. The Mesa-style corresponds to the second choice in that discussion.

You are not allowed to use the Mesa-style condition variables implemented in Nachos. Instead, you are required to implement the class of condition variable in the **Hoare-style**. The Hoare-style 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 Hoare-style 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.

16.3.3 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.

16.4 Submission

You need to submit:

(a) The answers to the questions in Section 16.2.

(b) 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 self-contained and include 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 Introductory 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 cut-and-paste 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.

(c) 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.

Chapter 17

Assignment 3: File System Interface and Implementation

Due: 18 October, 2002

Weight: 15%

17.1 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 17.2 below. The last part is similar to the second part which focuses on a programming task described in Section 17.3.

You need to submit your Nachos system with this assignment after you finish the work of Lab 5 and the programming task specified in 17.3.

17.2 Work of Lab 5

(a) 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.

(b) 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?

(c) 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`

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

17.3 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 programming task. Let us look at this command in more detail first.

17.3.1 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  \0       H 005  \b
0000040   (  \0  \0  \0 234    ▮ 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   ▮ 005  \b  \0  \0  \0  \0   (   ▮ 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    ▮ 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    ▮ 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  \0       ▮ 005  \b
0000340   (  \0  \0  \0 004    J 005  \b  \b   G 021    @    E 005  \b  \0
0000360        ▮ 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    ▮
0000620    ▮  \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  \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      ▮ 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      ▮ 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  \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   ▮   ▮       f   i   ▮   e       s
0001420   m   a   ▮   ▮       f   i   ▮   e       s   m   a   ▮   ▮
0001440   f   i   ▮   e  \n   s   m   a   ▮   ▮       f   i   ▮   e
0001460   s   m   a   ▮   ▮       f   i   ▮   e       s   m   a   ▮   ▮
0001500       f   i   ▮   e  \n   *   *   *   e   n   d       o   f
0001520   f   i   ▮   e   *   *   *  \n       F 021   @ 200   G 005  \b
0001540 030  \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 $200_8 \times 6_8 = 1400_8$, 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.

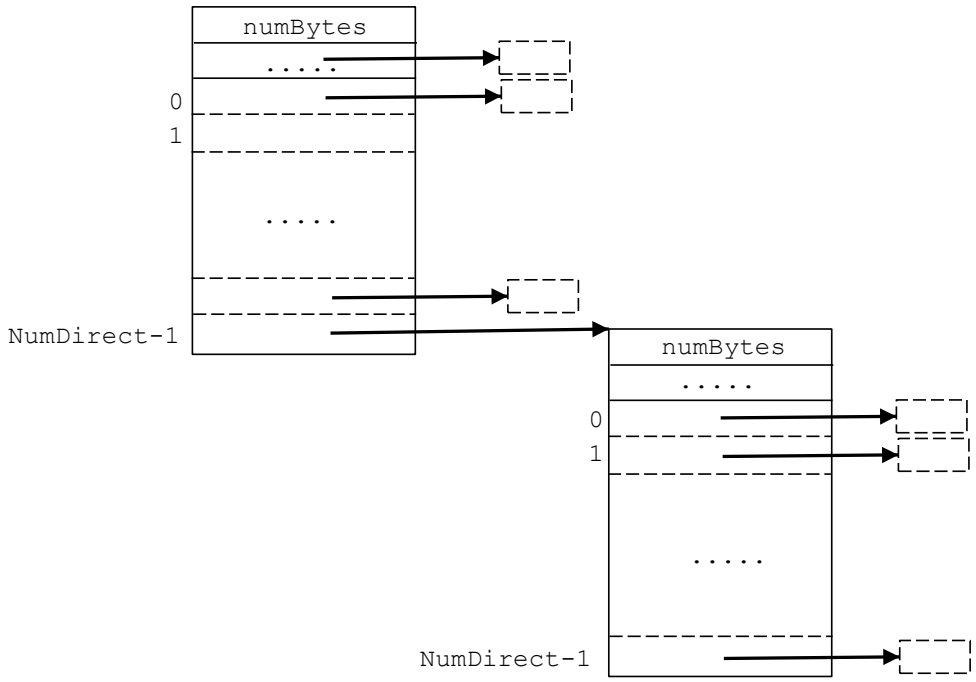
17.3.2 Objectives

The Nachos file system uses single-level index allocation method for data allocation of files. In the current design, the file header (i-node) 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 two-level 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 i-node when necessary. When the file is large, the i-node 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 (i-node) 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 i-node. The indirect i-node exists only if the size exceeds ((NumDirect-1) * SectorSize).

17.3.3 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.

17.3.4 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.

17.3.5 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
-rw-r--r--   1 ptang        337 Oct   2 17:15 big
-rw-r--r--   1 ptang       3804 Oct   2 17:13 huge
-rw-r--r--   1 ptang       3573 Oct   2 17:12 huge1
-rw-r--r--   1 ptang       3678 Oct   2 17:13 huge2
-rw-r--r--   1 ptang        169 Oct   2 17:14 medium
-rw-r--r--   1 ptang          1 Sep 28 18:07 onebyte
-rw-r--r--   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\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\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 i-node 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 continues on sectors 35 and 36, skipping sector 34 which is used for the indirect i-node.

```
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
*
```

17.3.6 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.

17.3.7 Questions to Assist Your Report

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

(a) 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.

(b) Show the relevant source code for these changes.

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

(d) In the indirect i-node, 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 i-node just as we did in the director i-node. 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.

17.4 What to submit?

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

(a) the paper work of the answers to the questions listed in Section 17.2.

(b) the report of this programming assignment.

(c) 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 subdirectories as well as the original Nachos system.