操作系统课程设计指南

Nachos (C++)

韩芳溪

hfx@sdu.edu.cn

山东大学计算机科学与技术学院

2021 年 9 月修订

目 录

第 1 章 Nachos 简介

Nachos 是由加州大学伯克利分校的 Tom Anderson 教授基于 C++实现的一个教学用操作系统。在学习了操作系统的概念与原理之后，通过阅读 Nachos 中相关内容的源代码并实践编程扩展文件系统、系统调用的部分功能，可以更加深入地理解操作系统的相关概念与工作原理，掌握实际操作系统的开发过程与方法。

掌握操作系统设计与实现方法的最有效途径就是阅读一个真正操作系统的实现代码。Nachos 操作系统设计精良、结构紧凑，由大约 9,500 行 C++代码组成，并做了大量的注释，被全球众多高校用来辅助操作系统课程的教学。

Nachos 系统运行在像 Ubuntu 之类的 Linux 操作系统之上，它作为 Linux 的一个进程来运行。

Nachos 系统包括两部分：模拟的硬件系统及在这些硬件之上运行的操作系统内核。

1.1 Nachos 的硬件系统

首先 Nachos 模拟了计算机的硬件系统，如 CPU、中断控制器、定时器、时钟、网卡、I/O 终端、磁盘、内存、MMU 等硬件设备，然后在这些模拟的硬件系统之上运行了一个操作系统内核。硬件系统的实现代码都在.../machine 目录中。

1.1.1 CPU

对于一台实际的机器，CPU 负责执行操作系统以及应用程序。但对于 Nachos 模拟的 CPU，只是用来执行基于 Nachos 操作系统环境下的应用程序（实验 6、7、8），不是用来执行 Nachos 本身。Nachos 内核本身在 Linux 环境下运行。

Nachos 模拟的 CPU 基于 MIPS 架构，执行 MIPS 指令集，用来执行 Nachos 的应用程序；因此 Nachos 的编译程序应该将 Nachos 的应用程序（类似于 C 程序）编译成包含 MIPS 指令集的可执行程序，以便在 Nachos 系统上运行。。

Nachos 本身没有提供编译器，其应用程序只能在 Linux 环境下编程，并在 Linux 环境下将其编译成基于 MIPS 指令集的应用程序，然后利用 Nachos 提供的工具 coff2noff 将其转换成 Nachos 的应用程序（noff 格式）。

Nachos 系统提供的交叉编译器 gcc-2.8.1-mips.tar.gz 提供的 gcc、g++、as、ld 等工具负责实现该功能。

在../test 目录中提供了几个 Nachos 应用程序示例，如 halt.c、matmult.c、sort.c、shell.c，这几个程序都是按照 Nachos 应用程序的语法及 Nachos 系统调用的要求编写的，尽管它们的扩展名为.c，但 Nachos 应用程序对标准 C 语言中的绝大部分函数均不提供支持；它的编译器及链接程序不是 UNIX 通用的 GCC，而是文件 gcc-2.8.1-mips.tar.gz 中提供的 gcc、as、ld 等程序；由于在 Makefile.dep 中将这些程序的路径设

置为/usr/local/mips/bin/decstation-ultrix-，**因此，应该按照第二章实验一中的说明，将交叉编译程序 gcc-2.8.1-mips.tar.gz 安装到指定目录/usr/local 中。**

与 Nachos CPU 执行用户程序指令的过程：

相关的几个类是 class Machine, class Instruction, class TranslationEntry， class AddrSpace,参见../machine/machine.cc，mipssim.cc，translate.cc，../userprof/addrspace.cc, progtest.cc 等；

1. 为用户程序分配内存空间，将程序装入内存，建立页表；（AddrSpace:: AddrSpace()）

2. 将应用进程映射到一个核心线程；（progtest.cc 中 startProcee()）

3. 将应用程序页表传递给内核中的系统页表，设置 PC 及栈指针；（progtest.cc 中 startProcee()）

4. 启动进程执行；（progtest.cc 中 startProcee()调用 Machine:;Run()。Machine:;Run() 在../machine/mipssim.cc 中实现）；

5. 系统依据应用进程的页表将 PC 给出的程序入口地址（该地址是程序的虚地址，NOFF 文件中是 0）转换成内存物理地址，CPU 依据该内存地址从用户空间代码段中取出相应的指令，并译码执行；

随后 PC+1->PC，取出第二条指令执行；该过程循环进行，直至该应用程序执行系统调用 Exit()结束，或者遇到异常退出；

当然，当应用进程所关联的核心线程发生上下文切换，或者执行完一条指令后响应中断，会保存进程的上下文暂停执行，以后相关联的线程被调度后会继续应用进程的执行；

注意 CPU 执行完一条指令后，执行 interrupt->OneTick()，以允许中断控制器可以响应中断请求。

1.1.2 中断控制器

中断控制器模拟了实际机器中的硬件中断控制器的功能。

1、几个供外部程序使用的调用接口；

（1）Interrupt::SetLevel(IntOff)：用于关中断；

（2）Interrupt::SetLevel(IntOn)：用于开中断，并返回原中断的开关状态；外部程序可通过这两个接口实现一些原子操作；

（3）Interrupt::Halt()：实现停机操作；

（4）Interrupt::Idle()：相当于一般操作系统中的 idle 进程；

当一个线程因等待某事件进入睡眠状态或执行结束（Nachos 中当一个线程执行结束后也是先进入睡眠状态，等待系统销毁。Nachos 中没有设置线程的终止状态），就会引起线程调度。

当线程调度时，如果就绪队列为空，即没有就绪进程可被调度执行，则 Nachos 调用 Interrupt::Idle()，处理已经到期的中断，所有到期的中断处理结束后，继续检查就绪队列是否为空。如果就绪队列不空，则调度符合条件的线程执行；如果就绪队列仍然为空，则再次执行 Interrupt::Idle()处理到期的中断。重复该过程，直到中断请求

队列中的所有中断都处理完毕，且就绪队列为空，则调用 Interrupt::Halt()停机；

上述过程参见../threads/thread.cc 中的 Finish ()、Sleep ()，以及../machine/interrupt.cc 中的 Idle()。

（5）硬件设备提出中断请求

外部程序可通过接口 Interrupt::Schedule(VoidFunctionPtr handler, _int arg, int fromNow, IntType type)模拟硬件设备提出中断请求。当系统响应中断后执行中断处理程序 handler，其参数是 arg；该中断将在系统时钟从当前开始计数 fromNow 后被中断控制器响应，IntType 是硬件中断类型，可以是 TimerInt、DiskInt、ConsoleWriteInt、ConsoleReadInt、NetworkSendInt、NetworkRecvInt，对应硬件设备或相应的硬件操作；

特别指出的是，中断控制器模拟的是硬件设备提出的中断请求，只能在硬件（如"timer", "disk", "console write", "console read", "network send", "network recv"）的相应操作中利用 Interrupt::Schedule()提出中断请求，如读键盘、写屏幕、硬盘读写、设置定时器中断、网卡收发信息等，其它情况不要随便调用；

Interrupt::Schedule()根据硬件提出的中断请求信息将该中断封装成一个 PendingInterrupt 对象，然后将其放入中断请求队列的相应位置（该队列是个有序队列）；

在封装一个 PendingInterrupt 中断对象时，将该中断响应的时间设置为“提出中断请求时刻+fromNow”，并将各中断请求在中断请求队列中按该时间升序排列；

Nachos 维护一个系统时钟，启动后开始计数（ticks）；因此中断控制器只需将每个 PendingInterrupt 对象中记录的时间与系统时间进行比较，就很容易从中断等待队列队首开始检查是否存在需要响应的中断；

当发现有到期的中断，中断控制器从请求队列队首取出执行其中断处理程序。

2、中断响应的时机

Nachos 中断控制器对中断的处理过程不同于一个实际的机器对中断处理过程；

在实际的机器中，当硬件设备提出中断请求并中断判优后，进入一个中断请求队列；如果目前处于开中断，则当 CPU 执行完一条指令后，会响应中断请求队列中优先级最高的一个中断；

Nachos 的硬件设备提出中断请求时，除了提供中断处理程序外，还需要提供一个等待时间，表示从目前系统时间开始，需要等待多长时间需要响应该中断；

Nachos 中断响应的时机也与实际硬件中断有所区别；Nachos 中断控制器只有在以下两种时机才检查是否存在到期的中断并响应中断（参见 Interrupt::OneTick()与 Interrupt::CheckIfDue()）：

a. 中断状态从关到开；

b. Nachos 的 CPU 执行完一条应用程序指令；

因为只有上述两种情况发生时，系统时钟才增量（Interrupt::OneTick()，对于情况 a 增 10，情况 b 增 1），这时 Nachos 中断控制器才检查是否有中断到期，如果有，则响应；

1.1.3 Timer

模拟硬件时钟中断。如果 Nachos 运行时带有-rs 参数并提供一个随机数种子，即 nachos –rs randomseeds，则在 system.cc 中 Initilize()函数初始化系统内核时会创建一个 Timer 设备，间隔一定时间（随机数）调用其中断处理程序，目前实现的代码如下：

```
static void TimerInterruptHandler(_int dummy)
{
    if (interrupt->getStatus() != IdleMode)
    interrupt->YieldOnReturn();
}
```

分析 Timer::Timer()可以看出，定时器间隔一段时间就会执行上述 Timer 处理程序；

通过 Interrupt::YieldOnReturn()、Interrupt::OneTick()与 Interrupt::CheckIfDue()等几个相关的函数可以看出，上述定时器中断处理程序的执行结果是当中断控制器满足中断响应的两个条件之一时，会执行 currentThread->Yield()，实现了“时间片+FCFS”线程调度算法，即 RR 线程调度算法。

1.1.4 Console

Nachos 利用其中断控制器所提供的中断机制，从 UNIX 的标准输入设备 stdin（0 号设备）读取数据，或将信息写入 UNIX 的标准输出设备 stdout（1 号设备），模拟了控制台的输入与输出功能。（参见../machine/console.cc 与 console.h）

1.1.5 内存与 MMU

在../machine/machine.h 中，利用语句 char *mainMemory 定义创建 Nachos 的内存。内存采用分页管理，默认包含 32 页（#define NumPhysPages 32），每页大小同硬盘的一个扇区（#define PageSize SectorSize），大小是 128 字节，NumPhysPages*PageSize 即为 Nachos 内存所包含的字节数。（参见../machine/machine.h，disk.h），

为应用程序分配内存以及建立页表的过程参见../ userprog/ addrspace.cc 中 class AddrSpace 的构造方法。

页表结构参见../machine/ translate.h 中的 class TranslationEntry，如下所示。

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

};

MMU 负责逻辑地址到物理地址的变换、存储保护以及 TLB 的管理等功能。Nachod 中../machine/ translate.cc 中的 Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)方法实现了 MMU 的功能。

读写内存采用../machine/ translate.cc 中的 Machine::ReadMem(int addr, int size, int *value)与 Machine::WriteMem(int addr, int size, int value)方法。

1.1.6 硬盘

Nachos 中使用一个文件模拟了磁盘的功能，包括磁盘格式化、寻道延迟、硬件中断读/写磁盘等功能。硬盘默认有 32 个道（NumTracks），每个道有 32 个扇区（SectorsPerTrack），每个扇区有 128 字节（SectorSize），因此磁盘大容量为 SectorsPerTrack * NumTracks* SectorSize=32*32*128=131072Bytes=128KB 。（参见../machine/disk.h）。

硬盘的创建过程参见../ filesys/synchdisk.cc 中 SynchDisk 类的构造方法。

1.2 Nachos 内核

Nachos 既然是一个操作系统，就要有一个操作系统内核，并基于该内核运行基于 Nachos 的应用程序；内核实现了一个真正操作系统的主要功能，包括线程的管理、系统调用、内存管理、虚存存储、硬盘及文件系统、I/O 等；

1.2.1 线程管理

线程管理包括线程的创建、睡眠、释放 CPU、调度、撤销等

1、线程的状态

Nachos 的进程在其生命期中包括 5 个状态，参见头文件 ../threads/thread.h；

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

其中，状态 JUST_CREATED 是新建一个 Nachos 线程后的状态，此时尚未为该新建的线程分配栈等资源。该状态相当于教材中进（线）程五状态图中的状态 New；

下述构造函数创建一个新的线程，并赋予 NEW 状态：（参见../threads/thread.cc）

```
Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED; //相当于教材中的 NEW
#ifdef USER_PROGRAM
    space = NULL;   //映射到该核心线程的用户进（线）程的地址空间
#endif
```

```
    }
```

2、线程创建

（1）主线程 main

在启动 Nachos 初始化其内核时（参见../threads/system.cc 中 Initialize(…)），Nachos 创建了它的第一个线程--主线程 main，该线程应该是 Nachos 所有其它线程的“始祖”，尽管 Nachos 中没有建立线程树来描述线程之间的家族关系；（其它线程利用 Thread::Fork()创建）；

主线程 main 创建时就绪队列为空，因此通过线程调度而直接将该线程的状态设为 RUNNING，作为当前正在执行的线程（currentThread = new Thread("main")），如若不然，currentThread 会为空，当新建线程在就绪队列被调度运行时，需要保持当前进程的上下文时无法正确执行；

从 system.cc 的 Initialize()创建主线程的过程中可以看出，主线程没有像利用 Fork()创建线程那样，显式的给出其执行体（执行代码）；但语句 currentThread = new Thread("main")后的代码即在主线程中执行，即为主线程的执行体；该观点可以在 threadtest.cc 中得到印证。

特别指出的是，当按下 ctrl+c，或者无就绪进程可调度时，Nachos 将会退出，这时主线程会主动调用 Finish()终止自己，因此在 Nachos 运行期间，主线程不会终止退出；

因为一旦在 Nachos 运行期间调用了 Finish()终止了主线程，主线程会进入睡眠，没有事件会将它唤醒，导致 Finish()之后的代码将不会被执行；Nachos 会调度主线程创建的所有子线程，当这些子线程执行结束后，Nachos 会调用 Interrupt:Idle()，处理完所有中断，然后退出 Nachos；

Nachos 中只有主线程将要退出时显式调用 Finish()，其它利用 Fork()创建的线程退出时不需要显式调用 Finish()，退出时会自动调用。（思考如何实现的？）

（2）一般线程的创建

../threads/threadtest.cc 中给出了创建一个 Nachos 线程的方法：

```
    Thread *t = new Thread("forked thread");
    t->Fork(SimpleThread, 1);
```

其中，"forked thread"是新建线程的名字，SimpleThread 是线程的执行体，1 是 SimpleThread 的参数，相当于新建线程执行 SimpleThread(1)；

SimpleThread()函数的代码如下：

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

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

Thread 类的构造函数如下：

```
Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;   //
#endif
}
```

从 Thread 类的构造函数可以看出，语句 Thread *t = new Thread("forked thread") 实例化 Thread 类的一个对象时，为该线程命名，由于系统尚未给该线程分配栈资源，也尚未制定其执行体，因此其状态为“**JUST_CREATED**”；（主线程 main 是个特例）；

当运行一个 Nachos 应用程序时，需要为该应用程序进（线）程分配一个核心线程，以便能够在 CPU 上运行，space 指向系统为应用进程所分配的地址空间，实现了用户进程与核心线程之间的映射关系；当调度到该线程时，会从 space 所指向的地址空间中取出应用程序指令运行；参见../threads/scheduler.cc，Scheduler::Run()中语句 currentThread->space->RestoreState()将用户应用程序进程的页表传递给系统页表；

注：目前 Nachos 尚未支持用户多线程机制，因此每个用户进程只有一个用户线程；

Thread::Fork()代码如下：

```
void Thread::Fork(VoidFunctionPtr func, _int arg)
{
    DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",
                name, (int) func, arg);

    StackAllocate(func, arg);

    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this); // ReadyToRun assumes that interrupts
                                 // are disabled!
```

```
        (void) interrupt->SetLevel(oldLevel);
    }
```

Thread::StackAllocate()的代码如下：

```
    void Thread::StackAllocate (VoidFunctionPtr func, _int arg)
    {
        stack = (int *) AllocBoundedArray(StackSize * sizeof(_int));
        machineState[PCState] = (_int) ThreadRoot;
        machineState[StartupPCState] = (_int) InterruptEnable;
        machineState[InitialPCState] = (_int) func;
        machineState[InitialArgState] = arg;
        machineState[WhenDonePCState] = (_int) ThreadFinish;
    }
```

从 Thread::Fork()代码可以看出，语句 t->Fork(SimpleThread, 1)执行时，调用 Thread::StackAllocate()为该线程分配栈资源，并设置线程入口为 ThreadRoot()，线程的执行代码为 SimpleThread()，SimpleThread()的参数是 1，线程执行结束时调用 Thread::Finish()；

然后 scheduler->ReadyToRun(this)将线程状态由 **JUST_CREATED** 转换为 **READY**，并将该线程加入就绪队列尾，等待线程调度；

思考 1：新建线程何时被调度执行？

思考 2：线程被调度执行后，其入口在哪里？从哪里开始执行？

思考 3：线程体执行结束后，如何终止该线程？

（3）Idle 线程

一般的系统中（像 Windows），有一个 idle 进（线）程，当就绪队列为空时，就执行该 idle 进程，如果有进程进入就绪队列，idle 进程就会被抢先；

idle 进程一般没有正式任务可完成，可让其清零空闲内存；

Nachos 没有创建一个这样的 idle 线程，但当就绪队列为空时，执行 Interrupt::Idle()，处理所有到期的中断，然后返回，以便系统检查就绪队列中是否有就绪线程等待调度；（参见 Thread::Sleep()）

3、线程就绪

下述代码将线程设置为就绪状态，并加入就绪队列尾等待线程调度；

```
void Scheduler::ReadyToRun (Thread *thread)
    {
        DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

        thread->setStatus(READY);
        readyList->Append((void *)thread);
    }
```

回顾教材中的线程状态转换图，有三种情况可能使一个线程变为就绪：

（1）JUST_CREATED→READY,

新建一个线程，为其分配资源并设置运行环境后，调用上述使线程变为就绪（参见 Thread::Fork()）；（../threads/thread.cc）

（2）BLOCKED→READY,

唤醒一个睡眠（等待、阻塞）的进程时调用上述函数将线程变为就绪（参见在 Semaphore::V()）；（../threads/ synch.cc）

（3）RUNNING,→READY,

参见 Thread::Yield()；（../threads/thread.cc）

4、线程睡眠（等待、阻塞）

下述代码使线程由执行状态 RUNNING→ BLOCKED；

```
    void Thread::Sleep()
    {
        Thread *nextThread;

        ASSERT(this == currentThread);
        ASSERT(interrupt->getLevel() == IntOff);

        DEBUG('t', "Sleeping thread \"%s\"\n", getName());

        status = BLOCKED;
        while ((nextThread = scheduler->FindNextToRun()) == NULL)
            interrupt->Idle();      // no one to run, wait for an interrupt

        scheduler->Run(nextThread); // returns when we've been signalled
    }
```

先考察信号量的 P()操作：

```
void Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // disable interrupts

    while (value = = 0) {                  // semaphore not available
        queue->Append((void *)currentThread);    // so go to sleep
        currentThread->Sleep();
    }
    value--;                               // semaphore available,    consume its value

    (void) interrupt->SetLevel(oldLevel); // re-enable interrupts
}
```

由 P()操作的代码可以看出，当一个正在执行的线程所申请的资源不可用时，首先将其加入该信号量（资源）的等待队列中，然后调用 Sleep()进入睡眠，当资源可用时 V()操作中再将其唤醒；

Sleep()中，将当前线程的状态由 RUNNING→ BLOCKED，然后从就绪队列中调度队首进程执行；

如果当前就绪队列为空，没有线程可执行，就循环调用 Interrupt::Idle()处理所有目前到期的中断，直到就绪队列中有就绪线程可以调度执行；

如果 Interrupt::Idle()中处理完中断请求队列中**所有中断**后（定时器 Timer 中断除外），还是没有就绪进程可以调度，则进入睡眠的线程也就不会被唤醒（可能出现了死锁线程），Nachos 就停机退出；（参见 Interrupt::Idle()；）

注：Thread::Finish()调用了 Thread ::Sleep()；

5、释放 CPU（Yield()）

使线程由 RUNNING,→READY,

```
void Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ASSERT(this == currentThread);
    DEBUG('t', "Yielding thread \"%s\"\n", getName());

    nextThread = scheduler->FindNextToRun();
    if (nextThread != NULL) {
        scheduler->ReadyToRun(this);
        scheduler->Run(nextThread);
    }
    (void) interrupt->SetLevel(oldLevel);
}
```

Thread::Yield ()的作用是若就绪队列中就绪进程，则将当前正在执行的线程进入就绪队列尾，然后调度队首的线程执行；

如果当前就绪队列为空，没有可以调度的线程，当前线程继续执行；

Thread::Yield ()与 Thread::Sleep()类似，区别在于：

（1）前者是线程由执行态转到就绪态，后者是由执行态转到睡眠（等待、阻塞）状态；

（2）对于 Thread::Yield ()，若就绪队列为空，没有可以调度的线程，当前线程继续执行（因为自己还具备继续执行的条件）；

而对于 Thread::Sleep()，若当前线程进入睡眠状态，该线程将无法继续执行，若目前就绪队列为空，CPU 将闲置，就循环调用 Interrupt::Idle()处理到期的中断，直到就绪队列不空；

如果中断请求队列中所有中断都处理完毕后（Timer 中断除外），仍然没有就绪线程可以调度，系统就真正地无事可做，就停机退出。

6、线程终止

尽管 Nachos 没有显式地定义线程的终止状态，但一个线程被终止后不能立即被撤销；此时线程所处的状态相当于教材中进程状态图中的 terminated 状态，线程终止但尚未撤销；

threadToBeDestroyed 存放执行结束的线程，然后调用 Sleep()将其状态由执行状态转换为阻塞状态，然后因此线程调度。代码如下：

注 1：在 Fork()中创建一个线程时，已经设置当线程结束时自动调用 Thread::Finish()，当然也可以在合适位置显式调用 Thread::Finish ()以终止一个线程；

注 2：这里只是将要撤销的线程放入 threadToBeDestroyed 中，等待系统将该线程的上下文切换到被调度线程的上下文时，才正式予以撤销；

这是因为此时 currentThread 仍然是将要撤销的线程，还在该线程的上下文中运行（还在其栈上运行），只有当切换到新调度的线程上下文后，currentThread 才发生了改变，即将终止的线程才可以予以真正撤销（释放为其所分配的栈）；

```
void Thread::Finish ()
{
    (void) interrupt->SetLevel(IntOff);
    ASSERT(this == currentThread);

    DEBUG('t', "Finishing thread \"%s\"\n", getName());

    threadToBeDestroyed = currentThread;
    Sleep();                    // invokes SWITCH
    // not reached
}
```

7、线程撤销

撤销一个终止的线程，就是释放为其分配的栈空间；

只有当一个线程不是 currentThread 时，也就是一个线程不在其栈上运行时，才可以予以真正撤销（释放栈）；

因此撤销线程的下述函数只能在终止线程的上下文切换到其它线程的上下文后才可以调用；最合适的位置就是线程调度程序中上下文发生切换后的位置；（参见../threads/scheduler.cc 中 Scheduler::Run()）

```
Thread::~Thread()
{
    DEBUG('t', "Deleting thread \"%s\"\n", name);

    ASSERT(this != currentThread);
    if (stack != NULL)
        DeallocBoundedArray((char *) stack, StackSize * sizeof(_int));
}
```

8、线程调度

线程调度程序按一定的策略，动态地把处理机分配给处于就绪队列中的某一个线程，以使之执行。

Nachos 默认的线程调度算法采用的是 FCFS，携带参数-rs 运行可实现“时间片+FCFS”的抢先式调度，即 RR 调度算法；

函数 Scheduler::FindNextToRun ()负责按照线程调度算法从就绪队列中选中一个符合条件的线程，由函数 Scheduler::Run ()将 CPU 分配给选中的线程；代码如下：

```
Thread * Scheduler::FindNextToRun ()
{
    return (Thread *)readyList->Remove();
}
```

由上述代码可以看出，首先调度的是队首线程，因此采用的是 FCFS 调度算法；

指派程序（dispatcher）代码如下：

```
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

    SWITCH(oldThread, nextThread);

     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();
    }
#endif
}
```

（1）当该线程没有关联到用户进程时（space==NULL），线程的上下文由 UNIX 管理（编译器），这里没有显式地保存于恢复线程的上下文；

（2）当该线程关联到用户进程时（space!=NULL），说明该线程执行的是用户应用程序，因此需要保存于恢复应用程序的上下文（更底层的上下文还是由 UNIX 管理）；

问题：

（1）就绪线程何时被调度执行？

回顾操作系统教材中引起调度的 4 种时机，可以发现：

（a）正在执行的 Nachos 线程执行 Thread::Yield()，引起线程调度；（执行→就绪）

（b）运行 nachos 时附带参数-rs，则定时器中断会间隔一定时间执行 Thread::Yield()，导致时间片轮转；（执行→就绪）

（c）当前线程进入睡眠（Sleep()），引起线程调度；（执行→阻塞）

（d）线程终止（Finish()），引起线程调度（Finish()→Sleep()）；（执行→终止）

（2）当就绪队列为空，新建的线程或被唤醒的线程进入就绪队列后，何时被调度执行？

当就绪队列为空，若此时触发调度，系统会执行 Interrupt::Idle()，当 Interrupt::Idle() 处理完所有到期的中断返回后，系统调度执行就绪线程；（见 Thread::Sleep()）

目前 Semaphore::P()与 Thread::Finish()调用了 Thread::Sleep()，你也可以在合适位置根据自己的需要调用 Thread:: Sleep ()（如在 Join()系统调用中）；

（3）新建线程从哪里开始执行？

ThreadRoot()是每个线程的入口地址（main 线程除外）；

（4）线程何时被撤销？

当终止一个线程，该线程还在其上下文中运行（尽管没有执行具体代码），因此只有当由该线程的上下文切换到其它线程的上下文时，该线程才实际停止执行，这时可以撤销该线程；

9、上下文切换

参见../threads/switch-linux.s;

1.2.2 信号量

Nachos 实现了信号量及相应的 P()、V()操作，用于实现进程或线程之间的同步；

Nachos 对信号量的实现与教材中介绍的信号量稍有区别，这里信号量的值不能为负，最小为 0。

1、信号量及其 P()、V()操作

参见../threads/synch.h 及 synch.cc;

Semaphore 类的构造函数对信号量进行命名，给信号量赋初值，并建立一个该信号量的等待队列，代码如下：

```
Semaphore::Semaphore(char* debugName, int initialValue)
{
    name = debugName;
    value = initialValue;
    queue = new List;
}
```

析构函数如下：

```
Semaphore::~Semaphore()
{
    delete queue;
}
```

P()操作：

```
void Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);      // disable interrupts

    while (value == 0) {                // semaphore not available
        queue->Append((void *)currentThread);     // so go to sleep
        currentThread->Sleep();
    }
    value--;                            // semaphore available, consume its value

    (void) interrupt->SetLevel(oldLevel);    // re-enable interrupts
}
```

可以看出，利用关中断保证 P()操作的原子性；

每调用一次 P()操作，相当于要为执行该 P()的线程分配一个资源；

当信号量值为 0，说明该信号量所对应的资源已经全部使用，没有空闲可用资源，申请该资源的线程进入等待队列，然后睡眠；否则，给线程分配一个资源，可用资源数减 1；

V()操作：

```
void Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    if (thread != NULL)    // make thread ready, consuming the V immediately
        scheduler->ReadyToRun(thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

每调用一次 V()操作，相当于释放一个资源，可用资源数加 1；如果有等待使用该资源的线程，唤醒队首的线程，将其进入就绪队列；

对照 P()操作的实现可以看出，若有等待该资源的线程，被唤醒后要使用该资源，会立即将可用资源数减 1，相当于将该资源分配给了被唤醒的线程；

2、信号量的使用

参照../monitor/prodcons++.cc 中对信号量的使用，在生产者-消费者问题中，定义三个信号量 mutex、nempty 及 nfull 实现生产者、消费者对缓冲区的互斥及同步；

初值分别为 1、缓冲区大小、0，然后在合适的位置执行相应的 P()与 V()操作；

```
#define BUFF_SIZE 4
Semaphore *mutex, *nempty, *nfull;
mutex = new Semaphore("mutex", 1);
nempty = new Semaphore("nempty", BUFF_SIZE);
nfull = new Semaphore("nfull", 0);
…..
nempty.P();
mutex,P();
…..
mutex,V();
nfull.V();
```

1.2.3 系统调用

Nachos 的系统调用主要是为用户提供服务，详细信息参见../userprog/syscall.h；

../userprog/syscall.h 给出了 Nachos 系统调用的声明，../test/start.s 系统调用的入口，供编译链接使用；

1、停机系统调用

（4）void Halt();

停机，关闭 Nachos。参见../test/halt.c

2、进程管理的系统调用

主要有如 Exec()，Exit()，Join()及 Halt()，其中

（1）SpaceId Exec(char *name);

为 Nachos 应用程序 name 创建一个进程，并返回系统为其所分配的内存空间的一个标识，即 pid；

../userprog/syscall.h 定义了一个数据类型 SpaceId（typedef int SpaceId;）来标识为应用程序所分配的内存空间，但在 progtest.cc 及 AddrSpace:: AddrSpace()中为应用程序创建进程及分配内存空间时，并没有为该进程建立任何的进程标识，包括内存空间标识 SpaceId；

为了实现像 UNIX 中创建一个进程时返回进程的 pid，需要你自己定义并给出进

程标识 SpaceId 或 pid，把 0~99 预留给系统进程（尽管目前没有系统进程），100 开始依次分配给应用程序进程；

当进程调用 Exit()退出时，应释放相应的 pid，以便将该进程号分配给新建进程；

注：理论上，应该是父进程创建一个子进程，子进程执行执行 file，相当于将 UNIX 的系统调用 fork()与 exec()组合到一起；

但目前 Nachos 中没有为进程建立如进程树那样的结构来表示进程之间的家族关系，也就没有显式地称谓父进程与子进程；

（2）void Exit(int status);

终止 Nachos 应用进程的执行，退出状态是 status；类似于 UNIX 的 exit();

Exit(0)需要释放进程的内存空间、页表、pid 等资源，然后终止与进程关联的核心线程；

（3）int Join(SpaceId id);

调用 Join(pid)的进程等待进程 pid 结束，并返回 pid 的退出码；类似于 UNIX 的 wait()，或 pthread 的 pthread_join(tid)；

3、文件管理系统调用

因时间关系，这些系统调用选做，在此不再赘述。参见../userprog/syscall.h；

```
typedef int OpenFileId;
void Create(char *name);
OpenFileId Open(char *name);
void Write(char *buffer, int size, OpenFileId id);
int Read(char *buffer, int size, OpenFileId id);
void Close(OpenFileId id);
```

4、用户线程系统调用

用户线程的创建等，这些用户线程共享用户进程的地址空间。

因时间关系，这些系统调用选做，在此不再赘述。参见../userprog/syscall.h；

```
void Fork(void (*func)());
void Yield();
```

目前 Nachos 只实现了停机的系统调用 Halt()，其它的系统调用需要你自己实现；因时间关系，目前只要求实现 Exec()与 Exit()。

5、Nachos 应用程序与系统调用的使用

可以在 Nachos 的应用程序中使用这些 Nachos 的系统调用。

目前在../test 目录中提供了几个 Nachos 应用程序实例，如 halt.c，sort.c，matmult.c 及 shell.c 等；

在../test 目录中运行 make，会将这些 Nachos 应用程序（.c 文件）编译并转换成 Nachos 可执行程序（.noff 格式），在../userprog 目录下运行 nachos –x ../test/halt.noff 可执行应用程序 halt.noff（由 halt.c 编译、转换而来）。

应用程序 halt.c 只使用系统调用 Halt()，而在../userprog/exception.cc 中已经实现了系统调用 Halt()，因此 Halt.c 对应的可执行程序可以正常执行；

sort.c，matmult.c 使用了 Exit()，但目前 Exit()系统调用尚未实现，因此这些应用程序无法正常执行；

由于 shell.c 使用了 Write()、Read()、Exec()及 Join()，因此在这些系统调用实现之前，shell.c 对应的可执行程序 shell.noff 无法正常执行。

6、编写自己的 Nachos 应用程序

在实现了某个（些）系统调用后，可以自己编写相应的测试程序，如 exec.c，write.c、等，在../test/Makefile 文件中的语句 targets = halt shell matmult sort 中添加上你的程序名，如 targets = halt shell matmult sort exec write，然后运行 make，会将你的.c 应用程序编译转换成相应.noff 可执行文件，就可以使用命令 nachos –x xxxx.noff 运行之；

7、关于 Nachos 应用程序语法

Nachos 的应用程序以.c 为扩展名，因此其语法也类似于 C 语言的语法，但由于编译链接使用的编译器不是标准的 C 编译器（gcc 或 g++），而是 Nachos 的作者自己编写的一个功能有限的编译器，该编译器可以根据 syscall.h 识别 Nachos 系统调用，根据 syscall.s 将含有系统调用的程序链接成可执行程序，但基本不支持 C 语言标准函数及一些数据操作方式，如不支持 printf()，不支持形如 char fileName[]=”test.txt”的方式对字符数组赋值；对于字符数组赋值只能通过如下方式：char fileName[20]; fileName[0]=‘t’, fileName[1]=‘x’,以此类推；

更多的关于 Nachos 应用程序的语法现象可参考../test 目录下的几个 Nachos 应用程序实例。

8、运行 Nachos 应用程序

命令 nachos –x halt.noff 使 Nachos 加载运行应用程序 halt.noff，为其创建进程，分配地址空间，建立用户线程与核心线程的映射等；参见../userprog/progtest.cc

1.2.4 应用程序进程

目前 Nachos 只支持单进程机制，不支持多进程机制；实现了 Exec()后，当主进程调用 Exec(file)，就为 Nachos 可执行程序 file 创建一个（子）进程，与主进程并发执行，可以认为实现了多进程机制；

目前 Nachos 也不支持用户多线程机制，系统调用 Fork()及 Yield()实现后可支持用户多线程；

理论上讲，系统应该为进程创建一个 PCB，为其代码、数据、栈、堆分配相应的地址空间；

但目前 Nachos 系统为应用程序创建的进程没有显式地为其创建 PCB，PCB 中包括的进程的属性分散到几个相关的类对象中（如 AddrSapce 类、Thread 类），地址空间中只有代码、数据及栈，不包含堆（由于 Nachos 的应用程序功能比较简单，主要是为了测试系统调用）。

Nachos 为应用程序分配地址空间后，创建相应的页表；

目前 Nachos 不支持用户多线程，因此可以认为进程中只有一个线程，将该线程映射到一个核心线程以执行该进程；

然后初始化 CPU 寄存器（特别是 PC 寄存器），将进程页表传递给内核中的系统页表，开始进程的执行；

1.2.5 内存管理

Nachos 采用一个字符数组模拟了主存储器，内存管理采用分页管理方式，采用页表或 TLB 实现虚页与实页（帧）的映射，可以采用位示图对空闲帧进行管理；

每个帧大小与一个硬盘块的字节数相等，每个硬盘块对应一个扇区，大小为 128 字节，默认有 32 个帧。（参见../machine/machine.h）

可以通过 Machine::ReadMem(…)与 Machine::WriteMem(…)实现内存的读写功能，可以读写 1 个字节、2 个字节及 4 个字节。（参见../machne/translate.cc）

可以通过 Machine::ReadRegister(…)与 Machine::WriteRegister(…)实现寄存器的读写功能。（参见../machne/machine.cc）

AddrSpace 的构造方法为应用程序分配内存，建立页表，将应用程序代码及数据读入内存，建立虚页与实页的对应关系（参见 addrspace.cc）

关于页表结构参见../machine/translate.h，虚实地址的变换参见 translate.cc 中 Machine::Translate(…)，页表的创建与使用参见../userprog/progtest.cc 及 addrspace.cc；

每个应用程序进程维护一个进程页表，内核维护一个系统页表，系统页表在 machine.h 中声明，在 translate.cc 中使用，参见 tanslate.cc 与 Machine::Translate(…)；

1.2.6 文件系统

Nachos 的硬盘大小 128KB，每个硬盘块包含一个扇区，大小为 128bytes；

每个文件由文件头+数据块组成；文件头占用一个扇区；

采用位管理示图空闲块，位示图数据块大小为 128 字节，占用一个扇区；

采用一级目录管理，目录项采用名号目录结构，最多可创建 10 个文件；目录文件大小为 200 字节，占用两个扇区；

每个文件数据最多由 30 个硬盘块组成，最大为 3KB；文件数据块采用索引方式分配；

1、硬盘

Nachos 利用一个 UNIX 文件模拟了 Nachos 的硬盘，默认的硬盘参数为：32 个磁道，每道包括 32 个扇区，每个扇区 128 字节，因此硬盘大小为 128KB；

通常情况下，一个硬盘的逻辑块包括若干个扇区，Nachos 中一个硬盘块对应一个扇区；（参见../machine/disk.h）

Nachos 的硬盘标识（魔数）为 0x456789ab，位于硬盘的前 4 个字节中。（参见../machine/disk.cc）

2、硬盘格式化（创建文件系统）

在../filesys 目录中编译生成的 Nachos 系统启动时，创建一个空盘 DISK，只是在文件 DISK 的开始 4 个字节写入其标识 0x456789ab，在文件的 128KB+4 位置写入 0，使硬盘大小为 128KB。

nachos –f 格式化该硬盘，在其上创建了一个文件系统，在 0 号扇区创建了硬盘空闲块管理所使用的位示图文件的文件头，在 1 号扇区创建了目录文件的文件头，在 2 号扇区存储位示图文件数据块，在 3、4 号扇区存储目录表；

有关内容参见../filesys/directory.cc，directory.cc、flehdr.cc 及 bitmap.cc；

因此在 2 号扇区中存储的位示图文件中，已经标注扇区 0~4 已经占用（即位示图文件的内容为 1111000…0）；

将硬盘空闲块位示图的文件头与文件目录表的文件头存放在 0 号与 1 号这两个特殊的扇区中，便于系统启动时方便访问这两个特殊文件。

注：为文件头即为我们所熟悉的 FCB 或 i-node，在以后的描述中，这三个术语不加区分；

3、空闲块的管理

文件系统中的硬盘空闲块管理采用位示图的方式，0 表示对应的硬盘块空闲，1 表示对应的硬盘块已经分配；（参见../filesys/filesys.cc 中 FileSystem:: FileSystem()）

4、目录管理

采用一级目录（根目录）管理方法，目录项采用与 UNIX 类似的文件名+索引节点（FCB）组成。

一个目录项是一个三元组<文件名，i-node，目录项空闲标记>，如表 1-1 所示。

表 1-1 Nachos 目录表结构（200 字节）

文件名	是否已分配	文件头（FCB、索引结点）所在的扇区号

Nachos 的目录表默认包括 10 个目录项，由于它只有一级目录，因此，该文件系统最多可创建 10 个文件；（关于目录表及目录项有关内容参见../filesys/directory.cc）

新建一个文件时，需要为该文件在目录表中分配一个空闲的目录项，为文件分配一个 FCB（文件头、索引节点），然后在目录项中建立文件名与 FCB 的映射关系；

检索一个文件的过程就是根据文件名在目录表中找到该文件的 FCB，然后根据 FCB 获取该文件的详细信息；

删除一个文件的过程就是根据文件名将对应的目录项中的使用标记清除，使该目录项变为空闲，可以分配给其它文件；其中的文件名及 FCB 并不清除，文件头及文件的数据块也不清除，便于对删除文件的恢复；（但为文件头及数据块所分配的数据块也在空闲块位示图中设置为空闲标记，只是文件头及文件数据块中的内容保持不变）；

5、文件

一个文件由文件头（索引节点、FCB）+数据块组成，为文件数据所分配的硬盘

块采用索引分配方式管理，每个文件头（FCB，索引结点）包括文件的属性，包括<文件大小，硬盘块数、数据块索引表>三项，如表 1-2 所示。

表 1-2 文件头结构（128 字节）

文件大小	扇区块数	文件数据块所在的扇区列表[30]

Nachos 默认一个文件的文件名最长为 9 字节，每个文件的数据块最多由 30 个硬盘块（扇区）组成，因此一个文件最大为 3KB。（参见../filesys/filehdr.h）

1.2.7 虚存与网络管理

课设暂不涉及。

1.3 Nachos 的文件及目录组织

Nachos 文件目录结构如图 1-1 所示。

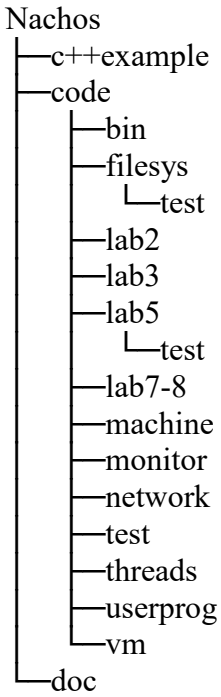


图1-1 Nachos文件目录

1.3.1 系统目录

（1）c++example：该目录包含几个 C++的示例程序，以及 Tom Anderson.教授编写的一本 C++入门参考资料(c++.ps)，其中介绍了 Nachos 源代码中所涉及到的一些 C++知识。在阅读 Nachos 源码时如果遇到关于 C++语言的问题时可参阅这些资料。如

果对 C++不是很熟悉，最好在阅读 Nachos 系统源码时先回顾一下这些知识，特别是要很好地理解一下示例程序 stack.cc。

（2）code/bin：用于将 Nachos 应用程序交叉编译成 Nachos 可执行文件（.noff）；为了测试 nachos 系统调用功能，需要编写一些使用 Nachos 系统调用的应用程序（.c），编译后在 nachos 上运行。

Nachos 模拟的 CPU 执行的 MIPS 架构的指令，因此需要将在 linux 环境下编写的 c 程序编译并转换成可在 nachos 环境下执行的程序（.noff）。转换程序为 coff2noff。

（3）code/filesys：nachos 的文件系统

- directory：文件目录
- filehdr：文件头（FCB、i-node）
- filesys：文件系统及其操作，包括硬盘格式化，创建、删除、打开等操作；
- fstest：文件系统测试程序，为 Nachos 应用程序创建进程并执行；
- openfile：文件操作，包括文件的读写等操作；

（4）code/machine: 模拟了机器的硬件，Nachos 作为一个操作系统运行在这些硬件之上。

- console:：Nachos 的控制台（键盘与显示器）
- disk：计算机硬盘
- interrupt：中断控制器
- machine：CPU
- mipssim：Nachos 应用程序中 MIPS 指令的执行过程
- network:：网卡
- stats：硬件工作的一些参数的模拟，如系统时钟，定时器间隔多长时间产生一次中断；磁盘的旋转时间与寻道时间应该多长等。
- sysdep:包含 nachos 用于系统管理的一些系统调用，包括文件的操作、进行的管理、网络的发送与接收等。Nachos 没有直接使用 linux 提供的系统调用，主要是为了便于 nachos 到其它平台的移植。
- timer:：定时器。
- Translate：模拟虚实地址的转换的转换过程

（5）code/monitor：实现 nachos 使用的锁、条件变量、信号量、管程的等同步机制。

（6）code/network: 实现网络的功能

（7）code/test:包含几个测试 nachos 系统调用的应用程序（基于 nachos 的 c 程序），可在 nachos 上运行（.noff 文件格式）

- start.s：各 Nachos 系统调用的入口；编译链接 Nachos 应用程序时使用；
- halt.c：示例 Nachos 应用程序编程方法与使用 Halt()系统调用；
- matmult.c，sort.c：示例 Nachos 应用程序编程方法与使用 Exit()系统调用；
- shell.c：示例 Nachos 应用程序编程方法与使用 Read()、Write()、Exec()及 Join()系统调用；

  如果实现了这几个系统调用，该程序可作为 Nachos 的 shell；

（8）code/threads:线程的管理，包括线程的创建、睡眠、终止、调度，以及信号

量等功能。

- list：Nachos 使用的队列，包括就绪队列、等待队列等；
- main.cc：Nachos 的主函数；
- scheduler：线程调度程序；
- switch-linux：线程的上下文切换；
- synch：锁机制、条件变量、信号量等；
- synchtest.cc：锁、条件变量测试程序，示例如何使用锁、条件变量等；
- system：系统程序，包括系统初始化，声明一个全局变量等；
- thread：线程的创建、睡眠、终止等操作；
- threadtest.cc：线程并发测试程序，示例如何创建线程；
- utility：Nachos 的工具软件

（9）code/userprog: Nachos 应用进程的管理，加载一个 Nachos 应用程序，创建相应的进程，将进程映射到一个核心线程，然后运行。

- addrspace：为应用程序分配内存地址空间；
- bitmap：位示图，管理内存空闲帧，管理硬盘空闲块；
- exception：Nachos 的系统调用；
- progtest.cc：应用进程及控制台测试程序，示例如何为应用程序创建进程并启动该进程
- syscall.h：声明 Nachos 系统调用接口原型；

（10）code/vm:虚拟存储管理

1.3.2 用户实验目录

lab2、lab3、lab5、lab7-8 是我们课程设计所使用的工作目录。

为了完成设计任务。我们需要对一些程序做相应的修改，需要修改的文件就将其复制到相应的 lab 目录中，与其它程序区别开来。

1.4 Nachos 基本内核的启动过程

Nachos 启动过程的主要工作流程：

处理命令行参数，根据需要初始化硬件设备，创建主线程，运行测试程序等；

../threads/main.cc 中的 main()函数是 Nachos 的入口；

1、调用../threads/system.cc 中的 Initialize(argc, argv)初始化内核

（1）处理核初始化内使用的一些命令行参数，如-d, -rs, -f, -s 等；

（2）初始化系统统计数据，如寻道时间、系统开始计时等；

```
stats = new Statistics();
```

（3）初始化中断控制器；

```
interrupt = new Interrupt;
```

（4）初始化调度程序，创建就绪队列；

```
scheduler = new Scheduler();
```

（5）如果运行时携带参数-rs，初始化定时器，实现时间片抢先调度

```
if (randomYield)    // start the timer (if needed)
```

```
timer = new Timer(TimerInterruptHandler, 0, randomYield);
```

（6）创建主线程 main，并作为当前运行的线程；

```
currentThread = new Thread("main");
currentThread->setStatus(RUNNING);
```

（7）允许相应中断

```
interrupt->Enable();
```

（8）如果在执行过程中，按下 ctrl+c，则清除所有设备，退出 Nachos；

CallOnUserAbort(Cleanup)→ (void)signal(SIGINT, Cleanup);

（9）对于实验 6、7、8，初始化 CPU；

```
#ifdef USER_PROGRAM
    machine = new Machine(debugUserProg);     // this must come first
#endif
```

（10）对于实验 4、5，初始化硬盘与文件系统；

```
#ifdef FILESYS
  synchDisk = new SynchDisk("DISK");
#endif

#ifdef FILESYS_NEEDED
    fileSystem = new FileSystem(format);
#endif
```

2、如果需要，运行线程测试程序与同步测试程序；

```
#ifdef THREADS
    ThreadTest();   //测试线程创建及多线程并发交替执行过程
#if 0
    SynchTest();    //测试锁机制及条件变量
#endif
#endif
```

3、处理其它的命令行参数，如-z, -x, -cp, -p, -r, -l, -c, -D, -t 等；

4、终止主线程

```
currentThread->Finish();
```

注：由于 Finish()→Sleep()，因此终止主线程，可调度执行就绪进程；当所有就绪线程执行结束，且无中断要处理，才退出 Nachos。

5、退出 Nachos

注：如果运行 Nachos 时加载运行 Nachos 的 shell 程序（nachos –x ../test/shell.noff），此时 Nachos 不会退出，主线程执行 shell 程序，shell 中循环等待用户输入命令并执行用户输入的命令，直到用户输入停机命令退出；

运行 shell 的前提是实现了系统调用 Read()、Write()与 Jion()；

参见 ../test/shell.cc ，及 ../userprog/progtest.cc

一般的操作系统中，还有一个特殊的进程 Idle，当 CPU 空闲且就绪队列为空时，系统会调度执行进程 Idle，操作系统不会退出。

1.5 相关软件及参考资料

1.5.1 相关软件

32 位 Linux 系统（如 ubuntu）；
Nachos-3.4 源代码（nachos-3.4.tar.gz）
gcc-2.8.1-mips 交叉编译（gcc-2.8.1-mips.tar.gz）

1.5.2 参考资料

nachos_introduction.pdf；
nachos_study_book.pdf；
a road map through nachos.pdf；
C++编程相关资料；
gcc，g++使用手册；
MIPS 汇编语言；
i386 汇编语言；
gdb 使用手册；
make 文件使用手册；

第 2 章 Nachos 系统的安装与调试（实验 1）

2.1 目的与任务

2.1.1 目的

（1）安装编译 Nachos 系统，理解 Nachos 系统的组织结构，熟悉 C++编程语言；

（2）安装测试 gcc MIPS 交叉编译器；

（3）掌握利用 Linux 调试工具 GDB 调试跟踪 Nachos 的执行过程；

（4）通过跟踪 Nachos 的 C++程序及汇编代码，理解 Nachos 中线程上下文切换的过程。

（5）阅读 Nachos 的相关源代码，理解 Nachos 内核的工作原理及其测试过程。

2.1.2 任务

（1） 安装 Linux 操作系统；

（2） 安装 Nachos 及 gcc mips 交叉编译程序；

（3） 编译测试 Nachos；

（4） 熟悉 gdb 调试工具；

（5） 熟悉 Nachos 中的上下文切换过程；

2.2 安装 Nachos

Nachos 3.4 运行在 32 位 Linux 环境下。我们可以把它移植到 64 位环境中运行。首先需要对 64 位的 Linux 环境进行设置。

2.2.1 64 位 Linux 环境设置

操作过程如下（以 64 位 Ubuntu 为例）：

1. 检查系统是否支持多架构

（1）确认主机系统为 64 位架构的内核

在命令终端中运行 dpkg --print-architecture，应该输出 amd64

（2）确认系统多架构功能已经打开，以支持 i386

在命令终端中运行 dpkg --print-foreign-architectures，应该输出 i386

（3）如果（2）中检测到多架构功能尚未打开，即没有输出 i386，则使用下述命令打开 sudo dpkg --add-architecture i386

2、安装 32 位编译环境与支持库

（1）检查 gcc 与 g++是否已经安装，如果尚未安装，使用如下命令安装它们。

sudo apt-get install gcc 与 sudo apt-get install g++

（2）安装 32 位编译环境与支持库

sudo apt-get install build-essential g++-multilib gcc-multilib

3、修改 Nachos

(1) 修改 code 目录下的 Makefile.dep 文件

在 C++的编译器 CC 与链接器 LD 后追加 -m32，在汇编编译器 AS 后追加 --32，修改后的内容如下所示。

```
.....
ifndef MAKEFILE_DEP
define MAKEFILE_DEP
yes
endef

# These definitions may change as the software is updated.
# Some of them are also system dependent
CPP=/lib/cpp
CC = g++  -m32
LD = g++  -m32
AS = as  --32

uname = $(shell uname)

mips_arch = dec-mips-ultrix
.....
```

（2）修改 code/bin/coff.h

文件 code/bin/coff.h 中的几条语句如下所示：

```
/* coff.h
 *     Data structures that describe the MIPS COFF format.
 */

#ifdef HOST_ALPHA       /* Needed because of gcc uses 64 bit long    */
#define _long int       /* integers on the DEC ALPHA architecture. */
#else
#define _long long    //修改为#define _long int
#endif

struct filehdr {
    unsigned short   f_magic;          /* magic number */
    unsigned short   f_nscns;          /* number of sections */
    _long            f_timdat;         /* time & date stamp */
    _long            f_symptr;         /* file pointer to symbolic header */
    _long            f_nsyms;          /* sizeof(symbolic hdr) */
    unsigned short   f_opthdr;         /* sizeof(optional hdr) */
```

```
        unsigned short    f_flags;          /* flags */
    };
.....
```

在 32 位环境中，宏命令#define _long long 将_long 定义为 long，系统为 long 类型数据分配 4 字节存储空间，因此结构 filehdr 的大小为 20 字节，而在 64 位系统中，由于数据类型 long 需要 8 字节存储空间，导致 filehdr 占用了 40 字节。而 Nachos 中要求 filehdr 大小必须为 20 字节，因此需要将 code/bin/coff.h 中的 long 改成 int，即将#define _long long 修改为#define _long int。

2.2.2 安装 Nachos

Nachos 安装步骤如下：

方法 1：基于终端命令

```
(a) cd ~
(b) mkdir OS    （注：目录名 OS 可根据你的喜好自行修改）
(c) cd OS
(d) copy file   “Nachos-3.4.tar.gz” to file fold “OS” with cp command.
(e) tar xzvf Nachos-3.4.tar.gz
(f) rm Nachos-3.4.tar.gz   (可选)
```

方法 2：基于 Linux 图形界面

(1) 进入你的用户工作目录

(2)在用户工作目录下，点击鼠标右键，根据提示创建一个用于“操作系统课程设计”的目录，如 OS （注：目录名 OS 可根据你的喜好自行修改）

(3) 进入目录 OS

(4) 将 Nachos 的压缩包 Nachos-3.4.tar.gz 复制到 OS 目录下

(5) 右键点击文件 Nachos-3.4-.tar.gz，根据提示解压缩该文件

(6) (该步骤可选) 删除目录下 OS 下的压缩包 Nachos-3.4.tar.gz

基于上述步骤安装 Nachos 系统后，目录$home/OS 下会产生一个新的目录 nachos-3.4，里面含有 Nachos 系统的源代码等相关资料。

2.3 gcc MIPS 交叉编译器的安装与测试

2.3.1 安装 gcc MIPS 交叉编译器

需要将 gcc MIPS 交叉编译器安装在/usr/local 目录下，因此安装时需要 root 用户的权限以访问目录/usr/local。

基于终端命令

```
(a) su
```

```
(b) cd /usr/local
(c) copy file “gcc-2.8.1-mips.tar.gz” to file fold “/usr/local” with cp command.
(d) tar xzvf gcc-2.8.1-mips.tar.gz
(e) rm gcc-2.8.1-mips.tar.gz
```

参照上述步骤完成安装后，gcc MIPS 交叉编译器包含在目录/usr/local/mips 中。

2.3.2 测试 gcc MIPS 交叉编译器

进入 Nachos 的目录 code/test（简记为../test），

（1） 删除../test/arch/unknown-i386-linux/depends 目录下的所有文件；

（2） 删除../test/arch/unknown-i386-linux/objects 目录下的所有文件

（3） 删除../test 目录下的所有扩展名为.noff 的文件

（4） 运行../test /make，如果交叉编译器安装成功，编译过程中没有出现错误信息，../test 目录下几个.c 文件都会产生一个对应的.noff 文件，同时屏幕应输出下述信息：（如果没有错误，输出信息可能稍有区别）

```
>>> Linking arch/unknown-i386-linux/objects/halt.coff <<<
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/halt.o -o arch/unknown-i386-linux/objects/halt.coff
>>> Converting to noff file: arch/unknown-i386-linux/bin/halt <<<
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/halt.coff arch/unknown-i386-linux/bin/halt
numsections 3
Loading 3 sections:
    ".text", filepos 0xd0, mempos 0x0, size 0x100
    ".data", filepos 0x1d0, mempos 0x100, size 0x0
    ".bss", filepos 0x0, mempos 0x100, size 0x0
ln -sf arch/unknown-i386-linux/bin/halt halt
>>> Linking arch/unknown-i386-linux/objects/shell.coff <<<
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/shell.o -o arch/unknown-i386-linux/objects/shell.coff
>>> Converting to noff file: arch/unknown-i386-linux/bin/shell <<<
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/shell.coff arch/unknown-i386-linux/bin/shell
numsections 3
Loading 3 sections:
    ".text", filepos 0xd0, mempos 0x0, size 0x200
    ".data", filepos 0x2d0, mempos 0x200, size 0x0
    ".bss", filepos 0x0, mempos 0x200, size 0x0
ln -sf arch/unknown-i386-linux/bin/shell shell
>>> Linking arch/unknown-i386-linux/objects/matmult.coff <<<
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/matmult.o -o arch/unknown-i386-linux/objects/matmult.coff
>>> Converting to noff file: arch/unknown-i386-linux/bin/matmult <<<
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/matmult.coff arch/unknown-i386-linux/bin/matmult
```

```
numsections 3
Loading 3 sections:
    ".text", filepos 0xd0, mempos 0x0, size 0x3c0
    ".data", filepos 0x490, mempos 0x3c0, size 0x0
    ".bss", filepos 0x0, mempos 0x3c0, size 0x12c0
ln -sf arch/unknown-i386-linux/bin/matmult matmult
>>> Linking arch/unknown-i386-linux/objects/sort.coff <<<
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o
arch/unknown-i386-linux/objects/sort.o -o arch/unknown-i386-linux/objects/sort.coff
>>> Converting to noff file: arch/unknown-i386-linux/bin/sort <<<
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/sort.coff
arch/unknown-i386-linux/bin/sort
numsections 3
Loading 3 sections:
    ".text", filepos 0xd0, mempos 0x0, size 0x2c0
    ".data", filepos 0x390, mempos 0x2c0, size 0x0
    ".bss", filepos 0x0, mempos 0x2c0, size 0x1000
ln -sf arch/unknown-i386-linux/bin/sort sort
```

思考：为什么 nachos-3.4.tar.gz 一定要安装在/usr/local 目录中？

打开 code/Makefile.dep，在大约 38 行左右，查看变量 GCCDIR 的值，会得到答案，GCCDIR = /usr/local/mips/bin/decstation-ultrix-

交叉编译器用于对../test 目录下的 Nachos 应用程序（如 sort.c）进行编译，经转换后会生成 Nachos 可执行的文件 sort.noff：

.noff 可执行文件中的指令基于 MIPS 架构，Nachos 模拟的 CPU 执行 MIPS 架构的指令；

2.4 测试 Nachos

Nachos 3.4 要求运行在 32 位 Linux 环境中。如果在 64 位 Linux 环境下运行 Nachos 3.4，需要安装 32 位的编译环境与支持库，并对 Nachos 3.4 进行少量的修改。

2.4.1 Nachos 的基本内核测试

参照以下步骤测试安装的 Nachos（Nachos 的基本内核）是否正常工作：

1、进入子目录 ~/OS /nachos-3.4/

该目录包含三个子目录：c++example、code 及 doc

2、进入子目录 code

该子目录下包含以下文件及目录：

```
Makefile.common    ass2/    bin/    lab2/    lab5/    machine/ test/        userprog/
Makefile.dep       ass3/    filesys/ lab3/    lab7-8/    network/ threads/    vm/
```

3、 在目录 code/threads 下测试 nachos 的基本内核

在终端窗口下进入目录 code/threads/，运行 make 命令，可编译生成一个基本的 Nachos 内核（多线程）。编译时如果屏幕输出的最后几行信息如下：

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

表示已经成功编译生成了一个最小的 Nachos 内核。目录 threads/下的文件 nachos 是一链接到可执行程序 arch/unknown-i386-linux/bin/nachos 的链接文件。

4、 运行 Nachos。

在终端窗口下，在目录 code/threads 下输入./nachos，屏幕应该输出：

```
*** 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...
```

分析输出结果：

从主程序./threads/main.cc 中的 main()函数开始考察 Nachos 的启动与退出过程：

（1）初始化 Nachos 的设备与内核

调用 Initialize(int argc, char **argv) （参见../threads/system.cc）

- 处理 Nachos 内核所使用的一些命令行参数；
- 根据需要创建了 Nachos 相应的硬件设备，如中断控制器，定时器、CPU、硬盘；
- 然后基于这些设备初始化了一个 Nachos 的内核；，如初始化了一个线程

调度程序，在硬盘上创建了文件系统，创建了 Nachos 的主线程（Nachos 的第一个线程），以及网络通信使用的邮箱；

- 讨论：mian 线程创建后，后续的代码是否在主线程中执行？

（2）对 Nachos 内核进行测试（线程的创建及并发执行）

调用 ThreadTest()测试内核是否工作正常（参见../threads/thresdtest.cc）

上述输出结果主要是该函数的执行结果；

- 主线程”main”创建了一个子线程“forked thread”，主线程执行 SimpleThread(0)，子线程执行 SimpleThread(1)，由于 SimpleThread()调用了 Thread::Yield()，因此两个线程会交替执行，输出上述结果；

（3）测试 Nachos 的锁机制及条件变量

你也可以执行 SynchTest()以测试 Nachos 的 locks 及 condition variables 实现的线程之间进行互斥与同步功能是否工作正常；

（4）处理 Nachos 其它的一些命令行参数

如显式版权信息、运行应用程序、文件系统及网络相关的一些运行参数；

（5）终止主线程，Nachos 退出

main()函数退出之前，执行 currentThread->Finish()终止主线程，我们注意到，Thread::Finish()调用了 Thread::Sleep()，引起线程调度，如果此时就绪队列中尚有就绪进程，则调度执行之；当该就绪线程执行结束后，也会自动执行 Thread::Finish()，致使所有的就绪线程都会被依次调度执行；

如果主线程执行 Thread::Finish()时就绪队列为空，或者最后一个就绪线程执行结束后终止，都会调用 Thread::Sleep()，进而循环调用 Interrupt:Idle()，当所有的中断请求都被处理完后，依然没有就绪线程等待调度，则 Interrupt:Idle()调用 Interrupt:Halt()关闭退出 Nachos。

Interrupt:Halt()调用 Cleanup()，将启动时创建的设备（中断控制器、定时器、硬盘、CPU），及文件系统、调度程序等一并删除后退出，至此 Nachos 运行结束。（参见../threads/system.cc）

Nachos 的启动过程简单总结如下：

Nachos 的启动入口是./threads/main.cc 中的 main()，完成的主要工作如下：

（1）调用../threads/system.cc 中的 Initialize(argc, argv)初始化内核

（a）处理核初始化内使用的一些命令行参数，如-d, -rs, -f, -s 等；

（b）初始化系统统计数据，如寻道时间、系统开始计时等；

```
stats = new Statistics();
```

（c）初始化中断控制器；

```
interrupt = new Interrupt;
```

（d）初始化调度程序，创建就绪队列；

```
scheduler = new Scheduler();
```

（e）如果运行时携带参数-rs，初始化定时器，实现时间片抢先调度

```
if (randomYield)   // start the timer (if needed)
        timer = new Timer(TimerInterruptHandler, 0, randomYield);
```

（f）创建主线程 main，并作为当前运行的线程；

```
currentThread = new Thread("main");
currentThread->setStatus(RUNNING);
```

（g）允许响应中断

```
interrupt->Enable();
```

（h）如果在执行过程中，按下 ctrl+c，则清除所有设备，退出 Nachos；

```
CallOnUserAbort(Cleanup)→  (void)signal(SIGINT, Cleanup);
```

（i）对于实验 6、7、8，初始化 CPU；

```
#ifdef USER_PROGRAM
    machine = new Machine(debugUserProg);      // this must come first
#endif
```

（j）对于实验 4、5，初始化硬盘与文件系统；

```
#ifdef FILESYS
    synchDisk = new SynchDisk("DISK");
#endif

#ifdef FILESYS_NEEDED
    fileSystem = new FileSystem(format);
#endif
```

（2）如果需要，运行线程测试程序与同步测试程序；

```
#ifdef THREADS
    ThreadTest();   //测试线程创建及多线程并发交替执行过程
                    //这里输出屏幕的运行结果
#if 0
    SynchTest();    //测试锁机制及条件变量
#endif
#endif
```

（3）处理其它的命令行参数，如-z, -x, -cp, -p, -r, -l, -c, -D, -t 等；

（4）终止主线程

```
currentThread->Finish();
```

注：由于 Finish()→Sleep()，因此终止主线程，可调度执行就绪进程；当所有就绪线程执行结束，且无中断要处理，才退出 Nachos。

5、退出 Nachos

2.4.2 测试其它模块的功能

由于文件按系统扩展（实验 5）、系统调用（实验 7、8）尚未完成，这些系统功能测试可能无法正常运行。

进入 code/ 目录下的其它子目录，如 code/filesys，如 code/ userprog，code/monitor 在相应目录的终端下运行 make 编译，可生成包含文件系统、用户程序（进程）、管程等功能的 Nachos 系统。输入./nachos 可运行具备相应功能的 Nachos 系统。

2.5 利用 gdb 调试 Nachos C++代码的过程与方法（C++与 gdb）

前面已经提到，Nachos 是用 C++语言实现的，在理解 Nachos 实现之前，建议参考目录 c++example/下的 c++.ps 把示例程序 stack.cc 很好地理解一下。

为了更好地理解 Nachos 的源码实现，在阅读器源码时可以借助于调试工具 gdb。

例如利用 gdb 调试 stack 的过程如下：

- 打开一个命令终端，进入目录 c++example，
- 运行 make 命令编译三个示例程序（这里我们以 stack 为例）
- 键入命令 gdb stack 在 gdb 中打开可执行程序 stack
  - 在 gdb 中会输出如下信息:

```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...done.
(gdb)
```

- 在 gdb 的提示符下，输入 gdb 命令 list（或 l），输出 stack 的前 10 行代码；继续输入 list（或 l），输出后续 10 行代码，以此类推；

- 可以看出地 129 行是 main 函数入口，因此，若在 main()的入口设置断点 break point，输入 break 129，或 break main，或（b 129 或 b main），gdb 输出：

```
(gdb) break 129
Breakpoint 1 at 0x8048a34: file stack.cc, line 129.
(gdb)
```

- 输入 run 或 r 运行 stack，程序将在设置的 break point 处停止：

```
(gdb) run
Starting program: /home/OS/nachos-3.4/c++example/stack

Breakpoint 1, main () at stack.cc:129
129    Stack *stack = new Stack(10);   // Constructor with an argument.
(gdb)
```

- 输入 next （或 n），可逐条跟踪语句的执行：

```
131 stack->SelfTest()
```

- 如果想跟踪方法 stack->SelfTest()的内部执行过程，可键入单步执行命令 step 或 s，gdb 则进入方法 stack->SelfTest()中执行该方法中的语句：

```
Stack::SelfTest (this=0x8050a10) at stack.cc:108
108     int count = 17;
(gdb)
```

- 持续键入 step 或 next， 可跟踪 stack 的执行过程。

- gdb 的 print 命令可输出程序中变量的值，如在合适的位置键入 print count，可输出此时变量 count 的值；

其它常用的 gdb 命令及 gdb 的使用可参阅 gdb 目录下的 gdb 参考资料。

2.6 Nachos 的上下文切换

Nachos 的 code/threads 目录中的主程序 main.cc 调用了函数 ThreadTest()，代码如下：

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

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

       t->Fork(SimpleThread, 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();
    }
}
```

设计任务：利用 gdb 工具跟踪 Nachos 的执行过程以及跟踪上下文切换函数 SWITCH()及函数 ThreadRoot()的执行过程。

熟悉跟踪过程后，回答下列问题：

（1）在你所生成的 Nachos 系统中，下述函数的地址是多少？并说明找到这些函数地址的过程及方法。

i. InterruptEnable()

ii. SimpleThread()

iii. ThreadFinish()

iv. ThreadRoot()

（2）下述线程对象的地址是多少？并说明找到这些对象地址的过程及方法。

i. the main thread of the Nachos

ii. the forked thread created by the main thread

（3）当主线程第一次运行 SWITCH()函数，执行到函数 SWITCH()的最后一条指令 ret 时，CPU 返回的地址是多少？ 该地址对应程序的什么位置？

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?

（4）当调用 Fork()新建的线程首次运行 SWITCH()函数时，当执行到函数 SWITCH()的最后一条指令 ret 时，CPU 返回的地址是多少？ 该地址对应程序的什么位置？

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?

2.7 课后作业

从 code/threads 目录下的 main.cc 开始，阅读、分析 Nachos 的.cc 源文件及相关的头文件，理解 Nachos 内核、线程的工作机理，

（1）Nachos 的命令行参数及其处理

（2）主线程（main）是如何创建的？

（3）我们应该如何创建线程；

（4）Nachos 是如何进行上下文切换的；

主要代码：

../threads/main.cc

../threads/system.cc（.h）

../threads/thread.cc（.h）

../threads/scheduler.cc（.h）

../threads/switch-linux.s

2.8 关于 2.6 的几点注记

在../threads 命令下运行 make，生成 nachos 系统；

gdb ./nachos 利用 gdb 调试 nachos；

2.8.1 关于函数的地址

gdb 利用命令 break（简写 b）可以在程序的某一位置设置断点，当程序执行到该

断点时暂停程序的执行（被设置断点的语句尚未执行）。

(gdb) list （1）显示程序源码，包括行号

(gdb) b x，（根据在 list 命令显示程序时所给出行号），在第 x 行设置断点；gdb 会输出这是第几个断点，同时输出该断点的地址；

(gdb) b function，在函数 function 的入口设置断点，并输出断点号及函数地址；（函数体尚未执行）

因此，可以设置函数 InterruptEnable()、SimpleThread() 、ThreadFinish()及ThreadRoot()的断点，输出的断点地址就是函数地址；如 InterruptEnable()函数的地址是 0x804a34c，以此类推。如图 2-1 所示。

```
(gdb)
(gdb) b InterruptEnable
Breakpoint 4 at 0x804a34c: file thread.cc, line 302.
(gdb) b SimpleThread
Breakpoint 5 at 0x804a505: file threadtest.cc, line 29.
(gdb) b ThreadFinish
Breakpoint 6 at 0x804a332: file thread.cc, line 301.
(gdb) b ThreadRoot
Breakpoint 7 at 0x804bb5a
(gdb)
```

图 2-1 断点及断点地址

特别指出的是关于 ThreadRoot()的地址问题。

一般情况下，汇编语言里的函数大部分情况下都符合以下的函数结构：

```
.globl fun_name
.type fun_name, @function
fun_name:
    pushl %ebp
     movl %esp, %ebp
=>  函数主体代码>
     leave
     ret
```

其中，寄存器%ebp 的内容是函数 fun_name 的栈的基地址（栈底），%esp 的内容是函数 fun_name 的栈的栈顶指针，开始的两条指令 pushl %ebp 与 movl %esp, %ebp 的作用是保存栈基地址，并将栈顶赋予%ebp；目的是函数返回之前将释放为该函数所分配的栈空间。

如果函数有初始化的局部变量，函数对应的汇编代码结构如下：

```
.globl fun_name
.type fun_name, @function
fun_name:
    pushl %ebp
    movl %esp, %ebp
=> subl $Stack_Size, %esp      #为初始化的局部变量，或为其它管理
                               #预留栈空间 Stack_Size 个字节
    movl $localVarValue, offset(%esp)      #将已初始化局部变量入栈
    依次利用 movl 指令将初始化的局部变量入栈
    <函数主体代码>
    leave
    ret
```

对于未初始化的局部变量，不在栈中为其预留空间（不入栈），只有将已经初始化的局部变量入栈；

在函数返回之前执行 leave 指令，其的作用是将栈中为函数预留的空间释放，并恢复栈基地址寄存器%ebp，该函数的栈就不存在了；因此，leave 相当于下述两条指令：

```
movl %ebp,%esp
pop $ebp
```

这就是开始的两条指令 pushl %ebp 与 movl %esp, %ebp 的作用。

由于指令 pushl %ebp 与 movl %esp, %ebp 是系统用于栈的管理，不是函数的主体，因此在 gdb 中为函数设置断点时给出的函数入口地址是从标有符号“=>”的位置。

注：假定内存地址由高到低，则 i386 的栈是向下生长的（即栈的生长方向是由高地址向低地址生长的）

如：SimpleThread()的源代码如下：

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

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

对应的汇编代码如图 2-2 所示。

```
Breakpoint 3, SimpleThread (which=0) at threadtest.cc:29
29          for (num = 0; num < 5; num++) {
(gdb) disass SimpleThread
Dump of assembler code for function SimpleThread(int):
   0x0804a4ff <+0>:     push   %ebp
   0x0804a500 <+1>:     mov    %esp,%ebp
   0x0804a502 <+3>:     sub    $0x18,%esp
=> 0x0804a505 <+6>:     movl   $0x0,-0xc(%ebp)
   0x0804a50c <+13>:    cmpl   $0x4,-0xc(%ebp)
   0x0804a510 <+17>:    jg     0x804a53f <SimpleThread(int)+64>
   0x0804a512 <+19>:    sub    $0x4,%esp
   0x0804a515 <+22>:    pushl  -0xc(%ebp)
   0x0804a518 <+25>:    pushl  0x8(%ebp)
   0x0804a51b <+28>:    push   $0x804bee0
   0x0804a520 <+33>:    call   0x8048a60 <printf@plt>
   0x0804a525 <+38>:    add    $0x10,%esp
   0x0804a528 <+41>:    mov    0x804f0e4,%eax
   0x0804a52d <+46>:    sub    $0xc,%esp
   0x0804a530 <+49>:    push   %eax
   0x0804a531 <+50>:    call   0x804a154 <Thread::Yield()>
   0x0804a536 <+55>:    add    $0x10,%esp
   0x0804a539 <+58>:    addl   $0x1,-0xc(%ebp)
   0x0804a53d <+62>:    jmp    0x804a50c <SimpleThread(int)+13>
   0x0804a53f <+64>:    nop
   0x0804a540 <+65>:    leave
   0x0804a541 <+66>:    ret
End of assembler dump.
(gdb)
```

图 2-2 gdb 中 SimpleThread()对应的汇编代码

由于变量 num 是一个未初始化的局部变量，不需要为其预留空间并入栈，即使 num 已经初始化了一个值，如 nun=9,，地址 0x0804a55 对应的指令也会被编译器优化成 movl $0x9, -0xc(%ebp)，也不需要专门的入栈操作。（也不需要为其它管理预留栈空间）

这就是在 gdb 中，命令 b SimpleThread 给出的 SimpleThread()函数的地址为 0x804a505 的原因。

InterruptEnable()函数对应的汇编代码如图 2-3 所示，从中可以看出为预留栈空间及释放栈空间等操作。

```
Dump of assembler code for function InterruptEnable():
   0x0804a346 <+0>:     push   %ebp
   0x0804a347 <+1>:     mov    %esp,%ebp
   0x0804a349 <+3>:     sub    $0x8,%esp
   0x0804a34c <+6>:     mov    0x804f0f0,%eax
   0x0804a351 <+11>:    sub    $0xc,%esp
   0x0804a354 <+14>:    push   %eax
   0x0804a355 <+15>:    call   0x804ab94 <Interrupt::Enable()>
   0x0804a35a <+20>:    add    $0x10,%esp
   0x0804a35d <+23>:    nop
   0x0804a35e <+24>:    leave
   0x0804a35f <+25>:    ret
End of assembler dump.
```

图 2-3 gdb 中 InterruptEnable ()对应的汇编代码

对于函数 ThreadRoot()，其汇编代码图 2-4 所示。

前两条指令是保存栈底与设置栈顶的操作，第三条指令 push %edx 是用于传递创建线程是所携带的参数。

如果该函数使用 C/C++实现，开始的三条指令应该属于该函数体；但由于该函数是用汇编代码实现，因此前 3 条指令是否属于 ThreadRoot()函数主体，有待于商榷。

(gdb) b ThreadRoot 给出的函数入口是 0x0804bb5a；

```
Breakpoint 1, 0x0804bb5a in ThreadRoot ()
(gdb) disass ThreadRoot
Dump of assembler code for function ThreadRoot:
   0x0804bb56 <+0>:     push   %ebp
   0x0804bb57 <+1>:     mov    %esp,%ebp
   0x0804bb59 <+3>:     push   %edx
=> 0x0804bb5a <+4>:     call   *%ecx
   0x0804bb5c <+6>:     call   *%esi
   0x0804bb5e <+8>:     call   *%edi
   0x0804bb60 <+10>:    mov    %ebp,%esp
   0x0804bb62 <+12>:    pop    %ebp
   0x0804bb63 <+13>:    ret
End of assembler dump.
(gdb)
```

图 2-4 gdb 中 ThreadRoot ()对应的汇编代码

2.8.2 关于对象的地址

../threads/system.cc 的 Initialize()函数中，语句 currentThread = new Thread("main")创建了 Nachos 的主线程”main”，并通过 currentThread->setStatus(RUNNING)将其状态设为就绪；

因此要查看主线程对象的地址，可以在 currentThread->setStatus(RUNNING)上设置断点，程序运行到该断点暂停后，利用(gdb) p currentThread 给出主线程对象的地

址。

过程：

(gdb) d 去掉以前设置的所有断点，在函数 Initialize()设为入口处设置断点（b Initialize）,然后(gdb) run 运行程序，程序会在函数 Initialize()的入口暂停，(gdb)list 列出 Initialize()的源代码，找到 currentThread->setStatus(RUNNING)语句所在的行号 num，(gdb) b num 在语句 currentThread->setStatus(RUNNING)上设置断点，，(gdb) c 继续运行，在该断点暂停后，利用(gdb) p currentThread 给出主线程对象的地址。

考察../threads/threadtest.cc，函数 ThreadTest()通过 t->Fork(SimpleThread, 1)创建了 Nachos 的第一个线程，因此可以在 t->Fork(SimpleThread, 1); 之后的语句 SimpleThread(0);上设置断点，程序运行到该断点暂停后，利用(gdb) p t 给出主线程对象的地址。(gdb) p *t 可显示该线程的一些详细信息。

过程：

(gdb) d 去掉以前设置的所有断点，在函数 ThreadTest ()设为入口处设置断点（b ThreadTest）,然后(gdb) run 运行程序，程序会在函数 ThreadTest ()的入口暂停，(gdb)list 列出 ThreadTest ()的源代码，找到 t->Fork(SimpleThread, 1)紧随的语句 SimpleThread(0) 所在的行号 num，(gdb) b num 在语句 SimpleThread(0)上设置断点，，(gdb) c 继续运行，在该断点暂停后，利用(gdb) p t 给出主线程对象的地址。

查看(gdb) p *t 会输出哪些信息。

2.8.3 关于 SWITCH()的返回值

../threads/scheduler.cc 中的 Scheduler::Run(..)调用了 SWITCH()函数，实现线程的上下文切换。

../threads/ switch-linux.s 中的 SWITCH()在 gdb 中汇编代码如图 2-5 所示。

```
(gdb) disass SWITCH
Dump of assembler code for function SWITCH:
   0x0804bb64 <+0>:     mov    %eax,0x804f104
   0x0804bb69 <+5>:     mov    0x4(%esp),%eax
   0x0804bb6d <+9>:     mov    %ebx,0x8(%eax)
   0x0804bb70 <+12>:    mov    %ecx,0xc(%eax)
   0x0804bb73 <+15>:    mov    %edx,0x10(%eax)
   0x0804bb76 <+18>:    mov    %esi,0x18(%eax)
   0x0804bb79 <+21>:    mov    %edi,0x1c(%eax)
   0x0804bb7c <+24>:    mov    %ebp,0x14(%eax)
   0x0804bb7f <+27>:    mov    %esp,(%eax)
   0x0804bb81 <+29>:    mov    0x804f104,%ebx
   0x0804bb87 <+35>:    mov    %ebx,0x4(%eax)
   0x0804bb8a <+38>:    mov    (%esp),%ebx
   0x0804bb8d <+41>:    mov    %ebx,0x20(%eax)
   0x0804bb90 <+44>:    mov    0x8(%esp),%eax
   0x0804bb94 <+48>:    mov    0x4(%eax),%ebx
   0x0804bb97 <+51>:    mov    %ebx,0x804f104
   0x0804bb9d <+57>:    mov    0x8(%eax),%ebx
   0x0804bba0 <+60>:    mov    0xc(%eax),%ecx
   0x0804bba3 <+63>:    mov    0x10(%eax),%edx
   0x0804bba6 <+66>:    mov    0x18(%eax),%esi
   0x0804bba9 <+69>:    mov    0x1c(%eax),%edi
   0x0804bbac <+72>:    mov    0x14(%eax),%ebp
   0x0804bbaf <+75>:    mov    (%eax),%esp
   0x0804bbb1 <+77>:    mov    0x20(%eax),%eax
   0x0804bbb4 <+80>:    mov    %eax,(%esp)
   0x0804bbb7 <+83>:    mov    0x804f104,%eax
   0x0804bbbc <+88>:    ret
   0x0804bbbd <+89>:    xchg   %ax,%ax
   0x0804bbbf <+91>:    nop
End of assembler dump.
```

图 2-5 gdb 中 SWITCH ()对应的汇编代码

查看 SWITCH()的返回地址有两种方法：

（1）方法 1：分析../threads/ switch-linux.s 中的 SWITCH()及上图中的代码可知，

执行完指令 0x0804bb69 <+5>　movl 4(%esp),%eax 后，寄存器 eax 中的内容是原线程（t1，oldThread）的地址；

可用(gdb) info r 查看寄存器的状态，看看原线程的地址是多少；

执行完指令 0x0804bb90 <+44>　movl 8(%esp),%eax 后，寄存器 eax 中的内容是新线程（t2，newThread）的地址；

可用(gdb) info r 查看寄存器的状态，看看新线程的地址是多少；

SIWTCH()执行完后不是直接返回到新线程的地址处开始执行新线程，系统还要做一些前期准备工作。

执行完指令 0x0804bbb1 <+77>　movl _PC(%eax),%eax 后，寄存器 eax 中的内容就是 SWITCH()的返回地址；

可用(gdb) info r 查看寄存器的状态，看看 SWITCH()的返回地址是多少；

因此，利用(gdb) d 删除所有断点，(gdb) b SWITCH 在函数 SWITCH()设置断点，(gdb) r 执行程序到该处暂停，利用(gdb) si 或(gdb) ni 单步执行，执行完地址 0x0804bbb1 处的指令后，(gdb) info r 查看寄存器的状态，eax 中的值即为 SWITCH()的返回值；

在我的环境下，本次运行 exa 的值即 SWITCH()的返回地址是 0x0804bb56，从 ThreadRoot()的汇编代码可知，该地址是 ThreadRoot()中的第一条指令。如图 2-6 所示。

```
0x0804bbaf in SWITCH ()
(gdb)
0x0804bbb1 in SWITCH ()
(gdb)
0x0804bbb4 in SWITCH ()
(gdb) info r
eax            0x804bb56        134527830
ecx            0x804a346        134521670
edx            0x1      1
ebx            0x0      0
esp            0x8059bb0        0x8059bb0
ebp            0x0      0x0
esi            0x804a4ff        134522111
edi            0x804a32c        134521644
eip            0x804bbb4        0x804bbb4 <SWITCH+80>
eflags         0x292    [ AF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

图 2-6　寄存器 exa 的值为 SWITCH()的返回值

（2）方法 2

单条指令跟踪 SWITCH()的执行，当执行完最后一条指令 0x0804bbbc <+88> ret 后，看看下条指令在何处执行，该指令地址就是 SWITCH()的返回地址。

利用(gdb) d 删除所有断点，(gdb) b SWITCH 在函数 SWITCH()设置断点，(gdb) r 执行程序到该处暂停，利用(gdb) si 或(gdb) ni 单步执行，执行完 0x0804bbbc <+88> ret 指令后，下条执行的指令地址是 0x0804bb56，是 ThreadRoot()的第一条指令。如图 2-7 所示。（思考：该地址所在程序中位置具体的含义是什么？）

```
0x0804bbb4 in SWITCH ()
(gdb)
0x0804bbb7 in SWITCH ()
(gdb)
0x0804bbbc in SWITCH ()
(gdb)
0x0804bb56 in ThreadRoot ()
(gdb)
```

图 2-7 第一次调用 SWITCH()返回地址

(gdb) c 继续执行，第二次在 SWITCH()暂停执行，利用(gdb) si 或(gdb) ni 单步执行，可以看出 SWITCH()返回到地址 0x080491d9 处，在 Scheduler::Run()中，如图 2-8

所示。

```
(gdb)
0x0804bbca in SWITCH ()
(gdb)
0x080491d9 in Scheduler::Run (this=0x8054ad0, nextThread=0x8054b60)
    at scheduler.cc:118
118         SWITCH(oldThread, nextThread);
(gdb) l
113         // This is a machine-dependent assembly language routine defined
114         // in switch.s.  You may have to think
115         // a bit to figure out what happens after this, both from the point
116         // of view of the thread and from the perspective of the "outside wo
rld".
117
118         SWITCH(oldThread, nextThread);
119
120         DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
121
```

图 2-8 第二次调用 SWITCH()返回地址

从图 2-9 中 Scheduler::Run()的部分汇编代码中可以看出，该地址对应 Scheduler::Run()中 SWITCH(oldThread, nextThread)紧随的一条指令（见图中符号=>所示的位置）。

```
   0x080491aa <+70>:    mov    %eax,%ebx
   0x080491ac <+72>:    sub    $0xc,%esp
   0x080491af <+75>:    pushl  -0xc(%ebp)
   0x080491b2 <+78>:    call   0x804927c <Thread::getName()>
   0x080491b7 <+83>:    add    $0x10,%esp
   0x080491ba <+86>:    push   %ebx
   0x080491bb <+87>:    push   %eax
   0x080491bc <+88>:    push   $0x804bd04
   0x080491c1 <+93>:    push   $0x74
   0x080491c3 <+95>:    call   0x804a48b <DEBUG(char, char*, ...)>
   0x080491c8 <+100>:   add    $0x10,%esp
---Type <return> to continue, or q <return> to quit---
   0x080491cb <+103>:   sub    $0x8,%esp
   0x080491ce <+106>:   pushl  0xc(%ebp)
   0x080491d1 <+109>:   pushl  -0xc(%ebp)
   0x080491d4 <+112>:   call   0x804bb72 <SWITCH>
=> 0x080491d9 <+117>:   add    $0x10,%esp
   0x080491dc <+120>:   mov    0x804f0e4,%eax
   0x080491e1 <+125>:   sub    $0xc,%esp
   0x080491e4 <+128>:   push   %eax
   0x080491e5 <+129>:   call   0x804927c <Thread::getName()>
   0x080491ea <+134>:   add    $0x10,%esp
   0x080491ed <+137>:   sub    $0x4,%esp
   0x080491f0 <+140>:   push   %eax
   0x080491f1 <+141>:   push   $0x804bd2f
   0x080491f6 <+146>:   push   $0x74
   0x080491f8 <+148>:   call   0x804a48b <DEBUG(char, char*, ...)>
   0x080491fd <+153>:   add    $0x10,%esp
   0x08049200 <+156>:   mov    0x804f0e8,%eax
   0x08049205 <+161>:   test   %eax,%eax
   0x08049207 <+163>:   je     0x8049235 <Scheduler::Run(Thread*)+209>
   0x08049209 <+165>:   mov    0x804f0e8,%ebx
   0x0804920f <+171>:   test   %ebx,%ebx
   0x08049211 <+173>:   je     0x804922b <Scheduler::Run(Thread*)+199>
   0x08049213 <+175>:   sub    $0xc,%esp
```

图 2-9 Scheduler::Run()的部分汇编代码

重复上述过程，你会发现 SWITCH()函数后续的返回位置与第二次返回的位置相同（参见图 2-8 及 2-9）；

为什么 SWITCH()函数的第一次返回位置与后续的返回位置不同？

分析../threads/ThreadTest()与 SimpleThread(_int which) 的代码：

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

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

    t->Fork(SimpleThread, 1);
    SimpleThread(0);
}
SimpleThread(_int which)
{
    int num;

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

从中可以看出，Nachos 的主线程”main”中的语句 t->Fork(SimpleThread, 1)调用 Thread::Fork()创建了一个子线程（命名为"forked thread"），将子线程设为就绪状态并进入就绪队列的尾部，子线程被调度时所执行的代码是 SimpleThread(1)（参见 Thread::Fork()实现代码）；主线程”main”被调度时所执行的执行的代码为 SimpleThread(0)；

当使用-rs 参数运行 nachos 时（nachos –rs random-seed，如 nachos –rs 10）时，nachos 会创建一个定时器（Timer）设备，每隔一段时间（随机数）触发一次定时器中断（目前在 Timer 类中设置的中断处理程序完成是执行 currentThread->Yield()，实现时间片轮转调度（RR）算法）；

如果运行 Nachos 时不使用参数-rs，则不创建定时器 Timer，线程调度采用的是 FCFS 线程调度算法；

不管采用 RR 还是 FCFS 线程调度算法，首先调度执行主线程”mian”（在 Initialize()函数中创建并进入就绪队列），输出”*** thread 0 looped 0 times”后调用了 Thread::Yield()；

```
Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ASSERT(this == currentThread);

    DEBUG('t', "Yielding thread \"%s\"\n", getName());

    nextThread = scheduler->FindNextToRun();
    if (nextThread != NULL) {
  scheduler->ReadyToRun(this);
  scheduler->Run(nextThread);
    }
    (void) interrupt->SetLevel(oldLevel);
}
```

Thread::Yield ()中将子线程从就绪队里中取出（nextThread = scheduler->FindNextToRun()），将主线程的状态从执行转到就绪并放入就绪队列尾（scheduler->ReadyToRun(this)），将子线程设为执行状态（currentThread = nextThread，currentThread->setStatus(RUNNING)），然后第调用 SWITCH()将主线的上下文切换到子线程的上下文，子线程开始执行（scheduler->Run(nextThread)）。

注意这里的 SWITCH()是第一次被调用。

这次 SWITCH()的返回到 ThreadRoot()的第一条指令处开始执行，由于子线程是从头开始执行，因此 ThreadRoot()是所有利用 Thread::Fork()创建的线程的入口。

子线程开始执行后，后续与主函数发生的上下文切换都是从上次被中断的地方开始执行，即 Scheduler::Run()中语句 SWITCH(oldThread, nextThread)之后。

第 3 章 Nachos 的 Makefiles（实验 2）

3.1 目的与任务

该实验在目录 lab2 中完成。

（1）熟悉 Nachos 的 makefiles 的结构；

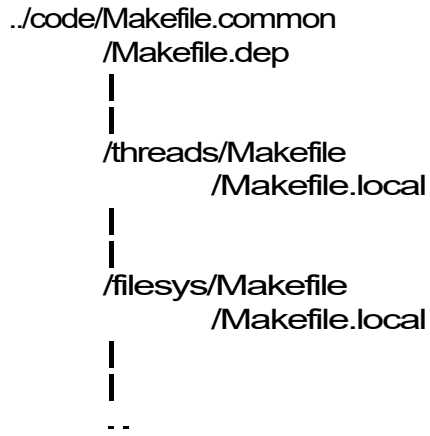
（2）熟悉如何在几个 lab 文件目录中构造相应的 Nachos 系统；

3.2 Nachos 的 Makefile 文件

code/的子目录下一般都有 Makefile 与 Makefile.local 两个工程文件，用于对包含该功能的 Nachos 系统进行编译与链接。例如基于 code/thread 目录下的这两个 makefile 文件，可以生成一个 Nachos 的最小内核，包含 Nachos 线程的创建、调度、撤销等功能；基于 code/filesys 目录下的两个 makefile 文件，可以生成包含文件系统的 Nachos 内核。

code/目录下还有两个 makefile 文件：Makefile.common 与 Makefile.dep，包含编译、链接 Nachos 系统所需的 makefile 公共语句，被 code 下子目录中的 Makefile 与 Makefile.local 所共享（Makefile 中利用语句 include ../Makefile.common 包含 Makefile.common，Makefile.common 中又包含 Makefile.dep）

Nachos 的 makefile 文件结构大致如下：



3.2.1 code/下子目录中的 Makefile 文件

在终端下进入相应目录，利用 make 或 make all 命令，可依据该目录下的 Makefile 文件生成包含相应功能的 Nachos 可执行程序。

Makefield 文件的内容主要包括下述两条语句：

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

3.2.2 code/下子目录中的 Makefile.local 文件

该文件的作用主要是对一些编译、链接及运行时所使用的宏进行定义。

• CCFILES：指定在该目录下生成 Nachos 时所涉及到的 C++源文件；

• INCPATH：指明所涉及的 C++源程序中的头文件(.h 文件)所在的路径，以便利用 g++进行编译链接时通过这路径查找这些头文件。

• DEFINES：传递个 g++的一些标号或者宏。例如 code/threads 下的 Makefile.local 中，DEFINES += -DTHREADS，这里 DTHREADS 的含义相当于在该目录的 C++程序（如 main.cc）中定义了一个宏 THREADS，即：#deffine THREADS

注：INCPATH 与 DEFINES 赋值时使用"+="而不是"="，如 DEFINES += -DTHREADS，表示将参数-DTHREADS 附加到字符串 DEFINES 的尾部。

3.2.3 code/目录下的 Makefile.dep 文件

目前 Nachos 可以在四种操作系统平台上进行编译，code/threads/arch 目录下的四个子目录分别对应着四个平台（也可参加其它目录，如 code/filesys/arch 目录）。其中，Ubuntu 使用的目录是 code/threads/arch/unknown-i386-linux.

Code/目录下的文件 Makefile.common 包含 Makefile.dep。Makefile.dep 文件根据你安装 Nachos 时所使用的操作系统环境，定义一些相应的宏，供 g++使用。

Makefile.dep 中，首先利用语句 uname = $(shell uname)获取安装 Nachos 所使用的操作系统平台（如 Ubuntu 系统返回"Linux"，可在 uname = $(shell uname)后添加语句$(warning $(uname))，或语句 echo $(uname)，或@echo $(uname) 查看变量 uname 的内容），然后利用语句 ifeq ($(uname),xxxx)根据所使用的平台定义相应的宏，为 g++所使用（xxxx 是相应的平台名，如 Linux）。

如 Ubuntu 平台：

```
uname = $(shell uname)
$(warning $(uname))     #输出变量 uname 的内容
                        #Ubuntu 平台输出 uname 的内容为"Linux"
```

#或利用下述两条 echo 语句输出变量 uname 的内容

```
#echo $(uname)
#@echo $(uname)
```

Makefile.dep 中关于 Ubuntu 等类 Linux 平台的宏定义如下：

```
# 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
```

从中可以看出，这些宏主要包括：HOST, arch, CPP, CPPFLAGS, GCCDIR, LDFLAGS 以及 ASFLAGS。

其中，GCCDIR 给出 gcc mips 交叉编译器所在的路径及其前缀，从中可以看出我们需要将 mips 交叉编译器编译器需要安装到/usr/local/目录的原因。

根据上述操作系统平台所依赖的宏（变量），在 Makefile.dep 文件的最后几行，给出了 makefile.common 所使用的几个宏（g++使用）：

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

code/目录下的几个子目录下有个 arch 目录。如 code\threads\arch，该目录下又有四个子目录：dec-alpha-osf/、dec-mips-ultrix/、sun-sparc-sunos/及 unknown-i386-linux/，分别对应所使用的操作系统平台。每个目录下又有 3 个子目录：bin/、depends/及 objects/，分别存放编译链接 Nachos 时所产生的 Nachos 可执行文件（nachos）、依赖文件（dependence files，形如 xxxx.d）以及目标文件（object files, 形如 xxx.o）。

如果我们使用的操作系统是 Ubuntu（Linux 系统），使用的是 arch/unknown-i386-linux/目录。也就是说，利用 make 命令生成的 Nachos 可执行程序会放在 arch/unknown-i386-linux/bin 中，即 arch/unknown-i386-linux/bin/nachos，然后 Makefile.common 中利用 ln 命令在 code/子目录下建立一个可执行文件 arch/unknown-i386-linux/bin/nachos 的链接文件 nachos，如在 code/threads/目录下会有一个符号链接文件 nachos，指向文件 arch/unknown-i386-linux/bin/nachos。

3.2.4 code/目录下的 Makefile.commom 文件

在这些 makefile 文件中，最复杂的是 Makefile.common 文件，它定义了编译链接生成一个完整的 Nachos 可执行文件所需要的所有规则。

文件 Makefile.common 首先利用 inlcude 语句把文件 Makefile.dep 包含进去（include ../Makefile.dep），然后利用 vpath 定义了一些编译时查找相关文件的路径，如：

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

当利用 make 命令编译生成 Nachos 系统时，如果在当前目录下找不到相应的文件，会在 vpath 给出的路径中查找这些文件。

实验中，若在一个目录中（如 code/lab3/）生成 Nachos 系统时，如果只是使用涉及到的文件而不需要修改它们（如.cc、.c 或.h 文件），则不需要将这些文件复制到 code/lab3/中，可以用 vpath 设置指向这些文件的路径，make 命令会依据 vpath 的设置自动查找这些文件。

语句 CFLAGS = -g -Wall -Wshadow $(INCPATH) $(DEFINES) $(HOST) –DCHANGED 定义了 g++使用的参数。

下述语句给出了生成 Nachos 系统所需要产生的的目标文件名（object files）、可执行文件名以及它们的存放路径。

```
s_ofiles = $(SFILES:%.s=$(obj_dir)/%.o)
c_ofiles = $(CFILES:%.c=$(obj_dir)/%.o)
cc_ofiles = $(CCFILES:%.cc=$(obj_dir)/%.o)

ofiles = $(cc_ofiles) $(c_ofiles) $(s_ofiles)
program = $(bin_dir)/nachos
```

利用语句$(warning $(bin_dir))可以查看（宏）变量 bin_dir 的值，Ubuntu 环境下，该值应该是 arch/unknown-i386-linux/bin/。

下述语句可链接生成 Nachos 的可执行文件 nachos（在 arch/unknown-i386-linux/bin/目录下，即 arch/unknown-i386-linux/bin/nachos），并在当前目录下建立一个指向该文件的符号链接文件 nachos。其中，%是一个通配符，可以匹配任意的非空字符串，自动化变量$@指的是目标文件名（最终要生成的可执行程序，如 nachos），$^指的是所有依赖文件，$<指的是第一个搜索到的依赖文件。（如）

```
$(program): $(ofiles)
$(bin_dir)/% :
   @echo ">>> Linking" $@ "<<<"
   $(LD) $^ $(LDFLAGS) -o $@
   ln -sf $@ $(notdir $@)       #在当前目录下建立文件
                                # arch/unknown-i386-linux/bin/nacho 的符号链接文件。
```

在当前目录下打开一个终端，输入./nachos，可执行生成的 nachos 系统，而不必到目录 arch/unknown-i386-linux/bin/下去执行它。

下述语句依据 C++源程序生成相应的 object 文件（.o 文件），存放在 arch/unknown-i386-linux/objects 目录中。

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

下述语句依据 C 源程序生成相应的 object 文件（.o 文件），存放在 arch/unknown-i386-linux/objects 目录中。

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

下述语句依据汇编程序生成相应的 object 文件（.o 文件），存放在 arch/unknown-i386-linux/objects 目录中。

```
$(obj_dir)/%.o: %.s
	@echo ">>> Assembling" $< "<<<"
	$(CPP) $(CPPFLAGS) $< > $(obj_dir)/tmp.s
	$(AS) -o $@ $(obj_dir)/tmp.s
	rm $(obj_dir)/tmp.s
```

下述语句依据.cc 文件本身以及相关的头文件（.h）文件生成相应的 dependence 文件（.d 文件），存放在 arch/unknown-i386-linux/ depends 目录中。其中，g++参数-MM 自动寻找源文件（.cc 文件）直接或间接关联的头文件（包括系统头文件，如果希望不包括头文件，可用参数-M），并生成相应的依赖关系文件（.d），即 g++自动维护源文件与头文件之间的关联关系，不需要用户去指定，以防止用户的疏忽而漏掉关联的头文件。

```
s_dfiles = $(SFILES:%.s=$(depends_dir)/%.d)
c_dfiles = $(CFILES:%.c=$(depends_dir)/%.d)
cc_dfiles = $(CCFILES:%.cc=$(depends_dir)/%.d)

dfiles = $(cc_dfiles) $(c_dfiles) $(s_dfiles)

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

下述语句依据.c 文件本身以及相关的.h 文件生成相应的 dependence 文件（.d 文件），存放在 arch/unknown-i386-linux/ depends 目录中。

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

下述语句依据.s 文件本身以及相关的.h 文件生成相应的 dependence 文件（.d

文件），存放在 arch/unknown-i386-linux/ depends 目录中。

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

语句 include $(dfiles)将所有创建的.d 文件包括进 makefile.common 中，其目的是通知 g++，当$(dfiles)中的任何一个.d 文件所依赖的源文件（.cc、.c、.s 文件）以及与其相关的头文件（.h 文件）被修改后，.d 文件需要重新编译，相应的.o 文件以及最终的可执行文件也需要重新编译生成。

这样做的目的是无论是源文件还是关联的 .h 文件有更新，都会对源文件进行重新编生成新的目标文件。

关于 makefile 更多的内容请参考 Nachos_Introduction.pdf.中 Lab2 的相关内容。

3.2.5 在其它目录中修改 Nachos 代码并生成修改后的 Nachos 系统

在目录 code/threads、code/filesys、code/userprog 及目录 code/monitor 所对应的终端运行 make 命令所编译生成功能的 Nachos 系统功能也有所不同。

理论上讲，对于 lab1 与 lab2，可在 code/threads 下完成；对于 lab3，可在 code/monitor 下完成；对于 lab4 与 lab5，可在 code/ filesys 下完成；对于 lab6、lab7 与 lab8，可在 code/ userprog 下完成。

我们课程设计的任务是要求你在阅读、理解 Nachos 代码的基础上，扩展修改 Nachos 的功能。因此，需要对 Nachos 现有的源代码做一定的修改，但最好不要在相应目录中直接修改其中的源文件，以免导致 Nachos 工作不正常。

为便于你的设计修改工作，在 code 目录下设计了几个工作目录 lab2、lab3、lab5 及 lab7-8，在做相应的 lab 所规定的任务时，你可以将要修改的文件复制到这些目录下，然后在这些目录下利用 make 命令生成相应的 Nachos 系统，来测试你的工作。

这就需要在这些工作目录下不仅包括要修改的源文件，还要建立正确的 Makefile 文件及 Makefile.local 文件。

例如，如果在 lab2 中需要修改 code/threads 目录下的类 scheduler.cc，然后根据你对 scheduler.cc 的修改生成一个新的 Nachos 系统以测试你的修改是否符合要求，可以将文件 scheduler.h 与 scheduler.cc 从 code/threads 目录复制到目录 lab2 中进行修改，而不要直接修改 code/threads 中的两个文件。

首先，将目录 code/threads 中的 code/threads/arch 及其子目录全部复制到目录 lab2 中，并将三个目录 arch/unknown-i386-linux/bin/ 、 arch\unknown-i386-

linux\depends\及 arch\unknown-i386-linux\objects\清空（我们一般使用 intel 公司的芯片及相应的 Linux 系统，因此使用这三个目录）。

然后将目录 code/threads 中 Makefile 和 Makefile.local 复制到目录 lab2 中。

最后，需要修改 lab2 中的文件 Makefile.local，以正确编译生成你修改后的 Nachos 系统，lab2 中的文件 Makefile 不需要修改。

文件 Makefile.local 定义了宏变量 CCFILES（说明需要使用哪些 C++源程序）及 INCPATH（说明需要从哪里查找.h 头文件）。在编译 Nachos 时，由于需要修改的 C++源程序（如 scheduler.cc）已经在当前目录 lab2 中，对于其它不需要修改的 C++源程序，make 会根据 vpaths 去查找它们，因此 CCFILES 不需要修改，但 INCPATH 需要修改，因为如果我们对目录 lab2 中的 scheduler.h 也做了修改，当在 lab2 中运行 make 命令时，编译目录 lab2 中的 scheduler.cc 时会关联修改后的 lab2 目录中的 scheduler.h，而其它目录中涉及 scheduler.h 的.cc 文件或.h 文件会仍然使用 code/threads 目录中的 scheduler.h。

修改 INCPATH 的方法：

在 INCPATH 中将目录../lab2 添加到../threads 之前，如下：

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

由于目录../lab2 在目录../threads 之前，g++的 C 预处理程序（cpp）在处理目录../lab2 中.cc 源文件的#include 语句时会首先从目录../lab2 中搜索相应的头文件，如果目录../lab2 中不存在，会依次从../threads 及../machine 中依次查找。

该方法比较简单，但只有目录 lab2 下的 scheduler.cc 使用目录 lab2 下的 scheduler.h，其他目录中的.cc 源文件仍然使用目录../threads 下的 scheduler.h。

如下示例可说明这一问题：

命令终端进入目录 lab2，将 lab2 作为当前目录。

1、利用 ls 显示目录 lab2 下的文件：

命令控制端：../lab2$ ls

```
Makefile  Makefile.local  arch/  scheduler.cc  scheduler.h
```

2、利用 make 编译链接生成 Nachos

命令控制端：../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
 ../lab2$ ls
arch/  Makefile  Makefile.local  nachos  scheduler.cc  scheduler.h
```

3、利用 touch 命令修改 lab2 目录中的 scheduler.h 的时间为当前时间，重新生成 Nachos，查看与 scheduler.h 相关联的源程序是否被重新编译。

命令控制端：.. /lab2$ touch scheduler.h

命令控制端：../lab2$ make

```
>>> Building dependency file for      scheduler.cc <<<
.....................
>>> Compiling scheduler.cc <<<
.....................
g++ -g -Wall -Wshadow -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
```

从上面的信息可以看出，文件 scheduler.h 被修改后（利用 touch 修改了文件的时间相当于文件内容被修改且保存），make 程序检测到该文件的修改重新对相关的源文件进行编译，只有目录 lab2 中的 scheduler.cc 被重新编译，其它与 scheduler.h 有关联的文件（如 threads 目录中的 main.cc，sysch.cc，syschtest.cc，system.cc，thread.cc 及 threadtest.cc 等）没有被重新处理。

4、如果利用 touch 命令修改 threads 目录中的 scheduler.h 的时间为当前时间，重新生成 Nachos，查看哪些文件被重新处理。

命令控制端：../lab2$　touch　../threads/scheduler.h

命令控制端：../lab2$　make

```
>>> Building dependency file for    ../machine/timer.cc <<<
>>> Building dependency file for    ../machine/sysdep.cc <<<
...
>>> Building dependency file for    ../threads/system.cc <<<
>>> Building dependency file for    ../threads/synch.cc <<<
>>> Building dependency file for    ../threads/main.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
```

从上面的信息可以看出，目录 threads 中的文件 scheduler.h 被修改后，目录 machine 与 threads 中涉及 scheduler.h 的源程序全部被重新编译。这是因为在 code 目录中的 Makefile.common 文件中，有如下命令：

```
# The following set of rules define how to build dependency files
# automatically from various source files.    These rules have been
# taken from the gmake documentation with minor modifications.

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

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

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

上述命令生成 Nachos 的依赖文件保存在相应目录的 arch\unknown-i386-linux\depends 中，如 code\lab2\arch\unknown-i386-linux\depends。

g++中参数–MM 的作用之一是搜索与.cc 文件相同目录下的.h 文件，生成依赖关系文件.d，保存在目录 arch\unknown-i386-linux\depends 中。

例如文件../threads/main.cc 通过 system.h 文件间接包含文件 scheduler.h（main.cc 包含 system.h 文件，system.h 包含 scheduler.h），因此 g++ -MM 首先搜索与../threads/main.cc 文件相同目录下的 scheduler.h 文件即../threads/scheduler.h（而不是../lab2/ scheduler.h）。具体信息可查看../lab2/arch/unknown-i386-linux/depends/中的文件 main.d。

我们希望当 lab2/scheduler.h 文件被修改后，不管是直接还是间接使用 lab2/scheduler.h 的.cc 源程序都应重新编译，以生成新的 Nachos 系统。因此需要将../threads 目录中直接或间接使用 scheduler.h 的文件都复制到 lab2 目录中。

Linux 提供的工具 grep 可以检查哪些文件中包含字符串 **scheduler.h**。

命令窗口进入 code/threads 目录，输入命令 **grep scheduler.h *，**屏幕输出与 scheduler.h 相关联的文件列表。

命令控制端：../threads$　grep scheduler.h *

```
grep: arch: Is a directory
scheduler.cc: #include "scheduler.h"
scheduler.h:// scheduler.h
system.h: #include "scheduler.h"
```

由于头文件 system.h 中也包含 scheduler.h，因此我们需要进一步查找包含 system.h 的文件。

**命令控制端：../threads$　grep system.h *
屏幕输出包含 system.h 的文件列表：**

```
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"
```

由此，我们只将下述几个文件从 code/threads 目录复制到 code/lab2 目录中即可（使用 schduler.h 的最小文件子集）：

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

现在 lab2 目录下包含下述文件：

命令控制端：../lab2$ ls

```
Makefile        arch/    scheduler.cc   synchtest.c   thread.cc
Makefile.local  main.cc  scheduler.h    system.cc     threadtest.c
Makefile.local~ nachos   synch.cc       system.h
```

现在，我们可以依据对 schduler.cc 及 schduler.h 的修改编译生成正确的 Nachos 系统。

（1）首先利用 touch 更新 shcduler.h 的时间

命令控制端：../lab2$ touch scheduler.h

命令控制端：../lab2$ make

```
>>> Building dependency file for    ../machine/timer.cc <<<
...
>>> Compiling main.cc <<<
g++ -g -Wall -Wshadow    -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
```

（2）利用 touch 更新 threads/scheduler.h 的时间，在 code/lab2 下重新运行 make 命令，测试对 threads/scheduler.h 的改变是否影响 Nachos 的生成。

命令控制端：../lab2$ touch   ../threads/scheduler.h

命令控制端：../lab2$ make

```
make:   'arch/unknown-i386-linux/bin/nachos' is up to date.
```

该信息说明在 lab2 目录下，修改 scheduler.h 后，重新编译生成的 Nachos 所涉及的模块都感知到了 scheduler.h 的改变，而不是使用 threads/scheduler.h 文件。

注 1：命令 touch fileName 的功能：若 fileName 已存在，则将文件 fileName 的时间标签更新为系统当前的时间（默认方式），文件的数据不做修改；若 fileName 不存在，则新建文件名为若 fileName 的空文件。

命令 touch 的有关参数及使用方法请参阅网上资料。

注 2：**grep**（global search regular expression(RE) and print out the line，全面搜索正则表达式并把行打印出来）是一个文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印（显示）出来。如命令 **grep scheduler.h *** 列出当前目录中包含字符串 **scheduler.h** 的所有文件。

命令 **grep** 的有关参数及使用方法请参阅网上资料。

第 4 章 利用信号量实现线程同步（实验 3）

4.1 目的

（1）理解 Nachos 中如何创建并发线程；
（2）理解 Nachos 中信号量与 P、V 操作是如何实现的
（3）如何创建与使用 Nachos 的信号量
（4）理解 Nachos 中是如何利用信号量实现 producer/consumer problem；
（5）理解 Nachos 中如何测试与调试程序；
（6）理解 Nachos 中轮转法（RR）线程调度的实现；

4.2 任务

阅读 code/lab3 目录下的 ring,h、ring.cc、main.cc 及 prodcons++.cc

在理解它们工作机理的基础上，补充目录 lab3 中提供的代码，利用 Nachos 实现的信号量写一个 producer/consumer problem 测试程序。主要是在 prodcons++.cc 中补充代码。

分析../threads/threadtest.cc，理解利用 Thread::Fork()创建线程的方法；

分析 Thread::Fork() ，理解内核创建线程的过程；

分析../threads/synch.cc，理解 Nachos 中信号量是如何实现的；

分析../monitor/prodcons++.cc，理解信号量的创建与使用方法；

分析 Thread::Fork() , Thread::Yiled(), Thread::Sleep(), Thread::Finish() ,, Scheduler:: Scheduler::ReadyToRun(), Scheduler::FindNextToRun(), Scheduler::Run()等相关函数，理解线程调度及上下文切换的工作过程；

分析 Nachos 对参数-rs 的处理过程，理解时钟中断的实现，以及 RR 调度算法的实现方法；

4.3 背景知识

4.3.1 信号量

并发进程或线程之间的同步机制有多种，信号量是最常用的一种解决方案。Nachos 在程序../threads/synch.cc 中利用 Semaphore 类实现了一种信号量机制及相应的 P、V 操作，这里的实现与教材中介绍的信号量有所不同，具体可阅读./threads/synch.cc。

4.3.2 生产者/消费者问题

生产者/消费者问题所描述的模型在操作系统中被广泛采用。生产者与消费者均

可访问共享内存中的一个环形缓冲区（ring buffer）。生产者生产出产品后将其放入环形缓冲区，而消费者从环形缓冲区中取出产品进行消费。当环形缓冲区满时生产者阻塞直到环形缓冲区有空时将其唤醒继续执行；同样，当环形缓冲区为空时消费者阻塞直到环形缓冲区非空时将其唤醒继续执行。因此，生产者与消费者需要一种同步机制对它们之间的操作加以控制。

4.3.3 Nachos 的 Main 程序

当运行 Nachos 时，首先运行的程序模块是 main 程序。每一个子目录中有可以有 mian.cc 程序（目前目录 code/filesys 及 code/userprog 中使用的是 code/threads 中的 main.cc）。

请阅读 code/threads 中的 main.cc，理解如下问题：

（1）Nachos 是如何解释处理命令行参数的；

（2）Nachos 的内核是如何初始化的；

（3）主程序（main program）所对应的线程是如何创建另一个线程并在该线程中执行函数 SimpleThread(int which)。

SimpleThread(int which).的源代码在../threads/threadtest.cc.中。

4.4 Things to Do

目录 code/lab3 中提供了几个文件：main.cc, prodcons++.cc, ring.cc 和 ring.h.

其中，文件 ring.cc 和 ring.h 定义并实现了一个环形缓冲区的类 Ring，供生产者与消费者访问。这两个文件不需要做任何修改。

main.cc 在../threads/main.cc 的基础上做了一定的修改，在此直接使用即可，不需要做任何修改。

../threads/main.cc 中在初始化 Nachos 内核后调用了 ThreadTest()函数以测试 Nachos 内核中线程的创建、上下文切换等功能；这里的 mian.cc（code/lab3 目录中的 mian.cc）没有调用 ThreadTest()函数，而是调用了文件 prodcons++.cc 中的 ProdCons()函数。prodcons++.cc 中给出了一个生产者/消费者进程的同步框架，需要你利用 Nachos 已实现的功能（如线程的创建、信号量及 P、V 操作等）补充完善代码，以完成生产者及消费者线程的创建与它们之间的同步。

文件 prodcons++.cc 包含涉及到的数据结构以及函数的接口。你可以根据文件中给出的详细的注释添加相应的代码，完成设计工作。

在 code/lab3 目录所对应的命令终端中运行 make，可编译生成 Nachos 系统，只是由于 prodcons++.cc 中缺少必要的代码，无法正常运行。

设计步骤：

在 code/lab3 目录中，

（a）详细阅读并深刻理解 ring.h 及 ring.cc 中的所有代码；

（b）阅读并理解 main.cc 的功能；

（c）详细阅读并深刻理解 prodcons++.cc 的程序结构，在 prodcons++.cc 中添加或修改相应的代码，满足设计要求。（可依据其中的注释添加相应的代码）

（d）利用 make 编译生成新的 Nachos，并测试其功能是否满足设计要求；

根据生产者/消费者问题的功能定义，你的实现应该满足如下条件:

（1）生产者线程所产生的所有的消息，都应该被消费者接收并保存到输出文件中（tem_0，temp_1，…）

（2）每个消息只能被接收一次且在文件保存一次

（3）来自于同一个生产者的消息，以及被同一个消费者接收到的消息，在文件保存的顺序应该按其序号升序排列；

例如，对于有两个生产者与两个消费者的情况下，如果每个生产者分别生产 4 个消息，其运行结果应该形如：

```
• 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;
```

如果其中一个文件为空，在运行./nachos 时，可以加上参数（随机数种子），如./nachos –rs *seed-number*。

nachos –rs 中的参数 rs 触发定时器中断，实按时间片轮转线程调度，实现线程的分时，参见../threads/system.cc 中函数 Initilize() 对参数 rs 的处理；

如果运行 nachos 不使用-rs 参数，系统不创建定时器设备，也不会实现定时器中断；

4.5 几点注记

该实验需要你：

理解生产者-消费者模型；

阅读../threads/sysch.cc 关于信号量及 P、V 操作的实现；

阅读 threadtest.cc 中线程的创建与使用；

阅读../threads/thread.cc 关于线程创建的过程；

阅读../threads/scheduler.cc 中就绪队列的管理及线程的调度；

阅读../monitor/ prodcons++.cc，理解线程的创建、信号量的创建与使用；

阅读../threads/ system.cc, Timer.cc，Interrupt.cc 的相关代码，理解 Nachos 对参数-

rs 的处理过程，及时间片轮转线程调度的实现；

4.5.1 线程的创建

可参考../threads/threadtest.cc 中的 ThreadTest()函数中提供的方法创建一个线程；

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

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

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

    Thread *t = new Thread("forked thread");
    t->Fork(SimpleThread, 1);   //线程体是 SimpleThread(1)，注意参数 1 的传递

    SimpleThread(0);
}
```

Thread 类的构造方法构造了一个 new 线程；

```
Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;       //用户线程内存标识
#endif
}
```

4.5.2 信号量

可参考../monitor/ prodcons++.cc 中函数 ProdCons()对信号量的创建与使用；

建立信号量：

```
Semaphore    *mutex ,*nempty, *nfull;

mutex = new Semaphore("mutex", 1);
nempty = new Semaphore("nempty", BUFF_SIZE);
nfull = new Semaphore("nfull", 0);
```

其后就可以使用信号量提供的 P()、V()操作实现线程的互斥与同步；如：

```
nempty->P();   //同步在前
mutex->P();    //互斥在后
mutex ->V();
nfull->V();
```

4.5.3 代码实现

可参考../monitor/ prodcons++.cc 中对生产者-消费者模型的部分实现；

在../lab3/prodcons++.cc 的函数 void ProdCons()中，

```
// Put the code to construct all the semaphores here.
//根据生产者-消费者模型的思想，创建几个信号量，并赋予初值
//参见../threads/ synch.cc 及 synch.h 对信号量及 P、V 的定义
//利用 Nachos 提供的信号量类 Semaphore 创建互斥信号量 mutex，初值为 1，
//实现生产者与消费者互斥访问缓冲池 ring，如
```

```
mutex = new Semaphore(“mutex”,1)
```

类似的可以创建其它两个同步信号量 nempty 与 nfull 可以利用类似的方法创建

```
nempty= new Semaphore("nempty",….);
nfull= new Semaphore("nfull", …);


// Put the code to construct a ring buffer object with size
 //BUFF_SIZE here.
 //利用 Nachos 提供的 Ring 类创建一个缓冲池对象 ring（参见../lab3/ring.cc）
 ring = new Ring(BUFF_SIZE);


//利用../threads/thread.cc 中 void Thread::Fork(VoidFunctionPtr func, _int arg)
//创建 N_PROD 个生产者线程，让其执行代码 Producer(_int which)，如：
// create and fork N_PROD of producer threads
      for (i=0; i < N_PROD; i++)
      {
        // this statemet is to form a string to be used as the name for
        // produder i.
        sprintf(prod_names[i], "producer_%d", i);

        // Put the code to create and fork a new producer thread using
        //         the name in prod_names[i] and
        //         integer i as the argument of function "Producer"
        //    ...

          //关于线程的创建，可参阅../threads/threadtest.cc 中线程创建与使用的示例

        //线程名为"producer_1"，"producer_2"等
        produce[i] = new Thread(prod_name[i]);
        //该线程执行 producer 函数，并将参数 i 的值传给 producer 函数，
        //即执行 producer(i)
        //调用 Fork()后，该线程处于就绪状态，并放到就绪队列尾部
        //参见 Thread::Fork(VoidFunctionPtr func, _int arg)及
        //Scheduler::ReadyToRun (Thread *thread)
        produce[i] -> Fork(producer,i);

          //参照该上述方法创建几个消费者线程
          ……
       };
```

对于生产者线程的执行的代码 void Producer(_int which)，主要工作是将产品放入缓冲池，但放入操作需要利用定义的信号量进行同步；

```
void Producer(_int which)
{
    int num;
    slot *message = new slot(0,0);    //参见 ring.cc 中的 slot 类

//  This loop is to generate N_MESSG messages to put into to ring buffer
//    by calling    ring->Put(message). Each message carries a message id
//    which is represened by integer "num". This message id should be put
//    into "value" field of the slot. It should also carry the id
//    of the producer thread to be stored in "thread_id" field so that
//    consumer threads can know which producer generates the message later
//    on. You need to put synchronization code
//    before and after the call ring->Put(message). See the algorithms in
//    page 182 of the textbook.

    for (num = 0; num < N_MESSG ; num++) {
      // Put the code to prepare the message here.
      // ...
      message->value = num;
      message-> thread_id = which;

      // Put the code for synchronization before    ring->Put(message) here.
      // ...
      nempty->P();    //同步在前
      mutex->P();     //互斥在后

      ring->Put(message);

      // Put the code for synchronization after    ring->Put(message) here.
      // ...
      mutex ->V();
      nfull ->V();
    }
}
```

利用类似的方法可以实现消费者进程 void Consumer(_int which)

4.5.4 测试

如果测试结果不能出现 4.3 所示的效果，可加上一个随机数种子，如 nachos –rs 5，以使 Nachos 在初始化内核时创建一个定时器 Timer，实现抢先式的时间片轮转线程调度算法，实现生产者线程与消费者线程之间的分时操作；

第 5 章 Nachos 的文件系统（实验 4）

5.1 目的

Nachos 模拟了一个硬盘（../lab5/DISK，或../filesys/DISK），实现的文件系统比较简单，该实验将熟悉一些文件系统的操作命令，观察这些命令对硬盘（DISK）的影响，根据结果分析理解 Nachos 文件系统的实现原理，为下一个实验（实验 5）实现 Nachos 文件的扩展功能奠定基础；

实验 5 要求能够在从一个文件的任何位置开始写入数据，即能够正确处理命令行参数 -ap, -hap, -nap；

该实验中需要理解一些 Nachos 文件的基本知识，特别是文件头（FCB 或索引节点）的结构与作用，空闲块的标识方法，空闲块的分配与回收过程等；

文件的扩展实质上是从一个给定的位置开始对文件进行写操作，涉及到文件的打开、定位，空闲块的分配等操作，写操作结束后还需要将文件头、空闲块位示图写到硬盘中，以保存修改后的信息。

该实验完成后，需要你：

（1）理解 Nachos 硬盘是如何创建的；

（2）熟悉查看 Nachos 硬盘上的内容的方法；

（3）理解硬盘初始化的过程（如何在硬盘上创建一个文件系统）；

（4）了解 Nachos 文件系统提供了哪些命令，哪些命令已经实现，哪些需要你自己实现；

（5）理解已经实现的文件系统命令的实现原理；

（6）理解硬盘空闲块的管理方法；

（7）理解目录文件的结构与管理；

（8）理解文件的结构与文件数据块的分配方法；

（9）了解一个文件系统命令执行后，硬盘的布局；

（10）分析目前 Nachos 不能对文件进行扩展的原因，考虑解决方案；

5.2 任务

Nachos 实现的文件系统实现了两个版本，依据宏 FILESYS_STUB 与 FILESYS 条件编译产生两个不同的实现（参见../filesys/filesys.h）；宏 FILESYS_STUB 实现的文件操作直接利用 UNIX 所提供的系统调用实现，操作的不是硬盘 DISK 上的文件；宏 FILESYS 实现的文件系统是通过 OpenFile 类对 DISK 上的文件进行操作（尽管最终也是使用 UNIX 的系统调用实现）；

考察../filesys/makefile 及 makefile.local 可以看出，实验 4 与 5 默认是使用宏 FILESYS 所定义的实现，即在硬盘 DISK 上对文件进行操作。

主要代码文件：

../filesys/fstest.cc
/synchdisk.cc
/openfile.cc
/filesys.cc
/directory.cc
/filehdr.cc
../threads/main.cc
../machine/disk.cc
../userprog/bitmap.cc

（1）../lab5/main.cc 调用了../threads/system.cc 中的 Initialize()创建了硬盘 DISK。分析../threads/ synchdisk.cc 及../machine/disk.cc，理解 Nachos 创建硬盘的过程与方法；

（2）分析../lab5/main.cc，了解 Nachos 文件系统提供了哪些命令，对每个命令进行测试，根据执行结果观察哪些命令已经实现（正确运行），哪些无法正确运行（尚未完全实现，需要你自己完善）；

分析../lab5/fstest.cc 及../filesys/filessys.cc，理解 Nachos 对这些命令的处理过程与方法；

（3）分析../filesys/filessys.cc，特别是构造函数 FileSystem::FileSystem(..)，理解 Nachos 硬盘"DISK"的创建及硬盘格式化（创建文件系统）的处理过程；

（4）利用命令 hexdump –C DISK 查看硬盘格式化后硬盘的布局，理解格式化硬盘所完成的工作，以及文件系统管理涉及到的一些数据结构组织与使用，如文件头（FCB）、目录表与目录项、空闲块管理位示图等；

结合输出结果，分析 FileSystem::FileSystem(..)初始化文件系统时涉及到的几个模块，如../filesys/filehdr.h(filehdr.cc)，directory.h(directory.cc)，../userprog/bitmap.h（bitmap.cc），**理解文件头（FCB）的结构与组织、硬盘空闲块管理使用的位示图文件、目录表文件及目录下的组织与结构**，以及它们在硬盘上的位置；

（5）利用命令 nachos –cp ../test/small samll 复制文件../test/small 到硬盘 DISK 中；

（6）利用命令 hexdump –C DISK 查看硬盘格式化后硬盘的布局，理解创建一个文件后相关的结构在硬盘上的存储布局；

（7）复制更多的文件到 DISK 中，然后删除一个文件，利用 hexdump –C DISK 查看文件的布局，分析文件系统的管理策略。

5.3 编译 Nachos 的文件系统

命令终端下进入目录 code/filesys，运行 make，会在该目录下编译生成一个支持文件系统功能的 Nachos 系统。

code/filesys 中的 Makefile 文件包含了目录 code/threads 及 code/filesys 中的 Makefile.local 文件，该目录下的 Makefile 内容如下：

```
ifndef MAKEFILE_FILESYS
define MAKEFILE_FILESYS
yes
endef

# You can re-order the assignments.    If filesys comes before userprog,
# just re-order and comment the includes below as appropriate.

include ../threads/Makefile.local
include ../filesys/Makefile.local
#include ../userprog/Makefile.local
#include ../vm/Makefile.local
#include ../filesys/Makefile.local

include ../Makefile.dep
include ../Makefile.common

endif # MAKEFILE_FILESYS
```

code/filesys 中的 Makefile.local 文件内容如下：

```
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
```

上述信息说明支持文件系统的 Nachos 系统除了使用 code/filesys 目录下的 C++文件外，还使用了 code/threads 以及 code/userprog 目录下的 C++文件。Make 工具通过 code 目录下 Makefile.common 中 VPATH 所定义的路径从这些目录中自动搜寻所需要

的文件。

5.4 Nachos 的硬盘及文件系统

Nachos 利用 UNIX 的系统调用 open()创建了一个名为”DISK”的文件作为 Nachos 的模拟硬盘（参见../machine/disk.h、disk.cc、sysdep.h 及 sysdep.cc）。

硬盘及文件系统具有以下特点：

（1）磁盘开始的 4 个字节（0~3 号字节）是硬盘标识（MagicNumber），其值为 0x456789ab，指明该硬盘是一个 Nachos 硬盘；（参见../machine/disk.cc 中对于魔数的定义以及 Disk 类的构造函数初始化硬盘的过程）

（2）硬盘的第 4 个字节（序号从 0 字节开始）至第 131 字节为其第 0 号扇区（128 字节），其后的 128 个字节为其第 1 号扇区，以此类推。

即，如果字节序号从 0 字节开始，则每个扇区对应的字节序如表 5-1 所示。

表 5-1 Nachos 文件系统硬盘布局

扇区号	起止字节号	存储内容	大小
0	0x4~0x83	**空闲块管理使用的位示图文件头**	128 字节
1	0x84~0x103	**根目录表文件头**	128 字节
2	0x104~0x183	**位示图文件数据块**	128 字节
3	0x184~0x203	根目录表（目录文件）	128 字节
4	0x204~0x283	根目录表（目录文件）	128 字节
5	0x284~0x303	**第一个文件的文件头**	128 字节
6	0x304~0x383	第一个文件的数据块	128 字节
7	0x384~0x403	第一个文件的数据块	128 字节
8	0x404~0x483	……	
9	……	…….	

思考：为什么将空闲块管理的位示图文件头，与目录表文件头存放在 0 号与 1 号这两个特殊的扇区中？

（3）硬盘包括 32 个道，每个道包括 32 个扇区，每个扇区大小是 128 字节；（参见../machine/disk.h，及../filesys/filehdr.h）。

故硬盘 DISK 共有 32*32=1024 个扇区，硬盘大小（DiskSize = (MagicSize + (NumSectors * SectorSize)=（4+32*32*128)B/1024=0x80KB。（参见../machine/disk.h 中个参数的值）

（4）为方便编程实现，将每个逻辑块大小也设置为 128 字节，与一个扇区对应。操作系统中一般都是一个逻辑块包含 2^n 个扇区（n>0）；

（5）采用一级目录结构（单级目录结构），最多可创建 10 个文件

（6）一个目录文件由“文件头+目录表”组成

目录文件 Directory 中的每个文件目录项包含三项：（参见../filesys/directory.h）

- inUser：　//该目录项是否已经分配
- int sector; // 文件头所在的扇区号，这里文件头是 FCB 或者是 i-node （文件头结构参见../filesys/filehdr.h）；
- char name[FileNameMaxLen + 1]; // 文件名，定义最长为 9 个字节，+1：末尾 '\0'

注 1：Nachos 的目录项采用的是 UNIX 的思想，即文件名+i-node（也就是 FCB）

注 2：系统初始化（创建）文件系统时，在目录文件中初始化了 10 个目录项，也就是说该文件系统中目前最多只能创建 10 个文件；（目录项内容参见../filesys/directory.h，文件系统创建参见../filesys/filesys.cc 的构造函数）

注 3：系统将目录看做一个文件，即目录文件，也包括一个文件头（i-node）内容是目录表，目录表由多个目录项组成，每个目录项由<文件名+文件头（i-node）+目录项状态>三部分组成。Nachos 所采用的目录结构类似于我们所熟悉的 UNIX 名号目录项。目录表及目录项结构如表 5-2 所示。

表 5-2 Nachos 文件目录表（目录文件结构）

文件名	inUse	文件头（索引节点）所在扇区号
main.cc	1	4

注 4：文件头（i-node）结构参见../filesys/filehdr.h；

（7）一个文件由“文件头+数据块组成”，

每个文件最多包括 30 个扇区(NumDirect =(SectorSize - 2 * sizeof(int)) / sizeof(int))（参见../filesys/filehdr.h），因此每个文件最大为 3780 字节（3KB）（30*128 字节）（参见../filesys/filehdr.h））；

（8）文件在硬盘上的分配方法及文件头（参见../filesys/filehdr.h 及../filesys/filehdr.cc）

文件头相当于 FCB（i-node in UNIX），说明文件的属性，以及文件的数据块在硬盘上的位置，在磁盘其结构如下：

- int numBytes; // 文件大小，以字节为单位
- int numSectors; // 文件的逻辑块数，这里就是扇区数
- int dataSectors[NumDirect]; // 直接块数组，依次存储文件的每个数据块所对应的扇区号；

注：文件在磁盘上的分配方法一般有三种：连续文件、链接文件及索引文件；

Nachos 的文件系统将文件的数据分配到连续的扇区中，并依次将各数据块所在的扇区号记录在数组 dataSectors[NumDirect]中，因此采用的是类似于 UNIX 中 i-node 的直接块的索引方式（没有采用多级索引）。

Nachos 一个文件头大小为 128 字节，恰好占用一个扇区（Nachos 有意将一个文件头存储到一个扇区中），由于 int numBytes 与 int numSectors 已经占用两个整数，因此直接块 dataSectors[NumDirect]数组的最大项数由下式确定：NumDirect =(SectorSize - 2 * sizeof(int)) / sizeof(int))=(128-2*4)/4=30 块，即每个文件的数据最多存储到 30 个

扇区中，因此每个文件最大为 30*128 字节=3KB，一个文件头大小=4+4+30*4=128 字节。（参见../filesys/filehdr.h 及../filesys/filehdr.cc）

（9）硬盘空闲块的管理

硬盘采用位示图（BitMap）的思想管理硬盘的空闲块，即根据硬盘的扇区数建立一个位置图（参见../filesys/filesys.cc 中 FileSystem 的构造函数，及../userprog/bitmap.h 及 bitmap.cc）；

当一个扇区空闲，位示图中对应的位为 0，否则为 1；

位示图也是一个文件，由文件头+数据块组成，文件头保存在第 0 号扇区中；

由于硬盘共有 1024 个扇区，需要 1024 位表示每个扇区的状态，因此位示图文件大小应为 1024/8=128 字节，故硬盘的位示图文件也可以恰好存储在一个扇区中。

成员函数 BitMap::FetchFrom()与 BitMap::WriteBack()可以将位示图文件读入内存及写入硬盘。

（11）目录文件（根目录）的文件头存储在第 1 号扇区；

注：关于 0 号与 1 号扇区及位示图文件头及目录表文件头

在初始化文件系统时（FileSystem 类的构造函数），将这两个特殊的扇区置位（已使用），然后将硬盘位示图文件头与目录表文件头写入到这两个特殊的扇区中；

对于一个真正的操作系统，由于系统启动时需要根据目录文件的文件头访问根目录，因此为这两个特殊的结构分配到 0 号扇区与 1 号扇区这两个特殊的扇区中，是为了便于系统启动时从已知的、固定的位置访问它们。

（12）当文件创建后，其大小不能改变；例如当复制一个文件到 Nachos 的硬盘 DISK 中，该文件大小将无法改变。

综上所述，Nachos 文件系统在硬盘 DISK 的布局如表 5-3 所示。

表 5-3 Nachos 的磁盘布局

起止字节	内容描述	内容	扇区号	大小
0x0~0x3	磁盘标识（魔数）	0x456789ab		4 字节
0x4~0x83	位示图文件头	Class FileHeader	0	128 字节
0x84~0x103	目录表文件头	Class FileHeader	1	128 字节
0x104~0x183	位示图文件数据块	Class BitMap	2	128 字节
0x184~0x203	根目录表（目录文件）	Class Directory	3	128 字节
0x204~0x283	根目录表（目录文件）	Class Directory	4	128 字节
0x284~0x303	第一个文件的文件头	Class FileHeader	5	128 字节
0x304~0x383	第一个文件的数据块		6	128 字节
	(文件需要的块数不定)	…….	7	
	第二个文件的文件头	Class FileHeader		128 字节
	第二个文件的数据块			128 字节
	……	……		
	第三个文件的文件头	Class FileHeader		128 字节
	第三个文件的数据块			128 字节
	…..	……		
	以此类推			

5.5 Nachos 的文件系统命令

这些文件操作命令在文件 ../threads/main.cc 及文件../threads/system.cc 中进行了定义说明。与文件系统相关的命令如下：

注：可选参数[-d f]的作用是打印出所有与文件系统有关的调试信息。

nachos [-d f] –f：格式化 Nachos 模拟的硬盘 DISK，在使用其它文件系统命令之前需要将该硬盘格式化；

格式化硬盘（参见../disk.cc 中 DISK 类的构造函数）所做的工作就是在硬盘 DISK 上创建一个文件系统（参见../filesys/filesys 中 FileSystem 类的构造函数），初始化用于空闲块管理的位示图文件及目录文件后，将位示图文件的文件头写入 0 号扇区，将目录表文件的头文件写入 1 号扇区，并为上述两个文件数据分配扇区后，再将位示图文件的数据块（128=0x80 字节）写入 2 号扇区，将目录文件的数据块（200=0xC8 字节）写入 3 号及 4 号扇区中。。

nachos [-d f] –cp UNIX_filename nachos_filename：将一个 Unix 文件系统中的文件 **UNIX_filename** 复制到 Nachos 文件系统中，重新命名为 **nachos_filename；**

目前实现的 Nachos 文件系统尚未提供 creat()系统调用，也就没有提供创建文件的命令。如果要在 Nachos 的硬盘中创建文件，目前只能通过该命令从你的 UNIX 系统中复制一个文件到 Nachos 硬盘中；

nachos [-d f] –p nachos_filename：该命令输出 nachos 文件 **nachos_filename** 的内容，类似于 UNIX 中的 cat 命令；

nachos [-d f] –r nachos_filename：删除 Nachos 文件 **nachos_filename**，类似于 UNIX 中的 rm 命令；

nachos [-d f] -l：输出当前目录中的文件名（类似于 DOS 中的 dir，UNIX 中的 ls）；

nachos [-d f] -t：测试 Nachos 文件系统的性能（目前尚未实现）；

nachos [-d f] -D：输出 Nachos 的文件系统在磁盘上的组织。打印出整个文件系统的所有内容，包括位图文件（bitmap）、文件头（file header）、目录文件（directory）和普通文件（file）

5.6 Nachos 提供的三个测试文件

目录../filesys/test 中的三个文件 small、medium 以及 big 将用于测试 Nachos 的文件系统，可以先看看它们的内容。

在调试 Nachos 的文件系统之前，需要使用 UNIX 的命令 **od（Octal Dump）**或 **hexdump（Hexadecimal Dump）**检查模拟硬盘 DISK 的内容。

5.6.1 UNIX 命令 od

在 UNIX 命令终端中进入目录../filesys，运行 od –c test/small，屏幕将输出：

```
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
```

每一行输出 16 个字符，字符在文件中的偏移量（左边数字）以 8 进制表示；

5.6.2 UNIX 命令 hexdump

在 UNIX 命令终端中进入目录../filesys，运行 hexdump -c test/small.，屏幕将输出：

```
0000000   T   h   i   s       i   s       t   h   e       s   p   r   i
0000010   n   g       o   f       o   u   r       d   i   s   c   o   n
0000020   t   e   n   t   .  \n
0000026
```

偏移量以 16 进制表示。

命令 hexdump -C test/small 以 ASCII 形式输出文件内容：

```
00000000  54 68 69 73 20 69 73 20  74 68 65 20 73 70 72 69  |This is the spri|
00000010  6e 67 20 6f  66 20 6f 75  72 20 64 69 73 63  6f 6e  |ng of our discon|
00000020  74 65 6e 74 2e 0a                                 |tent..|
00000026
```

关于 od 与 hexdump 的进一步使用可参阅 man od 及 man hexdump。

5.7 Things to Do

结合下列内容，完成 5.2 实验任务中要求完成的工作；

5.7.1 编译生成 Nachos 文件系统

Linux 命令终端中进入目录../filesys，键入 make 生成支持文件系统的 Nachos。

5.7.2 测试 Nachos 文件系统

在 Linux 目录终端中进入目录../filesys，运行下述 nachos 命令，查看输出结果：

（a）运行 nachos –f，将在当前目录下创建一个 Nachos 模拟硬盘 DISK 并创建一个文件系统；

（b）运行 nachos –D，显示硬盘 DISK 中的文件系统，屏幕输出：

```
Bit map file header :
FileHeader contents.    File size: 128. File blocks:
2
File contents:
\1f\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Directory file header:
FileHeader contents.    File size: 200. File blocks:
3   4
File contents:
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Bitmap set:
0, 1, 2, 3, 4,
Directory contents:

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

Ticks: total 5310, idle 5000, system 310, 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...
```

上述信息说明在硬盘 DISK 上已经成功创建了一个 Nachos 文件系统，不过此时该文件系统的唯一目录中没有任何文件。

（c）运行 **od –c DISK**，屏幕将输出下述转储信息：

```
0000000   ⟐   211  g   E   200  \0  \0  \0  001  \0  \0  \0  002  \0  \0  \0
0000020   \0  \0  \0  \0  \0  \0  \0  \0  \0   \0  \0  \0  \0   \0  \0  \0
*
0000200   \0  \0  \0  \0 310 \0  \0  \0 002   \0  \0  \0  003   \0  \0  \0
0000220  004  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000240   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000400   \0  \0  \0  \0  037   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000420   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0400000   \0  \0  \0  \0
0400004
```

（d）运行 **hexdump -c DISK**，屏幕将输出下述转储信息：

```
0000000   255   211   g   E   200    \0    \0    \0   001    \0    \0    \0   002    \0    \0    \0
0000010    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0
*
0000080    \0    \0    \0    \0    ?    \0    \0    \0 002    \0    \0    \0   003    \0    \0    \0
0000090   004    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0
00000a0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0
*
0000100    \0    \0    \0    \0   037    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0
0000110    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0    \0
*
0020004
```

（e）运行 **hexdump -C DISK**，屏幕将输出下述转储信息：

```
0000000 ab 89 67 45 80 00 00 00   00 00 00 00 00 00 00 00   |. . g E . . . . . . . . . . . .|
0000010 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   |. . . . . . . . . . . . . . . .|
*
0000080 00 00 00 00 c 8 00 00 00   02 00 00 00 03 00 00 00   |. . . . . . . . . . . . . . . .|
0000090 04 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   |. . . . . . . . . . . . . . . .|
00000a0 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   |. . . . . . . . . . . . . . . .|
*
0000100 00 00 00 00 1f 00 00 00   00 00 00 00 00 00 00 00   |. . . . . . . . . . . . . . . .|
0000110 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   |. . . . . . . . . . . . . . . .|
*
0020004
```

或如图 5-1 所示。

```
han@han-VirtualBox:~/NACHOS/nachos-3 .4-SDU-test/code/lab5$ hexdump -C DISK
00000000  ab 89 67 45 80 00 00 00  01 00 00 00 02 00 00 00  |..gE............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000080  00 00 00 00 c8 00 00 00  02 00 00 00 03 00 00 00  |................|
00000090  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000100  00 00 00 00 1f 00 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00020004
```

图 5-1 初始化 Nachos 的文件系统

（f）运行 **nachos -cp test/small small**，将 test 目录下的 UNIX 文件 small 复制到 Nachos 文件系统中；运行命令 nachos –l，nachos –p small 以及 nachos –D，根据输出结果检查 Nachos 文件系统中是否存在文件 small（在硬盘 DISK 中存储）。

利用命令 od –c DISK，hexdump –c DISK 及 hexdump –C DISK 根据转储内容查看

硬盘 DISK 有何变化。

（g）利用 **nachos -cp test/medium medium**，及 **nachos -cp test/big big** 将 UNIX 文件复制到 Nachos 文件系统中，重复（b）~（f）；

（h）结合 nachos –r 命令，删除 Nachos 硬盘上的某个文件，重复（b）~（f）；

5.8 Questions

（a）利用 nachos –cp 命令复制几个 UNIX 文件到 Nachos 文件系统后，运行 nachos –D， od –c DISK（and/or hexdump –c DISK, hexdump –C DISK），根据输出结果查看硬盘 DISK 上有几个文件？

（b）文件 big 的数据块（data blocks）的扇区号是多少？

（c）文件 big 的文件头（file header）的扇区号是多少？

（e）Nachos 硬盘的扇区大小是 128 字节。你能根据 od –c DISK（and/or hexdump –c DISK, hexdump –C DISK）命令的输出结果确认文件 big 的数据块及文件头（the data blocks and the file header of big）处于硬盘正确位置吗？

5.9 Nachos 文件系统在硬盘上的布局

5.9.1 硬盘格式化

（1）将“DISK”删除

（2）nachos –f 格式化硬盘（在硬盘 DISK 上创建一个文件系统）

（3）hexdump –C DISK，屏幕输出如图 5-2 所示。

```
han@han-VirtualBox:~/NACHOS/nachos-3 .4-SDU-test/code/lab5$ hexdump -C DISK
00000000  ab 89 67 45 80 00 00 00  01 00 00 00 02 00 00 00  |..gE............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000080  00 00 00 00 c8 00 00 00  02 00 00 00 03 00 00 00  |................|
00000090  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000100  00 00 00 00 1f 00 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00020004
```

图 5-2 Nachos 文件系统初始化状态

数据分析解读：

（a）0x0~0x3 字节：硬盘开始的 4 个字节（0x0~0x3）是该磁盘的标识（魔数），为 0x5678ab（参见 ../machine/disk/h 、disk.cc 及../filesys/filesys.h、../filesys/filesys.cc）；

（b）0x4~0x83：扇区 0，128 字节；存放位示图头文件（FCB，i-node）；

根据../filesys/filehdr.h 中对文件头的定义，文件头由三部分组成：（int numBytes, int numSectors, int dataSectors[NumDirect];），即文件头所描述的三元组<文件的大小，占用的扇区数，各数据块所在扇区列表>。

Nachos 将头文件的大小设计为 128 个字节，恰好保存到一个扇区中。

- 0x4~0x7：4 个字节，位示图文件大小（注：是位示图文件，不是头文件），值为 0x80，表示位示图文件大小为 0x80=128 字节
- 0x8~0xB：4 个字节，系统为位示图文件数据所分配的扇区数，其值为 0x1，表示位示图文件数据只需要一个扇区；
- 0xC~0xF：位示图文件数据块所在的扇区号，其值为 0x2，说明系统将位示图文件的数据保存在第 2 号扇区中。

（c）0x84~0x103：扇区 1，128 字节；存放目录表（根目录文件）头文件（FCB，i-node）；

- 0x84~0x87：4 个字节，目录表文件大小，值为 0xC8，表示目录表文件大小为 0xC8=200 字节；
  注：关于目录文件的大小，一个目录项（DirectoryEntry）大小为 20 个字节，Nachos 为目录文件建立了 10 个目录项（该文件系统最多可创建 10 个文件），因此目录文件大小为 200 字节；
- 0x88~0x8B：4 个字节，系统为目录文件数据所分配的扇区数，其值为 0x2，表示目录文件数据需要 2 个扇区（目录文件大小为 200 字节，需要占用两个扇区）；
- 0x8C~0x8F：目录表文件第 1 个数据块所在的扇区号，其值为 0x3，说明系统将目录表文件第 1 个数据块保存在第 3 号扇区中；
- 0x90~0x93：目录表文件第 2 个数据块所在的扇区号，其值为 0x4，说明系统将目录表文件第 2 个数据块保存在第 4 号扇区中；

（d）0x104~0x183：扇区 2，128 字节；位示图的数据块，存储位示图文件内容；

（e）0x184~0x203，扇区 3，0x204~0x283：扇区 4：这两个扇区是目录文件的数据块，存放目录项；目前没有任何文件，值为 0；

目前只有 5 个扇区被分配（扇区 0,1,2,3,4），其余都空闲，因此在位示图中只有这 5 个扇区对应的位被置 1，其余均为 0，所以位示图文件内容是：11111000 0000…..0000（1024 位），第一个字节 11111000 在 Nachos 中表示成 0x1F；

nachos -D 的输出也说明了上述观点，如图 5-3 所示。

```
han@han-VirtualBox:~/NACHOS/nachos-3 .4-SDU-test/code/lab5$ ./nachos -D
Bit map file header:
FileHeader contents.  File size: 128.  File blocks:
2
File contents:
\1f\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
0\0\0\0\0\0\0\0\0
Directory file header:
FileHeader contents.  File size: 200.  File blocks:
3 4
File contents:
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Bitmap set:
0, 1, 2, 3, 4,
Directory contents:

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

Ticks: total 5310, idle 5000, system 310, 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...
han@han-VirtualBox:~/NACHOS/nachos-3 .4-SDU-test/code/lab5$
```

图 5-3 nachos　-D 显示文件系统的初始化状态

5.9.2 复制一个文件到硬盘

（1）nachos –cp test/small small 复制一个 Linux 文件文件 small 到 NachosDISK 中；

（2）hexdump –C DISK，屏幕输出如图 5-4 所示。

```
han@han-VirtualBox:~/NACHOS/nachos-3 .4-SDU-test/code/lab5$ hexdump -C DISK
00000000  ab 89 67 45 80 00 00 00  01 00 00 00 02 00 00 00  |..gE............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000080  00 00 00 00 c8 00 00 00  02 00 00 00 03 00 00 00  |................|
00000090  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000100  00 00 00 00 7f 00 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000180  00 00 00 00 01 00 00 00  05 00 00 00 73 6d 61 6c  |............smal|
00000190  6c 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |l...............|
000001a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000280  00 00 00 00 54 00 00 00  01 00 00 00 06 00 00 00  |....T...........|
00000290  71 00 00 00 b0 47 dd b7  b0 47 dd b7 00 00 00 00  |q....G...G......|
000002a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000002f0  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  |................|
00000300  10 00 00 00 73 6d 61 6c  6c 20 66 69 6c 65 20 73  |....small file s|
00000310  6d 61 6c 6c 20 66 69 6c  65 20 73 6d 61 6c 6c 20  |mall file small |
00000320  66 69 6c 65 0a 73 6d 61  6c 6c 20 66 69 6c 65 20  |file.small file |
00000330  73 6d 61 6c 6c 20 66 69  6c 65 20 73 6d 61 6c 6c  |small file small|
00000340  20 66 69 6c 65 0a 2a 2a  2a 65 6e 64 20 6f 66 20  | file.***end of |
00000350  66 69 6c 65 2a 2a 2a 0a  00 00 00 00 00 00 00 00  |file***.........|
00000360  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00020004
```

图 5-4 复制一个文件 small 到硬盘

数据解读：

（a）0 号扇区（0x4~0x83）与 1 号扇区（0x84~0x103）存储的位示图文件头及目录表文件头内容不变，改变的是它们对应的文件内容；

（b） 2 号扇区（0x104~0x183）存储的位示图文件内容发生改变，位示图文件内容由 11111000 0000…..0000（1024 位）改变为 1111 1110 0000…..0000（1024 位），第一个字节 11111110 在 Nachos 中表示成 0x7F，说明 0~6 号 7 个扇区被使用；

（c）3 号扇区（0x184~0x203）存储的目录表也发生了改变，因为我们新建了一个文件 small，需要在一个空闲的目录项中添加文件 small 对应的信息；目录表有 10 个目录项组成，每个目录项是一个三元组< bool inUse， int sector， char name[FileNameMaxLen + 1]>，由上图中可以看出：

- 0x184~0x187 所示的 4 个字节对应三元组中的 inUse（注：尽管 inUse 占用一个字节，但根据编译器对各成员变量的起始地址对齐的原则，为 inUse 分配与 sector 相同的字节数，4 个字节）表示该目录项是否已被使用，该值为 1，说明该目录项正被一个文件使用。
- 0x188~0x18B 所示的 4 个字节对应三元组中的 sector，说明该目录项所记录的文件（即文件 samll）的文件头所在的扇区号，这里是 5，说明该目录项对应的文件（small）的文件头在第 5 号扇区；若要考察文件 small 的详细信息，需要到 5 号扇区访问其文件头。
- 0x18C~0x195 的 10 个字节是文件名（文件名占用 9 个字节，最后一个字

节是字符串结束符'\0')，这里文件名是 small；目前该硬盘上只有一个文件（samll)，目录文件中的其余目录项都是空的（目前共 19 个目录项)（3 号扇区其余部分及 4 号扇区全部都是空的)；

（d）4 号扇区（0x204~0x283)：目录表文件的第 2 个扇区，目前是空的；

（e）5 号扇区（0x284~0x303）是“samll”的文件头；

根据../filesys/filehdr.h 所定义的文件头是一个三元组< int numBytes; int numSectors; int dataSectors[NumDirect];>，分别指明文件数据大小，文件数据所占用的扇区数，以及文件数据所分配扇区索引表。

根据文件头的三元组（FCB，i-node）信息，考察 small 文件的属性：

- 0x284~0x287：4 个字节，文件大小，该值是 0x54，说明 small 文件大小是 0x54 个字节；
- 0x288~0x28B：4 个字节，系统为文件数据所分配的扇区数，该值为 1，说明系统只为文件 small 的数据分配了一个扇区(一个扇区大小为 128 字节，文件 small 的大小为 84 字节）
- 0x28C~0x28F：系统为文件数据所分配的扇区列表。这里 samll 文件数据只需一个扇区，将其存放在第 6 号扇区；
  如果想考察 samll 文件的内容，需要访问第 6 号扇区。
- 理论上讲，该扇区其余内容都应该为空(从 0x290~0x303)。由于 FileHeader 类没有显示的构造函数，编译器就为其自动设定了构造函数 FileHeader:: FileHeader {}，为该类实例化对象时，所分配的内存可能含有信息，因此可以 0x290~0x303 的内容，在此没有意义。
  注：最好自己为 FileHeader 类定义构造函数，将成员变量进行初始化。如构造函数如下：

```
FileHeader:: FileHeader() {
        numBytes=0;
        numSectors=0;
        for (int i=0;i<NumDirect;i++)
            dataSectors[i]=0;
}
```

（f）6 号扇区（0x304~0x383）是“samll”的文件的数据块，其中 0x304~0x357（0x357-0x304+1=0x54 是文件的长度）是文件内容，其余空闲；可以考察../filesys/test/small 文件的内容，与这里显示的内容进行比较，查看文件在硬盘的存储方式。其中 0x324 中的 0x0a 是换行符。

（g）小结

0x4~0x83：　0 号扇区，位示图文件头
0x84~0x103：　1 号扇区，目录表文件头
0x104~0x183；2 号扇区，位示图文件数据块
0x184~0x203：3 号扇区，目录表文件第一个数据块
0x204~0x283：4 号扇区，目录表文件第二个数据块

0x284~0x303：5 号扇区，small 文件头
0x304~0x383：6 号扇区，small 文件数据块

nachos　-D 的输出也说明了上述观点，如图 5-5 所示。

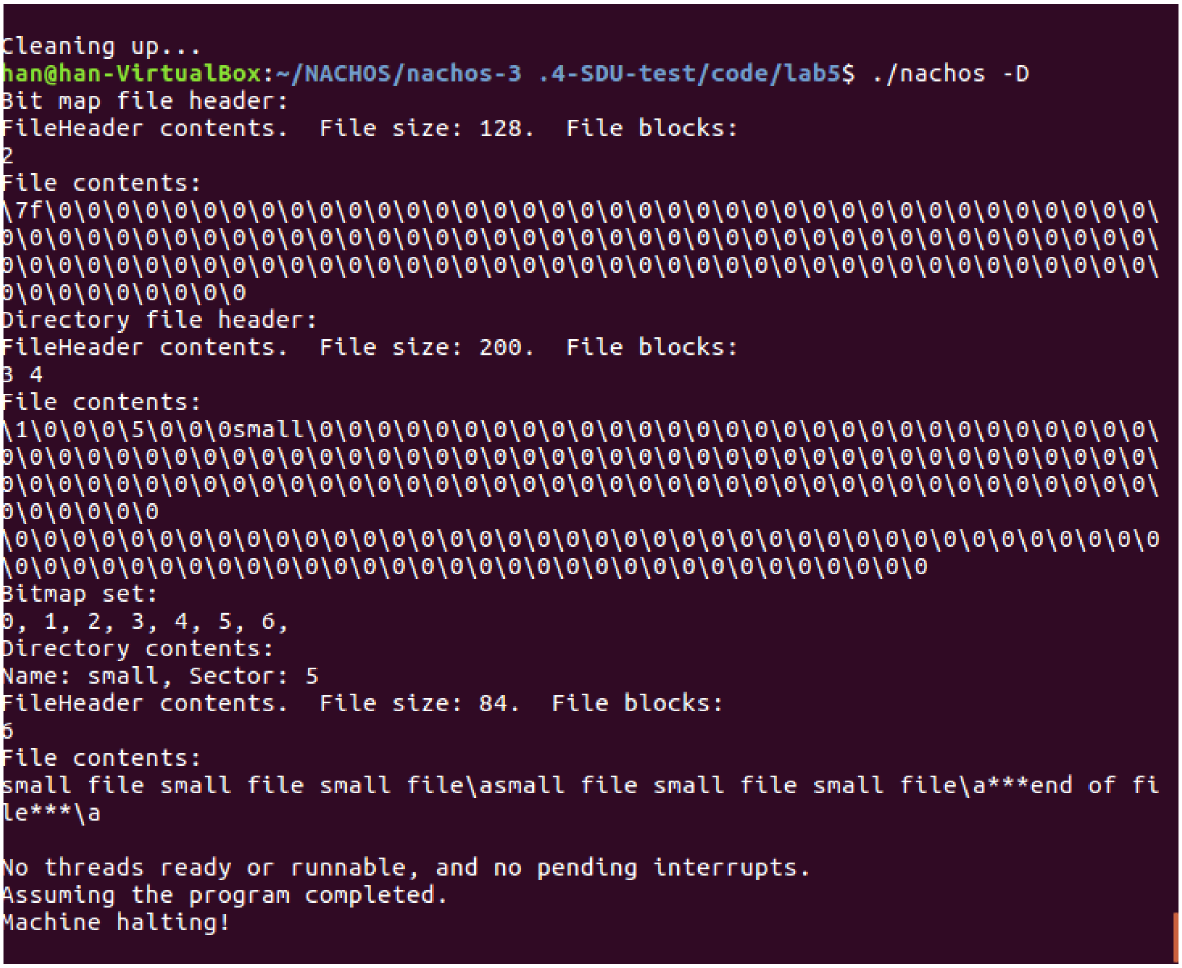


图 5-5 复制一个文件 small 到硬盘后 nachos –D 的输出

5.9.3 复制另一个文件到硬盘

（1）nachos –cp test/big big 复制另一个一个 Linux 文件 big 到 NachosDISK 中；
（2）hexdump –C DISK，屏幕输出如图 5-6 所示。

```
han@han-VirtualBox:~/NACHOS/nachos-3 .4-SDU-test/code/lab5$ hexdump -C DISK
00000000  ab 89 67 45 80 00 00 00  01 00 00 00 02 00 00 00  |..gE............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000080  00 00 00 00 c8 00 00 00  02 00 00 00 03 00 00 00  |................|
00000090  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000100  00 00 00 00 ff 07 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000180  00 00 00 00 01 00 00 00  05 00 00 00 73 6d 61 6c  |............smal|
00000190  6c 00 00 00 00 00 00 00  01 00 00 00 07 00 00 00  |l...............|
000001a0  62 69 67 00 00 00 00 00  00 00 00 00 00 00 00 00  |big.............|
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000280  00 00 00 00 54 00 00 00  01 00 00 00 06 00 00 00  |....T...........|
00000290  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000300  00 00 00 00 73 6d 61 6c  6c 20 66 69 6c 65 20 73  |....small file s|
00000310  6d 61 6c 6c 20 66 69 6c  65 20 73 6d 61 6c 6c 20  |mall file small |
00000320  66 69 6c 65 0a 73 6d 61  6c 6c 20 66 69 6c 65 20  |file.small file |
00000330  73 6d 61 6c 6c 20 66 69  6c 65 20 73 6d 61 6c 6c  |small file small|
00000340  20 66 69 6c 65 0a 2a 2a  2a 65 6e 64 20 6f 66 20  | file.***end of |
00000350  66 69 6c 65 2a 2a 2a 0a  00 00 00 00 00 00 00 00  |file***.........|
00000360  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000380  00 00 00 00 54 01 00 00  03 00 00 00 08 00 00 00  |....T...........|
00000390  09 00 00 00 0a 00 00 00  00 00 00 00 00 00 00 00  |................|
000003a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  00 00 00 00 62 69 67 20  66 69 6c 65 20 62 69 67  |....big file big|
00000410  20 66 69 6c 65 20 62 69  67 20 66 69 6c 65 20 62  | file big file b|
00000420  69 67 20 66 69 6c 65 20  62 69 67 20 66 69 6c 65  |ig file big file|
00000430  20 0a 62 69 67 20 66 69  6c 65 20 62 69 67 20 66  | .big file big f|
00000440  69 6c 65 20 62 69 67 20  66 69 6c 65 20 62 69 67  |ile big file big|
00000450  20 66 69 6c 65 20 62 69  67 20 66 69 6c 65 20 0a  | file big file .|
00000460  62 69 67 20 66 69 6c 65  20 62 69 67 20 66 69 6c  |big file big fil|
00000470  65 20 62 69 67 20 66 69  6c 65 20 62 69 67 20 66  |e big file big f|
00000480  69 6c 65 20 62 69 67 20  66 69 6c 65 20 0a 62 69  |ile big file .bi|
00000490  67 20 66 69 6c 65 20 62  69 67 20 66 69 6c 65 20  |g file big file |
000004a0  62 69 67 20 66 69 6c 65  20 62 69 67 20 66 69 6c  |big file big fil|
000004b0  65 20 62 69 67 20 66 69  6c 65 20 0a 62 69 67 20  |e big file .big |
000004c0  66 69 6c 65 20 62 69 67  20 66 69 6c 65 20 62 69  |file big file bi|
000004d0  67 20 66 69 6c 65 20 62  69 67 20 66 69 6c 65 20  |g file big file |
000004e0  62 69 67 20 66 69 6c 65  20 0a 62 69 67 20 66 69  |big file .big fi|
000004f0  6c 65 20 62 69 67 20 66  69 6c 65 20 62 69 67 20  |le big file big |
00000500  66 69 6c 65 20 62 69 67  20 66 69 6c 65 20 62 69  |file big file bi|
00000510  67 20 66 69 6c 65 20 0a  62 69 67 20 66 69 6c 65  |g file .big file|
00000520  20 62 69 67 20 66 69 6c  65 20 62 69 67 20 66 69  | big file big fi|
00000530  6c 65 20 62 69 67 20 66  69 6c 65 20 62 69 67 20  |le big file big |
00000540  66 69 6c 65 20 0a 2a 2a  2a 65 6e 64 20 6f 66 20  |file .***end of |
00000550  66 69 6c 65 2a 2a 2a 0a  00 00 00 00 00 00 00 00  |file***.........|
00000560  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00020004
```

图 5-6 复制第二个文件 big 到硬盘

数据解读：

（a）0 号扇区（0x4~0x83）与 1 号扇区（0x84~0x103）存储的位示图文件头及目

录表文件头内容不变，改变的是它们对应的文件内容；

（b） 2 号扇区（0x104~0x183）存储的位示图文件内容发生改变，位示图文件内容由 11111000 0000…..0000（1024 位）改变为 1111 1111 1110…..0000（1024 位），前 3 个字节 1111 1111 1110 在 Nachos 中表示成 0xFF7（复制 big 文件之前是 0x7F，比原来多了 4 个被使用的扇区），说明 0~0xa 号 11 个扇区被使用；

多出的 4 个扇区：big 文件头 1 个，big 数据块 3 个；

（c）3 号扇区（0x184~0x203）存储的目录表也发生了改变，增加了 big 文件的目录项（每个目录项占用 20 字节）；

（d）7 号扇区（0x384~0x403），big 文件的文件头；

samll 的文件的数据块存储在 6 号扇区，其后的第 7 号扇区分配给了 big 文件的文件头；

- 0x384~0x387：4 个字节，big 文件大小是 0x154 字节；
- 0x388~0x38B：4 个字节，big 文件的数据占用了 3 个扇区（一个扇区大小为 128 字节，文件 big 的大小为 340 个字节，需要分配 3 个扇区）
- 0x38C~0x38F：系统将 big 文件的数据分配到第 0x8、0x9、0xa 号 3 个扇区中；如果想考察 big 文件的内容，需要访问这三个扇区。

（e）8 号扇区（0x404~0x483）、9 号扇区（0x484~0x503）、10 号扇区（0x504~0x583）存放 big 文件的数据；

（f）小结

0x4~0x83： 0 号扇区，位示图文件头
0x84~0x103： 1 号扇区，目录表文件头
0x104~0x183：2 号扇区，位示图文件数据块
0x184~0x203：3 号扇区，目录表文件第一个数据块
0x204~0x283：4 号扇区，目录表文件第二个数据块
0x284~0x303：5 号扇区，small 文件头
0x304~0x383：6 号扇区，small 文件数据块
0x384~0x403：7 号扇区，big 文件头
0x404~0x483：8 号扇区，big 文件第 1 块数据（开始的 128 字节）
0x484~0x503：9 号扇区，big 文件第 2 块数据（中间的 128 字节）
0x504~0x583：10 号扇区，big 文件第 3 块数据（最后的 84 字节）

nachos -D 的输出也说明了上述观点，如图 5-7 所示。

```
han@han-VirtualBox:~/NACHOS/nachos-3 .4-SDU-test/code/lab5$ ./nachos -D
Bit map file header:
FileHeader contents.  File size: 128.  File blocks:
2
File contents:
\ff\7\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
0\0\0\0\0\0\0\0\0
Directory file header:
FileHeader contents.  File size: 200.  File blocks:
3 4
File contents:
\1\0\0\0\5\0\0\0small\0\0\0\0\0\0\0\1\0\0\0\7\0\0\0big\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\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,
Directory contents:
Name: small, Sector: 5
FileHeader contents.  File size: 84.  File blocks:
6
File contents:
small file small file small file\asmall file small file small file\a***end of fi
le***\a
Name: big, Sector: 7
FileHeader contents.  File size: 340.  File blocks:
8 9 10
File contents:
big file big file big file big file big file \abig file big file big file big fi
le big file \abig file big file big file big file
big file \abig file big file big file big file big file \abig file big file big
```

图 5-7 复制第二个文件 big 到硬盘（nachos -D）

5.9.4 删除 Nachos 硬盘上的文件

（1）nachos –r　small，删除 small 文件；

（2）hexdump –C DISK，屏幕输出如图 5-8 所示。

```
han@han-VirtualBox:~/NACHOS/nachos-3 .4-SDU-test/code/lab5$ hexdump -C DISK
00000000  ab 89 67 45 80 00 00 00  01 00 00 00 02 00 00 00  |..gE............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000080  00 00 00 00 c8 00 00 00  02 00 00 00 03 00 00 00  |................|
00000090  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000100  00 00 00 00 9f 07 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000180  00 00 00 00 00 00 00 00  05 00 00 00 73 6d 61 6c  |............smal|
00000190  6c 00 00 00 00 00 00 00  01 00 00 00 07 00 00 00  |l...............|
000001a0  62 69 67 00 00 00 00 00  00 00 00 00 00 00 00 00  |big.............|
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000280  00 00 00 00 54 00 00 00  01 00 00 00 06 00 00 00  |....T...........|
00000290  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000300  00 00 00 00 73 6d 61 6c  6c 20 66 69 6c 65 20 73  |....small file s|
00000310  6d 61 6c 6c 20 66 69 6c  65 20 73 6d 61 6c 6c 20  |mall file small |
00000320  66 69 6c 65 0a 73 6d 61  6c 6c 20 66 69 6c 65 20  |file.small file |
00000330  73 6d 61 6c 6c 20 66 69  6c 65 20 73 6d 61 6c 6c  |small file small|
00000340  20 66 69 6c 65 0a 2a 2a  2a 65 6e 64 20 6f 66 20  | file.***end of |
00000350  66 69 6c 65 2a 2a 2a 0a  00 00 00 00 00 00 00 00  |file***.........|
00000360  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000380  00 00 00 00 54 01 00 00  03 00 00 00 08 00 00 00  |....T...........|
00000390  09 00 00 00 0a 00 00 00  00 00 00 00 00 00 00 00  |................|
000003a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  00 00 00 00 62 69 67 20  66 69 6c 65 20 62 69 67  |....big file big|
00000410  20 66 69 6c 65 20 62 69  67 20 66 69 6c 65 20 62  | file big file b|
```

图 5-8 删除文件 small

删除 small 文件的过程：

（a）依据 small 文件头所提供的 small 文件数据块所在的扇区号，在位示图中清除 samll 文件数据块所占用的扇区（6 号扇区），然后清除 samll 文件头所占用的扇区；上图中可以看出，位示图数据所在的 2 号扇区中，0x104、0x105 的内容由 0xFF7 修改为 0x9F7，即 small 文件头所占用的 5 号扇区及数据块所占用的 6 号扇区变为空闲；

（b）清除目录表中为 samll 文件所分配的目录项中的 inUse 位（3 号扇区中 0x184 内容 1 变为 0，即位 small 文件分配的目录项变为空闲）

（c）可以看出，删除一个文件后，该文件在目录表中文件名、文件头所占的扇区号均未清除，只是将该目录项变为空闲；

文件头中的信息（文件大小、文件所占用的扇区数以及为文件数据分配的扇区列表）也未清除；

文件的内容也未清除；

命令 nachos –D 的输出也说明了上述观点，如图 5-9 所示。

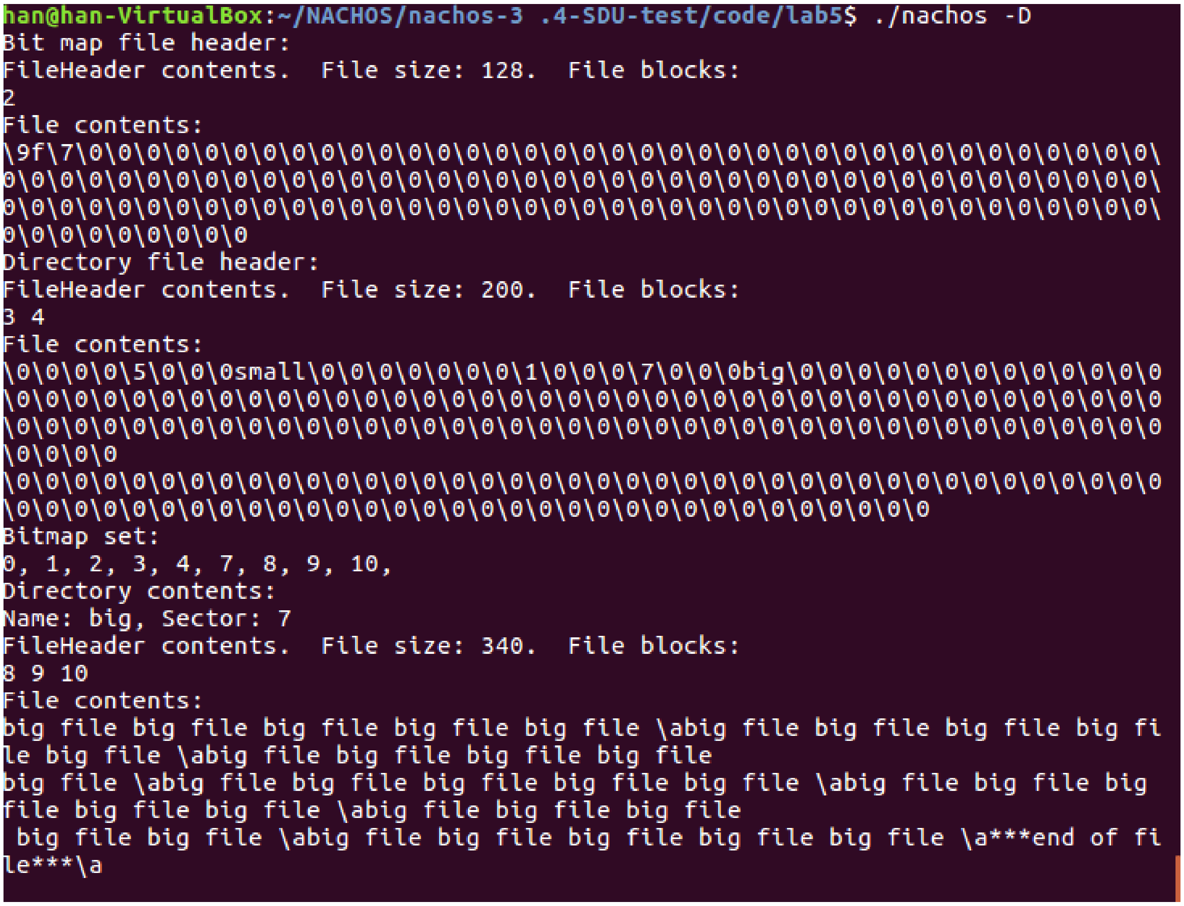


图 5-9 删除文件 small（nachos –D）

Nachos 调用 FileSystem::Remove(char *name) 删除一个文件，参见 FileSytem::Remove()，Directory::Remove()等对删除文件的实现。

因此，要恢复一个删除的文件，只要该文件的上述信息未被新建的文件覆盖，就可以根据文件名在目录项中找到该文件所对应的项，将对应的 inUser 位置 1，并在位示图中恢复文件头所占用的扇区号，再根据文件头的信息在位示图中恢复文件数据所占用的扇区号即可；

上述删除文件的策略为恢复一个删除的文件带来了极大的便利；（其实 FAT 文件系统就采用该策略）；

5.10 打开 Nachos 文件的过程

基于上述对 Nachos 磁盘布局以及对文件结构的分析，打开一个 Nachos 文件的过程为：

1、将存放到固定位置的目录表的文件头读到内存中；（这里是 1 号扇区）

2、根据目录表的文件头找到目录表的存放位置，将目录表的数据读到内存中；（3、4 号扇区）

3、根据要打开文件文件名查找目录项，如果文件不存在，返回错误信息；如果文件存在，从

该文件目录表项中获取要打开文件的文件头；
4、将该文件头读到内存中，称为活动的 inode；
5、返回打开文件的文件描述符；

第 6 章 扩展 Nachos 的文件系统（实验 5）

6.1 目的

理解文件系统中文件操作的实现方法，如文件打开、读、写、扩展、定位、关闭等；

理解如何管理硬盘空闲块；

创建文件时，如何为文件分配目录项及文件头（FCB）；

理解文件扩展时，如何为要扩展的数据查找并分配空闲块；

理解文件扩展后，文件大小是如何记录与保存的；

文件被删除后，如何回收为其分配的资源，如文件头、目录项、硬盘块等；

拓展：有精力的同学可进一步尝试多级目录（目录树）的设计与实现方法。

6.2 任务

目前 Nachos 实现的文件系统存在诸多限制，其中之一是文件大小不能扩展，即无法在已经存在的文件尾部追加数据；

该实验的任务就是让你修改 Nachos 的文件系统，以满足：

（1）文件创建时，其大小可初始化为 0；

（2）当一个文件写入更多的数据时，其大小可随之增大；

（3）要求能够在从一个文件的任何位置开始写入数据，即能够正确处理命令行参数 -ap, -hap,及-nap；

例如，如果一个文件的大小为 100 字节，当从其偏移量 50（第一个字节的偏移量是 0）开始写入 100 个字节后，该文件的大小应该为 150 字节，如图 6-1 所示。

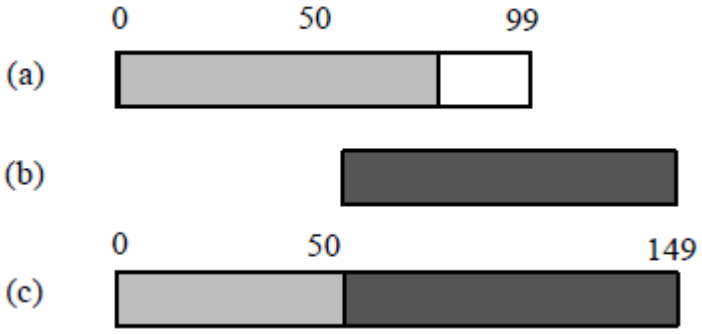


图 6-1 文件的扩展

Nachos 的文件系统包括如下模块：

- class Disk //see ../machine
- class SynchDisk //see ../filesys

- class BitMap //see ../userprog
- class FileHeader //see ../filesys
- class OpenFile //see ../filesys
- class Directory //see ../filesys
- class FileSystem //see ../filesys

其文件系统结构如图 6-2 所示。

该实现在../lab5 下完成，需要将；

如果直接在../filesys 中完成，由于../lab5 下的两个文件 fstest.cc 与 main.cc 是专门为实验 5 设计的，需要将其复制到../filesys 中。

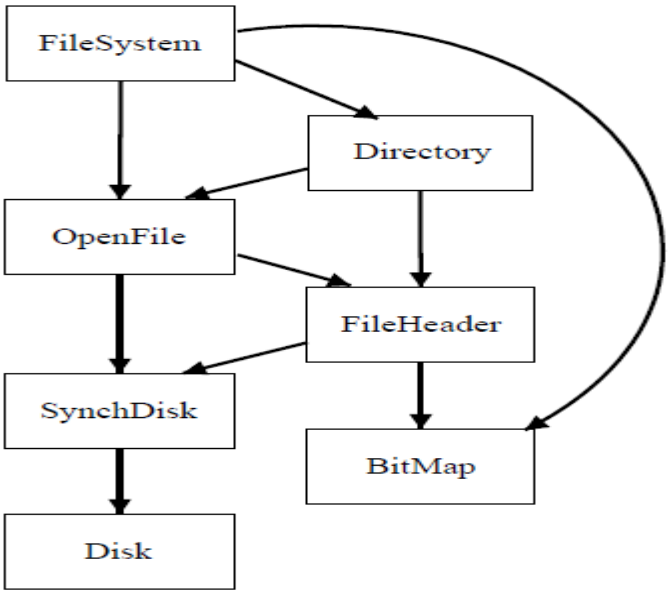


图 6-2 Nachos 文件系统结构

拓展（选做）：目前 Nachos 文件系统仅仅实现了单级目录结构，只有一个根目录。可以尝试采用目录树对文件进行管理。

6.3 Things to Do

6.3.1 问题分析

在设计与实现该实验所要求的任务之前，首先你应该先读懂../machine 及../filesys 目录下的相关源代码（class Disk、class SynchDisk、class BitMap、class FileHeader、class OpenFile、class Directory 及 class FileSystem 等）以及../lab5 下的两个模块（mian.cc 及 fstest.cc），然后根据任务要求，回答下列问题：

（1）需要修改哪些模块，需要使用哪些不需要修改的模块；

（2）在那些需要修改的模块中，哪些函数需要修改，如何修改；

（3）在那些需要修改的模块中，是否需要添加函数与变量；

（4）是否需要在修改的模块中移动变量，或者从一个模块移动到另一个模块；

6.3.2 设计与实现

该实验的工作目录../lab5 中有一个子目录 test，给目录包含文件可用来几个测试你所修改后的文件系统。

../lab5 目录中已经存在 main.cc 及 fstest.cc 两个新文件，这两个文件中包括几个新的 Nachos 文件系统命令，用来测试你修改后的 Nachos 文件系统。main.cc 不需要修改，fstest.cc 需要按如下提示做少量修改：

在 fstest.cc 中有两个函数 Append(…)与 NAppend(…)，其中函数 Append(…)实现了在一个 Nachos 文件的尾部附加一个 UNIX 文件，或从一个 Nachos 文件的中间位置开始写入一个 UNIX 文件；NAppend(…)实现了将一个 Nachos 文件附加到另一个 Nachos 文件的尾部。

Append(…)函数中有如下三条语句：

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

NAppend(…)函数中有如下三条语句：

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

当你在 OpenFile 类中自己编写代码实现了函数 WriteBack()的功能后，将上述语句的后两条语句的注释去掉以让其执行。

注：OpenFile:: WriteBack()的功能是将该文件的文件头（FCB、i-node）写入硬盘相应的扇区中；

阅读../lab5/fstest.cc 中的 Append()及 NAppend()函数，以及涉及的模块（特别是../filesys/openfile.cc 中 OpenFile::Writeat()函数的实现），分析目前 Nachos 实现的文件系统为什么不能对文件的大小进行扩展；

main.cc 模块中新添加了三 Nachos 文件系统命令：

（a）nachos [-d f] –ap unix_filename nachos_filename

该命令的功能是将一个 UNIX 文件（unix_filename）附加到一个 Nachos 文件（nachos_filename）的后面，目的是用来测试当我们在一个 Nachos 的文件尾部写入数据时，文件大写是否会增加；

（b）nachos [-d f] –hap unix_filename nachos_filename

该命令的功能是从一个 Nachos 文件（nachos_filename）的中间位置开始，将一个 UNIX 文件（unix_filename）写入到该 Nachos 文件中。如果这个 UNIX 文件大于 Nachos 文件的一半，则该目录执行后，新的 Nachos 文件的大小将增加。

上述两个命令调用了 fstest.cc 中的 Append(…)函数。

（c）nachos [-d f] –nap nachos_filename1 nachos_filename1

该命令的功能是将一个 nachos 文件（unix_filename1）附加到一个 Nachos 文件（nachos_filename2）的后面，目的是用来测试当我们在一个 Nachos 的文件尾部写入数据时，文件大写是否会增加；

该命令调用了 fstest.cc 中的 NAppend(…)函数。

注意：只有你在 OpenFile 类中自己编写代码实现了函数 WriteBack()的功能，并在这两个函数 Append(…)与 NAppend(…)中去掉语句//openFile->WriteBack();或//openFileTo->WriteBack();注释，上述三个命令才能正常执行。

注 1：你需要根据实验 2 的内容，

（1）将需要修改的模块从其它目录（如../filesys）复制到../lab5 目录中；

（2）将../filesys 目录中的 arch 目录、Makefile、Makefile.local 复制到../lab5 目录中，并根据需要对 Makefile.local 做相应的修改。

注 2：采取分而治之的策略

实现时，可以采取分而治之的策略，如先根据你的设计方案在需要修改的模块（.cc 及.h）中定义相应的接口函数，暂时不需要编写函数体，编译通过后，再逐个对接口进行实现；每完成一个目标，重新编译进行测试，防止一次修改太多导致无法差错与纠错。

6.4 对新文件系统进行测试

在../lab5 目录下参照 6.5 完成文件扩展的实现与测试；

详细的测试过程见 6.5。

我们的测试需要一个干净的 Nachos 硬盘。删除原来的硬盘 DISK，再利用命令 nachos –f 新建一个硬盘 DISK。

然后，按顺序运行下列 Nachos 文件系统命令：

```
nachos –cp test/small small    //复制 Linux 文件 small 到 Nachos 文件系统中
nachos –ap test/small small    //将 Linux 文件 small 附加到 Nachos 文件 small 的尾部
nachos –cp test/empty empty
nachos –ap test/empty empty
```

运行 nachos –D，屏幕输出系统转储的信息，内容大致如下：

```
Bit map file header:
FileHeader contents.    File size: 128. File blocks:
2
File contents:
\ff\7\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Directory file header:
```

```
FileHeader contents.      File size: 200. File blocks:
3 4
File contents:
\1\12\11@\5\0\0\0small\0\0\0\0G\5\8\1H\5\8\8\0\0\0empty\0\0\0\0\0\0\0\0G
\5\8\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\2\0
\0\0\a0F\5\8\0\0\0\0\0\0\0\18\0\0\0
\0\0\0\0\1\0\0\0\a8D\5\8\0\11\0\0\c8\15\11@H\0\0\0\f8\12\11@\f8\12\11@\0F
\5\8\0\0\0\0\0\0\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...
```

从上述系统转储信息中可以看出，

（1）bitmap 文件在 Nachos 硬盘的第 1 个扇区中，大小是 128 字节（1 个扇区），从中也可以看到该文件的内容；文件 bitmap 的 i-node 默认存储在硬盘的第 0 号扇区中。

（2）目录文件（文件目录表）占用 200 字节（2 个扇区），目录文件的数据块在第 3、4 扇区中，其 i-node 处于第 1 号扇区中（默认）；

（3）文件 small 大小是 168 字节（占用 2 个扇区），它的 i-node 在第 5 号扇区中，

数据块存储在第 6、7 号扇区中。

还需要运行 nachos –hap 等命令以测试你所设计实现的 Nachos 文件系统在各种情况下都正常运行。

6.5 扩展文件的实现与测试

nachos –ap UnixFile NachosFile 将一个 UNIX 文件 UnixFile 附加到 Nachos 文件 NachosFile（在硬盘 DISK 中）之后；

nachos –hap UnixFile NachosFile 将一个 UNIX 文件 UnixFile 从 Nachos 文件 NachosFile 的中间开始写入，覆盖 NachosFile 后半部分内容，如果 UnixFile 文件的长度大于 NachosFile 的一半，将在 NachosFile 后面追加 UnixFile 尚未写完的数据。

nachos –nap fromNachosFile toNachosFile 将 Nachos 硬盘 DISK 中的 Nachos 文件 fromNachosFile 附加到另一个 Nachos 文件 toNachosFile 尾部；

可以看出，上述三个 Nachos 命令都涉及到对 Nachos 文件的扩展；

main.cc 中关于参数的处理部分可以看出，nachos –cp 命令调用的是../lab5/fstest.cc 中的 Copy()函数，nachos –ap 与 nachos –hap 命令调用的是../lab5/fstest.cc 中的 Append()函数，nachos –nap 命令调用的是../lab5/fstest.cc 中的 NAppend()函数。

注：../lab5/ fstest.cc 与../filesys/ fstest.cc 两个文件的内容不用。

6.5.1 nachos -ap 与 nachos -hap 命令的实现

../lab5/fstest.cc 中的 Append()函数调用了 OpenFile::Write()，OpenFile::Write()又调用了 OpenFile::WriteAt()，OpenFile::WriteAt()试图从文件尾追加另一个文件内容。

分析 OpenFile::WriteAt()的实现，首先 OpenFile::WriteAt() 从要写入文件的文件头中获取长度（fileLength = hdr->FileLength()），如果开始写入的位置是文件尾，则函数退出，使文件无法扩展（if ((numBytes <= 0) || (position >= fileLength)) return 0; ）；同时，即使要开始写入的位置不在文件尾，但如果从该位置开始写入的数据过多，超出了原文件的长度，超出部分也不再写入（ if ((position + numBytes) > fileLength) numBytes = fileLength - position; // numBytes 是要写入文件的字节数），也使文件扩展操作无法进行。

如果取消将上述两个约束，数据是可以从文件尾部追加，但由于文件数据扩展后，文件头三元组<文件长度，占用扇区数，扇区列表>中的数据并没有更新，因此这种扩展是无意义的；更为严重的是，如果扩展的数据很多，文件最后的一个扇区原来剩余的空间无法容纳，数据无法确定剩余将写在何处，要么造成数据丢失，要么占用或覆盖其它的扇区空间（文件系统管理用的扇区或者其它文件的扇区），造成系统崩溃。

因此，Nachos 文件系统扩展功能主要是修改 OpenFile::WriteAt()函数：

（1）修正上面提到的两个约束；

（2）如果要扩展数据不多，原来文件的最后一个扇区的剩余空间足以容纳，需要修改文件头中的文件长度，然后将文件头写回硬盘原来文件头所占用的扇区中；

（3）如果原来文件的最后一个扇区的剩余空间无法容纳要扩展的数据，需要为

这些数据分配新的扇区，则需要修该空闲块管理使用的位示图文件以及文件头三元组中的三个数据，并将它们适时写回到硬盘原来的扇区中；

因此文件扩展操作需要涉及的内容有：OpenFile 类、FileHeader 类、BitMap 类、FileSystem 类以及 fstest.cc 中的 Append()函数及 NAppend()函数等；

需要修改的文件及函数：

（1）修改 OpenFile::WriteAt()，允许从文件尾部开始写数据，并可为要写入的数据分配新的扇区；

（2）修改 FileSystem 类，添加空闲块位示图文件的硬盘读写操作；

（3）修改 OpenFile::OpenFile()及 OpenFile::WriteBack()，实现文件头的硬盘读写；

（4）修改 FileHeader::Allocate()，为添加的数据分配硬盘块（扇区）；

（5）修改 fstest.cc 的 Append()函数，使下次的写指针指向新写入数据的尾部，并在扩展操作结束后调用 OpenFile::WriteBack()将修改后的文件头写入硬盘；

nachos –ap 命令具体实现方案（仅供参考）：

1、修改 OpenFile::WriteAt()

目的：允许从文件尾开始写数据；

（1）分析 main.cc，命令 nachos –cp 调用的是函数 Append()（参见../lab5/fstest.cc）；

（2）Append()函数调用了 OpenFile::Write()，OpenFile::Write()又调用了 OpenFile::WriteAt()，因此考虑如何修改 OpenFile::WriteAt()成员函数以实现；

（3）对 OpenFile::WriteAt()函数的修改

只需修改两个不能对文件进行扩展的两个约束即可，后续代码保持不变；

OpenFile::WriteAt()的两个约束：

```
if ((numBytes <= 0) || (position >= fileLength))      //约束 1
    return 0;  // check request
if ((position + numBytes) > fileLength)               //约束 2
    numBytes = fileLength - position;
```

将第一个约束修改为：

```
int fileLength = hdr->FileLength();   //第一次调用返回值是从硬盘读出的文件头中的值，
                                      //后续的每次调用都是获取的我们重载的
                                       //FileHeader::Allocate()中修改的值（numBytes）
if ((numBytes <= 0) || (position > fileLength))   //约束 1
    return -1;     //参数错误
```

对于第二个约束，如果条件(position > fileLength)成立，说明文件需要扩展；

（a）如果原来文件最后一个扇区的剩余空间足以容纳要写入的 numBytes 个字节，就不需要为写入操作分配新的扇区，在原文件的最后一个扇区中写入数据即可；但要修改文件头中文件大小属性；文件写操作结束后将文件头写

回硬盘原来的扇区中；

（b）如果原来文件最后一个扇区的剩余空间太小，无法容纳要写入的numBytes 个字节，就需要为写入操作分配新的扇区，在原文件的最后一个扇区写满后，将剩余数据写入新分配的扇区中；

（c）这里要修改文件头中文件大小属性，同时要将新分配的扇区在空闲块管理位示图中对应的位置 1（已分配），然后将位示图写回硬盘原来的扇区中；文件写操作结束后将文件头写回硬盘原来的扇区中；

因此，将第二个约束修改形如：

```
if ((position + numBytes) > fileLength) {    //约束 2
    int incrementBytes = (position + numBytes) – fileLength;
    BitMap *freeBitMap = fileSystem-> getBitMap();       //自己实现
    bool hdrRet;
    hdrRet = hdr->Allocate(freeBitMap, fileLength, incrementBytes);   //自己实现
    if ( !hdrRet )     // Insuficient Disk Space, or File is Too Big
        return -1;
    fileSystem-> setBitMap(freeBitMap);   //自己实现
}
// OpenFile::WriteAt()中的后续代码不需修改，保持不变
```

2、修改 FileSystem 类，增加 setBitMap()与 getBitMap()

目的：从硬盘读取空闲块位示图文件，内容被修改后再将其写回硬盘；

目前在 FileHeader::Allocate()中调用，可以将 setBitMap()与 getBitMap()与 FileHeader 的读写放在一起，以减少对硬盘的访问次数；

当文件需要扩展时，获取空闲块位示图文件（BitMap *freeBitMap = fileSystem-> getBitMap()），然后判断是否需要为写入数据新分配扇区，如果需要为扩展数据新分配扇区，就修改新分配扇区在位示图对应位的状态（hdr->Allocate(freeBitMap, fileLength, incrementBytes)），然后写回硬盘原来的扇区（fileSystem-> setBitMap(freeBitMap)）；

其中，getBitMap()与 setBitMap()完成从硬盘读取位示图文件与将位示图文件写回硬盘操作；

getBitMap() 调用了 ../userprog/bitmap.cc 中 BitMap 类的 FetchFrom(OpenFile *)，setBitMap()调用了 BitMap 类的 WriteBack(OpenFile *)完成。

fileSystem 中的 FileSystem 类没有定义实现 getBitMap()与 setBitMap()，需要你自己实现；

直观上，类 FileSystem 在其构造函数中，维护了两个一直处于打开状态的文件句柄 OpenFile* freeMapFile 与 OpenFile* directoryFile;directoryFile，freeMapFile = new OpenFile(FreeMapSector)，File* directoryFile;directoryFile = new OpenFile(DirectorySector)，一个是硬盘 DISK 上的位示图文件，一个 DISK

上的目录文件（参见./filesys/filesys.h 与./filesys/filesys.cc），我们可以直接使用它们实现对 DISK 上位示图文件与目录文件的读与写操作（BitMap 类中位示图的读写函数 FetchFrom(OpenFile *)与 WriteBack(OpenFile *)就是使用 freeMapFile 实现的），由于这两个 OpenFile 对象是 FileSystem 类的私有变量，因此需要在 FileSystem 类中定义实现 getBitMap()与 setBitMap()；

这两个函数的代码形如：

```
BitMap* FileSystem::getBitMap() {
   //numSector: DISK 上总扇区数（共有 32*32=1024 个扇区）
    BitMap *freeBitMap = new BitMap(numSector);
    freeBitMap->FetchFrom(freeMapFile);
    return freeBitMap;
}
```

```
void FileSystem::setBitMap(BitMap* freeMap) {
    freeMap->WriteBack(freeMapFile);
}
```

3、修改 OpenFile::OpenFile()及 OpenFile::WriteBack()

目的：将修改后的文件头写回硬盘；

OpenFile 类维护了一个 FileHeader 类对象 hdr（参见 Openfile 的构造函数，其代码如下）：

```
OpenFile::OpenFile(int sector)
{
      hdr = new FileHeader;
      hdr->FetchFrom(sector);
      seekPosition = 0;
}
```

可以看出，构造函数从硬盘的扇区 sector 中读取该文件的文件头（FCB 或 i-node），并将读写指针（偏移量）设置为开始位置（0）；

注：FileHeardер:: FetchFrom(int sectorNumber)函数从硬盘的扇区 sectorNumber 中读取一个文件的头文件信息，FileHeardер:: WriteBack(int sectorNumber)函数将一个文件的头文件写到硬盘的扇区 sectorNumber 中；

因此，需要在 OpenFile 类中定义一个私有变量，如 int hdrSector，在构造函数中记录该文件头所在的扇区号，以便 FetchFrom()函数 WriteBack()函数使用；代码形如：

```
OpenFile::OpenFile(int sector)
{
      hdr = new FileHeader;
      hdr->FetchFrom(sector);
      seekPosition = 0;
```

```
        hdrSector=sector;     /打开文件的文件头所在的扇区号
    }
```

构造函数 OpenFile(int sector)通过 FileHeader::FetchFrom(int sector)从硬盘读取并维护一个打开文件的文件头 hdr，私有变量 hdrSector 记录了该文件头所在的扇区号，因此，需要通过 FileHeader::WriteBack(int sector)实现函数 OpenFile::WriteBack()，在文件头被修改后将其回写到硬盘的扇区 hdrSector 中。代码形如：

```
    void OpenFile::WriteBack() {
        hdr-> WriteBack(hdrSector);
    }
```

4、修改 FileHeader::Allocate()

目的：为要写入的文件数据分配硬盘空间；（FileHeader::Allocate()）

写入数据可能利用文件的最后一个扇区的剩余空间，也可能为其新分配扇区（硬盘块）；

（1）FileHeader 构造函数

FileHeader 没有显式定义构造函数，编译器使用的是默认的构造函数，导致为一个文件所分配的文件头在有效数据之后可能含有一些无用的数据（但不影响使用），但这些无效数据可能影响对 hexdump –C DISK 输出的数据进行分析，可以在 FileHeader 的构造函数中对文件头内容进行清除，代码形如：

```
    FileHeader:: FileHeader()
    {
        numBytes=0;          //文件大小
        numSectors=0;        //文件扇区数
        for (int i=0;i<NumDirector;i++)   // NumDirector=30：文件最多拥有的扇区数
            dataSectors[i]=0;   /文件扇区索引表/
    }
```

（2）FileHeader::Allocate()

FileHeader::Allocate(BitMap *freeMap, int fileSize)函数根据文件大小为文件分配所需的所有扇区块，并在位示图中标记所分配的扇区块，设置头文件三元组< numBytes, numSectors, dataSectors[30] >，我们可以重载 FileHeader::Allocate(BitMap *freeMap, int fileSize, int incrementBytes)，以根据要扩展的数据大小 incrementBytes 判断是否需要分配新的扇区块，并更新文件头三元组；

原 FileHeader::Allocate()函数的代码如下：

```
Bool FileHeader::Allocate(BitMap *freeMap, int fileSize)
{
      numBytes = fileSize;
      numSectors   =   divRoundUp(fileSize, SectorSize);
      if (freeMap->NumClear() < numSectors)
       return FALSE;       // not enough space

      for (int i = 0; i < numSectors; i++)
          dataSectors[i] = freeMap->Find();
      return TRUE;
}
```

语句 hdr->Allocate(freeBitMap, fileLength, **incrementBytes**)判断是否需要为写入数据分配新的扇区，如果需要就为其分配，并更新位示图及文件头三元组；重载后的 FileHeader::Allocate()代码形如：

```
Bool FileHeader::Allocate(BitMap *freeMap, int fileSize, int incrementBytes)   {
    if (numSectors > 30)       //限定每个文件最多可分配 30 个扇区
        return false;              //超出限定的文件大小
    if ( (fileSize==0) &&( incrementBytes>0) ) {   //在一个空文件后追加数据
         if (freeMap->NumClear() <1)        //至少需要一个扇区块
             return false;          //磁盘已满，无空闲扇区可分配
        //为添加数据先分配一个空闲磁盘块，并更新文件头信息
        dataSectors[0] = freeMap-Find();
        numSectors = 1;
        numBytes = 0;
     }
    numBytes=fileSize;
    int offsetr= numBytes % SectorSize;   //原文件最后一个扇区块数据偏移量
    int newSectorBytes = increment    - (SectorSize - (offset + 1));
    //最后一个扇区块剩余空间足以容纳追加数据, 不需分配新的扇区块
    if (newSectorBytes < =    0 ) {
         numBytes = numBytes + incrementBytes;     //更新文件头中的文件大小
         return TRUE;
    }
    //最后一个扇区的剩余空间不足以容纳要写入的数据，分配新的磁盘块
    int moreSectors = divRoundUp(newSectorBytes, SectorSize);   //新加扇区块数
    if (numSectors + moreSectors > 30)
        return FALSE;   //文件过大，超出 30 个磁盘块
    if (freeMap->NumClear() < moreSectors)   //磁盘无足够的空闲块
         return false;
    //没有超出文件大小的限制，并且磁盘有足够的空闲块
    for ( int i = numSectors; i < numSectors + moreSectors; i++ )
```

```
            dataSectors[i] =    freeMap->Find();
        numBytes = numBytes + incrementBytes;     //更新文件大小
        numSectors = numSectors + moreSector;      //更新文件扇区块数
        return TRUE;
}
```

注：应该在文件扩展结束后，将文件头写入硬盘

4、修改 fstest.cc 的 Append()

（1）修改写指针

对 fstest.cc 中的 Append()函数还要做少量修改， while{ … }循环中的去掉语句 start += amountRead 的注释，使每次写操作都是从上次写入的数据之后的位置开始进行（第一次是从 start 开始，默认是从文件尾或文件中间开始写入）；

（2）将文件头写回硬盘

fstest.cc 中的 Append()函数调用了我们修改后的 OpenFile::WriteAt()函数，OpenFile::WriteAt()函数调用了函数 FileHeader::Allocate()（重载），FileHeader::Allocate()根据每次写入的数据修改文件头三元组，但一直在内存中，尚未写回硬盘，因此在 fstest.cc 中的 Append()函数中写操作结束后，应该调用 OpenFile::WriteBack()将修改后的文件头写回到硬盘的相应的扇区中。

代码大致如下：

```
void Append(char *from, char *to, int half)      //nachos -ap UNIX_from nachos_to
                                                 //nachos -hap UNIX_from nachos_to
{
      FILE *fp;
      OpenFile* openFile;
      int amountRead, fileLength;
      char *buffer;

      int start; //start position for appending
     //------------------------------------------------------------------------
     // Open UNIX file
      if ((fp = fopen(from, "r")) == NULL) {
         printf("Couldn't open source file \"%s\".\n", from);
         return;
      }

     // Figure out length of UNIX file
      fseek(fp, 0, 2);
      fileLength = ftell(fp);
      fseek(fp, 0, 0);

      if (fileLength == 0)
      {
         printf("Nothing to append from file \"%s\".\n", from);
         return;
      }
     // Nachos file
```

```
        if ( (openFile = fileSystem->Open(to)) == NULL)
        {
            if (!fileSystem->Create(to, 0))
            {
                printf("Couldn't create destination file \"%s\" to append.\n", to);
                fclose(fp);
                return;
            }
            openFile = fileSystem->Open(to);
        }
        ASSERT(openFile != NULL);
        start = openFile->Length();
        fileLength=openFile->Length();
        if (half)       //从文件中间位置开始添加
            start = start / 2;

        openFile->Seek(start);
        // Append the data in TransferSize chunks
        buffer = new char[TransferSize];          //TransferSize=10
        while ((amountRead = fread(buffer, sizeof(char), TransferSize, fp)) > 0)
        {
            int result;
            result = openFile->Write(buffer, amountRead);
            if (result < 0)     //文件过大，或空闲磁盘块不足
            {
                printf("\nERROR!!!!!!\n");
                printf("Insuficient Disk Space, or File is Too Big!\n");
                printf("Writting Terminated.\n\n");
                break;
            }
            ASSERT(result == amountRead);
            start += amountRead;
        } //emd of while
        delete [] buffer;
        openFile->WriteBack();   //将文件头写回硬盘
        DEBUG('f',"inodes have been written back\n");
        // Close the UNIX and the Nachos files
        delete openFile;
        fclose(fp);
    }
```

至此，对 fstest.cc 中的 Append()函数的修改全部完成，可以利用 nachos –ap 及 nachos –hap 测试你的修改。

6.5.2 nachos -nap 的实现

分析 fstest.cc 中的 NAppend()函数可以看出，NAppend()函数也是调用 OpenFile::Write()实现文件的写操作，我们也知道，OpenFile::Write()通过调用

OpenFile::WriteAt()实现，我们已经修改了 OpenFile::WriteAt()，实现了从文件的任何位置开始写入数据；

因此，你只需在 fstest.cc 中的 NAppend()函数中与 Append()做同样的修改即可：

去掉语句 start += amountRead 及 openFileTo->WriteBack();的注释；

代码大致如下：

```
    void NAppend(char *from, char *to)      //nachos -nap nachos_from nachos_to
    {
        OpenFile* openFileFrom;
        OpenFile* openFileTo;
        int amountRead, fileLength;
        char *buffer;

        //    start position for appending
        int start;

        if (!strncmp(from, to, FileNameMaxLen))
        {
            printf("Source and destination should be different files.\n");
            return;
        }
        if ( (openFileFrom = fileSystem->Open(from)) == NULL)
        {
            printf("Source file \"%s\" does not exist.\n", from);
            return;
        }
        fileLength = openFileFrom->Length();
        if (fileLength == 0)
        {
            printf("NAppend: nothing to append from file \"%s\"\n", from);
            return;
        }
        if ( (openFileTo = fileSystem->Open(to)) == NULL)
        {
            // file "to" does not exits, then create one
            if (!fileSystem->Create(to, 0))
            {
                printf("Couldn't create destination file \"%s\" to append.\n", to);
                printf("File already exists, or file too big, or files on disk over 12, or insufficient
disk space.\n");
                delete openFileFrom;
                return;
            }
            openFileTo = fileSystem->Open(to);
        }
        ASSERT(openFileTo != NULL);
        // append from position "start"
        start = openFileTo->Length();
```

```
        openFileTo->Seek(start);
        fileLength=openFileTo->Length();
        // Append the data in TransferSize chunks
        buffer = new char[TransferSize];
        openFileFrom->Seek(0);
        while ( (amountRead = openFileFrom->Read(buffer, TransferSize)) > 0)
        {
            int result;
            result = openFileTo->Write(buffer, amountRead);
            if (result < 0)
            {
                printf("\nERROR!!!!!!\n");
                printf("Insuficient Disk Space, or File is Too Big!\n");
                printf("Writting Terminated.\n\n");
                break;
            }
            ASSERT(result == amountRead);
            start += amountRead;
        }   //end of while
        delete [] buffer;
        openFileTo->WriteBack();     //将文件头写回磁盘
        DEBUG('f',"inodes have been written back\n");
        // Close both Nachos files
        delete openFileTo;
        delete openFileFrom;
        }   //end of NAppend(char *from, char *to)
```

6.5.3 nachos 文件系统测试

一、测试内容：

1. Nachos 文件系统的几个命令

nachos –f，nachos –D，nachos –r，nachos –cp，nachos –ap，nachos –hap，nachos –nap，nachos –l，nachos –p 等（参见 main.cc 中关于文件的操作命令）；

2．Nachos 文件系统的限制

最多可创建多少个文件，每个文件最大是多少字节等；

二、测试过程：

（1）为容易识别硬盘 DISK 信息的改变，将../file/test/下的 small 文件的内容修改为”This is a small file.”，

类似的，将../file/test/下的 medium 文件的内容修改为”This is a medium file.”，并设置多行；

将../file/test/下的 big 文件的内容修改为”This is a big file.”，并设置更多的多行；

为便于识别在 DISK 上的文件内容，你也可以在该目录下建立其它的文本文件，内容自定；

（2）nachos –f ，在硬盘 DISK 上初始化一个 Nachos 文件系统；

（3）nachos –D 及 hexdump –C DISK 考察 Nachos 在硬盘 DISK 上初始化的文件系统情况；

包括空闲块位示图的头文件（0 号扇区）、空闲块位示图文件数据块（2 号扇区）、目录表头文件（1 号扇区）、目录表数据块（目录表）（3、4 号扇区）

（4）nachos –cp test/small smal，复制 test 目录下的 UNIX 文件 small 到 DISK 中；

（5）nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息；

包括空闲块位示图的头文件（0 号扇区）、空闲块位示图文件数据块（2 号扇区）、目录表头文件（1 号扇区）、目录表数据块（目录表）（3、4 号扇区）、文件 small 的头文件、文件 small 的数据块等信息；

看看 0~6 号扇区有哪些改变；

（6）nachos –cp test/small small，将 test 目录下的 UNIX 文件 samll 附加到 Nachos 文件 small 中；测试给一个已存在的文件追加数据；

（7）nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息；

查看系统是否将 small 文件的内容扩展到 small 文件原扇区的剩余空间中；

（8）nachos –ap test/big small，将 UNIX 文件 big 附加到一个 small 中；测试为文件分配新扇区的功能；

（9）nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息；

考察系统是否将 samll 文件原扇区的剩余空间写满后，为 small 分配新的扇区块，写入 big 的内容；

（10）nachos –ap test/medium medium，测试给一个空文件追加数据的功能；在 Append()中如果 DISK 中不存在文件 medium，将会自动创建一个空的 medium 文件，然后将 test/medium 文件内容追加到 Nachos 空文件 medium 中；

（11）nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息；

（12）nachos –ap test/big small，将 UNIX 文件 big 附加到一个 small 中；测试 Nachos 为 small 新分配的扇区块的位置，即 Nachos 是否为 small 分配不连续的扇区块，理解操作系统中索引分配的机理；

（13）nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息；

在执行（12）之前，由于 small 的数据块之后的扇区是 medium 文件的文件头及其数据块，因此，（12）执行后，系统会在 medium 文件之后为 small 分配新的扇区；

体现出 small 文件的数据块在硬盘上不是连续的，体现出文件数据块索引分配的特点；

（14）nachos –hap test/medium small，测试从 small 的中间写入文件的功能；

（15）nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息；

查看系统是否在 small 的中间写入文件 test/medium 的内容；

（16）nachos –nap medium small，测试将一个 nachos 文件附加到另一个 nachos 文件的功能；

（17）nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息；

查看系统是否将 nachos 文件 medium 附加到 nachos 文件 small 的尾部；

（18）nachos –r small，测试文件删除功能，以及删除后硬盘 DISK 的信息变化；

（19）nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息；

考察位示图数据块、目录表数据块、small 的文件头以及 small 文件内容有哪些变化；

（20）测试 nachos –l（列目录），nachos –p small（显示 small 的内容）等命令；

（21）反复运行 nachos –ap test/big small，测试 nachos 文件系统中对一个文件长度的限制（一个文件最多可分配 30 个扇区，每个扇区 128 字节，因此文件最大限制为 3KB）；

（22）反复运行 nachos –ap 或 nachos –cp 在硬盘 DISK 上新建文件，测试 nachos 文件系统中最多可创建多少个文件（nachos 采用一级目录，最多有 10 个目录项，因此对多可存储 10 个文件）；

（23）测试其它你想测试的内容，并利用 nachos –D 及 hexdump –C DISK 查看结果是否是你的预期。

第 7 章 Nachos 用户程序与系统调用（实验 6）

7.1 目的

为后续实验中实现系统调用 Exec()与 Exit()奠定基础

理解 Nachos 可执行文件的格式与结构；

掌握 Nachos 应用程序的编程语法，了解用户进程是如何通过系统调用与操作系统内核进行交互的；

掌握如何利用交叉编译生成 Nachos 的可执行程序；

理解系统如何为应用程序创建进程，并启动进程；

理解如何将用户线程映射到核心线程，核心线程执行用户程序的原理与方法；

理解当前进程的页表是如何与 CPU 使用的页表进行关联的；

7.2 任务

该实验将体验 Nachos 的用户程序、应用进程进程及 Nachos 系统调用的相关概念；

具体工作可参见 7.4 Things to Do；

理论上讲，实验 6、7、8 应该基于 Nachos 所实现的文件系统实现，系统调用 Create、Open、Read、Write、Close 及 Exec 应该对模拟盘 DISK 上的文件进行操作；

Nachos 实现的文件系统实现了两个版本，通过宏 FILESYS_STUB 与 FILESYS 进行条件编译所产生的两个不同的实现（参见../filesys/filesys.h）；宏 FILESYS_STUB 实现的文件操作直接利用 UNIX 所提供的系统调用实现，操作的不是硬盘 DISK 上的文件；宏 FILESYS 实现的文件系统是通过 OpenFile 类对 DISK 上的文件进行操作（尽管最终也是使用 UNIX 的系统调用实现）；

考察../userprog/makefile 与 makefile.local 的内容可以看出，实验 6、7、8 默认使用的是 FILESYS_STUB 定义的相关实现，即不是对 DISK 上的文件进行操作，而是直接对 UNIX 文件进行操作；

如何设计进度比较慢，可以使用其默认设置（FILESYS_STUB），即从../test/目录中加载 Nachos 的应用程序并执行之。如果有时间，可以实现从 Nachos 的硬盘中加载 Nachos 的应用程序并执行之。

（1）阅读../bin/noff.h，分析 Nachos 可执行程序.noff 文件的格式组成；

（2）阅读../test 目录下的几个 Nachos 应用程序，理解 Nachos 应用程序的编程语法，了解用户进程是如何通过系统调用与操作系统内核进行交互的；

（3）阅读../test/Makefile，掌握如何利用交叉编译生成 Nachos 的可执行程序；

（4）阅读../threads/main.cc，../userprog/ progtest.cc，根据对命令行参数-x 的处理过程，理解系统如何为应用程序创建进程，并启动进程的；

（5）阅读../userprog/ progtest.cc，../threads/scheduler.cc（Run()），理解如何将用户

线程映射到核心线程，以及核心线程执行用户程序的原理与方法；

（6）阅读../userprog/ progtest.cc，../machine/translate.cc，理解当前进程的页表是如何与 CPU 使用的页表进行关联的；

7.3 背景知识

Nachos 是一个操作系统，可以运行 Nachos 应用程序。Nachos 的应用（用户）程序采用类 C 语言语法，通过 Nachos 提供的系统调用进行编写。详情可参阅../test 目录下的几个.c 文件。

由于 Nachos 模拟了一个执行 MIPS 指令的 CPU，因此需要将用户编写的 Nachos 应用程序编译成 MIPS 框架的可执行程序。

Nachos 提供了一个交叉编译程序 gcc-2.8.1-mips.tar.gz，可将 Nachos 用户编写的应用程序编译成 MIPS 指令集的可执行程序，然后在 Nachos 中运行。其安装与使用方法请参见实验一。

gcc MIPS 交叉编译器（gcc MIPS cross-compiler）将 Nachos 的应用程序编译成 COFF 格式的可执行文件，然后利用../test/coff2noff 将 COFF 格式的可执行程序转换成 Nachos CPU 可识别的 NOFF 可执行程序。

运行 Nachos 应用程序的方法：

（1）在../test 目录下运行 make，将该目下的几个现有的 Nachos 应用程序（.c 文件）交叉编译，并转换成 Nachos 可执行的.noff 格式文件。

现有的几个应用程序：halt.c，matmult.c，shell.c，sort.c

（2）在../userprog 目录下运行 make 编译生成 Nachos 系统，键入命令./nachos –x halt.noff 可让 Nachos 运行应用程序 halt.noff，参数-x 的作用是 Nachos 运行其应用程序。

7.4 Things to Do

7.4.1 Nachos 应用程序与可执行程序

该实验中，你需要研究一下../test 目录中的 Makefile 文件，理解 Nachos 生成 Nachos 可执行程序的过程。目录../test 提供了 4 个示例程序（.c 文件）。

可按下述步骤跟踪检查 Nachos 应用程序的启动与执行过程（以 halt.c 为例）：

（1）在./test 目录中将文件 halt.c 的修改成：

```
#include “syscall.h”
int main()
{
    int i,j,k;
    k=3;
    i=2;
    j=i-1;
    k=i-j+k;
    Halt();
```

```
    /* not reached*/
}
```

（2）在./test 目录中重新编译生成新的 halt

（3）在./test 目录中可以通过下述命令生成 halt.c 对应的汇编代码文件 halt.s；

/usr/local/mips/bin/decstation-ultrix-gcc -I ../userprog -I ../threads -S halt.c

（4）研究一下 halt.s，看看函数 main 的栈帧（stack frame）是如何创建于撤销的，编译程序将 C 程序中的语句编译成什么样的机器指令。

（5）进入目录../userprog

（a）运行 make 编译 Nachos 的内核

（b）如果../test 中指向../arch/unknown-i386-linux/bin/halt.noff 文件的符号链接文件是 halt，则通过命令 nachos –x ../test/halt 运行 Nachos 的应用程序 halt

如果../test 中指向../arch/unknown-i386-linux/bin/halt.noff 文件的符号链接文件是 halt.noff，则通过命令 nachos –x ../test/halt.noff 运行 Nachos 的应用程序 halt

你可以加上参数 –d m 输出显示 Nachos 模拟的 MIPS CPU 所执行的每条指令，即

nachos –d m –x ../test/halt.noff,

还可以再加上参数-s，以输出每条指令执行后对应寄存器的状态，如

nachos –d m –s –x ../test/halt.noff,

你也可以利用 gdp 跟踪、理解 Nachos 应用程序的启动与执行过程。

7.4.2 Nachos 可执行程序格式

阅读../bin/noff.h，分析 Nachos 可执行程序.noff 文件的格式组成；

7.4.3 页表的系统转储

在后续的设计任务中，需要在 Nachos 中运行多道程序，需要理解用户进程的创建过程。

Nachos 中的进程是通过形成一个地址空间从线程演化而来的。

Nachos 的存储管理采用分页管理方式，在类 AddrSpace 中添加成员函数 Print()，在为一个应用程序新建一个地址空间后调用该函数，输出该程序的页表（页面与帧的映射关系），显示信息有助于后续程序的调试与开发。

```
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");
}
```

如在../userprog/ progtest.cc 的 void StartProcess(char *filename)中，当为一个应用程序新建一个空间后，调用 Print()，输出页表信息。

```
    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);
        currentThread->space = space;
        space->Print();                  //输出该作业的页表信息
        delete executable;                  // close file

        space->InitRegisters();             // set the initial register values
        space->RestoreState();              // load page table register

        machine->Run();             // jump to the user progam
        ASSERT(FALSE);              // machine->Run never returns;
                                    // the address space exits
                                    // by doing the syscall "exit"
    }
```

在../userprog 中运行 nachos –x ../test/halt.noff，从输出结果中可以看看程序 halt.noff 的页面与帧（虚页与实页）的对应关系，以及 Nachos 为该程序分配的实页数（Nachos 为该程序分配了 11 个实页）。

7.4.4 应用程序进程的创建与启动

阅读../userprog/ progtest.cc 的 void StartProcess(char *filename)函数：

（1）理解 Nachos 为应用程序创建进程的过程；

（2）理解系统为用户进程分配内存空间、建立页表的过程，分析目前的处理方

法存在的问题？

（3）理解应用进程如何映射到一个核心线程；

（4）如何启动主进程（父进程）；

（5）理解当前进程的页表是如何与 CPU 使用的页表进行关联的

（6）思考如何在父进程中创建子进程，实现多进程机制；

（7）思考进程退出要完成的工作有哪些？

7.4.5 分配更大的地址空间

我们的用户程序有时需要系统为其分配更大的地址空间，方法之一就是在程序中定义一个静态数组。例如在用户程序 halt.c 定义了一个大小为 40 个元素的静态整型数组，运行时系统为其分配相应的地址空间，大小为 12 个页面。

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

7.5 几点注记

7.5.1 Nachos 应用程序

Nachos 的应用程序是作者自己定义的一种文件类型，文件头部分结构相对简单，编程方便。

分析../test/Makefile 可以看出，首先利用交叉编译器提供的 gcc、as、ld（注：不是通常的 gcc 等）等工具将 nachos 应用程序.c 编译链接成,coff 文件，然后利用../bin/arch/unknown-i386-linux/bin/coff2noff 将 coff 文件转变成 noff 文件，简单的可执行文件头结构便于程序实现（参见 AddrSpace:: AddrSpace()中将程序装入主存，并设置 PC 的值部分代码）。

NOFF 文件头结构如下（参见../bin/arch/unknown-i386-linux/bin/noff.h）：

```
#define NOFFMAGIC   0xbadfad        /* magic number denoting Nachos object code file */

typedef struct segment {
   int virtualAddr;        /* location of segment in virt addr space */   //progtam entry
   int inFileAddr;         /* location of segment in this file */
   int size;               /* size of segment */
} Segment;
```

```
typedef struct noffHeader {
    int noffMagic;          /* should be NOFFMAGIC */
    Segment code;           /* executable code segment */
    Segment initData;       /* initialized data segment */
    Segment uninitData;     /* uninitialized data segment –
                            /* should be zero'ed before use */
} NoffHeader;
```

具体内容可自己分析；可阅读../bin/arch/unknown-i386-linux/bin/coff2noff.cc 详细理解 COFF 文件与 NOFF 文件的格式；

可以看出，文件头给出了每个段的大小，在文件中的开始位置，以及程序的入口地址等信息；

AddrSpace 的构造函数在将 NOFF 文件装入内存之前，先打开该文件，读入文件头，然后根据文件头信息确定程序所占用的空间，将相应的段装入内存，将程序的入口地址，给 PC 赋值。

7.5.2 页表

将应用程序装入内存后，页表实现了虚页与实页（帧）的对应关系，系统根据页表实现存储保护，页面置换算法根据页表信息进行页面置换等操作；

Nachos 使用的页表结构如下：（参见../machine/translate.h）

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

7.5.3 用户进程的创建与启动

Nachos 的参数-x （nachos –x filename）调用../userprog/ progtest.cc 的 StartProcess(char *filename)函数，为用户程序创建 filename 创建相应的进程，并启动该进程的执行。

系统要运行一个应用程序，需要为该程序创建一个用户进程，为程序分配内存空间，将用户程序（代码段与数据段，数据段包括初始化的全局变量与未初始化的全局

变量，以及静态变量）装入所分配的内存空间，创建相应的页表，建立虚页与实页（帧）的映射关系；（参见 AddressSpace:: AddressSpace()）；

然后将用户进程映射到一个核心线程；（参见 StartProcess() in progtest.cc）

为使该核心线程能够执行用户进程的代码，需要核心在调度执行该线程时，根据用户进程的页表读取用户进程指令；因此需要将用户页表首地址传递给核心的地址变换机构；（machine.h 中维护一个 pageTable 指针，指向当前正在 运行的 Nachos 应用进程的页表）

Instruction 类封装了一条 Nachos 机器指令（参见 Machine 类）；

machine::ReadMem(registers[PCReg], 4, &raw))（在 translate.cc 中实现），实现读内存操作。当对内存进行读写时，MMU 需要进行虚实地址变换，并实现存储保护，还需要维护 TLB。machine::ReadMem() 调用了 Machine::Translate()（参见../machine/translate.cc），实现存储保护并对虚实地址进行转换（Machine::Translate()，然后根据转换后的物理地址从内存中读出一条指令；

Machine::OneInstruction(Instruction *instr)（../machine/mipssim.cc 中实现），对通过 machine::ReadMem(registers[PCReg], 4, &raw))读出的指令进行译码并根据指令规定的操作执行该条指令；

machine::Run()（./machine/mipssim.cc 中实现）循环调用 Machine::OneInstruction(Instruction *instr)执行程序指令，直到程序退出或遇到一个异常；

因此，需要将用户进程的页表传递给 Machine 类维护的页表，才能执行用户程序指令；

AddrSpace::RestoreState() 将用户进程的页表传递给 Machine 类，该页表在为用户进程分配地址空间时创建（参见 AddressSpace 构造函数）；

为便于上下文切换时保存与恢复寄存器状态，Nachos 设置了两组寄存器，一组是 CPU 使用的寄存器 int registers[NumTotalRegs]（参见 Machine 类 in Machine.h），用于保存执行完一条机器指令时该指令的执行状态；另一组是运行用户程序时使用的用户寄存器 int userRegisters[NumTotalRegs]，用户保存执行完一条用户程序指令后的寄存器状态（参见 Thread.h）；

由于 CPU 只有一个（系统初始化时创建了一个 CPU），因此 CPU 寄存器也只有一套；而每个核心线程都可能执行用户程序（一个用户进程至少需要映射到一个核心线程），因此每个核心线程都需要维护一套用户寄存器 userRegisters[]，因此 Nachos 为每个核心线程设置了一组用户寄存器，用于保存与恢复相应的用户程序指令的执行状态；

当用户进程进行上下文切换时（实质上是执行用户进程的核心线程发生上下文切换），将老进程的 CPU 的寄存器状态保存到用户寄存器 userRegisters[]中，并将新用户进程的寄存器状态恢复到 CPU 的寄存器中，以便 CPU 能够继续执行上次被中断的用户程序；参见 Scheduler::Run()中核心进程切换时对 CPU 寄存器与用户寄存器的保存与恢复；

考察 progtest.cc 中的函数 StartProcess(char *filename)的代码，分析系统为应用程

序创建进程与启动进程的过程。

```
StartProcess(char *filename)     // filename：应用程序文件名，如../test/halt.noff
{
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;
    if (executable == NULL) {
        printf("Unable to open file %s\n", filename);
        return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;
    delete executable;              // close file

    space->InitRegisters();         // set the initial register values
    space->RestoreState();          // load page table register
    machine->Run();                 // jump to the user progam
    ASSERT(FALSE);                  // machine->Run never returns;
                                    // the address space exits by doing the syscall "exit"
}
```

语句 space = new AddrSpace(executable)为应用程序 filename 分配内存空间并将其装入所分配的内存空间中，然后建立页表，并建立虚页与实页（帧）的映射关系；

这时 space 就是该进程的标识，这里并没有像 UNIX 中为每个进程分配一个进程号（pid，Nachos 中称为 SpaceId），你自己可以为进程分配一个 pid，并建立 pid 与 space 的映射关系，以后通过 pid 标识该进程；

语句 currentThread->space = space 将该进程映射到一个核心进程；对于第一个用户进程，映射到核心的主线程”main”，即 currentThread 指向主线程；

线程维护一个私有变量 AddrSpace *space，Thread 类的构造函数中，设置 space=NULL；当该线程与一个应用进程捆绑后，space 指向系统为该进程所分配的内存空间，以便被调度时执行该进程所对应的应用程序；

语句 space->InitRegisters()初始化 CPU 的寄存器，包括数据寄存器、PC 以及栈指针等；

其中由于 Nachos 将应用程序的可执行文件转变为.noff 格式时，将程序的入口地址设置为 0，因此，应用进程从虚地址 0 开始执行，因此 PC=0；

系统为应用程序栈分配了 1KB 的空间，将栈顶指针初始化为应用程序空间的尾部（向上生长），为防止发生偶然访问到程序地址空间外部，将栈指针从底部向上偏移了 16 字节（这是个实现问题）；

语句 space->RestoreState()看似是恢复用户寄存器的内容，但这里只是将用户进程的页表传递给系统核心（Machine 类），以便 CPU 能从用户进程的地址空间中读取应用程序指令；

CPU 的寄存器状态被设置后，就开始用户进程的执行；Machine::Run()从程序入口开始，完成取指令、译码、执行的过程，直到进程遇到 Exit()语句或者异常才退出；

对于一个真正的操作系统，理论上讲，不应该创建进程后立刻执行，而应该将进程放入就绪队列，等待进程的调度；

Nachos 目前不支持多进程，

在应用程序执行过程中，如果与应用进程所关联的核心线程发生上下文切换，则引起用户进程的上下文切换；

7.5.4 PCB

核心线程不需要单独分配内存；利用 Thread::Fork()创建的线程，只需调用 Thread::StackAllocate()为其分配栈空间，在几个 CPU 寄存器中设置线程入口 ThreadRoot()，线程的执行体，以及线程知悉结束时需要做的工作（线程结束时调用 Thread::Finish()）；

而应用程序运行时，需要为其创建一个用户进程，为程序分配内存空间，将用户程序（代码段与数据段，数据段包括初始化的全局变量与未初始化的全局变量，以及静态变量）装入所分配的内存空间，创建相应的页表，建立虚页与实页（帧）的映射关系；（参见 AddressSpace:: AddressSpace()）；

理论上讲，系统应该首先为应用程序分配一个 PCB，存放应用程序进程的相应信息；

进程号 pid 可以是 PCB 数组的索引号；

根据应用程序的文件头计算所需的实页数（帧数），根据内存的使用情况为其分配足够的空闲帧（不一定是连续的帧），将应用程序的代码与数据读入内存所分配的帧中，创建页表，建立虚页与实页的映射关系；

还要为应用程序分配栈与堆；

还应该在 PCB 中建立打开文件的列表，列表的索引即为文件描述符；

最后将 pid、页表位置、栈位置及堆位置等信息记录在 PCB 中，在 PCB 中建立三个标准设备的映射关系，并记录进程与线程的映射关系，以及进程的上下文等；

目前 Nachos 实现的比较简单，没有显式地定义 PCB（AddrSpace 类体现了 PCB 的概念，但信息不是很完整），而是将进程的信息分散到相应的类对象中；例如利用所分配内存空间的对象指针标识一个进程，该对象中含有进程的页表、栈指针、与核心线程的映射等信息；

进程的上下文保存在核心线程中，当一个线程被调度执行后，依据线程所保存的进程上下文中执行所对应的用户进程；

同时，进程被创建后不应立即执行，应该将程序入口等记录到 PCB 中，一旦相应的核心线程引起调度，就从 PCB 中获取所需的信息执行该进程；

7.5.5 用户线程映射到核心线程

在实现系统调用 Fork()之前，Nachos 不支持用户多线程机制（可以认为只有一个线程，否则无法建立进程与核心线程的映射关系）；要求系统调用 Fork()实现在同一个用户空间中创建多个用户线程。并建立用户线程与核心线程之间的映射，映射关系最简单的就是 One to One 模型；

7.5.6 线程调度算法

目前 Nachos 默认的线程调度算法是 FCFS，当然，你可以利用 Thread::Yield()模拟抢先机制；如果运行时加上-rs 参数，nachos –rs random-seeds，可以实现轮转法（RR）线程调度，参见../thread/system.cc 中 Initilize()函数初始化 Nachos 内核时对参数-rs 的处理。

第 8 章 地址空间的扩展（实验 7）

8.1 目的

通过考察系统加载应用程序过程，如何为其分配内存空间、创建页表并建立虚页与实页帧的映射关系，理解 Nachos 的内存管理方法；

理解如何系统对空闲帧的管理；

理解如何加载另一个应用程序并为其分配地址空间，以支持多进程机制；

理解进程的 pid；

理解进程退出所要完成的工作；

8.2 任务

该实验与下一个实验（实验 8）可在目录../lab7-8 中完成，参照实验 2 介绍的方法将该实验中需要修改的模块、头文件，以及依赖这些头文件的模块复制到该目录中。

如将需要的模块从../userprog 目录复制到该目录中，还要复制 arch 目录及其子目录、Makefile、Makefile.local 等文件，并对 Makefile 及 Makefile.local 做相应的修改。

该实验中，你需要完成：

（1）阅读../prog/protest.cc，深入理解 Nachos 创建应用程序进程的详细过程

（2）阅读理解类 AddrSpace，然后对其进行修改，使 Nachos 能够支持多进程机制，允许 Nachos 同时运行多个用户线程；

（3）在类 AddrSpace 中添加完善 Print()函数（在实验 6 中已经给出）

（4）在类 AddrSpace 中实例化类 Bitmap 的一个全局对象，用于管理空闲帧；

（5）如果将 SpaceId 直接作为进程号 Pid 是否合适？如果感觉不是很合适，应该如何为进程分配相应的 pid？

（6）为实现 Join(pid)，考虑如何在该进程相关联的核心线程中保存进程号；

（7）根据进程创建时系统为其所做的工作，考虑进程退出时应该做哪些工作；

（8）考虑系统调用 Exec()与 Exit()的设计实现方案；

（9）拓展：可以进一步考虑如何添加自己所需要的系统调用，即../userprog/syscall.h 中没有定义的系统调用，如 Time，以获取当前的系统时间。

8.2 背景知识

假设我们希望在一个 Nachos 应用程序中通过系统调用 Exec()装入并执

行另一个 Nachos 应用程序../test/exec.noff，代码如下：

程序../test/bar.noff 的代码如下：

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

其中../test/exec.noff 程序的代码如下：

```
#include “syscall.h”
int
main()
{
    Halt();
}
```

即当程序../test/bar.noff 执行到语句 Exec(“../test/exec.noff”)时，系统需要将../test/exec.noff 装入到内存，为其分配内存空间并执行它，遗憾的是目前 Nachos 无法实现上述功能。

考察下述 AddrSapce 的构造方法 AddrSpace::AddrSpace(OpenFile *executable)中的程序片段：

```
// 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
```

可以看出，目前实现的 Nachos 系统，当加载一个应用程序并为其分配内存时，总是将程序的第 0 号页面分配到内存的第 0 号帧，第 1 号页面分配到内存的第 1 号帧，…，以此类推。

因此，Nachos 在运行其应用程序../test/bar.noff 时，内存从的第 0 号开始的几个帧已经分配给程序../test/bar.noff，如果执行到语句 Exec(“../test/exec.noff”)加载../test/exec.noff 时，Nachos 仍然将从第 0 号开始的几个帧分配给../test/exec.noff，导致系统为../test/bar.noff 所分配的内存空间被../test/exec.noff 覆盖，程序显然无法正确执行。

8.3 Bitmap Class

该实验的任务涉及到../userprog 目录下的 bitmap 类。请阅读并理解../userprog/bitmap.h 与../userprog/bitmap.cc，然后确定如何使用它们。

思考如何利用位示图对内存的空闲帧进行管理；

第 9 章 系统调用 Exec()与 Exit()（实验 8）

9.1 目的

理解用户进程是如何通过系统调用与操作系统内核进行交互的；
理解系统调用是如何实现的；
理解系统调用参数传递与返回数据的回传机制；
理解核心进程如何调度执行应用程序进程；
理解进程退出后如何释放内存等为其分配的资源；
理解进程号 pid 的含义与使用；

9.1 任务

（1）阅读../userprog/exception.cc，理解系统调用 Halt()的实现原理；
（2）基于实现 6、7 中所完成的工作，利用 Nachos 提供的文件管理、内存管理及线程管理等功能，编程实现系统调用 Exec()与 Exit()（至少实现这两个）。

Nachos 目前仅实现了系统调用 Halt()，其实现代码参见../userprog/exception.cc 中的函数 void ExceptionHandler(ExceptionType which)，其余的几个系统调用都没有实现。

Nachos 系统调用对应的宏在../userprog/syscall.h 中声明如下：

```
#define SC_Halt       0
#define SC_Exit       1
#define SC_Exec       2
#define SC_Join       3
#define SC_Create     4
#define SC_Open       5
#define SC_Read       6
#define SC_Write      7
#define SC_Close      8
#define SC_Fork       9
#define SC_Yield      10
```

该实验的任务是在../userprog/ exception.cc 中实现 Exec()与 Exit()的代码，如果时间允许，可以同时实现 Read、Write、Join()，以便能够运行 Nachos 的 shell（代码参见../test/shell.c）。

该实验与上一个实验（实验 7）可在目录../lab7-8 中完成，参照实验 2 介绍的方法将该实验中需要修改的模块、头文件，以及依赖这些头文件的模块复制到

该目录中。

如将需要的模块从../userprog 目录复制到该目录中，还要复制 arch 目录及其子目录、Makefile、Makefile.local 等文件，并对 Makefile 及 Makefile.local 做相应的修改。

注：本实现也可以直接在../userprog 目录下。

9.2 编写自己的 Nachos 应用程序

我们需要编写一些 Nachos 应用程序，以测试 Nachos 相应的功能。

假如你自己需要编写一个 Nachos 应用程序 exec.c 用来测试你所实现的 Nachos 系统调用 Exec()，可采用如下方法与步骤：

（1）按照类 C 语言的语法以及 Nachos 所提供的系统调用（注：无法使用 Linux 的系统调用），在目录../test 中编写程序 exec.c 并保存到../test 目录中。代码形如：

```
#include “syscall.h”
int    main()
{
    SpaceId pid;
    pid=Exec(“../test/halt.noff”);      //利用你所实现的 Exec()执行../test/halt.noff
    Halt();
    /* not reached */
}
```

（2）将 exrc.c 交叉编译并转换成 Nachos 模拟的 CPU 可执行程序 exec.noff

（a）按如下方法将 exec 添加到../test/Makefile 文件的 targets 列表中

```
……
# 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）运行 make，生成 exec.noff

命令 make exec.s 可生成 exec.c 对应的汇编代码 exec.s

9.3 Things to Do

基于 Lab7 的工作，完成系统调用 Exec()及 Exit()。
如果有时间可以实现 Join()、涉及用户线程以及文件操作的几个系统调用。

9.4 设计与实现的有关问题

9.4.1 在哪里编写系统调用的代码

阅读../machine/machine.cc 及 mipssim.cc 中的实现可以看出，每一条用户程序中的指令在虚拟机中被读取后，被包装成一个 OneInstruction 对象，然后在 mipssim.cc 中调用 Machine::OneInstruction(Instruction *instr)对其解码执行。

Machine::OneInstruction(Instruction *instr)中有一个非常大的 SWITCH 语句，该语句分析所取出的指令类型执行这条指令（Nachos 的指令类型见../machine/ mipssim.h）。

在 mipssim.cc 中 Machine::OneInstruction(Instruction *instr)的 switch 语句，对 Nachos 系统调用的处理方法是当 Nachos 的 CPU 检测到该条指令是执行一个 Nachos 的系统调用，则抛出一个异常 SyscallException 以便从用户态陷入到核心态去处理这个系统调用，代码如下：：

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

该异常 SyscallException 在../userprog/exception.cc 中进行处理，代码如下：

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

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

从上述代码中可以看出，系统将系统调用号保存在 MIPS 的 2 号寄存器$2 中，语句 type = machine->ReadRegister(2)从寄存器$2 中获取系统调用号，如果该条指令要调用 0 号系统调用（对应的宏是 SC_Halt），则执行 Halt()系

统调用的处理程序：

```
DEBUG('a', "Shutdown, initiated by user program.\n");
interrupt->Halt();          //停机
```

如果该条指令执行的是其它系统调用（Nachos 提供的系统调用参见../userprog/syscall.h），则 Nachos 异常退出（ASSERT()调用来 Abort()）。

这说明 Nachos 目前除了实现了系统调用 Halt()，其它的系统调用均未实现。

因此你需要参照../userprog/exception.cc 中对 Halt()的处理方法编写其它系统调用的处理代码。

代码大致如下：

```
Void ExceptionHandler(ExceptionType which) {
    int type = machine->ReadRegister(2);
    if ((which == SyscallException)) {
        switch(type){
            case SC_Halt:{
                DEBUG('a', "Shutdown, initiated by user program.\n");
                interrupt->Halt();
                break;
            }
            case SC_Exec:{
               ….
               break;
            }
            //其它系统调用出来程序
            default:{
                printf("Unexpected syscall %d %d\n", which, type);
                ASSERT(FALSE);
            }
        } // switch(type)
    } else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
} // ExceptionHandler(ExceptionType which)
```

注：Nachos 在处理系统调用时，系统调用类型及相关参数的传递方式，是通过约定几个寄存器实现的。

9.4.2 Nachos 系统调用机制

对于 Nachos 系统调用，在用户程序中的表现是调用../userprog/syscall.h 中定义的一系列系统调用函数。这些函数的调用入口（System call stubs）参见../test/start.s。

```
/* ------------------------------------------------------------
 * System call stubs:
 *      Assembly language assist to make system calls to the Nachos kernel.
 *      There is one stub per system call, that places the code for the
 *      system call into register r2, and leaves the arguments to the
 *      system call alone (in other words, arg1 is in r4, arg2 is
 *      in r5, arg3 is in r6, arg4 is in r7)
 *
 *      The return value is in r2. This follows the standard C calling
 *      convention on the MIPS.
 * ------------------------------------------------------------
 */
```

如系统调用 Exec()的入口汇编代码如下：

```
        .globl Exec       ;全局变量
        .ent  Exec        ;入口地址
    Exec:
        addiu $2,$0,SC_Exec    ; SC_Exec+$0->$2
                                  ;其中 S0=0, SC_Exec:系统调用号，即$2 存放系统调
用号
        syscall                ;执行系统调用
        j   $31                ;从系统调用中返回到原程序中继续执行
                                  ;$31 存放的是系统调用的返回地址
        .end Exec
```

系统调用的执行过程：

先分析一下 exec.c 源代码与其对应的汇编代码：

exec.cc 源代码（../test/exec.cc）：

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

exec.c 对应的汇编代码：

```
    .file   1 "exec.c"
gcc2_compiled.:
__gnu_compiled_c:
    .rdata
  .align   2
$LC0:
  .ascii   "../test/halt.noff\000"   //用户地址空间
  .text
  .align   2     //2 字节对齐，即 2*2
  .globl   main     //全局变量
  .ent   main   //main 函数入口
main:
#汇编伪指令  frame 用来声明堆栈布局。
#它有三个参数：
# 1）第一个参数  framereg：声明用于访问局部堆栈的寄存器，一般为  $sp。
# 2）第二个参数  framesize：申明该函数已分配堆栈的大小，应该符合  $sp ＋
#framesize ＝ 原来的  $sp。
# 3）第三个参数  returnreg：这个寄存器用来保存返回地址。
 #.frame：$fp 为栈指针，该函数层栈大小为 32 字节，函数返回地址存放在$31
 frame   $fp,32,$31       # vars= 8, regs= 2/0, args= 16, extra= 0
 mask   0xc0000000,-4
 fmask   0x00000000,0
 #栈采用向下生长的方式，即由大地址向小地址生长，栈指针指向栈的最小地址
 subu     $sp,$sp,32   # $sp - 32->$sp，构造 mian()的栈 frmae
                          #$sp 的原值应该是执行 main()之前的栈顶
                          #上一函数对应栈 frmae 的顶（最小地址处）
  sw       $31,28($sp)   #$31->memory[$sp+28]   ; 函数返回地址
  sw       $fp,24($sp)   #$fp ->memory[$sp+24]   ;保存 fp
  move   $fp,$sp   #$sp->$fp，执行 Exec()会修改$sp
  jal    __main       #PC+4->$31, goto __main
  la   $4,$LC0     #$LC0->$4 ,将 Exec("../test/halt.noff\000")的参数的地址传给$4
```

```
                    #$4-$7：传递函数的前四个参数给子程序，不够的用堆栈
   jal    Exec       #转到 start.s 中的 Exec 处执行
                     #/PC+4->$31, goto Exec；
                      #PC 是调用函数时的指令地址，
                      #PC+4 是函数的下条指令地址，以便从函数返回时从调用
                      #函数的下条指令开始继续执行原程序

     sw    $2,16($fp)   #$2->memory[$fp+16]，Exec()的返回值
                        #$2,$3: 存放函数的返回值，当这两个寄存器不够存放
                        #返回值时，编译器通过内存来完成。
                        #$sp 一直指向 main()对应 stack frame 的栈顶（最小地址）
                        #由于在调用 Exec()时要用到$sp，前面将$sp->$fp，
                        #因此$fp 也是指向 main()对应 stack frame 的栈顶
    jal    Halt         #PC+4->$31, goto Halt，Halt()无参，无返回值
$L1:
   move    $sp,$fp     #$fp->$sp
   lw    $31,28($sp)   #memory[/$sp+28]->$31，取 main()的返回值
   lw    $fp,24($sp)   #memory[$sp+24]->$fp，恢复$fp
   addu    $sp,$sp,32 #$sp+32->$sp，释放 main()对应的在栈中的 frame
   j    $31            #goto $31, mian()函数返回
                       #$31: Return address for subrouting
 .end    main
```

汇编语句 la　$4,$LC0;**将 Exec("../test/halt.noff")的参数（即要执行的文件名）的地址传给$4**，语句 jal　Exec 转移到前面所述的 start.s 中的 Exec 处开始执行。start.s 的 Exec 中**将系统调用号 SC_Exec（2 号系统调用）保存到 2 号寄存器$2**，然后执行 syscall。

从上述汇编代码中，你也可以了解系统对于栈的操作是如何进行的。

mipssim.cc 中 Machine::OneInstruction(Instruction *instr)的 switch 语句检测到该条指令要调用系统调用（case OP_SYSCALL:），则抛出异常 SyscallException，见下述代码：

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

该异常 SyscallException 在../userprog/exception.cc 中进行处理，异常处理程序如下：

```
Void ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

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

从 2 号寄存器$2 中获取系统调用号（type = machine->ReadRegister(2)），然后应该根据系统调用号做相应的处理。

对于系统调用 Exec(“../test/halt.noff”)，只携带了一个参数，我们可以从 4 号寄存器$4 中获取参数在内存中的地址，然后读出该参数（文件名）执行它。

9.4.3 Nachos 系统调用参数传递

我们知道，Exec(FilenNme)作为 Nachos 的系统调用，Exec(FilenNme)执行时需要陷入到 Nachos 的内核中执行，因此需要将其参数 FileName 从用户地址空间传递（复制）到内核中，从而在内核中为 FileName 对应的应用程序创建相应的线程执行它。

一般参数传递有三种方式：

（a）通过寄存器；

（b）通过内存区域，将该内存区域的首地址存放在一个寄存器中；

（c）通过栈；

在基于 MIPS 架构中，对于一般的函数调用，一般利用$4-$7（4 到 7 号寄存器）传递函数的前四个参数给子程序，参数多于 4 个时，其余的利用堆栈进行传递；

对于 Nachos 的系统调用，一般也是将要传递的参数依次保存到寄存器$4-$7 中，然后根据这些寄存器中的地址从内存中读出相应的参数。

特别要注意的是字符串作为参数时的传递方式，这时寄存器中保存的是字符串在内存中的地址。

例如，如果第一个参数是字符串，则将该字符串在内存中的地址存入$5，如果第二个参数是数值，则将该值存入$5，以此类推。

对于 Nachos 的系统调用，一般也是将要传递的参数在内存中的地址依次保存到寄存器$4-$7 中，然后根据这些寄存器中的地址从内存中读出相应的参数。

如前面的 exec.c 对应的汇编代码中，在执行系统调用 Exec()之前，利用指令 la $4,$LC0 将 Exec("../test/halt.noff”)中的参数"../test/halt.noff” 在内存中的地址$LC0 传给$4，然后执行 Exec，因此内核在处理系统调用 Exec 时，应该首先从$4 中获取"../test/halt.noff”的内存地址，然后将参数从内存中读出。

你自己可以考察一下下述程序的汇编代码：

```
#include “syscall.h”
int    main()
{
   Exit(1);
}
```

系统调用 Exit(1)对应的汇编代码为：

li s4,1　　#将立即数 0x1 存入寄存器$4；传递 Exit(1)中的参数

jal Exit　　#转到 start.s 中的 Exit 的调用入口

可以看出，对于参数为数值的，系统调用时系统将参数值按顺序依次传入$4-$7 中。

成员函数 Machine::ReadRegister(int num)可读取寄存器 num 中的内容，Machine::ReadMem(int addr, int size, int *value)从内存 addr 处读取 size 个字节的内容存放到 value 所指向的单元中。

在 Ecex()的实现代码中可以利用 Machine::ReadRegister(4)从$4 中获取"../test/halt.noff”的内存地址，然后利用 Machine::ReadMem(…)获取 FileName "../test/halt.noff”，然后为"../test/halt.noff”创建相应的进程及相应的核心线程，并将该进程映射到新建的核心线程上执行它。

当"../test/halt.noff”执行结束后，需要返回该线程的 pid（对应 Nachos 中的 SpaceId—Address Space Identifier，A unique identifier for an executing user program (address space)），可以利用 Machine::ReadRegister(2)将线程"../test/halt.noff”对应的 SpaceId 写入 2 号寄存器中，MIPS 架构将寄存器 2 和 3 存放返回值。

注：Machine::ReadRegister(int num) 在 Machine.cc 中实现，Machine::ReadMem(int addr, int size, int *value)在 translate.cc 中实现。

示例如下：

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

    if ((which == SyscallException)) {
        switch(type){
            case SC_Halt:{
                DEBUG('a', "Shutdown, initiated by user program.\n");
                interrupt->Halt();
                break;
            }
            case SC_Exec:{
                printf("Execute system call of Exec()\n");
```

```
                    //read argument
                    char filename[50];
                    int addr=machine->ReadRegister(4);
                    int i=0;
                    do{
                        //read filename from mainMemory
                        machine->ReadMem(addr+i,1,(int *)&filename[i]);
                    }while(filename[i++]!='\0');

                    printf("Exec(%s):\n",filename);
                    ……
                    …..
                    //return spaceID
                    machine->WriteRegister(2,space->getSpaceID());
                    ……
  }
}
```

9.4.4 Openfile for the User program

Nachos 启动时（主函数在 main.cc 中），通过语句 (void) Initialize(argc, argv)（参见 system.cc）初始化了一个 Nachos 基本内核，其中通过 Thread 类（参见 thread.cc）创建了一个 Nachos 的主线程"main"作为当前线程，并将其状态设为 RUNNING （ currentThread = new Thread("main"); currentThread->setStatus(RUNNING);），全局变量 currentThread 指向当前正在执行的线程（见 system.cc）。

Nachos 中只有第一个线程即主线程 main 是通过内核直接创建的，其它线程均需通过调用 Thread::Fork(…)创建。（只有主线程 main 是一个特例）；

当通过命令 nachos –x filename.noff 加载运行 Nachos 应用程序 filename.noff 时，通过../userprog/ progtest.cc 中的函数 StartProcess(char *filename)为该应用程序创建一个用户进程，分配相应的内存，建立用户进程（线程）与核心线程的映射关系，然后启动运行。

函数 StartProcess(char *filename)代码如下：

```
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);
    currentThread->space = space;     // currentThread 是主线程”main”
    delete executable;                // close file

    space->InitRegisters();           // set the initial register values
    space->RestoreState();            // load page table register

    machine->Run();             // jump to the user progam
    ASSERT(FALSE);                    // machine->Run never returns;
                                      // the address space exits
                                      // by doing the syscall "exit"
}
```

Nachos 自己定义了一种可执行文件格式（.noff）文件，由.coff 转换而来，通过命令../bin/coff2noff 将 coff 文件转换成 noff 文件。与常用的 COFF 文件相比，NOFF 文件格式相对简单，便于 Nachos 编程。

NOFF 文件格式：（参见../bin/noff.h，../userprog/AddrSpace 类中的 SwapHeader()）

- 文件头
- 代码段
- 已初始化数据段
- 未初始化数据的

其中，文件头给出了文件的基本信息，如文件标识，各个段的位置等，由下述两个结构描述：（参见../bin/noff.h）

```
typedef struct noffHeader {
        int noffMagic;          /* should be NOFFMAGIC */
        Segment code;           /* executable code segment */
        Segment initData;       /* initialized data segment */
        Segment uninitData;     /* uninitialized data segment --
                                 * should be zero'ed before use    */
} NoffHeader;
typedef struct segment {
    int virtualAddr;            /* location of segment in virt addr space */
    int inFileAddr;       /* location of segment in this file */
    int size;             /* size of segment */
} Segment;
```

NOFF 文件的 noffMagic=NOFFMAFIC=0xbadfad; 该 Magic 作为文件头部的一部分，位于文件头开始的两个字节，用于标识一种文件格式，也同时用户标识一种系统平台+文件格式。例如对于常用的 COFF 可执行文件，0x014c 相对于 I386 平台，0x268 相对于 Motorola 68000 系列，0x170 for the PowerPC family of processors。

类 AddrSpace 的构造函数中，首先依据文件头中的 Magic 判定该文件是否为一个有效的 NOFF 文件（是否为 0xbadfad），然后根据文件头中各段的大小（size）及需要的栈确定该程序分配需要的内存空间（帧数），如果内存大小满足要求，则为该程序创建相应的页表，将代码及数据等读入内存（分配帧），并在页表中设置页号与帧的映射关系（为简单起见，目前 Nachos 为应用程序分配到连续的帧中，即各页面对应的帧在内存中是顺序且是连续的）。参见 AddrSpace 类构造函数中代码：

```
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
```

从../userprog/ progtest.cc 中的函数 StartProcess(char *filename)的代码中可以

看出，要运行一个 Nachos 的应用程序，要为其创建相应的进程：首先要打开该文件（OpenFile *executable = fileSystem->Open(filename);），为该其分配内存空间（space = new AddrSpace(executable);），该内存空间标识作为该进程的进程号。将该进程映射到一个核心线程，初始化寄存器，运行它。参见../userprog/ progtest.cc 中的函数 StartProcess(char *filename)的下述代码：

```
currentThread->space = space;      // currentThread 是主线程"main"
delete executable;                 // close file

space->InitRegisters();            // set the initial register values
space->RestoreState();             // load page table register

machine->Run();                    // jump to the user progam
```

注 1：进程的 PCB 比较简单，主要包括进程 pid（SpaceID）、页表（代码、数据、栈）。

注 2：由于 Nachos 加载用户程序为其分配的内存空间时，总是将该用户程序分配到从 0 号帧开始的连续区域中，因此目前 Nachos 不支持多进程机制。

因此在实现系统调用 Exec(filename)时，需要为 filename 所对应的程序创建一个新的进程，并为其分配另外的内存空间（不能覆盖当前进程的地址空间）。

注 3：用户线程与核心线程采用 one-to-one 线程映射模型。例如执行下述代码时：

```
#include "syscall.h"
int    main()
{
    SpaceId pid;
    pid=Exec("../test/halt.noff");     //利用你所实现的 Exec()执行../test/halt.noff
    Exit(0);
    //Halt();
    /* not reached */
}
```

Nachos 首先为主程序（main()）创建一个进程，分配内存空间（从 0#帧开始），并将该进程映射到核心线程 main，该线程是 Nachos 启动时创建的唯一的一个线程（见../threads/system.cc 中 void Initialize(int argc, char **argv)）。

当执行 pid=Exec("../test/halt.noff");时，需要为"../test/halt.noff"创建一个新的进程，为其分配与主进程不同的内存空间，同时要利用 Thread::Fork(..)创建一个核心线程，然后将"../test/halt.noff"对应的进程映射到该核心线程中。（这里一个用户进程对应一个用户线程，目前不支持一个用户进程对应多个用户线程，除非你

实现了系统调用 Fork()。实现系统调用 Fork()后，用户可以多次调用它以在用户空间中创建多个并发执行的用户线程）。

注 4：文件../userprog/ progtest.cc 中函数 StartProcess(char *filename)的参数 filename 是一个 Nachos 内核中的对象（string），而系统调用 Exec(filename)中的参数 filename 是一个用户程序中的对象（string），前面给出的用户程序 exec.c 所对应的汇编代码说明了这一点：

9.4.5 Advance PC

../machine/ mipssim.cc 中的成员函数 void Machine::OneInstruction(Instruction *instr)，从 Nachos 的内存中取出并执行一条 Nachos 应用程序的指令。其中有一个非常大的 switch 语句，是分析所取出的指令类型执行这条指令（Nachos 的指令类型见../machine/ mipssim.h）。

其中下述 case 语句是对 Nachos 系统调用的处理，即当 Nachos 模拟的 CPU 检测到该条指令是执行一个 Nachos 的系统调用，则抛出一个异常 SyscallException 以便从用户态陷入到核心态去处理这个系统调用，代码如下：

```
// Execute the instruction (cf. Kane's book)
 {
     ……..
     switch (instr->opCode) {
         case OP_ADD:
             …..
             break;
         ……..
         case OP_SYSCALL:
              RaiseException(SyscallException, 0);
         return;
         ……
         default:
             ASSERT(FALSE);
     } // switch (instr->opCode)
     // Now we have successfully executed the instruction.
         // Do any delayed load operation
         DelayedLoad(nextLoadReg, nextLoadValue);

        // Advance program counters.
        registers[PrevPCReg] = registers[PCReg];
        registers[PCReg] = registers[NextPCReg];
        registers[NextPCReg] = pcAfter;
}
```

从中可以看出，当一条指令正常执行结束后，需要将 PC 推进，指向下一条指令，但 case OP_SYSCALL:中，RaiseException(SyscallException, 0);后不是一条 break 语句，而是一条 return 语句，原因是通常情况下，当处理完一个异常后需要重启这条指令。但系统调用异常是个特例，异常处理结束后指令不需要重启。

处理方法有两种，一种是将 RaiseException(SyscallException, 0);后的 return 修改为 break；二是在你的系统调用处理程序中添加 PC 的推进操作，否则，由于 return 造成的后果是没有对 pc 进行推进，程序就会再次读入执行这条系统调用操作，造成无限循环。

如果采用后者，可以在 exception.cc 模块中添加如下 PC 推进代码，并在你的各系统调用处理程序最后调用函数 AdvancePC()，如：

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

```
void ExceptionHandler(ExceptionType which)
{
        int type = machine->ReadRegister(2);
        if ((which == SyscallException)) {
            switch(type){
                case SC_Halt:{
                    DEBUG('a', "Shutdown, initiated by user program.\n");
                    interrupt->Halt();
                    break;
                }
                case SC_Exec:{
                     printf("Execute system call of Exec()\n");
                     //read argument
                     char filename[50];
                     int addr=machine->ReadRegister(4);
                     int i=0;
                     do {
                         //read filename from mainMemory
                         machine->ReadMem(addr+i,1,(int *)&filename[i]);
                      } while(filename[i++]!='\0');
                      //printf("Exec(%s):\n",filename);
                          ……
                          …..
                      //return spaceID
                      machine->WriteRegister(2,space->getSpaceID());
                       AdvancePC();     //PC 增量指向下条指令
                       break;
                } // case SC_Exec:
            // switch(type)
        } // if ((which == SyscallException))
}
```

9.4.6 SpaceId

Exec()返回类型为 SpaceId 的值（参见../userporg/syscall.h 中 typedef int SpaceId;，其实就是 process id--pid），该值将来作为系统调用 Join()的参数以识别新建的用户进程。

该值涉及的两个问题：

（1）它是如何产生的；

（2）在内核中如何记录它，以便 Join()能够通过该值找到对应的线程；

参考解决方案：

（1）核心可支持多线程机制，系统初始化时创建了一个主线程 main，以后可以利用 Thread::Fork()创建多个核心线程，这些核心线程可以并发执行；

其实，nachos 要求用户自己实现系统调用 Fork()，以创建多个并发执行的用户线程，与系统调用 Yield()联合使用。

目前在系统调用 Folk()实现之前，系统只能运行一个用户线程，不支持多线程机制。系统调用 Exec()可以在一个用户进程中加载另一个程序运行。

因此，进程的 Pid 即 SpaceId 可以这样产生：

从代码../userprog/protest.cc 中的 StartProcess()可以看出，加载运行一个应用程序的过程就是首先打开这个程序文件，为该程序分配一个新的内存空间，并将该程序装入到该空间中，然后为该进程映射到一个核心线程，根据文件的头部信息设置相应的寄存器运行该程序。

这里进程地址空间的首地址是唯一的，理论上可以利用该值识别该进程，但该值不连续，且值过大。

借鉴 UNIX 的思想，我们可以为一个地址空间或该地址空间对应的进程分配一个唯一的整数，例如 0~99 预留为核心进程使用（目前没有核心进程的概念，核心中只有线程），用户进程号从 100 开始使用。

目前 Nachos 没有实现子进程或子线程的概念（进程、线程之间没有建立进程树），因此，对于 Nachos 的第一个应用程序，其 SpaceId 即 Pid=100，当该程序调用 Exec(filename)加载运行 filename 指定的文件时，为 filename 对应的文件分配 101，以此类推。

当一个程序调用系统调用 Exit()退出时，应当收回为其分配的 pid 号，以分配给后续的应用程序，同时应释放其所占用的内存空间及页表等信息，供后续进程使用。

（2）Thread:: Fork(VoidFunctionPtr func, _int arg)有两个参数，一个是线程要运行的代码，另一个是一个整数，可以考虑将用户进程映射到核心线程时，利用 Fork()的第二个参数将进程的 pid 传入到核心中。

9.4.7 Join()

系统调用 int Join(SpaceId)的功能类似于 Pthread 中的 pthread_join(tid)，UNXI 系统调用 wait()的功能，int Join(SpaceId id)的功能是调用 Join(SpaceId id)的程序等待用户进程 id 结束，当用户进程 id 结束后，Join()返回进程 id 的退出状态（退出码）。

如 Nachos 的 shell 程序../test/shell.cc 中使用了 Join()使父进程等待子进程结束；

```
#include "syscall.h"
int main()
{
    SpaceId newProc;
    OpenFileId input = ConsoleInput;
    OpenFileId output = ConsoleOutput;
    char prompt[2], ch, buffer[60];
    int i;

    prompt[0] = '-';
    prompt[1] = '-';

    while( 1 )
    {
        Write(prompt, 2, output);
        i = 0;
        do {
            Read(&buffer[i], 1, input);
        } while( buffer[i++] != '\n' );
        buffer[--i] = '\0';
        if( i > 0 ) {
                newProc = Exec(buffer);
            Join(newProc);
        }
      }
  }
```

设计实现过程：

当子线程执行结束后，父线程执行 Join()时应该直接返回，不需等待；

由于目前 Nachos 没有实现线程家族树的概念，给 Join()的实现带来一些困难；

前面已经提到，AddrSpace 类代替了 PCB 的功能，因此可以在 AddrSpace 中设法建立进程之间的家族关系；

1、在 Thread 类中实现

（1）在 thread.cc 中添加函数 Join(int spaceId)，系统调用 int Join(int spaceId) 通过调用 Thread::Join()实现；

（2）Thread::Join()将当前线程睡眠，由于当前线程负责执行当前用户进程，因此效果是当前进程进入睡眠；

（a）在 Scheduler::Scheduler()中初始化一个线程等待队列及线程终止队

列；

目前 Nachos 只是在信号量的 P()操作中使用了等待队列；

Thread::Sleep()只是将当前线程的状态设置为 BLOCKED，然后调度下一个线程执行；（目前 Sleep()在信号量的 P、V 操作、Thread::Finish()调用）；

因此需要在构造函数 Scheduler::Scheduler()中初始化一个线程等待队列；

```
Scheduler::Scheduler()
{
    readyList = new List;          //thread ready queue
 #ifdef USER_PROGRAM
    waitingList = new List;        //如果 Joinee 没有退出，Joiner 进入等待
    terminatedList =new List;      //线程调用 Finish()后进入该状态
                                   //Joiner 通过检查该队列确定 Joinee 是否已经
                                   //退出
 #endif
}
```

（b）Thread::Join(spaceId)中依据所等待进程的 spaceId 检查其对应的线程是否已经执行完毕，如果尚未退出，则将当前线程进入等待队列，然后调用 Thread::Sleep()使当前线程睡眠（会引起线程调度），被唤醒后返回进程的退出码；

如果 Join(spaceId)所等待的进程已经结束，Join()应该直接返回，不需要等待；

如何检查所等待的进程（所对应的线程）是否已经退出？

线程在执行过程中，可能处于执行、就绪及阻塞状态，当线程执行信号量的 P()可能进入阻塞，也可能执行 I/O 操作进入阻塞，当线程终止时也先进入阻塞然后被销毁。

因此需要通过就绪队列与所有的等待队列以确定一个线程是否已经退出；

就绪队列只有一个，比较容易检查，但等待队列可能有多个（每个信号量都对应一个，且信号量的个数未知）；

为方便起见，为线程增加一个 TERMINATED 状态，相应地增加一个 terminated 队列，将所有的线程在调用 Finish()后先进入该队列，再伺机销毁；

这样我们可以通过检查 terminated 队列以确定一个线程是否已经终止（通过 Joinee 的 SpaceId 与 Joiner 所等待的 SpaceId 确定线程的身份）；

当 Joiner 执行 Thread::Join(spaceId)时，若 Joinee 在 terminated 队列，则从 terminated 队列移除 Joinee 并将其销毁，然后返回 Joinee 的退出码；

如果 Joinee 不在 terminated 队列，说明其尚未终止，则 Joiner 进入睡眠队列 waitingList，当 Joinee 退出调用 Finish()时通过检查 waitingList 以确定是否需要唤醒 Joiner。

当 Joiner 被唤醒后，需要从 terminated 队列移除 Joinee 并将其销毁，然后返回 Joinee 的退出码；

带来的一个问题是，那些没有被 Join()等待的线程何时从从 terminated 队列移除 Joinee 并将其销毁？

应该当父进程退出时将它们销毁。但 Nachos 没有记录进程的家族关系，难以标识父进程。

你可以自己建立进程家族树。何时建立？在哪里建立？

临时解决方案：为父进程设置特殊的退出码，在 Exit()中识别父进程，由父进程负责销毁 terminated 队列中的所有线程。

该方案还是存在 BUG，是否存在这样一种可能，terminated 队列中是否存在不属于该父进程的子进程，但被该父进程一并销毁？

注 1：系统调用 Exit(exitcode)应将进程的返回码传递给与其相关联的核心线程（Thread 应该增加一个成员变量存储之）；

注 2：需要在 Thread 中增加一个成员变量，保存与之相关联的进程 spaceId（pid）；

（c）当被等待的进程 Joinee 结束时，其对应的核心线程会自动执行 Thread::Finish()，因此需要修改 Thread::Finish()函数，当被等待的进程 Joinee 退出时，唤醒线程 Joiner；

注：上述实现比较复杂，是否可以考虑在 Thread::Join()与 Thread::Finish()利用信号量实现？（没有实现，不敢妄加断言）

利用方法 1 实现的代码大致如下：

exception.中系统调用 Join()代码如下：

```
    case SC_Join: {
        int SpaceId=machine->ReadRegister(4);   //ie. ThreadId or SpaceId
         currentThread->Join(spaceId);
         //返回 Joinee 的退出码 waitProcessExitCode
         machine->WriteRegister(2, currentThread->waitProcessExitCode);
         AdvancePC();
         break;
 }
```

int Thread::Join(int SpaceId)代码大致如下：

```
#ifdef USER_PROGRAM
void Thread::Join(int SpaceId)    { //int join(SpaceId)
    intStatus oldLevel = interrupt->SetLevel(IntOff);
    currentThread->waitingProcessSpaceId = SpaceId;
    //if joinee is still in not in terminated list?
    Thread *thread;
    List *terminatedList = scheduler->getTerminatedList();
    List *waitingList = scheduler->getWaitingList();
    bool interminatedList = FALSE;
    int listLength = terminatedList->ListLength();    //length of ready queue
    for (int i = 1; i <= listLength; i++)
    {
        thread = (Thread *)terminatedList->getItem(i);
        if (thread == NULL)
            interminatedList = FALSE;      // joinee not finished
        //joinee is still in Ready queue, not finished
        if (thread->UserProgramId == SpaceId)
        {
            interminatedList = TRUE;    // joinee alreday finished
            break;
        }
        interminatedList = FALSE;      // joinee not finished
    }
    //Joinee is not finished, still in not in
    // terminated List(not at TERMINALTE),
    if (!interminatedList)
    // still in Nanchos system, maybe at READY or BLOCKED
    {
        waitingProcessSpaceId = SpaceId;
        waitingList->Append((void *)this);      //blocked Joiner
        currentThread->Sleep();
    }
    //joinee alreday finished, in terminated List, empty it, and return
    currentThread->waitProcessExitCode = waitingThreadExitCode;
    //delete terminated thread "Joinee"
    scheduler->deleteTerminatedThread(SpaceId);
    interrupt->SetLevel(oldLevel);
}
#endif
```

void Thread::Finish ()代码大致如下:

```
int waitingThreadExitCode;
void Thread::Finish ()
{
    (void) interrupt->SetLevel(IntOff);
    ASSERT(this == currentThread);
#ifdef USER_PROGRAM
    waitingThreadExitCode = currentThread->getExitStatus();
    //joinee finised, wakeup the join user program
    List *ReadyList = scheduler->getReadyList();
    List *waitingList = scheduler->getWaitingList();
    Thread *waitingThread;

    // if joiner is sleeping and in waitinglist, joinee wait up joiner
    // when joinee finish
    int listLength = waitingList->ListLength();
    for (int i = 1; i <= listLength; i++)
    {
        waitingThread = (Thread *)waitingList->getItem(i);
        if (currentThread->UserProgramId == waitingThread->waitingProcessSpaceId)
        {
            scheduler->ReadyToRun((Thread *)waitingThread);
            waitingList->RemoveItem(i);
            break;
        }
     }
    Terminated();
 #else
      threadToBeDestroyed = currentThread;
      Sleep();                          // invokes SWITCH
      // not reached
 #endif
}
```

void Thread:: Terminated ()代码大致如下:

```
#ifdef USER_PROGRAM
void Thread::Terminated()
{
      List *terminatedList = scheduler->getTerminatedList();

      Thread *nextThread;

      ASSERT(this == currentThread);      // a thread sleep by itsef
      ASSERT(interrupt->getLevel() == IntOff);
      status = TERNINATED;
      terminatedList->Append((void *)this);      //

      nextThread = scheduler->FindNextToRun();
      while(nextThread == NULL)
     {
         interrupt->Idle();
         nextThread = scheduler->FindNextToRun();
     }
     scheduler->Run(nextThread); // returns when we've been signalled
}
#endif
```

2、是否可以不修改内核，在系统在系统调用 Join()与 Exit()利用信号量实现？（未实现，需要商榷）

在系统调用 Join()中将父进程利用 P()操作进入睡眠，当等待的子进程执行 Exit()退出时利用 V()将其唤醒；

涉及到进程退出时（Exit()）需要唤醒等待该进程结束的父进程；特别是当父进程运行执行 Join()时，如果子进程已经退出，父进程不需等待，直接返回，在 Thread 类的外部几乎无法获知线程的这些操作信息，致使实现比较困难；

推荐利用第 1 种方法在内核的 **Thread 类**中实现

9.4.8 Exec()

系统调用 pid=Exec(filename)的功能是加载运行应用程序 filename，如：

```
int main() {
    SpaceId pid;
    pid=Exec("../test/halt.noff");      //filemame="../test/halt.noff"
    Exit(0);
}
```

设计与实现思路：

1、修改 exception.cc，根据系统调用类型对各系统调用进行处理；

(1) 从 2 号寄存器中获取当前的系统调用号（type=machine->ReadRegister(2)），根据 type 对系统调用分别处理；

系统调用号参见../userprog/syscall.h，根据系统调用号对本次系统调用做相应的处理；

(2)获取系统调用参数（寄存器 4、5、6、7，可以携带 4 个参数）

在 Exec(char *filename)的处理代码中，

(a)从第 4 号寄存器中获取 Exec()的参数 filename 在内存中的地址（addr=machine->ReadRegister(4)）；

(b)利用 Machine::ReadMem()从该地址读取应用程序文件名 filename；

(c) 打开该应用程序（OpenFile *executable = fileSystem->Open(filename)）；

(d) 为其分配内存空间、创建页表、分配 pid（space = new AddrSpace(executable)），至此为应用程序创建了一个进程；

注 1：目前 AddrSpace:: AddrSpace()中为应用程序分配内存时，都是分配了从 0#帧开始的一个连续的内存空间，因此执行 Exec(char *filename)时，为应用程序 filename 所分配的内存空间不应该从 0#帧开始，而应该从第一个空闲帧开始；

注 2：目前 AddrSpace:: AddrSpace()没有为进程分配 pid；

注 3：目前 AddrSpace:: AddrSpace()没有对空闲帧进行管理

因此，需要修改 AddrSpace:: AddrSpace()实现上述功能；同时，需要修改 AddrSpace:: ~AddrSpace()，进程退出时释放 pid，为其所分配的帧也应释放（修改空闲帧位示图）；

因此需要修改 AddrSpace:: AddrSpace()与 AddrSpace:: ~AddrSpace()；

(e)创建一个核心线程，并将该进程与新建的核心线程关联（thread = new Thread(forkedThreadName)，thread->Fork(StartProcess, space->getSpaceID())）；

需要特别指出的是，通过 Thread::Fork()创建的线程需要指明该线程要执行的代码（函数）及函数所需的参数；

我们可以重载函数 StartProcess(int spaceId)，作为新建线程执行的代码，并将进程的 pid 传递给系统，供其它系统调用（如 Join()）使用；

当调度到该线程时，就启动应用程序进程的执行；

2、修改 progtest.cc，重载函数 StatProcess(char *filename)

将该函数作为应用程序进程所关联的核心线程的执行代码，当调度到该线程时，Exec(filename)中 filename 所对应的应用程序进程随即执行；

```
void StartProcess(int spaceId)
{
    space->InitRegisters();     // set the initial register values
    space->RestoreState();          // load page table register

    machine->Run();             // jump to the user progam
    ASSERT(FALSE);              // machine->Run never returns;
                                // the address space exits by doing the syscall "exit"
}
```

3、修改 AddrSpace::AddrSpace()及 AddrSpace::~AddrSpace()

为管理空闲帧，建立一个全局的空闲帧管理位示图；

为管理 pid（spaceId），建立一个全局的 pid 数组；

从内存的第一个空闲帧为 Exec(filename)中的 filename 分配内存空间，创建该进程页表，建立虚实页表的映射关系，分配 pid；

在释放应用程序内存空间时，应该清除空闲帧位示图相应的标志，释放 pid，释放页表。

AddrSpace::AddrSpace()及 AddrSpace::~AddrSpace()的代码参考如下：

```
#define MAX_USERPROCESSES 256
BitMap *bitmap; //for free frame
bool ThreadMap[MAX_USERPROCESSES];    //pid or SpaceId
AddrSpace::AddrSpace(OpenFile *executable)
{
    //----added by yourself----------------------------------------
    //allow up to MAX_USERPROCESSES user processes executables concurrently
     //spaceID, i.e. pid
     bool hasAvailabePid = flase;
     for(int i = 100; i < MAX_USERPROCESSES; i++) {
          if(!ThreadMap[i]){
               ThreadMap[i] = true;
               spaceID = i;    //may be should reserved 0-99 for kernel Process,
              //even though there is no any process at present
              hasAvailabePid = true;    //ther is available Pid for new process
              break;
          }
     } //for
     if (!hasAvailabePid)      //no available Pid for new process
     {
          printf("Too many processes in Nachos!\n");
          return;
     }
     if(bitmap == NULL)      //used for free frames
          bitmap = new BitMap(NumPhysPages);
        //the remaining code
     //set up a new PageTable for a process, first, set up the translation
     pageTable = new TranslationEntry[numPages];
     for (i = 0; i < numPages; i++) {
         pageTable[i].virtualPage = i;      // virtual page #
         pageTable[i].physicalPage = bitmap->Find();    // find a free frame
         ASSERT(pageTable[i].physicalPage!=-1);
         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
     }
```

```
        //the remaining code
}
```

在系统调用 Exit()中调用析构函数释放为该进程所分配的资源，包括进程 pid，为进程分配的帧，以及页表。

```
AddrSpace::~AddrSpace()
{
   ThreadMap[spaceID] = 0; //false
   for (int i = 0; i < numPages; i++) {
        bitmap->Clear(pageTable[i].physicalPage);
    }

    delete [] pageTable;
}
```

Exec()系统调用小结：

（1）获取 Exec(char *filename)的参数 filename；通过读取$4 寄存器获得 filename 在内存中的地址（文件名在栈中的地址），然后从该地址中读出参数 filename；

（2）打开应用程序 filename；（参考 progtest.cc 函数 StatProcess()）

（3）为应用程序及栈分配内存空间并为其建立页表，读入应用程序代码及数据。

但目前的实现是 fstest.cc 的 StatProcess()每加载一个应用程序，都是从 0#帧开始为其分配内存。因此需要记录目前内存中从几号帧开始是空闲的（为内存帧建立一个位示图 bitmap 对象，用于管理空闲帧）。因此你需要修改 AddrSpace 的构造函数以满足上述要求；

AddrSpace 的构造函数中，首先根据文件头的信息确定文件大小，根据文件大小及栈大小确定需要为其分配多少个帧，如果剩余的内存空间大小满足该文件的要求，则为其创建页表，然后将文件内容读入为其所分配的帧中。如果目前内存不足，给出相应的提示后退出。

注：文件内容在内存中的顺序：先是代码段，然后是已初始化的数据段，未初始化的数据的没有读入主存，应该是存储在用户栈中。

还应该在 AddrSpace 的构造函数中产生应用程序的 SpaceId，即我们熟知的 pid。pid 可以从 0 开始，也可以从 100 开始，将 0~99 预留给核心进程（尽管目前没有核心进程的概念）。可以定义一个全局的 spaceId（pid 数组）；

（4）创建一个核心线程，将该应用程序映射到该核心线程。

系统将 main()对应的主程序映射到核心线程的主线程”main”，让主线程

执行主程序。（参见 progtest.cc 中 StatProcess()）。

对于应用程序 filename，需要我们自己调用 Thread::Folk()创建一个核心线程，并将该应用程序映射到该核心线程，以执行应用程序 filename。

（5）初始化寄存器，开始程序执行

将所有设计寄存器请 0。

由于读入程序时，从代码段的入口地址（虚地址）开始读入到内存的 0 号地址中（即第 0 帧的 0 号偏移量处），因此程序从逻辑地址 0 开始执行（物理地址也为 0），将 PC 设置为程序的逻辑地址入口 0。

../machine/translate.cc 中 machine::readmem()调用 translate()将 PC 中的逻辑地址转换为物理地址，然后从该物理地址中将指令读出。

（6）返回进程的 pid

（7）PC 增量

实现代码形如：

```
case SC_Exec:{
        char filename[128];
         int addr=machine->ReadRegister(4);
         int i=0;
         do{
              machine->ReadMem(addr+i,1,(int *)&filename[i]);
         }while(filename[i++]!='\0');
              OpenFile *executable = fileSystem->Open(filename);
         if (executable == NULL) {
              printf("Unable to open file %s\n", filename);
              return;
         }
         //new address space
         space = new AddrSpace(executable);
         delete executable;      // close file
         /new and fork thread
         char *forkedThreadName=filename;
         thread = new Thread(forkedThreadName);
         thread->Fork(StartProcess, space->getSpaceID());
         thread->space = space;   //用户线程映射到核心线程
         //return spaceID
         machine->WriteRegister(2,space->getSpaceID());
         AdvancePC();
         break;
 }
```

9.4.9 Exit() and Exit Status

系统调用 void Exit(int status)的参数 status 是用户程序的退出状态。系统调用 int Join(SpaceId id)需要返回该退出状态 status。由于可能在 id 结束之后，其它程序（如 parent）才调用 Join(SpaceId id)，因此在 id 执行 Exit(status)退出时需要将 id 的退出码 ststus 保存起来，以备 Join()使用。

关于系统调用 Exit()的实现，首先从 4 号寄存器读出退出码，然后释放该进程的内存空间及其其表，释放分配给该进程的实页（帧），释放其 pid（参见 AddrSpace::~AddrSpace()），调用 currentThread->Finish 结束该进程对应的线程。

管理空闲帧的位示图以及 pid 结构不能释放，因为它们是全局的。

实现代码形如：

```
case SC_Exit:{
    int ExitStatus=machine->ReadRegister(4);
    machine->WriteRegister(2,ExitStatus);
    currentThread->setExitStatus(ExitStatus);
    if (ExitStatus == 99)    //parent process exit, delele all terminated threads
    {
        List *terminatedList = scheduler->getTerminatedList();
        scheduler->emptyList(terminatedList);
    }
    delete currentThread->space;
    currentThread->Finish();
    AdvancePC();
    break;
}
```

9.4.10 Yield()

void Yield()功能：

Yield the CPU to another runnable thread, whether in this address space or not.

可简单调用 Thread::Yield()实现；

```
case SC_Yield:{
    currentThread->Yield();
    AdvancePC();
    break;
}
```

9.4.11 调试时注意的问题

在调试该部分代码时，如果使用参数-rs 实现分时，或利用系统调用 Yield()主动释放 CPU，Thread::getName()的输出的线程名可能有问题，但不影响线程的执行，这时由于 Thread 类的构造函数传入线程名时使用指针导致的问题。

考察 Thread 类的构造函数：

```
Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;
#endif
}
```

name = threadName 将线程名指针传给私有成员变量 name；

当一个函数调用 Thread::Thread(…)创建线程时，如：

```
char *thname = ‘forked thread’;
Thread *thread = new Thread(thname)，
```

当创建线程的函数退出时，字符指针 char *thname 也随之释放，Thread 类中的 name 会指向一个未知的位置，线程名就不是我们所期望的内容；

可以将 Thread 类中的 char* name;修改为 char* name= new char[50]，构造函数相应地修改为：

```
Thread::Thread(char* threadName)
{
    strcpy(name,threadName);
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;

#ifdef USER_PROGRAM
    space = NULL;
```

```
        //startUserProcess = FALSE;    //han, if this user process execute from begining
#endif
}
```

9.5 基于 FILESYS_STUB 实现文件的有关系统调用

尽管目前设计暂不要求实现该部分的内容，如有时间可以实现。

目前这些涉及文件的几个系统调用（Create、Open、Write、Read、Close）是基于 FILESYS_STUB 定义的方法实现的；

前面已经提到，Nachos 的文件系统有两个版本，基于宏 FILESYS_STUB 与 FILESYS 进行条件编译。参见../filesys/filesys.h；

利用宏 FILESYS_STUB 对 Nachos 模块进行条件编译，实现了文件系统的部分功能，直接使用 Linux 的系统调用实现了对文件的操作，访问的是 Linux 文件系统中的文件，如 Exec(filename)访问的是../test 目录中 Linux 文件而不是 Nachos 硬盘 DISK 上的文件 file，因此形式为 Exec("../test/halt.noff");

基于 FILESYS 条件编译实现的文件系统，文件的操作是基于 Nachos 的 Openfile 类实现的，访问的是 Nachos 硬盘 DISK 中的文件，如 Exec(filename)访问的是 Nachos 硬盘 DISK 上的文件 filename，形式为 Exec("halt.noff")，文件 halt.noff 应该在 Nachos 仿真的硬盘 DISK 中，否则无法访问;（由于 Nachos 硬盘只是支持一级目录，因此无需文件无需包含全路径）。

实现两个版本的原因是当实现 Exec()、Exit()及相应的文件操作的系统调用时，如果实验 4、5 关于 Nachos 的文件系统的相关功能尚未实现，可以使用 FILESYS_STUB 中实现的文件系统访问 Linux 中的文件；如果实验 4、5 已经完成，可以使用 FILESYS 中实现的文件系统，直接访问 Nachos 仿真硬盘 DISK 中的文件；

在../lab7-8/Makefile 文件的最后几行如下：

```
# If the filesystem assignment is done before userprog, then
# uncomment the include below

include ../threads/Makefile.local
#include ../filesys/Makefile.local
include ../lab7-8/Makefile.local          //
#include ../userprog/Makefile.local
include ../Makefile.dep
include ../Makefile.common
```

在../userprog/Makefile 文件的最后几行如下：

```
# If the filesystem assignment is done before userprog, then
# uncomment the include below
include ../threads/Makefile.local
#include ../filesys/Makefile.local
include ../userprog/Makefile.local
include ../Makefile.dep
include ../Makefile.common
```

在../lab7-8/makefile.local，或../userprog/Makefile.local 文件的最后几行如下：

```
ifdef MAKEFILE_FILESYS_LOCAL
DEFINES += -DUSER_PROGRAM
else
DEFINES += -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS_STUB
endif
```

从上面的 Makefile 文件中可以看出，**include ../filesys/Makefile.local 被默认加了注释**（Make 中使用#注释一行，不是用//），考察../filesys/Makedile.local 会发现，其中定义了宏 MAKEFILE_FILESYS_LOCAL，由于上面的 Makefile 文件中 include ../filesys/Makefile.local 被注释掉，因此宏 MAKEFILE_FILESYS_LOCAL 尚未定义；

再考察上面的 Makefile.local 文件，如果 MAKEFILE_FILESYS_LOCAL 未被声明，编译参数 DEFINES 取值 DEFINES += -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS_STUB；

其中参数-DUSER_PROGRAM 相当于在所有的 Nachos 源程序模块中利用#define USER_PROGRAM，效果是对源程序条件 #ifdef USER_PROGRAM …#endif 中的语句也进行编译，这些语句由实验 6、7、8 使用；对于实验 1~5，对#ifdef USER_PROGRAM …#endif 所包括的语句不进行编译；

参数-DFILESYS_NEEDED 在 system.cc 中可以发现它的作用，用于创建文件系统；

参数-DFILESYS_STUB 相当于在所有的 Nachos 源程序模块中利用#define FILESYS_STUB 声明了宏 FILESYS_STUB，使用的是 #ifdef FILESYS_STUB …#endif 中定义的文件系统操作，如果未定义 FILESYS_STUB 而是定义了 FILESYS（如果未定义宏，也相当于定义了宏 FILESYS），就使用 Nachos 的文件操作；

因此，如果在实验 4、5 尚未完成的情况下实现实验 6、7、8，就将上述 Makefile 中的 include ../filesys/Makefile.local 注释掉（形如#include ../filesys/Makefile.local），就是用 FILESYS_STUB 定义的文件操作；

如果在实验 4、5 已经完成的情况下实现实验 6、7、8，就将上述 Makefile 中的 #include ../filesys/Makefile.local 的注释去掉（形如 include ../filesys/Makefile.local），就是用 FILESYS 定义的文件操作；（参见../filesys/filesys.h）

下述系统调用使用的是 FILESYS_STUB 定义的文件系统实现的；

9.5.1 Create()

```
/* Create a Nachos file, with "name" */
//void Create(char *name);
```

```
case SC_Create: {
                int base=machine->ReadRegister(4);
                int value;
                int count=0;
                char *FileName= new char[128];

                do{
                      machine->ReadMem(base+count,1,&value);
                      FileName[count]=*(char*)&value;
                      count++;
                } while(*(char*)&value!='\0'&&count<128);
                  int fileDescriptor = OpenForWrite(FileName);
          if (fileDescriptor == -1)
                printf("create file %s failed!\n",FileName);
            else
                printf("create file %s succeed!, the file id is %d\n",FileName,fileDescriptor);

       Close(fileDescriptor);
            //machine->WriteRegister(2,fileDescriptor);

       AdvancePC();
break;
 }
```

9.5.2 Open()

```
/* Open the Nachos file "name", and return an "OpenFileId" that can
* be used to read and write to the file.
*/
    //OpenFileId Open(char *name);    //int OpenFileId
```

```
case SC_Open: {
        int base=machine->ReadRegister(4);
        int value;
        int count=0;
        char *FileName= new char[128];

        do{
            machine->ReadMem(base+count,1,&value);
            FileName[count]=*(char*)&value;
            count++;
        } while(*(char*)&value!='\0'&&count<128);
          int fileDescriptor = OpenForReadWrite(FileName, FALSE);
          if (fileDescriptor == -1)
                printf("Open file %s failed!\n",FileName);
                else
                    printf("Open file %s succeed!, the file id is %d\n",FileName,fileDescriptor);
                machine->WriteRegister(2,fileDescriptor);
          AdvancePC();
          break;
}
```

9.5.3 Write()

/* Write "size" bytes from "buffer" to the open file. */
//void Write(char *buffer, int size, OpenFileId id);

```
case SC_Write: {
        int base =machine->ReadRegister(4);    //buffer
        int size=machine->ReadRegister(5);      //bytes written to file
        int fileId=machine->ReadRegister(6); //fd
        int value;
     int count=0;

       // printf("base=%d, size=%d, fileId=%d \n",base,size,fileId );
        OpenFile* openfile =new OpenFile (fileId);
        ASSERT(openfile != NULL);

        char* buffer= new char[128];
        do{
           machine->ReadMem(base+count,1,&value);
           buffer[count] = *(char*)&value;
         count++;
        } while((*(char*)&value!='\0') && (count<size));
        buffer[size]='\0';

        int WritePosition;
        if (fileId==1)
            WritePosition = 0;
        else
            WritePosition = openfile->Length();

        int writtenBytes=openfile->WriteAt(buffer,size,WritePosition);
        if((writtenBytes)==0)
            printf("write file failed!\n");
        else
            printf("\"%s\" has wrote in file %d succeed!\n",buffer,fileId);
        //machine->WriteRegister(2,size);
        AdvancePC();
        break;
}
```

9.5.4 Read()

```
/* Read "size" bytes from the open file into "buffer".
  * Return the number of bytes actually read -- if the open file isn't
  * long enough, or if it is an I/O device, and there aren't enough
  * characters to read, return whatever is available (for I/O devices,
  * you should always wait until you can return at least one character).
  */
//int Read(char *buffer, int size, OpenFileId id);
```

```
  case SC_Read: {
        int base =machine->ReadRegister(4);
        int size    = machine->ReadRegister(5);
        int fileId=machine->ReadRegister(6);

    OpenFile* openfile=new OpenFile(fileId);
    char buffer[size];
    int readnum=0;
    readnum=openfile->Read(buffer,size);

       for(int i = 0;i < size; i++)
            if(!machine->WriteMem(base,1,buffer[i]))
              printf("This is something wrong.\n");
    buffer[size]='\0';
    printf("read succeed! the content is \"%s\" , the length is %d\n",buffer,size);
       machine->WriteRegister(2,readnum);
       AdvancePC();
       break;
  }
```

9.5.5 Close ()

```
/* Close the file, we're done reading and writing to it. */
//void Close(OpenFileId id);
```

```
case SC_Close: {
        int fileId =machine->ReadRegister(4);
        //printf("SC_Close: fileId in $4 = %d\n",fileId);
        //void Close(int fd) in sysdep.cc

        //OpenFile* openfile=new OpenFile(fileId);
        //delete openfile;    //does not work well
        Close(fileId);

        printf("File %d    closed succeed!\n",fileId);
        AdvancePC();
        break;
 }
```

9.6 基于 FILESYS 实现文件的有关系统调用

下述文件的系统调用是基于 FILESYS 定义的方法实现的（即基于实验 4、5 中实现的 Nachos 文件系统），需要将下述两个文件的 #include ../filesys/Makefile.local 的注释去掉，内容如下：

../lab7-8/Makefile 文件的最后几行：

```
# If the filesystem assignment is done before userprog, then
# uncomment the include below

include ../threads/Makefile.local
include ../filesys/Makefile.local
include ../lab7-8/Makefile.local            //
#include ../userprog/Makefile.local
include ../Makefile.dep
include ../Makefile.common
```

../userprog/Makefile 文件的最后几行：

```
# If the filesystem assignment is done before userprog, then
# uncomment the include below
```

```
include ../threads/Makefile.local
include ../filesys/Makefile.local
include ../userprog/Makefile.local
include ../Makefile.dep
include ../Makefile.common
```

在 AddrSpace 类中添加如下成员变量与成员函数：

```
int spaceID;    //pid
//文件描述符，0,1,2 分别为 stdin,stdout 与 stderr

OpenFile* fileDescriptor[10];
int getFileDescriptor(OpenFile * openfile);
OpenFile* getFileId(int fd);
void releaseFileDescriptor(int fd);
```

在 AddrSpace 的构造函数中初始化该进程打开文件的文件描述符：

```
……
  for (int i=3;i<10;i++)       //up to open 10 file for each process
        fileDescriptor[i] = NULL;

    OpenFile *StdinFile = new OpenFile("stdin");
    OpenFile *StdoutFile = new OpenFile("stdout");
    fileDescriptor[0] =    StdoutFile;
    fileDescriptor[1] =    StdoutFile;
    fileDescriptor[2] =    StdoutFile;
……..
```

几个函数的代码如下：

```
int AddrSpace::getFileDescriptor(OpenFile * openfile)
 {
    for (int i=3;i<10;i++)
    {
      if (fileDescriptor[i] == NULL)
        {
            fileDescriptor[i] = openfile;
            return i;
        }   //if
    }   //for
    return -1;
 }
```

```
OpenFile* AddrSpace::getFileId(int fd)
{
    return fileDescriptor[fd];
}
```

```
void AddrSpace::releaseFileDescriptor(int fd)
{
    fileDescriptor[fd] = NULL;
}
```

重载 OpenFile 类的构造函数：

```
OpenFile::OpenFile(char* type) {}
```

在 OpenFile 类中添加两个标准 I/O 设备：

```
int
OpenFile::WriteStdout(char *from, int numBytes) {
        int file = 1;    //stdout
        WriteFile(file, from, numBytes);
        //retVal = write(fd, buffer, nBytes);
        return numBytes;
    }
```

```
int
OpenFile::ReadStdin(char *into, int numBytes) {
        int file = 0;    //stdin
        return ReadPartial(file, into, numBytes);
}
```

9.6.1 Create()

```
/* Create a Nachos file, with "name" */
//void Create(char *name);
case SC_Create: {
    int base=machine->ReadRegister(4);
    int value;
    int count=0;
    char *FileName= new char[128];

    do{
        machine->ReadMem(base+count,1,&value);
        FileName[count]=*(char*)&value;
        count++;
    } while(*(char*)&value!='\0'&&count<128);

    //when calling Create(),    thread go to sleep, waked up when I/O finish
    if(!fileSystem->Create(FileName,0))    //call Create() in FILESYS,see filesys.h
        printf("create file %s failed!\n",FileName);
    else
        DEBUG('f',"create file %s succeed!\n",FileName);
    AdvancePC();
    break;
}
```

9.6.2 Open()

```
/* Open the Nachos file "name", and return an "OpenFileId" that can
* be used to read and write to the file.
*/
    //OpenFileId Open(char *name);    //int OpenFileId
```

```
case SC_Open: {
        int base=machine->ReadRegister(4);
        int value;
        int count=0;
        char *FileName= new char[128];

        do{
            machine->ReadMem(base+count,1,&value);
            FileName[count]=*(char*)&value;
            count++;
        } while(*(char*)&value!='\0'&&count<128);

          int fileid;
          //call Open() in FILESYS,see filesys.h,Nachos Open()
          OpenFile* openfile=fileSystem->Open(FileName);
          if(openfile == NULL ) {      //file not existes, not found
              printf("File \"%s\" not Exists, could not open it.\n",FileName);
              fileid = -1;
          }
          else {    //file found
              //set the opened file id in AddrSpace, which wiil be used in Read() and Write()
              fileid = currentThread->space->getFileDescriptor(openfile);
              if (fileid < 0)
                  printf("Too many files opened!\n");
              else
                  DEBUG('f',"file :%s open secceed!    the file id is %d\n",FileName,fileid);
          }
          machine->WriteRegister(2,fileid);
          AdvancePC();
          break;
}
```

9.6.3 Write()

/* Write "size" bytes from "buffer" to the open file. */
//void Write(char *buffer, int size, OpenFileId id);

```
case SC_Write: {
        int base =machine->ReadRegister(4);    //buffer
        int size=machine->ReadRegister(5);      //bytes written to file
        int fileId=machine->ReadRegister(6); //fd
        int value;
    int count=0;

      // printf("base=%d, size=%d, fileId=%d \n",base,size,fileId );
        OpenFile* openfile =new OpenFile (fileId);
        ASSERT(openfile != NULL);

        char* buffer= new char[128];
        do{
           machine->ReadMem(base+count,1,&value);
           buffer[count] = *(char*)&value;
         count++;
        } while((*(char*)&value!='\0') && (count<size));
        buffer[size]='\0';

         OpenFile*    openfile = currentThread->space->getFileId(fileId);
         //printf("openfile =%d\n",openfile);
         if (openfile == NULL)
           {
                printf("Failed to Open file \"%d\" .\n",fileId);
                AdvancePC();
                break;
           }

           if (fileId ==1 || fileId ==2)
           {
                openfile->WriteStdout(buffer,size);
                delete [] buffer;
                AdvancePC();
                break;
           }

           int WritePosition = openfile->Length();
```

```
            openfile->Seek(WritePosition);    //append write
            //openfile->Seek(0);            //write form    begining

            int writtenBytes;
            //writtenBytes = openfile->AppendWriteAt(buffer,size,WritePosition);
            writtenBytes = openfile->Write(buffer,size);
            if((writtenBytes)==0)
                DEBUG('f',"\nWrite file failed!\n");
            else
            {
                if (fileId != 1 && fileId != 2)
                {
                   DEBUG('f',"\n\"%s\" has wrote in file %d succeed!\n",buffer,fileId);
                   DEBUG('H',"\n\"%s\" has wrote in file %d succeed!\n",buffer,fileId);
                   DEBUG('J',"\n\"%s\" has wrote in file %d succeed!\n",buffer,fileId);
                }
                //printf("\n\"%s\" has wrote in file %d succeed!\n",buffer,fileId);
            }

          //delete openfile;
           delete [] buffer;
        AdvancePC();
        break;
}
```

9.6.4 Read()

```
/* Read "size" bytes from the open file into "buffer".
  * Return the number of bytes actually read -- if the open file isn't
  * long enough, or if it is an I/O device, and there aren't enough
  * characters to read, return whatever is available (for I/O devices,
  * you should always wait until you can return at least one character).
  */
//int Read(char *buffer, int size, OpenFileId id);
```

```
  case SC_Read: {
        int base =machine->ReadRegister(4);
        int size    = machine->ReadRegister(5);
        int fileId=machine->ReadRegister(6);

        OpenFile* openfile = currentThread->space->getFileId(fileId);

        char buffer[size];
        int readnum=0;
        if (fileId == 0)    //stdin
            readnum = openfile->ReadStdin(buffer,size);
        else
            readnum = openfile->Read(buffer,size);

        for(int i = 0;i < readnum; i++)
            machine->WriteMem(base,1,buffer[i]);
        buffer[readnum]='\0';

        for(int i = 0;i < readnum; i++)
           if (buffer[i] >=0 && buffer[i] <= 9)
                buffer[i] = buffer[i] + 0x30;

         char *buf = buffer;
         f (readnum > 0)
         {
              if (fileId != 0)
              {
                 DEBUG('f',"Read file (%d) succeed! the content is \"%s\" , the length is %d\n",fileId,buf,readnum);
                 DEBUG('H',"Read file (%d) succeed! the content is \"%s\" , the length is %d\n",fileId,buf,readnum);
                 DEBUG('J',"Read file (%d) succeed! the content is \"%s\" , the length is %d\n",fileId,buf,readnum);
```

```
            }
        }
        else
            printf("\nRead file failed!\n");

    machine->WriteRegister(2,readnum);
    AdvancePC();
    break;
}
```

9.6.5 Close ()

/* Close the file, we're done reading and writing to it. */
//void Close(OpenFileId id);

```
case SC_Close: {
    int fileId =machine->ReadRegister(4);
    OpenFile* openfile = currentThread->space->getFileId(fileId);
    if (openfile != NULL)
    {
        openfile->WriteBack();    // write file header back to DISK

        delete openfile;              // close file
        currentThread->space->releaseFileDescriptor(fileId);

        DEBUG('f',"File %d    closed succeed!\n",fileId);
        DEBUG('H',"File %d    closed succeed!\n",fileId);
        DEBUG('J',"File %d    closed succeed!\n",fileId);
    }
    else
        printf("Failded to Close File %d.\n",fileId);
    AdvancePC();
    break;
}
```

9.7 shell 与几个内部命令的实现

修改后的 shell.c 代码如下：

```
#include "syscall.h"

int
main()
{
        SpaceId newProc;
        OpenFileId input = ConsoleInput;
        OpenFileId output = ConsoleOutput;
        char prompt[7], ch, buffer[60];
        char Hbuffer[80];
        char Cbuffer[80];
        int i,j,k,h,m;
        char c;

        prompt[0] = 'N';
        prompt[1] = 'a';
        prompt[2] = 'c';
        prompt[3] = 'h';
        prompt[4] = 'o';
        prompt[5] = 's';
        prompt[6] = '$';
        prompt[7] = ':';

        while( 1 )
        {
           Write(prompt, 8, output);
           i = 0;
           k = 0;
           h = 0;
           m = 0;

      do {
            Read(&c, 1, input);

            buffer[i] = c;
            Hbuffer[h++] = buffer[i];

            k++;
```

```
    } while( buffer[i++] != '\n' );

          Hbuffer[--h] = '\0';


        for (m=0;j<80;j++)
        {
            Cbuffer[m] = buffer[m];
        }


    buffer[--i] = '.';
    buffer[i++] = '.';
    buffer[i++] = 'n';
    buffer[i++] = 'o';
    buffer[i++] = 'f';
    buffer[i++] = 'f';
    buffer[i] = '\0';

          if (k == 1)
             continue;

    if( h > 0 ) {
            newProc = Exec(buffer);

            if (newProc == -1)
                //Write("Invalid Command, Enter again.", 30, output);
                Write("Invalid Command, Enter again, or try \"help\"\n", 44, output);
            else if (newProc != 127)
            {
        Join(newProc);

          Write("Command \"", 9, output);
          Write(Hbuffer,k, output);
          Write("\" Execute Completely.\n", 22, output);

       } //    if (newProc == -1)
    }    //if( i > 0 ) {
          i = 0;
          //Yield();
      }    //    while( 1 )
}
```

系统调用 Exec()代码如下：

```
case SC_Exec:{
    //DEBUG('f',"Execute system call Exec()\n");
    //read argument (i.e. filename) of Exec(filename)
    char filename[128];
    int addr=machine->ReadRegister(4);
    int ii=0;
    //read filename from mainMemory
    do{
        machine->ReadMem(addr+ii,1,(int *)&filename[ii]);
    } while (filename[ii++]!='\0');

     //----------------------------------------------------------
     //
     // inner commands --begin
     //
     //----------------------------------------------------------
     //printf("Call Exec(%s)\n",filename);
     if (filename[0] == 'l' && filename[1] == 's')      //ls
     {
         printf("File(s) on Nachos DISK:\n");
         fileSystem->List();
         machine->WriteRegister(2,127);      //
         AdvancePC();
         break;
     }
     else if (filename[0] == 'r' && filename[1] == 'm')    //rm file
     {
         char fn[128];
         strncpy(fn,filename,strlen(filename) - 5);
         fn[strlen(filename) - 5] = '\0';
         fn[0] = ' ';
         fn[1] = ' ';
         char *file = EraseStrChar(fn,' ');
         if (file != NULL && strlen(file) >0)
         {
            fileSystem->Remove(file);
                machine->WriteRegister(2,127);
           }
           else
           {
```

```
            printf("Remove: invalid file name.\n");
            machine->WriteRegister(2,-1);
        }
        AdvancePC();
        break;
    } //rm
    else if (filename[0] == 'c' && filename[1] == 'a' && filename[2] == 't')    //cat file
    {
        char fn[128];
        strncpy(fn,filename,strlen(filename) - 5);
        fn[strlen(filename) - 5] = '\0';
        fn[0] = ' ';
        fn[1] = ' ';
        fn[2] = ' ';
         char *file = EraseStrChar(fn,' ');
        //printf("filename=%s, fn=%s, file=%s\n",filename,fn, file);
        if (file != NULL && strlen(file) >0)
        {
            Print(file);
               machine->WriteRegister(2,127);
         }
         else
         {
              printf("Cat: file not exists.\n");
              machine->WriteRegister(2,-1);
         }
         AdvancePC();
         break;
    }
    else if (filename[0] == 'c' && filename[1] == 'f')    //create a nachos file
    {   //create file
       char fn[128];
       strncpy(fn,filename,strlen(filename) - 5);
       fn[strlen(filename) - 5] = '\0';
       //printf("filename=%s, fn=%s\n",filename,fn);
       fn[0] = ' ';
       fn[1] = ' ';

       char *file = EraseStrChar(fn,' ');
       if (file != NULL && strlen(file) >0)
       {
            fileSystem->Create(file,0);
            machine->WriteRegister(2,127);
```

```
        }
        else
          {
              printf("Create: file already exists.\n");
              machine->WriteRegister(2,-1);
          }
          AdvancePC();
          break;
      }
        else if (filename[0] == 'c' && filename[1] == 'p')      //cp source dest
        {
            char fn[128];
            strncpy(fn,filename,strlen(filename) - 5);
            fn[strlen(filename) - 5] = '\0';
            fn[0] = '#';
            fn[1] = '#';
            fn[2] = '#';
            char source[128];
            char dest[128];
               int startPos = 3;
               int j = 0;
               int k = 0;
               for (int i = startPos; i < 128 /*, fn[i] != '\0'*/;i++)
                    if (fn[i] != ' ')
                         source[j++] = fn[i];
                    else
                         break;
               source[j] = '\0';
               j++;
               //printf("j = %d\n",j);

               for (int i = startPos + j; i < 128 /*,fn[i] != '\0'*/;i++)
                     if (fn[i] != ' ')
                         dest[k++] = fn[i];
                    else
                         break;
               dest[k] = '\0';

               if (source == NULL || strlen(source) <= 0)
               {
                    printf("cp: Source file not exists.\n");
                    machine->WriteRegister(2,-1);
                    AdvancePC();
```

```
                break;
            }
            if (dest != NULL && strlen(dest) > 0)
            {
                NAppend(source, dest);
                machine->WriteRegister(2,127);
            }
            else
            {
                printf("cp: Missing dest file.\n");
                machine->WriteRegister(2,-1);
            }
            AdvancePC();
            break;
        }
    //uap source(Unix) dest(nachos)
    else if ((filename[0] == 'u' && filename[1] == 'a' && filename[2] == 'p')
              || (filename[0] == 'u' && filename[1] == 'c' && filename[2] == 'p'))

    {
        char fn[128];
        strncpy(fn,filename,strlen(filename) - 5);
        fn[strlen(filename) - 5] = '\0';
        fn[0] = '#';
        fn[1] = '#';
        fn[2] = '#';
        fn[3] = '#';
        char source[128];
        char dest[128];
        int startPos = 4;
        int j = 0;
        int k = 0;
        for (int i = startPos; i < 128/*, fn[i] != '\0'*/;i++)
            if (fn[i] != ' ')
                source[j++] = fn[i];
            else
                break;
        source[j] = '\0';
        j++;

          for (int i = startPos + j; i < 128/*,fn[i] != '\0'*/;i++)
                if (fn[i] != ' ')
                    dest[k++] = fn[i];
```

```
            else
                break;
        dest[k] = '\0';

        if (source == NULL || strlen(source) <= 0)
        {
            printf("uap or ucp: Source file not exists.\n");
            machine->WriteRegister(2,-1);
            AdvancePC();
            break;
        }
        if (dest != NULL && strlen(dest) > 0)
        {
            if (filename[0] == 'u' && filename[1] == 'c' && filename[2] == 'p')
                Append(source, dest,0); //append dest file at the end of source file
            else
                Copy(source, dest);        //ucp
            machine->WriteRegister(2,127);
        }
        else
        {
            printf("uap or ucp: Missing dest file.\n");
            machine->WriteRegister(2,-1);
        }
    AdvancePC();
    break;
}
//nap source dest
else if (filename[0] == 'n' && filename[1] == 'a' && filename[2] == 'p')
{
    char fn[128];
    strncpy(fn,filename,strlen(filename) - 5);
    fn[strlen(filename) - 5] = '\0';
    fn[0] = '#';
    fn[1] = '#';
    fn[2] = '#';
    fn[3] = '#';
    //char *file = EraseStrSpace(fn, ' ');
    char source[128];
    char dest[128];
    int j = 0;
    int k = 0;
    int startPos = 4;
```

```
            for (int i = startPos; i < 128/*, fn[i] != '\0'*/;i++)
                if (fn[i] != ' ')
                    source[j++] = fn[i];
                else
                    break;
            source[j] = '\0';
            j++;
            //printf("j = %d\n",j);

            for (int i = startPos + j; i < 128/*,fn[i] != '\0'*/;i++)
                if (fn[i] != ' ')
                   dest[k++] = fn[i];
                else
                   break;
                    dest[k] = '\0';

              if (source == NULL || strlen(source) <= 0)
              {
                   printf("nap: Source file not exists.\n");
                   machine->WriteRegister(2,-1);
                   AdvancePC();
                   break;
              }
              if (dest != NULL && strlen(dest) > 0)
              {
                   NAppend(source, dest);
                   machine->WriteRegister(2,127);
              }
              else
              {
                   printf("ap: Missing dest file.\n");
                   machine->WriteRegister(2,-1);
              }
              AdvancePC();
              break;
        }
        //rename source dest
        else if (filename[0] == 'r' && filename[1] == 'e' && filename[2] == 'n')
        {
           char fn[128];
           strncpy(fn,filename,strlen(filename) - 5);
           fn[strlen(filename) - 5] = '\0';
           fn[0] = '#';
```

```
fn[1] = '#';
fn[2] = '#';
fn[3] = '#';
char source[128];
char dest[128];
int j = 0;
int k = 0;
int startPos = 4;
for (int i = startPos; i < 128/*, fn[i] != '\0'*/;i++)
    if (fn[i] != ' ')
        source[j++] = fn[i];
    else
        break;
source[j] = '\0';
j++;
//printf("j = %d\n",j);

for (int i = startPos + j; i < 128/*,fn[i] != '\0'*/;i++)
     if (fn[i] != ' ')
        dest[k++] = fn[i];
    else
        break;
dest[k] = '\0';

if (source == NULL || strlen(source) <= 0)
{
    printf("rename: Source file not exists.\n");
    machine->WriteRegister(2,-1);
    AdvancePC();
    break;
}
if (dest != NULL && strlen(dest) > 0)
{
    fileSystem->Rename(source, dest);
    machine->WriteRegister(2,127);
}
else
{
    printf("rename: Missing dest file.\n");
            machine->WriteRegister(2,-1);
    }
    AdvancePC();
    break;
```

```
    }
    else if (strstr(filename,"format") != NULL)      //format
    {
        printf("strstr(filename,\"format\"=%s \n",strstr(filename,"format"));
        printf("WARNING: Format Nachos DISK will erase all the data on it.\n");
        printf("Do you want to continue (y/n)? ");
        char ch;
        while (true)
        {
            ch = getchar();
            if (toupper(ch) == 'Y' || toupper(ch) == 'N')
                break;
       } //while
       if (toupper(ch) == 'N')
       {
           machine->WriteRegister(2,127);      //
           AdvancePC();
           break;
       }

        printf("Format the DISK and create a file system on it.\n");
        fileSystem->FormatDisk(true);
        machine->WriteRegister(2,127);      //
        AdvancePC();
        break;
     }
    else if (strstr(filename,"fdisk") != NULL)      //fdisk
    {
        fileSystem->Print();
        machine->WriteRegister(2,127);      //
        AdvancePC();
        break;
    }
    else if (strstr(filename,"perf") != NULL)      //Performance
    {
        PerformanceTest();
        machine->WriteRegister(2,127);      //
        AdvancePC();
        break;
    }
    else if (filename[0] == 'p' && filename[1] == 's')      //ps
    {
        scheduler->PrintThreads();
```

```
                machine->WriteRegister(2,127);      //
                AdvancePC();
                break;
            }
            else if (strstr(filename,"help") != NULL)      //fdisk
            {
                printf("Commands and Usage:\n");
                printf("ls                          : list files on DISK.\n");
                printf("fdisk                       : display DISK information.\n");
                printf("format                      : format DISK with creating a file system on
it.\n");
                printf("performence                 : test DISK performence.\n");
                printf("cf    name                  : create a file \"name\" on DISK.\n");
                printf("cp    source dest           : copy Nachos file \"source\" to \"dest\".\n");
                printf("nap   source dest           : Append Nachos file \"source\" to
\"dest\"\n");
                printf("ucp source dest             : Copy Unix file \"source\" to Nachos file
\"dest\"\n");
                printf("uap source dest             : Append Unix file \"source\" to Nachos file
\"dest\"\n");
                printf("cat name                    : print content of file \"name\".\n");
                printf("rm    name                  : remove file \"name\".\n");
                printf("rename source dest: Rename Nachos file \"source\" to
\"dest\".\n");
                printf("ps                          : display the system threads.\n");
                //------------------------------------------------------------

                machine->WriteRegister(2,127);      //
                AdvancePC();
                break;
            }
            else    //check if the parameter of exec(file), i.e file, is valid
            {
                if (strchr(filename,' ') != NULL || strstr(filename,".noff") == NULL)
                    //not an inner command, and not a valid Nachos executable, then return
                {
                    machine->WriteRegister(2,-1);
                    AdvancePC();
                    break;
                }
            }
            //-------------------------------------------------------
            //
```

```
                // inner commands --end
                //
                //-------------------------------------------------------
                //-------------------------------------------------------
                //
                // loading an executable and execute it
                //
                //-------------------------------------------------------

                //call open() in FILESYS, see filesys.h
                OpenFile *executable = fileSystem->Open(filename);
                if (executable == NULL) {
                      //printf("\nUnable to open file %s\n", filename);
                      DEBUG('f',"\nUnable to open file %s\n", filename);
                      machine->WriteRegister(2,-1);
                      AdvancePC();
                      break;
                      //return;
                }

                 //new address space
               space = new AddrSpace(executable);
               delete executable;              // close file

               DEBUG('H',"Execute system call Exec(\"%s\"), it's SpaceId(pid) = %d
\n",filename,space->getSpaceID());
               //new and fork thread
               char *forkedThreadName = filename;

               //-----------------------------------------------
               char *fname=strrchr(filename,'/');
               if (fname != NULL)    // there exists "/" in filename
                   forkedThreadName=strtok(fname,"/");
               //-----------------------------------------------
               thread = new Thread(forkedThreadName);
               //printf("exec -- new thread pid =%d\n",space->getSpaceID());
               thread->Fork(StartProcess, space->getSpaceID());
               thread->space = space;
               printf("user process \"%s(%d)\" map to kernel thread \" %s
\"\n",filename,space->getSpaceID(),forkedThreadName);

                //return spaceID
                machine->WriteRegister(2,space->getSpaceID());
```

```
            //printf("Exec()--space->getSpaceID()=%d\n",space->getSpaceID());

           //=========================================================
            //run the new thread,
            //otherwise, this process will not execute in order to release its memory,
            //thread "main" may continue to create new processes,
            //and will not have enough memory for new processes

            currentThread->Yield();

            //but introduce another problem:
            // when Joiner waits for a Joinee, the joinee maybe finish before Joiner call Join(),
                  //    but Joinee go to sleep after calling Finish()
                  //
                 //============================================================
                  //advance PC
                  AdvancePC();
                  break;
            }
```

几个需要添加的函数如下：

FileSystem::List()调用了 directory->List()，其代码修改如下：

```
extern char *setStrLength(char *str, int len);
extern char *numFormat(int num);

void
Directory::List()
{
    FileHeader *hdr = new FileHeader;

    int size = 0;
    printf("============================================================\n");
    printf("Name            Size    Sectors    SectorList\n");
    printf("------------------------------------------------------------\n");
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse)
        {
            hdr->FetchFrom(table[i].sector);
            size += hdr->getFileSize();
```

```
            char *fileName = table[i].name;
        int fileSize = hdr->getFileSize();
        int fileSectors = hdr->getNumSectors();

        printf("%s    ", setStrLength(fileName, 9));

        printf("%s    ", setStrLength(numFormat(fileSize),7));
        printf("%s", setStrLength(numFormat(fileSectors),6));
        printf("%s\n", hdr->getDataSectors());
          }
    printf("----------------------------------------------------------\n");

    // size:not include the file header
    //printf("Available Disk Space: %s bytes",numFormat(32*32*128 - size));
    //printf(" (%s KB)\n\n",numFormat((32*32*128 - size)/1024));

    BitMap *freeMap =    fileSystem->getBitMap();
    int freeSize = freeMap->NumClear() * 128;
    printf("Available Disk Space: %s bytes",numFormat(freeSize));
    printf(" (%s KB)\n\n",numFormat(freeSize/1024));

    delete hdr;
}
```

涉及的几个辅助函数如下：

```
//-----------------------------------------------------------------------
// delete all char 'c' in str
//
//-----------------------------------------------------------------------
char buf[128];
char *EraseStrChar(char *str,char c)
{
    int i = 0;
    //printf("str=%s\n",str);
    while (*str != '\0')
    {
       if (*str != c)
       {
            buf[i] = *str;
            str++;
            i++;
```

```
        }    //if
        else
            str++;
    } //while
    buf[i] = '\0';
    //printf("buf=%s\n",buf);
    return buf;
}
```

```
//-----------------------------------------------------------------
//
//    Convert integer to str
//
//-----------------------------------------------------------------
char istr[10] = "";
char* itostr(int num)
{
    int n;
    char ns[10];

    int k = 0;
    while (true)
    {
        n = num % 10;
        ns[k] =    0x30 + n;
        k++;
        num = num /10;
        if (num    == 0)
            break;
    }

    for (int i = 0; i < k; i++)
        istr[i] = ns[k-i-1] ;

    //printf("str=%s\n",str);
    return istr;
}
```

```
//-------------------------------------------------------------
//
//    set a string to fix length, with blank append at the end of it,
//    or truncate it to the fix length
//
//
//      parametors:
//                str, len
//
//-------------------------------------------------------------
char strSpace[10];
char *setStrLength(char *str, int len)
{
     //printf("str = %s\n",str);
     int strLength = strlen(str);

     if (strLength >= len)
     {
          for (int i = 0; i < len ; i++, str++)
               strSpace[i] = *str;
     }
     else
     {
        for (int i = 0; i < strLength ; i++, str++)
               strSpace[i] = *str;

        for (int i = strLength; i < len ; i++)
               strSpace[i] = ' ';
     }
     strSpace[len] = '\0';
     //printf("strSpace = %s, len=%d\n",strSpace,strlen(strSpace));
     return strSpace;
}
```

```
bool FileSystem::Rename(char *source, char *dest)
{
        bool success;
        Directory *directory = new Directory(NumDirEntries);
        directory->FetchFrom(directoryFile);

        success = directory->Rename(source,dest);
        if (success)
        {
              directory->WriteBack(directoryFile);
              printf("Rename success.\n");
        }
        else
             printf("Rename: file %s not exists.\n",source);

        delete directory;
        return success;
}
```

```
//----------------------------------------------------------------
//
// divide an integer with ",", such as int (1,234,567) to char * (1,234,567)
//
//----------------------------------------------------------------
char numStr[10];
char *numFormat(int num)
{
      numStr[0] = '\0';
      char nstr[10]="";
      int k = 0;
      while (true)
      {
           nstr[k++] = num % 10 + 0x30;
           num = num /10;
           if (num == 0 )
                break;
      }
      nstr[k] = '\0';

      int j = k-1;
```

```
    if (strlen(nstr)>=4 && strlen(nstr)<=6)
        j = k;
    else    if (strlen(nstr)>=7 && strlen(nstr)<=9)
        j = k + 1;
    else if (strlen(nstr)>=10 && strlen(nstr)<=12)
        j = k + 2;


    numStr[j+1] = '\0';
    for (int i=0;i<k;i++)
    {
        if (i>0 && i%3==0)
        {
            numStr[j--] = ',';
            numStr[j--] = nstr[i];
        }
        else
            numStr[j--] = nstr[i];
    }
    //printf("numStr= %s\n",numStr);
    return numStr;
}
//---------------------------------------------------------
```

```
void FileSystem::FormatDisk(bool format)
{
    if (!format)
        return;

    BitMap *freeMap = new BitMap(NumSectors);
//NumSectors=32*32=1024 sectors for DISK
    Directory *directory = new Directory(NumDirEntries);      //NumDirEntries=10 entrys for
(root) directory
    FileHeader *mapHdr = new FileHeader;
    FileHeader *dirHdr = new FileHeader;

    DEBUG('f', "Formatting the file system.\n");

    // First, allocate space for FileHeaders for the directory and bitmap
    // (make sure no one else grabs these!)
    freeMap->Mark(FreeMapSector);
```

```
    freeMap->Mark(DirectorySector);
    // Second, allocate space for the data blocks containing the contents
    // of the directory and bitmap files.    There better be enough space!

    ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));     //FreeMapFileSize=128,
allocates 1 sector for free bitmap file
    ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize)); //DirectoryFileSize=200,
allocates 2 sectors for directory file

    // Flush the bitmap and directory FileHeaders back to disk
    // We need to do this before we can "Open" the file, since open
    // reads the file header off of disk (and currently the disk has garbage
    // on it!).

    DEBUG('f', "Writing headers back to disk.\n");
    mapHdr->WriteBack(FreeMapSector);           //FreeMapSector=0,      write freemap
header to sector 0
    dirHdr->WriteBack(DirectorySector);       //DirectorySector=1, write directory header to
sector 1

    // OK to open the bitmap and directory files now
    // The file system operations assume these two files are left open
    // while Nachos is running.

    freeMapFile = new OpenFile(FreeMapSector);
    directoryFile = new OpenFile(DirectorySector);

    // Once we have the files "open", we can write the initial version
    // of each file back to disk.    The directory at this point is completely
    // empty; but the bitmap has been changed to reflect the fact that
    // sectors on the disk have been allocated for the file headers and
    // to hold the file data for the directory and bitmap.

    DEBUG('f', "Writing bitmap and directory back to disk.\n");
    freeMap->WriteBack(freeMapFile);    // flush changes to disk
    directory->WriteBack(directoryFile);

    //print each bit in freeMap
    //freeMap->PrintinBit();       //added by han

    freeMap->Print();
    directory->Print();
```

```
    delete freeMap;
    delete directory;
    delete mapHdr;
    delete dirHdr;
}
```

参考资料

[1] Peiyi Tang, Ron Addie, nachos_introduction.pdf，University of Southern Queensland, 2002
[2] Peiyi Tang, Ron Addie, nachos_study_book.pdf，University of Southern Queensland, 2002
[3] Nachos-3.4（C++）源代码
[4] A road map through nachos.pdf
[5] gdb 使用指南
[6] MIPS 指令集
[7] i386 指令集

附录 1 提交材料

1．设计结束后，每个同学提交的设计材料主要包括如下两部分：

（1）课程设计报告

注 1：提交 Word 文档格式

注 2：包括源代码分析，以及实验方案设计、实现、调试、测试等内容，同时包括运行界面与测试界面。具体请参阅附录 2：设计报告与要求。

（2）完整的设计源代码及测试源代码

注 1：对自己编写的代码要有注释，对已有源代码在阅读分析的过程中也要加上必要的注释说明。

注 2：包括代码运行环境：如 Ubuntu： nachos-3.4.tar.gz

注 3：在相应的目录下运行 make 可正确编译、执行，结果与报告中对应一致

2．提交材料打包及命名

将设计报告、设计源代码（包括测试代码）压缩成一个包提交。

压缩包命名：学号 姓名 报告名称

3．材料提交方式

以班级为单位提交。具体问题与助教联系。

4．材料提交日期

暂定于第 13 周周日 23:59:59 之前。

附录 2 设计报告要求

设计结束后，设计报告与源代码要提交给教务存档，因此，请大家认真对待，注意报告格式与内容。

1、设计报告内容（word 文档）

- 封面（题目、姓名、学号、班级等）
- 摘要（工作总结、特色等）
- 目录（自动生成）
- 正文（按实验分章节）

代码分析：模块功能、涉及到的类、类关系、数据结构及关键代码等；

任务要求，设计任务要求；

设计：详细的设计方案，相关的数据结构、算法描述，可采用伪代码等形式化描述

实现：修改哪些类、如何修改、为什么修改等；

测试：测试用例，测试结果及结果分析，测试运行界面等；

调试：调试方法，遇到的问题及解决方案等；

结论与展望：完成的主要工作、收获、进一步的工作，建议、体会、心得等；

- 参考文献