目录

[实验一、Nachos 系统的安装与调试 4](#_Toc72994578)

[一、描述问题： 4](#_Toc72994579)

[二、阐述基本方法： 4](#_Toc72994580)

[1.安装nachos 4](#_Toc72994581)

[2.安装 gcc MIPS 交叉编译器 4](#_Toc72994582)

[3.测试 gcc MIPS 交叉编译器 4](#_Toc72994583)

[4.运行nachos 5](#_Toc72994584)

[三、源代码+注释: 5](#_Toc72994585)

[Main.cc 5](#_Toc72994586)

[system.cc 6](#_Toc72994587)

[Initialize() 6](#_Toc72994588)

[thread.cc 7](#_Toc72994589)

[1.fork() 7](#_Toc72994590)

[2.StackAllocate 8](#_Toc72994591)

[3.Sleep 8](#_Toc72994592)

[4. Yield 9](#_Toc72994593)

[5.Finish 9](#_Toc72994594)

[threadtest.cc 10](#_Toc72994595)

[1.ThreadTest 10](#_Toc72994596)

[2.SimpleThread 10](#_Toc72994597)

[SWITCH.s 11](#_Toc72994598)

[四、实验中遇到的问题: 11](#_Toc72994599)

[实验三、利用信号量实现线程同步 15](#_Toc72994600)

[一、描述问题： 15](#_Toc72994601)

[二、阐述基本方法： 15](#_Toc72994602)

[1. ProdCons 15](#_Toc72994603)

[2.producer 16](#_Toc72994604)

[3.consumer 16](#_Toc72994605)

[三、源代码+注释: 17](#_Toc72994606)

[Main.cc 17](#_Toc72994607)

[ring.cc 18](#_Toc72994608)

[1.ring 18](#_Toc72994609)

[2.put 18](#_Toc72994610)

[3.get 19](#_Toc72994611)

[四、实验中遇到的问题: 19](#_Toc72994612)

[五、总结与收获 19](#_Toc72994613)

[实验四、Nachos 的文件系统 20](#_Toc72994614)

[一、描述问题： 20](#_Toc72994615)

[二、阐述基本方法： 21](#_Toc72994616)

[三、源代码+注释: 22](#_Toc72994617)

[Disk.cc 22](#_Toc72994618)

[Disk 24](#_Toc72994619)

[ReadRequest 24](#_Toc72994620)

[WriteRequest 25](#_Toc72994621)

[TimeToSeek 26](#_Toc72994622)

[ComputeLatency 26](#_Toc72994623)

[SynchDisk.cc 27](#_Toc72994624)

[SynchDisk 27](#_Toc72994625)

[ReadSector 27](#_Toc72994626)

[WriteSector 27](#_Toc72994627)

[OpenFile.cc 28](#_Toc72994628)

[OpenFile 28](#_Toc72994629)

[ReadAt 28](#_Toc72994630)

[Read 29](#_Toc72994631)

[WriteAt 29](#_Toc72994632)

[Directory.cc 31](#_Toc72994633)

[DirectoryEntry 31](#_Toc72994634)

[FetchFrom 31](#_Toc72994635)

[WriteBack 31](#_Toc72994636)

[FindIndex 32](#_Toc72994637)

[Find 32](#_Toc72994638)

[Add 32](#_Toc72994639)

[BitMap 33](#_Toc72994640)

[FileHeader 33](#_Toc72994641)

[Allocate 33](#_Toc72994642)

[Deallocate 34](#_Toc72994643)

[四、实验过程 34](#_Toc72994644)

[1.硬盘格式化 34](#_Toc72994645)

[2.复制一个文件到硬盘 36](#_Toc72994646)

[复制另一个文件到硬盘 38](#_Toc72994647)

[删除 Nachos 硬盘上的文件 39](#_Toc72994648)

[打开 Nachos 文件的过程 40](#_Toc72994649)

[五、实验中遇到的问题 41](#_Toc72994650)

[实验五、扩展 Nachos 的文件系统 42](#_Toc72994651)

[一、 描述问题： 42](#_Toc72994652)

[二、阐述基本方法： 43](#_Toc72994653)

[三、 源代码+注释 43](#_Toc72994654)

[1.nachos–ap 与 nachos–hap 命令的实现 43](#_Toc72994655)

[四、实验过程 50](#_Toc72994656)

[实验六、Nachos 用户程序与系统调用 57](#_Toc72994657)

[一、描述问题： 57](#_Toc72994658)

[二、阐述基本方法： 58](#_Toc72994659)

[三、源代码+注释: 58](#_Toc72994660)

[noff.h 58](#_Toc72994661)

[Segment 59](#_Toc72994662)

[noffHeader 59](#_Toc72994663)

[halt.c 59](#_Toc72994664)

[sort.c 60](#_Toc72994665)

[shell.c 60](#_Toc72994666)

[syscall.h 61](#_Toc72994667)

[addrspace.cc 64](#_Toc72994668)

[AddrSpace 64](#_Toc72994669)

[InitRegisters 66](#_Toc72994670)

[translate.cc 67](#_Toc72994671)

[progtest.cc 67](#_Toc72994672)

[StartProcess 68](#_Toc72994673)

[Machine.cc 68](#_Toc72994674)

[Run 68](#_Toc72994675)

[Translate 69](#_Toc72994676)

[scheduler.cc 71](#_Toc72994677)

[Run 71](#_Toc72994678)

[四、实验过程 73](#_Toc72994679)

[nachos应用程序的编译 73](#_Toc72994680)

[Nachos 应用程序与可执行程序 73](#_Toc72994681)

[Nachos 可执行程序格式 73](#_Toc72994682)

[页表的系统转储 73](#_Toc72994683)

[实验七、地址空间的扩展 75](#_Toc72994684)

[一、描述问题： 75](#_Toc72994685)

[二、阐述基本方法： 75](#_Toc72994686)

[三、源代码+注释: 76](#_Toc72994687)

[AddrSpace.cpp 76](#_Toc72994688)

[AddrSpace 76](#_Toc72994689)

[~AddrSpace 78](#_Toc72994690)

[实验八、系统调用 Exec()与 Exit() 78](#_Toc72994691)

[一、描述问题： 78](#_Toc72994692)

[二、阐述基本方法： 78](#_Toc72994693)

[三、源代码+注释: 79](#_Toc72994694)

[Exec（）： 79](#_Toc72994695)

[Join（）与Exit（）： 83](#_Toc72994696)

[Yield（） 90](#_Toc72994697)

[基于FILESYS\_STUB实现文件的有关系统调用 90](#_Toc72994698)

[Create 91](#_Toc72994699)

[Open 91](#_Toc72994700)

[Write 92](#_Toc72994701)

[Read 92](#_Toc72994702)

[Close 93](#_Toc72994703)

[四、实验过程 93](#_Toc72994704)

[1. 对Exec（）、Exit（）、Yield（）、Join（）进行综合测试 93](#_Toc72994705)

[2. 基于FILESYS\_STUB实现文件的有关系统调用 95](#_Toc72994706)

# 实验一、Nachos 系统的安装与调试

## 一、描述问题：

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;

## 二、阐述基本方法：

### 1.安装nachos

(1)进入用户工作目录

(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,OS 。

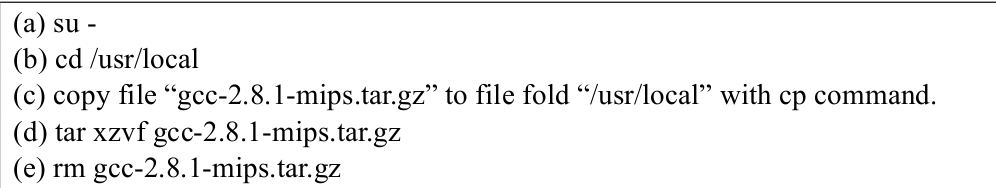
(3) 进入目录 OS

(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)删除目录下 OS 下的压缩包 Nachos-3.4.tar.gz

### 2.安装 gcc MIPS 交叉编译器



### 3.测试 gcc MIPS 交叉编译器

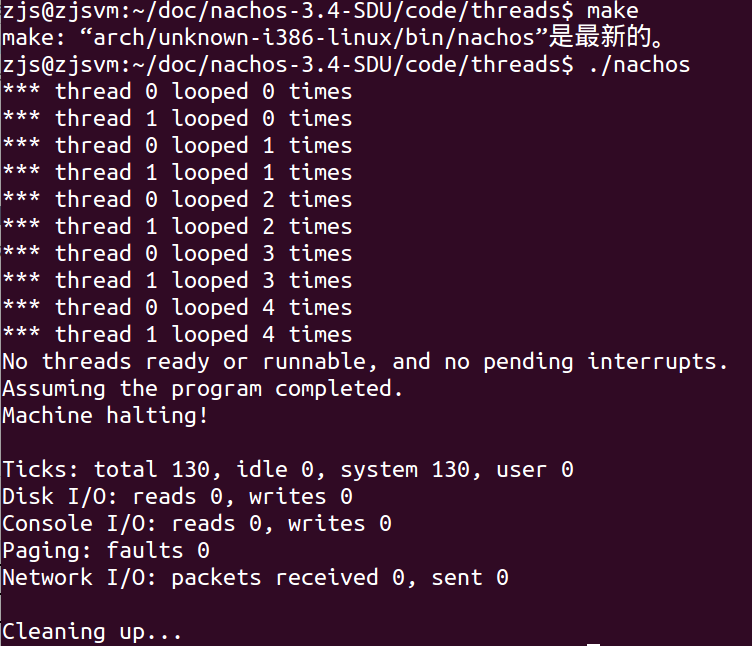
(1) 删除../test/arch/unknown-i386-linux/depends 目录下的所有文件;

(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 make,输出如实验指导书所示的编译信息。

### 4.运行nachos



## 三、源代码+注释:

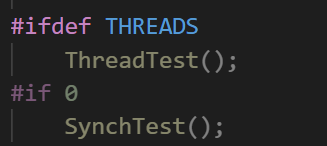
### Main.cc

Main函数中，可以看到在首先调用的 Initialize()进行初始化操作，并且在这个函数中定义了主线程。

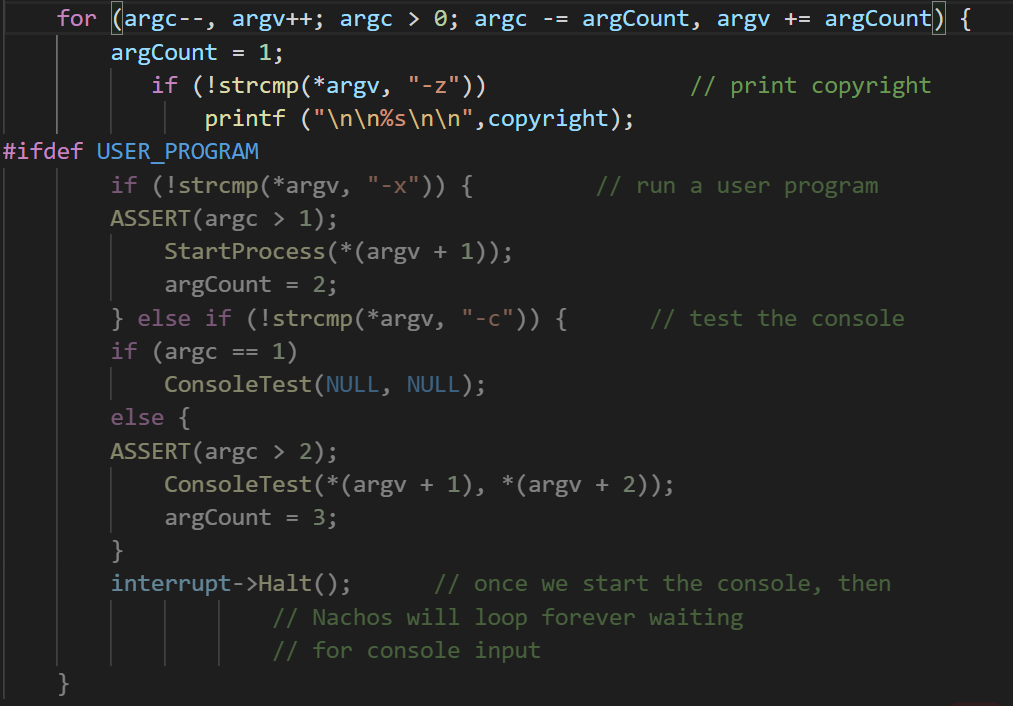
currentThread = new Thread("main");

    currentThread->setStatus(RUNNING);

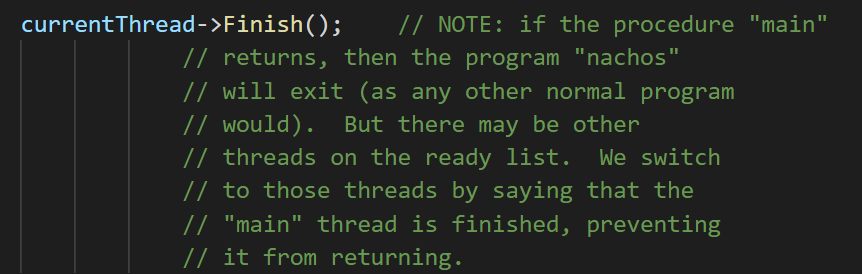
然后根据是否定义 THREADS 宏执行 ThreadTest()。



再根据宏定义对命令行参数进行处理。



最后执行 finish()。实为销毁线程。如果是main执行finish，则退出nachos。



### system.cc

#### Initialize()

根据命令行参数-d 设置调试功能,根据命令行参数-rs 设置轮转调度功能。

   for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {

        argCount = 1;

        if (!strcmp(\*argv, "-d")) {

           if (argc == 1)

            debugArgs = "+";    // turn on all debug flags

           else {

                debugArgs = \*(argv + 1);

                argCount = 2;

            }

        } else if (!strcmp(\*argv, "-rs")) {

        ASSERT(argc > 1);

        RandomInit(atoi(\*(argv + 1)));  // initialize pseudo-random

                        // number generator

        randomYield = TRUE;

        argCount = 2;

    }

进行初始化

   DebugInit(debugArgs);           // initialize DEBUG messages

    stats = new Statistics();           // collect statistics

    interrupt = new Interrupt;          // start up interrupt handling

    scheduler = new Scheduler();        // initialize the ready queue

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

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

Statistics() 初始化系统状态,Interrupt()初始化中断控制器,Scheduler()初始化调度器, Timer()初始化定时器。

设置主线程

    currentThread = new Thread("main");

    currentThread->setStatus(RUNNING);

开中断，设置Ctrl+C中断。

 interrupt->Enable();

    CallOnUserAbort(Cleanup);           // if user hits ctl-C

### thread.cc

#### 1.fork()

void Thread::Fork(VoidFunctionPtr func, \_int arg)

{

#ifdef HOST\_ALPHA

    DEBUG('t', "Forking thread \"%s\" with func = 0x%lx, arg = %ld\n",

      name, (long) func, arg);

#else

    DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",

      name, (int) func, arg);

#endif

    StackAllocate(func, arg);

    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts

                    // are disabled!

    (void) interrupt->SetLevel(oldLevel);

}

他分配了一个栈，并且完成了初始化栈，将线程放到了就绪队列里面。

#### 2.StackAllocate

machineState[PCState] = (\_int) ThreadRoot;

machineState[StartupPCState] = (\_int) InterruptEnable;

machineState[InitialPCState] = (\_int) func;

machineState[InitialArgState] = arg;

machineState[WhenDonePCState] = (\_int) ThreadFinish;

初始化栈。

#### 3.Sleep

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

}

当前线程被阻塞，等待唤醒，并将其放回到就绪队列中，以便可以对其进行重新调度。在就绪队列中找到下一个 READY 状态的线程,切换上下文并执行。如果没有则调用 Idle()处理中断，如中断队列中只有定时器中断,则不会再有线程处于 READY 状态，Nachos 终止。

#### 4. Yield

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()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))。

该函数将当前线程的状态设置为可运行态，将下一个可运行态线程调度出来运行。该函数与 Sleep 函数的区别是，Sleep 函数将线程设置为了阻塞态，需要唤醒。而 Yield 函数只是将下一个线程调度出来运行，之前运行的线程仍然属于可运行态。

#### 5.Finish

void

Thread::Finish ()

{

    (void) interrupt->SetLevel(IntOff);

    ASSERT(this == currentThread);

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

    threadToBeDestroyed = currentThread;//当前线程准备删除

    Sleep();                    // invokes SWITCH

    // not reached

}

用于结束当前线程，首先需要将该线程阻塞并进行线程切换，之后再由其它线程将该线程删除。

### threadtest.cc

创建两个线程，并进行上下文切换，通过调用Thread :: Yield在彼此之间来回切换。

#### 1.ThreadTest

Void ThreadTest()

{

    DEBUG('t', "Entering SimpleTest");

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

    t->Fork(SimpleThread, 1);

    SimpleThread(0);

}

这个函数创建了新的线程对象forked thread，调用Fork方法，绑定SimpleThread这个函数，主进程执行函数SimpleThread。

#### 2.SimpleThread

Void SimpleThread(\_int which)

{

    int num;

    for (num = 0; num < 5; num++) {

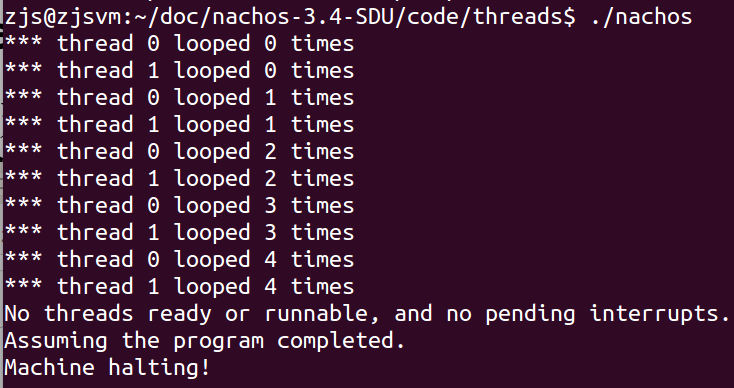
    printf("\*\*\* thread %d looped %d times\n", (int) which, num);

        currentThread->Yield();

    }

}

打印出线程n的循环次数i，然后调用yield，继续拧线程切换，最后的结果如下图，两个线程交替进行，全部完成后，没有要执行的线程了，就退出nachos。



### SWITCH.s

将当前正在执行的状态保存，把要替换的线程的信息放到寄存器中，开始运行新的线程。跳转，跳转到ra也就是将要执行的代码地址。

## 四、实验中遇到的问题:

1. 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,GCCDIR = /usr/local/mips/bin/decstation-ultrix-

1. Nachos 的命令行参数及其处理

在nachos中，提供了命令行参数，它们的用法在main函数前面的注释里面。

// Usage: nachos -d <debugflags> -rs <random seed #>

//      -s -x <nachos file> -c <consoleIn> <consoleOut>

//      -f -cp <unix file> <nachos file>

//      -p <nachos file> -r <nachos file> -l -D -t

//              -n <network reliability> -e <network orderability>

//              -m <machine id>

//              -o <other machine id>

//              -z

//

//    -d causes certain debugging messages to be printed (cf. utility.h)

//    -rs causes Yield to occur at random (but repeatable) spots

//    -z prints the copyright message

//

//  USER\_PROGRAM

//    -s causes user programs to be executed in single-step mode

//    -x runs a user program

//    -c tests the console

在main函数中，通过循环，处理对应的命令行参数。

for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {

        argCount = 1;

           if (!strcmp(\*argv, "-z"))               // print copyright

               printf ("\n\n%s\n\n",copyright);

#ifdef USER\_PROGRAM

        if (!strcmp(\*argv, "-x")) {         // run a user program

        ASSERT(argc > 1);

            StartProcess(\*(argv + 1));

            argCount = 2;

        } else if (!strcmp(\*argv, "-c")) {      // test the console

        if (argc == 1)

            ConsoleTest(NULL, NULL);

        else {

        ASSERT(argc > 2);

            ConsoleTest(\*(argv + 1), \*(argv + 2));

            argCount = 3;

        }

        interrupt->Halt();      // once we start the console, then

                    // Nachos will loop forever waiting

                    // for console input

    }

#endif // USER\_PROGRAM

#ifdef FILESYS

    if (!strcmp(\*argv, "-cp")) {        // copy from UNIX to Nachos

        ASSERT(argc > 2);

        Copy(\*(argv + 1), \*(argv + 2));

        argCount = 3;

    } else if (!strcmp(\*argv, "-p")) {  // print a Nachos file

        ASSERT(argc > 1);

        Print(\*(argv + 1));

        argCount = 2;

    } else if (!strcmp(\*argv, "-r")) {  // remove Nachos file

        ASSERT(argc > 1);

        fileSystem->Remove(\*(argv + 1));

        argCount = 2;

    } else if (!strcmp(\*argv, "-l")) {  // list Nachos directory

            fileSystem->List();

    } else if (!strcmp(\*argv, "-D")) {  // print entire filesystem

            fileSystem->Print();

    } else if (!strcmp(\*argv, "-t")) {  // performance test

            PerformanceTest();

    }

#endif // FILESYS

#ifdef NETWORK

        if (!strcmp(\*argv, "-o")) {

        ASSERT(argc > 1);

            Delay(2);               // delay for 2 seconds

                        // to give the user time to

                        // start up another nachos

            MailTest(atoi(\*(argv + 1)));

            argCount = 2;

        }

#endif // NETWORK

    }

1. 主线程(main)是如何创建的?

在Initialize函数中。

    // We didn't explicitly allocate the current thread we are running in.

    // But if it ever tries to give up the CPU, we better have a Thread

    // object to save its state.

    currentThread = new Thread("main");

    currentThread->setStatus(RUNNING);

4、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。

使用GDB调试，可以直接查看函数的地址。

1. InterruptEnable()



1. SimpleThread()



1. ThreadFinish()



1. ThreadRoot()



5、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

通过阅读代码，我发现，在Initialize函数中，有这样一句话。

    // We didn't explicitly allocate the current thread we are running in.

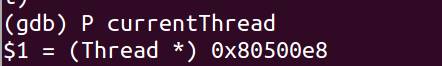
    // But if it ever tries to give up the CPU, we better have a Thread

    // object to save its state.

    currentThread = new Thread("main");

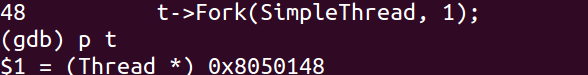
    currentThread->setStatus(RUNNING);

他创建了main线程，并且让该线程处于”RUNNING”的状态。因此通过gdb进入这个函数，然后查看他的地址。

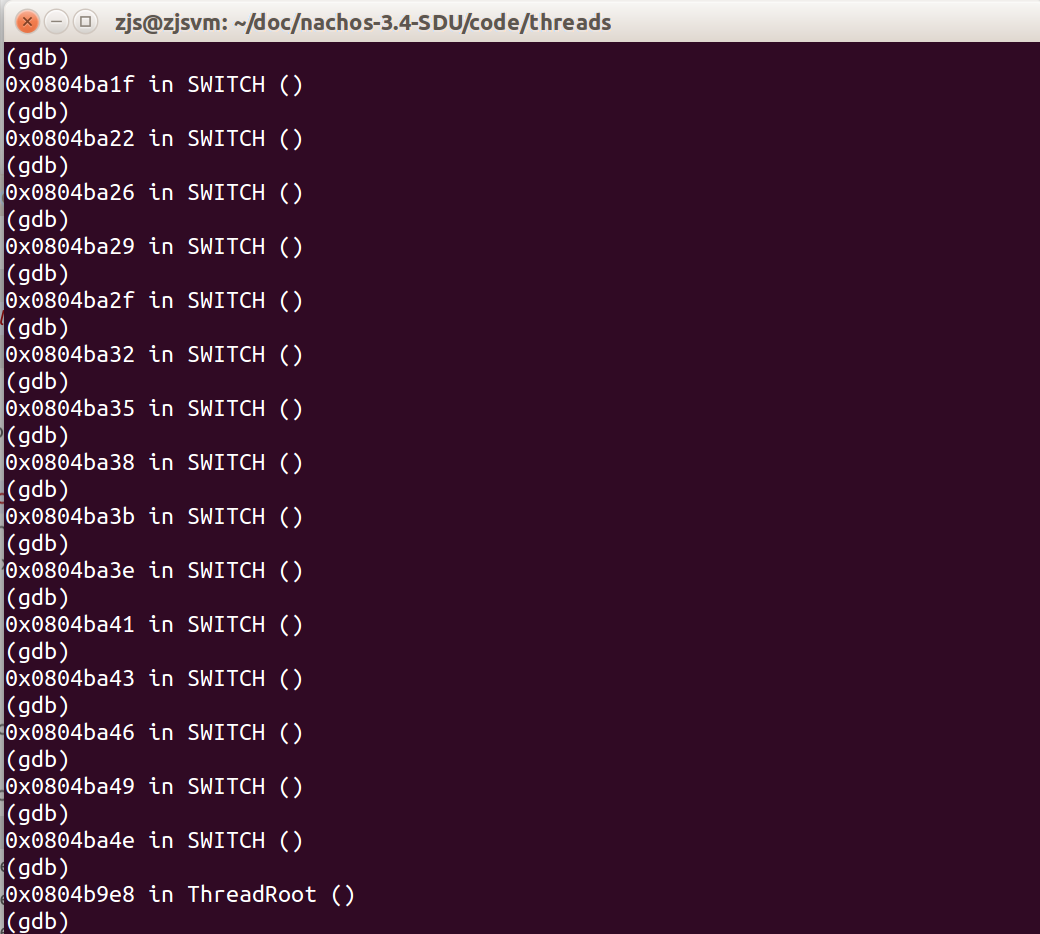


ii. the forked thread created by the main thread

通过查看代码发现，主线程创建线程在ThreadTest函数里面。因此，使用GDB打上断点，然后查看t的地址，就能找到这个的地址了。

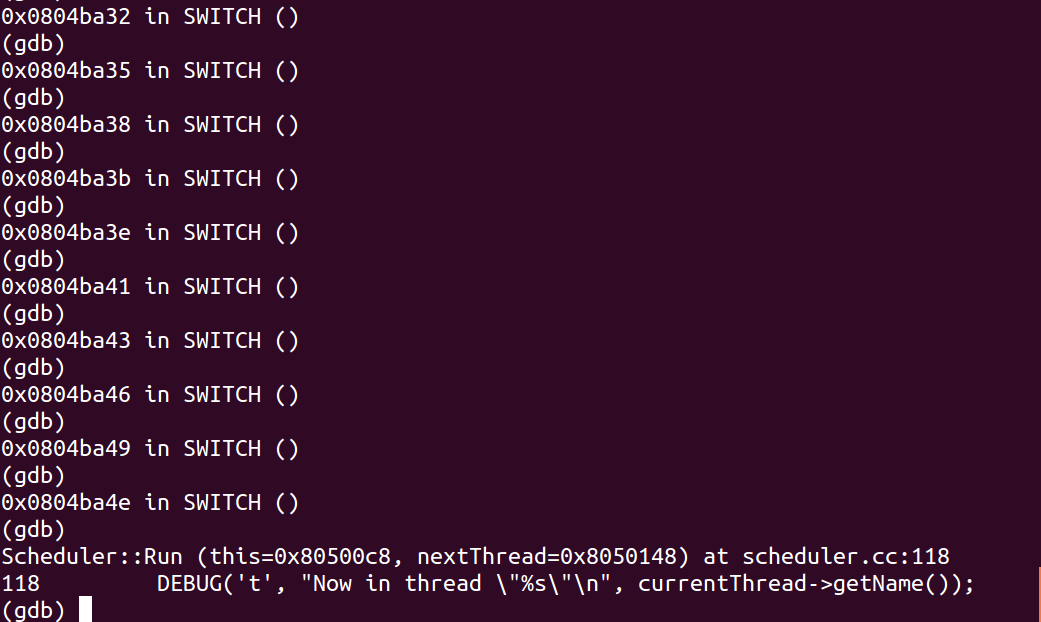


6、当主线程第一次运行 SWITCH()函数,执行到函数 SWITCH()的最后一条指令 ret 时,CPU 返回的地址是多少? 该地址对应程序的什么位置?



单步走了一遍，发现ret结束的时候，返回的地址如图，这个地址是ThreadRoot的首地址。

7、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?



又走了一遍SWITCH，发现返回地址回到了RUN函数里，走到了SWITCH的下一句，是DEBUG。

8、为什么两次不一样？

ThreadRoot() 是所有利用 Thread::Fork() 创建的线程的入口，在第一次调用yield时，在调用run的时候，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。

# 实验三、利用信号量实现线程同步

## 一、描述问题：

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

2.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。

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

4.分析 Thread::Fork() ,理解内核创建线程的过程;

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

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

7.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;

## 二、阐述基本方法：

这个实验主要是实现prodcons++.cc中的三个函数。

### 1. ProdCons

void ProdCons()

{

  int i;

  DEBUG('t', "Entering ProdCons");

  nempty = new Semaphore("nempty", BUFF\_SIZE);

  nfull = new Semaphore("nfull", 0);

  mutex = new Semaphore("mutex", 1);

  ring = new Ring(BUFF\_SIZE);

  for (i = 0; i < N\_PROD; i++)

  {

    sprintf(prod\_names[i], "producer\_%d", i);

    producers[i] = new Thread(prod\_names[i]);

    producers[i]->Fork(Producer, i);

  };

  for (i = 0; i < N\_CONS; i++)

  {

    consumers[i] = new Thread(cons\_names[i]);

    consumers[i]->Fork(Consumer, i);

  };

}

这个函数的主要作用是初始化生产者与消费者，首先定义了信号量，用于保证同步和互斥将每一个生产者和消费者都绑定上对应的函数。

### 2.producer

void Producer(\_int which){

  int num;

  slot \*message = new slot(0, 0);

  for (num = 0; num < N\_MESSG; num++)

  {

    message->thread\_id = which;

    message->value = num;

    nempty->P();

    mutex->P();

    ring->Put(message);

    mutex->V();

    nfull->V();

  }

}

这个是生产者，做的操作是从循环队列中放置一个信息。

### 3.consumer

void Consumer(\_int which){

  char str[MAXLEN];

  char fname[LINELEN];

  int fd;

  slot \*message = new slot(0, 0);

  sprintf(fname, "tmp\_%d", which);

  if ((fd = creat(fname, 0600)) == -1)

  {

    perror("creat: file create failed");

  exit(1);

  }

  for (;;)

  {

    nfull->P();

    mutex->P();

    ring->Get(message);

    mutex->V();

    nempty->V();

    sprintf(str, "producer id --> %d; Message number --> %d;\n",

    message->thread\_id,

    message->value);

    if (write(fd, str, strlen(str)) == -1)

    {

      perror("write: write failed");

      exit(1);

    }

  }

}

消费者首先新建了一个文件，然后去循环队列中取出消息，然后在文件中输出。

## 三、源代码+注释:

### Main.cc

Main函数中，可以看到在首先调用的 Initialize()进行初始化操作。

int

main(int argc, char \*\*argv)

{

    int argCount;           // the number of arguments

                    // for a particular command

    DEBUG('t', "Entering main");

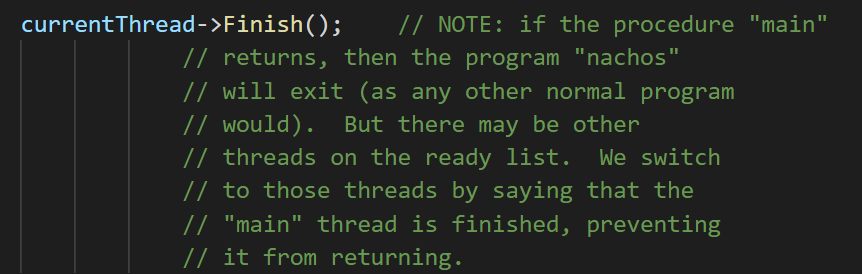
    (void) Initialize(argc, argv);

#ifdef THREADS

    ProdCons();

#endif

然后调用了ProdCons()函数，进行这个实验。



### ring.cc

#### 1.ring

Ring::Ring(int sz)

{

    if (sz < 1) {

    fprintf(stderr, "Error: Ring: size %d too small\n", sz);

    exit(1);

    }

    // Initialize the data members of the ring object.

    size = sz;

    in = 0;

    out = 0;

    buffer = new slot[size]; //allocate an array of slots.

}

这里是初始化循环队列，sz是循环队列的大小，in是尾指针，out是头指针，buffer中存的是具体的内容。

#### 2.put

Void Ring::Put(slot \*message)

{

    buffer[in].thread\_id = message->thread\_id;

    buffer[in].value = message->value;

    in = (in + 1) % size;

}

生产者往循环队列里面放置，先放置，然后头指针加一。

#### 3.get

Void Ring::Get(slot \*message)

{

    message->thread\_id = buffer[out].thread\_id;

    message->value = buffer[out].value;

    out = (out + 1) % size;

}

消费者取元素，取出头部后，然后头指针+1。

## 四、实验中遇到的问题:

为什么当消费者生产者的数量都是2的时候，直接运行nachos，tmp\_1为空，而命令行参数加上-rs 加随机数后，就有东西了。

因为不加-rs的话，使用的是FCFS方法，消费者1会把所有的生产者生产的全部抢走，因此tmp\_1是空的。如果加上-rs，这时候使用的是RR方法，经过时间片轮转，消费者1就会取出循环队列中的一部分。

## 五、总结与收获

1.这次实验熟悉掌握了 GDB 调试方法，主要是使用了，设置断点，查看寄存器，单步进入等命令。使用GDB查看函数地址非常方便。

2．此次实验对于生产者、消费者问题的实现与加深理解，通过阅读代码深入理解了nachos中为同步所设置的信号量、锁、条件变量等数据结构。

3.熟悉掌握了nachos中的线程调度、线程运行的各个系统函数以及时间片轮转调度的具体原理和方法。Nachos一般情况下使用的是FCFS调度方法，当带上参数时，使用了RR方法。

目录

[实验四、Nachos 的文件系统 2](#_Toc70162376)

[一、描述问题： 2](#_Toc70162377)

[二、阐述基本方法： 2](#_Toc70162378)

[三、源代码+注释: 3](#_Toc70162379)

[Disk.cc 3](#_Toc70162380)

[Disk 5](#_Toc70162381)

[ReadRequest 6](#_Toc70162382)

[WriteRequest 6](#_Toc70162383)

[TimeToSeek 7](#_Toc70162384)

[ComputeLatency 7](#_Toc70162385)

[SynchDisk.cc 8](#_Toc70162386)

[SynchDisk 8](#_Toc70162387)

[ReadSector 8](#_Toc70162388)

[WriteSector 9](#_Toc70162389)

[OpenFile.cc 9](#_Toc70162390)

[OpenFile 9](#_Toc70162391)

[ReadAt 9](#_Toc70162392)

[Read 10](#_Toc70162393)

[WriteAt 11](#_Toc70162394)

[Directory.cc 12](#_Toc70162395)

[DirectoryEntry 12](#_Toc70162396)

[FetchFrom 12](#_Toc70162397)

[WriteBack 13](#_Toc70162398)

[FindIndex 13](#_Toc70162399)

[Find 13](#_Toc70162400)

[Add 13](#_Toc70162401)

[BitMap 14](#_Toc70162402)

[FileHeader 14](#_Toc70162403)

[Allocate 14](#_Toc70162404)

[Deallocate 15](#_Toc70162405)

[四、实验过程 15](#_Toc70162406)

[1.硬盘格式化 15](#_Toc70162407)

[2.复制一个文件到硬盘 17](#_Toc70162408)

[复制另一个文件到硬盘 19](#_Toc70162409)

[删除 Nachos 硬盘上的文件 21](#_Toc70162410)

[打开 Nachos 文件的过程 22](#_Toc70162411)

[五、实验中遇到的问题 22](#_Toc70162412)

[实验五、扩展 Nachos 的文件系统 24](#_Toc70162413)

[一、 描述问题： 24](#_Toc70162414)

[二、阐述基本方法： 24](#_Toc70162415)

[三、 源代码+注释 25](#_Toc70162416)

[1.nachos–ap 与 nachos–hap 命令的实现 25](#_Toc70162417)

[四、实验过程 31](#_Toc70162418)

# 实验四、Nachos 的文件系统

## 一、描述问题：

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;

## 二、阐述基本方法：

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。

## 三、源代码+注释:

### Disk.cc

硬盘的基本信息：

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。硬盘的每个逻辑块大小为128字节，与一个扇区对应。硬盘采用位示图（BitMap）的思想管理硬盘的空闲块，当一个扇区空闲，位示图中对应的位为0，否则为1。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。

#define SectorSize    128 // 扇区的大小：128字节

#define SectorsPerTrack   32  // 每个磁道上扇区的数目

#define NumTracks     32  // 每个硬盘上磁道的数目

#define NumSectors    (SectorsPerTrack \* NumTracks)//每个硬盘上扇区的总数

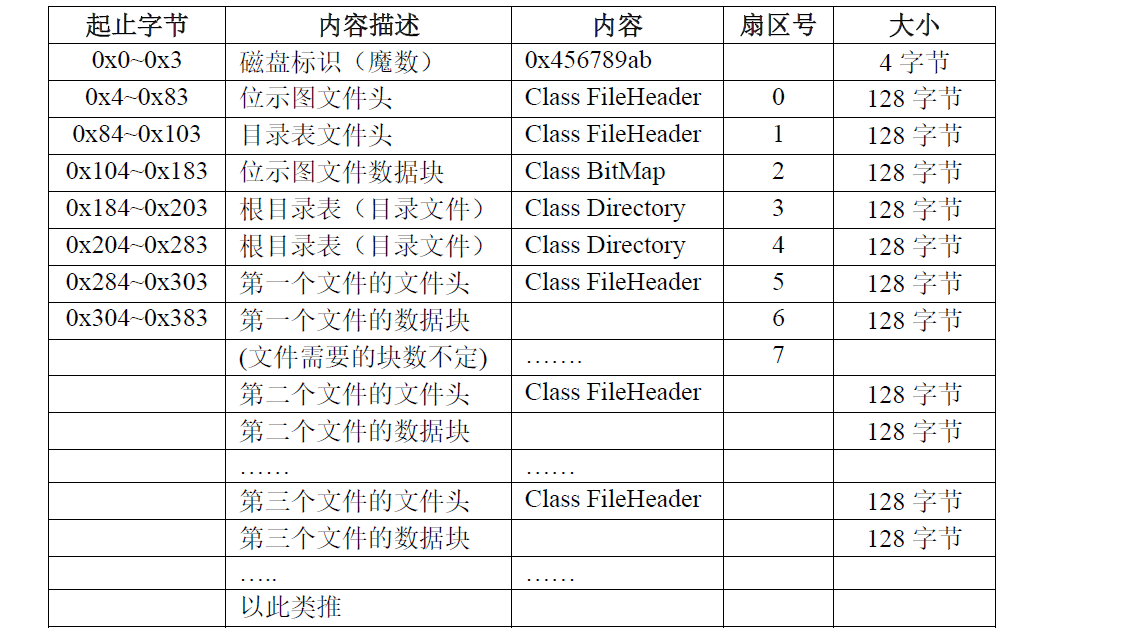
#define MagicNumber     0x456789ab//魔数

#define MagicSize   sizeof(int)

#define DiskSize    (MagicSize + (NumSectors \* SectorSize))//硬盘大小：魔数的空间+（扇区总数\*每个扇区大小）

0~3 Bytes是硬盘标识（MagicNumber，0x456789ab），指明该硬盘是一个Nachos硬盘

之后，每128个Bytes为一个扇区（Sector), 即4~131Bytes为0号扇区，132~259Bytes为1号扇区，以此类推。



(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;

#### Disk

硬盘初始化。

Disk::Disk(char\* name, VoidFunctionPtr callWhenDone, \_int callArg)

{

    int magicNum;

    int tmp = 0;

    DEBUG('d', "Initializing the disk, 0x%x 0x%x\n", callWhenDone, callArg);

    handler = callWhenDone;

    handlerArg = callArg;

    lastSector = 0;

    bufferInit = 0;

fileno = OpenForReadWrite(name, FALSE);

// Disk用文件来模拟硬盘， 若相应name的文件已经存在，就检查MagicNumber是否正确；若不存在，则创建并写入MagicNumber

if (fileno >= 0) {

//检查magicnumber

     Read(fileno, (char \*) &magicNum, MagicSize);

     ASSERT(magicNum == MagicNumber);

    } else {                // file doesn't exist, create it

        fileno = OpenForWrite(name);

    magicNum = MagicNumber;

    WriteFile(fileno, (char \*) &magicNum, MagicSize); // write magic number

// need to write at end of file, so that reads will not return EOF

//确定模拟硬盘文件的大小（结束位置），用0充当硬盘文件的结束符

    Lseek(fileno, DiskSize - sizeof(int), 0);

    WriteFile(fileno, (char \*)&tmp, sizeof(int));

    }

    active = FALSE;

}

#### ReadRequest

读取指定扇区的数据,Read函数即调用库函数read，执行读文件（模拟磁盘）的操作。然后执行中断，中断计算出来的时间。

Void Disk::ReadRequest(int sectorNumber, char\* data)

{

    int ticks = ComputeLatency(sectorNumber, FALSE);

    ASSERT(!active);                // only one request at a time

    ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));

    DEBUG('d', "Reading from sector %d\n", sectorNumber);

    Lseek(fileno, SectorSize \* sectorNumber + MagicSize, 0);

    Read(fileno, data, SectorSize);//读取指定扇区的信息，fileno：硬盘的文件描述符，用文件模拟的硬盘。

    if (DebugIsEnabled('d'))

    PrintSector(FALSE, sectorNumber, data);

    active = TRUE;

    UpdateLast(sectorNumber);

    stats->numDiskReads++;

    interrupt->Schedule(DiskDone, (\_int) this, ticks, DiskInt);

}

#### WriteRequest

WriteRequest 将数据写入指定扇区，WriteFile即调用库函数write，执行写文件的操作

Void Disk::WriteRequest(int sectorNumber, char\* data)

{

    int ticks = ComputeLatency(sectorNumber, TRUE);

    ASSERT(!active);

    ASSERT((sectorNumber >= 0) && (sectorNumber < NumSectors));

    DEBUG('d', "Writing to sector %d\n", sectorNumber);

    Lseek(fileno, SectorSize \* sectorNumber + MagicSize, 0);

    WriteFile(fileno, data, SectorSize);//写到指定扇区里。

    if (DebugIsEnabled('d'))

    PrintSector(TRUE, sectorNumber, data);

    active = TRUE;

    UpdateLast(sectorNumber);

    stats->numDiskWrites++;

    interrupt->Schedule(DiskDone, (\_int) this, ticks, DiskInt);

}

#### TimeToSeek

通过计算上一次磁头所在的磁道和目标磁道之间的差值，来计算寻道时间。

int

Disk::TimeToSeek(int newSector, int \*rotation)

{

    int newTrack = newSector / SectorsPerTrack;

    int oldTrack = lastSector / SectorsPerTrack;

    int seek = abs(newTrack - oldTrack) \* SeekTime;

                // how long will seek take?

    int over = (stats->totalTicks + seek) % RotationTime;

                //RotationTime是转过一个扇区所用的时间，若硬盘存取总的耗时可以整除RotationTime，则说明磁头位于扇区开始位置

    \*rotation = 0;

    if (over > 0)       // if so, need to round up to next full sector

    \*rotation = RotationTime - over;

    return seek;

}

#### ComputeLatency

一次硬盘操作的访问时间，访问时间= 寻道时间+旋转延迟+传输时间（磁头转过一个扇区的时间）

Int Disk::ComputeLatency(int newSector, bool writing)

{

    int rotation;

    int seek = TimeToSeek(newSector, &rotation);

    int timeAfter = stats->totalTicks + seek + rotation;

#ifndef NOTRACKBUF  // turn this on if you don't want the track buffer stuff

    // check if track buffer applies

    if ((writing == FALSE) && (seek == 0)

        && (((timeAfter - bufferInit) / RotationTime)

                > ModuloDiff(newSector, bufferInit / RotationTime))) {

        DEBUG('d', "Request latency = %d\n", RotationTime);

    return RotationTime; // time to transfer sector from the track buffer

    }

#e

    rotation += ModuloDiff(newSector, timeAfter / RotationTime) \* RotationTime;

    DEBUG('d', "Request latency = %d\n", seek + rotation + RotationTime);

    return(seek + rotation + RotationTime);

}

### SynchDisk.cc

SynchDisk为访问Disk提供了接口。Disk对象的读/写函数，在执行read/write后，就直接返回，不会等待模拟的读/写时间到达；而 SynchDisk提供了阻塞的读/写操作，只有当读/写时间到了之后，才会返回。SynchDisk提供了互斥访问Disk对象的方法。

#### SynchDisk

定义Lock来实现互斥访问；定义semaphore来阻塞进程，等待读/写时间到达

SynchDisk::SynchDisk(char\* name)

{

    semaphore = new Semaphore("synch disk", 0);

    lock = new Lock("synch disk lock");

    disk = new Disk(name, DiskRequestDone, (\_int) this);

}

#### ReadSector

将磁盘扇区的内容读入缓冲区。只有读取完数据后才返回。

Void SynchDisk::ReadSector(int sectorNumber, char\* data)

{

    lock->Acquire();            //同一时间只允许一个

    disk->ReadRequest(sectorNumber, data);

    semaphore->P();         // wait for interrupt

    lock->Release();

}

#### WriteSector

将缓冲区的内容写入磁盘扇区。只有在写入数据之后才返回。

Void SynchDisk::WriteSector(int sectorNumber, char\* data)

{

    lock->Acquire();            // only one disk I/O at a time

    disk->WriteRequest(sectorNumber, data);

    semaphore->P();         // wait for interrupt

    lock->Release();

}

### OpenFile.cc

OpenFile 负责访问具体的文件，通过OpenFile对象，来对特定的文件进行读写操作。每个OpenFile对象都有一个关联的读写位置指针，用于跟踪上一次读或写操作的结束位置。

FileHeader \*hdr;            // 文件头

    int seekPosition;           // 文件读写指针的位置

    int hdrSector;//文件头扇区编号

#### OpenFile

OpenFile 构造函数初始化文件头，hdr->FetchFrom调用了SynchDisk::ReadSector, 将文件头所在扇区的数据读取出来。将文件读写位置指针清零。

OpenFile::OpenFile(int sector)

{

    hdr = new FileHeader;

    hdr->FetchFrom(sector);

    seekPosition = 0;

}

#### ReadAt

"into"——包含要从磁盘读取的数据的缓冲区

"numBytes"——要传输的字节数

"position"——第一个字节在文件中的偏移量

Int OpenFile::ReadAt(char \*into, int numBytes, int position)

{

    int fileLength = hdr->FileLength();

    int i, firstSector, lastSector, numSectors;

    char \*buf;

    if ((numBytes <= 0) || (position >= fileLength))

        return 0;               // check request

    if ((position + numBytes) > fileLength)

    numBytes = fileLength - position;

    DEBUG('f', "Reading %d bytes at %d, from file of length %d.\n",

            numBytes, position, fileLength);

首先需要根据读取的位置、读取数据的大小，确定第一个和最后一个逻辑块的编号。

    firstSector = divRoundDown(position, SectorSize);

    lastSector = divRoundDown(position + numBytes - 1, SectorSize);

    numSectors = 1 + lastSector - firstSector;

    // read in all the full and partial sectors that we need

buf = new char[numSectors \* SectorSize];

//将扇区中的数据读入buf。

for (i = firstSector; i <= lastSector; i++)

//逻辑块号转换为物理扇区号

        synchDisk->ReadSector(hdr->ByteToSector(i \* SectorSize),

                    &buf[(i - firstSector) \* SectorSize]);

    // copy the part we want

    bcopy(&buf[position - (firstSector \* SectorSize)], into, numBytes);

    delete [] buf;

    return numBytes;

}

#### Read

"into"——要从磁盘读取数据的缓冲区。

"numBytes"——要传输的字节数。

Read直接调用ReadAt函数，并修改读写指针的位置。

Int OpenFile:: Read (char \*into, int numBytes)

{

   int result = ReadAt(into, numBytes, seekPosition);

   seekPosition += result;

   return result;

}

#### WriteAt

对于写操作,我们必须首先读取将部分写入的任何扇区,这样就不会覆盖未修改的部分。然后,我们复制要修改的数据,并写回请求中的所有完整或部分扇区。

Int OpenFile::WriteAt(char \*from, int numBytes, int position)

{

    int fileLength = hdr->FileLength();

    int i, firstSector, lastSector, numSectors;

    bool firstAligned, lastAligned;

    char \*buf;

    if ((numBytes <= 0) || (position >= fileLength))

    return 0;               // check request

    if ((position + numBytes) > fileLength)

    numBytes = fileLength - position;

    DEBUG('f', "Writing %d bytes at %d, from file of length %d.\n",

            numBytes, position, fileLength);

    firstSector = divRoundDown(position, SectorSize);

    lastSector = divRoundDown(position + numBytes - 1, SectorSize);

    numSectors = 1 + lastSector - firstSector;

    buf = new char[numSectors \* SectorSize];

//判断第一块和最后一块会不会被修改

    firstAligned = (bool)(position == (firstSector \* SectorSize));

    lastAligned = (bool)((position + numBytes) == ((lastSector + 1) \* SectorSize));

// read in first and last sector, if they are to be partially modified

//若第一个逻辑块会发生部分修改的问题，要在buf最开始，先读入第一个逻辑块；最后一个逻辑块也同理，但是要放在buf的最后。

    if (!firstAligned)

        ReadAt(buf, SectorSize, firstSector \* SectorSize);

    if (!lastAligned && ((firstSector != lastSector) || firstAligned))

        ReadAt(&buf[(lastSector - firstSector) \* SectorSize],

                SectorSize, lastSector \* SectorSize);

// copy in the bytes we want to change

    bcopy(from, &buf[position - (firstSector \* SectorSize)], numBytes);

// write modified sectors back

    for (i = firstSector; i <= lastSector; i++)

        synchDisk->WriteSector(hdr->ByteToSector(i \* SectorSize),

                    &buf[(i - firstSector) \* SectorSize]);

    delete [] buf;

    return numBytes;

}

### Directory.cc

目录是一个包含固定长度条目的表;每个条目表示单个文件，并包含文件名和文件头在磁盘上的位置。每个目录条目的固定大小意味着我们对文件名的固定最大大小有限制。

FileNameMaxLen    9

文件最长长度是9

#### DirectoryEntry

三元组<使用位，文件头扇区号，文件名称>

class DirectoryEntry {

  public:

    bool inUse;       // Is this directory entry in use?

    int sector;       // Location on disk to find the

          //   FileHeader for this file

    char name[FileNameMaxLen + 1];  // Text name for file, with +1 for

          // the trailing '\0'

};

#### FetchFrom

从磁盘读取目录的内容。

File是包含目录内容的nachos文件。

Void Directory::FetchFrom(OpenFile \*file)

{

    (void) file->ReadAt((char \*)table, tableSize \* sizeof(DirectoryEntry), 0);

}

#### WriteBack

将对目录的任何修改写回磁盘

Void Directory::WriteBack(OpenFile \*file)

{

    (void) file->WriteAt((char \*)table, tableSize \* sizeof(DirectoryEntry), 0);

}

#### FindIndex

找到file在目录里的ID，如果没有返回-1。

Int Directory::FindIndex(char \*name)

{

    for (int i = 0; i < tableSize; i++)

        if (table[i].inUse && !strncmp(table[i].name, name, FileNameMaxLen))

        return i;

    return -1;      // name not in directory

}

#### Find

查找名字为name的文件，并返回其文件头所在的扇区号

int

Directory::Find(char \*name)

{

    int i = FindIndex(name);

    if (i != -1)

    return table[i].sector;

    return -1;

}

#### Add

向目录对象中添加目录项。

Bool Directory::Add(char \*name, int newSector)

{

    if (FindIndex(name) != -1)

    return FALSE;

    for (int i = 0; i < tableSize; i++)

        if (!table[i].inUse) {

            table[i].inUse = TRUE;

            strncpy(table[i].name, name, FileNameMaxLen);

            table[i].sector = newSector;

        return TRUE;

    }

    return FALSE;   // no space.  Fix when we have extensible files.

}

### BitMap

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。

/数据结构:位图

位的数组，每个位都可以被独立置位、清空、检查

可以用于内存中空闲页管理、磁盘上空闲扇区管理等

每个位代表相应的页/扇区是否被使用

### FileHeader

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

#define MaxFileSize   (NumDirect \* SectorSize)

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。

#### Allocate

初始化一个新创建的文件头。从空闲磁盘块的映射中分配文件的数据块。如果没有足够的空闲块容纳新文件，则返回FALSE。

boolFileHeader::Allocate(BitMap \*freeMap, int fileSize)

{

//freemap 的作用就是在空闲快位图中分配空闲块

    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;

}

#### Deallocate

重新分配为该文件的数据块分配的所有空间。

void FileHeader::Deallocate(BitMap \*freeMap)

{

    for (int i = 0; i < numSectors; i++) {

    ASSERT(freeMap->Test((int) dataSectors[i]));  // ought to be marked!

    freeMap->Clear((int) dataSectors[i]);

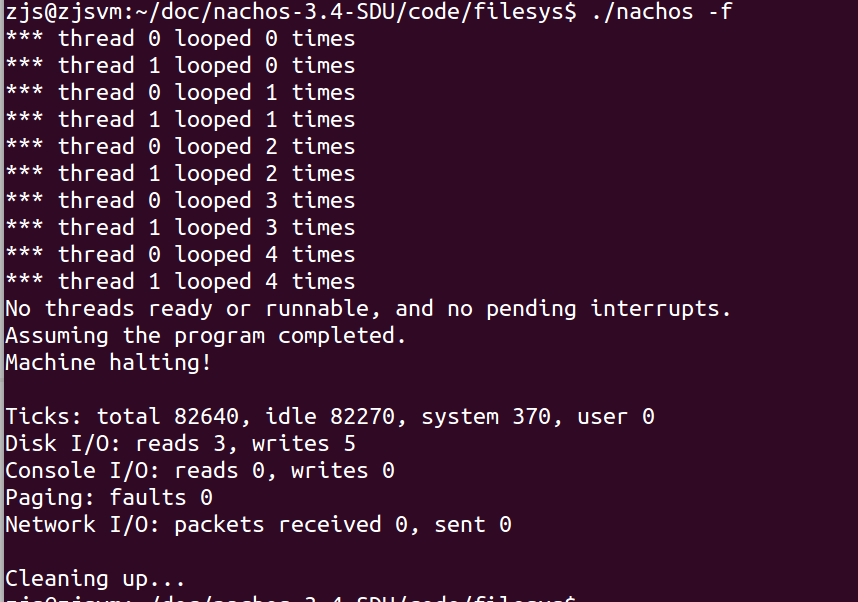
    }

}

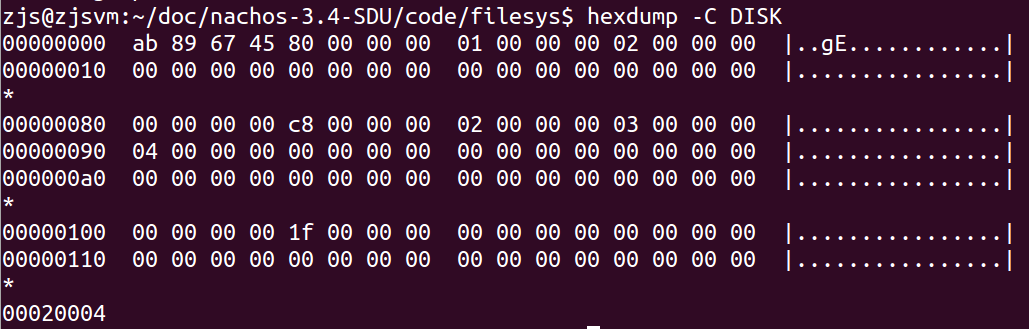
## 四、实验过程

### 1.硬盘格式化

1.初始化硬盘。



2.使用hexdump -C DISK查看DISK 的内容。



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。

(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;

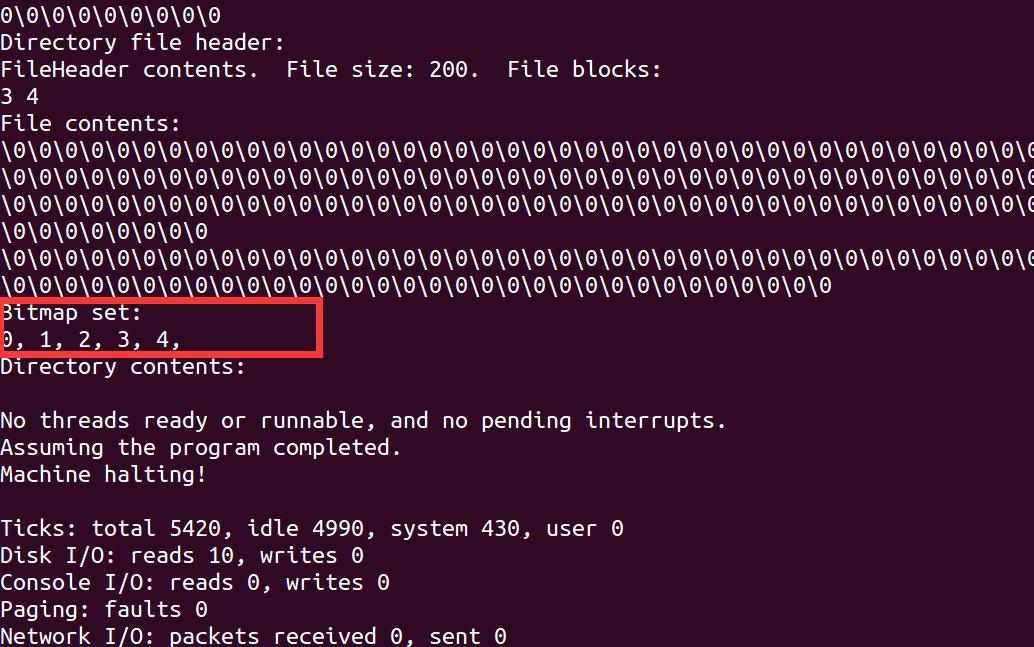
0x90~0x93:目录表文件第 2 个数据块所在的扇区号,其值为 0x4,说明系统将目录表文件第 2 个数据块保存在第 4 号扇区中;

(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;

3.使用./nachos -D查看

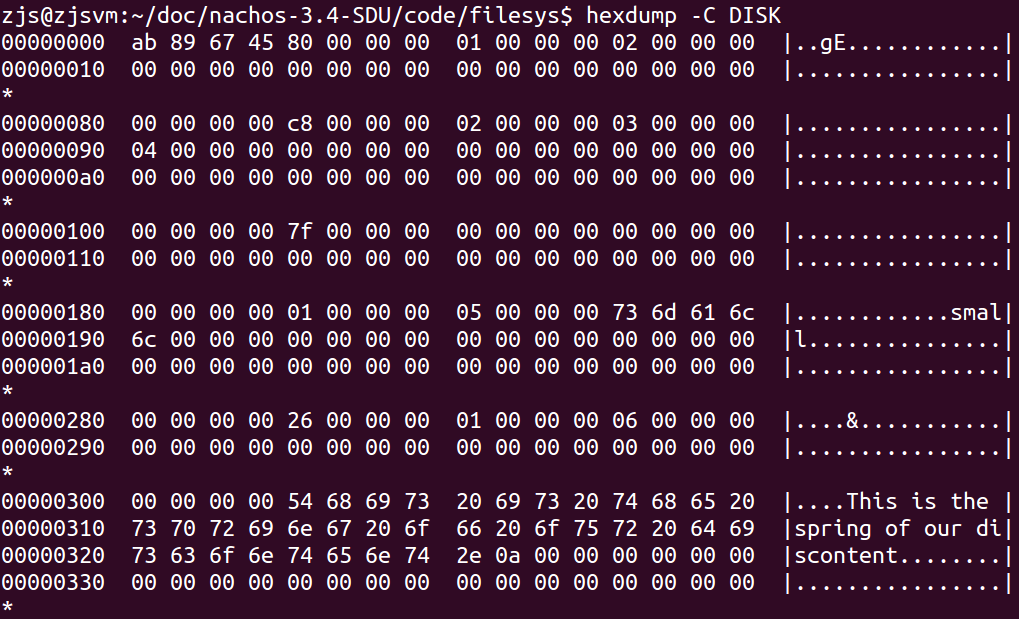


上述信息表示0号扇区为位图文件头，1号扇区为文件目录表文件头，2号扇区为位图文件，3、4号扇区为文件目录表文件。

### 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。



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 , charname[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78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 0x26,说明 small 文件大小是 0x26 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。

(f)6 号扇区(0x304~0x383)是“samll”的文件的数据块,其中 0x304~0x329( 0x329-0x304+1=0x26 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

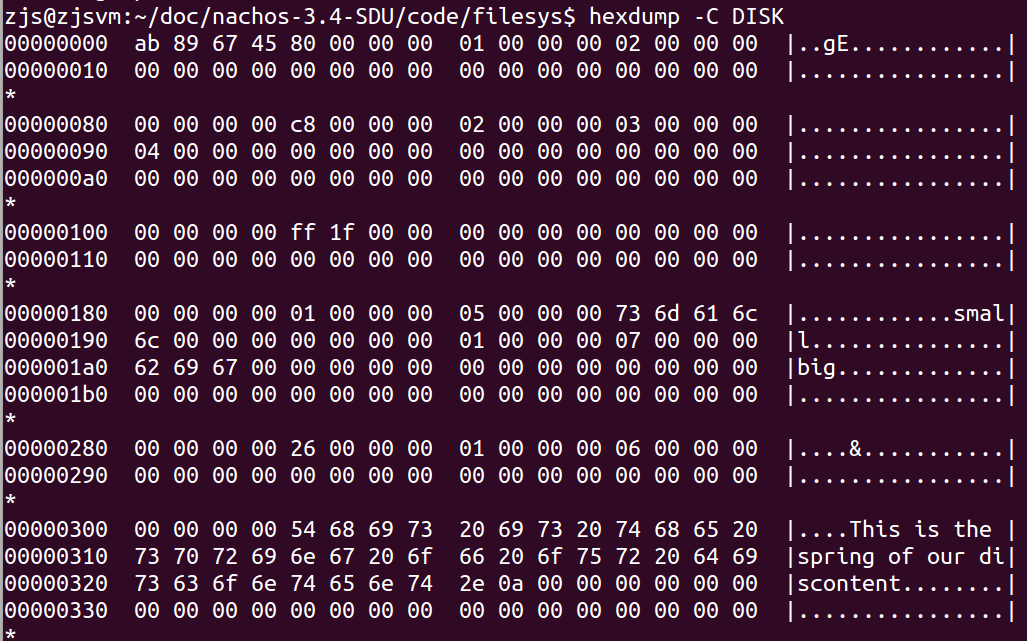
0x284~0x303:5 号扇区,small 文件头

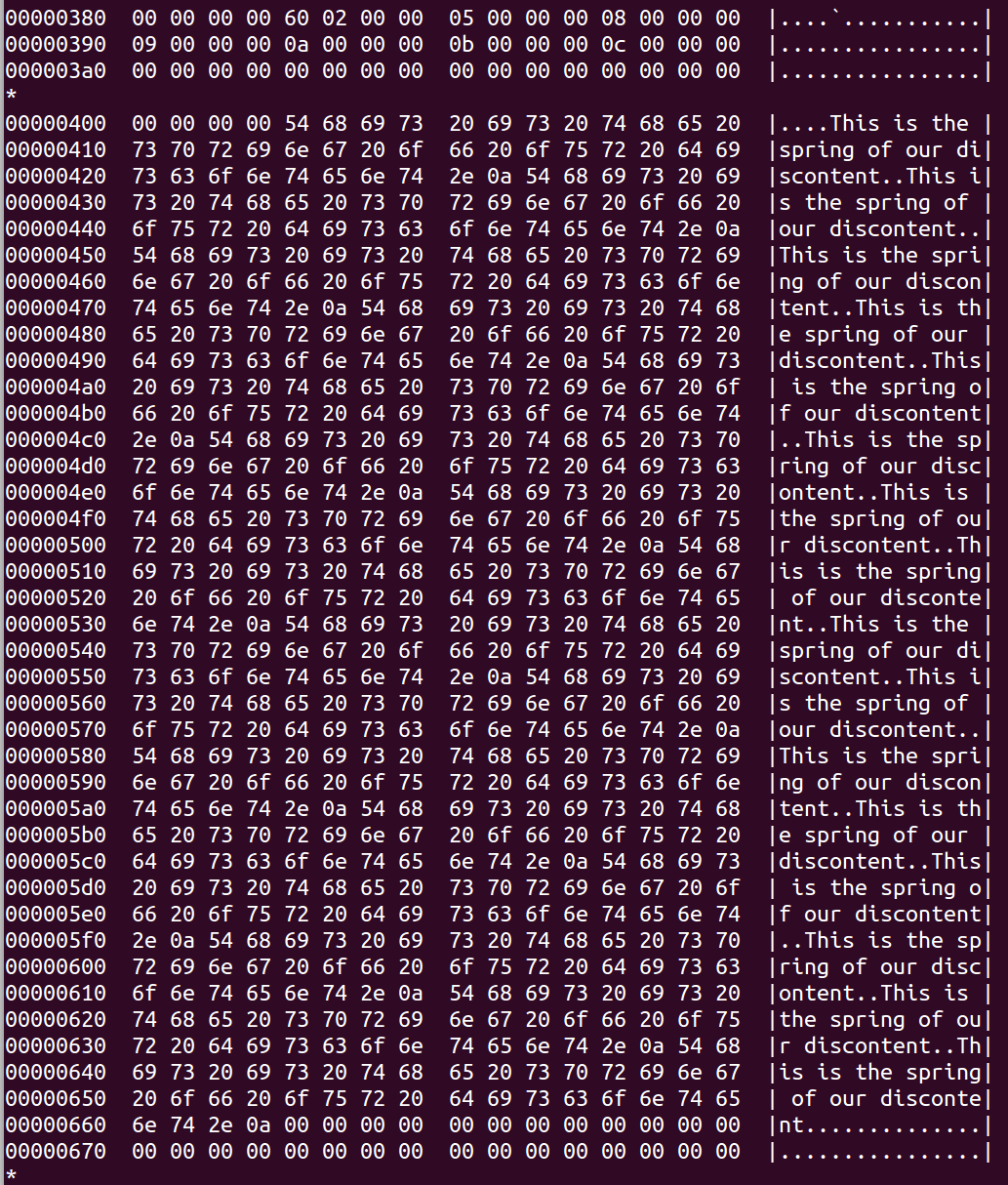
0x304~0x383:6 号扇区,small 文件数据块

### 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。





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81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)

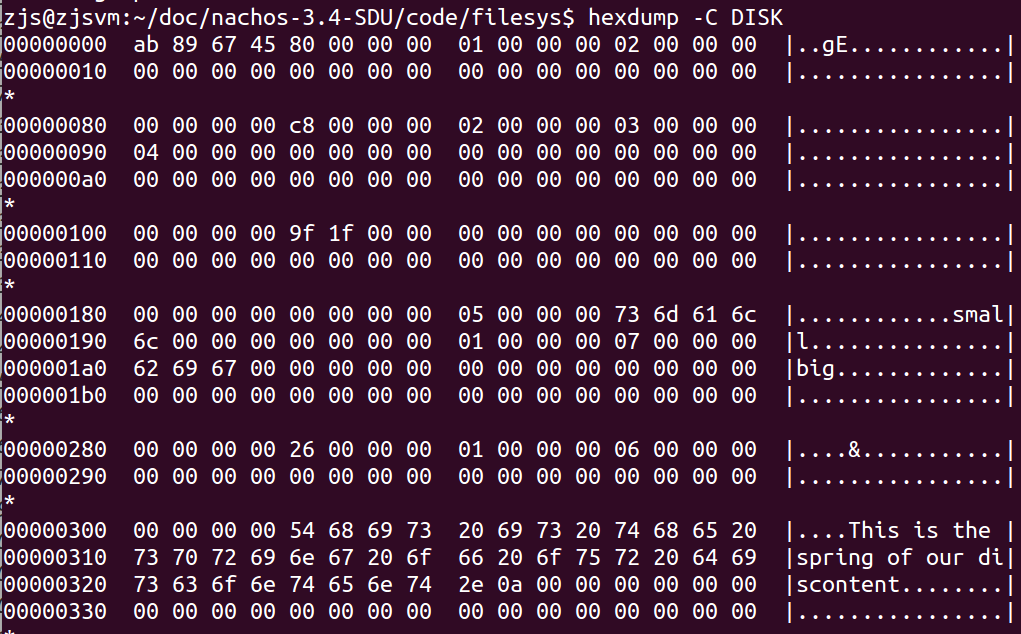
0x484~0x503:9 号扇区,big 文件第 2 块数据(中间的 128 字节)

0x504~0x583:10 号扇区,big 文件第 3 块数据(最后的 84 字节)

### 删除 Nachos 硬盘上的文件

1.nachos –r small,删除 small 文件;

2.hexdump –C DISK,



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0xFF1F 修改为 0x9F1F,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;

### 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;

OpenFile \*FileSystem::Open(char \*name)

{

    Directory \*directory = new Directory(NumDirEntries);

    OpenFile \*openFile = NULL;

    int sector;

    DEBUG('f', "Opening file %s\n", name);

    directory->FetchFrom(directoryFile);

    sector = directory->Find(name);

    if (sector >= 0)

    openFile = new OpenFile(sector);    // name was found in directory

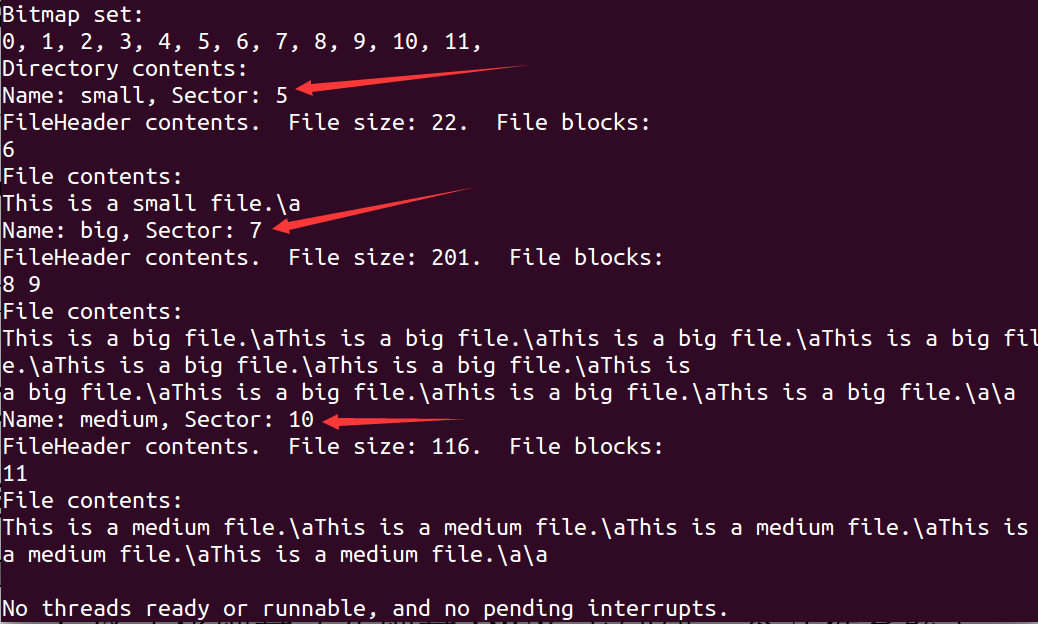
    delete directory;

    return openFile;                // return NULL if not found

}

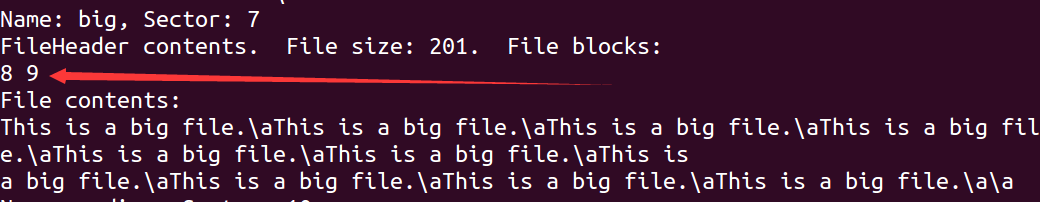
## 五、实验中遇到的问题

(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?

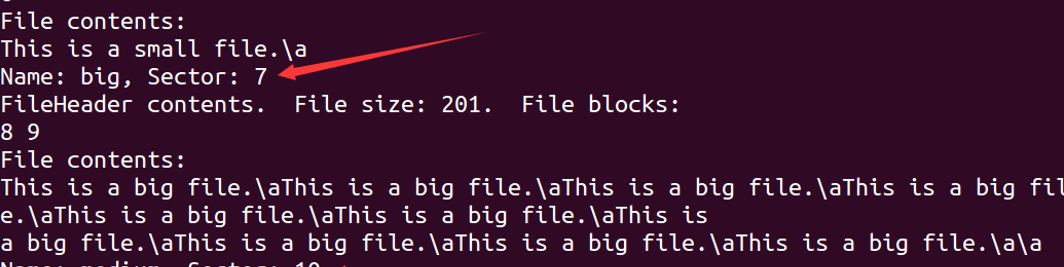


共有三个文件。

(b)文件 big 的数据块(data blocks)的扇区号是多少?

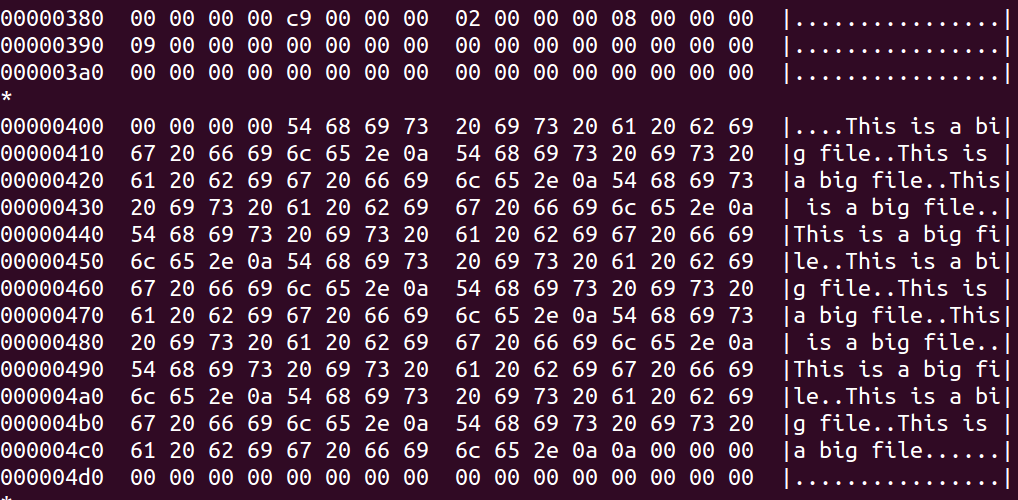


(c)文件 big 的文件头(file header)的扇区号是多少?



(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(thedata blocks and the file header of big)处于硬盘正确位置吗?

扇区号是7,7\*128=896，换成16进制就是380.



完全一致。

# 实验五、扩展 Nachos 的文件系统

## 描述问题：

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;

## 二、阐述基本方法：

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

## 源代码+注释

### 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

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))

return 0; // check request//约束 1

if ((position + numBytes) > fileLength)

numBytes = fileLength - position;

//约束 2

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 : FileHearder:: 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,FileHearder:: WriteBack(intsectorNumber)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。

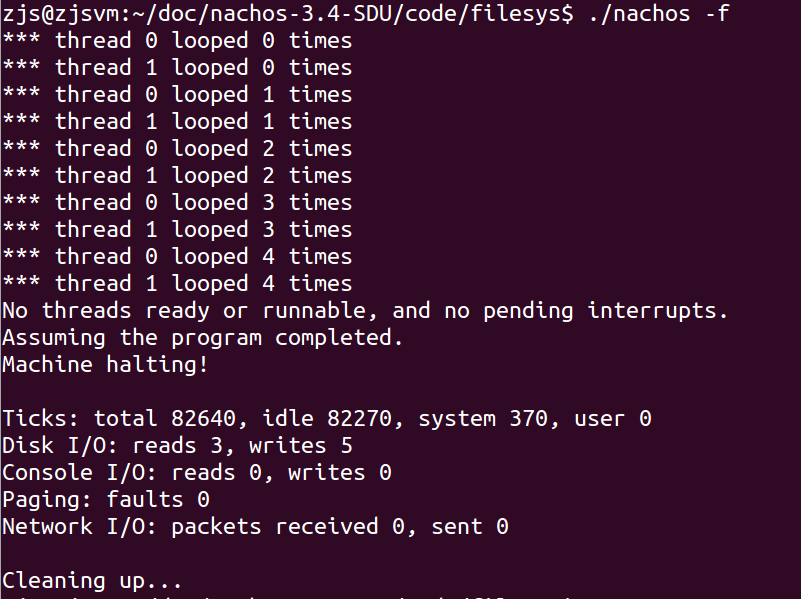
## 四、实验过程

(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;

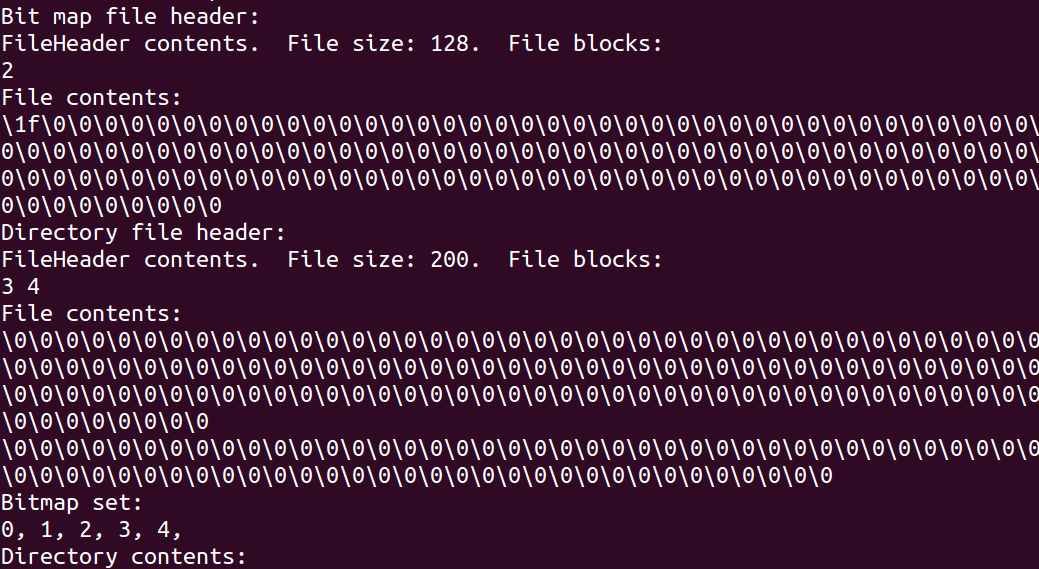


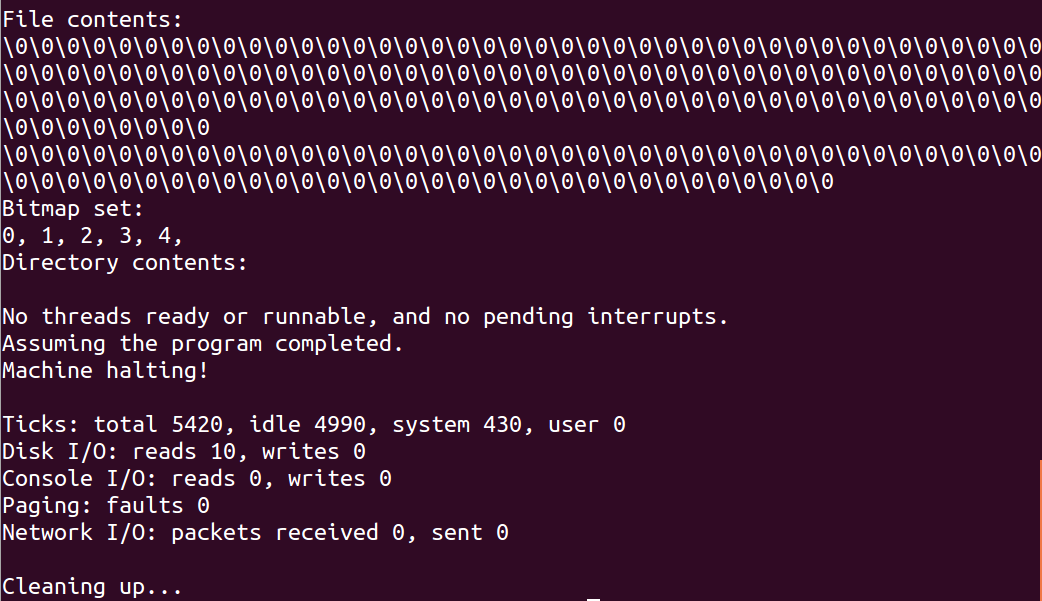
在filesys文件夹下，有了DISK文件，就是nachos模拟出来的硬盘。



(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)

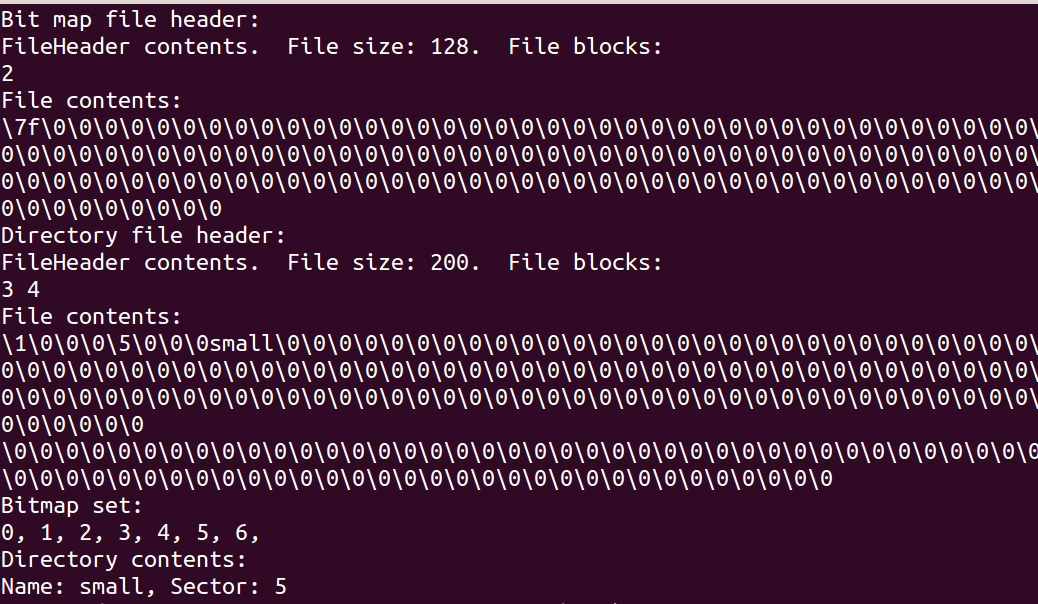


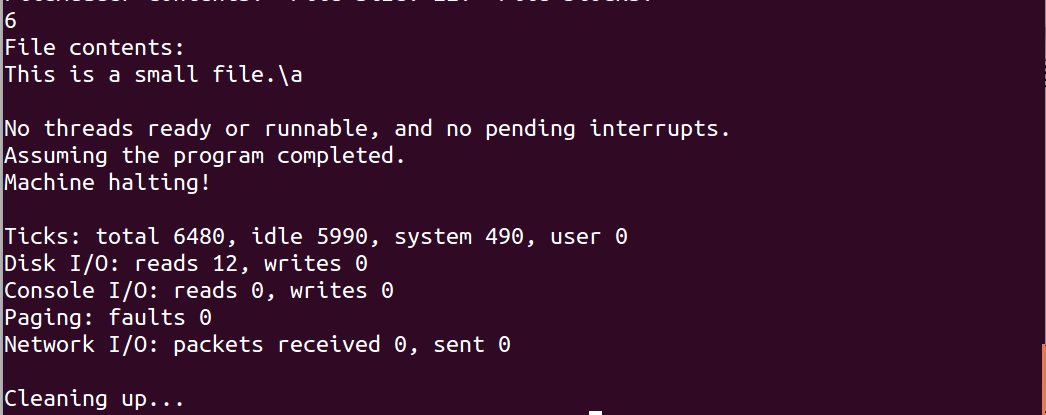


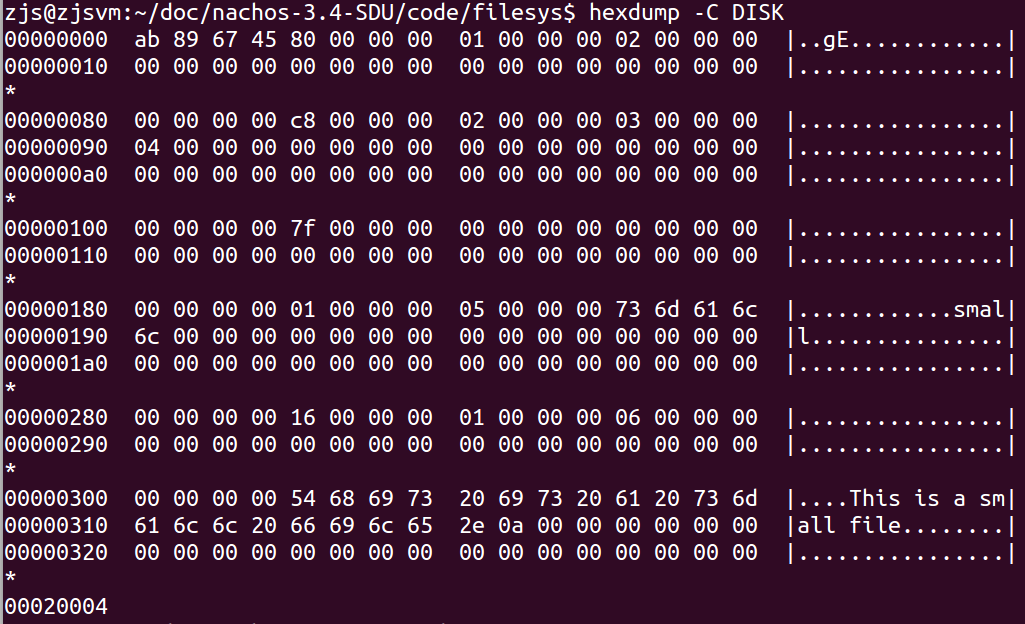
上述信息表示0号扇区为位图文件头，1号扇区为文件目录表文件头，2号扇区为位图文件，3、4号扇区为文件目录表文件。

(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;

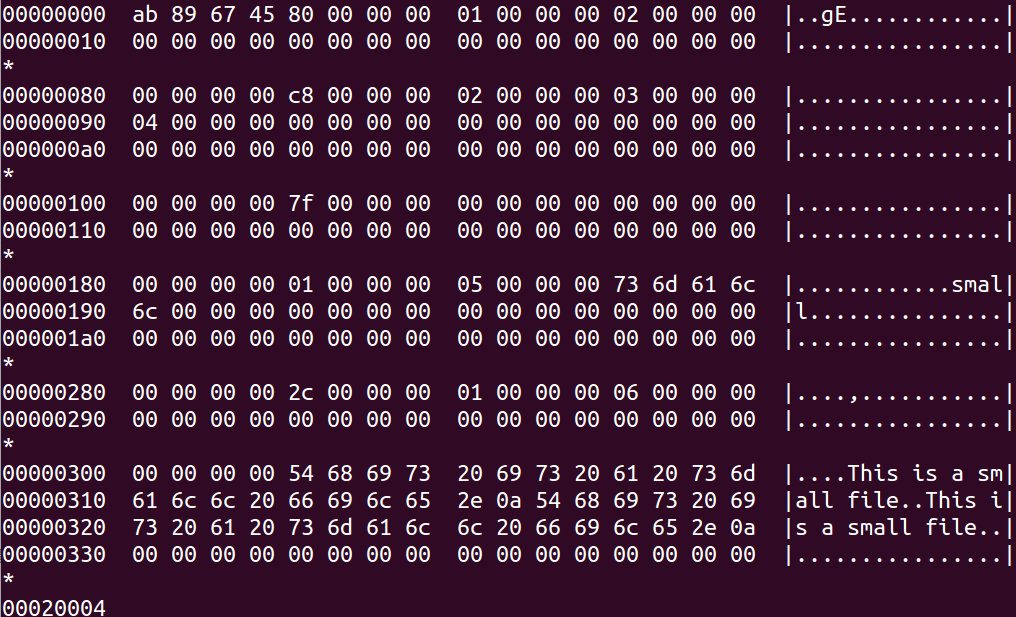




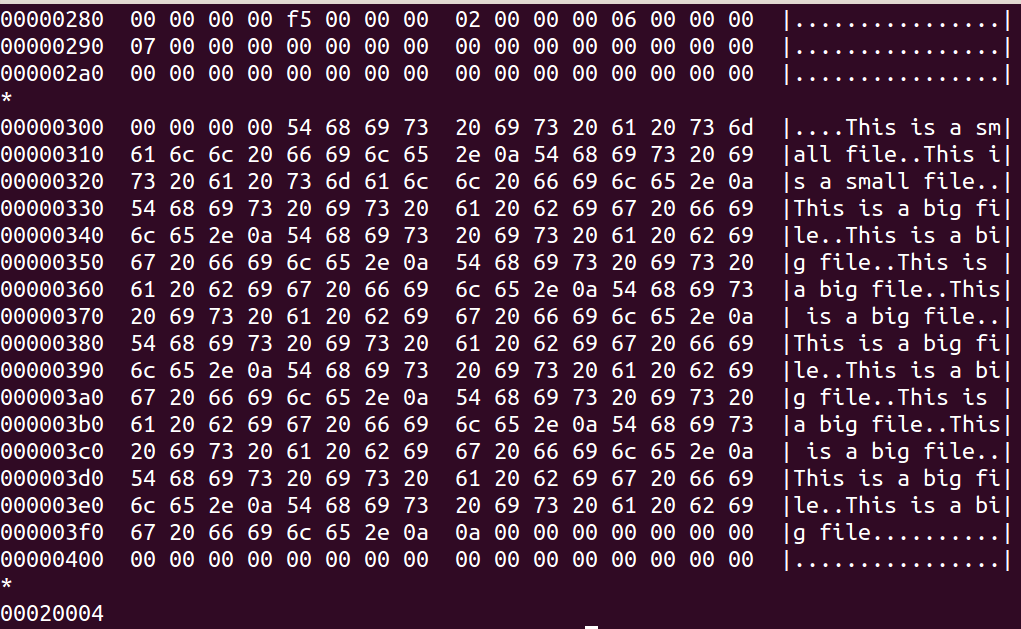


(6)nachos –ap test/small small,将 test 目录下的 UNIX 文件 samll 附加到 Nachos文件 small 中;测试给一个已存在的文件追加数据;

(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;



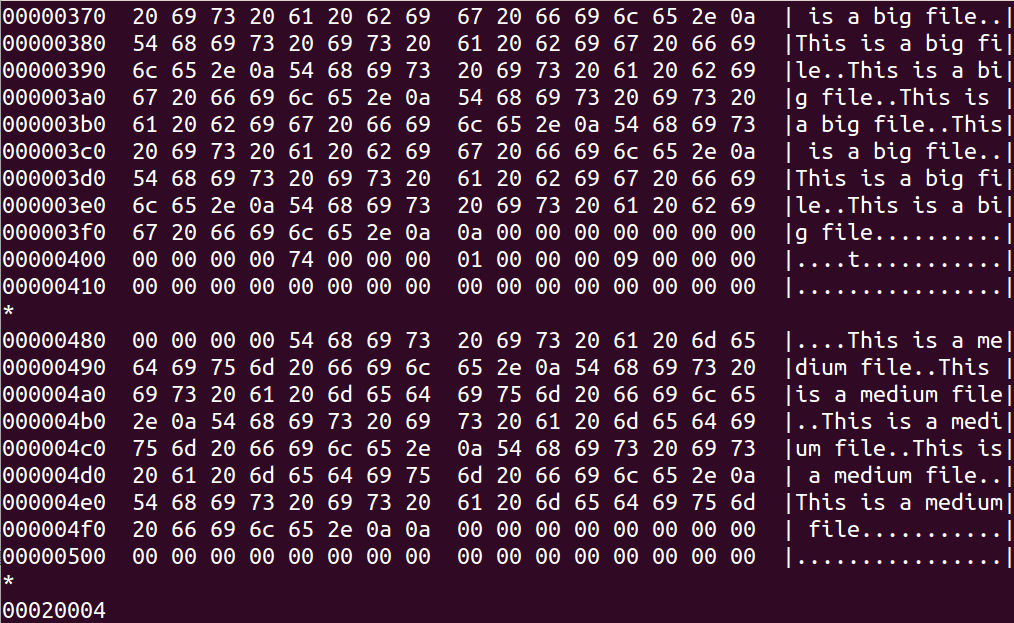
(8)nachos –ap test/big small,将 UNIX 文件 big 附加到一个 small 中;测试为文件分配新扇区的功能;

(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;

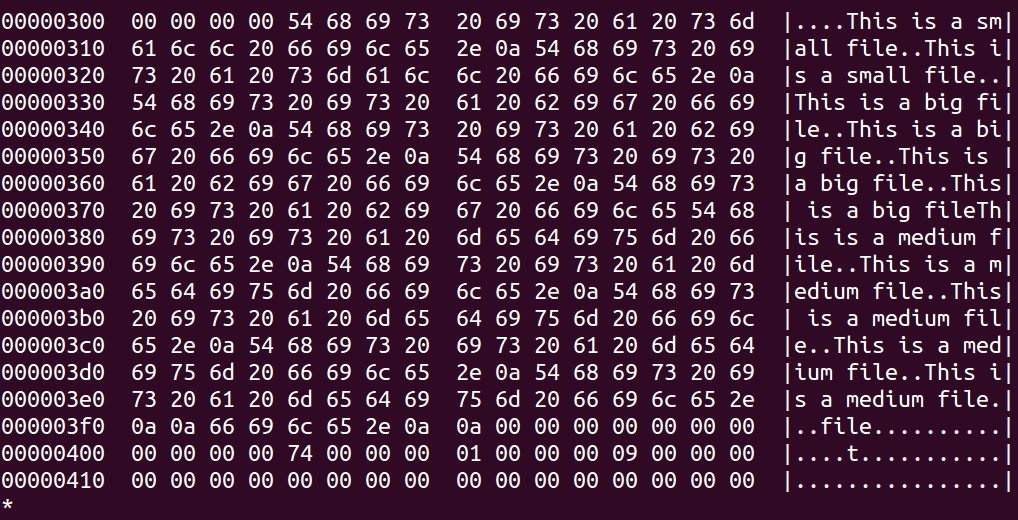
(12)nachos –ap test/big small,将 UNIX 文件 big 附加到一个 small 中;测试Nachos 为 small 新分配的扇区块的位置,即 Nachos 是否为 small 分配不连续的扇区101块,理解操作系统中索引分配的机理;



(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;

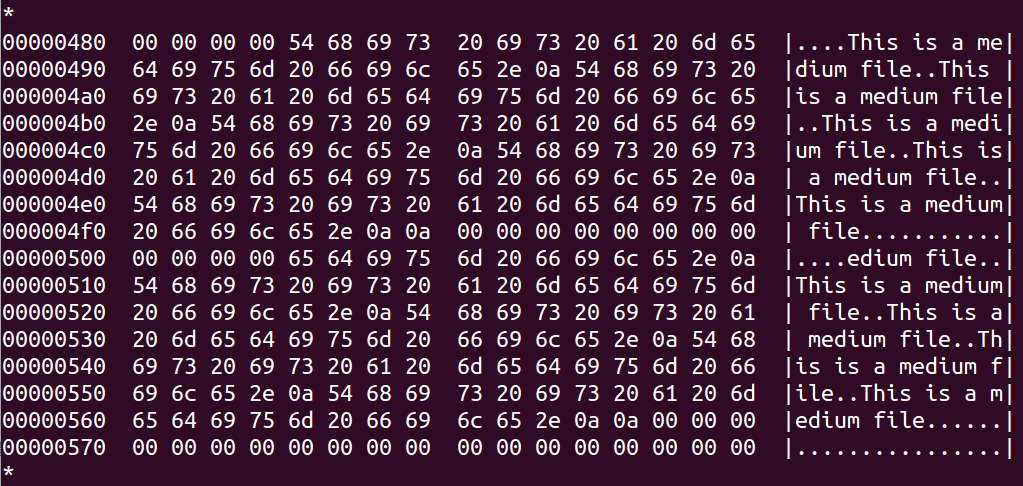
(15)nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息;查看系统是否在 small 的中间写入文件 test/medium 的内容;



从中间部分开始插入，然后覆盖后面的内容。

(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;

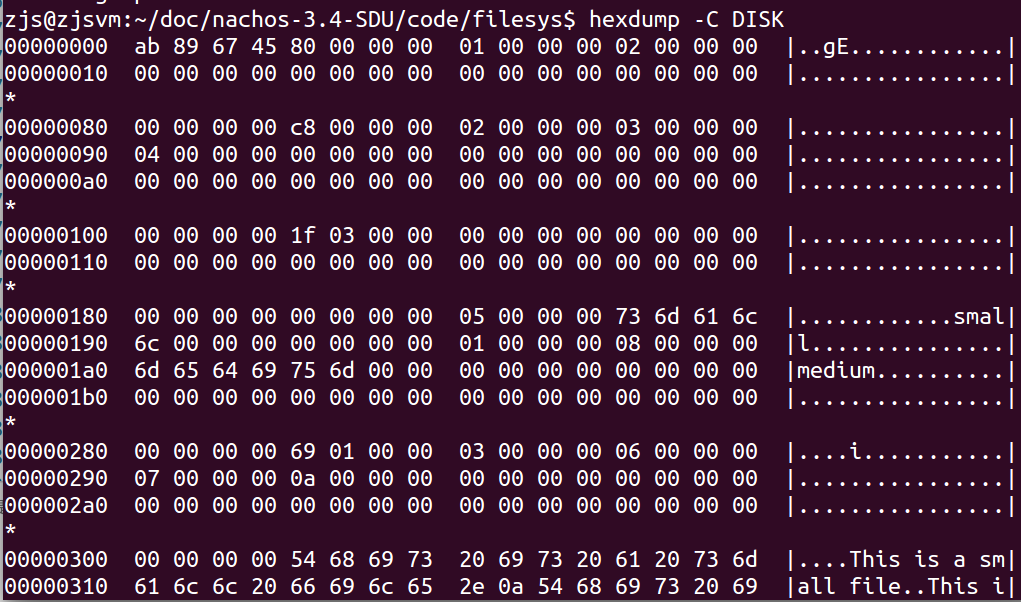


(18)nachos –r small,测试文件删除功能,以及删除后硬盘 DISK 的信息变化;

(19)nachos –D 及 hexdump –C DISK 查看硬盘 DISK 中的文件信息;

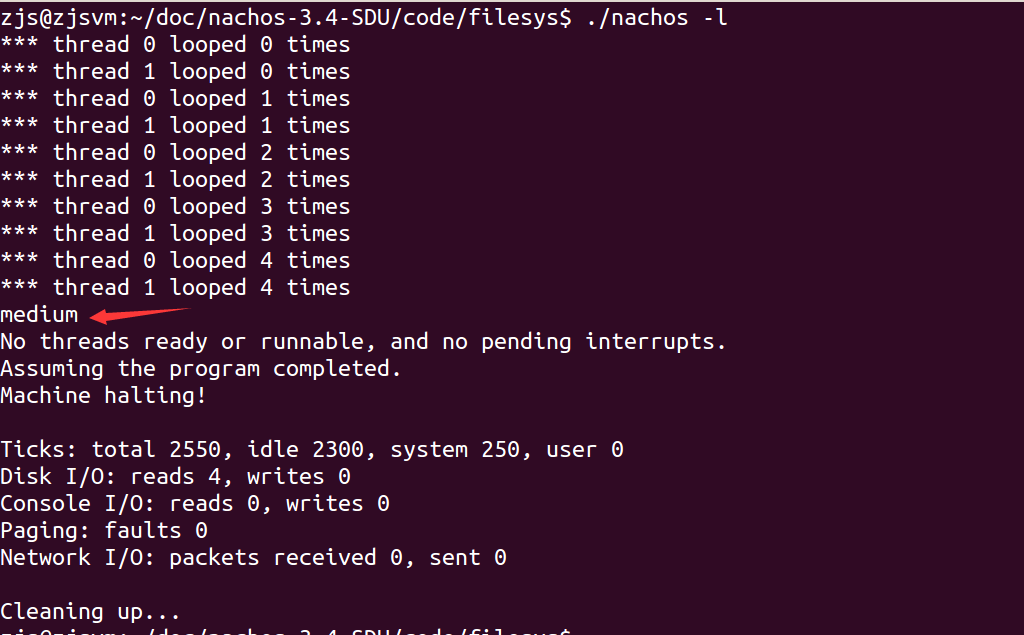
考察位示图数据块、目录表数据块、small 的文件头以及 small 文件内容有哪些变

化;

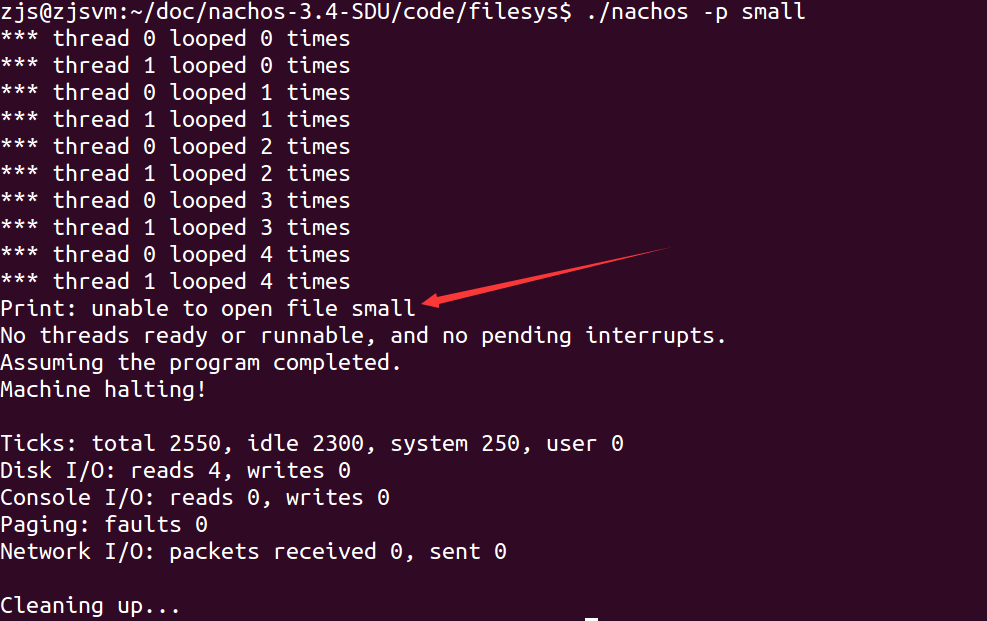


相应的文件都还在，因为这个空间没有被覆盖，但是位图文件已经被更改，标记那一块的内容可以被使用了。

(20)测试 nachos –l(列目录),nachos –p small(显示 small 的内容)等命令;

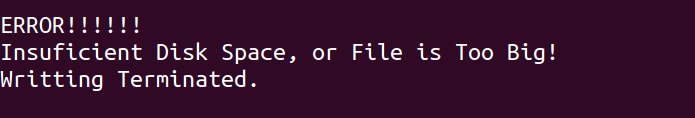


文件只剩下了medium。



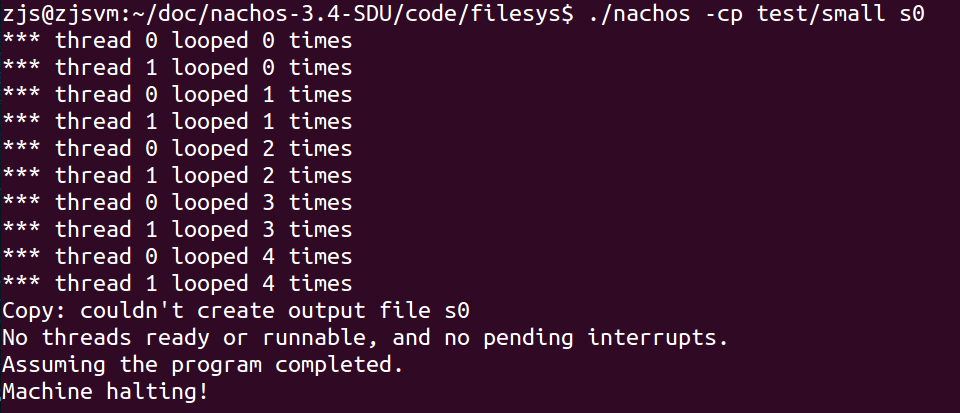
无法打开small文件。Find函数并没有找到文件名为small的文件

(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);



会提示文件太大，没法继续进行复制了。

(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);



目录

[实验六、Nachos 用户程序与系统调用 1](#_Toc71097535)

[一、描述问题： 1](#_Toc71097536)

[二、阐述基本方法： 1](#_Toc71097537)

[三、源代码+注释: 2](#_Toc71097538)

[noff.h 2](#_Toc71097539)

[Segment 2](#_Toc71097540)

[noffHeader 3](#_Toc71097541)

[halt.c 3](#_Toc71097542)

[sort.c 3](#_Toc71097543)

[shell.c 4](#_Toc71097544)

[syscall.h 5](#_Toc71097545)

[addrspace.cc 7](#_Toc71097546)

[progtest.cc 8](#_Toc71097547)

[StartProcess 8](#_Toc71097548)

[Deallocate 9](#_Toc71097549)

[四、实验过程 9](#_Toc71097550)

[1nachos应用程序的编译 9](#_Toc71097551)

# 实验六、Nachos 用户程序与系统调用

## 一、描述问题：

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;

## 二、阐述基本方法：

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;

## 三、源代码+注释:

### noff.h

定义nachos可执行文件文件头相关的数据结构。

#define NOFFMAGIC 0xbadfad  /\* magic number denoting Nachos

           \* object code file

           \*/

目标代码文件的标志。

#### Segment

typedef struct segment {

  int virtualAddr;    段首的虚拟地址

  int inFileAddr;   段首在可执行文件的位置

  int size;     段的大小

} Segment;

#### noffHeader

typedef struct noffHeader {

   int noffMagic;   /\* should be NOFFMAGIC \*/

   Segment code;    代码段

   Segment initData;    已初始化的数据段

   Segment uninitData;    未初始化的数据段

         \* should be zero'ed before use

         \*/

} NoffHeader;

### halt.c

#include "syscall.h"

int

main()

{

    int i,j,k;

    k=3;

    i=2;

    j=j-1;

    k=i-j+k;

    Halt();---系统调用，停机指令。

    /\* not reached \*/

}

### sort.c

#include "syscall.h"

/\* size of physical memory; with code, we'll run out of space!\*/

#define ARRAYSIZE 1024

int A[ARRAYSIZE];

int

main()

{

    int i, j, tmp;

    /\* first initialize the array, in reverse sorted order \*/

    for (i = 0; i < ARRAYSIZE; i++)

        A[i] = ARRAYSIZE - i - 1;

    /\* then sort! \*/

    for (i = 0; i < (ARRAYSIZE - 1); i++)

        for (j = 0; j < ((ARRAYSIZE - 1) - i); j++)

       if (A[j] > A[j + 1]) {   /\* out of order -> need to swap ! \*/

          tmp = A[j];

          A[j] = A[j + 1];

          A[j + 1] = tmp;

           }

    Exit(A[0]);--系统调用，退出

}

### shell.c

实现了一个简单的shell

#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);

    }

    }

}

### syscall.h

void Halt();

/\* Address space control operations: Exit, Exec, and Join \*/

/\* This user program is done (status = 0 means exited normally). \*/

void Exit(int status);

/\* A unique identifier for an executing user program (address space) \*/

typedef int SpaceId;

/\* Run the executable, stored in the Nachos file "name", and return the

 \* address space identifier

 \*/

SpaceId Exec(char \*name);

/\* Only return once the the user program "id" has finished.

 \* Return the exit status.

 \*/

int Join(SpaceId id);

/\* File system operations: Create, Open, Read, Write, Close

 \* These functions are patterned after UNIX -- files represent

 \* both files \*and\* hardware I/O devices.

 \*

 \* If this assignment is done before doing the file system assignment,

 \* note that the Nachos file system has a stub implementation, which

 \* will work for the purposes of testing out these routines.

 \*/

/\* A unique identifier for an open Nachos file. \*/

typedef int OpenFileId;

/\* when an address space starts up, it has two open files, representing

 \* keyboard input and display output (in UNIX terms, stdin and stdout).

 \* Read and Write can be used directly on these, without first opening

 \* the console device.

 \*/

#define ConsoleInput    0

#define ConsoleOutput   1

/\* Create a Nachos file, with "name" \*/

void Create(char \*name);

/\* Open the Nachos file "name", and return an "OpenFileId" that can

 \* be used to read and write to the file.

 \*/

OpenFileId Open(char \*name);

参数：文件名

/\* Write "size" bytes from "buffer" to the open file. \*/

void Write(char \*buffer, int size, OpenFileId id);

将buffer中指定的字节数写入到打开的文件中

/\* 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);

将指定的字节数从文件中读入到buffer里，返回值是实际读入的字节数

/\* Close the file, we're done reading and writing to it. \*/

void Close(OpenFileId id);

/\* User-level thread operations: Fork and Yield.  To allow multiple

 \* threads to run within a user program.

 \*/

/\* Fork a thread to run a procedure ("func") in the \*same\* address space

 \* as the current thread.

 \*/

void Fork(void (\*func)());

/\* Yield the CPU to another runnable thread, whether in this address space

 \* or not.

 \*/

void Yield();

void Halt();

/\* Address space control operations: Exit, Exec, and Join \*/

/\* This user program is done (status = 0 means exited normally). \*/

void Exit(int status);

/\* A unique identifier for an executing user program (address space) \*/

typedef int SpaceId;

/\* Run the executable, stored in the Nachos file "name", and return the

 \* address space identifier

 \*/

SpaceId Exec(char \*name);

 为nachos应用程序name创建一个进程，并且返回系统为其所分配的内存空间里的一个标识（PID）

/\* Only return once the the user program "id" has finished.

 \* Return the exit status.

 \*/

int Join(SpaceId id);

     调用join（pid）进程等待进程pid结束，并且返回pid的退出码

### addrspace.cc

用于存储正在执行的用户程序的地址空间的数据结构。存储虚拟页到内存页的映射关系。

class AddrSpace {

  public:

AddrSpace(OpenFile \*executable); 根据可执行文件构成用户的地址空间

// Create an address space,

          // initializing it with the program

          // stored in the file "executable"

~AddrSpace();   回收地址空间

 // De-allocate an address space

void InitRegisters();初始化用户寄存器组

   // Initialize user-level CPU registers,

          // before jumping to user code

    void SaveState();  在上下文切换时保存   // Save/restore address space-specific

    void RestoreState(); 恢复地址空间信息   // info on a context switch

  private:

    TranslationEntry \*pageTable; 用户程序页表 // Assume linear page table translation

          // for now!

    unsigned int numPages;   用户程序续页数 // Number of pages in the virtual

          // address space

};

#### AddrSpace

为应用程序分配内存，建立页表，将代码的内存和数据都读入到内存里面，

AddrSpace::AddrSpace(OpenFile \*executable)

{

    NoffHeader noffH;

    unsigned int i, size;

    executable->ReadAt((char \*)&noffH, sizeof(noffH), 0);

    if ((noffH.noffMagic != NOFFMAGIC) &&

        (WordToHost(noffH.noffMagic) == NOFFMAGIC))

        SwapHeader(&noffH);

    ASSERT(noffH.noffMagic == NOFFMAGIC);

页表所需要的空间大小，向上整除计算需要多少页

    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size

            + UserStackSize;    // we need to increase the size

                        // to leave room for the stack

代码+初始化的数据段+非初始化的数据段+用户栈的大小

    numPages = divRoundUp(size, PageSize);

    size = numPages \* PageSize;

创建页表并且初始化

    ASSERT(numPages <= NumPhysPages);       // check we're not trying

                        // to run anything too big --

                        // at least until we have

                        // virtual memory

    DEBUG('a', "Initializing address space, num pages %d, size %d\n",

                    numPages, size);

// 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 #

让物理页表和虚拟页表一一对应

    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

    }

// zero out the entire address space, to zero the unitialized data segment

// and the stack segment

将整个地址空间归零，使统一数据段和堆栈段归零

    bzero(machine->mainMemory, size);

// then, copy in the code and data segments into memory

把代码段加载到内存里

    if (noffH.code.size > 0) {

        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",

            noffH.code.virtualAddr, noffH.code.size);

        executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),

            noffH.code.size, noffH.code.inFileAddr);

}

把已经初始化的数据段加载到内存里

    if (noffH.initData.size > 0) {

        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",

            noffH.initData.virtualAddr, noffH.initData.size);

        executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),

            noffH.initData.size, noffH.initData.inFileAddr);

    }

}

#### InitRegisters

用户寄存器的初始化操作

//我们直接将这些写入到“machine”寄存器中，以便我们可以立即跳转到用户代码。注意，当这个线程被上下文切换时，这些内容将被保存/恢复到currentThread->userRegisters中。

void

AddrSpace::InitRegisters()

{

    int i;

每一个用户寄存器初始化为0

    for (i = 0; i < NumTotalRegs; i++)

    machine->WriteRegister(i, 0);

// Initial program counter -- must be location of "Start"

将PC寄存器设置为0

    machine->WriteRegister(PCReg, 0);

    // Need to also tell MIPS where next instruction is, because

// of branch delay possibility

存储下一条指令的地址，因为会有分支指令，一条指令的长度是4字节，所以初始是4

    machine->WriteRegister(NextPCReg, 4);

   // Set the stack register to the end of the address space, where we

   // allocated the stack; but subtract off a bit, to make sure we don't

   // accidentally reference off the end!

设置栈顶指针的位置，栈顶在地址空间的末位再减去16，减去一点，防止越界

    machine->WriteRegister(StackReg, numPages \* PageSize - 16);

    DEBUG('a', "Initializing stack register to %d\n", numPages \* PageSize - 16);

}

### translate.cc

用于管理虚页到实页的转换

class TranslationEntry {

  public:

    int virtualPage;    // The page number in virtual memory.

    虚拟页号

int physicalPage;  物理页号 // The page number in real memory (relative to the

      //  start of "mainMemory"

    bool valid;     有效位    // If this bit is set, the translation is ignored.

      // (In other words, the entry hasn't been initialized.)

    bool readOnly;  是否只读// If this bit is set, the user program is not allowed

      // to modify the contents of the page.

    bool use;   是否被使用过        // This bit is set by the hardware every time the

      // page is referenced or modified.

    bool dirty;   是否被修改过      // This bit is set by the hardware every time the

      // page is modified.

};

### progtest.cc

加载Nachos一个用户程序并执行它。

#### StartProcess

运行用户程序。打开可执行文件，将其加载到内存中，并跳转到它。

void

StartProcess(char \*filename)

{

OpenFile \*executable = fileSystem->Open(filename);

打开一个可执行文件

AddrSpace \*space;

创建一个地址空间

    if (executable == NULL) {

    printf("Unable to open file %s\n", filename);

    return;

    }

    space = new AddrSpace(executable);

currentThread->space = space;

把当前地址的空间设置为这个可执行文件的地址空间，实现了用户线程到核心线程的映射。

    delete executable;          // close file

space->InitRegisters();     // set the initial register values

初始化寄存器

space->RestoreState();      // load page table register

加载页表寄存器

machine->Run();         // jump to the user progam

运行

    ASSERT(FALSE);          // machine->Run never returns;

                    // the address space exits

                    // by doing the syscall "exit"

}

### Machine.cc

#### Run

Void Machine::Run(){

    Instruction \*instr = new Instruction;  // storage for decoded instruction

    if(DebugIsEnabled('m'))

        printf("Starting thread \"%s\" at time %d\n",

           currentThread->getName(), stats->totalTicks);

interrupt->setStatus(UserMode);

把系统的状态设置为用户模式

    for (;;) {

        OneInstruction(instr);读取并解码一条指令

    interrupt->OneTick();CPU前进时钟

    if (singleStep && (runUntilTime <= stats->totalTicks))是否有中断需要处理。

      Debugger();

    }

}

#### Translate

使用页表或TLB将虚拟地址转换为物理地址。检查对齐和各种其他错误，如果一切正常，在转换表项中设置use/dirty位，并将转换后的物理地址存储在“physAddr”中。如果有错误，返回异常的类型。

virtAddr——要转换的虚拟地址

physAddr——存储物理地址的地方

size——正在读取或写入的内存数量

writing——如果为TRUE，检查TLB中的"read-only"位

ExceptionType Machine::Translate(int virtAddr, int\* physAddr, int size, bool writing)

{

    int i;

    unsigned int vpn, offset;

    TranslationEntry \*entry;

    unsigned int pageFrame;

    DEBUG('a', "\tTranslate 0x%x, %s: ", virtAddr, writing ? "write" : "read");

// check for alignment errors

检查地址的对齐是否有错

    if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr & 0x1))){

    DEBUG('a', "alignment problem at %d, size %d!\n", virtAddr, size);

    return AddressErrorException;

    }

    // we must have either a TLB or a page table, but not both!

    ASSERT(tlb == NULL || pageTable == NULL);

    ASSERT(tlb != NULL || pageTable != NULL);

// calculate the virtual page number, and offset within the page,

// from the virtual address

计算虚拟地址的页号以及业内偏移量

    vpn = (unsigned) virtAddr / PageSize;

    offset = (unsigned) virtAddr % PageSize;

    if (tlb == NULL) {      // => page table => vpn is index into table

    if (vpn >= pageTableSize) {

        DEBUG('a', "virtual page # %d too large for page table size %d!\n",

            virtAddr, pageTableSize);

        return AddressErrorException;

    } else if (!pageTable[vpn].valid) {

        DEBUG('a', "virtual page # %d too large for page table size %d!\n",

            virtAddr, pageTableSize);

        return PageFaultException;

    }

    entry = &pageTable[vpn];设置为对应的页表项

    } else {

        for (entry = NULL, i = 0; i < TLBSize; i++)

            if (tlb[i].valid && ((unsigned int)tlb[i].virtualPage == vpn)) {

        entry = &tlb[i];            // FOUND!

        break;

        }

    if (entry == NULL) {                // not found

            DEBUG('a', "\*\*\* no valid TLB entry found for this virtual page!\n");

            return PageFaultException;      // really, this is a TLB fault,

                        // the page may be in memory,

                        // but not in the TLB

    }

    }

若是在只读文件中写入

    if (entry->readOnly && writing) {   // trying to write to a read-only page

    DEBUG('a', "%d mapped read-only at %d in TLB!\n", virtAddr, i);

    return ReadOnlyException;

    }

    pageFrame = entry->physicalPage;

    // if the pageFrame is too big, there is something really wrong!

    // An invalid translation was loaded into the page table or TLB.

    if (pageFrame >= NumPhysPages) {

    DEBUG('a', "\*\*\* frame %d > %d!\n", pageFrame, NumPhysPages);

    return BusErrorException;

    }

    entry->use = TRUE;      // set the use, dirty bits

    if (writing)

    entry->dirty = TRUE;

    \*physAddr = pageFrame \* PageSize + offset;

    ASSERT((\*physAddr >= 0) && ((\*physAddr + size) <= MemorySize));

    DEBUG('a', "phys addr = 0x%x\n", \*physAddr);

    return NoException;

}

### scheduler.cc

#### Run

进行线程上下文切换,首先调用

SaveUserState() SaveState()将进程 PCB 内容保存起来,包括用户寄存器、进程页表。然后调用 SWITCH(oldThread, nextThread)完成内核线程切换。

void

Scheduler::Run (Thread \*nextThread)

{

首先保存当前线程

    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

设置新的线程，标记为执行态

    currentThread = nextThread;         // switch to the next thread

    currentThread->setStatus(RUNNING);      // nextThread is now running

    DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",

      oldThread->getName(), nextThread->getName());

    // This is a machine-dependent assembly language routine defined

    // in switch.s.  You may have to think

    // a bit to figure out what happens after this, both from the point

    // of view of the thread and from the perspective of the "outside world".

切换上下文

    SWITCH(oldThread, nextThread);

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

    // If the old thread gave up the processor because it was finishing,

    // we need to delete its carcass.  Note we cannot delete the thread

    // before now (for example, in Thread::Finish()), because up to this

// point, we were still running on the old thread's stack!

已经完成的线程就给删除

    if (threadToBeDestroyed != NULL) {

        delete threadToBeDestroyed;

    threadToBeDestroyed = NULL;

    }

恢复进程页表的内容

#ifdef USER\_PROGRAM

    if (currentThread->space != NULL) {     // if there is an address space

        currentThread->RestoreUserState();     // to restore, do it.

    currentThread->space->RestoreState();

    }

#endif

}

## 四、实验过程

### nachos应用程序的编译

halt.c---(gcc MIPS交叉编译器)------🡪halt.coff------(../test/coff2noff)-----🡪halt.noff

对于生成Nachos可执行程序这一过程，编译的目标文件是（.noff）文件和（.flat）文件，编译过程是先通过（.c）文件编译出（.o）文件，再通过（.o）文件编译出（.coff）文件，再通过（.coff）文件编译出（.noff）文件与（.flat）文件；

### Nachos 应用程序与可执行程序

阅读../bin/noff.h,分析 Nachos 可执行程序.noff 文件的格式组成;

### Nachos 可执行程序格式

定义nachos可执行文件文件头相关的数据结构。

#define NOFFMAGIC 0xbadfad  /\* magic number denoting Nachos

           \* object code file

           \*/

目标代码文件的标志。

Segment

typedef struct segment {

  int virtualAddr;    段首的虚拟地址

  int inFileAddr;   段首在可执行文件的位置

  int size;     段的大小

} Segment;

noffHeader

typedef struct noffHeader {

   int noffMagic;   /\* should be NOFFMAGIC \*/

   Segment code;    代码段

   Segment initData;    已初始化的数据段

   Segment uninitData;    未初始化的数据段

         \* should be zero'ed before use

         \*/

} NoffHeader;

### 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。

Nachos 为该程序分配了 11 个实页



目录

[实验七、地址空间的扩展 1](#_Toc72919703)

[一、描述问题： 1](#_Toc72919704)

[二、阐述基本方法： 2](#_Toc72919705)

[三、源代码+注释: 2](#_Toc72919706)

[AddrSpace.cpp 2](#_Toc72919707)

[AddrSpace 3](#_Toc72919708)

[~AddrSpace 4](#_Toc72919709)

[实验八、系统调用 Exec()与 Exit() 5](#_Toc72919710)

[一、描述问题： 5](#_Toc72919711)

[二、阐述基本方法： 5](#_Toc72919712)

[三、源代码+注释: 6](#_Toc72919713)

[Exec（）： 6](#_Toc72919714)

[Join（）与Exit（）： 10](#_Toc72919715)

[Yield（） 16](#_Toc72919716)

[基于FILESYS\_STUB实现文件的有关系统调用 16](#_Toc72919717)

[Create 17](#_Toc72919718)

[Open 17](#_Toc72919719)

[Write 18](#_Toc72919720)

[Read 19](#_Toc72919721)

[Close 19](#_Toc72919722)

[四、实验过程 20](#_Toc72919723)

[1. 对Exec（）、Exit（）、Yield（）、Join（）进行综合测试 20](#_Toc72919724)

[2. 基于FILESYS\_STUB实现文件的有关系统调用 21](#_Toc72919725)

# 实验七、地址空间的扩展

## 一、描述问题：

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;

## 二、阐述基本方法：

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。

## 三、源代码+注释:

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。

所以定义全局变量bitmap，使用位图来管理帧。每次要分配物理帧时，使用bitmap->Find()函数来寻找一个可用的物理帧进行分配。

### AddrSpace.cpp

加入声明 static BitMap \*userMap，pidMap以及spaceId。

 private:

    static BitMap \*userMap, \*pidMap;         // 全局位图

    TranslationEntry \*pageTable;  // 线性页表

    unsigned int numPages, spaceId;       // 页表中的页表项以及地址编号

NumPhysPages表示最大物理页，即32。

BitMap \*AddrSpace::userMap = new BitMap(NumPhysPages);

BitMap \*AddrSpace::pidMap = new BitMap(MAX\_USERPROCESSES);

#### AddrSpace

在AddrSpace类中通过userMap对象用于管理空闲内存页表，修改虚实页分配的部分代码，对于每一个虚页，在位图中找一个未被分配的页面作为虚页映射，同时保证在分配之前物理内存中空闲页面数量大于等于需要的页面数量。

分配完虚实页映射后，我们需要将noff文件中的数据拷贝到machine的物理内存中，因此需要将虚拟地址所对应的物理地址求出，需要先定位具体的物理页面，再根据页面偏移量定义具体物理地址。

注释bzero。

AddrSpace::AddrSpace(OpenFile \*executable) {

    ASSERT(pidMap->NumClear() >= 1);  // 保证还有线程号可以分配

    spaceId = pidMap->Find()+100;     // 0-99留给内核线程

    // 可执行文件中包含了目标代码文件

    NoffHeader noffH;               // noff文件头

    unsigned int i, size;

    executable->ReadAt((char \*)&noffH, sizeof(noffH), 0);   // 读出noff文件

    if ((noffH.noffMagic != NOFFMAGIC) && (WordToHost(noffH.noffMagic) == NOFFMAGIC))

        SwapHeader(&noffH);           // 检查noff文件是否正确

    ASSERT(noffH.noffMagic == NOFFMAGIC);

    // 确定地址空间大小，其中还包括了用户栈大小

    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size + UserStackSize;

    numPages = divRoundUp(size, PageSize);  // 确定页数

    size = numPages \* PageSize;             // 计算真实占用大小

    ASSERT(numPages <= NumPhysPages);       // 确认运行文件大小可以运行

    DEBUG('a', "Initializing address space, num pages %d, size %d\n", numPages, size);

    // 第一步，创建页表，并对每一页赋初值

    pageTable = new TranslationEntry[numPages];

    ASSERT(userMap->NumClear() >= numPages);    // 确认页面足够分配

    for (i = 0; i < numPages; i++) {

      pageTable[i].virtualPage = i;         // 虚拟页

      pageTable[i].physicalPage = userMap->Find();  // 在位图找空闲页进行分配

      pageTable[i].valid = TRUE;

      pageTable[i].use = FALSE;

      pageTable[i].dirty = FALSE;

      pageTable[i].readOnly = FALSE;        // 只读选项

    }

    // 第二步，将noff文件数据拷贝到物理内存中

    if (noffH.code.size > 0) {

      int pagePos = pageTable[noffH.code.virtualAddr/PageSize].physicalPage \* PageSize;

      int offSet = noffH.code.virtualAddr % PageSize;

      executable->ReadAt(&(machine->mainMemory[pagePos+offSet]),

        noffH.code.size, noffH.code.inFileAddr); // ReadAt调用了bcopy函数

    }

    if (noffH.initData.size > 0) {

      int pagePos = pageTable[noffH.initData.virtualAddr/PageSize].physicalPage \* PageSize;

      int offSet = noffH.initData.virtualAddr % PageSize;

      executable->ReadAt(&(machine->mainMemory[pagePos+offSet]),

                noffH.initData.size, noffH.initData.inFileAddr);

    }

#ifdef FILESYS

    for(int i = 3; i < 10; i++) fileDescriptor[i] = NULL;

    OpenFile \*StdinFile = new OpenFile("stdin");

    OpenFile \*StdoutFile = new OpenFile("stdout");

    fileDescriptor[0] = StdoutFile;

    fileDescriptor[1] = StdoutFile;

    fileDescriptor[2] = StdoutFile;

#endif

}

#### ~AddrSpace

修改AddrSpace类的析构函数，需要在AddrSpace析构的时候将对应的物理页面释放

AddrSpace::~AddrSpace()

{

  pidMap->Clear(spaceId-100);

  for(int i = 0; i < numPages; i++)

    userMap->Clear(pageTable[i].physicalPage);

  delete [] pageTable;

}

# 实验八、系统调用 Exec()与 Exit()

## 一、描述问题：

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;

## 二、阐述基本方法：

(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。

## 三、源代码+注释:

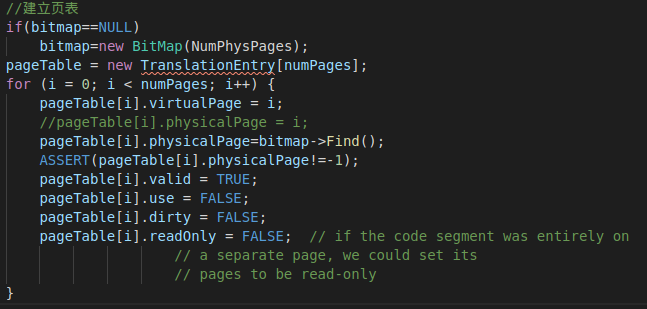
### Exec（）：

系统调用 pid=Exec(filename)的功能是加载运行应用程序 filename

（1）修改AddrSpace，使用位图来分配物理帧，使用数组来分配pid。

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，然后根据文件头中各段的大小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（为简单器件，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）。

因此，为了管理空闲帧，建立一个全局的空闲帧管理位示图。每次要分配物理帧时，使用bitmap->Find()函数来寻找一个可用的物理帧进行分配。



接下来考虑如何管理pid：该值涉及两个问题，它是如何产生的，以及如何在内核中记录它，以便Join（）能够通过该值找到对应的线程。

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。从函数StartProcess()中可以看出，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，理论上课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没有实现子进程或者子线程的概念（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=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。

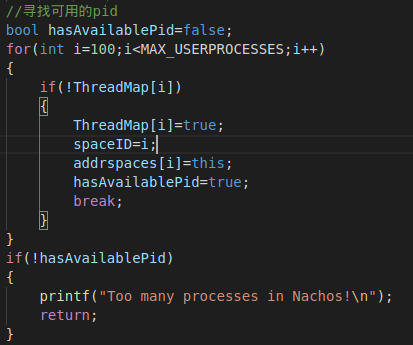
当一个程序调用系统调用Exit（）退出时，应当收回为其分配的pid号，以分配给后续的应用程序，同时应释放其所占用的内存空间及页表等信息，供后续进程使用。

因此，为了管理pid，建立一个全局的pid数组。从内存的第一个空闲帧为Exec(filename)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。

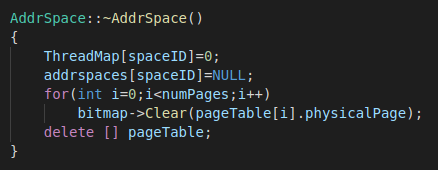
同时要记录pid与AddrSpace的映射关系，以便void StartProcess(int spaceID)使用。



在AddrSpace的构造函数中，为其分配pid。



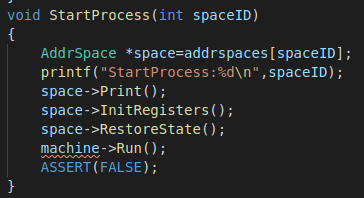
在AddrSpace的析构函数中，回收pid，清除空闲帧位示图相应的标志，释放页表。



（2）重载StartProcess(int spaceID)，作为新建线程执行的代码。

系统调用Exec（）需要创建一个核心现场才能，并将该进程与新建的核心线程关联。通过Thread::Fork（）创建的线程需要指明该线程要执行的函数及函数所需的参数。因此我们可以重载函数void 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，供其他系统调用（如Join（））使用。

void StartProcess(int spaceID)函数根据AddrSpace.cc中定义的全局变量addrspaces[]（pid与AddrSpace的映射）来获得该spaceID对应的地址空间space。之后初始化用户寄存器和系统页表，使用machine->Run()函数开始执行用户程序。



因为addrspaces数组是在AddrSpace.cc中定义的，这里使用extern声明数组。



（3）在void ExceptionHandler()中添加SC\_Exec的代码。

我们需要读出寄存器或者内存中的内容。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。

所以在Exec（）的实现代码中可以利用ReadRegister(4)从$4中获取参数的内存地址，然后利用ReadMem（）获取filename，然后为其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Exec（）执行结束后，需要返回该线程的pid，可以利用WriteRegister(2，spaceID)将pid写入到2号寄存器中。MIPS架构用寄存器2和3存放返回值。

case SC\_Exec:{

                printf("This is SC\_Exec, CurrentThreadId: %d\n",(currentThread->space)->getSpaceId());

                //4号寄存器，存储的是参数在内存中的地址

                int addr = machine->ReadRegister(4);

                char filename[50];

                //从内存中读取文件名

                for(int i = 0; ; i++){

                    machine->ReadMem(addr+i,1,(int \*)&filename[i]);

                    if(filename[i] == '\0') break;

                }

                OpenFile \*executable = fileSystem->Open(filename);

                if(executable == NULL) {

                    printf("Unable to open file %s\n",filename);

                    return;

                }

                // 建立新地址空间

                AddrSpace \*space = new AddrSpace(executable);

                // space->Print();   // 输出新分配的地址空间

                delete executable;  // 关闭文件

                // 建立新核心线程

                Thread \*thread = new Thread(filename);

                printf("new Thread, SpaceId: %d, Name: %s\n",space->getSpaceId(),filename);

                // 将用户线程映射到核心线程上

                thread->space = space;

                thread->Fork(StartProcess,(int)space->getSpaceId());

                machine->WriteRegister(2,space->getSpaceId());

                AdvancePC();

                break;

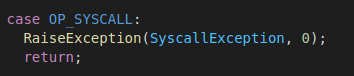
            }

这里要用到StartProcess（）函数，应在之前使用extern声明。



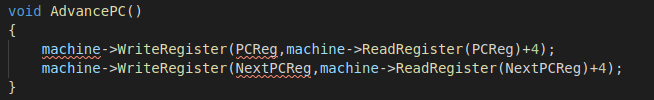
（4）执行完系统调用后，要手动增加PC。

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。其中对Nachos系统调用的处理如下，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。



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中使用的return语句，没有增加PC。该原因是通常情况下，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return语句改为break语句，另一种是在系统调用处理程序中添加PC的推进操作。这里使用第二种方法，在exception.cc模块中添加AdvancePC()函数，并在各系统调用处理程序最后调用函数AdvancePC()。



（5）总结：

获取Exec的参数filename：通过读取$4寄存器获得filename在内存中的地址，然后从该地址中读出参数filename。

打开应用程序filename：类似StartProcess（）函数。

为应用程序分配内存空间及栈并为其建立页表，读入应用程序代码及数据。

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。对于应用程序filename，需要我们自己调用Thread::Fork（）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。

初始化寄存器，开始程序执行。

返回进程的pid。这里放在2号寄存器中。

增加PC。

### Join（）与Exit（）：

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(SpaceIdid)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)。

该系统调用用于一个父线程（Joiner）等待一个子线程（Joinee）运行结束后再继续运行，常用于同步设计中。

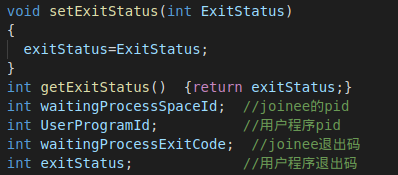
系统调用 void Exit(int status)的参数 status 是用户程序的退出状态。系统调用 int Join(SpaceId id)需要返回该退出状态 status。由于可能在 id 结束之后,其它程序(如 parent)才调用 Join(SpaceId id),因此在 id 执行 Exit(status)退出时需要将 id 的退出码 ststus 保存起来,以备 Join()使用。

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。

（1）在Thread类中增加 pid与退出码。

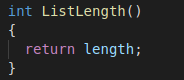
系统调用void Exit(int status)的参数status是用户程序的退出状态。系统调用int Join(spaceID id)需要返回该退出状态status。由于可能在id结束之后，其他程序才调用Join()，因此在id执行Exit（）退出时需要将id的退出码保存起来，以备Join（）使用。



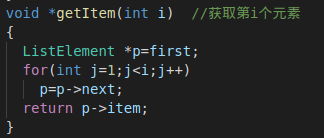
（2）修改List类，记录长度，并增加操作函数。

私有变量int length记录长度。在构造函数中初始化为0，在Append（）、Prepend（）和SortedInsert（）函数中将其加一，在SortedRemove（）函数中将其减一。

int ListLength（）函数：返回长度length。



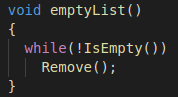
void \*getItem（int i）：获取第i个元素。从1开始。



void RemoveByItem（void \*item）：删掉item。

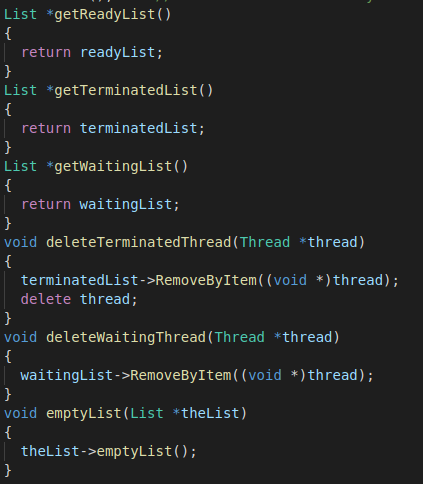


void emptyList（）：清空List。



（3）在Scheduler类中增加线程等待队列waitingList和线程终止队列terminatedList。

在构造函数中将两个队列进行初始化。同时要增加对队列的操作：获得readyList、waitingList、terminatedList，在waitingList中删除一个线程，在terminatedList中删除一个线程，以及清空某队列。



（4）修改Thread类，增加TERMINATED状态。修改Finish（）函数，增加Terminated（）函数和Join（）函数。

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？

线程再执行过程中，可能处于执行、就绪及阻塞状态，当线程执行信号量的P（）操作可能进入阻塞，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，则从队列中移除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。

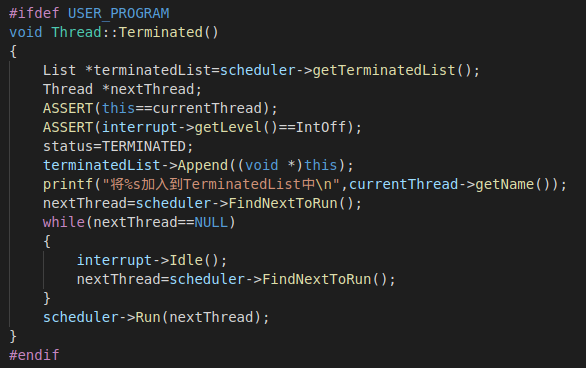
当Joiner被唤醒后，需要从terminated队列移除Joinee并将其销毁，然后返回Joinee的退出码。



Finish（）函数：获取等待队列waitingList，判断当前线程是否有Joiner在waitingList中等待它执行完毕。如果waitingList中的waitingThread的等待线程id waitingProcessSpaceId与当前线程的spaceId相同，则通过scheduler->ReadyToRun（）函数唤醒waitingThread，然后将其从waitingList中删除。最后将当前线程终止，即调用Terminated（）函数。



Terminated（）函数：类似于Sleep（）函数。设置当前线程状态为TERMINATED，然后将线程加入到terminatedList末尾，获取下一个线程nextThread并运行。



Thread：：Join（int spaceId）函数：关中断，将joinee的spaceId记录下来，放在currentThread->waitingProcessSpaceId中。接下来判断joinee是否在terminatedList中，如果不在，则此时joinee仍在就绪/阻塞，将当前线程加入到waitingList中，然后调用Sleep（）函数进行睡眠。等到执行Sleep（）之后的代码时，joinee已经结束了（要么Sleep（）被唤醒，要么joinee早已结束，在terminatedList中），此时joinee一定在terminatedList中，将其找到，在currentThread->waitingProcessExitCode中记录下joinee的退出码，然后将其从terminatedList中删除。

void Thread::Join(int SpaceId) {

    IntStatus oldLevel = interrupt->SetLevel(IntOff);       // 关中断

    waitProcessSpaceId = SpaceId;                        // 设置当前线程所等待进程的spaceId

    List \*terminatedList = scheduler->getTerminatedList();  // 终止队列

    List \*waitingList = scheduler->getWaitingList();        // 等待队列

    // 确定Joinee在不在终止队列中

    bool interminatedList = FALSE;

    ListElement \*first = terminatedList->listFirst(); // 队列首

    while(first != NULL){

        Thread \*thread = (Thread \*)first->item;     // 强转成Thread指针

        if(thread->userProgramId() == SpaceId){       // 在队列中

            interminatedList = TRUE;

            waitProcessExitCode = thread->ExitCode();  // 设置父线程等待子线程退出码

            break;

        }

        first = first->next;

    }

    // Joinee不在终止队列中, 可运行态或阻塞态

    if(!interminatedList){

        waitingList->Append((void \*)this);  // 阻塞Joiner

        currentThread->Sleep();             // Joiner阻塞

    }

    // 被唤醒且Joinee在终止队列中，在终止队列中删除Joinee

    scheduler->deleteTerminatedThread(SpaceId);

    (void) interrupt->SetLevel(oldLevel);   // 开中断

}

（5）在void ExceptionHandler()中添加SC\_Exit的代码。

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，其他程序才调用Join（spaceID id），因此在id执行Exit（status）退出时需要将id的退出码status保存起来，以备Join（）使用。

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页表，释放分配给该进程的实页（帧），释放其pid（参见AddrSpace的析构函数），调用Finish（）结束该进程对应的线程。最后要把退出码写进2号寄存器中。

Join（）的实现带来的一个问题是，那些没有被Join（）等待的线程何时从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。临时解决方案：为父进程设置特殊的退出码99，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。

case SC\_Exit:{

                printf("This is SC\_Exit, CurrentThreadId: %d\n",(currentThread->space)->getSpaceId());

                int exitCode = machine->ReadRegister(4);

                machine->WriteRegister(2,exitCode);

                currentThread->setExitCode(exitCode);

                // 父进程的退出码特殊标记，由 Join 的实现方式决定

                if(exitCode == 99)

                    scheduler->emptyList(scheduler->getTerminatedList());

                delete currentThread->space;

                currentThread->Finish();

                AdvancePC();

                break;

            }

（6）在void ExceptionHandler()中添加SC\_Join的代码。

首先从4号寄存器中读出参数spaceID，然后调用Join（spaceID）函数。执行完毕后将joinee的退出码写到2号寄存器中，最后增加PC即可。

case SC\_Join:{

                printf("This is SC\_Join, CurrentThreadId: %d\n",(currentThread->space)->getSpaceId());

                int SpaceId = machine->ReadRegister(4);

                currentThread->Join(SpaceId);

                // waitProcessExitCode —— 返回 Joinee 的退出码

                machine->WriteRegister(2, currentThread->waitExitCode());

                AdvancePC();

                break;

            }

### Yield（）

将CPU让给另一个可运行线程，不管是在这个地址空间还是不是。

调用Thread::Yield（）实现。增加PC。

该系统调用用于一个线程从运行态变成可运行态，将 CPU 调度给另一个线程执行，具体实现代码如下。

case SC\_Yield:{

                printf("This is SC\_Yield, CurrentThreadId: %d\n",(currentThread->space)->getSpaceId());

                currentThread->Yield();

                AdvancePC();

                break;

            }

### 基于FILESYS\_STUB实现文件的有关系统调用

目前设计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：

Nachos的文件系统有两个版本，基于宏FILESYS\_STUB与FILESYS进行条件编译，使用宏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。

#### Create

case SC\_Create:{

                int addr = machine->ReadRegister(4);

                char filename[128];

                for(int i = 0; i < 128; i++){

                    machine->ReadMem(addr+i,1,(int \*)&filename[i]);

                    if(filename[i] == '\0') break;

                }

                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;

            }

#### Open

case SC\_Open:{

                int addr = machine->ReadRegister(4);

                char filename[128];

                for(int i = 0; i < 128; i++){

                    machine->ReadMem(addr+i,1,(int \*)&filename[i]);

                    if(filename[i] == '\0') break;

                }

                int fileDescriptor = OpenForWrite(filename);

                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,fileDescrip tor);

                AdvancePC();

                break;

            }

#### Write

case SC\_Write:{

                // 读取寄存器信息

                int addr = machine->ReadRegister(4);

                int size = machine->ReadRegister(5);       // 字节数

                int fileId = machine->ReadRegister(6);      // fd

                // 打开文件

                OpenFile \*openfile = new OpenFile(fileId);

                ASSERT(openfile != NULL);

                // 读取具体数据

                char buffer[128];

                for(int i = 0; i < size; i++){

                    machine->ReadMem(addr+i,1,(int \*)&buffer[i]);

                    if(buffer[i] == '\0') break;

                }

                buffer[size] = '\0';

                // 写入数据

                int writePos;

                if(fileId == 1) writePos = 0;

                else writePos = openfile->Length();

                // 在 writePos 后面进行数据添加

                int writtenBytes = openfile->WriteAt(buffer,size,writePos);

                if(writtenBytes == 0) printf("write file failed!\n");

                else printf("\"%s\" has wrote in file %d succeed!\n",buffer,fileId);

                AdvancePC();

                break;

            }

#### Read

case SC\_Read:{

                // 读取寄存器信息

                int addr = machine->ReadRegister(4);

                int size = machine->ReadRegister(5);       // 字节数

                int fileId = machine->ReadRegister(6);      // fd

                // 打开文件读取信息

                char buffer[size+1];

                OpenFile \*openfile = new OpenFile(fileId);

                int readnum = openfile->Read(buffer,size);

                for(int i = 0; i < size; i++)

                    if(!machine->WriteMem(addr,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;

            }

#### Close

case SC\_Close:{

                int fileId = machine->ReadRegister(4);

                Close(fileId);

                printf("File %d closed succeed!\n",fileId);

                AdvancePC();

                break;

            }

## 四、实验过程

### 1. 对Exec（）、Exit（）、Yield（）、Join（）进行综合测试

修改join.c文件

#include "syscall.h"

int main() {

    SpaceId newProc = Exec("../test/yield.noff");

    Join(newProc);

    Exit(0);

}

Yield.c的内容是

#include "syscall.h"

int main() {

    Exec("../test/exit.noff");

    Yield();

    Exit(0);

}

Exit.c的内容是

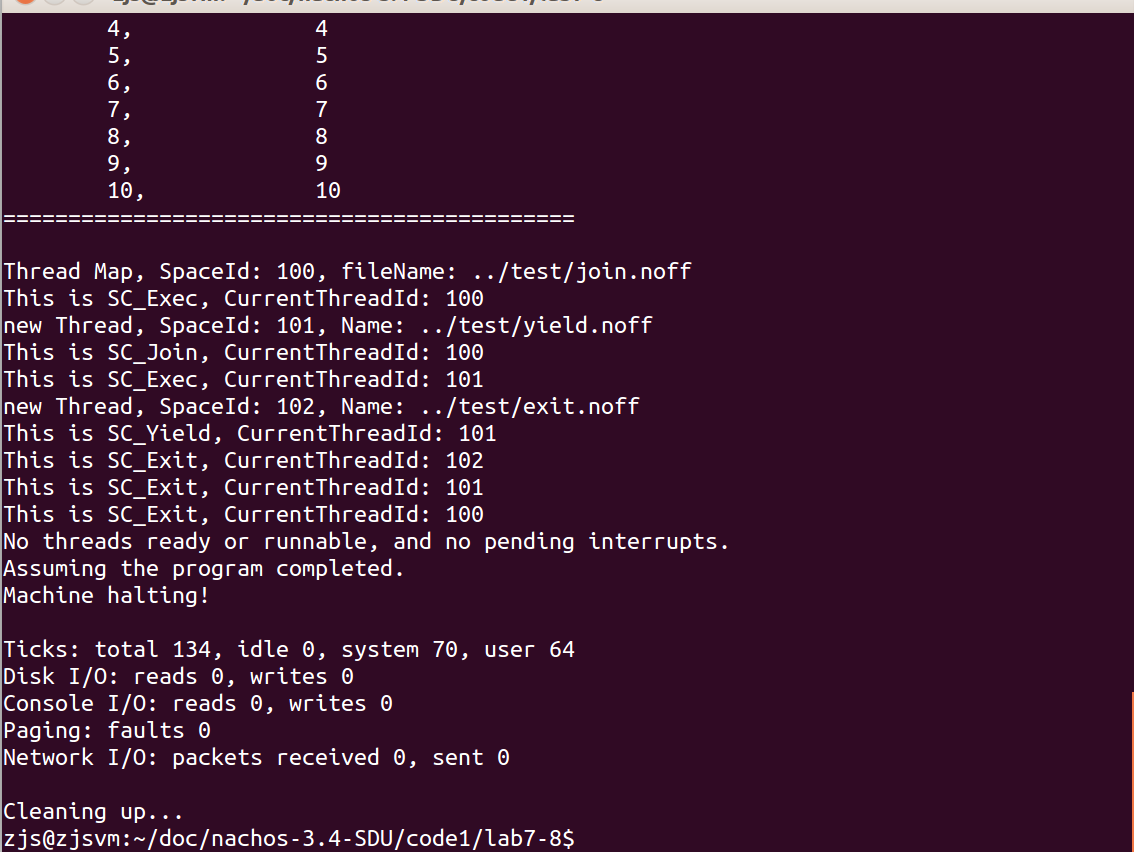
#include "syscall.h"

int main() {

    Exit(1);

}

实验结果：



1. 将join.noff用户程序映射到main线程中，分配SpaceId=100
2. main线程执行Exec（../test/yield.noff）系统调用，并产生新用户程序../test/yield.noff,并分配SpaceId=101。
3. main线程继续执行Join（）系统调用，等待../test/yield.noff执行完毕，因此此时调度到线程../test/yield.noff上执行
4. ../test/yield.noff线程执行Exec（../test/exit.noff）系统调用，并产生新用户程序../test/exit.noff，并分配SpaceId=102
5. ../test/yield.noff继续线程执行Yield（）系统调用，此时线程调度到../test/exit.noff上继续执行，此时没有调度到main线程上继续执行，证明main线程此时在等待队列中，而非可运行态队列，证实了Join系统调用正确执行。
6. ../test/exit.noff执行Exit（）系统调用，成为第一个结束的用户程序，接下来调度到../test/join.noff线程上继续执行
7. ../test/join.noff执行Exit（）系统调用，成为第二个人结束的用户程序，由于../test/join.noff运行结束，因此重新调度到main上执行
8. main线程执行Exit（）系统调用，成为第三个结束的用户程序，join.noff程序运行结束。

### 2. 基于FILESYS\_STUB实现文件的有关系统调用

#include "syscall.h"

int main() {

    OpenFileId Fp;

    char buffer[50];    // 读出的数据

    int sz;             // 读出数据大小

    // 创建文件

    Create("Ftest");

    // 打开文件

    Fp = Open("Ftest");

    // 写文件

    Write("hello nachos!",10,Fp);

    // 读文件

    sz = Read(buffer,6,Fp);

    // 关闭文件

    Close(Fp);

    // 退出程序

    Exit(0);

}

1. 文件创建成功，分配的field为3，Create（）系统调用实现正确。
2. 文件打开成功，Open（）系统调用实现正确。
3. 成功写入10个字节到文件中，期待写入的字符串为hello nachos!，但是最终只写入了10个字节，说明Write（）系统调用实现正确。
4. 从文件中读出6个字节，即“labos ”，Read（）系统调用实现正确。
5. 文件成功关闭，Close（）系统调用实现正确。

除上述终端输出，查看../lab7-8可看出Ftest文件，打开文件，可以看到写入的数据

