**山东大学操作系统课程设计实验报告**

**姓名：杜瀛川**

**学号：201700301147**

**学院：计算机科学与技术学院**

**专业：计算机科学与技术专业**

**班级：4班**

**日期：2020.5**

目录

[一、 摘要 3](#_Toc24603)

[实验一 Nachos系统的安装与调试 3](#_Toc2163)

[工作总结 3](#_Toc8256)

[实验三 利用信号量实现线程同步 3](#_Toc15625)

[工作总结 3](#_Toc13843)

[实验四 实验五 Nachos的文件系统 3](#_Toc10485)

[工作总结 3](#_Toc4542)

[特色 3](#_Toc16581)

[实验六 实验七 实验八 Nachos用户程序与系统调用 3](#_Toc11900)

[工作总结 3](#_Toc4128)

[特色 4](#_Toc22858)

[二、正文 5](#_Toc10300)

[1 实验一 Nachos系统的安装与调试 5](#_Toc7687)

[1.1 Nachos系统的安装与调试过程 5](#_Toc12018)

[1.2 Nachos系统的安装与调试过程中的思考 9](#_Toc5817)

[3 实验三 利用信号量实现线程同步 11](#_Toc31615)

[3.1 基础实验代码分析 11](#_Toc30158)

[3.2 基础实验设计任务 17](#_Toc22062)

[3.3 基础实验调试结果 20](#_Toc11155)

[4 实验四 Nachos的文件系统 20](#_Toc23174)

[4.1 基础实验代码分析 20](#_Toc18488)

[4.2 基础实验调试结果 28](#_Toc14255)

[5 实验五 Nachos的文件系统扩展 35](#_Toc1814)

[5.1 文件扩展设计任务 35](#_Toc10534)

[5.2 文件扩展扩展调试结果 40](#_Toc5404)

[5.3 多级目录设计任务 41](#_Toc4605)

[5.4 多级目录调试结果 59](#_Toc29566)

[6 实验六 Nachos用户程序与系统调用 63](#_Toc14141)

[6.1 基础实验代码分析 63](#_Toc8489)

[6.2 打印进程页表设计任务 69](#_Toc14878)

[6.3 打印进程页表调试结果 69](#_Toc18564)

[7 实验七 地址空间的扩展 70](#_Toc14640)

[7.1 地址空间扩展设计任务 70](#_Toc23685)

[8 实验八 系统调用 72](#_Toc30639)

[8.1 Exec()设计任务 72](#_Toc10021)

[8.2 Join() Exit()设计任务 75](#_Toc9367)

[8.3 Yield()设计任务 81](#_Toc7348)

[8.4 Create() Open() Write() Read() Close()设计任务 82](#_Toc26419)

[8.5 Nachos用户程序与系统调用测试程序 87](#_Toc8868)

[8.6 Nachos用户程序与系统调用调试结果 92](#_Toc17718)

[三、总结 98](#_Toc11708)

[四、参考文献 99](#_Toc22186)

1. **摘要**

**实验一 Nachos系统的安装与调试**

**工作总结**

1、完成了Nachos和GCC的安装和调试。

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

**实验三 利用信号量实现线程同步**

**工作总结**

1、分析了main.cc system.cc utility.cc interrupt.h timer.h threads.h

switch.s scheduler.h synch.h ring.h Threadtest prodcons.cc等文件中的代码。

2、在prodcons.cc中加入ProdCons() Producer() Consumer()实现了生产者消费者模型。利用信号量实现线程同步。完成了相关调试工作。

**实验四 实验五 Nachos的文件系统**

**工作总结**

1. 分析了mian.cc disk.h synchDisk.h bitmap.h filehdr.h openfile.h directory.h filesys.h fstest.cc等文件中的代码。

2、通过修改OpenFile类 FileSystem类 FileHeader类实现了文件的大小可扩展性。可实现写入过程中文件大小的增长。实现了ap nap hap等shell命令。

3、通过修改DirectoryEntry类 FileSystem类 Directory类 fstest.cc main.cc，增加CurDir类实现多级目录。实现相对路径和绝对路径的处理。实现了当前路径的变更。实现了级联删除。实现了cd pwd ls mkdir cat rm rmdir cp rmdir-r等shell命令。

4、完成了相关调试工作。

**特色**

1. 实现了文件的大小可扩展性。可实现写入过程中文件大小的增长。
2. 实现了ap nap hap等shell命令。
3. 实现了多级目录。
4. 实现相对路径和绝对路径的处理。
5. 实现了当前路径的变更。
6. 实现了级联删除。
7. 实现了cd pwd ls mkdir cat rm rmdir cp rmdir-r等shell命令。

**实验六 实验七 实验八 Nachos用户程序与系统调用**

**工作总结**

1. 分析了machine.h translate.h noff.h mipssim.h addrSpace.h thread.h schedule.h syscall.h start.s progtest.cc exception.cc等文件中的代码。
2. 通过AddrSpace类添加Print()方法实现进程页表打印。
3. 通过修改system.cc AddrSpace类实现进程地址空间扩展，实现多道程序。
4. 通过修改AddrSpace类 progtest.cc exception.cc，增加Pcb类实现Exec()系统调用。

5、通过修改Pcb类 AddrSpace类 List类 Scheduler类 Thread类 system.cc exception.cc实现Join() Exit()系统调用。

6、通过修改exception.cc实现Yield()系统调用。

7、shell脚本除运行noff可执行文件外，还实现了ls等shell命令。

8、通过修改Pcb类 exception.cc实现Create() Open() Write() Read() Close()系统调用。主要实现了进程打开文件表。

9、编写sh.noff read.noff exec.noff yiel.noff halt.noff作为测试文件，拷贝到Nachos文件系统中。

10、完成了相关调试工作。

**特色**

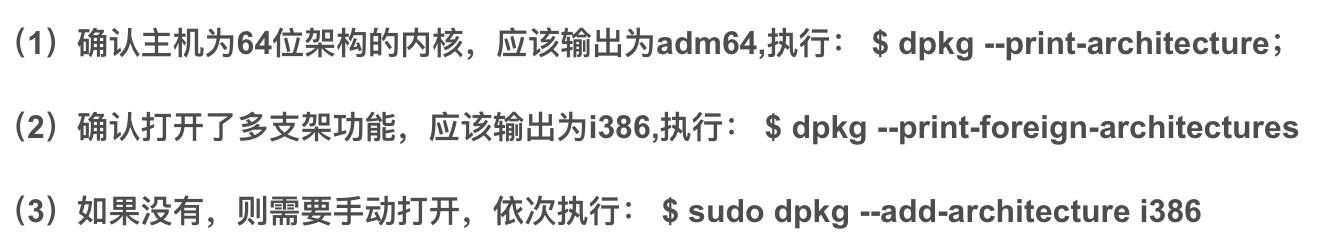
1. 实现了Halt() Exec() Join() Exit() Yield() Create() Open() Write() Read() Close()系统调用。
2. 编写sh.noff read.noff exec.noff yiel.noff halt.noff作为测试文件，实现了上述系统调用的全面测试。
3. noff文件拷贝到Nachos文件系统中。在Nachos文件系统多级目录中完成调试工作。
4. sh.noff作为shell脚本程序，除运行noff可执行文件外，还实现了ls等shell命令。
5. Pcb类的加入使进程的概念更加清晰、整体。
6. 实现了进程打开文件表、进程文件描述符。

**二、正文**

**1 实验一 Nachos系统的安装与调试**

**1.1 Nachos系统的安装与调试过程**

**1.1.1 检查系统是否支持多架构**



**1.1.2 安装32位编译环境与支持库**

如果未安装gcc及g++ 编译器，请提前安装它们

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

**1.1.3 解压 Nachos 源码**

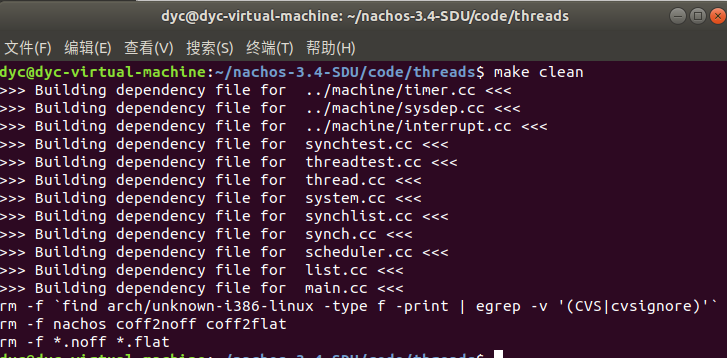
将nachos-3.4-SDU-64-.tar.gz文件复制到你所希望的目录中

tar -xzvf nachos-3.4-SDU-64.tar.gz

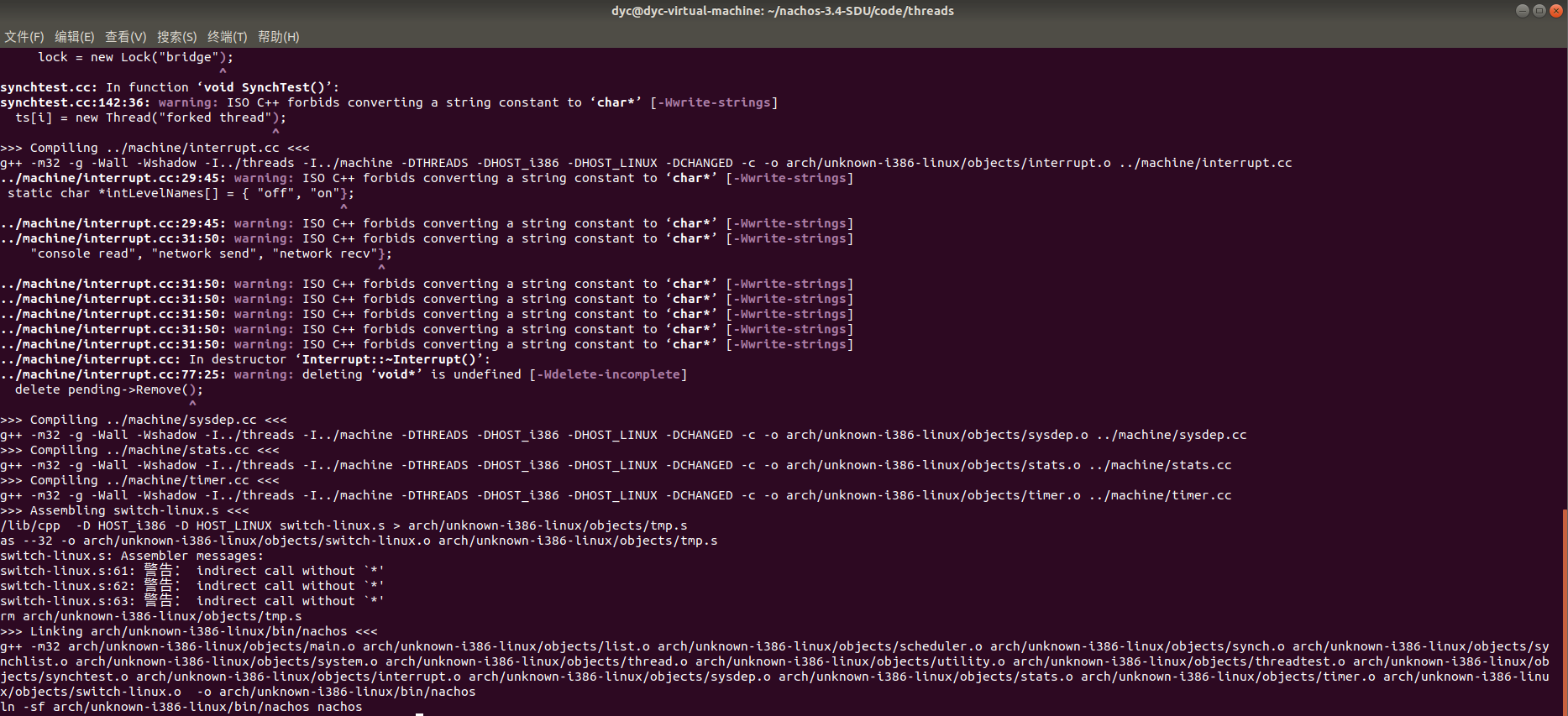
**1.1.4 进入目录 ../nachos-3.4-SDU/code/threads进行测试**

cd nachos-3.4-SDU-64/code/threads

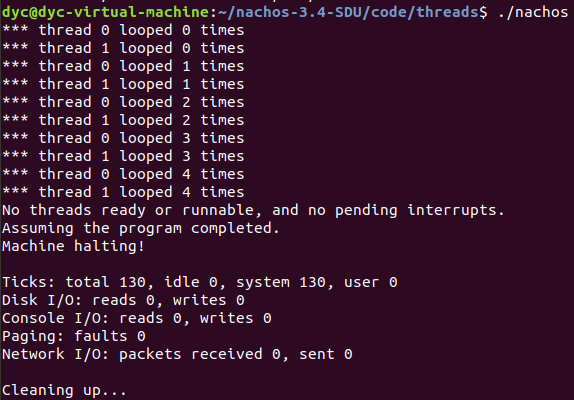
Make clean



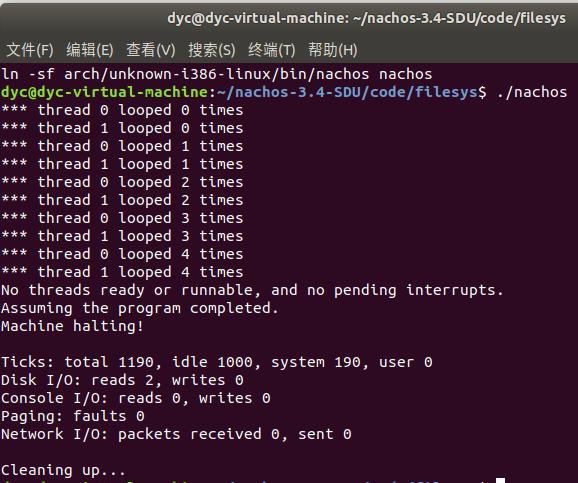
Make

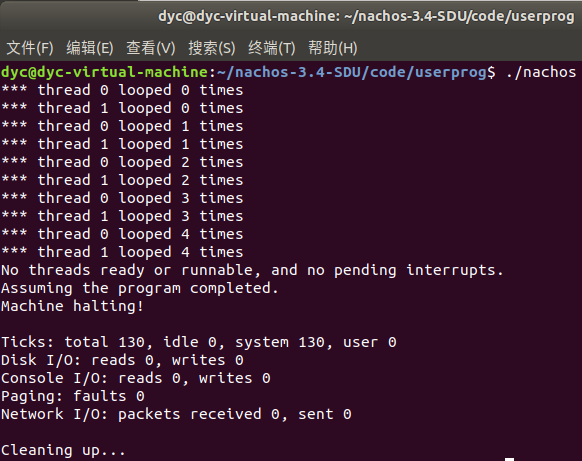


./nachos



**1.1.5 在../code/filesys，以及../code/userprog目录下采用同样的方法测试Nachos的相关代码是否正常运行。**



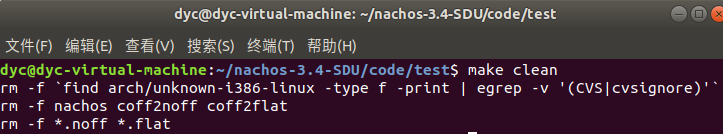


**1.1.6 测试../code/test**

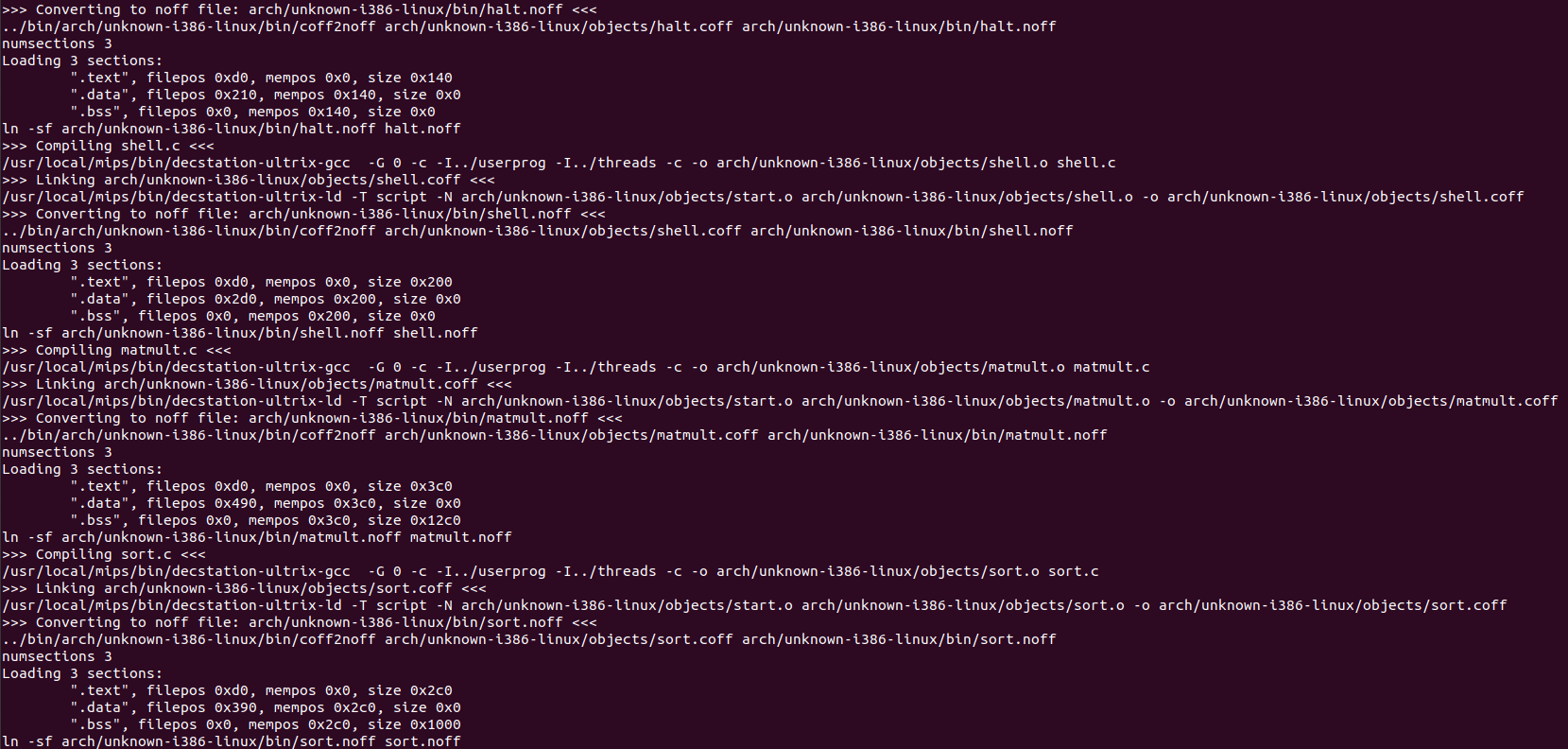
将交叉编译器gcc-2.8.1-mips.tar.gz复制到/usr/local目录中并解压；

进入../code/test，在命令终端中依次运行下述命令

make clean

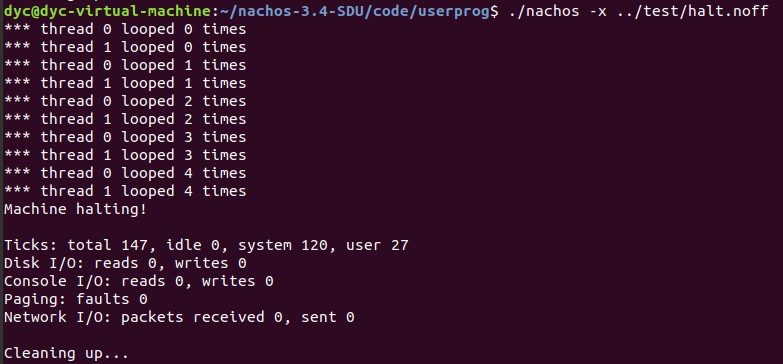


Make



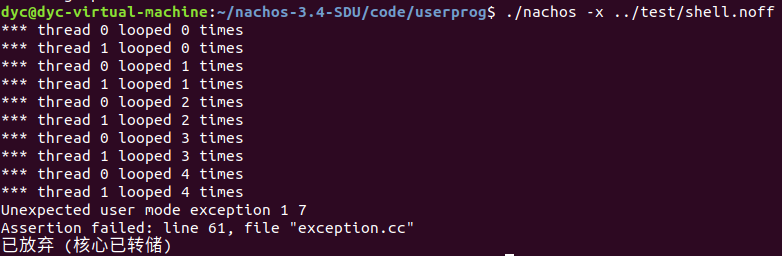
**1.1.7 测试生成的Nachos可执行程序noff**

./nachos -x ../test/halt.noff



./nachos -x ../test/shell.noff

对于shell.c、sort.c、matmult.c等几个应用程序，由于它们调用了Nachos的几个系统调用（如Exec、Join、Exit）；这些系统调用目前尚未实现，需要你们实现；因此无法正常运行。如shell.noff输出信息如下：



**1.2 Nachos系统的安装与调试过程中的思考**

**1.2.1 为什么gcc-2.8.1-mips.tar.gz一定要安装在/usr/local目录中**

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

**1.2.2 交叉编译器用于将../test目录下的Nachos应用程序（如halt.c）编译成coff格式文件，经转换后会生成Nachos可执行的文件halt.noff**

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

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

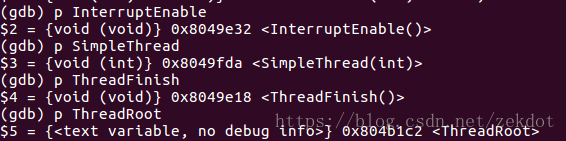
**i. InterruptEnable()**

**ii. SimpleThread()**

**iii. ThreadFinish()**

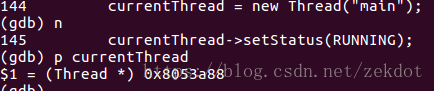
**iv. ThreadRoot()**

gdb自带命令可以查看某个函数的地址空间，指令为p 函数名借此可以查看题目给出的四个函数在内存中的地址空间

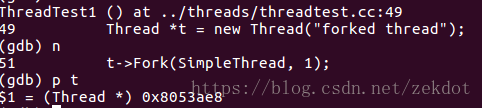


**1.2.4 下述线程对象的地址是多少？并说明找到这些对象地址的过程及方法。**

要想找到两个线程的地址空间，我们首先应该对源码进行分析，然后确定好断点要打的位置。i.通过分析，我们发现在位于system.cc中的Initialize函数有这样的代码：而根据Initialize函数在main函数中十分靠前的位置，我们可以确定，这里就是主线程的初始处。因此，我们只需要给Initialize函数打断点，进入后执行到currentThread设置状态的部分，然后查看其内存地址即可，效果如下：如图所示，其内存地址为0x8053a88。



通过分析源码，我们知道主线程Fork子线程的地方在threadtest.cc的ThreadTest1函数中：因此我们只需要在ThreadTest1函数打上断点，并执行到给t指针申请空间的部分，再查看t的地址即可，如图：通过调试，我们可以知道子线程的内存地址为0x8053ae8。

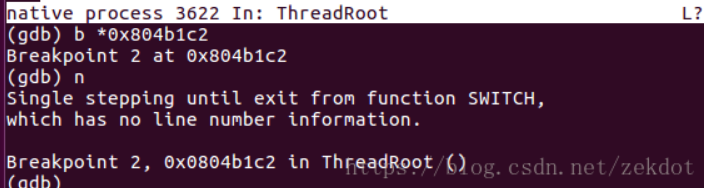


**1.2.5 当主线程第一次运行SWITCH()函数，执行到函数SWITCH()的最后一条指令ret返回，对应程序的什么位置？**

1、在ret指令的倒数第三行提到了返回地址被存入了寄存器eax中 ，接下来我们调试分析汇编部分的代码。而通过对汇编部分的查看，可以看到其对应的是SWITCH指令偏移77的位置，所以应该在SWITCH+80处打断点然后分析eax寄存器的数值。

2、这里也是通过查阅资料知道原来gdb可以将断点打在汇编语句上面，并且还能查看寄存器的值，真的涨姿势。。

3、内部打断点和查看寄存器的值的指令如下：如图可见，此时eax寄存器内部的值为0x804b1c2，我们可以进一步为这个地址打上断点然后执行。可见，此时返回值为ThreadRoot函数的地址空间。

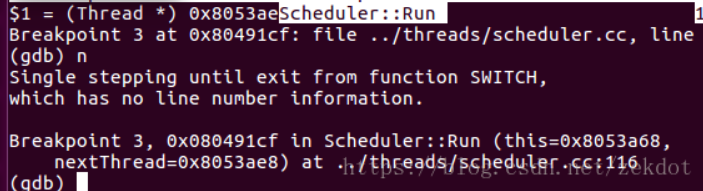


4、SWITCH和ThreadRoot是汇编代码，SWITCH负责切换当前两个进程的上下文，ThreadRoot是整个进程的生命周期。单步ni执行至SWITCH切换到ThreadRoot时，是主进程切换至了子进程，子进程刚刚进入生命周期。此时ThreadRoot第一句的地址是SWITCH的ret，代表跳转到以下地址执行。

1. \_ThreadRoot:   ;ret返回位置
2. pushl   %ebp
3. movl    %esp,%ebp
4. pushl   InitialArg
5. call    StartupPC
6. call    InitialPC
7. call    WhenDonePC
9. # NOT REACHED
10. movl    %ebp,%esp
11. popl    %ebp
12. ret

**1.2.6 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返回，对应程序的什么位置？**

通过分析源代码我们知道每一次SWITCH都会完成一次进程切换。该值为子线程的地址空间，可见，此时已经进入了子进程的线程空间，我们按下n直至再次执行到SWITCH函数。我们按上一问的方式继续查看eax寄存器的值，其值变成了0x80491cf。我们再次为该值打上断点然后执行到那里。



1. SWITCH(oldThread, nextThread);
2. DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());  //ret返回位置

**3 实验三 利用信号量实现线程同步**

**3.1 基础实验代码分析**

**3.1.1 main.cc main()**

1、Nachos本质是个进程。既然是进程，那么必然要一切从main说起。

2、可以看到在首先调用的Initialize()，在system.cc中实现。

3、调用DEBUG()输出调试信息。

4、根据是否定义THREADS宏执行ThreadTest()。

5、根据宏定义对命令行参数进行处理。

6、当前线程执行finish()。实为销毁主线程。

**3.1.2 system.cc Initialize()**

1. 可以看到在Initialize中首先是一些#ifdef #endif之类的宏，这部分是通过Makefile中设置的，主要是做为gcc的编译参数。这部分主要是用来编译不同功能的Nachos。
2. 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。
3. Statistics() Interrupt() Scheduler() Timer(),调用上述构造函数初始化系统，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。
4. 调用Thread()函数初始化当前线程，作为"main"线程。随后打开中断。
5. CallOnUserAbort(Cleanup)设置ctrl c中断nachos。

**3.1.3 utility.cc Debug()**

1. void DebugInit(char \*flagList)初始化静态常量enableFlags。
2. bool DebugIsEnabled(char flag)判断enableFlags中是否有flag或者+。
3. void DEBUG()根据DebugIsEnabled返回布尔值决定是否输出。

**3.1.4 interrupt.h PendingInterrupt类**

class PendingInterrupt就是我们要加入到中断队列的元素。中断队列里的元素类型都是这个类。有成员变量VoidFunctionPtr handler中断处理函数指针、int arg函数参数、int when中断发送时间、IntType type引发中断设备号。

**3.1.5 interrupt.h Interrupt类**

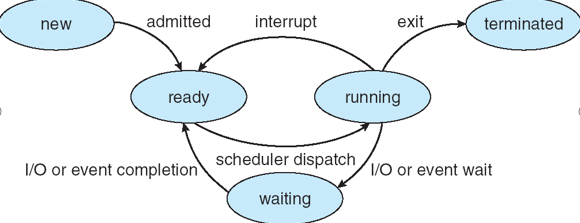
1. 成员变量IntStatus level表示中断开闭，List \*pending为中断队列，bool inHandler表示是否在处设立中断，bool yieldOnReturn表示是否需要上下文切换，status表示IdleMode SystemMode UserMode三种状态。
2. IntStatus SetLevel(IntStatus level); void Enable();void ChangeLevel(IntStatus old,IntStatus now);三个函数进行开中断或关中断。
3. bool CheckIfDue(bool advanceClock)如果是由Onetick()调用的，advanceClock是false，先检查中断执行的时间when和当前的系统时间，如果还没到，那么就将其再次加入到中断队列，如果到了，就对中断进行处理。如果是idle()调用，advanceClock是true,则前进到下一个中断的时间，对中断进行处理。
4. Onetick()只有在开中断或Nachos的CPU执行一条指令后才调用，前进系统时钟，调用CheckIfDue(false)判断是否有中断，如有中断进行处理。
5. void Idle()在线程主动放弃cpu进入阻塞状态时被调用，调用CheckIfDue(true)方法检查中断队列并处理中断，如中断时间没到则前进系统时钟,如果中断处理后就绪队列仍无线程，则Nachos终止。如果中断处理后就绪队列有线程，则切换到该线程继续执行。
6. void Schedule(VoidFunctionPtr handler, \_int arg, int fromNow, IntType type)向中断队列中插入一个中断，handler为中断处理函数指针，arg为函数参数，fromNow为中断发生时间，type为引发中断硬件号。

**3.1.6 timer.h Timer类**

1. bool randomize表示是否随机生成时间，VoidFunctionPtr handler表示中断处理函数，\_int arg函数参数。
2. Initialize()中调用构造函数，函数指针指向systme.cc中的static void TimerInterruptHandler(\_int dummy),该函数调用YieldOnReturn()引发上下文切换。
3. Timer()构造函数中调用interrupt->Schedule(TimerHandler, (\_int)this, TimeOfNextInterrupt(),TimerInt)，从而发出中断请求，加入中断队列。
4. static void TimerHandler(\_int arg)调用TimerExpired()，此函数重复发出一个中断请求interrupt->Schedule(TimerHandler, (\_int)this, TimeOfNextInterrupt(),TimerInt)，并调用Timer类成员函数指针执行static void TimerInterruptHandler(\_int dummy)，从而调用YieldOnReturn()引发上下文切换。
5. 定时器引发中断函数嵌套依次为TimerHandler() TimerExpired() TimerInterruptHandler() YieldOnReturn(),从而调用引发上下文切换。
6. 使用多个静态函数指针的原因：类成员函数指针无法传递。通过静态函数指针和类对象指针实现替代。
7. int TimeOfNextInterrupt()生成随机数实现RR调度。

**3.1.7 threads.h Threads类**

1. int \*stackTop为线程内核栈栈顶指针。
2. \_int machineState[MachineStateSize]为线程上下文存储;
3. int \*stack线程内核栈栈底，如没有分配，则为null。
4. ThreadStatus status表示线程状态，有JUST\_CREATED RUNNING READY BLOCKED四种状态。
5. char \*name为线程名。
6. void StackAllocate(VoidFunctionPtr func, \_int arg)在堆上初始化内核栈，为线程上下文存储初始化。



7、new Thread::Thread()

new->ready Thread::Fork()

ready->running Scheduler::Run()

running->ready Thread::Yield()

running->waiting Thread::Sleep()

running->terminated Thread::Finish()

new/waiting->ready Scheduler::ReadyToRun()

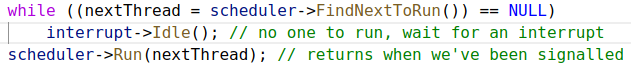
Terminated Thread::~Thread()

8、void Fork(VoidFunctionPtr func, \_int arg)调用StackAllocate(func, arg)申请内核栈栈空间并初始化线程上下文，调用scheduler->ReadyToRun(this)把线程放入调度器的就绪队列。

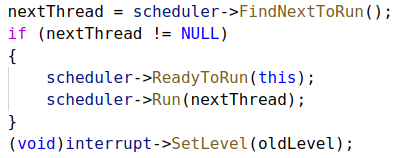
9、void CheckOverflow()判断线内核栈程是否溢出。

10、void Finish()线程进入终止态，threadToBeDestroyed = currentThread表示当前线程将被销毁，然后调用sleep()使当前线程BLOCKED，进而切换上下文。

11、void Sleep()中status = BLOCKED表示当前线程进入阻塞态。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。



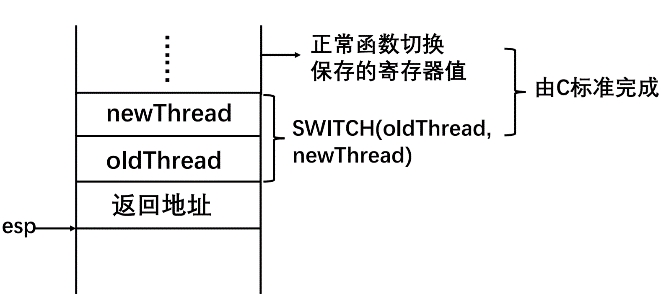
1. void Yield()首先判断就绪队列中是否有线程，如果有则切换上下文执行该线程，如果没有则继续执行当前线程。此函数为线程主动放弃cpu时调用。



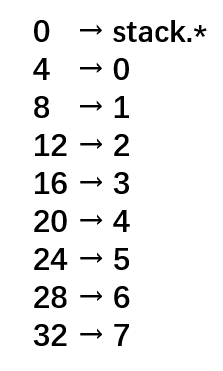
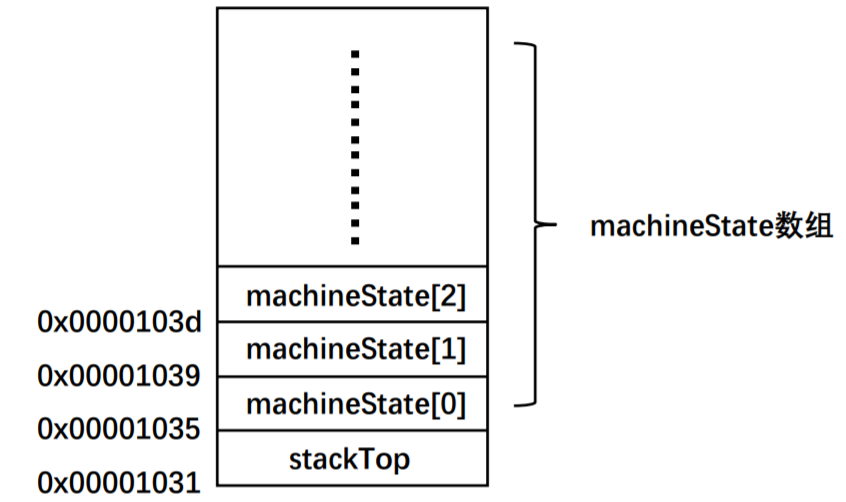
1. static void ThreadFinish()调用currentThread->Finish()，static void InterruptEnable()调用interrupt->Enable()，都作为函数指针保存在线程上下文中。
2. void ThreadRoot() void SWITCH(Thread \*oldThread, Thread \*newThread)都是汇编语言实现的函数。

**3.1.8 switch.s**

1. void SWITCH(Thread \*oldThread, Thread \*newThread)使用汇编语言实现，首先将cpu寄存器内容保存到oldThread线程控制块的上下文数组machineState中。然后newThread线程控制块的上下文数组中取出上下文存放到cpu寄存器，包括esp指向newThread内核栈栈顶。



1. 线程控制块在内存中的示意图如下



3、esi = eax+\_ESI = eax +24 = machine[5]

edx = eax+\_EDX = eax + 16 = machine[3]

edi = eax + \_EDI = eax + 28 = machine[6]

ecx = eax+\_ECX = eax + 12 = machine[2]

StartupPcState = (\_ECX / 4 – 1)=(12/4-1) = 2

InitialPCState = ( \_ESI / 4 – 1)=(24/4-1) = 5

InitialArgState = ( \_EDX / 4 – 1)=(16/4-1) = 3

WhenDonePCState = (\_EDI / 4 - 1) = (28/4-1) = 6

esi = machine[5] = func

edx = machine[3] = arg

ecx = machine[2] = InterruptEnable

edi = machine[6] = ThreadFinish

返回地址变为machineState[7]，ret指令后进入machineState[7]继续执行。

1. 如果为新线程，则machineState[7] = ThreadRoot，ThreadRoot()执行如下

pushl InitiaArg ==pushl edx------将函数所需要的参数压入栈中。

call StartupPC ==call ecx ------执行函数InterruptEnable，开中断。

call InitialPC == call esi ---------执行函数func。

call WhenDonePC == call edi ---------线程执行完成。

如为已经在cpu上执行过的线程，此时machineState[7]为中断前pc的值(返回地址)，ret指令后继续执行中断前的指令。

1. 假设主线程创建一个线程，进行线程切换，过程如下

（1）main线程执行了yield，让出了CPU

（2）通过run调用SWITCH，在SWITCH中，main保存了自己的上下文。

（3）现在nachos里只有一个从来没上过CPU的t线程。所以现在需要让t线程上去，切换上下文，包括esp和返回地址。

（4）在SIWTCH中通过ret，我们进入了ThreadRoot，这时，我们已经完成了线程的切换，也就是说现在是在t线程上了。

（5）Thread三部曲，第一步通过call来开中断，开完后返回。

（6）开始执行我们的线程函数func。执行完后，返回.

（7）通过上学期的学习我们知道，线程执行其实就是执行某一个函数，函数执行完成后，线程就可以销毁了

（8）call ThreadFinis。进行线程销毁

（9）虽然X要毁灭了，但是CPU的使用权还得传下去，所以需要X在最后时刻，将一个处于就绪态的线程切换上去，在这里，就是main。

（10）为此，它通过run调用了SWITCH，此时，在SIWTCH中，先保存了线程X的上下文，然后恢复了main的上下文（主要是栈顶，且注意！！！这里的main不是第一次上CPU了，所以它的ret是不会去执行ThreadRoot，而是直接执行它上次被打断的地方）。

（11）我们仍记得，main线程最后的操作是进行了SWITCH，所以main 的栈顶是这样。

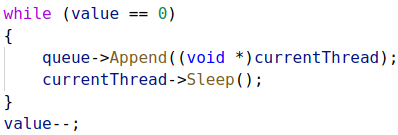
（12）然后ret，自然的回去了main的run。（因为是run调用了SWITCH，所以ret自然是回到了run）

**3.1.9 scheduler.h Scheduler类**

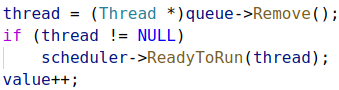
1. List \*readyList表示线程就绪队列。
2. Scheduler()初始化就绪队列。
3. void ReadyToRun(Thread \*thread)将当前线程从RUN转变为READY，加入就绪队列中。
4. Thread \*FindNextToRun()在就绪队列中找到下一个可以运行的进程。
5. void Run(Thread \*nextThread)主要调用SWITCH(oldThread, nextThread)完成上下文切换，如果切换回threadToBeDestroyed != NULL则销毁threadToBeDestroyed所指向的线程。

**3.1.10 synch.h Semaphore类**

1. char\* name表示信号量名称，int value表示信号量值，List \*queue表示该信号量阻塞线程队列。
2. void P()申请资源，如value为0则阻塞当前线程，加入信号量阻塞线程队列。value大于0，则成功获得资源，value--。



1. void V()放弃资源，检察阻塞队列，如有线程阻塞则将其放入调度器就绪队列。成功放弃资源，value++。



**3.1.11 synch.h Lock类**

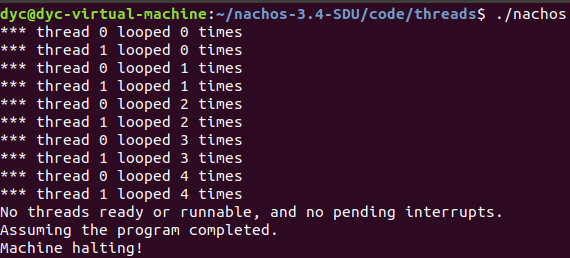
1. char\* name表示锁名称，Thread \*owner表示锁的拥有线程，Semaphore \*lock表示实现锁的信号量。
2. Lock(char \*debugName)调用lock = new Semaphore(name, 1)，锁通过值为1的信号量实现。
3. void Acquire()通过信号量P()实现申请锁，并记录拥有线程。void Release()必须拥有线程才能释放锁，通过信号量V()实现释放锁。
4. bool isHeldByCurrentThread()判断当前线程是否拥有该锁。

**3.1.12 ring.h slot类 Ring类**

1. slot成员变量int thread\_id表示发送该slot的线程id，int value表示slot的值。
2. Ring类成员变量int size表示环容量，int in, out表示输入输出位置指针，slot \*buffer为环形缓冲区。
3. Ring(int sz)初始化in out输入输出位置指针，初始化缓冲区。
4. void Put(slot \*message)将slot存入缓冲区中，void Get(slot \*message)将slot从缓冲区中取出。缓冲区指针进行取模运算。

**3.1.13 Threadtest.cc**

1. ThreadTest()创建一个"forked thread"线程，该线程执行SimpleThread(1)函数，随后主线程执行SimpleThread(0)。
2. void SimpleThread(\_int which)函数有5次循环，每次循环which号线程号和当前循环次数，随后调用currentThread->Yield()主动放弃cpu。
3. 0号线程首先执行SimpleThread(0)，输出”thread 0 looped 0 times”,随后主动放弃cpu。1号线程首先执行SimpleThread(1)，输出”thread 1 looped 0 times”,随后主动放弃cpu。这样循环5次，0号线程首先执行Finish()，然后1号线程执行Finish()。此事就绪队列中没有线程，Idle()检察中断队列中只有定时器中断或没有中断，则Nachos终止。运行结果如下图所示。



**3.2 基础实验设计任务**

**3.2.1 prodcons.cc ProdCons()**

1. Semaphore \*nempty, \*nfull表示生产者和消费者的两个同步信号量。Semaphore \*mutex;表示互斥锁。
2. Ring \*ring;表示生产者消费者所共用的环形缓冲区。
3. nempty = new Semaphore("nempty", BUFF\_SIZE)表示该同步信号量初值为BUFF\_SIZE，nfull = new Semaphore("nfull", 0)表示该同步信号量初值为0，

mutex = new Semaphore("mutex", 1)初始化互斥锁。

1. ring = new Ring(BUFF\_SIZE)初始化容量为BUFF\_SIZE的环形缓冲区。

5、随后产生N\_PROD个生产者线程，每个线程执行Producer()函数。

6、随后产生N\_CONS个消费者线程，每个线程执行Consumer()函数。

1. **void** ProdCons()
2. {
3. **int** i;
4. DEBUG('t', "Entering ProdCons");
6. nempty = **new** Semaphore("nempty", BUFF\_SIZE);
7. nfull = **new** Semaphore("nfull", 0);
8. mutex = **new** Semaphore("mutex", 1);
10. ring = **new** Ring(BUFF\_SIZE);
12. **for** (i = 0; i < N\_PROD; i++)
13. {
14. sprintf(prod\_names[i], "producer\_%d", i);
16. producers[i] = **new** Thread(prod\_names[i]);
17. producers[i]->Fork(Producer, i);
18. };
20. **for** (i = 0; i < N\_CONS; i++)
21. {
22. consumers[i] = **new** Thread(cons\_names[i]);
23. consumers[i]->Fork(Consumer, i);
24. };
25. }

**3.2.2 prodcons.cc Producer()**

进行N\_MESSG次生产，首先申请空资源，同步信号量nempty执行P()操作。

再申请环形缓冲区的互斥访问锁mutex。将slot放入环形缓冲区中。释放互斥访问锁。释放一个满资源，同步信号量nfull执行V()操作。

1. **void** Producer(\_int which)
2. {
3. **int** num;
4. slot \*message = **new** slot(0, 0);
6. **for** (num = 0; num < N\_MESSG; num++)
7. {
8. message->thread\_id = which;
9. message->value = num;
11. nempty->P();
12. mutex->P();
14. ring->Put(message);
16. mutex->V();
17. nfull->V();
19. }
20. }

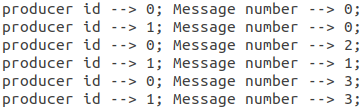
**3.2.3 prodcons.cc Consumer()**

循环从环形缓冲区中获得slot，首先申请满资源，同步信号量nfull执行P()操作。再申请环形缓冲区的互斥访问锁mutex。从环形缓冲区中取出slot。释放互斥访问锁。释放一个空资源，同步信号量nempty执行V()操作。

1. **void** Consumer(\_int which)
2. {
3. **char** str[MAXLEN];
4. **char** fname[LINELEN];
5. **int** fd;
7. slot \*message = **new** slot(0, 0);
9. sprintf(fname, "tmp\_%d", which);
10. **if** ((fd = creat(fname, 0600)) == -1)
11. {
12. perror("creat: file create failed");
13. exit(1);
14. }
16. **for** (;;)
17. {
19. nfull->P();
20. mutex->P();
22. ring->Get(message);
24. mutex->V();
25. nempty->V();
27. sprintf(str, "producer id --> %d; Message number --> %d;\n",
28. message->thread\_id,
29. message->value);
30. **if** (write(fd, str, strlen(str)) == -1)
31. {
32. perror("write: write failed");
33. exit(1);
34. }
35. }
36. }

**3.3 基础实验调试结果**

在lab3目录下执行./nachos -rs 5，上方为tmp\_0内容，下方为tmp\_1内容。





符合如下条件：每个消息只能被接收一次且在文件保存一次。来自于同一个生产者的消息，以及被同一个消费者接收到的消息，在文件保存的顺序应该按其序号升序排列。

**4 实验四 Nachos的文件系统**

**4.1 基础实验代码分析**

**4.1.1 disk.h Disk类**

1. int fileno表示模拟磁盘的unix文件号，VoidFunctionPtr handler表示中断处理程序，在任何磁盘请求完成时被调用，\_int handlerArg为函数参数，bool active表示是否正在进行磁盘操作，int lastSector表示上一个请求的磁盘扇区，int bufferInit表示当开始加载磁盘缓冲区时的开始扇区号。
2. 模拟磁盘的unix文件，开头为一个MagicNumber 0x456789ab。磁盘扇区为SectorSize 128个字节，每个磁道有32 SectorsPerTrack个扇区，每个磁盘有32 NumTracks个磁道，NumSectors为32 \* 32，DiskSize (MagicSize + (NumSectors \* SectorSize))为4 + 32 \* 32 \* 128。
3. Disk(char \*name, VoidFunctionPtr callWhenDone, \_int callArg)初始化成员变量。通过文件名字符串name打开文件。如文件存在，打开成功，则读入4字节数据，如果等于0x456789ab则检查成功。如文件不存在，打开失败，则创建新文件，并在文件开头写入4字节的0x456789ab，并调用Lseek(fileno, DiskSize - sizeof(int), 0)把文件指针定位到磁盘最后一个双字，写入EOF。
4. PrintSector(bool writing, int sector, char \*data)打印出data指向的扇区32个int。
5. int TimeToSeek(int newSector, int \*rotation)先计算新旧磁盘请求的磁道号，返回值int seek为寻道时间。当寻道完成到达目标磁道时，可能处于两个扇区中间，需要旋转等待，磁盘一直在旋转，RotationTime转过一个扇区，总时间求模后被RotationTime减去，即可得到旋转等待时间，通过指针int \*rotation返回。
6. int ModuloDiff(int to, int from)计算从当前扇区到目标扇区需要旋转的扇区数。
7. int ComputeLatency(int newSector, bool writing)首先调用TimeToSeek(newSector, &rotation)计算寻道和旋转到扇区头的时间。

int timeAfter = stats->totalTicks + seek + rotation为此时的系统时间。

如果没有磁盘缓冲区或为写操作或新旧磁道不同，

rotation += ModuloDiff(newSector, timeAfter / RotationTime) \* RotationTime计算出在目标磁道内剩余的旋转等待时间。再加上读取一个扇区的时间RotationTime，返回(seek + rotation + RotationTime)为总耗费时间。

如果有磁盘缓冲区并且为读操作并且新旧磁道为同一磁道，bufferInit为目标到达目标磁道的某一扇区边界时的系统时间，此时开始缓存。

((timeAfter - bufferInit) / RotationTime) > ModuloDiff(newSector, bufferInit / RotationTime)为判断目标扇区是否已经被缓存，即是否已经转到过的扇区数大于到达某一扇区边界后需要再转的扇区数。这种清空下，只需从缓冲区中读取数据，返回总耗费时间RotationTime。

1. void UpdateLast(int newSector)更新bufferInit为目标到达目标磁道的某一扇区边界时的系统时间，此时开始缓存。
2. void ReadRequest(int sectorNumber, char \*data)

int ticks = ComputeLatency(sectorNumber, FALSE)计算出完成磁盘操作所需要的时间。随后判断扇区号是否合法，磁盘是否处于空闲状态。调用unix文件操作系统调用移动指针读取模拟磁盘上的扇区。



active = TRUE磁盘进入操作状态，UpdateLast(sectorNumber)更新磁盘缓冲区起始始终，interrupt->Schedule(DiskDone, (\_int)this, ticks, DiskInt)在中断队列中加入磁盘操作完成后所产生的中断，时间为ticks后。

1. void Disk::WriteRequest(int sectorNumber, char \*data)

int ticks = ComputeLatency(sectorNumber, FALSE)计算出完成磁盘操作所需要的时间。随后判断扇区号是否合法，磁盘是否处于空闲状态。调用unix文件操作系统调用移动指针读取模拟磁盘上的扇区。



active = TRUE磁盘进入操作状态，UpdateLast(sectorNumber)更新磁盘缓冲区起始始终，interrupt->Schedule(DiskDone, (\_int)this, ticks, DiskInt)在中断队列中加入磁盘操作完成后所产生的中断，时间为ticks后。

1. DiskDone()中断处理函数通过传递过来的Disk\*指针调用HandleInterrupt()

该函数调用该磁盘的成员函数指针所指向的函数(\*handler)(handlerArg)。成员函数指针和参数在构造函数disk()时被初始化。



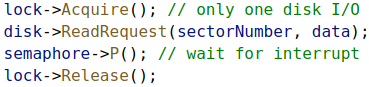
**4.1.2 synchDisk.h SynchDisk类**

1、定义了一个“同步”磁盘抽象。与其他I/O设备一样，原始物理磁盘是一个异步设备——请求读取或写入磁盘的一部分会立即返回，并且稍后会发生中断，以表示操作已完成。（同样，磁盘设备的物理特性假设只有一个可以一次请求操作）。这个类提供了一个抽象，对于发出请求的任何单个线程，它都会一直等到操作完成后再返回。

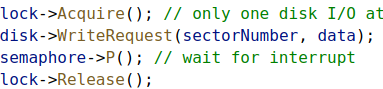
2、成员变量Disk \*disk表示模拟磁盘，Semaphore \*semaphore为进行磁盘操作时的同步信号量，Lock \*lock保证一次只能有一个磁盘请求。

3、SynchDisk(char \*name)初始化成员变量，semaphore同步信号量初值设置为0。disk = new Disk(name, DiskRequestDone, (\_int)this)初始化模拟磁盘，模拟磁盘操作完成时中断处理函数为DiskRequestDone()，此函数再调用dsk->RequestDone()，执行semaphore->V()，阻塞线程解除阻塞，加入到就绪队列。

4、void ReadSector(int sectorNumber, char \*data)从磁盘中同步读取指定扇区，首先申请互斥锁，磁盘进行异步操作，之后同步信号量P()阻塞线程，直到中断处理后同步信号量V()解除阻塞，线程放弃互斥锁。



5、void WriteSector(int sectorNumber, char \*data)从磁盘中同步读取指定扇区，首先申请互斥锁，磁盘进行异步操作，之后同步信号量P()阻塞线程，直到中断处理后同步信号量V()解除阻塞，线程放弃互斥锁。



**4.1.3 bitmap.h BitMap类**

1. 下面的类定义了一个“位图”，一个位数组，每个位都可以独立设置、清除和测试。对于管理数组元素（例如磁盘扇区或主内存页）的分配最有用。每一位表示对应的扇区或页是否正在使用或空闲。
2. 成员变量int numBits表示位图中的位数，numWords表示位图存储字数（如果numBits不是单词位数的倍数，则向上舍入），unsigned int \*map为位图存储数组。
3. BitMap(int nitems)初始化成员变量，调用Clear(i)初始化位图。
4. void Mark(int which)在位图中设置“which”位,“which”是要设置的位号。首先判断位号是否合法再把该位置1。
5. void Clear(int which)判断位号合法性后，把某一位清0。
6. bool Test(int which)判断位号合法性后，返回指定位值。
7. int Find()返回清晰的第一个位的编号。作为副作用，设置该位（将其标记为正在使用中）（换句话说，查找并分配一个位），如果没有找到，返回-1。
8. int NumClear()调用Test()返回空闲位数。
9. void Print()打印位所有置1位的编号。
10. void FetchFrom(OpenFile \*file)调用OpenFile::ReadAt()从Nachos文件初始化位图的内容，file是文件指针。
11. void WriteBack(OpenFile \*file)调用OpenFile::WriteAt()把位图保存到Nachos文件中。

**4.1.4 filehdr.h FileHeader类**

1. 下面的类定义了Nachos“文件头”（在UNIX术语中是“i-node”），描述在磁盘上哪里可以找到文件中的所有数据。文件头被组织为指向数据块的指针的简单表。
2. FileHeader数据结构可以存储在内存或磁盘上。在磁盘上时，FileHeader存储在一个扇区中——这意味着我们假设这个数据结构的大小与一个磁盘扇区相同。如果没有间接寻址，这将最大文件长度限制在4K字节以下。
3. NumDirect ((SectorSize - 2 \* sizeof(int)) / sizeof(int))表示索引项为30个。MaxFileSize (NumDirect \* SectorSize)文件大小最大为30 \* 128字节。
4. 没有构造函数；相反，可以通过为文件分配块（如果是新文件）或从磁盘读取来初始化文件头。
5. 成员变量int numBytes表示文件字节数，int numSectors表示文件扇区数，int dataSectors[NumDirect]为索引数组（文件中每个数据块的磁盘扇区号）。
6. 用于管理磁盘文件头的例程（在UNIX中，这称为i节点）。文件头用于定位文件数据在磁盘上的存储位置。我们将其实现为一个固定大小的指针表——表中的每个条目都指向包含该部分文件数据的磁盘扇区（换句话说，不存在间接块或双间接块）。选择表大小是为了使文件头大小刚好能够容纳一个磁盘扇区。与实际系统不同，我们不会在文件头中跟踪文件权限、所有权、上次修改日期等。文件头可以通过两种方式初始化：对于新文件，通过修改内存中的数据结构来指向新分配的数据块。对于已经在磁盘上的文件，通过从磁盘读取文件头。
7. bool Allocate(BitMap \*freeMap, int fileSize)，BitMap \*freeMap表示管理当前磁盘空闲块的位图，int fileSize为文件字节大小，计算所需扇区数，并判断是否有足够的空闲块，如果没有则返回false，如果有则freeMap->Find()进行分配并返回true。
8. void Deallocate(BitMap \*freeMap)取消分配为此文件的数据块分配的所有空间，BitMap \*freeMap是磁盘扇区空闲块的位图。依次检察索引表，首先检察某一项所对应扇区是否在位图中置1，freeMap->Clear((int)dataSectors[i])清楚分配给该文件的扇区（位图中置0）。
9. void FetchFrom(int sector)，因为一个FileHeader对象即为一个扇区，在同步磁盘中读取扇区存放入本对象指针所指向的地址，完成从磁盘中读取FileHeader。



1. void WriteBack(int sector)因为一个FileHeader对象即为一个扇区，把本FileHeader指针所指向的地址存储到在同步磁盘的一个扇区中，完成FileHeader写入磁盘。



1. int ByteToSector(int offset)返回哪个磁盘扇区正在文件中存储特定字节。这实际上是从虚拟地址（文件中的偏移量）到物理地址（偏移量数据存储的扇区）的转换。offset是该字节文件中的位置。
2. void Print()打印fileheader的内容，以及fileheader指向的所有数据块（扇区）的内容。

**4.1.5 openfile.h OpenFile类**

1. 用于打开、关闭、读取和写入单个文件的数据结构。有两种实现方式。一个是直接将文件操作转换为底层UNIX操作 "STUB"（参见filesys.h中的注释）。另一个是“真正的”实现，它将这些操作转换为读写磁盘扇区请求。在这个文件系统的基线实现中，我们不担心不同线程对文件系统的并发访问——这是分配的一部分。
2. 管理打开的Nachos文件的例程。和UNIX一样，文件在读写之前必须是打开的，完成后，我们可以关闭它（在Nachos中，通过删除OpenFile数据结构）。同样在UNIX中，为了方便起见，我们在文件打开时将file header保存在内存中。
3. 成员变量FileHeader \*hdr为文件头，int seekPosition表示文件读写指针，int hdrSector表示初始化FileHeader \*hdr的扇区号。
4. OpenFile(int sector)打开一个Nachos文件进行读写。hdr->FetchFrom(sector)从磁盘扇区中初始化文件头。打开文件时将文件头放入内存。
5. void Seek(int position)更改文件读写指针的位置。
6. ReadAt() WriteAt()读/写文件的一部分，从“位置”开始。返回实际写入或读取的字节数，但没有副作用（当然，write修改文件除外）。无法保证请求在磁盘扇区边界上开始或结束；但是磁盘一次只知道如何读/写整个磁盘扇区。对于读操作，我们阅读了请求中的全部或部分扇区，但只复制了感兴趣的部分。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。
7. int ReadAt(char \*into, int numBytes, int position)先读取全部扇区到buf，再从buf复制部分内容到into。
8. int WriteAt(char \*from, int numBytes, int position)先读取头尾扇区到buf，再将from内容复制到buf，再从buf复制部分内容到磁盘。
9. Read() Write()读/写文件的一部分，从seekPosition开始。返回实际写入或读取的字节数，作为副作用，移动文件中的当前位置指针。使用更原始的ReadAt() WriteAt()实现。
10. int Read(char \*into, int numBytes)调用ReadAt(into, numBytes, seekPosition)，并移动seekPosition。
11. int Write(char \*into, int numBytes)调用WriteAt(into, numBytes, seekPosition)，并移动seekPosition。

**4.1.6 directory.h DirectoryEntry类**

1. directory entry，表示目录中的文件。每个条目都给出了文件名，以及文件头在磁盘上的位置。内部数据结构保持公共，以便目录操作可以直接访问它们。
2. 成员变量bool inUse为有效位，int sector表示该文件FileHeader所在磁盘扇区号，char name[FileNameMaxLen + 1]表示文件名。

**4.1.7 directory.h Directory类**

1. 定义了一个类似UNIX的“目录”。目录中的每个条目描述一个文件，以及在磁盘上的位置。目录数据结构可以存储在内存中，也可以存储在磁盘上。当目录数据结构在磁盘上时，它作为一个普通的Nachos文件存储。构造函数初始化内存中的目录结构；FetchFrom/WriteBack操作将目录信息保存到磁盘。
2. 此外，此实现具有目录大小无法扩展的限制。换句话说，一旦使用了目录中的所有条目，就不能再创建文件了。解决这一问题是分配的一部分。
3. 成员变量int tableSize表示文件数。DirectoryEntry \*table为目录数组数据结构。
4. Directory(int size)初始化目录；最初，目录是完全空的。如果正在格式化磁盘，我们只需要一个空目录，否则，我们需要调用FetchFrom来从磁盘初始化它。table = new DirectoryEntry[size]初始化目录数组数据结构。table[i].inUse = FALSE并把每个目录有效位设为无效。
5. void FetchFrom(OpenFile \*file)调用OpenFile中的ReadAt()从磁盘文件中读取目录。



1. void WriteBack(OpenFile \*file)调用OpenFile中的WriteAt()将目录写入磁盘文件。



1. int FindIndex(char \*name)在目录中查找文件名，并返回其在目录项表中的位置。如果名称不在目录中，则返回-1。
2. int Find(char \*name)在目录中查找文件名，并返回存储文件头的磁盘扇区号。如果名称不在目录中，则返回-1。
3. bool Add(char \*name, int newSector)将文件添加到目录中。如果成功，则返回TRUE；如果文件名已在目录中，或者目录已满，并且没有更多的空间容纳其他文件名，则返回FALSE。
4. bool Remove(char \*name)通过改变有效位从目录中删除文件名。如果成功，则返回TRUE；如果文件不在目录中，则返回FALSE。
5. void List()列出目录中的所有文件名。
6. void Print()列出目录中的所有文件名、它们的文件头扇区号，每个文件的头文件内容，每个文件的文件内容。用于调试。

**4.1.8 filesys.h Filesys类**

1. Nachos文件系统的数据结构。文件系统是存储在磁盘上的一组文件，按目录组织。文件系统上的操作与“命名”有关，即在给定文本文件名的情况下创建、打开和删除文件。对已经打开的文件文件（读、写、关闭等方法）可以在open file类（OpenFile.h）中找到。
2. 我们定义了两个独立的文件系统实现，“STUB”版本只是将Nachos文件系统操作重新定义为运行Nachos模拟的机器上的本机UNIX文件系统上的操作。这是在多道程序设计和虚拟内存分配（使用文件系统）在文件系统分配之前完成的情况下提供的。
3. 另一个版本是一个“real”的文件系统，建立在磁盘模拟器之上。使用本机UNIX文件系统（在名为“disk”的文件中）模拟磁盘。
4. 在“real”实现中，文件系统中使用了两种关键数据结构。只有一个“root”目录，列出文件系统中的所有文件；与UNIX不同，基线系统不提供分层目录结构。此外，还有一个用于分配磁盘扇区的位图。根目录和位图本身都作为文件存储在Nachos文件系统中——这在初始化模拟磁盘时会导致一个有趣的引导问题。
5. Filesys管理文件系统整体操作的例程。实现从文本文件名映射到文件的例程。
6. 文件系统中的每个文件都有：一个文件头，存储在磁盘上的一个扇区中（文件头数据结构的大小被精确地安排为1个磁盘扇区的大小），许多数据块，文件系统目录中的条目。
7. 文件系统由几个数据结构组成：bitmap directory，位图和目录都表示为普通文件。它们的文件头位于特定的扇区（扇区0和扇区1）中，以便文件系统可以在启动时找到它们。文件系统假设位图和目录文件在Nachos运行时一直保持“打开”。对于那些修改目录和/或位图的操作（如创建、删除），如果操作成功，更改将立即写入磁盘（这两个文件始终保持打开状态）。如果操作失败，并且我们已经修改了部分目录和/或位图，我们只需放弃更改的版本，而不将其写回磁盘。
8. 存在限制如下，不存在同步。因为并发访问文件的大小是固定的，在创建文件时设置的文件大小不能大于3KB左右没有分层目录结构，只能向系统中添加有限数量的文件没有试图使系统对故障保持健壮（如果Nachos在中间退出修改文件系统的操作可能会损坏磁盘）。
9. 成员变量OpenFile \*freeMapFile表示位图的打开文件，OpenFile \*directoryFile表示根目录的打开文件。
10. FileSystem(bool format)初始化文件系统。如果format=TRUE，磁盘上没有任何内容，我们需要初始化磁盘以包含一个空目录和一个空闲扇区位图（几乎但不是所有扇区都标记为空闲）。如果format=FALSE，我们只需打开表示位图和目录的文件。首先，为目录和位图的文件头分配空间（确保没有其他人获取这些文件！）。



其次，为包含目录和位图文件内容的数据块分配空间。最好有足够的空间！



将位图和目录文件头刷新回磁盘我们需要在“打开”文件之前执行此操作，因为Open从磁盘中读取文件头（当前磁盘上有垃圾！）。



确定打开位图和目录文件现在文件系统操作假设这两个文件在Nachos运行时保持打开状态。



一旦文件“打开”，我们就可以将每个文件的初始版本写回磁盘。此时的目录是完全空的；但是位图已更改，以反映磁盘上的扇区已分配给文件头，并保存目录和位图的文件数据。



如果不格式化磁盘，只需打开表示位图和目录的文件；这些文件在Nachos运行时保持打开状态



1. bool Create(char \*name, int initialSize)在Nachos文件系统中创建一个文件（类似于UNIX Create），因为我们不能动态地增加文件的大小，所以我们必须给出Create文件的初始大小。注意，这个实现假定没有对文件系统的并发访问！

创建文件的步骤如下：确保文件不存在，为文件头分配一个扇区，为文件的数据块分配磁盘空间，将名称添加到目录，将新文件头存储在磁盘上，将对位图和目录的更改刷新回磁盘。

引发错误的原因：文件已在目录中，文件头没有可用空间，目录中没有文件的可用项，文件的数据块没有可用空间。

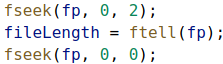
1. OpenFile \*Open(char \*name)打开文件进行读写，查找文件头的位置，使用目录将头放入内存。将文件放入系统打开文件表。
2. bool Remove(char \*name)从文件系统中删除文件。这需要：将其从目录中删除删除其标题的空间删除其数据块的空间将更改写入目录，将位图回磁盘。

void List()调用void Directory::List()列出文件系统根目录中的所有文件名。

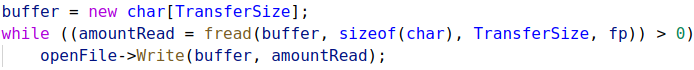
1. void Print()打印文件系统的所有内容。

**4.1.9 fstest.cc**

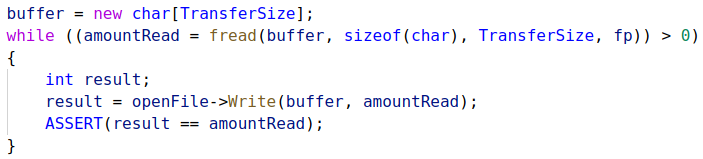
1. void Copy(char \*from, char \*to)将UNIX文件的内容“from”复制到Nachos文件“to”。UNIX系统调用fp = fopen(from, "r")打开文件，通过fseek()确定文件大小。



fileSystem->Create(to, fileLength)在Nachos创建新文件，openFile = fileSystem->Open(to)打开该文件。之后循环读取写入新文件。



1. void Append(char \*from, char \*to, int half)将UNIX文件的内容“from”附加到Nachos文件“to”。如果“half”为非零，则从文件“to”的中间开始追加“from”的内容；否则，在文件“to”的结尾后进行真正的追加。如果Nachos文件“to”不存在，请创建长度为0的Nachos文件“to”，然后将UNIX文件“from”的内容追加到其末尾。UNIX系统调用fp = fopen(from, "r")打开文件，通过fseek()确定文件大小。如果half非0，则调用openFile->Seek(start)移动指针到文件中间。之后循环读取写入新文件。



1. void NAppend(char \*from, char \*to)NAppend与Append相同，只是“from”文件是Nachos文件，而不是UNIX文件。它将Nachos文件的内容“from”附加到Nachos文件的末尾“to”。如果Nachos文件“to”不存在，请创建长度为0的Nachos文件“to”，然后将UNIX文件“from”的内容追加到其末尾。首先判断两文件名不相同。openFileFrom = fileSystem->Open(from)打开from文件并获得文件长度。打开to文件，如不存则则创建。

**4.1.10 数据单位对比**

1. Disk SynchDisk读取单位为扇区（128 byte）。
2. FileHeader从磁盘初始化单位为扇区（128 byte）。
3. OpenFile从磁盘初始化单位为扇区（128 byte），读取单位为字节（byte）。
4. BitMap从磁盘初始化单位为4 byte。
5. Directory从磁盘初始化单位为字节（byte）。

**4.1.11 输出区别**

1. FileHeader::Print()打印文件头的内容，以及文件头指向的所有数据块的内容。
2. void Directory::List()列出目录中的所有文件名。
3. void Directory::Print()列出目录中的所有文件名、它们的文件头扇区号，每个文件的头文件内容，每个文件的文件内容。用于调试。
4. Disk::PrintSector(bool writing, int sector, char \*data)打印出data指向的扇区32个int。
5. void FileSystem::List()调用void Directory::List()列出文件系统根目录中的所有文件名。
6. void FileSystem::Print()打印文件系统的所有内容。

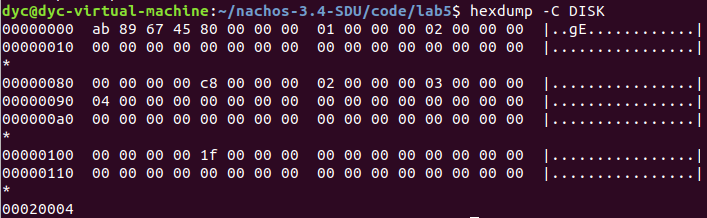
**4.2 基础实验调试结果**

**4.2.1 硬盘格式化**

1、将DISK文件删除。

2、./nachos -f格式化硬盘（在模拟硬盘DISK上创建一个文件系统）。

3、hexdump -C DISK



（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）；76 根据../filesys/filehdr.h 中对文件头的定义，文件头由三部分组成：（int numBytes, int numSectors, int dataSectors[NumDirect];），即文件头所描述的三元组<文件的大 小，占用的扇区数，各数据块所在扇区列表>。 Nachos 将头文件的大小设计为 128 个字节，恰好保存到一个扇区中。

0x4~0x7：4 个字节，位示图文件大小（注：是位示图文件，不是头文 件），值为 0x80，表示位示图文件大小为 0x80=128 字节 

0x8~0xB：4 个字节，系统为位示图文件数据所分配的扇区数，其值为 0x1，表示位示图文件数据只需要一个扇区；

0xC~0xF：位示图文件数据块所在的扇区号，其值为 0x2，说明系统 将位示图文件的数据保存在第 2 号扇区中。

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

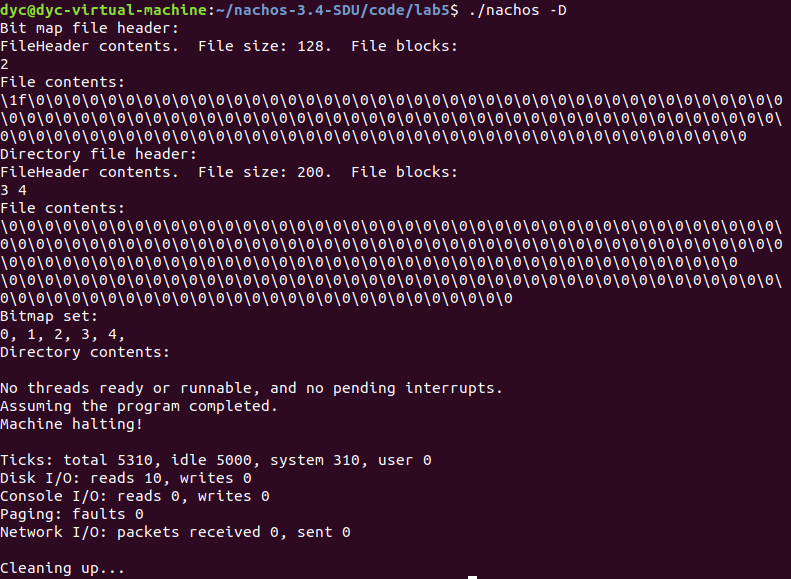
0x84~0x87：4 个字节，目录表文件大小，值为 0xC8，表示目录表文件 大小为 0xC8=200 字节； 注：关于目录文件的大小，一个目录项（DirectoryEntry）大小为 20 个 字节，Nachos 为目录文件建立了 10 个目录项（该文件系统最多可创 建 10 个文件），因此目录文件大小为 200 字节；

0x88~0x8B：4 个字节，系统为目录文件数据所分配的扇区数，其值为 0x2，表示目录文件数据需要 2 个扇区（目录文件大小为 200 字节，需 要占用两个扇区）；

0x8C~0x8F：目录表文件第 1 个数据块所在的扇区号，其值为 0x3，说 明系统将目录表文件第 1 个数据块保存在第 3 号扇区中；

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

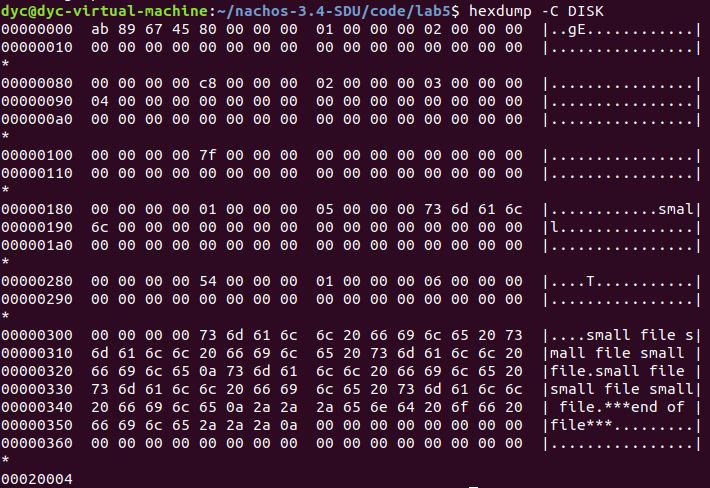
（d）0x104~0x183：扇区 2，128 字节；位示图的数据块，存储位示图文件内 容； （e）0x184~0x203，扇区 3，0x204~0x283：扇区 4：这两个扇区是目录文件 的数据块，存放目录项；目前没有任何文件，值为 0； 目前只有 5 个扇区被分配（扇区 0,1,2,3,4），其余都空闲，因此在位示图中只有这 5 个扇区对应的位被置 1，其余均为 0，所以位示图文件内容是：11111000 0000…..0000 （1024 位），第一个字节 11111000 在 Nachos 中表示成 0x1F； nachos -D 的输出也说明了上述观点。



**4.2.2 复制一个文件到硬盘**

1、./nachos –cp test/small small 复制一个文件 small 到 DISK 中；

2、hexdump –C DISK，屏幕输出如图 5-4 所示。



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

（c）3号扇区（0x184~0x203）存储的目录表也发生了改变，因为我们新建了一

个文件 small，需要在一个空闲的目录项中添加文件 small 对应的信息；目录表有 10 个目录项组成，每个目录项是一个三元组 < bool inUse ， int sector，char name[FileNameMaxLen + 1]>，由上图中可以看出：

0x184~0x187 所示的 4 个字节对应三元组中的 inUse（注：尽管 inUse 占

用一个字节，但根据编译器对各成员变量的起始地址对齐的原则，为inUse

分配与 sector 相同的字节数，4 个字节）表示该目录项是否已被使用，该

值为 1，说明该目录项正被一个文件使用。

0x188~0x18B 所示的 4 个字节对应三元组中的 sector，说明该目录项所记

录的文件（即文件 samll）的文件头所在的扇区号，这里是 5，说明该目录

项对应的文件（small）的文件头在第 5 号扇区；若要考察文件 small 的详

细信息，需要到 5 号扇区访问其文件头。

0x18C~0x195 的 10 个字节是文件名（文件名占用 9 个字节，最后一个字

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 0x54，说明 small 文件大小

是 0x54 个字节；

 0x288~0x28B：4 个字节，系统为文件数据所分配的扇区数，该值为 1，说

明系统只为文件 small 的数据分配了一个扇区（一个扇区大小为 128 字节，

文件 small 的大小为 84 字节）

 0x28C~0x28F：系统为文件数据所分配的扇区列表。这里 samll 文件数据

只需一个扇区，将其存放在第 6 号扇区；

如果想考察samll文件的内容，需要访问第 6 号扇区。 理论上讲，该扇区其余内容都应该为空（从 0x290~0x303）。由于FileHeader

类没有显示的构造函数，编译器就为其自动设定了构造函数 FileHeader::

FileHeader {}，为该类实例化对象时，所分配的内存可能含有信息，因此

可以 0x290~0x303 的内容，在此没有意义。

（f）6号扇区（0x304~0x383）是“samll”的文件的数据块，其中 0x304~0x357

（ 0x357-0x304+1=0x54 是文件的长度）是文件内容，其余空闲；可以考

察../filesys/test/small文件的内容，与这里显示的内容进行比较，查看文件在硬盘的存 储方式。其中 0x324 中的0x0a是换行符。

（g）小结

0x4~0x83： 0 号扇区，位示图文件头

0x84~0x103： 1 号扇区，目录表文件头

0x104~0x183；2 号扇区，位示图文件数据块

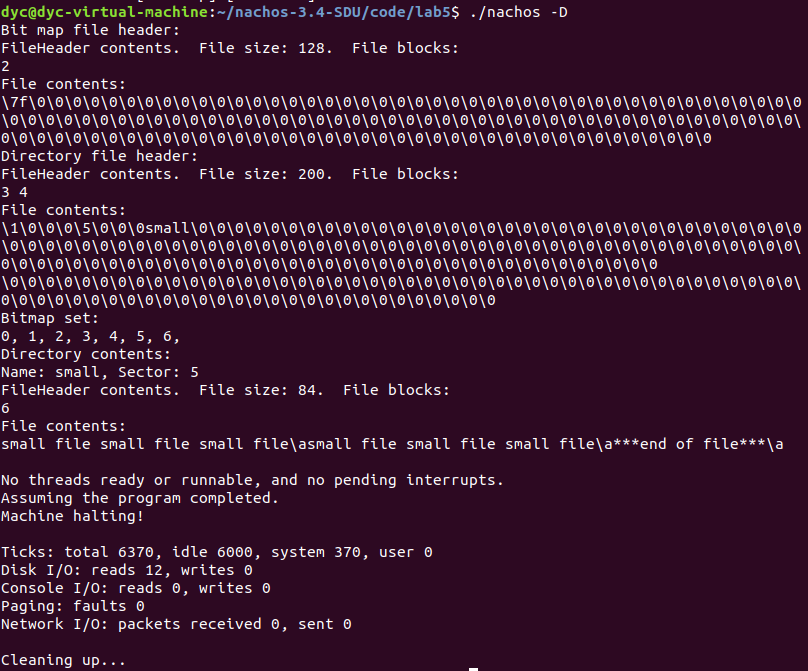
0x184~0x203：3 号扇区，目录表文件第一个数据块

0x204~0x283：4 号扇区，目录表文件第二个数据块

790x284~0x303：5 号扇区，small 文件头

0x304~0x383：6 号扇区，small 文件数据块

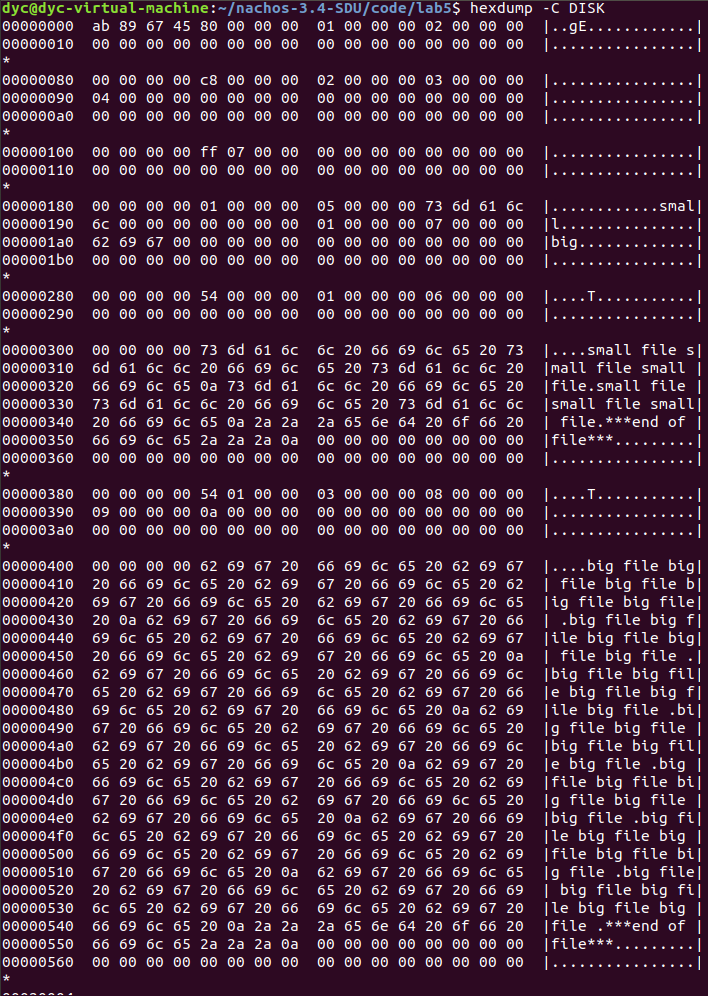
./nachos -D 的输出也说明了上述观点。



**4.2.3 复制另一个文件到硬盘**

1、./nachos –cp test/big big 复制一个文件 big 到 DISK 中；

2、hexdump –C DISK，屏幕输出如图 5-6 所示。



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

（b）2号扇区（0x104~0x183）存储的位示图文件内容发生改变，位示图文件内容由 11111000 0000…..0000（1024 位）改变为 1111 1111 1110…..0000（1024 位），前3个字节 1111 1111 1110 在 Nachos 中表示成 0xFF7（复制 big 文件之前是 0x7F，比原来多了4个被使用的扇区），说明 0~0xa 号 11 个扇区被使用；多出的4个扇区：big 文件头1个，big 数据块3个；

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

（d）7号扇区（0x384~0x403），big 文件的文件头；small 的文件的数据块存储在6号扇区，其后的第7号扇区分配给了big文件的文件头；

0x384~0x387：4 个字节，big 文件大小是 0x154 字节；

0x388~0x38B：4 个字节，big 文件的数据占用了 3 个扇区（一个扇区大小为128字节，文件 big 的大小为 340 个字节，需要分配 3 个扇区）

0x38C~0x38F：系统将 big 文件的数据分配到第 0x8、0x9、0xa 号 3 个扇区中；如果想考察 big 文件的内容，需要访问这三个扇区。

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

（f）小结

0x4~0x83： 0号扇区，位示图文件头

0x84~0x103： 1号扇区，目录表文件头

0x104~0x183；2号扇区，位示图文件数据块

0x184~0x203：3号扇区，目录表文件第一个数据块

0x204~0x283：4号扇区，目录表文件第二个数据块

0x284~0x303：5号扇区，small 文件头

0x304~0x383：6号扇区，small 文件数据块

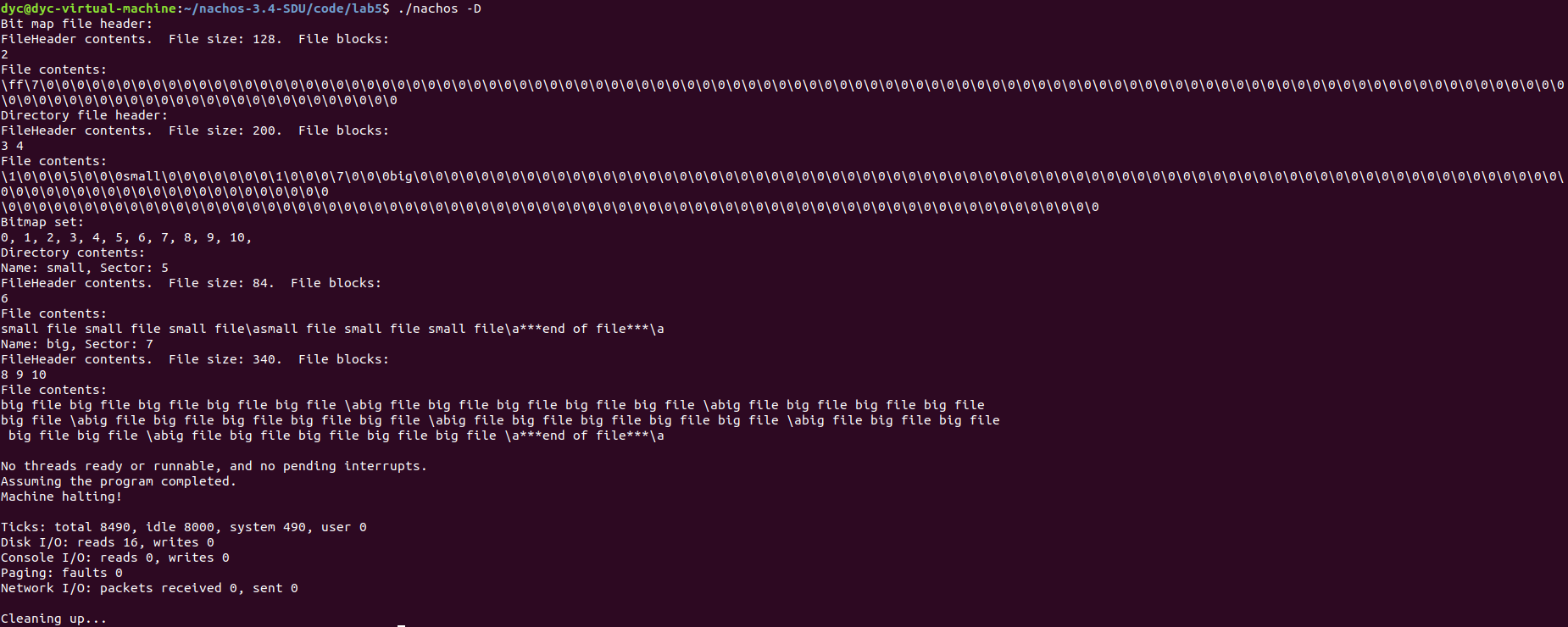
0x384~0x403：7号扇区，big 文件头

0x404~0x483：8号扇区，big 文件第 1 块数据（开始的 128 字节）

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

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

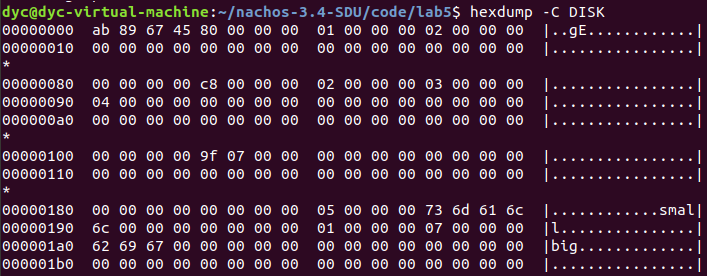
nachos -D 的输出也说明了上述观点



**4.2.4 在硬盘上删除文件**

1、./nachos –r small，删除 small 文件；

2、hexdump –C DISK，屏幕输出如图 5-8 所示



删除 small 文件的过程：

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

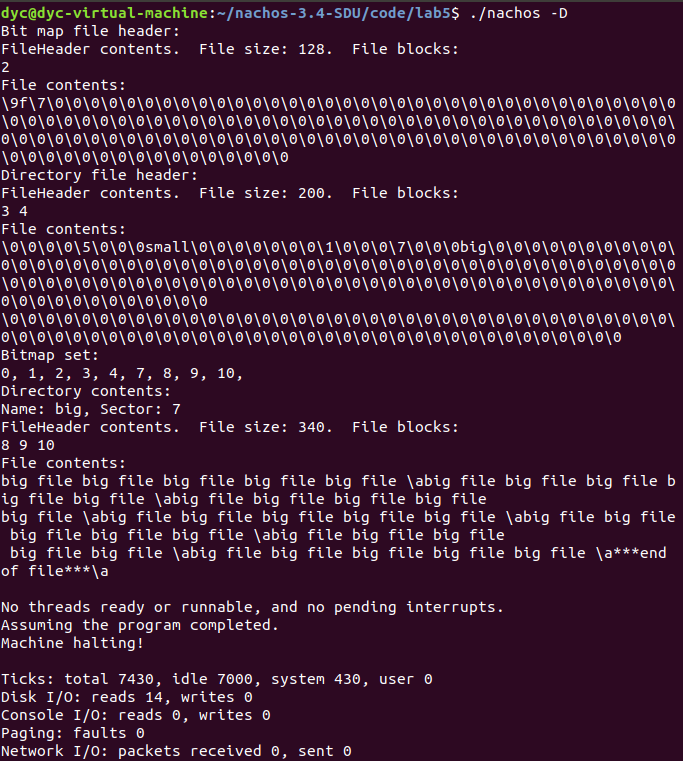
（b）清除目录表中为 samll 文件所分配的目录项中的 inUse 位（3号扇区中

0x184内容 1 变为 0，即位small文件分配的目录项变为空闲）

（c）可以看出，删除一个文件后，该文件在目录表中文件名、文件头所占的扇

区号均未清除，只是将该目录项变为空闲；文件头中的信息（文件大小、文件所占用的扇区数以及为文件数据分配的扇区列表）也未清除；文件的内容也未清除；

命令 nachos –D 的输出也说明了上述观点。



Nachos调用 FileSystem::Remove(char \*name)删除一个文件，参FileSytem::Remove()，Directory::Remove()等对删除文件的实现。因此，要恢复一个删除的文件，只要该文件的上述信息未被新建的文件覆盖，就可以根据文件名在目录项中找到该文件所对应的项，将对应的 inUser位置1，并在位示图中恢复文件头所占用的扇区号，再根据文件头的信息在位示图中恢复文件数据所占用的扇区号即可；上述删除文件的策略为恢复一个删除的文件带来了极大的便利；（其实FAT文件系统就采用该策略）。

**5 实验五 Nachos的文件系统扩展**

**5.1 文件扩展设计任务**

**5.1.1 分析文件扩展**

../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。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，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。

（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；

**5.1.2 修改 OpenFile::WriteAt()**

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。如果写入为0或指针溢出，则返回错误代码-1。如果需要文件大小不足，则进行文件扩展。

1. **int** OpenFile::WriteAt(**char** \*from, **int** numBytes, **int** position)
2. {
3. **int** fileLength = hdr->FileLength();
4. **int** i, firstSector, lastSector, numSectors;
5. **bool** firstAligned, lastAligned;
6. **char** \*buf;
8. **if** ((numBytes <= 0) || (position > fileLength)) //约束 1
9. **return** -1;
10. **if** ((position + numBytes) > fileLength)
11. { //约束 2
12. **int** incrementBytes = (position + numBytes) - fileLength;
13. BitMap \*freeBitMap = fileSystem->getBitMap();
14. **bool** hdrRet;
15. hdrRet = hdr->Allocate(freeBitMap, fileLength, incrementBytes);
16. **if** (!hdrRet)                                                    // Insuficient Disk Space, or File is Too Big
17. **return** -1;
18. fileSystem->setBitMap(freeBitMap);
19. **delete** freeBitMap;
20. }
22. DEBUG('f', "Writing %d bytes at %d, from file of length %d.\n",
23. numBytes, position, fileLength);
25. firstSector = divRoundDown(position, SectorSize);
26. lastSector = divRoundDown(position + numBytes - 1, SectorSize);
27. numSectors = 1 + lastSector - firstSector;
29. buf = **new** **char**[numSectors \* SectorSize];
31. firstAligned = (**bool**)(position == (firstSector \* SectorSize));
32. lastAligned = (**bool**)((position + numBytes) == ((lastSector + 1) \* SectorSize));
34. // read in first and last sector, if they are to be partially modified
35. **if** (!firstAligned)
36. ReadAt(buf, SectorSize, firstSector \* SectorSize);
37. **if** (!lastAligned && ((firstSector != lastSector) || firstAligned))
38. ReadAt(&buf[(lastSector - firstSector) \* SectorSize],
39. SectorSize, lastSector \* SectorSize);
41. // copy in the bytes we want to change
42. bcopy(from, &buf[position - (firstSector \* SectorSize)], numBytes);
44. // write modified sectors back
45. **for** (i = firstSector; i <= lastSector; i++)
46. synchDisk->WriteSector(hdr->ByteToSector(i \* SectorSize),
47. &buf[(i - firstSector) \* SectorSize]);
48. **delete**[] buf;
49. **return** numBytes;
50. }

**5.1.3 修改 FileSystem 类，增加 setBitMap()与 getBitMap()**

对文件进行扩展需要使用和修改空闲块表，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。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

1. BitMap \*FileSystem::getBitMap()
2. {
3. //NumSectors: DISK 上总扇区数（共有 32\*32=1024 个扇区）
4. BitMap \*freeBitMap = **new** BitMap(NumSectors);
5. freeBitMap->FetchFrom(freeMapFile);
6. **return** freeBitMap;
7. }
9. **void** FileSystem::setBitMap(BitMap \*freeMap)
10. {
11. freeMap->WriteBack(freeMapFile);
12. }

**5.1.4 修改OpenFile::OpenFile()及 OpenFile::WriteBack()**

1、将修改后的文件头写回硬盘，OpenFile 类维护了一个 FileHeader类对象hdr，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(int sectorNumber)函数将一个文件的头文件写到硬盘的扇区sectorNumber中；因此，需要在 OpenFile 类中定义一个私有变量，如int hdrSector，在构造函数中记录该文件头所在的扇区号，以便 FetchFrom()函数 WriteBack()函数使用。

1. OpenFile::OpenFile(**int** sector)
2. {
3. hdr = **new** FileHeader;
4. hdr->FetchFrom(sector);
5. seekPosition = 0;
6. hdrSector=sector;
7. }

2、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。

1. **void** OpenFile::WriteBack()
2. {
3. hdr-> WriteBack(hdrSector);
4. }

**5.1.5 修改 FileHeader::Allocate()**

1、在需要扩展情况下，为要写入的文件数据分配硬盘空间，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. **bool** FileHeader::Allocate(BitMap \*freeMap, **int** fileSize, **int** incrementBytes)
2. {
3. **if** (numSectors > 30) //限定每个文件最多可分配 30 个扇区
4. **return** **false**;    //超出限定的文件大小
5. **if** ((fileSize == 0) && (incrementBytes > 0))
6. {                                //在一个空文件后追加数据
7. **if** (freeMap->NumClear() < 1) //至少需要一个扇区块
8. **return** **false**;            //磁盘已满，无空闲扇区可分配
9. //为添加数据先分配一个空闲磁盘块，并更新文件头信息
10. dataSectors[0] = freeMap->Find();
11. numSectors = 1;
12. numBytes = 0;
13. }
14. numBytes = fileSize;
15. **int** offset = numBytes % SectorSize; //原文件最后一个扇区块数据偏移量
16. **int** newSectorBytes = incrementBytes - (SectorSize - (offset + 1));
17. //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
18. **if** (newSectorBytes <= 0)
19. {
20. numBytes = numBytes + incrementBytes; //更新文件头中的文件大小
21. **return** TRUE;
22. }
23. //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
24. **int** moreSectors = divRoundUp(newSectorBytes, SectorSize); //新加扇区块数
25. **if** (numSectors + moreSectors > 30)
26. **return** FALSE;                      //文件过大，超出 30 个磁盘块
27. **if** (freeMap->NumClear() < moreSectors) //磁盘无足够的空闲块
28. **return** **false**;
29. //没有超出文件大小的限制，并且磁盘有足够的空闲块
30. **for** (**int** i = numSectors; i < numSectors + moreSectors; i++)
31. dataSectors[i] = freeMap->Find();
32. numBytes = numBytes + incrementBytes; //更新文件大小
33. numSectors = numSectors + moreSectors; //更新文件扇区块数
34. **return** TRUE;
35. }

2、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。

1. FileHeader::FileHeader()
2. {
3. numBytes = 0;                       //文件大小
4. numSectors = 0;                     //文件扇区数
5. **for** (**int** i = 0; i < NumDirect; i++) // NumDirector=30：文件最多拥有的扇区数
6. dataSectors[i] = 0;
7. }

**5.2 文件扩展扩展调试结果**

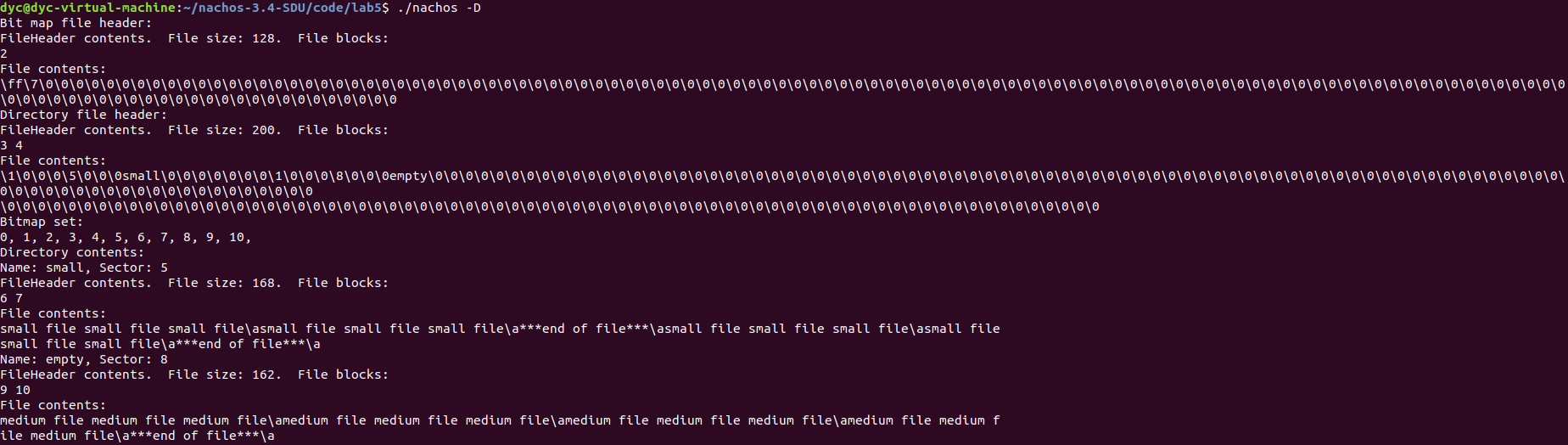
./nachos -cp test/small small

./nachos -ap test/small small

./nachos -cp test/empty empty

./nachos -ap test/empty empty

./nachos -ap test/medium empty



从上述系统转储信息中可以看出，

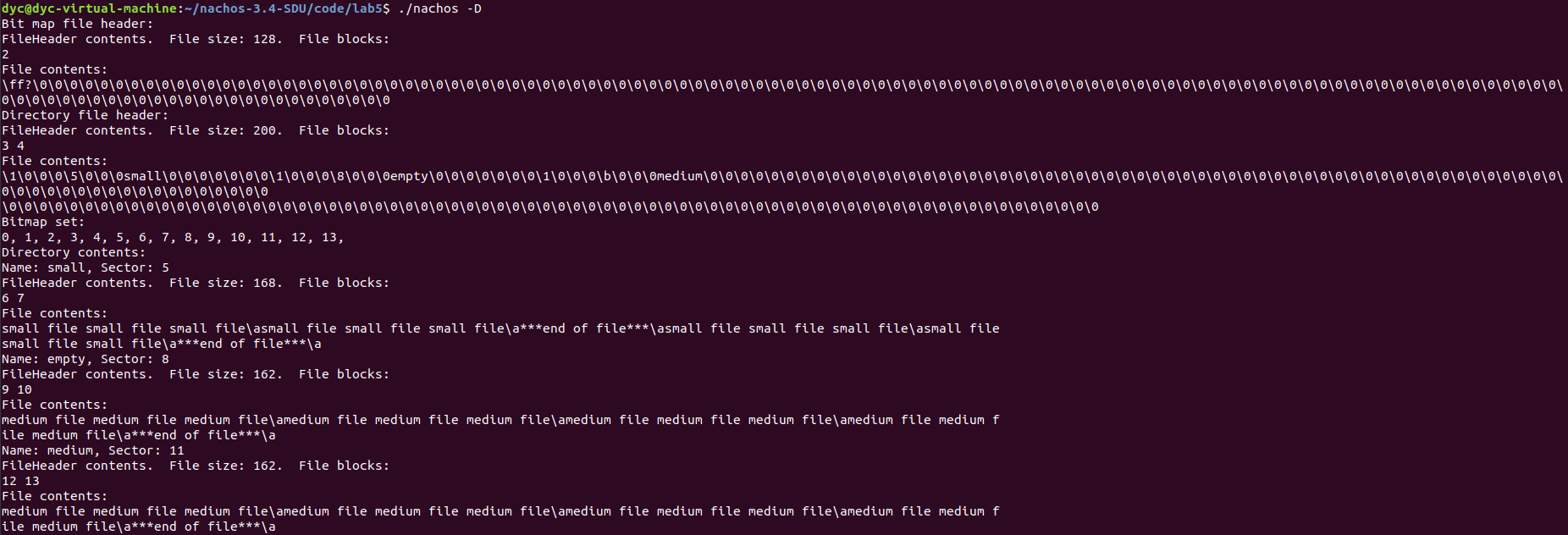
（1）bitmap 文件在Nachos 硬盘的第1个扇区中，大小是128字节（1个扇区），

从中也可以看到该文件的内容；文件bitmap的i-node默认存储在硬盘的第0号扇区中。

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

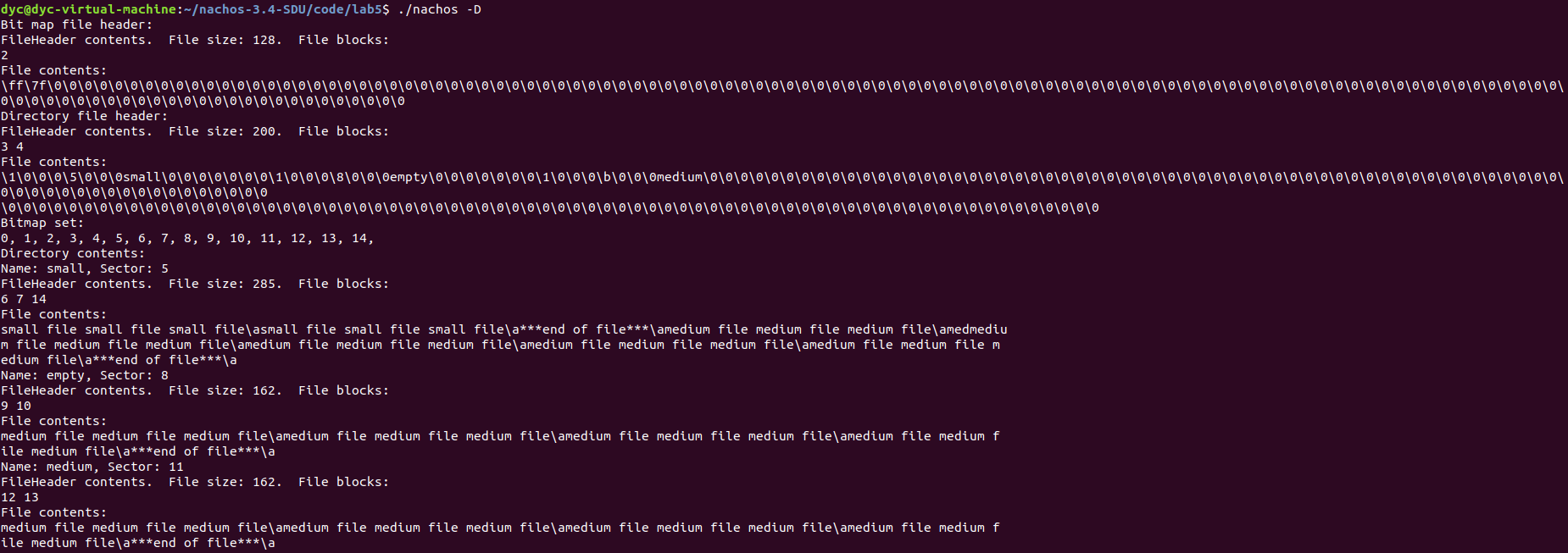
（3）文件 small 大小是 168 字节（占用 2 个扇区），它的 i-node 在第 5 号扇区中， 数据块存储在第 6、7 号扇区中。还需要运行 nachos –hap 等命令以测试你所设计实现的 Nachos 文件系统在各种情 况下都正常运行

./nachos -nap empty medium



./nachos -hap test/medium small

./nachos -hap test/medium small



测试nachos –hap 和./nachos -nap，均正常运行。

**5.3 多级目录设计任务**

**5.3.1 分析多级目录扩展**

1、shell中的常用文件系统操作有cd pwd ls mkdir cat rm rmdir cp rmdir-r，要实现这些命令，就必须实现多级目录。实现的基本思路是：记录文件路径，在创建文件，打开文件，删除文件时进行递归查找修改。

2、文件路径分为绝对路径和相对路径，绝对路径如/test/big，相对路径如big，分两种情况处理。

3、为实现cd ls pwd等命令，需要将当前文件夹信息存储在磁盘上。

4、通过递归，文件夹级联删除的实现，。

**5.3.2 DirectoryEntry类增加成员变量**

1、成员变量int type表示文件类型：0位文件夹，1为文件。

2、成员变量char path[FilePathMaxLen + 1]存储文件路径字符串。

**5.3.3 增加CurDir类**

1、该类存储当前目录有关信息。成员变量int sector表示当前目录扇区号，char path[FilePathMaxLen]表示当前目录绝对路径字符串。

1. **class** CurDir
2. {
3. **public**:
4. CurDir();
6. **void** FetchFrom(OpenFile \*file);
7. **void** WriteBack(OpenFile \*file);
9. **int** sector;
10. **char** path[FilePathMaxLen];
11. };

2、void FetchFrom(OpenFile \*file)从磁盘中获取CurDir数据结果。

1. **void** CurDir::FetchFrom(OpenFile \*file)
2. {
3. (**void**)file->ReadAt((**char** \*)§or, **sizeof**(CurDir), 0);
4. }

3、void WriteBack(OpenFile \*file)从磁盘读取数据写入CurDir对象。

1. **void** CurDir::WriteBack(OpenFile \*file)
2. {
3. (**void**)file->WriteAt((**char** \*)§or, **sizeof**(CurDir), 0);
4. }

**5.3.4 修改FileSystem类**

1、增加如下成员函数和成员变量

1. **int** FindDir(**char** \*name);
2. **bool** FindName(**char** \*name, **char** \*fileName);
3. **bool** CreateTest(**char** \*name, **int** initialSize);
4. OpenFile \*OpenTest(**char** \*name);
5. **bool** RemoveTest(**char** \*name, **int** cascade); // Delete a file (UNIX unlink)
6. **bool** cd(**char** \*name);
7. **void** pwd();
8. **void** ListTest(); // List all the files in the file system
9. **bool** cat(**char** \*name);
11. OpenFile \*curDirFile;
12. OpenFile \*curDirectoryFile;

2、增加int FileSystem::FindDir(char \*name)。name为文件路径字符串，FindDir()获得输入文件名的父文件夹扇区号。输入路径字符串可以是绝对路径或相对路径。循环获得sector号，打开文件。直到到达目标文件的父文件。关键代码如下。

1. **int** FileSystem::FindDir(**char** \*name)
2. {
3. **int** sector;
4. OpenFile \*openFile;
5. Directory \*directory;
6. **if** (name[0] == '/')
7. {
8. sector = DirectorySector;
9. directory = **new** Directory(NumDirEntries);
10. directory->FetchFrom(directoryFile);
11. **int** str\_pos = 1;
12. **int** sub\_str\_pos = 0;
13. **char** sub\_str[10];
14. **while** (str\_pos < strlen(name))
15. {
16. sub\_str[sub\_str\_pos++] = name[str\_pos++];
17. **if** (name[str\_pos] == '/')
18. {
19. sub\_str[sub\_str\_pos] = '\0';
20. **if** ((sector = directory->Find(sub\_str)) == -1 || directory->getType(sub\_str))
21. {
22. DEBUG('f', "FindDir\n");
23. **delete** directory;
24. **return** -1;
25. }
27. openFile = **new** OpenFile(sector);
28. directory->FetchFrom(openFile);
29. **delete** openFile;
30. str\_pos++;
31. sub\_str\_pos = 0;
32. }
33. }
34. **delete** directory;
35. }
36. **else**
37. {
38. CurDir \*curDir = **new** CurDir;
39. curDir->FetchFrom(curDirFile);
40. sector = curDir->sector;
41. **delete** curDir;
43. directory = **new** Directory(NumDirEntries);
44. directory->FetchFrom(curDirectoryFile);
45. **int** str\_pos = 0;
46. **int** sub\_str\_pos = 0;
47. **char** sub\_str[10];
48. **while** (str\_pos < strlen(name))
49. {
50. sub\_str[sub\_str\_pos++] = name[str\_pos++];
51. **if** (name[str\_pos] == '/')
52. {
53. sub\_str[sub\_str\_pos] = '\0';
54. **if** ((sector = directory->Find(sub\_str)) == -1 || directory->getType(sub\_str))
55. {
56. DEBUG('f', "FindDir\n");
57. **delete** directory;
58. **return** -1;
59. }
61. openFile = **new** OpenFile(sector);
62. directory->FetchFrom(openFile);
63. **delete** openFile;
64. str\_pos++;
65. sub\_str\_pos = 0;
66. }
67. }
68. DEBUG('f', "%d\n", sector);
69. **delete** directory;
70. }
71. **return** sector;
72. }

3、增加bool FileSystem::FindName(char \*name, char \*fileName)对输入路径字符串进行处理，获得文件名，如对/test/big处理，获得big字符串，存储在fileName指向内存中。

1. **bool** FileSystem::FindName(**char** \*name, **char** \*fileName)
2. {
3. **int** pos = -1;
4. **for** (**int** i = strlen(name) - 1; i >= 0; i--)
5. **if** (name[i] == '/')
6. {
7. pos = i + 1;
8. **break**;
9. }
10. **if** (pos == -1)
11. strcpy(fileName, name);
12. **else**
13. {
14. **int** i;
15. **for** (i = 0; i < strlen(name) - pos; i++)
16. fileName[i] = name[pos + i];
17. fileName[i] = '\0';
18. }
19. **return** **true**;
20. }

4、修改bool FileSystem::CreateTest(char \*name, int initialSize)。如initialSize = -1则创建文件夹。首先调用FindDir()和FindName()获得相应父文件夹和文件名。处理相对地址字符串，通过拼接字符串方式转换为绝对地址字符串。完成文件创建，创建文件夹。关键代码如下。

1. **bool** FileSystem::CreateTest(**char** \*name, **int** initialSize)
2. {
3. OpenFile \*openFile;
4. Directory \*directory;
5. BitMap \*freeMap;
6. FileHeader \*hdr;
8. **int** sector;
9. **int** dir\_sector;
10. **char** file\_name[FileNameMaxLen + 1];
11. DEBUG('f', "CreateTest file %s, size %d\n", name, initialSize);
13. **if** (((dir\_sector = FindDir(name)) == -1) || !FindName(name, file\_name))
14. {
15. DEBUG('f', "CreateTest findDir fail\n");
16. **return** **false**;
17. }
19. openFile = **new** OpenFile(dir\_sector);
20. directory = **new** Directory(NumDirEntries);
21. directory->FetchFrom(openFile);
23. **if** (directory->Find(file\_name) != -1)
24. {
25. DEBUG('f', "CreateTest find(filename) fail\n");
26. **delete** openFile;
27. **delete** directory;
28. **return** **false**; // file is already in directory
29. }
30. freeMap = **new** BitMap(NumSectors);
31. freeMap->FetchFrom(freeMapFile);
32. sector = freeMap->Find(); // find a sector to hold the file header
33. **if** (sector == -1)
34. {
35. DEBUG('f', "CreateTest freeMap->Find() fail\n");
36. **delete** openFile;
37. **delete** directory;
38. **delete** freeMap;
39. **return** **false**; // no free block for file header
40. }
42. //处理相对地址字符
43. **if** (name[0] != '/')
44. {
45. DEBUG('f', "CreateTest name[0] != '/'\n");
46. **char** tmp\_file\_name[FileNameMaxLen + 1];
47. CurDir \*curDir = **new** CurDir;
48. curDir->FetchFrom(curDirFile);
49. strcpy(tmp\_file\_name, curDir->path);
50. **if** (!strcmp(tmp\_file\_name, "/"))
51. {
52. strcat(tmp\_file\_name, name);
53. strcpy(name, tmp\_file\_name);
54. }
55. **else**
56. {
57. strcat(tmp\_file\_name, "/");
58. strcat(tmp\_file\_name, name);
59. strcpy(name, tmp\_file\_name);
60. }
61. **delete** curDir;
62. }
64. **if** (initialSize == -1)
65. {
66. **if** (!directory->AddTest(file\_name, name, sector, DirType))
67. {
68. DEBUG('f', "CreateTest AddTest fail\n");
69. **delete** openFile;
70. **delete** directory;
71. **delete** freeMap;
72. **return** **false**;
73. }
74. hdr = **new** FileHeader;
75. initialSize = DirectoryFileSize;
76. **if** (!hdr->Allocate(freeMap, initialSize))
77. {
78. DEBUG('f', "CreateTest hdr->Allocate fail\n");
79. **delete** openFile;
80. **delete** directory;
81. **delete** freeMap;
82. **delete** hdr;
83. **return** **false**;
84. }
85. hdr->WriteBack(sector);
86. OpenFile \*tmpOpenFile = **new** OpenFile(sector);
87. Directory \*tmpDirectory = **new** Directory(NumDirEntries);
88. tmpDirectory->WriteBack(tmpOpenFile);
89. **delete** tmpOpenFile;
90. **delete** tmpDirectory;
92. directory->WriteBack(openFile);
93. freeMap->WriteBack(freeMapFile);
94. }
95. **else**
96. {
97. **if** (!directory->AddTest(file\_name, name, sector, FileType))
98. {
99. DEBUG('f', "CreateTest AddTest fail\n");
100. **delete** openFile;
101. **delete** directory;
102. **delete** freeMap;
103. **return** **false**;
104. }
105. hdr = **new** FileHeader;
106. **if** (!hdr->Allocate(freeMap, initialSize))
107. {
108. DEBUG('f', "CreateTest hdr->Allocate fail\n");
109. **delete** openFile;
110. **delete** directory;
111. **delete** freeMap;
112. **delete** hdr;
113. **return** **false**;
114. }
115. hdr->WriteBack(sector);
116. directory->WriteBack(openFile);
117. freeMap->WriteBack(freeMapFile);
118. }
119. **delete** freeMap;
120. **delete** hdr;
122. **delete** openFile;
123. **delete** directory;
124. **return** **true**;
125. }
126. 增加OpenFile \*FileSystem::OpenTest(char \*name)。参数name为路径字符串，可能为相对地址，可能为绝对地址。首先调用FindDir()和FindName()获得相应父文件夹和文件名。通过扇区号打开文件并返回。
127. OpenFile \*FileSystem::OpenTest(**char** \*name)
128. {
129. Directory \*directory;
130. OpenFile \*resFile = NULL;
131. OpenFile \*openFile;
132. **int** sector;
133. **int** dir\_sector;
134. **char** file\_name[FileNameMaxLen + 1];
136. DEBUG('f', "OpenTest file %s\n", name);
138. **if** (((dir\_sector = FindDir(name)) == -1) || !FindName(name, file\_name))
139. {
140. DEBUG('f', "OpenTest FindDir\n");
141. **return** resFile;
142. }
144. openFile = **new** OpenFile(dir\_sector);
145. directory = **new** Directory(NumDirEntries);
146. directory->FetchFrom(openFile);
148. **if** ((sector = directory->Find(file\_name)) != -1)
149. resFile = **new** OpenFile(sector); // name was found in directory
150. **delete** openFile;
151. **delete** directory;
152. **return** resFile; // return NULL if not found
153. }

5、增加bool FileSystem::RemoveTest(char \*name, int cascade)。参数name为路径字符串，可能为相对地址，可能为绝对地址。参数cascade为是否级联删除的表示符，1位级联删除。首先调用FindDir()和FindName()获得相应父文件夹和文件名。打开父文件夹调用Find()寻找要删除的文件是否存在。

如果要删除的文件是文件夹，则调用isEmpty()需要判断文件夹是否为空，如为空则可以直接删除。如不为空的话，不允许级联删除的情况下删除失败并返回，允许级联删除的情况下递归删除此文件夹下的每一个文件后，再删除此文件。

1. **bool** FileSystem::RemoveTest(**char** \*name, **int** cascade)
2. {
3. Directory \*directory;
4. BitMap \*freeMap;
5. FileHeader \*fileHdr;
6. OpenFile \*openFile;
7. **int** sector;
8. **int** dir\_sector;
9. **char** file\_name[FileNameMaxLen + 1];
11. **if** (((dir\_sector = FindDir(name)) == -1) || !FindName(name, file\_name))
12. {
13. DEBUG('f', "RemoveTest %s FindDir fasle\n", name);
14. **return** **false**;
15. }
17. openFile = **new** OpenFile(dir\_sector);
18. directory = **new** Directory(NumDirEntries);
19. directory->FetchFrom(openFile);
20. **if** ((sector = directory->Find(file\_name)) == -1)
21. {
22. DEBUG('f', "RemoveTest %s Find fasle\n", name);
23. **delete** openFile;
24. **delete** directory;
25. **return** **false**; // file not found
26. }
27. **if** (directory->getType(file\_name) == 0)
28. {
29. OpenFile \*removeFile = **new** OpenFile(sector);
30. Directory \*removeDirectory = **new** Directory(NumDirEntries);
31. removeDirectory->FetchFrom(removeFile);
32. **if** (!removeDirectory->isEmpty())
33. {
34. **if** (cascade == 0)
35. {
36. DEBUG('f', "RemoveTest %s -r fasle\n", name);
37. **delete** openFile;
38. **delete** directory;
39. **delete** removeFile;
40. **delete** removeDirectory;
41. **return** **false**;
42. }
43. **int** tableSize;
44. DirectoryEntry \*table;
45. removeDirectory->getTable(tableSize, table);
46. **for** (**int** i = 0; i < tableSize; i++)
47. **if** (table[i].inUse && !RemoveTest(table[i].path, 1))
48. {
49. DEBUG('f', "RemoveTest %s -r fasle\n", name);
50. **delete** openFile;
51. **delete** directory;
52. **delete** removeFile;
53. **delete** removeDirectory;
54. **return** **false**;
55. }
56. }
57. **delete** removeFile;
58. **delete** removeDirectory;
59. }
60. fileHdr = **new** FileHeader;
61. fileHdr->FetchFrom(sector);
63. freeMap = **new** BitMap(NumSectors);
64. freeMap->FetchFrom(freeMapFile);
66. fileHdr->Deallocate(freeMap); // remove data blocks
67. freeMap->Clear(sector);       // remove header block
68. directory->Remove(file\_name);
70. freeMap->WriteBack(freeMapFile); // flush to disk
71. directory->WriteBack(openFile);  // flush to disk
72. **delete** fileHdr;
73. **delete** directory;
74. **delete** freeMap;
75. **return** **true**;
76. }

6、增加bool FileSystem::cd(char \*name)变更当前目录。参数name为路径字符串，可能为相对地址，可能为绝对地址。首先调用FindDir()和FindName()获得相应父文件夹和文件名。打开父文件夹调用Find()寻找要删除的文件是否存在，判断文件类型是否是文件夹。将相对路径字符串转换为绝对路径字符串。

将当前目录信息存储到磁盘上，可以在下次启动时读取。用到CurDir类存储这些信息。目录变更后并重新打开curDirectoryFile为当前目录。

1. **bool** FileSystem::cd(**char** \*name)
2. {
3. OpenFile \*openFile;
4. Directory \*directory;
5. **int** sector;
6. **int** dir\_sector;
7. **char** file\_name[FileNameMaxLen + 1];
9. **if** (!strcmp(name, "/"))
10. {
11. sector = 1;
12. CurDir \*curDir = **new** CurDir;
13. curDir->sector = sector;
14. strcpy(curDir->path, name);
15. curDir->WriteBack(curDirFile);
16. **delete** curDir;
17. **delete** curDirectoryFile;
18. curDirectoryFile = **new** OpenFile(sector);
19. **return** **true**;
20. }
22. **if** (((dir\_sector = FindDir(name)) == -1) || !FindName(name, file\_name))
23. {
24. DEBUG('f', "cd %s FindDir fasle\n", name);
25. **return** **false**;
26. }
27. openFile = **new** OpenFile(dir\_sector);
28. directory = **new** Directory(NumDirEntries);
29. directory->FetchFrom(openFile);
30. **if** ((sector = directory->Find(file\_name)) == -1 || directory->getType(file\_name))
31. {
32. DEBUG('f', "cd %s Find fasle\n", name);
33. **delete** openFile;
34. **delete** directory;
35. **return** **false**;
36. }
38. //处理相对地址字符
39. **if** (name[0] != '/')
40. {
41. DEBUG('f', "CreateTest name[0] != '/'\n");
42. **char** tmp\_file\_name[FileNameMaxLen + 1];
43. CurDir \*curDir = **new** CurDir;
44. curDir->FetchFrom(curDirFile);
45. strcpy(tmp\_file\_name,curDir->path);
46. **if** (!strcmp(tmp\_file\_name, "/"))
47. {
48. strcat(tmp\_file\_name, name);
49. strcpy(name, tmp\_file\_name);
50. }
51. **else**
52. {
53. strcat(tmp\_file\_name, "/");
54. strcat(tmp\_file\_name, name);
55. strcpy(name, tmp\_file\_name);
56. }
57. **delete** curDir;
58. }
60. CurDir \*curDir = **new** CurDir;
61. curDir->sector = sector;
62. strcpy(curDir->path, name);
63. curDir->WriteBack(curDirFile);
64. **delete** curDir;
65. **delete** curDirectoryFile;
66. curDirectoryFile = **new** OpenFile(sector);
68. **delete** openFile;
69. **delete** directory;
70. **return** **true**;
71. }

8、增加void FileSystem::pwd()显示当前目录。获取磁盘上存储的当前目录信息到CurDir类对象，读取并输出。

1. **void** FileSystem::pwd()
2. {
3. CurDir \*curDir = **new** CurDir;
4. curDir->FetchFrom(curDirFile);
5. printf("curpath: %s\n", curDir->path);
6. }

9、增加void FileSystem::ListTest()显示当前目录中的文件。curDirectoryFile为当前目录打开文件。

1. **void** FileSystem::ListTest()
2. {
3. Directory \*directory = **new** Directory(NumDirEntries);
5. directory->FetchFrom(curDirectoryFile);
6. directory->List();
7. **delete** directory;
8. }
9. 修改FileSystem::FileSystem(bool format)。初始化时加入对CurDir对象的磁盘存储。初始化CurDir对象。CurDir对象存储到磁盘。非格式化时从磁盘读取CurDir对象，关键代码如下。
10. FileSystem::FileSystem(**bool** format)
11. {
12. DEBUG('f', "Initializing the file system.\n");
13. **if** (format)
14. {
15. BitMap \*freeMap = **new** BitMap(NumSectors);
16. Directory \*directory = **new** Directory(NumDirEntries);
17. CurDir \*curDir = **new** CurDir;
18. FileHeader \*mapHdr = **new** FileHeader;
19. FileHeader \*dirHdr = **new** FileHeader;
20. FileHeader \*curDirHdr = **new** FileHeader;
22. DEBUG('f', "Formatting the file system.\n");
24. // First, allocate space for FileHeaders for the directory and bitmap
25. // (make sure no one else grabs these!)
26. freeMap->Mark(FreeMapSector);
27. freeMap->Mark(DirectorySector);
28. freeMap->Mark(CurDirSector);
30. // Second, allocate space for the data blocks containing the contents
31. // of the directory and bitmap files.  There better be enough space!
33. ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
34. ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));
35. ASSERT(curDirHdr->Allocate(freeMap, CurDirFileSize));
37. // Flush the bitmap and directory FileHeaders back to disk
38. // We need to do this before we can "Open" the file, since open
39. // reads the file header off of disk (and currently the disk has garbage
40. // on it!).
42. DEBUG('f', "Writing headers back to disk.\n");
43. mapHdr->WriteBack(FreeMapSector);
44. dirHdr->WriteBack(DirectorySector);
45. curDirHdr->WriteBack(CurDirSector);
47. // OK to open the bitmap and directory files now
48. // The file system operations assume these two files are left open
49. // while Nachos is running.
51. freeMapFile = **new** OpenFile(FreeMapSector);
52. directoryFile = **new** OpenFile(DirectorySector);
53. curDirFile = **new** OpenFile(CurDirSector);
55. DEBUG('f', "Writing bitmap and directory back to disk.\n");
56. freeMap->WriteBack(freeMapFile); // flush changes to disk
57. directory->WriteBack(directoryFile);
59. curDir->sector = 1;
60. strcpy(curDir->path, "/");
61. curDir->WriteBack(curDirFile);
62. curDirectoryFile = **new** OpenFile(curDir->sector);
64. **if** (DebugIsEnabled('f'))
65. {
66. freeMap->Print();
67. directory->Print();
68. }
69. **delete** freeMap;
70. **delete** directory;
71. **delete** curDir;
72. **delete** mapHdr;
73. **delete** dirHdr;
74. **delete** curDirHdr;
75. }
76. **else**
77. {
78. // if we are not formatting the disk, just open the files representing
79. // the bitmap and directory; these are left open while Nachos is running
80. freeMapFile = **new** OpenFile(FreeMapSector);
81. directoryFile = **new** OpenFile(DirectorySector);
82. curDirFile = **new** OpenFile(CurDirSector);
84. CurDir \*curDir = **new** CurDir;
85. curDir->FetchFrom(curDirFile);
86. curDirectoryFile = **new** OpenFile(curDir->sector);
88. **delete** curDir;
89. }
90. }

11、增加bool FileSystem::cat(char \*name)显示文件或文件夹的内容。参数name为路径字符串，可能为相对地址，可能为绝对地址。首先调用FindDir()和FindName()获得相应父文件夹和文件名。打开父文件夹调用Find()寻找要删除的文件是否存在，判断文件类型是否是文件夹。将相对路径字符串转换为绝对路径字符串。判断为文件或为文件夹，分两种清空进行显示，关键代码如下。

1. **bool** FileSystem::cat(**char** \*name)
2. {
3. Directory \*directory;
4. OpenFile \*openFile;
5. **int** sector;
6. **int** dir\_sector;
7. **char** file\_name[FileNameMaxLen + 1];
9. DEBUG('f', "cat file %s\n", name);
11. **if** (!strcmp(name, "/"))
12. {
13. Print();
14. **return** **true**;
15. }
17. **if** (((dir\_sector = FindDir(name)) == -1) || !FindName(name, file\_name))
18. {
19. DEBUG('f', "cat FindDir\n");
20. **return** **false**;
21. }
23. openFile = **new** OpenFile(dir\_sector);
24. directory = **new** Directory(NumDirEntries);
25. directory->FetchFrom(openFile);
27. **if** ((sector = directory->Find(file\_name)) == -1)
28. {
29. **delete** openFile;
30. **delete** directory;
31. **return** **false**;
32. }
33. directory->cat(file\_name);
34. **if** (!directory->getType(file\_name))
35. {
36. **delete** openFile;
37. openFile = **new** OpenFile(sector);
38. directory->FetchFrom(openFile);
39. directory->Print();
40. }
41. **delete** openFile;
42. **delete** directory;
43. **return** **true**;
44. }

**5.3.5 修改Directory类**

1、Directory类增加如下成员函数。

1. **int** getType(**char** \*name);
3. **bool** isEmpty();
5. **bool** AddTest(**char** \*name, **char** \*path, **int** newSector, **int** type);
7. **void** getTable(**int** &tableSize, DirectoryEntry \*&table);
9. **bool** cat(**char** \*name);

2、增加int getType(char \*name)获得文件名为name的文件的类型。

1. **int** Directory::getType(**char** \*name)
2. {
3. **int** i = FindIndex(name);
4. **if** (i != -1)
5. **return** table[i].type;
6. **return** -1;
7. }
8. 增加bool isEmpty()判断目录是否为空。
9. **bool** Directory::isEmpty()
10. {
11. **for** (**int** i = 0; i < tableSize; i++)
12. **if** (table[i].inUse)
13. **return** **false**;
14. **return** **true**;
15. }
16. bool AddTest(char \*name, char \*path, int newSector, int type)向目录表中添加DirectoryEntry项。name为文件名字符串。path为文件绝对路径字符串。newSector为文件头扇区号。type为文件类型，0为目录，1为文件。
17. **bool** Directory::AddTest(**char** \*name, **char** \*path, **int** newSector, **int** type)
18. {
19. **if** (FindIndex(name) != -1)
20. **return** FALSE;
22. **for** (**int** i = 0; i < tableSize; i++)
23. **if** (!table[i].inUse)
24. {
25. table[i].inUse = TRUE;
26. strncpy(table[i].name, name, FileNameMaxLen + 1);
27. strncpy(table[i].path, path, FilePathMaxLen + 1);
28. table[i].sector = newSector;
29. table[i].type = type;
30. **return** TRUE;
31. }
32. **return** FALSE; // no space.  Fix when we have extensible files.
33. }

5、void getTable(int &tableSize, DirectoryEntry \*&table)获取目录中的文件表和目录项数量。在级联删除时被调用。

1. **void** Directory::getTable(**int** &tableSize, DirectoryEntry \*&table)
2. {
3. tableSize = **this**->tableSize;
4. table = **this**->table;
5. }

6、bool cat(char \*name)输出目录中文件名为name的文件内容。

1. **bool** Directory::cat(**char** \*name)
2. {
3. FileHeader \*hdr = **new** FileHeader;
5. **int** i = FindIndex(name);
6. **if** (i == -1)
7. **return** **false**;
8. printf("File contents:\n");
9. printf("Name: %s, Path: %s, Sector: %d, Type: %s\n", table[i].name, table[i].path, table[i].sector, table[i].type ? "File" : "Directory");
10. hdr->FetchFrom(table[i].sector);
11. hdr->Print();
12. printf("\n");
13. **delete** hdr;
14. **return** **true**;
15. }

**5.3.6 修改fstest.cc**

增加如下函数，将原函数中调用的FileSystem类中的函数用上述函数进行替换，如Create()替换为CreateTest()，Open()替换为OpenTest()，Remove()替换为RemoveTest()，List()替换为ListTest()。

1. **extern** **void** CopyTest(**char** \*unixFile, **char** \*nachosFile);
2. **extern** **void** AppendTest(**char** \*unixFile, **char** \*nachosFile, **int** half);
3. **extern** **void** NAppendTest(**char** \*nachosFileFrom, **char** \*nachosFileTo);

**4.5.7 修改main.cc**

增加-cd -pwd -ls -cat -mkdir -rmdir -rmdir-r -rm操作，同时更新-cp -ap -hap -nap等操作。

1. **else** **if** (!strcmp(\*argv, "-mkdir"))
2. { // copy from UNIX to Nachos
3. ASSERT(argc > 1);
4. fileSystem->CreateTest(\*(argv + 1), -1);
5. argCount = 2;
6. }
7. **else** **if** (!strcmp(\*argv, "-cptest"))
8. { // copy from UNIX to Nachos
9. ASSERT(argc > 2);
10. CopyTest(\*(argv + 1), \*(argv + 2));
11. argCount = 3;
12. }
13. **else** **if** (!strcmp(\*argv, "-rm"))
14. { // remove Nachos file
15. ASSERT(argc > 1);
16. fileSystem->RemoveTest(\*(argv + 1), 0);
17. argCount = 2;
18. }
19. **else** **if** (!strcmp(\*argv, "-rmdir"))
20. { // remove Nachos dir
21. ASSERT(argc > 1);
22. fileSystem->RemoveTest(\*(argv + 1), 0);
23. argCount = 2;
24. }
25. **else** **if** (!strcmp(\*argv, "-rmdir-r"))
26. { // remove Nachos dir
27. ASSERT(argc > 1);
28. fileSystem->RemoveTest(\*(argv + 1), 1);
29. argCount = 2;
30. }
31. **else** **if** (!strcmp(\*argv, "-aptest"))
32. { // append from UNIX to Nachos
33. ASSERT(argc > 2);
34. AppendTest(\*(argv + 1), \*(argv + 2), 0);
35. argCount = 3;
36. }
37. **else** **if** (!strcmp(\*argv, "-haptest"))
38. {
39. // cut half and append from UNIX to Nachos
40. ASSERT(argc > 2);
41. AppendTest(\*(argv + 1), \*(argv + 2), 1);
42. argCount = 3;
43. }
44. **else** **if** (!strcmp(\*argv, "-naptest"))
45. { // append from Nachos to Nachos
46. ASSERT(argc > 2);
47. NAppendTest(\*(argv + 1), \*(argv + 2));
48. argCount = 3;
49. }
50. **else** **if** (!strcmp(\*argv, "-cd"))
51. { // remove Nachos file
52. ASSERT(argc > 1);
53. fileSystem->cd(\*(argv + 1));
54. argCount = 2;
55. }
56. **else** **if** (!strcmp(\*argv, "-pwd"))
57. {
58. fileSystem->pwd();
59. }
60. **else** **if** (!strcmp(\*argv, "-ls"))
61. { // list Nachos directory
62. fileSystem->ListTest();
63. }
64. **else** **if** (!strcmp(\*argv, "-cat"))
65. { // remove Nachos file
66. ASSERT(argc > 1);
67. fileSystem->cat(\*(argv + 1));
68. argCount = 2;
69. }

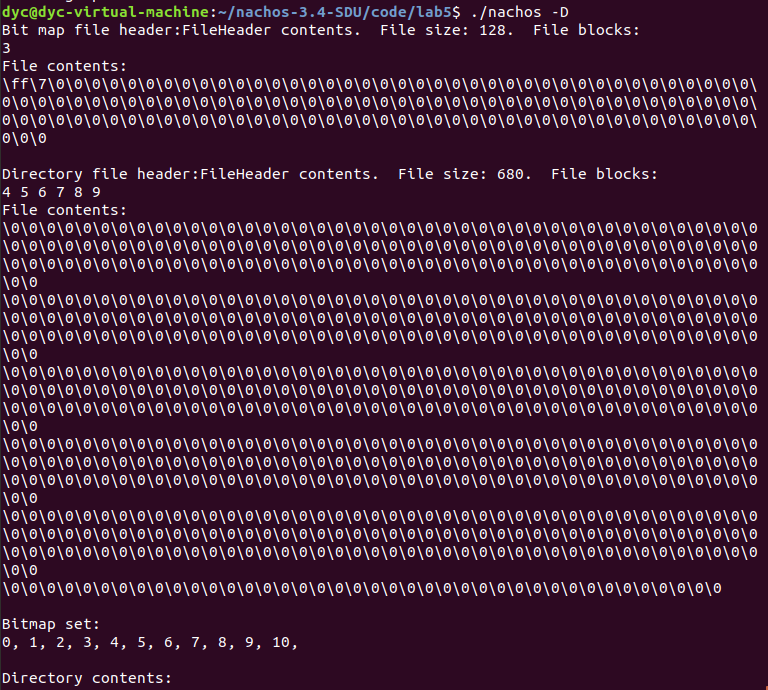
**5.4 多级目录调试结果**

**5.4.1 硬盘格式化**

1、将DISK文件删除。

2、./nachos -f格式化硬盘（在模拟硬盘DISK上创建一个文件系统）。

3、./nachos -D



4、初始化后的扇区分布。

0号扇区为BitMap文件头存储扇区。

1号扇区为Directory文件头存储扇区。

2号扇区为CurDir文件头存储扇区。

3号扇区为BitMap文件内容存储扇区。

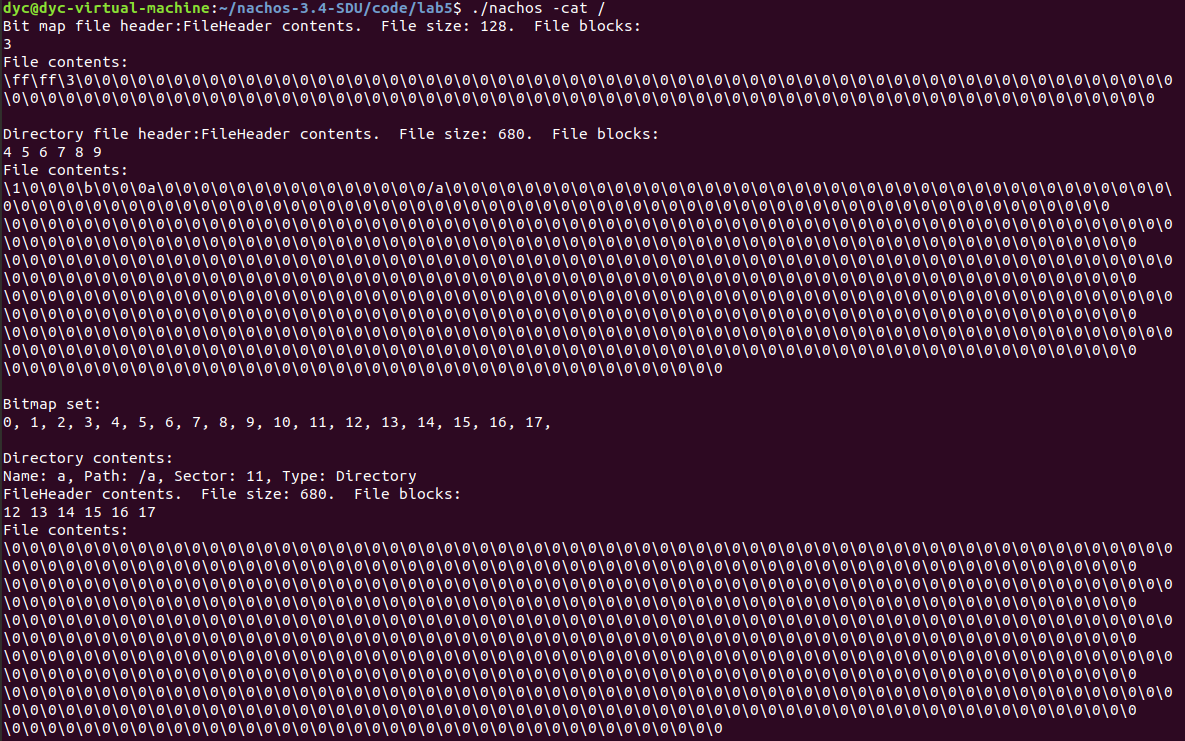
4 5 6 7 8 9号扇区为Directory文件内容存储扇区。

10号扇区为CurDir文件内容存储扇区。

**5.4.2 绝对路径创建/a目录**

1、./nachos -mkdir /a创建文件。

2、./nachos -cat / 输出/目录文件内容。



1. 显示目录文件/a已在根目录下创建完成。11号扇区为/a文件头存储扇区，12 13 14 15 16 17号扇区为/a文件内容存储扇区。

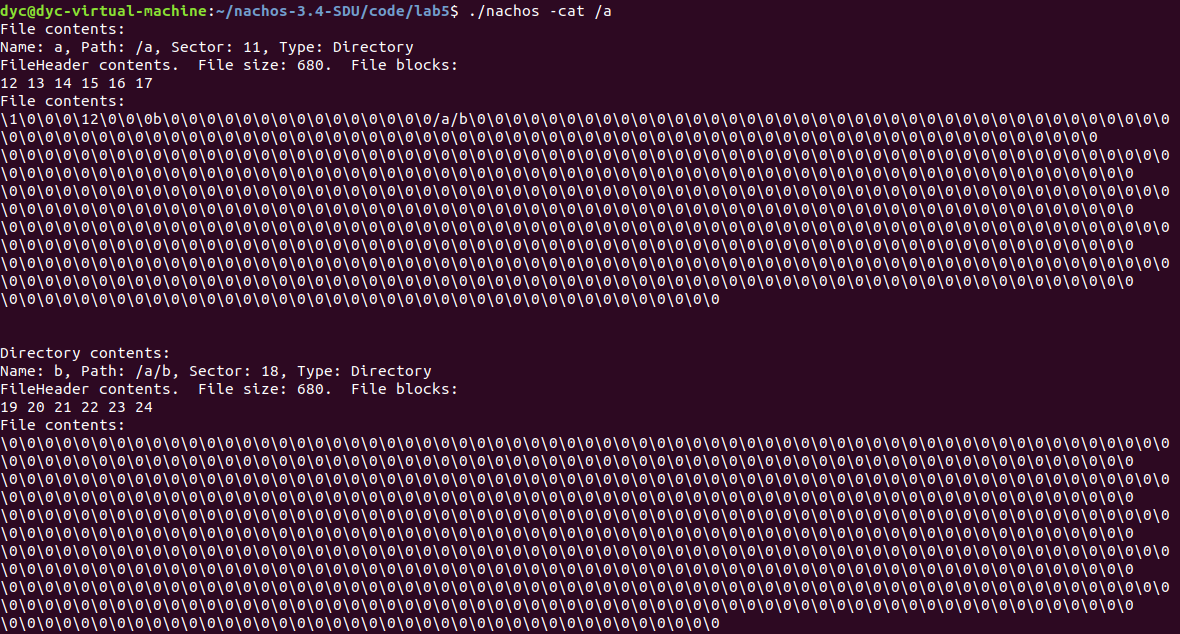
**5.4.3 相对路径创建/a/b目录**

1、./nachos -cd a进入a目录，使用相对路径。

2、./nachos -pwd，显示当前目录为/a



1. ./nachos -mkdir b创建文件，使用相对路径。
2. ./nachos -cat /a 输出/a目录文件内容，使用绝对路径。



显示目录文件/a/b已在/a目录下创建完成。18号扇区为/a文件头存储扇区，19 20 21 22 23 24号扇区为/a/b文件内容存储扇区。

**5.4.4 相对路径复制test/small到/a/b/small，绝对路径复制test/big到/a/b/big**

1、./nachos -cd b进入a目录，使用相对路径。

2、./nachos -pwd，显示当前目录为/a/b



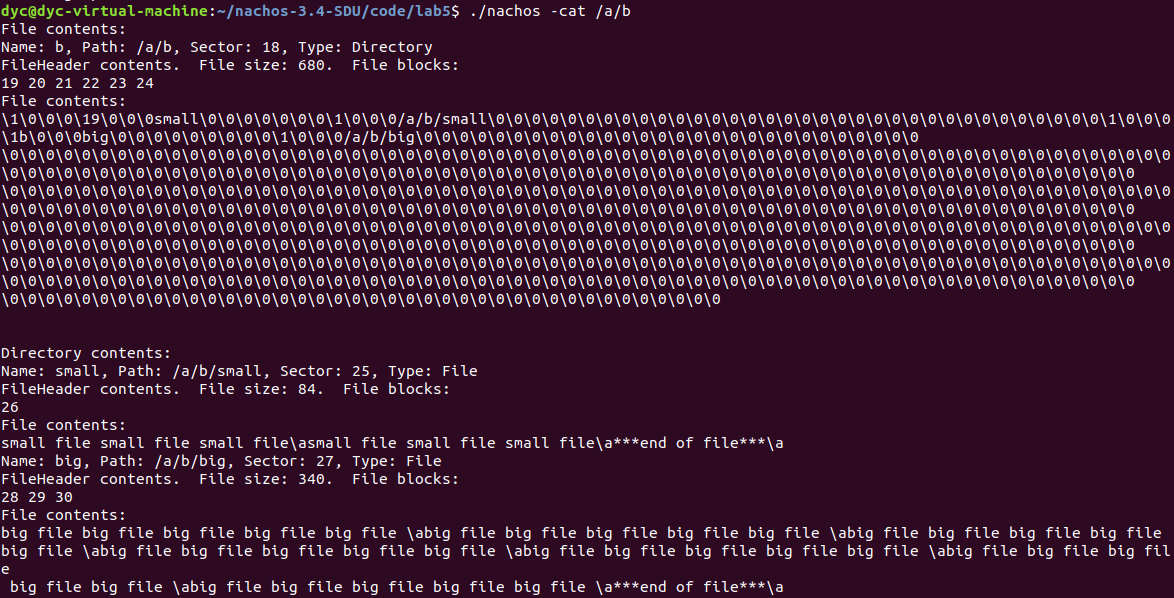
3、./nachos -cp test/small small创建文件，使用相对路径。

4、./nachos -cp test/big /a/b/big创建文件，使用绝对路径。

5、./nachos -ls 输出当前目录/a/b目录文件内容，有两个文件。



1. ./nachos -cat /a/b 输出/a/b目录文件内容，使用绝对路径。



7、显示目录文件/a/b/small /a/b/big已在/a/b目录下负责创建完成。

25号扇区为/a/b/small文件头存储扇区。

26号扇区为/a/b/small文件内容存储扇区。

27号扇区为/a/b/big文件头存储扇区。

28 29 30号扇区为/a/b/big文件内容存储扇区。

**5.4.5 绝对路径级联删除/a**

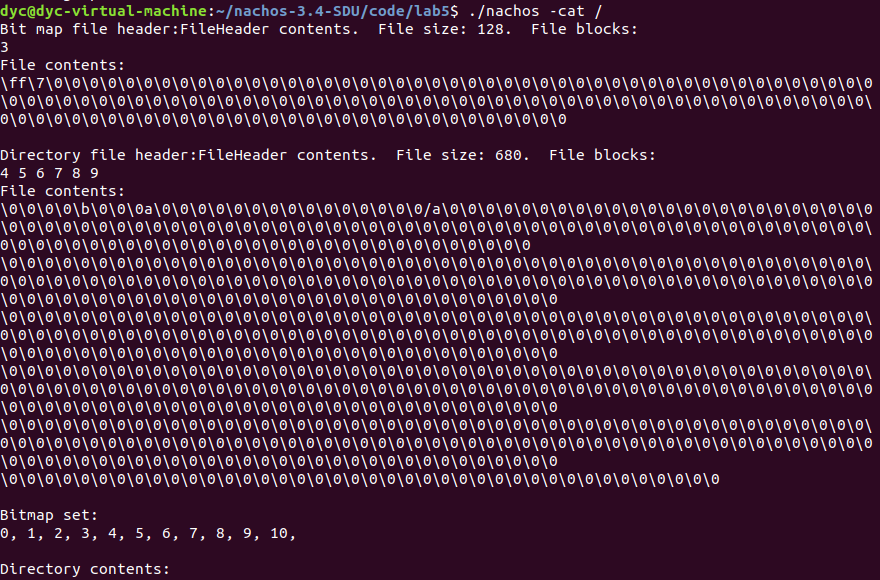
1、./nachos -cd /进入/目录，使用相对路径。

2、./nachos -pwd，显示当前目录为/。



3、./nachos -rmdir-r /a级联删除/a目录，使用绝对路径。

4、./nachos -cat / 输出/目录文件内容，使用绝对路径。



5、恢复初始化后的扇区分布。

0号扇区为BitMap文件头存储扇区。

1号扇区为Directory文件头存储扇区。

2号扇区为CurDir文件头存储扇区。

3号扇区为BitMap文件内容存储扇区。

4 5 6 7 8 9号扇区为Directory文件内容存储扇区。

10号扇区为CurDir文件内容存储扇区。

**6 实验六 Nachos用户程序与系统调用**

**6.1 基础实验代码分析**

**6.1.1 machine.h Instruction类**

1、unsigned int类成员变量value存放二进制指令，char opCode存放操作码，char rs, rt, rd，int extra存放操作数。

2、Decode()方法进行指令译码和分析。

**6.1.2 machine.h Machine类**

1、成员变量char \*mainMemory模拟主存，int registers[NumTotalRegs]模拟cpu寄存器，TranslationEntry \*tlb模拟快表，TranslationEntry \*pageTable模拟系统页表，unsigned int pageTableSize存放当前进程页数。bool singleStep判断是否在每一条模拟指令完成后，返回到调试器中，int runUntilTime表示当模拟时间达到此值时，返回到调试器。

2、Machine::Machine(bool debug)初始化机器的寄存器、主存和TLB，检察当前执行环境是否是小根端。

3、void Machine::RaiseException(ExceptionType which, int badVAddr)抛出异常，将控制从用户模式传输到Nachos内核，因为用户程序调用了系统调用，或者发生了一些异常（例如地址转换失败）。ExceptionType which表示内核陷阱的原因，int badVAddr存放引起陷阱的虚拟地址（如果需要）。如果是地址失效，则存入寄存器。执行延迟加载，确保上一条指令执行完。进入SystemMode。调用ExceptionHandler()进行异常处理，处理完成后返回UserMode模式。

4、ExceptionHandler(ExceptionType which)是异常处理函数，是进入Nachos内核的入口点。当用户程序正在执行时调用，或者执行系统调用，或者生成寻址或算术异常。对于系统调用，以下是调用约定：（system call code -- r2）（arg1 -- r4）（arg2 -- r5）（arg3 -- r6）（arg4 -- r7）。从2号寄存器中取出异常号，分别进行异常处理。

5、void Machine::DumpState()打印模拟CPU寄存器。

6、void Machine::Debugger()根据是否单步调试和到达设置断点进入，实现了简单的gdb功能：设置断点、继续执行、单步调试。

7、int Machine::ReadRegister(int num)读取寄存器并返回，void Machine::WriteRegister(int num, int value)写回寄存器。

8、void Machine::OneInstruction(Instruction \*instr)为解释指令过程，分为取值、分析、执行、访存。访存是通过延迟加载实现。且MIPS为延迟转移，即转移指令后的一条指令绝对会执行，提高流水线性能。如果有任何异常或中断，我们调用异常处理程序，当它返回时，我们返回Run（），它将在循环中重新调用我们。这允许我们从头开始重新执行指令，以防我们的状态发生变化。在系统调用中，操作系统软件必须递增PC，因此在系统调用后立即从指令开始执行。这个例程是可重入的，因为它可以同时调用多次——每个线程执行一个用户代码。我们从不缓存任何数据，从而获得重入性——每次调用时（或在异常或中断时捕获回Nachos内核后），我们总是从头开始模拟，并且在离开之前，我们总是将所有数据存储回机器寄存器和内存。这允许Nachos内核通过控制内存、翻译表和寄存器集的内容来控制我们的行为。

9、void Machine::DelayedLoad(int nextReg, int nextValue)模拟MIPS延迟加载。RaiseException/CheckInterrupts还必须调用delayed load，因为在捕获内核之前必须应用任何延迟的加载，保证前一条指令执行完成。

1. **void** Machine::DelayedLoad(**int** nextReg, **int** nextValue)
2. {
3. registers[registers[LoadReg]] = registers[LoadValueReg];
4. registers[LoadReg] = nextReg;
5. registers[LoadValueReg] = nextValue;
6. registers[0] = 0; // and always make sure R0 stays zero.
7. }

10、void Instruction::Decode()进行指令译码。从指令中获取操作码和操作数。

11、Mult(int a, int b, bool signedArith, int \*hiPtr, int \*loPtr)模拟MIPS乘法指令执行。

12、void Machine::Run()在Nachos上模拟用户级程序的执行。程序启动时由内核调用；从不返回。这个例程是可重入的，因为它可以同时调用多次——每个线程执行一个用户代码。每执行完一条指令后，首先检察中断并处理，然后判断是否进行调试。

1. **for** (;;)
2. {
3. OneInstruction(instr);
4. interrupt->OneTick();
5. **if** (singleStep && (runUntilTime <= stats->totalTicks))
6. Debugger();
7. }

**6.1.3 translate.h TranslationEntry类**

1、定义了转换表中的一个条目——在页表或TLB中。每个条目定义从一个虚拟页到一个物理页的映射。此外，还有一些额外的位用于访问控制（有效和只读）和一些位用于使用信息（使用和脏位）。

2、int virtualPage为虚拟页号，int physicalPage为主存页号，bool valid为有效位，bool readOnly为是否只读，bool use为是否使用过，bool dirty为脏位。

3、模拟的MIPS CPU是小根端，WordToHost(unsigned int word) ShortToHost(unsigned short shortword)

WordToMachine(unsigned int word) ShortToMachine(unsigned short shortword)根据宿主计算机是大根端还是小根端实现数据在模拟的MIPS CPU和宿主计算机之间的转换。

4、ExceptionType Machine::Translate(int virtAddr, int \*physAddr, int size, bool writing)使用页表或TLB将虚拟地址转换为物理地址。检查对齐和其他各种错误，如果一切正常，在translation table条目中设置use/dirty位，并将翻译后的物理地址存储在“physAddr”中。如果有错误，则返回异常的类型。

5、bool Machine::ReadMem(int addr, int size, int \*value)调用Translate()将“addr”处的虚拟内存“size”（1、2或4）字节读取到“value”所指的位置。

6、bool Machine::WriteMem(int addr, int size, int value)调用Translate()将“value”写入到“addr”处的虚拟内存“size”（1、2或4）字节。

**6.1.4 noff.h Segment类**

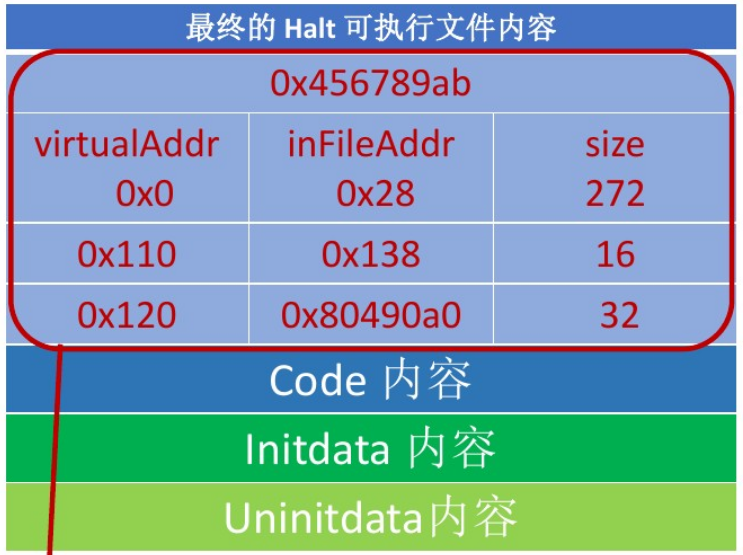
int virtualAddr存放段在虚拟地址空间中的位置，int inFileAddr存放段在可执行文件中的位置，int size存放段的大小。

1. **typedef** **struct** segment {
2. **int** virtualAddr;      /\* location of segment in virt addr space  段在virt addr空间中的位置 \*/
3. **int** inFileAddr;       /\* location of segment in this file  段在文件中的位置 \*/
4. **int** size;         /\* size of segment \*/
5. } Segment;

**6.1.5 noff.h noffHeader类**

1、定义Nachos目标代码格式的数据结构。基本上，我们只知道三种类型的段代码（只读）、初始化数据和未初始化数据。该数据结构存放在可执行文件的头部。

2、int noffMagic应该是NOFFMAGIC，用来进行检查。



3、虚拟地址空间有三个段，Segment code表示代码段，Segment initData表示已初始化数据段，Segment uninitData表示未初始化数据段。实际运行时还有用户栈。

1. **typedef** **struct** noffHeader {
2. **int** noffMagic;       /\* should be NOFFMAGIC \*/
3. Segment code;        /\* executable code segment \*/
4. Segment initData;        /\* initialized data segment \*/
5. Segment uninitData;      /\* uninitialized data segment --
6. \* should be zero'ed before use
7. \*/
8. } NoffHeader;

**6.1.6 addrSpace.h AddrSpace类**

1、用于跟踪正在执行的用户程序（地址空间）的数据结构。存储虚拟页到内存页的映射关系。

2、成员变量TranslationEntry \*pageTable表示进程页表。unsigned int numPages表示进程虚拟地址空间页数。

3、static void SwapHeader(NoffHeader \*noffH)对对象文件头中的字节进行小endian到大endian的转换，以防文件是在一个小endian机器上生成的，我们现在在一个大endian机器上运行。

4、AddrSpace(OpenFile \*executable)创建一个地址空间来运行一个用户程序。从一个“可执行”文件中加载程序，并设置所有设置，以便我们可以开始执行用户指令。假设目标代码文件为NOFF格式。首先，设置从程序内存到物理内存的转换。现在，这真的很简单（1:1），因为我们只是一个单一的编程，而且我们有一个未分段的页表。executable是包含要加载到内存中的目标代码的文件。首先读入NOFF文件头到NoffHeader数据结构。获取虚拟地址空间大小和初始外页表。建立虚拟页到内存页的映射关系存储在进程页表中。然后将可执行文件读入到内存中。

5、void AddrSpace::InitRegisters()初始化用户寄存器，PCReg寄存器初始为0，NextPCReg寄存器初始为4，StackReg寄存器设置到地址空间的末尾，在那里我们分配了堆栈；但是减去一点，以确保我们不会意外地引用结尾。

6、void AddrSpace::RestoreState()在上下文开关上，还原计算机页表状态，以便此地址空间可以运行，告诉机器在哪里可以找到分页表。把系统页表设置为当前进程页表。跟新系统页表页数。

1. **void** AddrSpace::RestoreState()
2. {
3. machine->pageTable = pageTable;
4. machine->pageTableSize = numPages;
5. }

7、void AddrSpace::SaveState()在上下文切换上，保存任何需要保存的计算机页表状态（特定于此地址空间）。

**6.1.7 thread.h Thread类**

1、运行用户程序的线程实际上有两组CPU寄存器，一组用于保存执行用户代码时的状态，一组用于保存执行内核代码时的状态。有两个堆栈，thread中的为内核栈，用户地址空间中的为用户栈。int userRegisters[NumTotalRegs]存放进程用户级CPU寄存器状态。AddrSpace \*space存放进程页表和页表数。可以将二者理解为进程PCB。

2、模拟的MIPS CPU中有一个系统页表及页数和用户寄存器。

3、void Thread::SaveUserState()在上下文交换上保存用户程序的CPU状态。

4、void Thread::RestoreUserState()在上下文交换上恢复用户程序的CPU状态。

**6.1.8 schedule.h Scheduler类**

void Scheduler::Run(Thread \*nextThread)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。

1. **if** (currentThread->space != NULL)
2. {                                   // if this thread is a user program,
3. currentThread->SaveUserState(); // save the user's CPU registers
4. currentThread->space->SaveState();
5. }

如果需要则进行内核线程销毁

1. **if** (threadToBeDestroyed != NULL)
2. {
3. **delete** threadToBeDestroyed;
4. threadToBeDestroyed = NULL;
5. }

恢复新进程的PCB内容保存起来，包括用户寄存器、进程页表。

1. **if** (currentThread->space != NULL)
2. {                                      // if there is an address space
3. currentThread->RestoreUserState(); // to restore, do it.
4. currentThread->space->RestoreState();
5. }

**6.1.9 syscall.h**

1、作为头文件被c文件引用，实现系统调用。Nachos系统调用接口。这些是Nachos内核操作，可以从用户程序调用，通过“syscall”指令捕获到内核。这个文件包含在用户程序和Nachos内核中。

2、宏定义设置系统调用号。

3、系统调用接口void Halt(); void Exit(int status); SpaceId Exec(char \*name); int Join(SpaceId id); void Create(char \*name); OpenFileId Open(char \*name); void Write(char \*buffer, int size, OpenFileId id); int Read(char \*buffer, int size, OpenFileId id); void Close(OpenFileId id); void Fork(void (\*func)()); void Yield();系统调用接口。这些是Nachos内核需要支持的操作，以便能够运行用户程序。其中的每一个都由用户程序通过简单地调用过程来调用；汇编语言存根将系统调用代码填充到寄存器中，并陷阱到内核中。然后，在适当的错误检查之后，在Nachos内核中从exception.cc中的系统调用入口点调用内核过程。

**6.1.10 start.s**

1、#include "syscall.h"引入头文件。

2、对void Halt()等系统调用接口函数用汇编语言实现，如HALT的实现如下，把系统调用号放到2号寄存器后，下一条指令位syscall系统调用，执行时会产生异常，在异常处理函数中完成系统调用。c文件中调用halt()函数，链接时会在start.s 中找到halt()的定义。生成机器代码在模拟的MIPS CPU上执行。

1. .globl Halt
2. .ent    Halt
3. Halt:
4. addiu $2,$0,SC\_Halt     //system call code in r2
5. syscall
6. j   $31
7. .end Halt
9. .globl Exit
10. .ent    Exit

3、最终连接形成的可执行文件，如执行到最后，则调用exit(0)。

**6.1.11 progtest.cc**

void StartProcess(char \*filename)首先根据文件名打开可执行文件，新建进程PCB（包括进程地址空间和存放用户寄存器内容的位置），完成到当前内核线程的映射，初始化用户寄存器和系统页表，模拟的MIPS CPU开始执行用户程序。

1. space->InitRegisters(); // set the initial register values
2. space->RestoreState();  // load page table register
3. machine->Run(); // jump to the user progam

**6.1.12 exception.cc ExceptionHandler()**

1、从 2 号寄存器中获取当前的系统调用号,根据 type 对系统调用分别处理。

2、Halt()停机系统调用，通过调用interrupt->Halt()实现。

**6.1.13 系统调用参数传递**

1、我们知道，Exec(FilenNme)作为 Nachos 的系统调用，Exec(FilenNme)执行时需要陷入到 Nachos 的内核中执行，因此需要将其参数 FileName 从用户地址空间传递（复制）到内核中，从而在内核中为 FileName 对应的应用程序创建相应的线程执行它。 一般参数传递有三种方式：

（a）通过寄存器；

（b）通过内存区域，将该内存区域的首地址存放在一个寄存器中；

（c）通过栈；

2、在基于 MIPS 架构中，对于一般的函数调用，一般利用$4-$7（4 到 7 号寄存器）传递函数的前四个参数给子程序，参数多于 4 个时，其余的利用堆栈进行传递；对于 Nachos 的系统调用，一般也是将要传递的参数依次保存到寄存器$4-$7 中，然后根据这些寄存器中的地址从内存中读出相应的参数。特别要注意的是字符串作为参数时的传递方式，这时寄存器中保存的是字符串在内存中的地址。 例如，如果第一个参数是字符串，则将该字符串在内存中的地址存入$4，如 果第二个参数是数值，则将该值存入$5，以此类推。对于 Nachos 的系统调用，一般也是将要传递的参数在内存中的地址依次保存到寄存器$4-$7 中，然后根据这些寄存器中的地址从内存中读出相应的参数。

3、系统调用 Exit(1)对应的汇编代码为：

li s4,1 #将立即数 0x1 存入寄存器$4；传递 Exit(1)中的参数

jal Exit #转到 start.s 中的 Exit 的调用入口

可以看出，对于参数为数值的，系统调用时系统将参数值按顺序依次传入$4-$7 中。

4、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。

**6.2 打印进程页表设计任务**

**6.2.1 AddrSpace类添加Print()方法**

1、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。

1. **void** AddrSpace::Print()
2. {
3. printf("page table dump: %d pages in total\n", numPages);
4. printf("=============================\n");
5. printf("\tVirtPage, \tPhysPage\n");
6. **for** (**int** i = 0; i < numPages; i++)
7. {
8. printf("\t %d, \t\t%d\n", pageTable[i].virtualPage, pageTable[i].physicalPage);
9. }
10. printf("============================================\n\n");
11. }
12. 如在../lab7-8/ progtest.cc 的 void StartProcess(char \*filename)中，当为一个应用 程序新建一个空间后，调用 Print()，输出页表信息。

**6.3 打印进程页表调试结果**

在../userprog 中运行 nachos –x ../test/halt.noff，从输出结果中可以看看程序halt.noff 的页面与帧（虚页与实页）的对应关系，以及 Nachos 为该程序分配的实页数（Nachos 为该程序分配了 11 个实页）。



**7 实验七 地址空间的扩展**

**7.1 地址空间扩展设计任务**

**7.1.1 增加BitMap类全局变量bitmap**

1、假设我们希望在一个 Nachos 应用程序中通过系统调用 Exec()装入并执行另一个 Nachos 应用程序，即当程序执行到语句 Exec()时，系统需要将noff可执行装入到内存，为其分配内存空间并执行它，遗憾的是目前 Nachos 无法实现上述功能。当前nachos为从主存第0页开始新进程分配地址空间，而无法判断该主存页是否已经分配给其他进程。新进程地址空间会覆盖旧进程地址空间，程序无法正确执行。

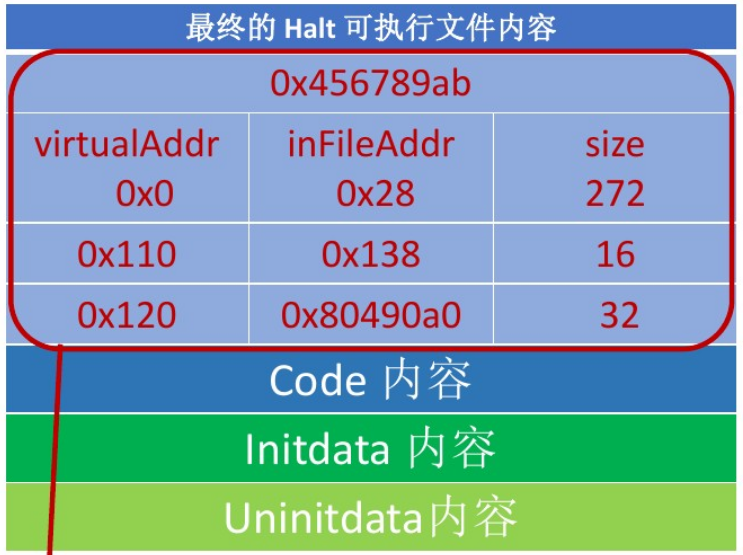
2、为记录主存页是否被使用，在system.cc中加入BitMap类全局变量bitmap，位图页数初始化为NumPhysPages位。

**7.1.2 修改AddrSpace::AddrSpace()**

1、更改进程地址空间分配方式，不再是从头开始依次分配，而是选择空闲的主存页分配给新进程。主存页管理使用全局变量bitmap。同时注释掉bzero，不需清空主存。

1. **for** (**int** i = 0; i < numPages; i++)
2. {
3. pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
4. pageTable[i].physicalPage = bitmap->Find();
5. ;
6. pageTable[i].valid = TRUE;
7. pageTable[i].use = FALSE;
8. pageTable[i].dirty = FALSE;
9. pageTable[i].readOnly = FALSE; // if the code segment was entirely on

2、修改AddrSpace构造函数中复制代码段部分，NOFF可执行文件虚拟地址空间如下图所示，首先按照虚拟地址将各段从可执行文件中读出，利用了类似外页表的方式管理虚拟地址空间。之后根据进程页表写入到主存对应的页中。完成虚实地址转换。



1. tempMainMemory = **new** **char**[MemorySize];
3. // then, copy in the code and data segments into memory
4. **if** (noffH.code.size > 0)
5. {
6. DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
7. noffH.code.virtualAddr, noffH.code.size);
8. executable->ReadAt(&(tempMainMemory[noffH.code.virtualAddr]),
9. noffH.code.size, noffH.code.inFileAddr);
10. }
11. **if** (noffH.initData.size > 0)
12. {
13. DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
14. noffH.initData.virtualAddr, noffH.initData.size);
15. executable->ReadAt(&(tempMainMemory[noffH.initData.virtualAddr]),
16. noffH.initData.size, noffH.initData.inFileAddr);
17. }
19. **for** (**int** i = 0; i < numPages; i++)
20. memcpy(&(machine->mainMemory[pageTable[i].physicalPage \* PageSize]), &(tempMainMemory[i \* PageSize]), PageSize);
21. **delete**[] tempMainMemory;

**7.1.3 修改AddrSpace::~AddrSpace()**

1、进程执行完毕，会收进程地址空间时，部件要会收页表空间，还要调用bitmap的Clear()函数更新位图，使得相应主存也可以被新进程使用。

1. **for** (**int** i = 0; i < numPages; i++)
2. bitmap->Clear(pageTable[i].physicalPage);

**8 实验八 系统调用**

**8.1 Exec()设计任务**

**8.1.1 增加Pcb类**

1、应用程序运行时，需要为其创建一个用户进程，为程序分配内存空间，将用户程序（代码段与数据段，数据段包括初始化的全局变量与未初始化的全局变量，以及静态变量）装入所分配的内存空间，创建相应的页表，建立虚页与实页（帧）的映射关系；

2、系统应该首先为应用程序分配一个 PCB，存放应用程序进程的相应信息；进程号 pid 可以是 PCB 数组的索引号；根据应用程序的文件头计算所需的实页数（帧数），根据内存的使用情况为其分配足够的空闲帧（不一定是连续的帧），将应用程序的代码与数据读入内存所分配的帧中，创建页表，建立虚页与实页的映射关系；还要为应用程序分配栈与堆； 还应该在 PCB 中建立打开文件的列表，列表的索引即为文件描述符； 最后将 pid、页表位置、栈位置及堆位置等信息记录在 PCB 中，在 PCB 中建立三个标准设备的映射关系，并记录进程与线程的映射关系，以及进程的上下文等；

3、目前 Nachos 实现的比较简单，没有显式地定义 PCB。而是将进程的信息分散到相应的类对象中；例如利用所分配内存空间的对象指针标识一个进程，该对象中含有进程的页表、栈指针、与核心线程的映射等信息；进程的上下文保存在核心线程中，当一个线程被调度执行后，依据线程所保存的进程上下文中执行所对应的用户进程；同时，进程被创建后不应立即执行，应该将程序入口等记录到 PCB 中，一旦相应的核心线程引起调度，就从 PCB 中获取所需的信息执行该进程；

4、新建Pcb类，有原Thread类成员变量AddrSpace \*space存储进程地址空间。int userRegisters[NumTotalRegs]存储进程上下文。Pcb类作为Thread类成员变量。

**8.1.2 建立spaceId到进程地址空间的双向映射关系**

1、AddrSpace类增加成员变量spaceId，增加成员int getSpaceId()函数获得spcaeId。

2、建立spaceId到进程地址空间的双向映射关系。AddrSpace到spcaeId的映射通过getSpaceId()实现。spcaeId到AddrSpace的映射通过增加以下全局变量实现。

bool ThreadMap[MAX\_USERPROCESSES]记录某一spcaeId是否被使用。

AddrSpace \*addrspaces[MAX\_USERPROCESSES]完成映射关系。

1. AddrSpace:AddrSpace()进程地址空间创建过程中增加spaceId的控制创建。
2. **bool** hasAvailabePid = **false**;
3. **for** (**int** i = 100; i < MAX\_USERPROCESSES; i++)
4. {
5. **if** (!ThreadMap[i])
6. {
7. ThreadMap[i] = **true**;
8. spaceId = i; //may be should reserved 0-99 for kernel Process,
9. //even though there is no any process at present
10. addrspaces[i] = **this**;
11. hasAvailabePid = **true**; //ther is available Pid for new process
12. **break**;
13. }
14. }                    //for
15. **if** (!hasAvailabePid) //no available Pid for new process
16. {
17. printf("Too many processes in Nachos !\n");
18. **return**;
19. }
20. ASSERT(hasAvailabePid);
21. AddrSpace:~AddrSpace()进程地址空间回收过程中增加spaceId及其映射的回收。
22. ThreadMap[spaceId] = 0;
23. addrspaces[spaceId] = NULL;

**8.1.3 重载void StartProcess(int spaceId)**

1、原函数void StartProcess(char \*filename)根据可执行文件名创建进程地址空间，并将新进程地址空间映射到当前核心线程。初始化用户寄存器，初始化系统页表，进入用户模式开始执行用户程序。

2、重载的void StartProcess(int spaceId)根据spaceId，使用双向映射获得已创建好的进程地址空间，并将新进程地址空间映射到当前核心线程。初始化用户寄存器，初始化系统页表，进入用户模式开始执行用户程序。即将进程地址空间的创建和映射分开。

1. **void** StartProcess(**int** spaceId)
2. {
3. AddrSpace \*space;
5. space = addrspaces[spaceId];
6. currentThread->pcb->space = space;
7. space->Print();
9. space->InitRegisters();
10. space->RestoreState();  // load page table register
12. machine->Run(); // jump to the user progam
13. ASSERT(FALSE);  // machine->Run never returns;
14. }

**8.1.4 exception.cc增加AdvancePC()函数**

1、从中可以看出，当一条指令正常执行结束后，需要将 PC 推进，指向下一条指令，但 case OP\_SYSCALL:中，RaiseException(SyscallException, 0);后不是一条 break 语句，而是一条 return 语句，原因是通常情况下，当处理完一个异常后需要重启这条指令。

2、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。

1. **void** AdvancePC()
2. {
3. machine->WriteRegister(PCReg, machine->ReadRegister(PCReg) + 4);
4. machine->WriteRegister(NextPCReg, machine->ReadRegister(NextPCReg) + 4);
5. }

**8.1.5 ExceptionHandler()增加处理SC\_Exec的分支**

1、从第 4 号寄存器中获取 Exec()的参数 filename 在内存中的地址，利用 Machine::ReadMem()从该地址读取应用程序文件名 filename，打开该应用程序可执行文件，为其分配内存空间、创建页表、分配spaceId ，至此为应用程序创建了一个进程。创建一个核心线程，并将该进程与新建的核心线程关联。我们可以重载函数 StartProcess(int spaceId)，作为新建线程执行的代码，并将进程的spaceId 传递给系统，供其它系统调用（如 Join()）使用。

2、初始化寄存器，开始程序执行。将所有设计寄存器请 0。由于读入程序时，从代码段的入口地址（虚地址）开始读入到内存的 0 号地址中（即第 0 帧的 0 号偏移量处），因此程序从逻辑地址 0 开始执行（物理地址也为 0），将 PC 设置为程序的逻辑地址入口 0。

3、返回进程的 spaceId，PC 增量。

4、根据filename判断是否是shell命令，如ls，则调用fileSystem->List()打印根目录文件。

1. **char** filename[50];
2. **int** addr = machine->ReadRegister(4);
3. **int** i = 0;
4. **do**
5. {
6. //read filename from mainMemory
7. machine->ReadMem(addr + i, 1, (**int** \*)&filename[i]);
8. } **while** (filename[i++] != '\0');
9. // printf("Exec(%s):\n",filename);
10. DEBUG('x', "thread:%s\tExec(%s):\n", currentThread->getName(), filename);
12. **if** (filename[0] == 'l' && filename[1] == 's') //ls
13. {
14. DEBUG('x', "thread:%s\tFile(s) on Nachos DISK:\n", currentThread->getName());
15. fileSystem->List();
16. machine->WriteRegister(2, 127); //
17. AdvancePC();
18. **return**;
19. }
21. OpenFile \*executable = fileSystem->OpenTest(filename);
22. AddrSpace \*space;
24. **if** (executable == NULL)
25. {
26. printf("Unable to open file %s\n", filename);
27. // return;
29. ASSERT(**false**);
30. }
31. space = **new** AddrSpace(executable);
33. Thread \***thread** = **new** Thread(filename);
34. **thread**->Fork(StartProcess, space->getSpaceId());
36. machine->WriteRegister(2, space->getSpaceId());
37. AdvancePC();
39. // currentThread->Yield();
41. **delete** executable;

**8.2 Join() Exit()设计任务**

**8.2.1 修改Pcb类，增加退出码相关成员变量及相关函数**

1、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。

2、修改PCB类，增加int exitStatus成员变量存放进程退出码。

int waitingProcessSpaceId表示此进程等待进程的spaceId。

int waitProcessExitCode表示此进程等待进程的退出吗。

增加int getExitStatus()和void setExitStatus(int ExitStatus)提供对exitStatus的访问。

1. **void** Pcb::setExitStatus(**int** ExitStatus)
2. {
3. **this**->exitStatus = ExitStatus;
4. }
6. **int** Pcb::getExitStatus()
7. {
8. **return** exitStatus;
9. }

**8.2.2 修改List类，增加列表操作函数**

1、List类增加length成员变量，表示List长度。在插入和删除时进行维护。增加int ListLength()获得List长度。

2、增加根据value值删除的RemoveByItem(void \*item)函数。

1. **void** List::RemoveByItem(**void** \*item)
2. {
3. ListElement \*prev, \*ptr;
4. **void** \*removed;
5. **if** (item == first->item)
6. {
7. removed = Remove();
8. ASSERT(item == removed);
9. }
10. **else**
11. {
12. prev = first;
13. **for** (ptr = first->next; ptr != NULL; prev = ptr, ptr = ptr->next)
14. {
15. **if** (item == ptr->item)
16. {
17. prev->next = ptr->next;
18. **if** (prev->next == NULL)
19. {
20. last = prev;
21. }
22. **delete** ptr;
23. numInList--;
24. **break**;
25. }
26. }
27. ASSERT(ptr != NULL); // should always find item!
28. }
29. //ASSERT(!IsInList(item));
30. }

3、增加返回第i个项的getItem(int i)函数。提供了List遍历功能。

1. **void** \*List::getItem(**int** i)
2. {
3. ListElement \*ptr = first;
4. **for** (**int** k = 0; k < i; k++)
5. ptr = ptr->next;
6. **return** ptr->item;
7. }

**8.2.3 修改Scheduler类，增加waitingList terminatedList**

1、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）；

2、Scheduler增加waitingList terminatedList及相关函数。deleteTerminatedThread()从终止队列中删除并释放线程。deleteWaitingThread()从等待队列中删除。

1. **void** Scheduler::deleteTerminatedThread(Thread \***thread**)
2. {
3. terminatedList->RemoveByItem((**void** \*)**thread**);
4. **delete** **thread**;
5. }
7. **void** Scheduler::deleteWaitingThread(Thread \***thread**)
8. {
9. waitingList->RemoveByItem((**void** \*)**thread**);
10. }

**8.2.4 修改Thread类，增加TERMINATED状态及相关函数**

1、线程原有4种状态JUST\_CREATED,RUNNING,READY,BLOCKED,新建TERMINATED状态表示进程终止状态。

2、新增void Thread::Terminated()，将当前线程的状态设为TERMINATED状态。加入terminatedList队列，切换线程。

1. **void** Thread::Terminated()
2. {
3. List \*terminatedList = scheduler->getTerminatedList();
4. Thread \*nextThread;
6. ASSERT(**this** == currentThread); // a thread sleep by itsef
7. ASSERT(interrupt->getLevel() == IntOff);
8. status = TERMINATED;
9. terminatedList->Append((**void** \*)**this**);
11. nextThread = scheduler->FindNextToRun();
12. **while** (nextThread == NULL)
13. {
14. interrupt->Idle();
15. nextThread = scheduler->FindNextToRun();
16. }
17. scheduler->Run(nextThread); // returns when we've been signalled
18. }

**8.2.5 修改Thread类，增加join()并修改finish()**

1、Join系统调用分析：当 Joiner 执行 Thread::Join(spaceId)时，若 Joinee 在 terminated 队列，则从 terminated 队列移除 Joinee 并将其销毁，然后返回 Joinee 的退出码；如果 Joinee 不在 terminated 队列，说明其尚未终止，则 Joiner 进入睡眠队列 waitingList，当 Joinee 退出调用 Finish()时通过检查 waitingList 以确定是否需要唤醒 Joiner。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。

2、增加Thread::join()函数。首先设置当前进程的waitingProcessSpaceId为参数spaceId。Thread::Join()将当前线程睡眠，由于当前线程负责执行当前用户进程，因此效果是当前进程进入睡眠；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,并销毁等待的线程。更新当前进程的waitProcessExitCode。

1. **int** Thread::Join(**int** SpaceId)
2. {
3. IntStatus oldLevel = interrupt->SetLevel(IntOff);
4. pcb->waitingProcessSpaceId = SpaceId;
5. //if joinee is still in not in terminated list?
6. Thread \***thread**;
7. List \*terminatedList = scheduler->getTerminatedList();
8. List \*waitingList = scheduler->getWaitingList();
10. //是否在terminatedList中
11. **bool** interminatedList = FALSE;
12. **int** listLength = terminatedList->ListLength();
13. **for** (**int** i = 0; i < listLength; i++)
14. {
15. **thread** = (Thread \*)terminatedList->getItem(i);
16. **if** (**thread** == NULL)
17. interminatedList = FALSE; // joinee not finished
18. //joinee is still in Ready queue, not finished
19. **if** (**thread**->pcb->space->getSpaceId() == SpaceId)
20. {
21. interminatedList = TRUE; // joinee alreday finished
22. **break**;
23. }
24. interminatedList = FALSE; // joinee not finished
25. }
26. **if** (!interminatedList)
27. // still in Nanchos system, maybe at READY or BLOCKED
28. {
29. waitingList->Append((**void** \*)**this**); //blocked Joiner
30. DEBUG('x', "thread%s\tjoin sleep\n", currentThread->getName());
31. currentThread->Sleep();
32. DEBUG('x', "thread%s\tjoin wake up\n", currentThread->getName());
34. listLength = terminatedList->ListLength();
35. **for** (**int** i = 0; i < listLength; i++)
36. {
37. **thread** = (Thread \*)terminatedList->getItem(i);
38. **if** (**thread**->pcb->space->getSpaceId() == SpaceId)
39. **break**;
40. }
41. }
42. //joinee alreday finished, in terminated List, empty it, and return
43. pcb->waitProcessExitCode = **thread**->pcb->getExitStatus();
44. //delete terminated thread "Joinee"
45. scheduler->deleteTerminatedThread(**thread**);
46. interrupt->SetLevel(oldLevel);
47. }

3、修改Thread::finish()函数，Joinee 退出调用 Finish()时通过检查waitingList ，通过检查自己spaceId与waitingList中进程的spaceId是否相等，以确定是否有 Joiner等待自己，如果有将其唤醒，从waitingList中删除，加入就绪队列。然后调用Terminated()函数进入TERMINATED状态。

1. **void** Thread::Finish()
2. {
3. (**void**)interrupt->SetLevel(IntOff);
4. ASSERT(**this** == currentThread);
6. #ifdef USER\_PROGRAM
7. //joinee finised, wakeup the join user program
8. List \*waitingList = scheduler->getWaitingList();
9. Thread \*waitingThread;
11. // if joiner is sleeping and in waitinglist, joinee wait up joiner
12. // when joinee finish
13. **int** listLength = waitingList->ListLength();
14. **for** (**int** i = 0; i < listLength; i++)
15. {
16. waitingThread = (Thread \*)waitingList->getItem(i);
17. **if** (currentThread->pcb->space->getSpaceId() == waitingThread->pcb->waitingProcessSpaceId)
18. {
19. scheduler->ReadyToRun((Thread \*)waitingThread);
20. scheduler->deleteWaitingThread(waitingThread);
21. **break**;
22. }
23. }
24. Terminated();
25. #else
26. DEBUG('t', "Finishing thread \"%s\"\n", getName());
28. threadToBeDestroyed = currentThread;
29. Sleep(); // invokes SWITCH
30. // not reached
31. #endif
32. }

**8.2.6 修改system.cc，打印waitingList terminatedList**

1、主要调用List::Mapcar()函数对队列中的每一项调用printThreadName(),打印waitingList terminatedList中的进程名。

2、在调用Cleanup(),停机时进行打印,便于调试。

1. **static** **void** printThreadName(\_int ptr)
2. {
3. Thread \***thread** = (Thread \*)ptr;
4. printf("%s\t", **thread**->getName());
5. }
6. **void** Cleanup()
7. {
8. #ifdef USER\_PROGRAM
9. DEBUG('x', "TerminatedList length:%d\n", scheduler->getTerminatedList()->ListLength());
10. scheduler->getTerminatedList()->Mapcar((VoidFunctionPtr)printThreadName);
11. DEBUG('x', "\nWaitingList length:%d\n", scheduler->getWaitingList()->ListLength());
12. scheduler->getWaitingList()->Mapcar((VoidFunctionPtr)printThreadName);
13. #endif
14. }

**8.2.7 ExceptionHandler()增加处理SC\_Exit的分支**

1、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)退出时需要使用PCB类的setExitStatus()函数将 id 的退出码 ststus 保存起来，以备 Join()使用。

2、首先从 4 号寄存器读出退出码，然后释放PCB、 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 spaceId（参见 AddrSpace::~AddrSpace()），调用 currentThread->Finish()结束该进程对应的线程。

3、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。

1. **int** ExitStatus = machine->ReadRegister(4);
2. DEBUG('x', "thread:%s\tExit(%d):\n", currentThread->getName(), ExitStatus);
3. currentThread->pcb->setExitStatus(ExitStatus);
5. List \*terminatedList = scheduler->getTerminatedList();
6. **if** (currentThread->pcb->getExitStatus() == 99)
7. {
8. DEBUG('x', "thread:%s\tparent delete terminatedList:\n", currentThread->getName());
9. Thread \***thread**;
10. **while** ((**thread** = (Thread \*)(terminatedList->Remove())) != NULL)
11. **delete** **thread**;
12. }
14. currentThread->Finish();
15. AdvancePC();

**8.2.8 ExceptionHandler()增加处理SC\_Join的分支**

1、首先从 4 号寄存器读出参数spaceId，调用Thread::Join(spaceId)函数，执行结束后将currentThread->pcb->waitProcessExitCode写回到2号寄存器作为返回值，返回spaceId进程的退出码。

1. **int** spaceId = machine->ReadRegister(4);
2. DEBUG('x', "thread:%s\tJoin %d:\n", currentThread->getName(), spaceId);
3. currentThread->Join(spaceId);
4. //返回 Joinee 的退出码 waitProcessExitCode
5. machine->WriteRegister(2, currentThread->pcb->waitProcessExitCode);
6. DEBUG('x', "waitProcessExitCode %d:\n", currentThread->pcb->waitProcessExitCode);
7. AdvancePC();

**8.3 Yield()设计任务**

ExceptionHandler()增加处理SC\_Yield的分支，Yield()系统调用实现类进程切换，由于用户进程和内核线程是一对一的映射关系，所以使用调用currentThread->Yield()实现。

**8.4 Create() Open() Write() Read() Close()设计任务**

**8.4.1 修改Pcb类，增加进程打开文件表及相关函数**

1、修改PCB类，增加bool fileIdUse[MaxFileId]记录文件描述符是否被使用。

OpenFile \*files[MaxFileId]完成进程文件描述符到打开文件OpenFile\*的映射。

1. 为OpenFile\*申请进程文件描述符的函数addFile(OpenFile \*openfile)。参数OpenFile \*，首先确定是否已经存在该文件描述符，如果存在则报错。如果不存在则找到未被使用的文件描述符返回。
2. **int** Pcb::addFile(OpenFile \*openfile)
3. {
4. **for** (**int** i = 2; i < MaxFileId; i++)
5. **if** (fileIdUse[i] && files[i] == openfile)
6. {
7. printf("duplicate file\n");
8. ASSERT(**false**);
9. **break**;
10. }
11. **for** (**int** i = 2; i < MaxFileId; i++)
12. **if** (!fileIdUse[i])
13. {
14. DEBUG('x', "addFile success\n");
15. fileIdUse[i] = **true**;
16. files[i] = openfile;
17. **return** i;
18. }
19. printf("file full\n");
20. ASSERT(**false**);
21. **return** -1;
22. }
23. 实现从进程文件描述符到OpenFile \*的转换函数getFile(int fileId)。
24. OpenFile \*Pcb::getFile(**int** fileId)
25. {
26. **if** (fileIdUse[fileId])
27. **return** files[fileId];
28. **else**
29. **return** NULL;
30. }

4、释放进程文件描述符资源的函数。同时释放OpenFile \*，关闭文件。

1. **void** Pcb::releaseFile(**int** fileId)
2. {
3. **if** (fileIdUse[fileId])
4. {
5. fileIdUse[fileId] = **false**;
6. **delete** files[fileId];
7. files[fileId] = NULL;
8. }
9. **else**
10. {
11. printf("thr file to close not find\n");
12. ASSERT(**false**);
13. }
14. }

**8.4.2 ExceptionHandler()增加处理SC\_Create的分支**

通过ReadRegister()从4号寄存器获得文件名字符串地址，调用ReadMem()获取文件名字符串，使用nachos文件系统CreateTest()创建文件。

1. **int** base = machine->ReadRegister(4);
2. **int** value;
3. **int** count = 0;
4. **char** \*FileName = **new** **char**[128];
5. **do**
6. {
7. machine->ReadMem(base + count, 1, &value);
8. FileName[count] = \*(**char** \*)&value;
9. count++;
10. } **while** (\*(**char** \*)&value != '\0' && count < 128);
11. **if** (!fileSystem->CreateTest(FileName, 0))
12. {
13. printf("thread:%s\tcreate file:%s failed!\n", currentThread->getName(), FileName);
14. ASSERT(**false**);
15. }
16. **else**
17. DEBUG('x', "thread:%s\tcreate file:%s success\n", currentThread->getName(), FileName);
18. AdvancePC();

**8.4.3 ExceptionHandler()增加处理SC\_Open的分支**

1、通过ReadRegister()从4号寄存器获得文件名字符串地址，调用ReadMem()获取文件名字符串。使用nachos文件操作函数OpenTest()打开文件并返回OpenFile\*，调用PCB::addFile(openfile)申请进程文件描述符。通过WriteRegister()写入2号寄存器将进程文件描述符返回用户地址空间。

2、如果文件打开失败则会导致停机。

1. **int** base = machine->ReadRegister(4);
2. **int** value;
3. **int** count = 0;
4. **char** \*FileName = **new** **char**[128];
5. **do**
6. {
7. machine->ReadMem(base + count, 1, &value);
8. FileName[count] = \*(**char** \*)&value;
9. count++;
10. } **while** (\*(**char** \*)&value != '\0' && count < 128);
12. OpenFile \*openfile;
13. openfile = fileSystem->OpenTest(FileName);
14. **int** fileId;
15. **if** (openfile == NULL)
16. {
17. //file not existes, not found
18. printf("thread:%s\topen file:%s failed!\n", currentThread->getName(), FileName);
19. fileId = -1;
20. ASSERT(**false**);
21. }
22. **else**
23. {
24. fileId = currentThread->pcb->addFile(openfile);
25. DEBUG('x', "thread:%s\topen file:%s success\tfileId:%d\n", currentThread->getName(), FileName, fileId);
26. }
27. machine->WriteRegister(2, fileId);
28. AdvancePC();

**8.4.4 ExceptionHandler()增加处理SC\_Write的分支**

1、通过ReadRegister()从4号寄存器获得字符串数组地址。通过ReadRegister()从5号寄存器获得写入字符数组的长度。调用ReadMem()从模拟NACHOS主存中获取字符串数组。

2、通过ReadRegister()从6号寄存器获得进程文件描述符。如果进程文件描述符为1，则通过标准输出stdout输出结果字符数组。

3、如果进程文件描述符不是1，则调用Pcb::getFile()从进程打开文件表中获得OpenFile\*。调用openfile->Write(buffer, size)写入文件，openfile->WriteBack()更新文件头。

1. **int** base = machine->ReadRegister(4);
2. **int** size = machine->ReadRegister(5);
3. //bytes written to file
4. **int** fileId = machine->ReadRegister(6); //fd
5. **int** value;
6. **int** count = 0;
8. **if** (fileId == 0)
9. {
10. printf("thread:%s\tstdin cannot write!\n", currentThread->getName());
11. ASSERT(**false**);
12. }
14. **char** \*buffer = **new** **char**[128];
15. **do**
16. {
17. machine->ReadMem(base + count, 1, &value);
18. buffer[count] = \*(**char** \*)&value;
19. count++;
20. } **while** (count < size);
21. buffer[size] = '\0';
23. **if** (fileId == 1)
24. printf("thread:%s\toutput to stdout:%s\n", currentThread->getName(), buffer);
25. **else**
26. {
27. OpenFile \*openfile;
28. **if** ((openfile = currentThread->pcb->getFile(fileId)) == NULL)
29. {
30. printf("thread:%s\tfileId:%d not open!\n", currentThread->getName(), fileId);
31. ASSERT(**false**);
32. }
34. **int** res = openfile->Write(buffer, size);
35. openfile->WriteBack();
36. **if** (res != size)
37. {
38. printf("thread:%s\tfileId:%d write failed!\n", currentThread->getName(), fileId);
39. ASSERT(**false**);
40. }
41. **else**
42. DEBUG('x', "thread:%s\tfileId:%d write success\tlength:%d!\n", currentThread->getName(), fileId, size);
43. }
44. AdvancePC();

**8.4.5 ExceptionHandler()增加处理SC\_Read的分支**

1、通过ReadRegister()从4号寄存器获得字符串数组地址。通过ReadRegister()从5号寄存器获得写入字符数组的长度。

2、通过ReadRegister()从6号寄存器获得进程文件描述符。如果进程文件描述符为0，则通过标准输出stdin输入字符数组。调用machine->WriteMem()写回NACHOS模拟主存。

3、如果进程文件描述符不是1，则调用Pcb::getFile()从进程打开文件表中获得OpenFile\*。调用openfile->Read(buffer, size)读取文件。调用machine->WriteMem()写回NACHOS模拟主存。

1. **int** base = machine->ReadRegister(4);
2. **int** size = machine->ReadRegister(5);
3. //bytes written to file
4. **int** fileId = machine->ReadRegister(6); //fd
5. **int** value;
6. **int** count = 0;
8. **if** (fileId == 1)
9. {
10. printf("thread:%s\tstdout cannot read!\n", currentThread->getName());
11. ASSERT(**false**);
12. }
14. **char** \*buffer = **new** **char**[128];
16. **if** (fileId == 0)
17. {
18. **for** (**int** i = 0; i < size; i++)
19. {
20. buffer[i] = getchar();
21. machine->WriteMem(base + i, 1, buffer[i]);
22. }
23. buffer[size] = '\0';
24. DEBUG('x', "thread:%s\tinput from stdin:%s\n", currentThread->getName(), buffer);
25. }
26. **else**
27. {
28. OpenFile \*openfile;
29. **if** ((openfile = currentThread->pcb->getFile(fileId)) == NULL)
30. {
31. printf("thread:%s\tfileId:%d not open!\n", currentThread->getName(), fileId);
32. ASSERT(**false**);
33. }
35. **int** res = openfile->Read(buffer, size);
36. buffer[size] = '\0';
37. **if** (res != size)
38. {
39. printf("thread:%s\tfileId:%d read failed!\tres:%d\n", currentThread->getName(), fileId, res);
40. ASSERT(**false**);
41. }
42. **else**
43. DEBUG('x', "thread:%s\tfileId:%d read success\tlength:%d!\tcontent:%s\n", currentThread->getName(), fileId, size, buffer);
45. **for** (**int** i = 0; i < size; i++)
46. machine->WriteMem(base + i, 1, buffer[i]);
47. }
48. AdvancePC();

**8.4.6 ExceptionHandler()增加处理SC\_Close的分支**

通过ReadRegister()从6号寄存器获得进程文件描述符。调用Pcb::releaseFile()释放进程描述符，同时释放OpenFile\*，如果该文件描述符没有使用，则会报错停机。

1. **int** fileId = machine->ReadRegister(4);
2. currentThread->pcb->releaseFile(fileId);
3. DEBUG('x', "thread:%s\tfileId:%d close success\n", currentThread->getName(), fileId);
4. AdvancePC();

**8.5 Nachos用户程序与系统调用测试程序**

**8.5.1 测试程序sh.noff**

1、sh.noff作为shell脚本，write()系统调用首先向标准输出stdin输出Nachos$:。

2、read()系统调用从标准输入stdin中获得要执行的文件名filename或shell命令。回车符替换为\0。

3、exec()系统调用执行filename或shell命令。

4、如果是执行可执行文件noff，则通过系统调用join()等待该可执行文件执行完成，会收该进程。

1. #include "syscall.h"
2. **int** main()
3. {
4. SpaceId newProc;
5. OpenFileId input = ConsoleInput;
6. OpenFileId output = ConsoleOutput;
7. **char** prompt[7], ch, buffer[60];
8. **char** Hbuffer[80];
9. **char** Cbuffer[80];
10. **int** i, j, k, h, m;
11. **char** c;
12. prompt[0] = 'N';
13. prompt[1] = 'a';
14. prompt[2] = 'c';
15. prompt[3] = 'h';
16. prompt[4] = 'o';
17. prompt[5] = 's';
18. prompt[6] = '$';
19. prompt[7] = ':';
21. **while** (1)
22. {
23. Write(prompt, 8, output);
24. i = 0;
25. k = 0;
26. h = 0;
27. m = 0;
28. **do**
29. {
31. Read(&c, 1, input);
33. buffer[i] = c;
34. Hbuffer[h++] = buffer[i];
36. k++;
37. } **while** (buffer[i++] != '\n');
39. Hbuffer[--h] = '\0';
41. buffer[--i] = '.';
42. buffer[i++] = '.';
43. buffer[i++] = 'n';
44. buffer[i++] = 'o';
45. buffer[i++] = 'f';
46. buffer[i++] = 'f';
47. buffer[i] = '\0';
49. **if** (k == 1)
50. **continue**;
51. **if** (h > 0)
52. {
53. newProc = Exec(Hbuffer);
55. **if** (newProc == -1)
56. //Write("Invalid Command, Enter again.", 30, output);
57. Write("Invalid Command, Enter again, or try \"help\"\n", 44, output);
58. **else** **if** (newProc != 127)
59. {
60. Join(newProc);
62. Write("Command \"", 9, output);
63. Write(Hbuffer, k, output);
64. Write("\" Execute Completely.\n", 22, output);
65. }
66. }
67. }
68. }

**8.5.2 测试程序read.noff**

1、本程序为文件操作测试程序。涉及两个文件/file/a和/file/b。

2、Create()系统调用创建/file/a和/file/b，Open()系统调用分别打开两个文件。

3、read()系统调用从标准输入stdin中读入一串字符，以回车结尾，存入字符数组ary中。Write()系统调用将字符数组ary写入/file/a，Close()系统调用关闭文件/file/a。重新打开后，read()系统调用读取/file/a文件获得刚才输入的内容，Write()系统调用把内容输出到标准输出stdout。

4、定义两个字符串"aaaaaaaaaa" "bbbbbbbbbb"，Write()系统调用将两个字符串依次写入/file/b。Close()系统调用关闭文件/file/b。重新打开后，read()系统调用读取/file/b文件读出刚才写入的"aaaaaaaaaabbbbbbbbbb"，Write()系统调用把内容输出到标准输出stdout。

5、关闭文件/file/a和/file/b。退出码为5。

1. #include "syscall.h"
3. **int** main()
4. {
5. OpenFileId file1;
6. OpenFileId file2;
8. **int** i = 0;
10. **char** ary[30];
12. **char** \*context1 = "aaaaaaaaaa";
13. **char** \*context2 = "bbbbbbbbbb";
15. **char** \*filename1 = "/file/a";
16. **char** \*filename2 = "/file/b";
18. Create(filename1);
19. Create(filename2);
21. file1 = Open(filename1);
22. file2 = Open(filename2);
24. **do**
25. {
26. Read(&ary[i], 1, ConsoleInput);
27. } **while** (ary[i++] != '\n');
29. ary[--i] = '\0';
31. Write(ary, i, file1);
33. Close(file1);
34. file1 = Open(filename1);
36. Read(ary, i, file1);
37. Write(ary, i, ConsoleOutput);
39. Write(context1, 10, file2);
40. Write(context2, 10, file2);
42. Close(file2);
43. file2 = Open(filename2);
45. Read(ary, 20, file2);
46. Write(ary, 20, ConsoleOutput);
48. Close(file1);
49. Close(file2);
50. Exit(5);
51. }

**8.5.3 测试程序exec.noff yiel.noff**

1、exec.noff系统调用Exec()执行/test/yiel.noff。随后Yield()系统调用使当前线程转为ready，切换线程。

2、yiel.noff开始执行。Yield()系统调用使当前线程转为ready，切换线程。

3、exec.noff执行join()系统调用，线程sleep。

4、yiel.noff执行Yield()系统调用，但exec.noff线程处于sleep。于是yiel.noff线程继续执行。

5、yiel.noff执行Exit(),进程退出码为6。

6、exec.noff被唤醒，销毁yiel.noff线程。

7、exec.noff执行Exit()，进程退出码为99，销毁在terminatedList中的全部线程。

1. #include "syscall.h"
3. **int** main()
4. {
5. SpaceId pid = Exec("/test/yiel.noff");
6. Yield();
7. Join(pid);
8. Exit(99);
9. }
10. #include "syscall.h"
12. **int** main()
13. {
14. Yield();
15. Yield();
16. Exit(6);
17. }

**8.5.4 测试程序halt.noff**

主要执行Halt()系统调用使nachos停机。

1. #include "syscall.h"
3. **int** main()
4. {
5. **int** i, j, k;
6. k = 3;
7. i = 2;
8. j = j - 1;
9. k = i - j + k;
10. Halt();
11. /\* not reached \*/
12. }

**8.6 Nachos用户程序与系统调用调试结果**

**8.6.1 初始状态**

1、./nachos -f 初始化DISK。

2、./nachos -mkdir /file 创建文件操作子目录。

3、./nachos -mkdir /test 创建进程切换操作子目录。

4、./nachos -cp ../test/sh.noff sh.noff

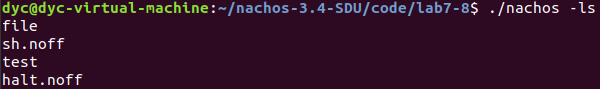
5、./nachos -cp ../test/halt.noff halt.noff

6、./nachos -cp ../test/read.noff /file/read.noff

7、./nachos -rm /file/a

8、./nachos -rm /file/b

9、./nachos -ls 显示根目录下的文件。

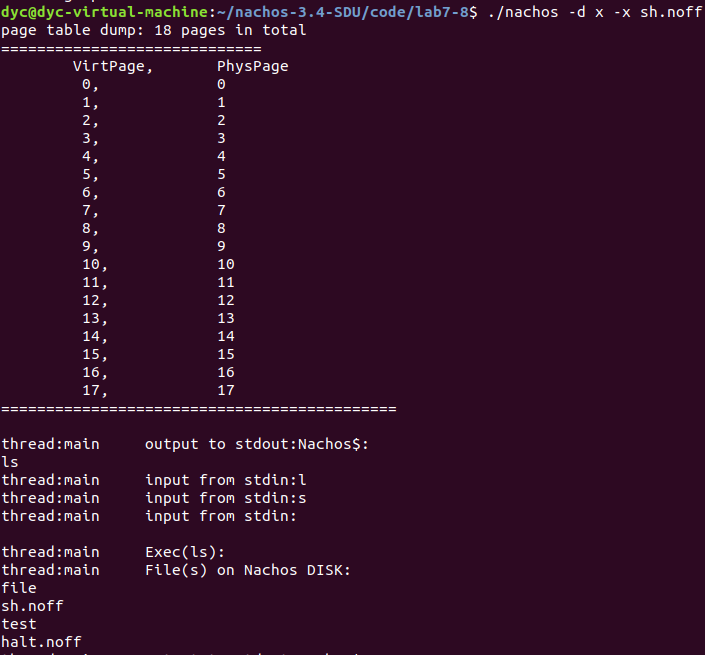


**8.6.2 执行sh.noff**

1、./nachos -d x -x sh.noff

2、ls

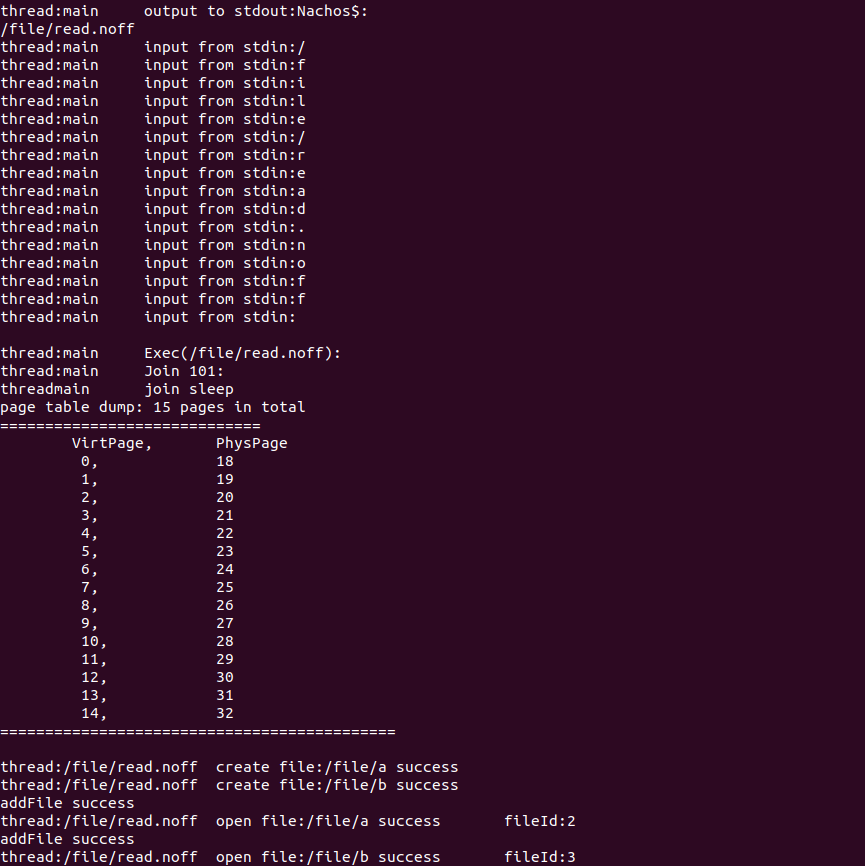
3、ls命令显示根目录下的文件。可以看到stdin和stdout的输入输出内容。非调试模式仅输出stdout的输出内容。

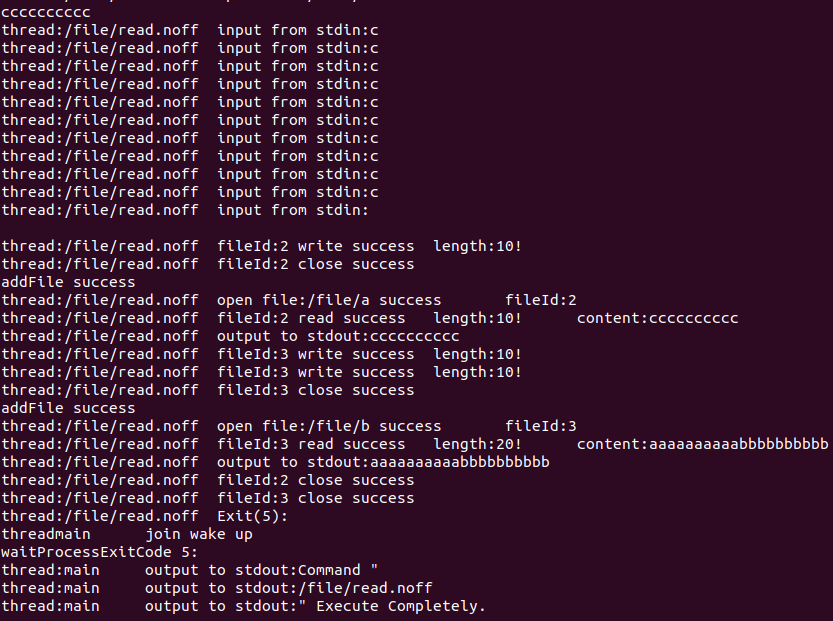


**8.6.3 执行/file/read.noff**

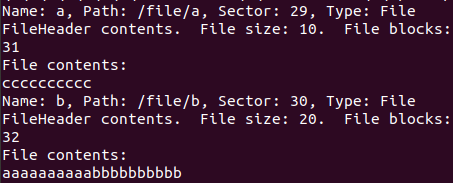
1、/file/read.noff

2、cccccccccc 这是/file/read调用标准输入，输入将输出到文件/file/a。



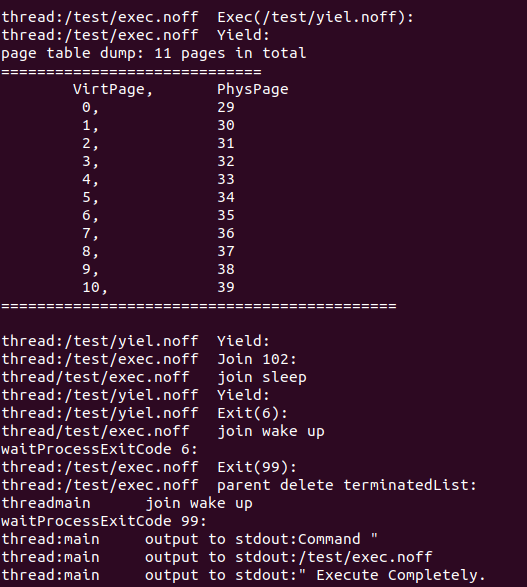
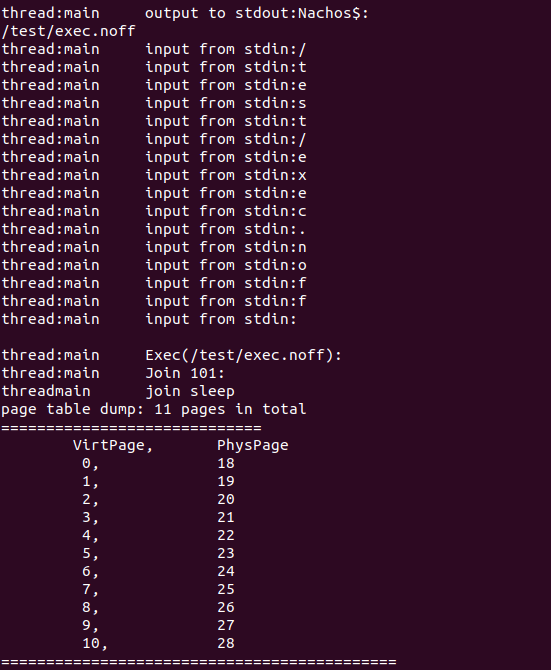


1. ./nachos -cat /file。可以看到/file/a内容为cccccccccc，/file/b内容为aaaaaaaaaabbbbbbbbbb。



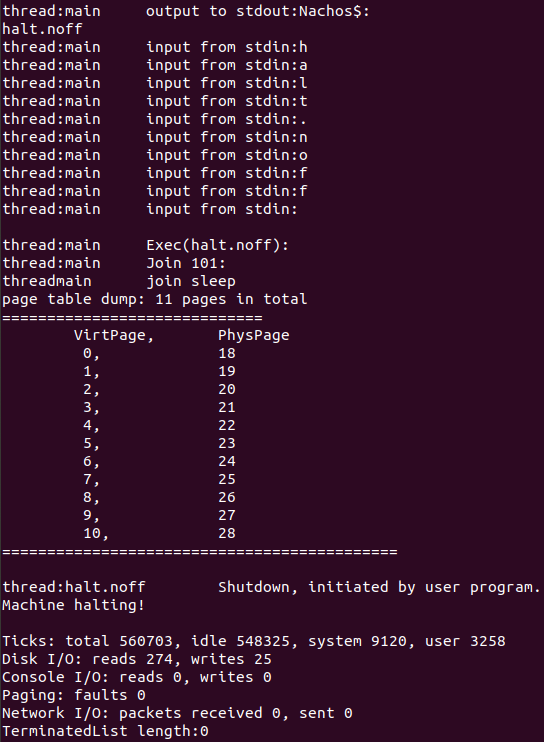
**8.6.4 执行/test/exec.noff**

1、/test/exec.noff



**8.6.5 执行halt.noff**

1、halt.noff



**三、总结**

1、操作系统课程设计和理论课程相得益彰。通过本次课程设计，我强化了对操作系统相关知识的理解，锻炼了编程能力。

2、通过实验三，我在Nachos上利用信号量实现了线程同步。阅读源码和理论课程相结合，对线程调度、线程切换、线程同步、生产者消费者模型等概念有了更深入的理解。

3、通过实验四和实验五，我在Nachos实现了具有多级目录和文件大小扩展的文件系统。对Nachos文件系统的源码进行阅读，并与理论课程相结合，对FCB、打开文件表、文件描述符、目录、inode、磁盘读写、驱动程序、IO中断等概念有了更加深刻的理解。

4、通过实验六、实验七和实验八，我在Nachos上实现了运行用户程序、运行shell脚本、地址空间扩展、PCB整合、系统调用。对Nachos模拟CPU和执行用户进程的源码进行阅读，并与理论课程相结合，对PCB、中断、异常、系统调用、进程映射内核线程、进程打开文件表、pid等概念有了更加深刻的理解。

5、通过对Nachos的学习，对操作系统有了更深刻更具体的认识，课本上的知识不再是空中楼阁。操作系统是计算机科学中的基础课程，操作系统课程中学习的知识使我对计算机有了更加系统性的认识。

6、感谢老师和助教的辛苦付出。

**四、参考文献**

[1] OS课程设计指南(C++)

[2] Nachos-3.4（C++）源代码

[3] gdb 使用指南

[4] MIPS 指令集

[5] i386 指令集