**操作系统课程设计指南**

**Nachos (C++)**

**韩芳溪**

**hfx@sdu.edu.cn**

**山东大学计算机科学与技术学院**

**2018年8月**

目 录

[第1章 Nachos简介 1](#_Toc523156848)

[1.1 Nachos的硬件系统 1](#_Toc523156849)

[1.1.1 CPU 1](#_Toc523156850)

[1.1.2 中断控制器 2](#_Toc523156851)

[1.1.3 Timer 3](#_Toc523156852)

[1.1.4 Console 3](#_Toc523156853)

[1.2 Nachos内核 4](#_Toc523156854)

[1.2.1 线程管理 4](#_Toc523156855)

[1.2.2 系统调用 12](#_Toc523156856)

[1.2.3 应用程序进程 14](#_Toc523156857)

[1.2.4 内存管理 15](#_Toc523156858)

[1.2.5 文件系统 15](#_Toc523156859)

[1.2.6 虚存与网络管理 17](#_Toc523156860)

[1.3 Nachos的文件及目录组织 17](#_Toc523156861)

[1.3.1 系统目录 17](#_Toc523156862)

[1.3.2 用户实验目录 19](#_Toc523156863)

[1.4 相关软件及参考资料 19](#_Toc523156864)

[1.4.1 相关软件 19](#_Toc523156865)

[1.4.2 参考资料 19](#_Toc523156866)

[第2章 Nachos系统的安装与调试（实验1） 21](#_Toc523156867)

[2.1目的与任务 21](#_Toc523156868)

[2.1.1 目的 21](#_Toc523156869)

[2.1.2 任务 21](#_Toc523156870)

[2.2 安装Nachos系统 21](#_Toc523156871)

[2.3 gcc MIPS交叉编译器的安装与测试 22](#_Toc523156872)

[2.3.1 安装gcc MIPS交叉编译器 22](#_Toc523156873)

[2.3.2 测试gcc MIPS交叉编译器 22](#_Toc523156874)

[2.4 测试Nachos 23](#_Toc523156875)

[2.4.1 Nachos的基本内核测试 23](#_Toc523156876)

[2.4.2 测试其它模块的功能 25](#_Toc523156877)

[2.5 利用gdb调试Nachos C++代码的过程与方法（C++与gdb） 25](#_Toc523156878)

[2.6 Nachos的上下文切换 27](#_Toc523156879)

[2.7 课后作业 28](#_Toc523156880)

[2.8 关于2.6的几点注记 28](#_Toc523156881)

[2.8.1 关于函数的地址 28](#_Toc523156882)

[2.8.2 关于对象的地址 32](#_Toc523156883)

[2.8.3 关于SWITCH()的返回值 33](#_Toc523156884)

[第3章Nachos的Makefiles（实验2，可选） 40](#_Toc523156885)

[3.1 目的与任务 40](#_Toc523156886)

[3.2 Nachos的Makefile文件 40](#_Toc523156887)

[3.2.1 code/下子目录中的Makefile文件 40](#_Toc523156888)

[3.2.2 code/下子目录中的Makefile.local文件 41](#_Toc523156889)

[3.2.3 code/目录下的Makefile.dep文件 41](#_Toc523156890)

[3.2.4 code/目录下的Makefile.commom文件 42](#_Toc523156891)

[3.2.5 在其它目录中修改Nachos代码并生成修改后的Nachos系统 45](#_Toc523156892)

[第4章 利用信号量实现线程同步（实验3） 53](#_Toc523156893)

[4.1 目的与任务 53](#_Toc523156894)

[4.2 背景知识 53](#_Toc523156895)

[4.2.1 信号量 53](#_Toc523156896)

[4.2.2 生产者/消费者问题 53](#_Toc523156897)

[4.2.3 Nachos的Main程序 54](#_Toc523156898)

[4.3 Things to Do 54](#_Toc523156899)

[4.4 几点注记 55](#_Toc523156900)

[4.4.1 线程的创建 55](#_Toc523156901)

[4.4.2 信号量 56](#_Toc523156902)

[4.4.3 代码实现 57](#_Toc523156903)

[4.4.4 测试 59](#_Toc523156904)

[第5章 Nachos的文件系统（实验4） 60](#_Toc523156905)

[5.1 目的与任务 60](#_Toc523156906)

[5.2 编译Nachos的文件系统 60](#_Toc523156907)

[5.3 Nachos的硬盘及文件系统 62](#_Toc523156908)

[5.4 Nachos的文件系统命令 66](#_Toc523156909)

[5.5 测试文件 66](#_Toc523156910)

[5.5.1 UNIX命令od 66](#_Toc523156911)

[5.5.2 UNIX命令hexdump 67](#_Toc523156912)

[5.6 Things to Do 67](#_Toc523156913)

[5.6.1 编译生成Nachos文件系统 67](#_Toc523156914)

[5.6.2 测试Nachos文件系统 67](#_Toc523156915)

[5.7 Questions 70](#_Toc523156916)

[5.8 Nachos文件系统在硬盘上的布局 70](#_Toc523156917)

[5.8.1 硬盘格式化 70](#_Toc523156918)

[5.8.2 复制一个文件到硬盘 72](#_Toc523156919)

[5.8.3 复制另一个文件到硬盘 75](#_Toc523156920)

[5.8.4 在硬盘上删除文件 78](#_Toc523156921)

[第6章 扩展Nachos的文件系统（实验5） 81](#_Toc523156922)

[6.1目的与任务 81](#_Toc523156923)

[6.2 Things to Do 82](#_Toc523156924)

[6.2.1 问题分析 82](#_Toc523156925)

[6.2.2 设计与实现 82](#_Toc523156926)

[6.3 对新文件系统进行测试 84](#_Toc523156927)

[6.4 关于扩展文件的几点注记 85](#_Toc523156928)

[6.4.1 nachos –ap 与nachos –hap命令的实现 86](#_Toc523156929)

[6.4.2 nachos –nap的实现 92](#_Toc523156930)

[6.4.3 nachos文件系统测试 92](#_Toc523156931)

[第7章 Nachos用户程序与系统调用（实验6） 95](#_Toc523156932)

[7.1 目的与任务 95](#_Toc523156933)

[7.2 背景知识 95](#_Toc523156934)

[7.3 Things to Do 96](#_Toc523156935)

[7.3.1 Nachos可执行程序（Nachos executables） 96](#_Toc523156936)

[7.3.2 页表的系统转储 97](#_Toc523156937)

[7.3.3 分配更大的地址空间 98](#_Toc523156938)

[7.4 几点注记 98](#_Toc523156939)

[7.4.1 Nachos应用程序 98](#_Toc523156940)

[7.4.2 页表 99](#_Toc523156941)

[7.4.3 用户进程的创建与启动 100](#_Toc523156942)

[7.4.4 PCB 103](#_Toc523156943)

[7.4.5 用户线程映射到核心线程 103](#_Toc523156944)

[7.4.6 线程调度算法 104](#_Toc523156945)

[第8章 地址空间的扩展（实验7） 105](#_Toc523156946)

[8.1 目的与任务 105](#_Toc523156947)

[8.2 背景知识 105](#_Toc523156948)

[8.3 Things to Do 106](#_Toc523156949)

[8.4 Bitmap Class 106](#_Toc523156950)

[第9章 系统调用Exec()与Exit()（实验8） 107](#_Toc523156951)

[9.1 目的与任务 107](#_Toc523156952)

[9.2 编写Nachos应用程序 107](#_Toc523156953)

[9.3 设计与实现的有关问题 108](#_Toc523156954)

[9.3.1 在哪里编写系统调用的代码 108](#_Toc523156955)

[9.3.2 Nachos系统调用机制 110](#_Toc523156956)

[9.3.3 Nachos系统调用参数传递 113](#_Toc523156957)

[9.3.4 Openfile for the User program 116](#_Toc523156958)

[9.3.5 Advance PC 119](#_Toc523156959)

[9.3.6 SpaceId 122](#_Toc523156960)

[9.3.7 Join() 122](#_Toc523156961)

[9.3.8 Exec() 123](#_Toc523156962)

[9.3.9 Exit() and Exit Status 127](#_Toc523156963)

[9.3.10 Things to Do 128](#_Toc523156964)

[9.4 关于文件的几个系统调用 128](#_Toc523156965)

[9.4.1 Create() 129](#_Toc523156966)

[9.4.2 Open() 130](#_Toc523156967)

[9.4.3 Write() 131](#_Toc523156968)

[9.4.4 Read() 132](#_Toc523156969)

[9.4.5 Close () 133](#_Toc523156970)

[参考资料 134](#_Toc523156971)

# 第1章 Nachos简介

Nachos是由加州大学伯克利分校的Tom Anderson基于C++实现的一个教学用操作系统。在学习了操作系统的概念与原理之后，通过阅读相关内容的Nachos源代码，可以更加深入地理解操作系统的相关概念与工作原理，掌握实用操作系统的开发过程与方法。

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

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

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

## 1.1 Nachos的硬件系统

首先Nachos模拟了计算机的硬件系统，像CPU、中断控制器、定时器、时钟、网卡、I/O终端等，然后在这些硬件之上运行了一个操作系统内核。

### 1.1.1 CPU

Nachos模拟的CPU是用来执行Nachos应用程序的（实验6、7、8）；

该CPU基于MIPS架构，执行MIPS指令集，用于执行Nachos的应用程序；因此Nachos的可执行程序应该由MIPS指令集组成。可以采用随Nachos一起提供的交叉编译器gcc-2.8.1-mips.tar.gz将一个Nachos的C程序（包含）编译转换成基于MIPS指令集的可执行程序，以在Nachos系统上运行。

在../test目录中提供了几个Nachos应用程序示例，如halt.c、matmult.c、sort.c、shell.c，这几个程序都是按照Nachos应用程序的语法及Nachos系统调用的要求编写的，尽管它们的扩展名为.c，但对C语言中的绝大部分函数均不提供支持；它的编译器及链接程序不是UNIX通用的GCC，而是gcc-2.8.1-mips.tar.gz中提供的gcc、as、ld等程序；由于在Makefile.dep中将这些程序的路径设置为/usr/local/mips/bin/decstation-ultrix-**，因此，一定按照第二章实验一中的说明，将交叉编译程序gcc-2.8.1-mips.tar.gz安装到/usr/local目录中。**

与Nachos CPU执行用户程序指令的过程：

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

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

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

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

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

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

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

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

注意CPU执行完一条指令后，执行interrupt->OneTick()，致使中断控制器可以响应中断；

### 1.1.2 中断控制器

中断控制器模拟了硬件中断，它提供了几个接口供外部程序使用；

1、Interrupt::SetLevel(IntOff)用于关中断；

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

3、Interrupt::Halt()实现停机操作；

4、Interrupt::Idle()当线程调度时，如果就绪队列为空，执行该函数处理到期的中断，然后返回；如果中断请求队列中的所有中断都处理完毕后，就绪队列仍然为空，则调用Interrupt::Halt()停机；

5、中断响应的时机

Nachos中断控制器对中断的处理过程不同于一个实际的机器对中断处理过程；

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

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

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

**a. 中断状态从关到开；**

**b. Nachos的CPU执行完一条应用程序指令；**

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

6、硬件设备提出中断请求

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

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

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

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

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

当发现有到期的中断，中断控制器从请求队列队首取出执行其中断处理程序。

### 1.1.3 Timer

如果运行时带有-rs参数，即nachos –rs xxx，则在system.cc中Initilize()函数初始化系统内核时会创建一个Timer设备，间隔一定时间（随机数）调用其中断处理程序，目前实现的代码如下：

static void TimerInterruptHandler(\_int dummy)

{

if (interrupt->getStatus() != IdleMode)

interrupt->YieldOnReturn();

}

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

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

### 1.1.4 Console

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

## 1.2 Nachos内核

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

### 1.2.1 线程管理

线程管理包括线程的创建、睡眠、释放CPU、调度、撤销等

**1、线程的状态**

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

enum ThreadStatus { JUST\_CREATED, RUNNING, READY, BLOCKED };

其中，状态JUST\_CREATED是新建一个线程，但尚未分配栈等资源，相当于教材中进（线）程五状态图中的状态New；

参见下述代码：（参见../threads/thread.cc）

Thread::Thread(char\* threadName)

{

name = threadName;

stackTop = NULL;

stack = NULL;

status = **JUST\_CREATED**;

#ifdef USER\_PROGRAM

space = NULL; //程映射到该核心线程的用户进（线）程的地址空间

#endif

}

**2、线程创建**

**（1）主线程main**

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

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

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

**（2）一般线程的创建**

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

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

t->Fork(SimpleThread, 1);

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

SimpleThread()函数的代码如下：

void SimpleThread(\_int which)

{

int num;

for (num = 0; num < 5; num++) {

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

currentThread->Yield();

}

}

Thread类的构造函数如下：

Thread::Thread(char\* threadName)

{

name = threadName;

stackTop = NULL;

stack = NULL;

status = **JUST\_CREATED**;

#ifdef USER\_PROGRAM

space = NULL; //

#endif

}

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

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

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

Thread::Fork()代码如下：

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

{

#ifdef HOST\_ALPHA

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

name, (long) func, arg);

#else

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

name, (int) func, arg);

#endif

StackAllocate(func, arg);

IntStatus oldLevel = interrupt->SetLevel(IntOff);

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

// are disabled!

(void) interrupt->SetLevel(oldLevel);

}

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

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

思考：该线程何时被调度执行？

**（3）Idle线程**

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

idle进程一般没有正式任务可完成，可让其清零空闲内存；

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

**3、线程就绪**

下述代码将线程设置为就绪状态，并加入就绪队列尾等待线程调度；

void Scheduler::ReadyToRun (Thread \*thread)

{

DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

thread->setStatus(READY);

readyList->Append((void \*)thread);

}

回顾教材中的线程状态转换图，有三种情况可能使一个线程变为就绪：

（1）JUST\_CREATED🡪READY,

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

（2）BLOCKED🡪READY,

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

（3）RUNNING,🡪READY,

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

**4、线程睡眠（等待、阻塞）**

下述代码使线程由执行状态RUNNING🡪 BLOCKED；

void Thread::Sleep()

{

Thread \*nextThread;

ASSERT(this == currentThread);

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

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

status = BLOCKED;

while ((nextThread = scheduler->FindNextToRun()) == NULL)

interrupt->Idle(); // no one to run, wait for an interrupt

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

}

先考察信号量的P()操作：

void Semaphore::P()

{

IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts

while (value = = 0) { // semaphore not available

queue->Append((void \*)currentThread); // so go to sleep

currentThread->Sleep();

}

value--; // semaphore available, consume its value

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

}

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

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

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

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

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

**5、释放CPU**

使线程由RUNNING,🡪READY,

void Thread::Yield ()

{

Thread \*nextThread;

IntStatus oldLevel = interrupt->SetLevel(IntOff);

ASSERT(this == currentThread);

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

nextThread = scheduler->FindNextToRun();

if (nextThread != NULL) {

scheduler->ReadyToRun(this);

scheduler->Run(nextThread);

}

(void) interrupt->SetLevel(oldLevel);

}

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

如果当前就绪队列为空，没有可以调度的线程，当前线程继续执行；

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

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

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

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

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

**6、线程终止**

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

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

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

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

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

void Thread::Finish ()

{

(void) interrupt->SetLevel(IntOff);

ASSERT(this == currentThread);

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

threadToBeDestroyed = currentThread;

Sleep(); // invokes SWITCH

// not reached

}

**7、线程撤销**

撤销一个终止的线程，就是释放为其分配的栈空间；

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

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

Thread::~Thread()

{

DEBUG('t', "Deleting thread \"%s\"\n", name);

ASSERT(this != currentThread);

if (stack != NULL)

DeallocBoundedArray((char \*) stack, StackSize \* sizeof(\_int));

}

**8、线程调度**

线程调度程序按一定的策略，动态地把[处理机](https://baike.baidu.com/item/%E5%A4%84%E7%90%86%E6%9C%BA/128842)分配给处于就绪队列中的某一个[线程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/382503)，以使之执行。

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

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

Thread \* Scheduler::FindNextToRun ()

{

return (Thread \*)readyList->Remove();

}

由上述代码可以看出，首先调度的是队首线程，因此采用的是FCFS调度算法；

指派程序（dispatcher）代码如下：

void Scheduler::Run (Thread \*nextThread)

{

Thread \*oldThread = currentThread;

#ifdef USER\_PROGRAM // ignore until running user programs

if (currentThread->space != NULL) { // if this thread is a user program,

currentThread->SaveUserState(); // save the user's CPU registers

currentThread->space->SaveState();

}

#endif

oldThread->CheckOverflow(); // check if the old thread

// had an undetected stack overflow

currentThread = nextThread; // switch to the next thread

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

SWITCH(oldThread, nextThread);

if (threadToBeDestroyed != NULL) {

delete threadToBeDestroyed;

threadToBeDestroyed = NULL;

}

#ifdef USER\_PROGRAM

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

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

currentThread->space->RestoreState();

}

#endif

}

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

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

**问题：**

（1）就绪线程何时被调度执行？

回顾教材中引起调度的4种时机，可以发现：

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

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

（c）当前线程进入睡眠，因此线程调度；（执行🡪阻塞）

（d）线程终止，引起线程调度；（执行🡪终止）

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

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

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

（3）新建线程从哪里开始执行？

ThreadRoot()是每个线程的入口地址（main线程除外）；

（4）线程何时被撤销？

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

### 1.2.2 系统调用

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

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

**1、停机系统调用**

（4）void Halt();

停机，关闭Nachos。参见../test/halt.c

**2、进程管理的系统调用**

主要有如Exec()，Exit()，Join()及Halt()，其中

（1）SpaceId Exec(char \*name);

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

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

为了实现像UNIX中创建一个进程时返回进程的pid，需要你自己定义并给出进程标识SpaceId或pid，把0~99预留给系统进程（尽管目前没有系统进程），100开始依次分配给应用程序进程；

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

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

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

（2）void Exit(int status);

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

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

（3）int Join(SpaceId id);

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

**3、文件管理系统调用**

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

typedef int OpenFileId;

void Create(char \*name);

OpenFileId Open(char \*name);

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

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

void Close(OpenFileId id);

**4、用户线程系统调用**

用户线程的创建等，这些用户线程共享用户进程的地址空间。

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

void Fork(void (\*func)());

void Yield();

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

**5、Nachos应用程序与系统调用的使用**

可以在Nachos的应用程序中使用这些Nachos的系统调用。

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

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

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

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

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

**6、编写自己的Nachos应用程序**

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

**7、关于Nachos应用程序语法**

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

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

### 1.2.3 应用程序进程

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

系统调用Fork()及Yield()

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

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

Nachos为应用程序分配地址空间后，创建相应的页表；

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

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

### 1.2.4 内存管理

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

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

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

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

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

### 1.2.5 文件系统

**1、硬盘**

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

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

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

**2、格式化硬盘**

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

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

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

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

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

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

**3、空闲块的管理**

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

**4、目录管理**

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

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

表1-1 Nachos目录表结构（200字节）

|  |  |  |
| --- | --- | --- |
| 文件名 | 是否已分配 | 文件头（FCB、索引结点）所在的扇区号 |
|  |  |  |
|  |  |  |
|  |  |  |

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

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

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

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

**5、文件**

一个文件由文件头（索引节点、FCB）+数据块组成，为文件数据所分配的硬盘块采用索引分配方式管理，每个文件头（FCB，索引结点）包括文件的属性，包括<文件大小，硬盘块数、数据块索引表>三项，如表1-2所示。

表1-2 文件头结构（128字节）

|  |  |  |
| --- | --- | --- |
| 文件大小 | 扇区块数 | 文件数据块所在的扇区列表[30] |
|  |  |  |

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

### 1.2.6 虚存与网络管理

课设暂不涉及。

## 1.3 Nachos的文件及目录组织

Nachos文件目录结构如图1-1所示。

Nachos

├─c++example

├─code

│ ├─bin

│ ├─filesys

│ │ └─test

│ ├─lab2

│ ├─lab3

│ ├─lab5

│ │ └─test

│ ├─lab7-8

│ ├─machine

│ ├─monitor

│ ├─network

│ ├─test

│ ├─threads

│ ├─userprog

│ └─vm

└─doc

图1-1 Nachos文件目录

### 1.3.1 系统目录

**（1）/**c++example：该目录包含几个C++的示例程序，以及Tom Anderson.教授编写的一本C++入门参考资料(c++.ps)，其中介绍了Nachos源代码中所涉及到的一些C++知识。在阅读Nachos源码时如果遇到关于C++语言的问题时可参阅这些资料。如果对C++不是很熟悉，最好在阅读Nachos系统源码时先回顾一下这些知识，特别是要很好地理解一下示例程序stack.cc。

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

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

（3）code/filesys：nachos的文件系统

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

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

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

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

（6）code/network: 实现网络的发送与接收过程

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

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

如果实现了这几个系统调用，该程序可作为Nachos的shell；

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

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

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

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

（10）code/vm:虚拟存储管理

### 1.3.2 用户实验目录

lab2、lab3、lab5、lab7-8是我们课程设计所使用的工作目录。

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

## 1.4 相关软件及参考资料

### 1.4.1 相关软件

32位Linux系统（如ubuntu）；

Nachos-3.4源代码（nachos-3.4.tar.gz）

gcc-2.8.1-mips交叉编译（gcc-2.8.1-mips.tar.gz）

### 1.4.2 参考资料

nachos\_introduction.pdf；

nachos\_study\_book.pdf；

a road map through nachos.pdf；

C++编程相关资料；

gcc，g++使用手册；

MIPS汇编语言；

i386汇编语言；

gdb使用手册；

make文件使用手册；

# 第2章 Nachos系统的安装与调试（实验1）

## 2.1目的与任务

### 2.1.1 目的

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

（2）安装测试gcc MIPS交叉编译器；

（3）掌握利用Linux调试工具GDB调试跟踪Nachos的执行过程；

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

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

### 2.1.2 任务

• install the LINUX operating system (optional)

• install the Nachos system and gcc mips cross compiler

• compile and test the Nachos system installed

• exercise with gdb

• context switch in Nachos

## 2.2 安装Nachos系统

Nachos安装步骤如下：

方法1：基于终端命令

(a) cd ~

(b) mkdir OSDesign （注：目录名OSDesign 可根据你的喜好自行修改）

(c) cd OSDesign

(d) copy file “Nachos-3.4-SDU.tar.gz” to file fold “OSDesign” with cp command.

(e) tar xzvf Nachos-3.4-SDU.tar.gz

(f) rm Nachos-3.4-SDU.tar.gz (注：该步骤可选)

方法2：基于Linux图形界面

(1) 进入你的用户工作目录

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

(3) 进入目录OSDesign

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

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

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

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

## 2.3 gcc MIPS交叉编译器的安装与测试

### 2.3.1 安装gcc MIPS交叉编译器

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

基于终端命令

(a) su -

(b) cd /usr/local

(c) copy file “gcc-2.8.1-mips.tar.gz” to file fold “/usr/local” with cp command.

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

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

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

### 2.3.2 测试gcc MIPS交叉编译器

在命令终端进入目录code/test，运行make，如果交叉编译器安装成功，屏幕应输出下述信息：

>>> Linking arch/unknown-i386-linux/objects/halt.coff <<<  
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/halt.o -o arch/unknown-i386-linux/objects/halt.coff  
>>> Converting to noff file: arch/unknown-i386-linux/bin/halt <<<  
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/halt.coff arch/unknown-i386-linux/bin/halt  
numsections 3   
Loading 3 sections:  
    ".text", filepos 0xd0, mempos 0x0, size 0x100  
    ".data", filepos 0x1d0, mempos 0x100, size 0x0  
    ".bss", filepos 0x0, mempos 0x100, size 0x0  
ln -sf arch/unknown-i386-linux/bin/halt halt  
>>> Linking arch/unknown-i386-linux/objects/shell.coff <<<  
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/shell.o -o arch/unknown-i386-linux/objects/shell.coff  
>>> Converting to noff file: arch/unknown-i386-linux/bin/shell <<<  
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/shell.coff arch/unknown-i386-linux/bin/shell  
numsections 3   
Loading 3 sections:  
    ".text", filepos 0xd0, mempos 0x0, size 0x200  
    ".data", filepos 0x2d0, mempos 0x200, size 0x0  
    ".bss", filepos 0x0, mempos 0x200, size 0x0  
ln -sf arch/unknown-i386-linux/bin/shell shell  
>>> Linking arch/unknown-i386-linux/objects/matmult.coff <<<  
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/matmult.o -o arch/unknown-i386-linux/objects/matmult.coff  
>>> Converting to noff file: arch/unknown-i386-linux/bin/matmult <<<  
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/matmult.coff arch/unknown-i386-linux/bin/matmult  
numsections 3   
Loading 3 sections:  
    ".text", filepos 0xd0, mempos 0x0, size 0x3c0  
    ".data", filepos 0x490, mempos 0x3c0, size 0x0  
    ".bss", filepos 0x0, mempos 0x3c0, size 0x12c0  
ln -sf arch/unknown-i386-linux/bin/matmult matmult  
>>> Linking arch/unknown-i386-linux/objects/sort.coff <<<  
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/start.o arch/unknown-i386-linux/objects/sort.o -o arch/unknown-i386-linux/objects/sort.coff  
>>> Converting to noff file: arch/unknown-i386-linux/bin/sort <<<  
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/sort.coff arch/unknown-i386-linux/bin/sort  
numsections 3   
Loading 3 sections:  
    ".text", filepos 0xd0, mempos 0x0, size 0x2c0  
    ".data", filepos 0x390, mempos 0x2c0, size 0x0  
    ".bss", filepos 0x0, mempos 0x2c0, size 0x1000  
ln -sf arch/unknown-i386-linux/bin/sort sort

## 2.4 测试Nachos

### 2.4.1 Nachos的基本内核测试

参照以下步骤测试安装的Nachos（Nachos的基本内核）是否正常工作：

**1、进入子目录 ~ /OSDesign /nachos-3.4/**

该目录包含三个子目录：c++example、code及doc

**2、进入子目录code**

该子目录下包含以下文件及目录：

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

**3、 在目录code/threads下测试nachos的基本内核**

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

....>>> Linking arch/unknown-i386-linux/bin/nachos <<<

g++ arch/unknown-i386-linux/objects/main.o ........................

..................................................................

..................................................................

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

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

**4、 运行Nachos。**

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

\*\*\* thread 0 looped 0 times

\*\*\* thread 1 looped 0 times

\*\*\* thread 0 looped 1 times

\*\*\* thread 1 looped 1 times

\*\*\* thread 0 looped 2 times

\*\*\* thread 1 looped 2 times

\*\*\* thread 0 looped 3 times

\*\*\* thread 1 looped 3 times

\*\*\* thread 0 looped 4 times

\*\*\* thread 1 looped 4 times

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

Machine halting!

Ticks: total 130, idle 0, system 130, user 0

Disk I/O: reads 0, writes 0

Console I/O: reads 0, writes 0

Paging: faults 0

Network I/O: packets received 0, sent 0

Cleaning up...

**分析输出结果：**

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

**（1）初始化Nachos的设备与内核**

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

* 处理Nachos内核所使用的一些命令行参数；
* 根据需要创建了Nachos相应的硬件设备，如中断控制器，定时器、CPU、硬盘；
* 然后基于这些设备初始化了一个Nachos的内核；，如初始化了一个线程调度程序，在硬盘上创建了文件系统，创建了Nachos的主线程（Nachos的第一个线程），以及网络通信使用的邮箱；
* 讨论：mian线程创建后，后续的代码是否在主线程中执行？

**（2）对Nachos内核进行测试（线程的创建及并发执行）**

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

上述输出结果主要是该函数的执行结果；

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

**（3）测试Nachos的锁机制及条件变量**

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

**（4）处理Nachos其它的一些命令行参数**

如显式版权信息、运行应用程序、文件系统及网络相关的一些运行参数；

**（5）终止主线程，Nachos退出**

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

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

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

### 2.4.2 测试其它模块的功能

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

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

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

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

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

例如利用gdb调试stack的过程如下：

• 打开一个命令终端，进入目录c++example，

• 运行make命令编译三个示例程序（这里我们以stack为例）

• 键入命令gdb stack在gdb中打开可执行程序stack

- 在gdb中会输出如下信息:

GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1

Copyright (C) 2016 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law. Type "show copying"

and "show warranty" for details.

This GDB was configured as "i686-linux-gnu".

Type "show configuration" for configuration details.

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>.

Find the GDB manual and other documentation resources online at:

<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from stack...done.

(gdb)

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

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

(gdb) break 129

Breakpoint 1 at 0x8048a34: file stack.cc, line 129.

(gdb)

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

(gdb) run

Starting program: /home/OSdesign/nachos-3.4/c++example/stack

Breakpoint 1, main () at stack.cc:129

129  Stack \*stack = new Stack(10);   // Constructor with an argument.  
(gdb)

- 输入next （或n），可逐条跟踪语句的执行：

131 stack->SelfTest()

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

Stack::SelfTest (this=0x8050a10) at stack.cc:108

108 int count = 17;

(gdb)

- 持续键入step或next， 可跟踪stack的执行过程。

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

其它常用的gdb命令及gdb的使用可参阅gdb目录下的gdb参考资料。

## 2.6 Nachos的上下文切换

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

void ThreadTest()

{

DEBUG(’t’, "Entering SimpleTest");

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

t->Fork(SimpleThread, 1);

SimpleThread(0);

}

//The SimpleThread() function used above is as follows:

Void SimpleThread(\_int which)

{

Int num;

for (num = 0; num < 5; num++) {

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

currentThread->Yield();

}

}

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

熟悉跟踪过程后，回答下列问题：

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

i. InterruptEnable()

ii. SimpleThread()

iii. ThreadFinish()

iv. ThreadRoot()

（2）下述线程对象的地址是多少？并说明找到这些对象地址的过程及方法。

i. the main thread of the Nachos

ii. the forked thread created by the main thread

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

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

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

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

## 2.7 课后作业

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

（1）Nachos的命令行参数及其处理

（2）主线程（main）是如何创建的？

（3）我们应该如何创建线程；

（4）Nachos是如何进行上下文切换的；

主要代码：

../threads/main.cc

../threads/system.cc（.h）

../threads/thread.cc（.h）

../threads/scheduler.cc（.h）

../threads/switch-linux.s

## 2.8 关于2.6的几点注记

在../threads命令下运行make，生成nachos系统；

gdb ./nachos利用gdb调试nachos；

### 2.8.1 关于函数的地址

gdb利用命令break（简写b）可以在程序的某一位置设置断点，当程序执行到该断点时暂停程序的执行（被设置断点的语句尚未执行）。

(gdb) list （l）显示程序源码，包括行号

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

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

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

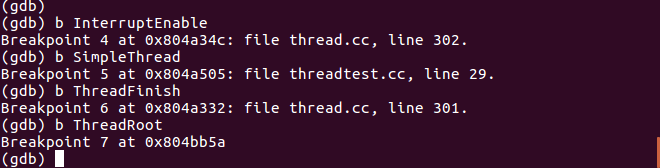


图 2-1 断点及断点地址

特别指出的是关于ThreadRoot()的地址问题。

一般情况下，汇编语言里的函数大部分情况下都符合以下的函数结构：

.globl fun\_name

.type fun\_name, @function

fun\_name:

pushl %ebp

movl %esp, %ebp

=> 函数主体代码>

leave

ret

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

如果函数有初始化的局部变量，函数对应的汇编代码结构如下：

**.globl fun\_name**

**.type fun\_name, @function**

**fun\_name:**

**pushl %ebp**

**movl %esp, %ebp**

=> **subl $Stack\_Size, %esp** #为初始化的局部变量，或为其它管理

#预留栈空间Stack\_Size个字节

**movl $localVarValue, offset(%esp)**  #将已初始化局部变量入栈

依次利用movl指令将初始化的局部变量入栈

**<函数主体代码>**

**leave**

**ret**

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

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

movl %ebp,%esp

pop $ebp

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

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

**注：i386的栈是向下生长的**

如：SimpleThread()的源代码如下：

void SimpleThread(\_int which)

{

int num;

for (num = 0; num < 5; num++) {

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

currentThread->Yield();

}

}

对应的汇编代码如图2-2所示。

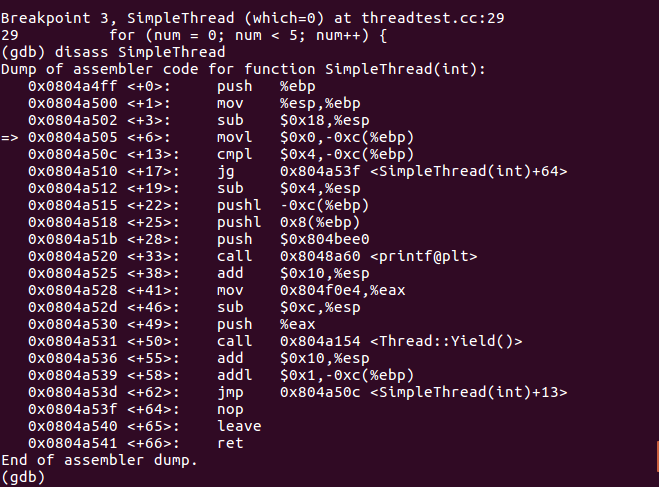


图2-2 gdb中SimpleThread()对应的汇编代码

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

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

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

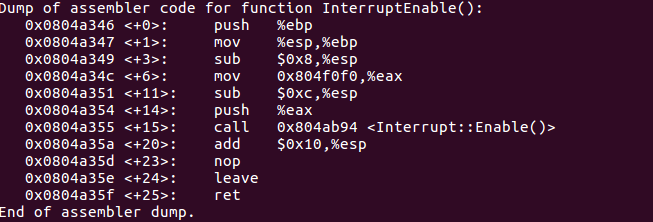


图2-3 gdb中InterruptEnable ()对应的汇编代码

对于函数ThreadRoot()，其汇编代码图2-4所示。

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

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

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

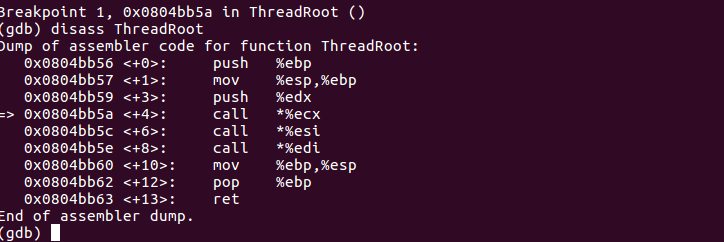


图2-4 gdb中ThreadRoot ()对应的汇编代码

### 2.8.2 关于对象的地址

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

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

过程：

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

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

过程：

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

查看(gdb) p \*t会输出哪些信息。

### 2.8.3 关于SWITCH()的返回值

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

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

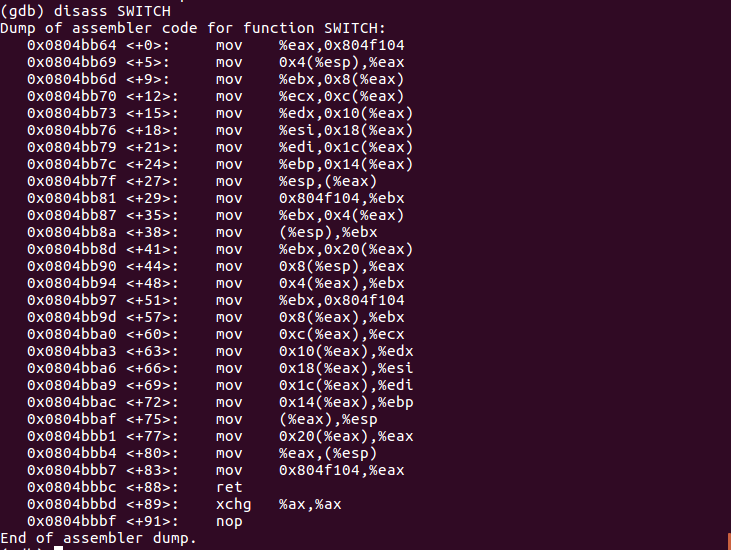


图2-5 gdb中SWITCH ()对应的汇编代码

查看SWITCH()的返回地址有两种方法：

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

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

可用(gdb) info r查看寄存器的状态，看看原线程的地址是多少；

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

可用(gdb) info r查看寄存器的状态，看看新线程的地址是多少；

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

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

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

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

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

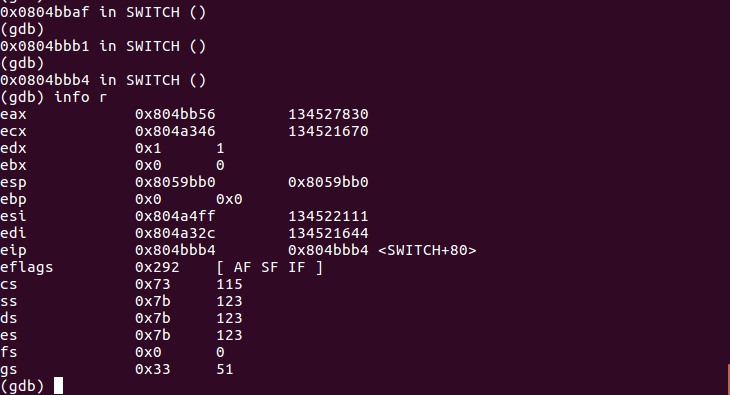


图2-6 寄存器exa的值为SWITCH()的返回值

（2）方法2

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

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



图2-7第一次调用SWITCH()返回地址

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

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

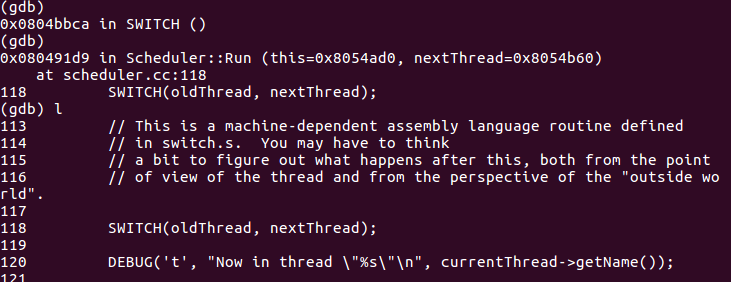


图2-8 第二次调用SWITCH()返回地址

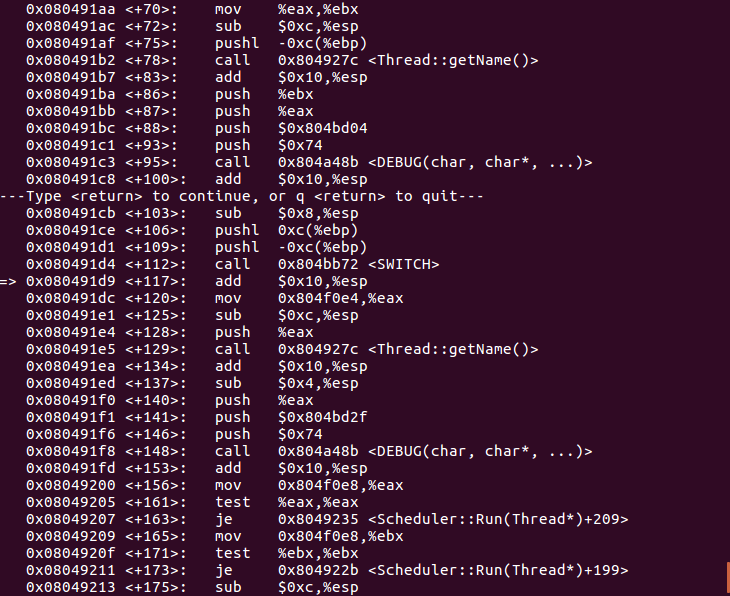


图2-9 Scheduler::Run()的部分汇编代码

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

**为什么SWITCH()函数的第一次返回位置与后续的返回位置不同？**

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

ThreadTest()

{

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

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

t->Fork(SimpleThread, 1);

SimpleThread(0);

}

SimpleThread(\_int which)

{

int num;

for (num = 0; num < 5; num++) {

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

currentThread->Yield();

}

}

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

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

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

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

Thread::Yield ()

{

Thread \*nextThread;

IntStatus oldLevel = interrupt->SetLevel(IntOff);

ASSERT(this == currentThread);

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

nextThread = scheduler->FindNextToRun();

if (nextThread != NULL) {

scheduler->ReadyToRun(this);

scheduler->Run(nextThread);

}

(void) interrupt->SetLevel(oldLevel);

}

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

注意这里的SWITCH()是第一次被调用。

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

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

# 第3章Nachos的Makefiles（实验2，可选）

## 3.1 目的与任务

该实验在目录lab2中完成。

（1）熟悉Nachos的makefiles的结构；

（2）熟悉如何在几个lab文件目录中构造相应的Nachos系统；

## 3.2 Nachos的Makefile文件

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

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

Nachos的makefile文件结构大致如下：

../code/Makefile.common

/Makefile.dep

|

|

/threads/Makefile

/Makefile.local

|

|

/filesys/Makefile

/Makefile.local

|

|

..

### 3.2.1 code/下子目录中的Makefile文件

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

Makefield文件的内容主要包括下述两条语句：

include Makefile.local

include ../Makefile.common

### 3.2.2 code/下子目录中的Makefile.local文件

该文件的作用主要是对一些编译、链接及运行时所使用的宏进行定义。

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

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

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

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

### 3.2.3 code/目录下的Makefile.dep文件

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

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

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

如Ubuntu平台：

uname = $(shell uname)

$(warning $(uname)) #输出变量uname的内容

#Ubuntu平台输出uname的内容为”Linux”

#或利用下述两条echo语句输出变量uname的内容

#echo $(uname)

#@echo $(uname)

Makefile.dep中关于Ubuntu等类Linux平台的宏定义如下：

# 386, 386BSD Unix, or NetBSD Unix (available via anon ftp

# from agate.berkeley.edu)

ifeq ($(uname),Linux)

HOST\_LINUX=-linux

HOST = -DHOST\_i386 -DHOST\_LINUX

CPP=/lib/cpp

CPPFLAGS = $(INCDIR) -D HOST\_i386 -D HOST\_LINUX

arch = unknown-i386-linux

ifdef MAKEFILE\_TEST

#GCCDIR = /usr/local/nachos/bin/decstation-ultrix-

GCCDIR = /usr/local/mips/bin/decstation-ultrix-

LDFLAGS = -T script -N

ASFLAGS = -mips2

endif

endif

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

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

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

arch\_dir = arch/$(arch)

obj\_dir = $(arch\_dir)/objects

bin\_dir = $(arch\_dir)/bin

depends\_dir = $(arch\_dir)/depends

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

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

### 3.2.4 code/目录下的Makefile.commom文件

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

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

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

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

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

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

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

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

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

s\_ofiles = $(SFILES:%.s=$(obj\_dir)/%.o)

c\_ofiles = $(CFILES:%.c=$(obj\_dir)/%.o)

cc\_ofiles = $(CCFILES:%.cc=$(obj\_dir)/%.o)

ofiles = $(cc\_ofiles) $(c\_ofiles) $(s\_ofiles)

program = $(bin\_dir)/nachos

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

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

$(program): $(ofiles)

$(bin\_dir)/% :

@echo ">>> Linking" $@ "<<<"

$(LD) $^ $(LDFLAGS) -o $@

ln -sf $@ $(notdir $@) #在当前目录下建立文件

# arch/unknown-i386-linux/bin/nacho的符号链接文件。

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

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

$(obj\_dir)/%.o: %.cc

@echo ">>> Compiling" $< "<<<"

$(CC) $(CFLAGS) -c -o $@ $<

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

$(obj\_dir)/%.o: %.c

@echo ">>> Compiling" $< "<<<"

$(CC) $(CFLAGS) -c -o $@ $<

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

$(obj\_dir)/%.o: %.s

@echo ">>> Assembling" $< "<<<"

$(CPP) $(CPPFLAGS) $< > $(obj\_dir)/tmp.s

$(AS) -o $@ $(obj\_dir)/tmp.s

rm $(obj\_dir)/tmp.s

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

s\_dfiles = $(SFILES:%.s=$(depends\_dir)/%.d)

c\_dfiles = $(CFILES:%.c=$(depends\_dir)/%.d)

cc\_dfiles = $(CCFILES:%.cc=$(depends\_dir)/%.d)

dfiles = $(cc\_dfiles) $(c\_dfiles) $(s\_dfiles)

$(depends\_dir)/%.d: %.cc

@echo ">>> Building dependency file for " $< "<<<"

@$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \

| sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@'

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

$(depends\_dir)/%.d: %.c

@echo ">>> Building dependency file for" $< "<<<"

@$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \

| sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@'

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

$(depends\_dir)/%.d: %.s

@echo ">>> Building dependency file for" $< "<<<"

@$(SHELL) -ec '$(CPP) -MM $(CPPFLAGS) $< \

| sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@'

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

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

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

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

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

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

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

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

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

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

首先，将目录code/threads中的code/threads/arch及其子目录全部复制到目录lab2中，并将三个目录arch/unknown-i386-linux/bin/、arch\unknown-i386-linux\depends\及arch\unknown-i386-linux\objects\清空（我们一般使用intel公司的芯片及相应的Linux系统，因此使用这三个目录）。

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

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

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

修改INCPATH的两种方法：

**方法1：**在INCPATH中将目录../lab2添加到../threads之前，如下：

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

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

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

如下示例可说明这一问题：

命令终端进入目录lab2，将lab2作为当前目录。

（1）利用ls显示目录lab2下的文件：

**命令控制端：../lab2$ ls**

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

（2）利用make编译链接生成Nachos

**命令控制端：../lab2$ make**

…...

>>> Linking arch/unknown-i386-linux/bin/nachos <<<

g++ arch/unknown-i386-linux/objects/main.o ........

.......................

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

**../lab2$ ls**

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

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

**命令控制端：.. /lab2$ touch scheduler.h**

**命令控制端：../lab2$ make**

>>> Building dependency file for scheduler.cc <<<

....................

>>> Compiling scheduler.cc <<<

....................

g++ -g -Wall -Wshadow -I../lab2 -I../threads -I../machine

-DTHREADS -DHOST\_i386 -DHOST\_LINUX -DCHANGED

-c -o arch/unknown-i386-linux/objects/scheduler.o scheduler.cc

>>> Linking arch/unknown-i386-linux/bin/nachos <<<

g++ arch/unknown-i386-linux/objects/main.o ............

....................

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

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

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

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

**命令控制端：../lab2$ make**

>>> Building dependency file for ../machine/timer.cc <<<

>>> Building dependency file for ../machine/sysdep.cc <<<

...

>>> Building dependency file for ../threads/system.cc <<<

>>> Building dependency file for ../threads/synch.cc <<<

>>> Building dependency file for ../threads/main.cc <<<

>>> Compiling ../threads/main.cc <<<

…….

>>> Linking arch/unknown-i386-linux/bin/nachos <<<

g++ arch/unknown-i386-linux/objects/main.o .........

..........

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

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

# The following set of rules define how to build dependency files

# automatically from various source files. These rules have been

# taken from the gmake documentation with minor modifications.

$(depends\_dir)/%.d: %.cc

@echo ">>> Building dependency file for " $< "<<<"

@$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \

| sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@'

$(depends\_dir)/%.d: %.c

@echo ">>> Building dependency file for" $< "<<<"

@$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \

| sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@'

$(depends\_dir)/%.d: %.s

@echo ">>> Building dependency file for" $< "<<<"

@$(SHELL) -ec '$(CPP) -MM $(CPPFLAGS) $< \

| sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@'

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

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

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

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

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

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

**命令控制端：../threads$ grep scheduler.h \***

grep: arch: Is a directory

scheduler.cc: #include "scheduler.h"

scheduler.h:// scheduler.h

system.h: #include "scheduler.h"

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

**命令控制端：../threads$** grep system.h \*

**屏幕输出包含system.h的文件列表：**

grep: arch: Is a directory

main.cc:#include "system.h"

scheduler.cc:#include "system.h"

synch.cc:#include "system.h"

synchtest.cc:#include "system.h"

system.cc:#include "system.h"

system.h:// system.h

thread.cc:#include "system.h"

threadtest.cc:#include "system.h"

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

system.h

main.cc

synch.cc

synchtest.cc

system.cc

thread.cc

threadtest.cc

现在lab2目录下包含下述文件：

**命令控制端：../lab2$ ls**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Makefile | arch/ | scheduler.cc | synchtest.cc | thread.cc |
| Makefile.local | main.cc | scheduler.h | system.cc | threadtest.cc |
| Makefile.local˜ | nachos | synch.cc | system.h |  |

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

（i）首先利用touch更新shcduler.h的时间

**命令控制端：../lab2$ touch scheduler.h**

**命令控制端：../lab2$ make**

>>> Building dependency file for ../machine/timer.cc <<<

...

>>> Compiling main.cc <<<

g++ -g -Wall -Wshadow -I../lab2 -I../threads -I../machine –DTHREADS

-DHOST\_i386 -DHOST\_LINUX -DCHANGED

-c -o arch/unknown-i386-linux/objects/main.o main.cc

........

>>> Linking arch/unknown-i386-linux/bin/nachos <<<

g++ arch/unknown-i386-linux/objects/main.o .......

............

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

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

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

**命令控制端：../lab2$ make**

make: ‘arch/unknown-i386-linux/bin/nachos’ is up to date.

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

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

命令touch的有关参数及使用方法请参阅网上资料。

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

命令**grep**的有关参数及使用方法请参阅网上资料。

**方法2：**该方法所采用的解决措施比方法1简单的多。

该方法利用g++的预处理器定义的参数 -I-。在linux 中命令man gcc所给出的关于-I-的描述如下：

-I-

……………..

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

……………..

也就是说在g++命令中包含参数**-I-**时**，**预处理程序在处理.cc源程序的头文件时，强制预处理程序从紧随-I-其后的-I所给出的目录中查找源程序所使用的头文件，而不是使用.cc源程序所在的目录中的头文件。因此，我可以修改、code/lab2/目录中的Makefile.loca中的宏（变量）INCPATH += -I- -I../lab2 -I../threads -I../machine

这样，g++的预处理器会依次从目录../lab2、../threads及../machine中使用所涉及的头文件。

因此，我们只需从code/threads目录中复制scheduler.cc及scheduler.h文件到code/lab2中，其它文件不需要复制。当code/lab2/scheduler.h被修改后，在code/lab2目录中使用make命令编译生成新的Nachos时，所有模块就会感知到code/lab2/scheduler.h文件的变化。

首先，我们确认一下目录lab2中只有scheduler.cc及scheduler.h两个需要修改的文件。

**命令控制端：../lab2$ ls**

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

将code/lab2/arch/unknown-i386-linux/depends及code/lab2/arch/unknown-i386-linux/objects两个目录清空。命令窗口中进入code/lab2，运行make，编译生成Nachos后，再利用**touch ../threads/scheduler.h**命令更新threads目录下的scheduler.h文件的时间，再运行make，你会发现各程序模块没有使用threads目录下的scheduler.h（而是使用lab2目录下的scheduler.h）

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

**命令控制端：../lab2$ make，**屏幕输出：

**make: ‘arch/unknown-i386-linux/bin/nachos’ is up to date.**

说明各程序模块没有使用threads目录下的scheduler.h。

但如果更新code/lab2下的scheduler.h的时间后，再运行make，会引起相关模块的重新编译，如下：

**命令控制端：../lab2$ touch scheduler.h**

**命令控制端：../lab2$ make，**屏幕输出：

>>> Building dependency file for ../machine/timer.cc <<<

...

>>> Compiling ../threads/main.cc <<<

g++ -g -Wall -Wshadow -I- -I../lab2 -I../threads -I../machine –DTHREADS

-DHOST\_i386 -DHOST\_LINUX -DCHANGED

-c -o arch/unknown-i386-linux/objects/main.o ../threads/main.cc

...

>>> Linking arch/unknown-i386-linux/bin/nachos <<<

g++ arch/unknown-i386-linux/objects/main.o ...........

....................

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

说明各程序模块使用code/lab2目录下的scheduler.h。

# 第4章 利用信号量实现线程同步（实验3）

## 4.1 目的与任务

目录code/threads中，code/threads目录中synch.cc中的Semaphore类实现了Nachos的信号量及相应的P、V操作。

首先，阅读synch.cc， 理解Nachos中信号量是如何实现的；

阅读code/lab3目录下的ring,h、ring.cc、main.cc及prodcons++.cc，在理解它们工作机理的基础上，补充目录lab3中提供的代码，利用Nachos实现的信号量写一个producer/consumer problem测试程序。

需要重点阅读的代码：../lab3/ring.h, ring.cc, prodcons++.cc；../threads/synch.h, synch.cc； ../threads/thread.cc，../threads/shecduler以及../threads/threadtest.cc。

../threads/threadtest.cc示例了如何利用Thread::Fork()创建Nachos’线程的方法；

该任务完成后，你会有如下收获：

（1）理解Nachos中信号量是如何实现的

（2）理解Nachos中是如何利用信号量实现producer/consumer problem；

（3）理解Nachos中如何创建并发线程；

（4）理解Nachos中如何测试与调试程序；

## 4.2 背景知识

### 4.2.1 信号量

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

### 4.2.2 生产者/消费者问题

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

### 4.2.3 Nachos的Main程序

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

请阅读code/threads中的main.cc，理解如下问题：

（1）Nachos是如何解释处理命令行参数的；

（2）Nachos的内核是如何初始化的；

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

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

## 4.3 Things to Do

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

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

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

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

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

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

设计步骤：

在code/lab3目录中，

（a）详细阅读并深刻理解ring.h及ring.cc中的所有代码；

（b）阅读并理解main.cc的功能；

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

（d）利用make编译生成新的Nachos，并测试其功能是否满足设计要求；

根据生产者/消费者问题的功能定义，你的实现应该满足如下条件:

（1）生产者线程所产生的所有的消息，都应该被消费者接收并保存到输出文件中（tem\_0，temp\_1，…）

（2）每个消息只能被接收一次且在文件保存一次

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

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

*•* the contents of tmp 0:

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

• the contents of tmp 1:

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

在运行./nachos时，可以加上参数（随机数种子），如./nachos -rs *seed-number。*

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

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

## 4.4 几点注记

该实验需要你：

阅读../threads/sysch.cc关于信号量及P、V操作的实现；

阅读threadtest.cc中线程的创建与使用；

阅读../threads/thread.cc关于线程创建的方法；

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

理解生产者-消费者模型；

### 4.4.1 线程的创建

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

void SimpleThread(\_int which)

{

int num;

for (num = 0; num < 5; num++) {

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

currentThread->Yield();

}

}

void ThreadTest()

{

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

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

t->Fork(SimpleThread, 1); //线程体是SimpleThread(1)，注意参数1的传递

SimpleThread(0);

}

Thread类的构造方法构造了一个new线程；

Thread::Thread(char\* threadName)

{

name = threadName;

stackTop = NULL;

stack = NULL;

status = JUST\_CREATED;

#ifdef USER\_PROGRAM

space = NULL; //用户线程内存标识

#endif

}

### 4.4.2 信号量

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

建立信号量：

Semaphore \*mutex ,\*nempty, \*nfull;

mutex = new Semaphore("mutex", 1);

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

nfull = new Semaphore("nfull", 0);

其后就可以使用信号量提供的P()、V()操作实现线程的互斥与同步；如：

nempty->P(); //同步在前

mutex->P(); //互斥在后

mutex ->V();

nfull->V();

### 4.4.3 代码实现

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

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

// Put the code to construct all the semaphores here.

//根据生产者-消费者模型的思想，创建几个信号量，并赋予初值

//参见../threads/ synch.cc及synch.h对信号量及P、V的定义

//利用Nachos提供的信号量类Semaphore创建互斥信号量mutex，初值为1，

//实现生产者与消费者互斥访问缓冲池ring，如

mutex = new Semaphore(“mutex”,1)

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

nempty= new Semaphore(“nempty”,….);

nfull= new Semaphore(“nfull”, …);

// Put the code to construct a ring buffer object with size

//BUFF\_SIZE here.

//利用Nachos提供的Ring类创建一个缓冲池对象ring（参见../lab3/ring.cc）

ring = new Ring(BUFF\_SIZE);

//利用../threads/thread.cc中void Thread::Fork(VoidFunctionPtr func, \_int arg)

//创建N\_PROD个生产者线程，让其执行代码Producer(\_int which)，如：

// create and fork N\_PROD of producer threads

for (i=0; i < N\_PROD; i++)

{

// this statemet is to form a string to be used as the name for

// produder i.

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

// Put the code to create and fork a new producer thread using

// the name in prod\_names[i] and

// integer i as the argument of function "Producer"

// ...

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

//线程名为"producer\_1"，"producer\_2"等

produce[i] = new Thread(prod\_name[i]);

//该线程执行producer函数，并将参数i的值传给producer函数，

//即执行producer(i)

//调用Fork()后，该线程处于就绪状态，并放到就绪队列尾部

//参见Thread::Fork(VoidFunctionPtr func, \_int arg)及

//Scheduler::ReadyToRun (Thread \*thread)

produce[i] -> Fork(producer,i);

//参照该上述方法创建几个消费者线程

……

};

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

void Producer(\_int which)

{

int num;

slot \*message = new slot(0,0); //参见ring.cc中的slot类

// This loop is to generate N\_MESSG messages to put into to ring buffer

// by calling ring->Put(message). Each message carries a message id

// which is represened by integer "num". This message id should be put

// into "value" field of the slot. It should also carry the id

// of the producer thread to be stored in "thread\_id" field so that

// consumer threads can know which producer generates the message later

// on. You need to put synchronization code

// before and after the call ring->Put(message). See the algorithms in

// page 182 of the textbook.

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

// Put the code to prepare the message here.

// ...

message->value = num;

message-> thread\_id = which;

// Put the code for synchronization before ring->Put(message) here.

// ...

nempty->P(); //同步在前

mutex->P(); //互斥在后

ring->Put(message);

// Put the code for synchronization after ring->Put(message) here.

// ...

mutex ->V();

nfull ->V();

}

}

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

### 4.4.4 测试

如果测试结果不能出现4.3所示的效果，可加上一个随机数种子，如

nachos –rs 5，以使Nachos在初始化内核时创建一个定时器Timer，实现抢先式的时间片轮转线程调度算法，实现生产者线程与消费者线程之间的分时操作；

# 第5章 Nachos的文件系统（实验4）

## 5.1 目的与任务

Nachos模拟了一个硬盘（code/filesys/DISK），实现的文件系统比较简单，该实验将熟悉一些文件系统的操作命令，观察这些命令对硬盘（DISK）的影响。

该实验完成后的收获：

（1）Nacho文件系统提供了哪些功能；

（2）如何检查Nachos模拟的硬盘上的内容；

Nachos实现的文件系统实现了两个版本，通过宏FILESYS\_STUB与FILESYS进行条件编译所产生的两个不同的实现（参见../filesys/filesys.h）； 宏FILESYS\_STUB实现的文件操作直接利用UNIX所提供的系统调用实现，操作的不是硬盘DISK上的文件；宏FILESYS实现的文件系统是通过OpenFile类对DISK上的文件进行操作（尽管最终也是使用UNIX的系统调用实现）；

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

主要代码文件：

../filesys/fstest.cc

/synchdisk.cc(.h)

/openfile.cc(.h)

/filesys.cc(.h)

/directory.cc(.h)

/filehdr.cc(.h)

../threads/main.cc

../machine/disk.cc(.h)

../userprog/bitmap.cc(.h)

## 5.2 编译Nachos的文件系统

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

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

ifndef MAKEFILE\_FILESYS

define MAKEFILE\_FILESYS

yes

endef

# You can re-order the assignments. If filesys comes before userprog,

# just re-order and comment the includes below as appropriate.

include ../threads/Makefile.local

include ../filesys/Makefile.local

#include ../userprog/Makefile.local

#include ../vm/Makefile.local

#include ../filesys/Makefile.local

include ../Makefile.dep

include ../Makefile.common

endif # MAKEFILE\_FILESYS

code/filesys中的Makefile.local文件内容如下：

ifndef MAKEFILE\_FILESYS\_LOCAL

define MAKEFILE\_FILESYS\_LOCAL

yes

endef

# Add new sourcefiles here.

CCFILES +=bitmap.cc\

directory.cc\

filehdr.cc\

filesys.cc\

fstest.cc\

openfile.cc\

synchdisk.cc\

disk.cc

ifdef MAKEFILE\_USERPROG\_LOCAL

DEFINES := $(DEFINES:FILESYS\_STUB=FILESYS)

else

INCPATH += -I../userprog -I../filesys

DEFINES += -DFILESYS\_NEEDED -DFILESYS

endif

endif # MAKEFILE\_FILESYS\_LOCAL

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

## 5.3 Nachos的硬盘及文件系统

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

**硬盘及文件系统具有以下特点：**

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

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

即，如果字节序号从0字节开始，则每个扇区对应的字节序如表5-1所示。

表5-1 Nachos文件系统磁盘布局

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

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

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

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

**（5）采用一级目录结构（单级目录结构），最多可创建10个文件**

**（6）一个目录文件由“文件头+目录表”组成**

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

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

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

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

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

表5-2 Nachos文件目录表（目录文件结构）

|  |  |  |
| --- | --- | --- |
| **文件名** | **inUse** | **文件头（索引节点）所在扇区号** |
| main.cc | 1 | 4 |
|  |  |  |
|  |  |  |

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

**（7）一个文件由“文件头+数据块组成”，**

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

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

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

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

注：文件在磁盘上的分配方法一般有三种：连续文件、链接文件及索引文件；

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

Nachos一个文件头大小为128字节，恰好占用一个扇区（Nachos有意将一个文件头存储到一个扇区中），由于int numBytes与int numSectors已经占用两个整数，因此直接块dataSectors[NumDirect]数组的最大项数由下式确定：NumDirect =(SectorSize - 2 \* sizeof(int)) / sizeof(int))=(128-2\*4)/4=30块，即每个文件的数据最多存储到30个扇区中，因此每个文件最大为30\*128字节=3KB，一个文件头大小=4+4+30\*4=128字节。（参见../filesys/filehdr.h及../filesys/filehdr.cc）

**（9）硬盘空闲块的管理**

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

当一个扇区空闲，位示图中对应的位为0，否则为1；

位示图也是一个文件，由文件头+数据块组成，文件头保存在第0号扇区中；

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

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

**（11）目录文件（根目录）的文件头存储在第1号扇区；**

**注：关于0号与1号扇区及位示图文件头及目录表文件头**

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

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

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

综上所述，Nachos文件系统在硬盘DISK的布局如表5-3所示。

硬盘标识（魔数）：4bytes

表5-3 Nachos的磁盘布局

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

## 5.4 Nachos的文件系统命令

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

注：可选参数[-d f]的作用是打印出所有与文件系统有关的调试信息。

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

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

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

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

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

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

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

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

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

## 5.5 测试文件

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

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

### 5.5.1 UNIX命令od

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

0000000 T h i s i s t h e s p r i

0000020 n g o f o u r d i s c o n

0000040 t e n t . \n

0000046

每一行输出16个字符，字符在文件中的偏移量（左边数字）以8进制表示；

### 5.5.2 UNIX命令hexdump

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

0000000 T h i s i s t h e s p r i

0000010 n g o f o u r d i s c o n

0000020 t e n t . \n

0000026

偏移量以16进制表示。

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

00000000 54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69 |This is the spri|

00000010 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e |ng of our discon|

00000020 74 65 6e 74 2e 0a |tent..|

00000026

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

## 5.6 Things to Do

### 5.6.1 编译生成Nachos文件系统

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

### 5.6.2 测试Nachos文件系统

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

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

（b）运行nachos –D，显示硬盘DISK中的文件系统，屏幕输出：

Bit map file header :

FileHeader contents. File size: 128. File blocks:

2

File contents:

\1f\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

Directory file header:

FileHeader contents. File size: 200. File blocks:

3 4

File contents:

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

Bitmap set:

0, 1, 2, 3, 4,

Directory contents:

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

Machine halting!

Ticks: total 5310, idle 5000, system 310, user 0

Disk I/O: reads 10, writes 0

Console I/O: reads 0, writes 0

Paging: faults 0

Network I/O: packets received 0, sent 0

Cleaning up...

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

（c）运行**od –c DISK**，屏幕将输出下述转储信息：

0000000 🞚 211 g E 200 \0 \0 \0 001 \0 \0 \0 002 \0 \0 \0

0000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

\*

0000200 \0 \0 \0 \0 310 \0 \0 \0 002 \0 \0 \0 003 \0 \0 \0

0000220 004 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

0000240 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

\*

0000400 \0 \0 \0 \0 037 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

0000420 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

\*

0400000 \0 \0 \0 \0

0400004

（d）运行**hexdump -c DISK**，屏幕将输出下述转储信息：

0000000 255 211 g E 200 \0 \0 \0 001 \0 \0 \0 002 \0 \0 \0

0000010 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

\*

0000080 \0 \0 \0 \0  \0 \0 \0 002 \0 \0 \0 003 \0 \0 \0

0000090 004 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

00000a0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

\*

0000100 \0 \0 \0 \0 037 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

0000110 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

\*

0020004

（e）运行**hexdump -C DISK**，屏幕将输出下述转储信息：

0000000 ab 89 67 45 80 00 00 00 00 00 00 00 00 00 00 00 |. . g E . . . . . . . . . . . .|

0000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |. . . . . . . . . . . . . . . .. .|

\*

0000080 00 00 00 00 c 8 00 00 00 02 00 00 00 03 00 00 00 |. . . . . . . . . . . . . . . .. .|

0000090 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |. . . . . . . . . . . . . . . .. .|

00000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |. . . . . . . . . . . . . . . .. .|

\*

0000100 00 00 00 00 1f 00 00 00 00 00 00 00 00 00 00 00 |. . . . . . . . . . . . . . . .. .|

0000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |. . . . . . . . . . . . . . . .. .|

\*

0020004

或如图5-1所示。

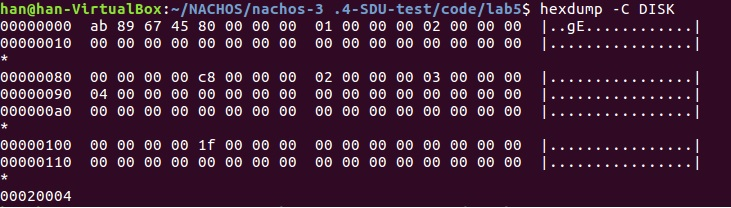
****

图5-1 初始化Nachos的文件系统

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

利用命令od –c DISK，hexdump –c DISK及hexdump –C DISK根据转储内容查看硬盘DISK有何变化。

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

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

## 5.7 Questions

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

（b）文件big的数据块（data blocks）的扇区号是多少？

（c）文件big的文件头（file header）的扇区号是多少？

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

## 5.8 Nachos文件系统在硬盘上的布局

### 5.8.1 硬盘格式化

（1）将“DISK”删除

（2）nachos –f 格式化硬盘（在硬盘DISK上创建一个文件系统）

（3）hexdump –C DISK，屏幕输出如图5-2所示。

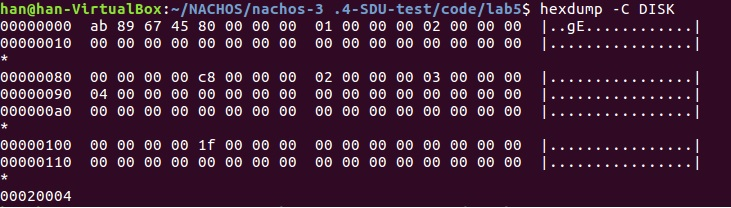
****

图5-2 Nachos文件系统初始化状态

**数据分析解读：**

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

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

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

Nachos将头文件的大小设计为128个字节，恰好保存到一个扇区中。

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

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

* 0x84~0x87：4个字节，目录表文件大小，值为0xC8，表示目录表文件大小为0xC8=200字节；

注：关于目录文件的大小，一个目录项（DirectoryEntry）大小为20个字节，Nachos为目录文件建立了10个目录项（该文件系统最多可创建10个文件），因此目录文件大小为200字节；

* 0x88~0x8B：4个字节，系统为目录文件数据所分配的扇区数，其值为0x2，表示目录文件数据需要2个扇区（目录文件大小为200字节，需要占用两个扇区）；
* 0x8C~0x8F：目录表文件第1个数据块所在的扇区号，其值为0x3，说明系统将目录表文件第1个数据块保存在第3号扇区中；
* 0x90~0x93：目录表文件第2个数据块所在的扇区号，其值为0x4，说明系统将目录表文件第2个数据块保存在第4号扇区中；

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

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

nachos -D的输出也说明了上述观点，如图5-3所示。

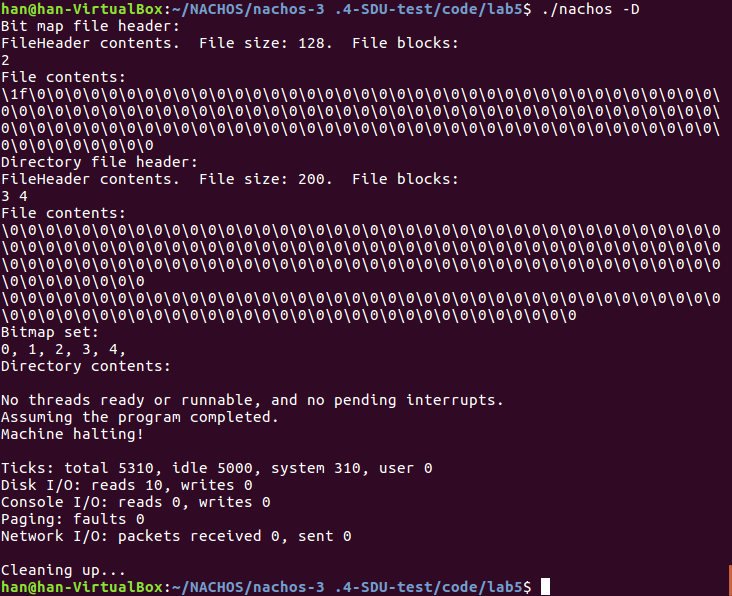


图5-3 nachos -D显示文件系统的初始化状态

### 5.8.2 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。

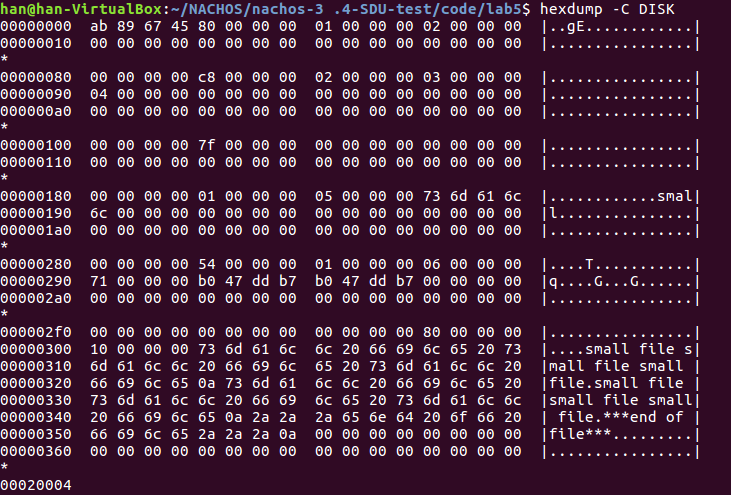


图5-4 复制一个文件small到硬盘

**数据解读：**

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

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

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

* 0x184~0x187所示的4个字节对应三元组中的inUse（注：尽管inUse占用一个字节，但根据编译器对各成员变量的起始地址对齐的原则，为inUse分配与sector相同的字节数，4个字节）表示该目录项是否已被使用，该值为1，说明该目录项正被一个文件使用。
* 0x188~0x18B所示的4个字节对应三元组中的sector，说明该目录项所记录的文件（即文件samll）的文件头所在的扇区号，这里是5，说明该目录项对应的文件（small）的文件头在第5号扇区；若要考察文件small的详细信息，需要到5号扇区访问其文件头。
* 0x18C~0x195的10个字节是文件名（文件名占用9个字节，最后一个字节是字符串结束符’\0’），这里文件名是small；目前该硬盘上只有一个文件（samll），目录文件中的其余目录项都是空的（目前共19个目录项）（3号扇区其余部分及4号扇区全部都是空的）；

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

（e）5号扇区（0x284~0x303）是“samll”的文件头；

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

根据文件头的三元组（FCB，i-node）信息，考察small文件的属性：

* 0x284~0x287：4个字节，文件大小，该值是0x54，说明small文件大小是0x54个字节；
* 0x288~0x28B：4个字节，系统为文件数据所分配的扇区数，该值为1，说明系统只为文件small的数据分配了一个扇区（一个扇区大小为128字节，文件small的大小为84字节）
* 0x28C~0x28F：系统为文件数据所分配的扇区列表。这里samll文件数据只需一个扇区，将其存放在第6号扇区；

如果想考察samll文件的内容，需要访问第6号扇区。

* 理论上讲，该扇区其余内容都应该为空（从0x290~0x303）。由于FileHeader类没有显示的构造函数，编译器就为其自动设定了构造函数FileHeader:: FileHeader {}，为该类实例化对象时，所分配的内存可能含有信息，因此可以0x290~0x303的内容，在此没有意义。

注：最好自己为FileHeader类定义构造函数，将成员变量进行初始化。如构造函数如下：

FileHeader:: FileHeader() {

numBytes=0;

numSectors=0;

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

dataSectors[i]=0;

}

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

（g）小结

0x4~0x83： 0号扇区，位示图文件头

0x84~0x103： 1号扇区，目录表文件头

0x104~0x183；2号扇区，位示图文件数据块

0x184~0x203：3号扇区，目录表文件第一个数据块

0x204~0x283：4号扇区，目录表文件第二个数据块

0x284~0x303：5号扇区，small文件头

0x304~0x383：6号扇区，small文件数据块

nachos -D的输出也说明了上述观点，如图5-5所示。

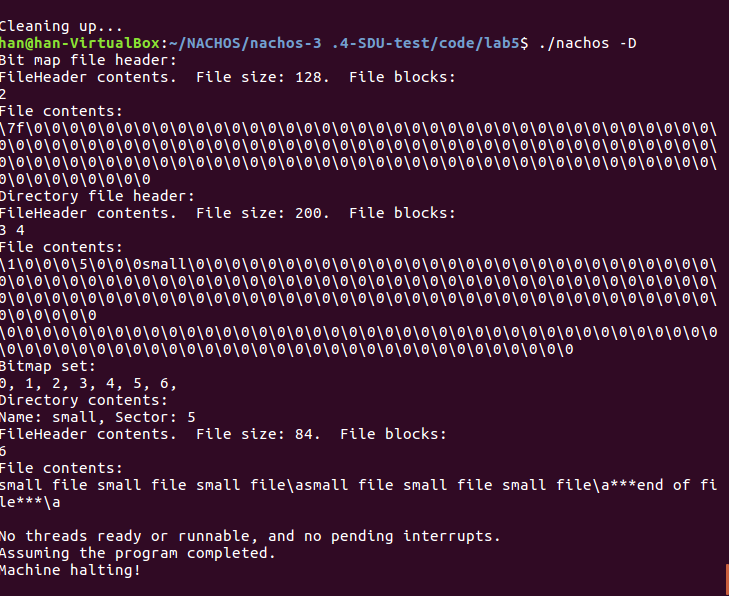
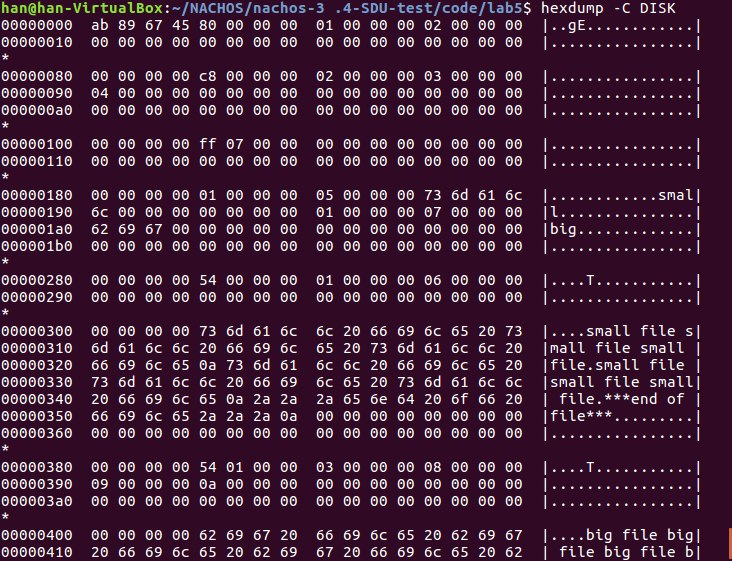


图5-5复制一个文件small到硬盘后nachos –D的输出

### 5.8.3 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。



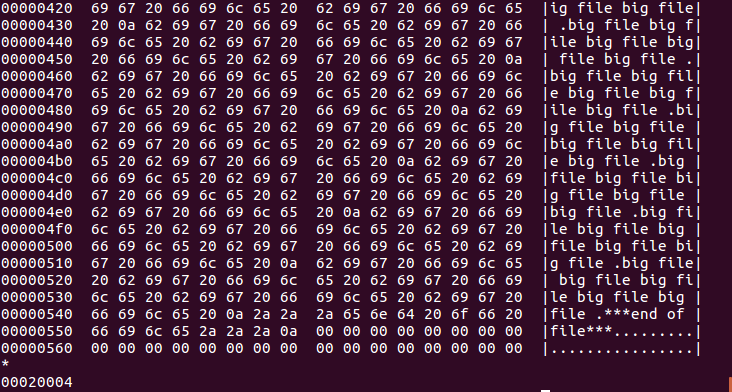


图5-6复制第二个文件big到硬盘

**数据解读：**

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

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

多出的4个扇区：big文件头1个，big数据块3个；

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

（d）7号扇区（0x384~0x403），big文件的文件头；

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

* 0x384~0x387：4个字节，big文件大小是0x154字节；
* 0x388~0x38B：4个字节，big文件的数据占用了3个扇区（一个扇区大小为128字节，文件big的大小为340个字节，需要分配3个扇区）
* 0x38C~0x38F：系统将big文件的数据分配到第0x8、0x9、0xa号3个扇区中；

如果想考察big文件的内容，需要访问这三个扇区。

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

（f）小结

0x4~0x83： 0号扇区，位示图文件头

0x84~0x103： 1号扇区，目录表文件头

0x104~0x183；2号扇区，位示图文件数据块

0x184~0x203：3号扇区，目录表文件第一个数据块

0x204~0x283：4号扇区，目录表文件第二个数据块

0x284~0x303：5号扇区，small文件头

0x304~0x383：6号扇区，small文件数据块

0x384~0x403：7号扇区，big文件头

0x404~0x483：8号扇区，big文件第1块数据（开始的128字节）

0x484~0x503：9号扇区，big文件第2块数据（中间的128字节）

0x504~0x583：10号扇区，big文件第3块数据（最后的84字节）

nachos -D的输出也说明了上述观点，如图5-7所示。

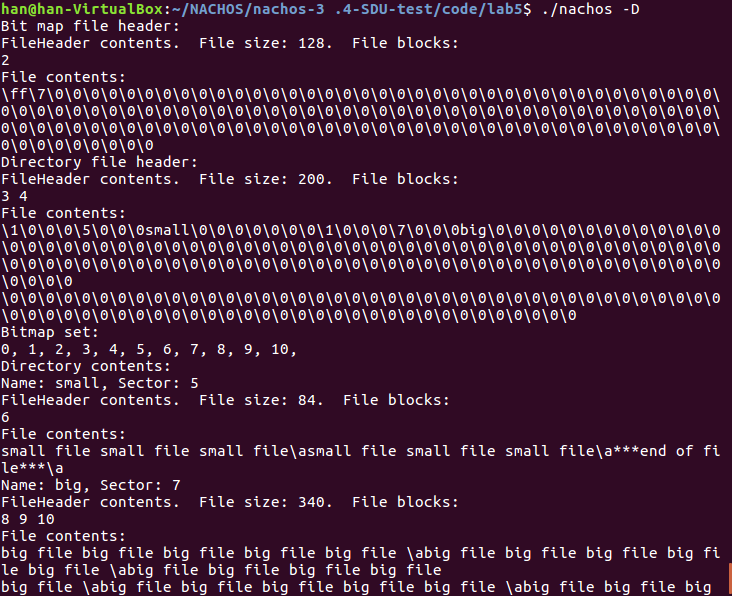


图5-7复制第二个文件big到硬盘（nachos -D）

### 5.8.4 在硬盘上删除文件

（1）hexdump –r small，删除small文件；

（2）hexdump –C DISK，屏幕输出如图5-8所示。

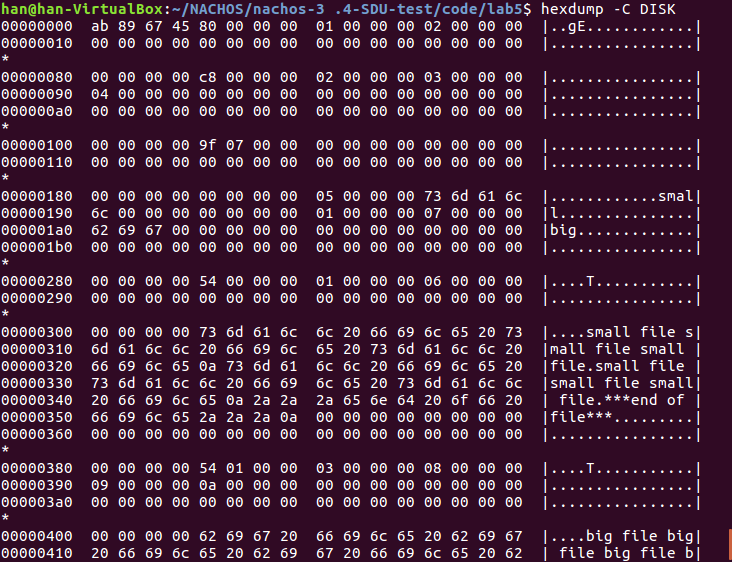


图5-8 删除文件small

删除small文件的过程：

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

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

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

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

文件的内容也未清除；

命令nachos –D的输出也说明了上述观点，如图5-9所示。

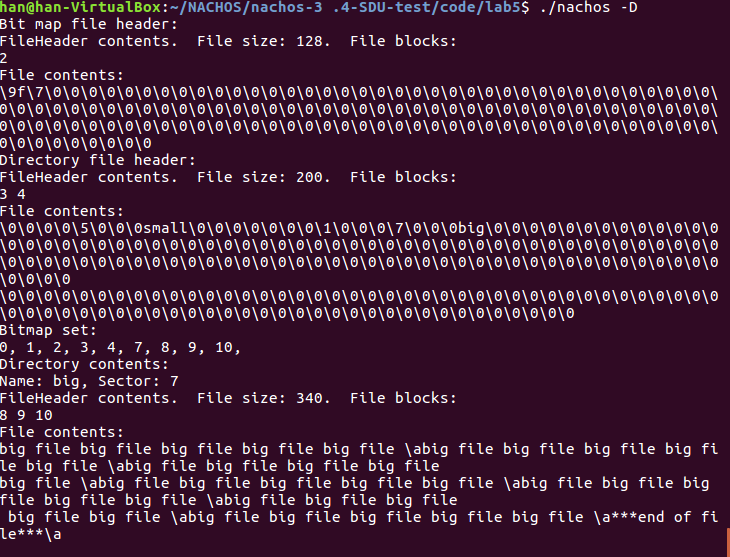


图5-9 删除文件small（nachos –D）

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

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

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

# 第6章 扩展Nachos的文件系统（实验5）

## 6.1目的与任务

目前Nachos实现的文件系统存在诸多限制，其中之一是文件大小不能扩展，即在创建文件时

该实验的任务就是让你修改Nachos的文件系统，以满足：

（1）文件创建时，其大小可初始化为0；

（2）当一个文件写入更多的数据时，其大小可随之增大；

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



图6-1 文件的扩展

Nachos的文件系统包括如下模块：

*•* class Disk //see ../machine

*•* class SynchDisk //see ../filesys

*•* class BitMap //see ../userprog

*•* class FileHeader //see ../filesys

*•* class OpenFile //see ../filesys

*•* class Directory //see ../filesys

*•* class FileSystem //see ../filesys

其文件系统结构如图6-2所示。

该实现在../lab5下完成，需要将；

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



图6-2 Nachos 文件系统结构

## 6.2 Things to Do

### 6.2.1 问题分析

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

（1）需要修改哪些模块，需要使用哪些不需要修改的模块；

（2）在那些需要修改的模块中，哪些函数需要修改，如何修改；

（3）在那些需要修改的模块中，是否需要添加函数与变量；

（4）是否需要在修改的模块中移动变量，或者从一个模块移动到另一个模块；

### 6.2.2 设计与实现

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

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

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

Append(…)函数中有如下三条语句：

// Write the inode back to the disk, because we have changed it

// openFile->WriteBack();

// printf("inodes have been written back\n");

NAppend(…)函数中有如下三条语句：

// Write the inode back to the disk, because we have changed it

// openFileTo->WriteBack();

// printf("inodes have been written back\n");

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

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

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

main.cc模块中新添加了两个Nachos文件系统命令：

**（a）nachos [-d f] –ap unix\_filename nachos\_filename**

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

**（b）nachos [-d f] –hap unix\_filename nachos\_filename**

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

上述两个命令调用了fstest.cc中的Append(…)函数。

**（c）nachos [-d f] –nap nachos\_filename1 nachos\_filename1**

该命令的功能是将一个nachos文件（unix\_filename1）附加到一个Nachos文件（nachos\_filename2）的后面，目的是用来测试当我们在一个Nachos的文件尾部写入数据时，文件大写是否会增加；

该命令调用了fstest.cc中的NAppend(…)函数。

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

**注1：你需要根据实验2的内容，**

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

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

**注2：采取分而治之的策略**

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

## 6.3 对新文件系统进行测试

该过程在../lab5命令下完成。

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

然后，按顺序运行下列Nachos文件系统命令：

nachos –cp test/small small //复制Linux文件small到Nachos文件系统中

nachos –ap test/small small //将Linux文件small附加到Nachos文件small的尾部

nachos –cp test/empty empty

nachos –ap test/empty empty

运行nachos –D，屏幕输出系统转储的信息，内容大致如下：

Bit map file header:

FileHeader contents. File size: 128. File blocks:

2

File contents:

\ff\7\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

Directory file header:

FileHeader contents. File size: 200. File blocks:

3 4

File contents:

\1\12\11@\5\0\0\0small\0\0\0\0G\5\8\1H\5\8\8\0\0\0empty\0\0\0\0\0\0\0\0G

\5\8\0\0\0\0\0\0\0\00\0\0\0\e0\12\11@\0\12\11@\10G\5\8X\13\0\0\d0\15\11@

\18\0\0\0\0\12\11@\c8\12\11@\80G\5\8\18\0\0\0‘\0\0\0\0\0\0\0\0\0\0\0\2\0

\0\0\a0F\5\8\0\0\0\0\0\0\0\0\18\0\0\0

\0\0\0\0\1\0\0\0\a8D\5\8\0\11\0\0\c8\15\11@H\0\0\0\f8\12\11@\f8\12\11@\0F

\5\8\0\0\0\0\0\0\0\00\0\0\0\e0\12\11@\0\12\11@\90D\5\8\c8\15\11@HH\5\8\18

\0\0\0

Bitmap set:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Directory contents:

Name: small, Sector: 5

FileHeader contents. File size: 168. File blocks:

6 7

File contents:

small file small file small file\asmall file small file small

file\a\*\*\*end of file\*\*\*\asmall file small file small file\asm

all file small file small file\a\*\*\*end of file\*\*\*\a

Name: empty, Sector: 8

FileHeader contents. File size: 162. File blocks:

9 10

File contents:

medium file medium file medium file\amedium file medium file med

ium file\amedium file medium file medium file\amedium file medi

um file medium file\a\*\*\*end of file\*\*\*\a

No threads ready or runnable, and no pending interrupts.

Assuming the program completed.

Machine halting!

Ticks: total 8490, idle 8000, system 490, user 0

Disk I/O: reads 16, writes 0

Console I/O: reads 0, writes 0

Paging: faults 0

Network I/O: packets received 0, sent 0

Cleaning up...

从上述系统转储信息中可以看出，

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

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

（3）文件small大小是168字节（占用2个扇区），它的i-node在第5号扇区中，数据块存储在第6、7号扇区中。

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

## 6.4 关于扩展文件的几点注记

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

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

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

可以看出，上述三个Nachos命令都涉及到对Nachos文件的扩展；

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

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

### 6.4.1 nachos –ap 与nachos –hap命令的实现

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

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

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

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

（1）修正上面提到的两个约束；

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

（3）如果原来文件的最后一个扇区的剩余空间无法容纳要扩展的数据，需要为这些数据分配新的扇区，则需要修该空闲块管理使用的位示图文件以及文件头三元组中的三个数据，并将它们适时写回到硬盘原来的扇区中；

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

需要修改的文件及函数：

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

（2）修改FileSystem类，添加空闲块位示图文件的硬盘读写操作；

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

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

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

**nachos –ap命令具体实现方案（仅供参考）：**

**1、修改OpenFile::WriteAt()**

**目的：允许从文件尾开始写数据；**

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

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

（3）对OpenFile::WriteAt()函数的修改

只需修改两个不能对文件进行扩展的两个约束即可，后续代码保持不变；

两个约束修改后的代码如下：

int fileLength = hdr->FileLength(); //第一次调用返回值是从硬盘读出的文件头中的值，

//后续的每次调用都是获取的我们重载的

//FileHeader::Allocate()中修改的值（numBytes）

if ((numBytes <= 0) || (position >= fileLength)) //约束1

return 0;

if ((position + numBytes) > fileLength) //约束2

//从position 开始写入numBytes个字节，fileLength是写之前的文件长度

numBytes = fileLength - position;

将第一个约束修改为：

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

return 0;

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

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

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

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

因此，将第二个约束修改形如：

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

{

int incrementBytes = (position + numBytes) – fileLength;

BitMap \*freeBitMap = fileSystem-> getBitMap(); //自己实现

hdr->Allocate(freeBitMap, fileLength, incrementBytes); //自己实现

fileSystem-> setBitMap(freeBitMap)； //自己实现

}

// OpenFile::WriteAt()中的后续代码不需修改，保持不变

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

**目的：从硬盘读取空闲块位示图文件，内容被修改后再将其写回硬盘；**

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

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

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

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

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

直观上，类FileSystem在其构造函数中，维护了两个一直处于打开状态的文件句柄OpenFile\* freeMapFile与OpenFile\* directoryFile;directoryFile，freeMapFile = new OpenFile(FreeMapSector)，File\* directoryFile;directoryFile = new OpenFile(DirectorySector)，一个是硬盘DISK上的位示图文件，一个DISK上的目录文件（参见./filesys/filesys.h与./filesys/filesys.cc），我们可以直接使用它们实现对DISK上位示图文件与目录文件的读与写操作（BitMap类中位示图的读写函数FetchFrom(OpenFile \*)与WriteBack(OpenFile \*)就是使用freeMapFile实现的），由于这两个OpenFile对象是FileSystem类的私有变量，因此需要在FileSystem类中定义实现getBitMap()与setBitMap()；

这两个函数的代码形如：

BitMap\* FileSystem::getBitMap() {

BitMap \*freeBitMap = new BitMap(numSector); // DISK上numSector总数

freeBitMap->FetchFrom(freeMapFile);

return freeBitMap;

}

void FileSystem::setBitMap(BitMap\* freeMap) {

freeMap->WriteBack(freeMapFile);

}

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

**目的：将修改后的文件头写回硬盘；**

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

OpenFile::OpenFile(int sector)

{

hdr = new FileHeader;

hdr->FetchFrom(sector);

seekPosition = 0;

}

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

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

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

OpenFile::OpenFile(int sector)

{

hdr = new FileHeader;

hdr->FetchFrom(sector);

seekPosition = 0;

hdrSector=sector;

}

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

void OpenFile::WriteBack() {

hdr-> WriteBack(hdrSector);

}

**4、修改FileHeader::Allocate()**

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

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

**（1）FileHeader构造函数**

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

FileHeader:: FileHeader()

{

numBytes=0; //文件大小

numSectors=0; //文件扇区数

for (int i=0;i<NumDirector;i++) // NumDirector=30：文件最多拥有的扇区数

dataSectors[i]=0; /文件扇区索引表/

}

**（2）FileHeader::Allocate()**

FileHeader::Allocate(BitMap \*freeMap, int fileSize)函数根据文件大小为文件分配所需的所有扇区块，并在位示图中标记所分配的扇区块，设置头文件三元组< numBytes, numSectors, dataSectors[30] >，我们可以重载FileHeader::Allocate(BitMap \*freeMap, int fileSize, int incrementBytes)，以根据要扩展的数据大小incrementBytes判断是否需要分配新的扇区块，并更新文件头三元组；

原FileHeader::Allocate()函数的代码如下：

Bool FileHeader::Allocate(BitMap \*freeMap, int fileSize)

{

numBytes = fileSize;

numSectors = divRoundUp(fileSize, SectorSize);

if (freeMap->NumClear() < numSectors)

return FALSE; // not enough space

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

dataSectors[i] = freeMap->Find();

return TRUE;

}

语句hdr->Allocate(freeBitMap, fileLength, **incrementBytes**)判断是否需要为写入数据分配新的扇区，如果需要就为其分配，并更新位示图及文件头三元组；重载后的FileHeader::Allocate()代码形如：

Bool FileHeader::Allocate(BitMap \*freeMap, int fileSize, int incrementBytes) {

if ( (fileSize==0) &&( incrementBytes>0) ) { //在一个空文件后追加数据

if (freeMap->NumClear() <1) { //至少需要一个扇区块

printf(“Insufficient Disk Space.\n”);

return false;

}

dataSectors[0] = freeMap-Find();

numSectors = 1;

}

numBytes=fileSize;

int offsetr= numBytes % SectorSize; //原文件最后一个扇区块数据偏移量

int newSectorBytes = **incrementBytes – (**SectorSize - offsetr **);**

//最后一个扇区块剩余空间足以容纳追加数据, 不需分配新的扇区块

if (newSectorBytes <= 0 ) {

numBytes = numBytes + **incrementBytes; /**/更新文件大小

return TRUE; }

int moreSectors = divRoundUp(newSectorBytes, SectorSize); //新加扇区块数

if (freeMap->NumClear() <1) { //至少需要一个扇区块

printf(“Insufficient Disk Space.\n”);

return false;

}

for ( int i = numSectors; i < numSectors + moreSectors; i++ )

dataSectors[0] = freeMap-Find(); //更新扇区块索引表

numBytes = numBytes + incrementBytes; //更新文件大小

numSectors = numSectors + moreSector; //更新文件扇区块数

return TRUE; }

}

注：应该在文件扩展结束后，将文件头写入硬盘

**4、修改fstest.cc的Append()**

**（1）修改写指针**

对fstest.cc中的Append()函数还要做少量修改， while{ … }循环中的去掉语句start += amountRead的注释，使每次写操作都是从上次写入的数据之后的位置开始进行（第一次是从start开始，默认是从文件尾或文件中间开始写入）；

**（2）将文件头写回硬盘**

fstest.cc中的Append()函数调用了我们修改后的OpenFile::WriteAt()函数，OpenFile::WriteAt()函数调用了函数FileHeader::Allocate()（重载），FileHeader::Allocate()根据每次写入的数据修改文件头三元组，但一直在内存中，尚未写回硬盘，因此在fstest.cc中的Append()函数中写操作结束后，应该调用OpenFile::WriteBack()将修改后的文件头写回到硬盘的相应的扇区中。

至此，对fstest.cc中的Append()函数的修改全部完成，可以利用nachos –ap 及nachos –hap 测试你的修改。

### 6.4.2 nachos –nap的实现

分析fstest.cc中的NAppend()函数可以看出，NAppend()函数也是调用OpenFile::Write()实现文件的写操作，我们也知道，OpenFile::Write()通过调用OpenFile::WriteAt()实现，我们已经修改了OpenFile::WriteAt()，实现了从文件的任何位置开始写入数据；

因此，你只需在fstest.cc中的NAppend()函数中与Append()做同样的修改即可：

去掉语句start += amountRead 及openFileTo->WriteBack();的注释；

### 6.4.3 nachos文件系统测试

**一、测试内容：**

1. Nachos文件系统的几个命令

nachos –f，nachos –D，nachos –r，nachos –cp，nachos –ap，nachos –hap，nachos –nap，nachos –l，nachos –p等（参见main.cc中关于文件的操作命令）；

2．Nachos文件系统的限制

最多可创建多少个文件，每个文件最大是多少字节等；

**二、测试过程：**

（1）为容易识别硬盘DISK信息的改变，将../file/test/下的small文件的内容修改为”This is a small file.”，

类似的，将../file/test/下的medium文件的内容修改为”This is a medium file.”，并设置多行；

将../file/test/下的big文件的内容修改为”This is a big file.”，并设置更多的多行；

为便于识别在DISK上的文件内容，你也可以在该目录下建立其它的文本文件，内容自定；

（2）nachos –f ，在硬盘DISK上初始化一个Nachos文件系统；

（3）nachos –D 及hexdump –C DISK考察Nachos在硬盘DISK上初始化的文件系统情况；

包括空闲块位示图的头文件（0号扇区）、空闲块位示图文件数据块（2号扇区）、目录表头文件（1号扇区）、目录表数据块（目录表）（3、4号扇区）

（4）nachos –cp test/small smal，复制test目录下的UNIX文件small到DISK中；

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

包括空闲块位示图的头文件（0号扇区）、空闲块位示图文件数据块（2号扇区）、目录表头文件（1号扇区）、目录表数据块（目录表）（3、4号扇区）、文件small的头文件、文件small的数据块等信息；

看看0~6号扇区有哪些改变；

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

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

查看系统是否将small文件的内容扩展到small文件原扇区的剩余空间中；

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

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

考察系统是否将samll文件原扇区的剩余空间写满后，为small分配新的扇区块，写入big的内容；

（10）nachos –ap test/medium medium，测试给一个空文件追加数据的功能；在Append()中如果DISK中不存在文件medium，将会自动创建一个空的medium文件，然后将test/medium文件内容追加到Nachos空文件medium中；

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

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

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

在执行（12）之前，由于small的数据块之后的扇区是medium文件的文件头及其数据块，因此，（12）执行后，系统会在medium文件之后为small分配新的扇区；

体现出small文件的数据块在硬盘上不是连续的，体现出文件数据块索引分配的特点；

（14）nachos –hap test/medium small，测试从small的中间写入文件的功能；

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

查看系统是否在small的中间写入文件test/medium的内容；

（16）nachos –nap medium small，测试将一个nachos文件附加到另一个nachos文件的功能；

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

查看系统是否将nachos文件medium附加到nachos文件small的尾部；

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

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

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

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

（21）反复运行nachos –ap test/big small，测试nachos文件系统中对一个文件长度的限制（一个文件最多可分配30个扇区，每个扇区128字节，因此文件最大限制为3KB）；

（22）反复运行nachos –ap 或nachos –cp在硬盘DISK上新建文件，测试nachos文件系统中最多可创建多少个文件（nachos采用一级目录，最多有10个目录项，因此对多可存储10个文件）；

（23）测试其它你想测试的内容，并利用nachos –D 及hexdump –C DISK查看结果是否是你的预期。

# 第7章 Nachos用户程序与系统调用（实验6）

## 7.1 目的与任务

该实验将体验Nachos的用户程序及Nachos系统调用的实现；

完成该实验后，你会理解：

（1）用户进程是如何启动的；

（2）用户进程是如何通过系统调用与操作系统内核进行交互的；

（3）系统调用是如何实现的；

理论上讲，实验6、7、8应该基于Nachos所实现的文件系统实现，系统调用所涉及到的文件操作（如create、open、read、write、close等，ecec与exit两个系统调用也涉及到这些文件操作）应该对模拟盘DISK上的文件进行操作；

Nachos实现的文件系统实现了两个版本，通过宏FILESYS\_STUB与FILESYS进行条件编译所产生的两个不同的实现（参见../filesys/filesys.h）； 宏FILESYS\_STUB实现的文件操作直接利用UNIX所提供的系统调用实现，操作的不是硬盘DISK上的文件；宏FILESYS实现的文件系统是通过OpenFile类对DISK上的文件进行操作（尽管最终也是使用UNIX的系统调用实现）；

考察../userprog/makefile与makefile.local的内容可以看出，实验6、7、8默认使用的是FILESYS\_STUB定义的相关实现，即不是对DISK上的文件进行操作，而是直接对UNIX文件进行操作；

由于时间关系，可以使用其默认设置。

## 7.2 背景知识

Nachos是一个操作系统，可以运行Nachos应用程序。Nachos的应用（用户）程序采用类C语言语法，通过Nachos提供的系统调用进行编写。详情可参阅../test目录下的几个.c文件。

由于Nachos模拟了一个执行MIPS指令的CPU，因此需要将用户编写的Nachos应用程序编译成MIPS框架的可执行程序。

Nachos提供了一个交叉编译程序gcc-2.8.1-mips.tar.gz，可将Nachos用户编写的应用程序编译成MIPS指令集的可执行程序，然后在Nachos中运行。其安装与使用方法请参见实验一。

gcc MIPS交叉编译器（gcc MIPS cross-compiler）将Nachos的应用程序编译成COFF格式的可执行文件，然后利用../test/coff2noff将COFF格式的可执行程序转换成Nachos CPU可识别的NOFF可执行程序。

**运行Nachos应用程序的方法：**

（1）在../test目录下运行make，将该目下的几个现有的Nachos应用程序（.c文件）交叉编译，并转换成Nachos可执行的.noff格式文件。

现有的几个应用程序：halt.c，matmult.c，shell.c，sort.c

（2）在../userprog目录下运行make编译生成Nachos系统，键入命令./nachos –x halt.noff可让Nachos运行应用程序halt.noff，参数-x的作用是Nachos运行其应用程序。

## 7.3 Things to Do

### 7.3.1 Nachos可执行程序（Nachos executables）

该实验中，你需要研究一下../test目录中的Makefile文件，理解Nachos生成Nachos可执行程序的过程。目录../test提供了4个示例程序（.c文件）。

可按下述步骤跟踪检查Nachos应用程序的启动与执行过程（以halt.c为例）：

（1）在./test目录中将文件halt.c的修改成：

#include “syscall.h”

int main()

{

int i,j,k;

k=3;

i=2;

j=i-1;

k=i-j+k;

Halt();

/\* not reached\*/

}

（2）在./test目录中重新编译生成新的halt

（3）在./test目录中可以通过下述命令生成halt.c对应的汇编代码文件halt.s；

/usr/local/mips/bin/decstation-ultrix-gcc –I ../userprog –I ../threads –S halt.c

（4）研究一下halt.s，看看函数main的栈帧（stack frame）是如何创建于撤销的，编译程序将C程序中的语句编译成什么样的机器指令。

（5）进入目录../userprog

（a）运行make编译Nachos的内核

（b）如果../test中指向../arch/unknown-i386-linux/bin/halt.noff文件的符号链接文件是halt，则通过命令nachos –x ../test/halt运行Nachos的应用程序halt

如果../test中指向../arch/unknown-i386-linux/bin/halt.noff文件的符号链接文件是halt.noff，则通过命令nachos –x ../test/halt.noff运行Nachos的应用程序halt

你可以加上参数 –d m输出显示Nachos模拟的MIPS CPU所执行的每条指令，即

nachos –d m –x ../test/halt.noff,

还可以再加上参数-s，以输出每条指令执行后对应寄存器的状态，如

nachos –d m –s –x ../test/halt.noff,

你也可以利用gdp跟踪、理解Nachos应用程序的启动与执行过程。

### 7.3.2 页表的系统转储

在后续的设计任务中，需要在Nachos中运行多道程序，需要理解用户进程的创建过程。

Nachos中的进程是通过形成一个地址空间从线程演化而来的。

Nachos的存储管理采用分页管理方式，可以在类AddrSpace中添加成员函数Print()，在为一个应用程序新建一个地址空间后调用该函数，输出该程序的页表（页面与帧的映射关系），显示信息有助于后续程序的调试与开发。

void AddrSpace::Print() {

printf("page table dump: %d pages in total\n", numPages);

printf("=============================\n"); printf("\tVirtPage, \tPhysPage\n");

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

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

}

printf("============================================\n\n");

}

如在../userprog/ progtest.cc的void StartProcess(char \*filename)中，当为一个应用程序新建一个空间后，调用Print()，输出页表信息。

void StartProcess(char \*filename)

{

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

AddrSpace \*space;

if (executable == NULL) {

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

return;

}

space = new AddrSpace(executable);

currentThread->space = space;

space->Print(); //输出该作业的页表信息

delete executable; // close file

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

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

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

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

// the address space exits

// by doing the syscall "exit"

}

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

### 7.3.3 分配更大的地址空间

我们的用户程序有时需要系统为其分配更大的地址空间，方法之一就是在程序中定义一个静态数组。例如在用户程序halt.c定义了一个大小为40个元素的静态整型数组，运行时系统为其分配相应的地址空间，大小为12个页面。

#include “syscall.h”

static int a[40];

int

main()

{

Halt();

.\*/ not reached \*/

}

## 7.4 几点注记

### 7.4.1 Nachos应用程序

Nachos的应用程序是作者自己定义的一种文件类型，文件头部分结构相对简单，编程方便。

分析../test/Makefile可以看出，首先利用交叉编译器提供的gcc、as、ld（注：不是通常的gcc等）等工具将nachos应用程序.c编译链接成,coff文件，然后利用../bin/arch/unknown-i386-linux/bin/coff2noff将coff文件转变成noff文件，简单的可执行文件头结构便于程序实现（参见AddrSpace:: AddrSpace()中将程序装入主存，并设置PC的值部分代码）。

NOFF文件头结构如下（参见../bin/arch/unknown-i386-linux/bin/noff.h）：

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

typedef struct segment {

int virtualAddr; /\* location of segment in virt addr space \*/ //progtam entry

int inFileAddr; /\* location of segment in this file \*/

int size; /\* size of segment \*/

} Segment;

typedef struct noffHeader {

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

Segment code; /\* executable code segment \*/

Segment initData; /\* initialized data segment \*/

Segment uninitData; /\* uninitialized data segment –

/\* should be zero'ed before use \*/

} NoffHeader;

具体内容可自己分析；可阅读../bin/arch/unknown-i386-linux/bin/coff2noff.cc详细理解COFF文件与NOFF文件的格式；

可以看出，文件头给出了每个段的大小，在文件中的开始位置，以及程序的入口地址等信息；

AddrSpace的构造函数在将NOFF文件装入内存之前，先打开该文件，读入文件头，然后根据文件头信息确定程序所占用的空间，将相应的段装入内存，将程序的入口地址，给PC赋值。

### 7.4.2 页表

将应用程序装入内存后，页表实现了虚页与实页（帧）的对应关系，系统根据页表实现存储保护，页面置换算法根据页表信息进行页面置换等操作；

Nachos使用的页表结构如下：（参见../machine/translate.h）

class TranslationEntry {

public:

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

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

// start of "mainMemory"

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

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

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

// to modify the contents of the page.

bool use; // This bit is set by the hardware every time the

// page is referenced or modified.

bool dirty; // This bit is set by the hardware every time the page is modified.

};

### 7.4.3 用户进程的创建与启动

Nachos的参数-x （nachos –x filename）调用../userprog/ progtest.cc的StartProcess(char \*filename)函数，为用户程序创建filename创建相应的进程，并启动该进程的执行。

系统要运行一个应用程序，需要为该程序创建一个用户进程，为程序分配内存空间，将用户程序（代码段与数据段，数据段包括初始化的全局变量与未初始化的全局变量，以及静态变量）装入所分配的内存空间，创建相应的页表，建立虚页与实页（帧）的映射关系；（参见AddressSpace:: AddressSpace()）；

然后将用户进程映射到一个核心线程；（参见StartProcess() in progtest.cc）

为使该核心线程能够执行用户进程的代码，需要核心在调度执行该线程时，根据用户进程的页表读取用户进程指令；因此需要将用户页表传递给核心的地址变换机构；

Instruction类封装了一条Nachos机器指令（参见Machine类）；

machine::ReadMem(registers[PCReg], 4, &raw))（在translate.cc中实现），基于Machine类维护的页表信息对虚实地址进行转换（Machine::Translate()， in ../machine/translate.cc），然后根据转换后的物理地址从内存中读出一条指令；

Machine::OneInstruction(Instruction \*instr)（../machine/mipssim.cc中实现），对通过machine::ReadMem(registers[PCReg], 4, &raw))读出的指令进行译码并根据指令规定的操作执行该条指令；

machine::Run() （./machine/mipssim.cc中实现）循环调用Machine::OneInstruction(Instruction \*instr)执行程序指令，直到程序退出或遇到一个异常；

因此，需要将用户进程的页表传递给Machine类维护的页表，才能执行用户程序指令；

AddrSpace::RestoreState() 将用户进程的页表传递给Machine类，该页表在为用户进程分配地址空间时创建（参见AddressSpace构造函数）；

为便于上下文切换时保存与恢复寄存器状态，Nachos设置了两组寄存器，一组是CPU使用的寄存器int registers[NumTotalRegs]（参见Machine类in Machine.h），用于保存执行完一条机器指令时该指令的执行状态；另一组是运行用户程序时使用的用户寄存器int userRegisters[NumTotalRegs]，用户保存执行完一条用户程序指令后的寄存器状态（参见Thread.h）；

由于CPU只有一个（系统初始化时创建了一个CPU），因此CPU寄存器也只有一套；而每个核心线程都可能执行用户程序（一个用户进程至少需要映射到一个核心线程），因此每个核心线程都需要维护一套用户寄存器userRegisters[]，因此Nachos为每个核心线程设置了一组用户寄存器，用于保存与恢复相应的用户程序指令的执行状态；

当用户进程进行上下文切换时（实质上是执行用户进程的核心线程发生上下文切换），将老进程的CPU的寄存器状态保存到用户寄存器userRegisters[]中，并将新用户进程的寄存器状态恢复到CPU的寄存器中，以便CPU能够继续执行上次被中断的用户程序；参见Scheduler::Run()中核心进程切换时对CPU寄存器与用户寄存器的保存与恢复；

考察progtest.cc中的函数StartProcess(char \*filename)的代码，分析系统为应用程序创建进程与启动进程的过程。

StartProcess(char \*filename) // filename：应用程序文件名，如../test/halt.noff

{

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

AddrSpace \*space;

if (executable == NULL) {

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

return;

}

space = new AddrSpace(executable);

currentThread->space = space;

delete executable; // close file

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

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

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

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

// the address space exits

// by doing the syscall "exit"

}

语句space = new AddrSpace(executable)为应用程序filename分配内存空间并将其装入所分配的内存空间中，然后建立页表，并建立虚页与实页（帧）的映射关系；

这时space就是该进程的标识，这里并没有像UNIX中为每个进程分配一个进程号（pid，Nachos中称为SpaceId），你自己可以为进程分配一个pid，并建立pid与space的映射关系，以后通过pid标识该进程；

语句currentThread->space = space将该进程映射到一个核心进程；对于第一个用户进程，映射到核心的主线程”main”，即currentThread指向主线程；

线程维护一个私有变量AddrSpace \*space，Thread类的构造函数中，设置space=NULL；当该线程与一个应用进程捆绑后，space指向系统为该进程所分配的内存空间，以便被调度时执行该进程所对应的应用程序；

语句space->InitRegisters()初始化CPU的寄存器，包括数据寄存器、PC以及栈指针等；

其中由于Nachos将应用程序的可执行文件转变为.noff格式时，将程序的入口地址设置为0，因此，应用进程从虚地址0开始执行，因此PC=0；

系统为应用程序栈分配了1KB的空间，将栈顶指针初始化为应用程序空间的尾部（向上生长），为防止发生偶然访问到程序地址空间外部，将栈指针从底部向上偏移了16字节（这是个实现问题）；

语句space->RestoreState()看似是恢复用户寄存器的内容，但这里只是将用户进程的页表传递给系统核心（Machine类），以便CPU能从用户进程的地址空间中读取应用程序指令；

CPU的寄存器状态被设置后，就开始用户进程的执行；Machine::Run()从程序入口开始，完成取指令、译码、执行的过程，直到进程遇到Exit()语句或者异常才退出；

对于一个真正的操作系统，理论上讲，不应该创建进程后立刻执行，而应该将进程放入就绪队列，等待进程的调度；

Nachos目前不支持多进程，

在应用程序执行过程中，如果与应用进程所关联的核心线程发生上下文切换，则引起用户进程的上下文切换；

### 7.4.4 PCB

核心线程不需要单独分配内存；利用Thread::Fork()创建的线程，只需调用Thread::StackAllocate()为其分配栈空间，在几个CPU寄存器中设置线程入口ThreadRoot()，线程的执行体，以及线程知悉结束时需要做的工作（线程结束时调用Thread::Finish()）；

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

理论上讲，系统应该首先为应用程序分配一个PCB，为其分配一个进程号（pid），进程号可以是PCB数组的索引号；

根据应用程序的文件头计算所需的实页数（帧数），根据内存的使用情况为其分配足够的空闲帧（不一定是连续的帧），将应用程序的代码与数据读入内存所分配的帧中，创建页表，建立虚页与实页的映射关系；

还要为应用程序分配栈与堆；

还应该在PCB中建立打开文件的列表，列表的索引即为文件描述符；

最后将pid、页表位置、栈位置及堆位置等信息记录在PCB中，在PCB中建立三个标准设备的映射关系，并记录进程与线程的映射关系，以及进程的上下文等；

目前Nachos实现的比较简单，没有体现出PCB的概念，只是将这些信息分散到相应的类对象中；例如利用所分配内存空间的对象指针标识一个进程，该对象中含有进程的页表、栈指针、与核心线程的映射等信息；

进程的上下文保存在核心线程中，当一个线程被调度执行后，依据线程所保存的进程上下文中执行所对应的用户进程；

同时，进程被创建后不应立即执行，应该将程序入口等记录到PCB中，一旦相应的核心线程引起调度，就从PCB中获取所需的信息执行该进程；

### 7.4.5 用户线程映射到核心线程

在实现系统调用Fork()之前，Nachos不支持用户多线程机制（可以认为只有一个线程，否则无法建立进程与核心线程的映射关系）；要求系统调用Fork()实现在同一个用户空间中创建多个用户线程。并建立用户线程与核心线程之间的映射，映射关系最简单的就是One to One模型；

### 7.4.6 线程调度算法

目前Nachos默认的线程调度算法是FCFS，当然，你可以利用Thread::Yield()模拟抢先机制；如果运行时加上-rs参数，nachos –rs random-seeds，可以实现轮转法（RR）线程调度，参见../thread/system.cc中Initilize()函数初始化Nachos内核时对参数-rs的处理。

# 第8章 地址空间的扩展（实验7）

## 8.1 目的与任务

该实验中，你将完成如下任务，为下一实验中实现系统调用Exec()与Exit()奠定基础：

（1）阅读理解类AddrSpace，然后对其进行修改，使Nachos能够支持多进程机制，允许Nachos同时运行多个用户进程；

（2）在类AddrSpace中添加完善Print()函数（在实验6中已经给出）

（3）阅读并理解../userprog/bitmap.h与../userprog/bitmap.cc，然后确定如何使用它们。

该实验与下一个实验（实验8）可在目录../lab7-8中完成，参照实验2介绍的方法将该实验中需要修改的模块、头文件，以及依赖这些头文件的模块复制到该目录中。

如将需要的模块从../userprog目录复制到该目录中，还要复制arch目录及其子目录、Makefile、Makefile.local等文件，并对Makefile及Makefile.local做相应的修改。

## 8.2 背景知识

假设我们希望在一个Nachos应用程序中通过系统调用Exec()装入并执行另一个Nachos应用程序../test/exec.noff，代码如下：

程序../test/bar.noff的代码如下：

#include “syscall.h”

int

main()

{

Exec(“../test/exec.noff”);

Halt();

}

其中../test/exec.noff程序的代码如下：

#include “syscall.h”

int

main()

{

Halt();

}

即当程序../test/bar.noff执行到语句 Exec(“../test/exec.noff”)时，系统需要将../test/exec.noff装入到内存，为其分配内存空间并执行它，遗憾的是目前Nachos无法实现上述功能。

考察下述AddSapce的构造方法AddrSpace::AddrSpace(OpenFile \*executable)中的程序片段：

// first, set up the translation

pageTable = new TranslationEntry[numPages];

for (i = 0; i < numPages; i++) {

pageTable[i].virtualPage = i; // for now, virtual page # = phys page #

pageTable[i].physicalPage = i;

pageTable[i].valid = TRUE;

pageTable[i].use = FALSE;

pageTable[i].dirty = FALSE;

pageTable[i].readOnly = FALSE; // if the code segment was entirely on

// a separate page, we could set its

// pages to be read-only

可以看出，目前实现的Nachos系统，当加载一个应用程序并为其分配内存时，总是将程序的第0号页面分配到内存的第0号帧，第1号页面分配到内存的第1号帧，…，以此类推。

因此，Nachos在运行其应用程序../test/bar.noff时，内存从的第0号开始的几个帧已经分配给程序../test/bar.noff，如果执行到语句 Exec(“../test/exec.noff”)加载../test/exec.noff时，Nachos仍然需要将从第0号开始的几个帧分配给../test/exec.noff，导致系统为../test/bar.noff所分配的内存空间被../test/exec.noff覆盖，程序显然无法正确执行。

## 8.3 Things to Do

（1）阅读理解类AddrSpace，然后对其进行修改，使Nachos能够支持多进程机制，允许Nachos同时运行多个用户线程；

（2）在类AddrSpace中添加完善Print()函数（在实验6中已经给出）

## 8.4 Bitmap Class

该实验的任务涉及到../userprog 目录下的bitmap类。请阅读并理解../userprog/bitmap.h与../userprog/bitmap.cc，然后确定如何使用它们。

# 第9章 系统调用Exec()与Exit()（实验8）

## 9.1 目的与任务

实现两个系统调用Exec()与Exit()。

其实，Nachos目前仅实现了系统调用Halt()，其实现代码参见../userprog/ exception.cc中的函数void ExceptionHandler(ExceptionType which)，其余的几个系统调用都没有实现。

Nachos系统调用对应的宏在../userprog/syscall.h中声明如下：

#define SC\_Halt 0

#define SC\_Exit 1

#define SC\_Exec 2

#define SC\_Join 3

#define SC\_Create 4

#define SC\_Open 5

#define SC\_Read 6

#define SC\_Write 7

#define SC\_Close 8

#define SC\_Fork 9

#define SC\_Yield 10

该实验的任务是在../userprog/ exception.cc中实现Exec()与Exit()的代码，如果时间允许，可以同时实现Read、Write、Join()，以便能够运行Nachos的shell（代码参见../test/shell.c）。

该实验与上一个实验（实验7）可在目录../lab7-8中完成，参照实验2介绍的方法将该实验中需要修改的模块、头文件，以及依赖这些头文件的模块复制到该目录中。

如将需要的模块从../userprog目录复制到该目录中，还要复制arch目录及其子目录、Makefile、Makefile.local等文件，并对Makefile及Makefile.local做相应的修改。

注：本实现也可以直接在../userprog目录下。

## 9.2 编写Nachos应用程序

我们需要编写一些Nachos应用程序，以测试Nachos相应的功能。

假如你自己需要编写一个Nachos应用程序exec.c用来测试你所实现的Nachos系统调用Exec()，可采用如下方法与步骤：

（1）按照类C语言的语法以及Nachos所提供的系统调用（注：无法使用Linux的系统调用），在目录../test中编写程序exec.c并保存到../test目录中。代码形如：

#include “syscall.h”

int main()

{

SpaceId pid;

pid=Exec(“../test/halt.noff”); //利用你所实现的Exec()执行../test/halt.noff

Halt();

/\* not reached \*/

}

（2）将exrc.c交叉编译并转换成Nachos模拟的CPU可执行程序exec.noff

（a）按如下方法将exec添加到../test/Makefile文件的targets列表中

………

# User programs. Add your own stuff here.

#

# Note: The convention is that there is exactly one .c file per target.

# The target is built by compiling the .c file and linking the

# corresponding .o with start.o. If you want to have more than

# one .c file per target, you will have to change stuff below.

targets = halt shell matmult sort exec

………..

（b）运行make，生成exec.noff

**命令make exec.s可生成exec.c对应的汇编代码exec.s**

## 9.3 设计与实现的有关问题

### 9.3.1 在哪里编写系统调用的代码

阅读../machine/machine.cc及mipssim.cc中的实现可以看出，每一条用户程序中的指令在虚拟机中被读取后，被包装成一个OneInstruction对象，然后在mipssim.cc中调用Machine::OneInstruction(Instruction \*instr)对其解码执行。

Machine::OneInstruction(Instruction \*instr)中有一个非常大的SWITCH语句，该语句分析所取出的指令类型执行这条指令（Nachos的指令类型见../machine/ mipssim.h）。

在mipssim.cc中Machine::OneInstruction(Instruction \*instr)的switch语句，对Nachos系统调用的处理方法是当Nachos的CPU检测到该条指令是执行一个Nachos的系统调用，则抛出一个异常SyscallException以便从用户态陷入到核心态去处理这个系统调用，代码如下：：

case OP\_SYSCALL:

RaiseException(SyscallException, 0);

return;

该异常SyscallException在../userprog/exception.cc中进行处理，代码如下：

void ExceptionHandler(ExceptionType which)

{

int type = machine->ReadRegister(2);

if ((which == SyscallException) && (type == SC\_Halt)) {

DEBUG('a', "Shutdown, initiated by user program.\n");

interrupt->Halt();

} else {

printf("Unexpected user mode exception %d %d\n", which, type);

ASSERT(FALSE);

}

}

从上述代码中可以看出，系统将系统调用号保存在MIPS的2号寄存器$2中，语句type = machine->ReadRegister(2)从寄存器$2中获取系统调用号，如果该条指令要调用0号系统调用（对应的宏是SC\_Halt），则执行Halt()系统调用的处理程序：

DEBUG('a', "Shutdown, initiated by user program.\n");

interrupt->Halt(); //停机

如果该条指令执行的是其它系统调用（Nachos提供的系统调用参见../userprog/syscall.h），则Nachos异常退出（ASSERT()调用来Abort()）。

这说明Nachos目前除了实现了系统调用Halt()，其它的系统调用均未实现。

因此你需要参照../userprog/exception.cc中对Halt()的处理方法编写其它系统调用的处理代码。

代码大致如下：

Void ExceptionHandler(ExceptionType which) {

int type = machine->ReadRegister(2);

if ((which == SyscallException)) {

switch(type){

case SC\_Halt:{

DEBUG('a', "Shutdown, initiated by user program.\n");

interrupt->Halt();

break;

}

case SC\_Exec:{

….

break;

}

//其它系统调用出来程序

default:{

printf("Unexpected syscall %d %d\n", which, type);

ASSERT(FALSE);

}

} // switch(type)

} else {

printf("Unexpected user mode exception %d %d\n", which, type);

ASSERT(FALSE);

}

} // ExceptionHandler(ExceptionType which)

注：Nachos在处理系统调用时，系统调用类型及相关参数的传递方式，是通过约定几个寄存器实现的。

### 9.3.2 Nachos系统调用机制

对于Nachos系统调用，在用户程序中的表现是调用../userprog/syscall.h中定义的一系列系统调用函数。这些函数的调用入口（System call stubs）参见../test/start.s。

/\* -------------------------------------------------------------

\* System call stubs:

\* Assembly language assist to make system calls to the Nachos kernel.

\* There is one stub per system call, that places the code for the

\* system call into register r2, and leaves the arguments to the

\* system call alone (in other words, arg1 is in r4, arg2 is

\* in r5, arg3 is in r6, arg4 is in r7)

\*

\* The return value is in r2. This follows the standard C calling

\* convention on the MIPS.

\* -------------------------------------------------------------

\*/

如系统调用Exec()的入口汇编代码如下：

.globl Exec ;全局变量

.ent Exec ;入口地址

Exec:

addiu $2,$0,SC\_Exec ; SC\_Exec+$0->$2

;其中S0=0, SC\_Exec:系统调用号，即$2存放系统调用号

syscall ;执行系统调用

j $31 ;从系统调用中返回到原程序中继续执行

;$31存放的是系统调用的返回地址

.end Exec

**系统调用的执行过程**：

先分析一下exec.c源代码与其对应的汇编代码：

exec.cc源代码（../test/exec.cc）：

#include “syscall.h”

int

main()

{

Exec(“../test/exec.noff”);

Halt();

}

exec.c对应的汇编代码：

.file 1 "exec.c"  
gcc2\_compiled.:  
\_\_gnu\_compiled\_c:  
 .rdata  
 .align 2  
$LC0:  
 .ascii "../test/halt.noff\000" //用户地址空间

.text  
 .align 2 //2字节对齐，即2\*2

.globl main //全局变量  
 .ent main //main函数入口  
main:

#汇编伪指令  frame 用来声明堆栈布局。  
#它有三个参数：  
# 1）第一个参数  framereg：声明用于访问局部堆栈的寄存器，一般为  $sp。  
# 2）第二个参数  framesize：申明该函数已分配堆栈的大小，应该符合  $sp ＋ #framesize ＝ 原来的  $sp。  
# 3）第三个参数  returnreg：这个寄存器用来保存返回地址。

#.frame：$fp为栈指针，该函数层栈大小为32字节，函数返回地址存放在$31

frame $fp,32,$31 # vars= 8, regs= 2/0, args= 16, extra= 0

mask 0xc0000000,-4

fmask 0x00000000,0

#栈采用向下生长的方式，即由大地址向小地址生长，栈指针指向栈的最小地址

subu $sp,$sp,32 # $sp - 32->$sp，构造mian()的栈frmae

#$sp的原值应该是执行main()之前的栈

#上一函数对应栈frmae的顶（最小地址处）

sw $31,28($sp) #$31->memory[$sp+28]，

sw $fp,24($sp) #$fp ->memory[$sp+24]

move $fp,$sp #$sp->$fp，执行Exec()会修改$sp

jal \_\_main #PC+4->$31, goto \_\_main

la $4,$LC0 #$LC0->$4 ,将Exec("../test/halt.noff\000")的参数的地址传给$4

#$4-$7：传递函数的前四个参数给子程序，不够的用堆栈

jal Exec #转到start.s中的Exec处执行

#/PC+4->$31, goto Exec；

#PC是调用函数时的指令地址，

#PC+4是函数的下条指令地址，以便从函数返回时从调用

#函数的下条指令开始继续执行原程序

sw $2,16($fp) #$2->memory[$fp+16]，Exec()的返回值

#$2,$3: 存放函数的返回值，当这两个寄存器不够存放

#返回值时，编译器通过内存来完成。

#$sp一直指向main()对应stack frame的栈顶（最小地址）

#由于在调用Exec()时要用到$sp，前面将$sp->$fp，

#因此$fp也是指向main()对应stack frame的栈顶

jal Halt #PC+4->$31, goto Halt，Halt()无参，无返回值

$L1:

move $sp,$fp #$fp->$sp

lw $31,28($sp) #memory[/$sp+28]->$31，取main()的返回值

lw $fp,24($sp) #memory[$sp+24]->$fp，恢复$fp

addu $sp,$sp,32 #$sp+32->$sp，释放main()对应的在栈中的frame

j $31 #goto $31, mian()函数返回

#$31: Return address for subrouting

.end main

汇编语句la $4,$LC0;**将Exec("../test/halt.noff")的参数（即要执行的文件名）的地址传给$4**，语句 jal Exec 转移到前面所述的start.s中的Exec处开始执行。start.s的Exec中**将系统调用号SC\_Exec（2号系统调用）保存到2号寄存器$2**，然后执行syscall。

从上述汇编代码中，你也可以了解系统对于栈的操作是如何进行的。

mipssim.cc中Machine::OneInstruction(Instruction \*instr)的switch语句检测到该条指令要调用系统调用（case OP\_SYSCALL:），则抛出异常SyscallException，见下述代码：

case OP\_SYSCALL:

RaiseException(SyscallException, 0);

return;

该异常SyscallException在../userprog/exception.cc中进行处理，异常处理程序如下：

Void ExceptionHandler(ExceptionType which)

{

int type = machine->ReadRegister(2);

if ((which == SyscallException) && (type == SC\_Halt)) {

DEBUG('a', "Shutdown, initiated by user program.\n");

interrupt->Halt();

} else {

printf("Unexpected user mode exception %d %d\n", which, type);

ASSERT(FALSE);

}

}

从2号寄存器$2中获取系统调用号（type = machine->ReadRegister(2)），然后应该根据系统调用号做相应的处理。

对于系统调用Exec(“../test/halt.noff”)，只携带了一个参数，我们可以从4号寄存器$4中获取参数在内存中的地址，然后读出该参数（文件名）执行它。

### 9.3.3 Nachos系统调用参数传递

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

一般参数传递有三种方式：

（a）通过寄存器；

（b）通过内存区域，将该内存区域的首地址存放在一个寄存器中；

（c）通过栈；

**在基于MIPS架构中，对于一般的函数调用，一般利用$4-$7（4到7号寄存器）传递函数的前四个参数给子程序，参数多于4个时，其余的利用堆栈进行传递；**

对于Nachos的系统调用，一般也是将要传递的参数依次保存到寄存器$4-$7中，然后根据这些寄存器中的地址从内存中读出相应的参数。

特别要注意的是字符串作为参数时的传递方式，这时寄存器中保存的是字符串在内存中的地址。

例如，如果第一个参数是字符串，则将该字符串在内存中的地址存入$5，如果第二个参数是数值，则将该值存入$5，以此类推。

对于Nachos的系统调用，一般也是将要传递的参数在内存中的地址依次保存到寄存器$4-$7中，然后根据这些寄存器中的地址从内存中读出相应的参数。

**如前面的exec.c对应的汇编代码中，在执行系统调用Exec()之前，利用指令la   $4,$LC0将Exec("../test/halt.noff”)中的参数"../test/halt.noff” 在内存中的地址$LC0传给$4，然后执行Exec，因此内核在处理系统调用Exec时，应该首先从$4中获取"../test/halt.noff”的内存地址，然后将参数从内存中读出。**

你自己可以考察一下下述程序的汇编代码：

#include “syscall.h”

int main()

{

Exit(1);

}

**系统调用Exit(1)对应的汇编代码为：**

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

**jal Exit #转到start.s中的Exit的调用入口**

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

成员函数Machine::ReadRegister(int num)可读取寄存器num中的内容，Machine::ReadMem(int addr, int size, int \*value)从内存addr处读取size个字节的内容存放到value所指向的单元中。

在Ecex()的实现代码中可以利用Machine::ReadRegister(4)从$4中获取"../test/halt.noff”的内存地址，然后利用Machine::ReadMem(…)获取FileName "../test/halt.noff”，然后为"../test/halt.noff”创建相应的进程及相应的核心线程，并将该进程映射到新建的核心线程上执行它。

当"../test/halt.noff”执行结束后，需要返回该线程的pid（对应Nachos中的SpaceId—Address Space Identifier，A unique identifier for an executing user program (address space)），可以利用Machine::ReadRegister(2)将线程"../test/halt.noff”对应的SpaceId写入2号寄存器中，MIPS架构将寄存器2和3存放返回值。

注：Machine::ReadRegister(int num)在Machine.cc中实现，Machine::ReadMem(int addr, int size, int \*value)在translate.cc中实现。

示例如下：

void ExceptionHandler(ExceptionType which)

{

int type = machine->ReadRegister(2);

if ((which == SyscallException)) {

switch(type){

case SC\_Halt:{

DEBUG('a', "Shutdown, initiated by user program.\n");

interrupt->Halt();

break;

}

case SC\_Exec:{

printf("Execute system call of Exec()\n");

//read argument

char filename[50];

int addr=machine->ReadRegister(4);

int i=0;

do{

//read filename from mainMemory

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

}while(filename[i++]!='\0');

printf("Exec(%s):\n",filename);

……

…..

//return spaceID

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

……

}

}

### 9.3.4 Openfile for the User program

Nachos启动时（主函数在main.cc中），通过语句 (void) Initialize(argc, argv)（参见system.cc）初始化了一个Nachos基本内核，其中通过Thread类（参见thread.cc）创建了一个Nachos的主线程”main”作为当前线程，并将其状态设为RUNNING（currentThread = new Thread("main"); currentThread->setStatus(RUNNING);），全局变量currentThread指向当前正在执行的线程（见system.cc）。

Nachos中只有第一个线程即主线程main是通过内核直接创建的，其它线程均需通过调用Thread::Fork(…)创建。（只有主线程main是一个特例）；

当通过命令nachos –x filename.noff加载运行Nachos应用程序filename.noff时，通过../userprog/ progtest.cc中的函数StartProcess(char \*filename)为该应用程序创建一个用户进程，分配相应的内存，建立用户进程（线程）与核心线程的映射关系，然后启动运行。

函数StartProcess(char \*filename)代码如下：

void StartProcess(char \*filename)

{

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

AddrSpace \*space;

if (executable == NULL) {

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

return;

}

space = new AddrSpace(executable);

currentThread->space = space; // currentThread是主线程”main”

delete executable; // close file

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

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

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

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

// the address space exits

// by doing the syscall "exit"

}

Nachos自己定义了一种可执行文件格式（.noff）文件，由.coff转换而来，通过命令../bin/coff2noff将coff文件转换成noff文件。与常用的COFF文件相比，NOFF文件格式相对简单，便于Nachos编程。

NOFF文件格式：（参见../bin/noff.h，../userprog/AddrSpace类中的SwapHeader()）

* 文件头
* 代码段
* 已初始化数据段
* 未初始化数据的

其中，文件头给出了文件的基本信息，如文件标识，各个段的位置等，由下述两个结构描述：（参见../bin/noff.h）

typedef struct noffHeader {

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

Segment code; /\* executable code segment \*/

Segment initData; /\* initialized data segment \*/

Segment uninitData; /\* uninitialized data segment --

\* should be zero'ed before use \*/

} NoffHeader;

typedef struct segment {

int virtualAddr; /\* location of segment in virt addr space \*/

int inFileAddr; /\* location of segment in this file \*/

int size; /\* size of segment \*/

} Segment;

NOFF文件的noffMagic=NOFFMAFIC=0xbadfad; 该Magic作为文件头部的一部分，位于文件头开始的两个字节，用于标识一种文件格式，也同时用户标识一种系统平台+文件格式。例如对于常用的COFF可执行文件，0x014c 相对于 I386 平台，0x268 相对于 Motorola 68000系列，0x170 for the PowerPC family of processors。

类AddrSpace的构造函数中，首先依据文件头中的Magic判定该文件是否为一个有效的NOFF文件（是否为0xbadfad），然后根据文件头中各段的大小（size）及需要的栈确定该程序分配需要的内存空间（帧数），如果内存大小满足要求，则为该程序创建相应的页表，将代码及数据等读入内存（分配帧），并在页表中设置页号与帧的映射关系（为简单起见，目前Nachos为应用程序分配到连续的帧中，即各页面对应的帧在内存中是顺序且是连续的）。参见AddrSpace类构造函数中代码：

pageTable = new TranslationEntry[numPages];

for (i = 0; i < numPages; i++) {

pageTable[i].virtualPage = i; // for now, virtual page # = phys page #

pageTable[i].physicalPage = i;

pageTable[i].valid = TRUE;

pageTable[i].use = FALSE;

pageTable[i].dirty = FALSE;

pageTable[i].readOnly = FALSE; // if the code segment was entirely on

// a separate page, we could set its

// pages to be read-only

从../userprog/ progtest.cc中的函数StartProcess(char \*filename)的代码中可以看出，要运行一个Nachos的应用程序，要为其创建相应的进程：首先要打开该文件（OpenFile \*executable = fileSystem->Open(filename);），为该其分配内存空间（space = new AddrSpace(executable);），该内存空间标识作为该进程的进程号。将该进程映射到一个核心线程，初始化寄存器，运行它。参见../userprog/ progtest.cc中的函数StartProcess(char \*filename)的下述代码：

currentThread->space = space; // currentThread是主线程”main”

delete executable; // close file

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

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

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

注1：进程的PCB比较简单，主要包括进程pid（SpaceID）、页表（代码、数据、栈）。

注2：由于Nachos加载用户程序为其分配的内存空间时，总是将该用户程序分配到从0号帧开始的连续区域中，因此目前Nachos不支持多进程机制。

因此在实现系统调用Exec(filename)时，需要为filename所对应的程序创建一个新的进程，并为其分配另外的内存空间（不能覆盖当前进程的地址空间）。

注3：用户线程与核心线程采用one-to-one线程映射模型。例如执行下述代码时：

#include “syscall.h”

int main()

{

SpaceId pid;

pid=Exec(“../test/halt.noff”); //利用你所实现的Exec()执行../test/halt.noff

Exit(0);

//Halt();

/\* not reached \*/

}

Nachos首先为主程序（main()）创建一个进程，分配内存空间（从0#帧开始），并将该进程映射到核心线程main，该线程是Nachos启动时创建的唯一的一个线程（见../threads/system.cc中void Initialize(int argc, char \*\*argv)）。

当执行pid=Exec(“../test/halt.noff”);时，需要为“../test/halt.noff”创建一个新的进程，为其分配与主进程不同的内存空间，同时要利用Thread::Fork(..)创建一个核心线程，然后将“../test/halt.noff”对应的进程映射到该核心线程中。（这里一个用户进程对应一个用户线程，目前不支持一个用户进程对应多个用户线程，除非你实现了系统调用Fork()。实现系统调用Fork()后，用户可以多次调用它以在用户空间中创建多个并发执行的用户线程）。

注4：文件../userprog/ progtest.cc中函数StartProcess(char \*filename)的参数filename是一个Nachos内核中的对象（string），而系统调用Exec(filename)中的参数filename是一个用户程序中的对象（string），前面给出的用户程序exec.c所对应的汇编代码说明了这一点：

### 9.3.5 Advance PC

../machine/ mipssim.cc中的成员函数void Machine::OneInstruction(Instruction \*instr)，从Nachos的内存中取出并执行一条Nachos应用程序的指令。其中有一个非常大的switch语句，是分析所取出的指令类型执行这条指令（Nachos的指令类型见../machine/ mipssim.h）。

其中下述case语句是对Nachos系统调用的处理，即当Nachos模拟的CPU检测到该条指令是执行一个Nachos的系统调用，则抛出一个异常SyscallException以便从用户态陷入到核心态去处理这个系统调用，代码如下：

// Execute the instruction (cf. Kane's book)

{

……..

switch (instr->opCode) {

case OP\_ADD:

…..

break;

……..

**case OP\_SYSCALL:**

**RaiseException(SyscallException, 0);**

**return;**

………

default:

ASSERT(FALSE);

} // switch (instr->opCode)

**// Now we have successfully executed the instruction.**

// Do any delayed load operation

DelayedLoad(nextLoadReg, nextLoadValue);

// Advance program counters.

registers[PrevPCReg] = registers[PCReg];

registers[PCReg] = registers[NextPCReg];

registers[NextPCReg] = pcAfter;

}

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

处理方法有两种，一种是将RaiseException(SyscallException, 0);后的return修改为break；二是在你的系统调用处理程序中添加PC的推进操作，否则，由于return造成的后果是没有对pc进行推进，程序就会再次读入执行这条系统调用操作，造成无限循环。

如果采用后者，可以在exception.cc模块中添加如下PC推进代码，并在你的各系统调用处理程序最后调用函数AdvancePC()，如：

void AdvancePC() {

machine->WriteRegister(PCReg, machine->ReadRegister(PCReg) + 4);

machine->WriteRegister(NextPCReg, machine->ReadRegister(NextPCReg) + 4);

}

void ExceptionHandler(ExceptionType which)

{

int type = machine->ReadRegister(2);

if ((which == SyscallException)) {

switch(type){

case SC\_Halt:{

DEBUG('a', "Shutdown, initiated by user program.\n");

interrupt->Halt();

break;

}

case SC\_Exec:{

printf("Execute system call of Exec()\n");

//read argument

char filename[50];

int addr=machine->ReadRegister(4);

int i=0;

do {

//read filename from mainMemory

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

} while(filename[i++]!='\0');

//printf("Exec(%s):\n",filename);

……

…..

//return spaceID

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

AdvancePC(); //PC增量指向下条指令

break;

} // case SC\_Exec:

// switch(type)

} // if ((which == SyscallException))

}

### 9.3.6 SpaceId

Exec()返回类型为SpaceId的值（参见../userporg/syscall.h中 typedef int SpaceId;，其实就是process id--pid），该值将来作为系统调用Join()的参数以识别新建的用户进程。

该值涉及的两个问题：

（1）它是如何产生的；

（2）在内核中如何记录它，以便Join()能够通过该值找到对应的线程；

解决方案：

（1）核心可支持多线程机制，系统初始化时创建了一个主线程main，以后可以利用Thread::Fork()创建多个核心线程，这些核心线程可以并发执行；

其实，nachos要求用户自己实现系统调用Fork()，以创建多个并发执行的用户线程，与系统调用Yield()联合使用。

目前在系统调用Folk()实现之前，系统只能运行一个用户线程，不支持多线程机制。系统调用Exec()可以在一个用户进程中加载另一个程序运行。

因此，进程的Pid即SpaceId可以这样产生：

从代码../userprog/protest.cc中的StartProcess()可以看出，加载运行一个应用程序的过程就是首先打开这个程序文件，为该程序分配一个新的内存空间，并将该程序装入到该空间中，然后为该进程映射到一个核心线程，根据文件的头部信息设置相应的寄存器运行该程序。

这里进程地址空间的首地址是唯一的，理论上可以利用该值识别该进程，但该值不连续，且值过大。

借鉴UNIX的思想，我们可以为一个地址空间或该地址空间对应的进程分配一个唯一的整数，例如0~99预留为核心进程使用（目前没有核心进程的概念，核心中只有线程），用户进程号从100开始使用。

目前Nachos没有实现子进程或子线程的概念（进程、线程之间没有建立进程树），因此，对于Nachos的第一个应用程序，其SpaceId即Pid=100，当该程序调用Exec(filename)加载运行filename指定的文件时，为filename对应的文件分配101，以此类推。

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

（2）Thread:: Fork(VoidFunctionPtr func, \_int arg)有两个参数，一个是线程要运行的代码，另一个是一个整数，可以考虑将用户进程映射到核心线程时，利用Fork()的第二个参数将进程的pid传入到核心中。

### 9.3.7 Join()

类似于Pthread中的pthread\_join(tid)，UNXI系统调用wait()的功能，int Join(SpaceId id)的功能是调用Join(SpaceId id)的程序等待用户进程id结束，当用户进程id结束后，Join()返回进程id的退出状态。

### 9.3.8 Exec()

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

int main() {

SpaceId pid;

pid=Exec(“../test/halt.noff”); //filemame=“../test/halt.noff”

Exit(0);

}

**设计与实现思路：**

**1、修改exception.cc，根据系统调用类型对各系统调用进行处理**；

(1)从2号寄存器中获取当前的系统调用号（type=machine->ReadRegister(2)），根据type对系统调用分别处理；

系统调用号参见../userprog/syscall.h，根据系统调用号对本次系统调用做相应的处理；

**(2)获取系统调用参数（寄存器4、5、6、7，可以携带4个参数）**

在Exec(char \*filename)的处理代码中，

(a)从第4号寄存器中获取Exec()的参数filename在内存中的地址（addr=machine->ReadRegister(4)）；

(b)利用Machine::ReadMem()从该地址读取应用程序文件名filename；

(c) 打开该应用程序（OpenFile \*executable = fileSystem->Open(filename)）；

(d)为其分配内存空间、创建页表、分配pid（space = new AddrSpace(executable)），至此为应用程序创建了一个进程；

注1：目前AddrSpace:: AddrSpace()中为应用程序分配内存时，都是分配了从0#帧开始的一个连续的内存空间，因此执行Exec(char \*filename)时，为应用程序filename所分配的内存空间不应该从0#帧开始，而应该从第一个空闲帧开始；

注2：目前AddrSpace:: AddrSpace()没有为进程分配pid；

注3：目前AddrSpace:: AddrSpace()没有对空闲帧进行管理

因此，需要修改AddrSpace:: AddrSpace()实现上述功能；同时，需要修改AddrSpace:: ~AddrSpace()，进程退出时释放pid，为其所分配的帧也应释放（修改空闲帧位示图）；

**因此需要修改AddrSpace:: AddrSpace()与AddrSpace:: ~AddrSpace()；**

(e)创建一个核心线程，并将该进程与新建的核心线程关联（thread = new Thread(forkedThreadName)，thread->Fork(StartProcess, space->getSpaceID())）；

需要特别指出的是，通过Thread::Fork()创建的线程需要指明该线程要执行的代码（函数）及函数所需的参数；

**我们可以重载函数StartProcess(int spaceId)，作为新建线程执行的代码，**并将进程的pid传递给系统，供其它系统调用（如Join()）使用；

当调度到该线程时，就启动应用程序进程的执行；

**2、修改progtest.cc，重载函数StatProcess(char \*filename)**

将该函数作为应用程序进程所关联的核心线程的执行代码，当调度到该线程时，Exec(filename)中filename所对应的应用程序进程随即执行；

void StartProcess(int spaceId)

{

currentThread->space = space;

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

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

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

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

// the address space exits

// by doing the syscall "exit"

}

**3、修改AddrSpace::AddrSpace()及AddrSpace::~AddrSpace()**

为管理空闲帧，建立一个全局的空闲帧管理位示图；

为管理pid（spaceId），建立一个全局的pid数组；

从内存的第一个空闲帧为Exec(filename)中的filename分配内存空间，创建该进程页表，建立虚实页表的映射关系，分配pid；

在释放应用程序内存空间时，应该清除空闲帧位示图相应的标志，释放pid，释放页表。

AddrSpace::AddrSpace()及AddrSpace::~AddrSpace()的代码参考如下：

BitMap \*bitmap; //for free frame

bool ThreadMap[MAX\_USERPROCESSES]; //pid or SpaceId

AddrSpace::AddrSpace(OpenFile \*executable)

{

//----added by yourself-------------------------------------------

//allow up to MAX\_USERPROCESSES user processes executables concurrently

//spaceID, i.e. pid

for(int i = 0; i < MAX\_USERPROCESSES; i++) {

if(!ThreadMap[i]){

ThreadMap[i] = true;

spaceID=i+100; //may be should reserved 0-99 for kernel Process,

//even though there is no any process at present

break;

} else {

printf(“Too many processes in Nachos!\n”);

return;

}

} //for

if(bitmap == NULL) //used for free frames

bitmap = new BitMap(NumPhysPages);

**//the remaining code**

//set up a new PageTable for a process, first, set up the translation

pageTable = new TranslationEntry[numPages];

for (i = 0; i < numPages; i++) {

pageTable[i].virtualPage = i; // virtual page #

pageTable[i].physicalPage = bitmap->Find(); // find a free frame

ASSERT(pageTable[i].physicalPage!=-1);

pageTable[i].valid = TRUE;

pageTable[i].use = FALSE;

pageTable[i].dirty = FALSE;

pageTable[i].readOnly = FALSE; // if the code segment was entirely on

// a separate page, we could set its

// pages to be read-only

}

**//the remaining code**

}

AddrSpace::~AddrSpace()

{

ThreadMap[spaceID-100] = 0; //false

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

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

}

delete [] pageTable;

}

**Exec()系统调用小结：**

（1）获取Exec(char \*filename)的参数filename；通过读取$4寄存器获得filename在内存中的地址（文件名在栈中的地址），然后从该地址中读出参数filename；

（2）打开应用程序filename；（参考progtest.cc函数StatProcess()）

（3）为应用程序及栈分配内存空间并为其建立页表，读入应用程序代码及数据。

但目前的实现是fstest.cc的StatProcess()每加载一个应用程序，都是从0#帧开始为其分配内存。因此需要记录目前内存中从几号帧开始是空闲的（为内存帧建立一个位示图bitmap对象，用于管理空闲帧）。因此你需要修改AddrSpace的构造函数以满足上述要求；

AddrSpace的构造函数中，首先根据文件头的信息确定文件大小，根据文件大小及栈大小确定需要为其分配多少个帧，如果剩余的内存空间大小满足该文件的要求，则为其创建页表，然后将文件内容读入为其所分配的帧中。如果目前内存不足，给出相应的提示后退出。

注：文件内容在内存中的顺序：先是代码段，然后是已初始化的数据段，未初始化的数据的没有读入主存，应该是存储在用户栈中。

**还应该在AddrSpace的构造函数中产生应用程序的SpaceId，即我们熟知的pid。pid可以从0开始，也可以从100开始，将0~99预留给核心进程（尽管目前没有核心进程的概念）。可以定义一个全局的spaceId（pid数组）；**

（4）创建一个核心线程，将该应用程序映射到该核心线程。

系统将main()对应的主程序映射到核心线程的主线程”main”，让主线程执行主程序。（参见progtest.cc中StatProcess()）。

对于应用程序filename，需要我们自己调用Thread::Folk()创建一个核心线程，并将该应用程序映射到该核心线程，以执行应用程序filename。

（5）初始化寄存器，开始程序执行

将所有设计寄存器请0。

由于读入程序时，从代码段的入口地址（虚地址）开始读入到内存的0号地址中（即第0帧的0号偏移量处），因此程序从逻辑地址0开始执行（物理地址也为0），将PC设置为程序的逻辑地址入口0。

../machine/translate.cc中machine::readmem()调用translate()将PC中的逻辑地址转换为物理地址，然后从该物理地址中将指令读出。

（6）返回进程的pid

（7）PC增量

实现代码形如：

case SC\_Exec:{

char filename[128];

int addr=machine->ReadRegister(4);

int i=0;

do{

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

}while(filename[i++]!='\0');

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

if (executable == NULL) {

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

return;

}

//new address space

space = new AddrSpace(executable);

delete executable; // close file

/new and fork thread

char \*forkedThreadName=filename;

thread = new Thread(forkedThreadName);

thread->Fork(StartProcess, space->getSpaceID());

currentThread->Yield();

//return spaceID

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

AdvancePC();

break;

}

### 9.3.9 Exit() and Exit Status

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

关于系统调用Exit()的实现，首先从4号寄存器读出退出码，然后释放该进程的内存空间及其其表，释放分配给该进程的实页（帧），释放其pid（参见AddrSpace::~AddrSpace()），调用currentThread->Finish结束该进程对应的线程。

管理空闲帧的位示图以及pid结构不能释放，因为它们是全局的。

实现代码形如：

case SC\_Exit:{

int ExitStatus=machine->ReadRegister(4);

machine->WriteRegister(2,ExitStatus);

delete currentThread->space;

currentThread->setExitStatus(ExitStatus);

currentThread->Finish();

AdvancePC();

/\* not reached \*/

break;

}

### 9.3.10 Things to Do

基于Lab7的工作，完成系统调用Exec()及Exit()。

如果有时间可以实现Join()、涉及用户线程以及文件操作的几个系统调用。

## 9.4 关于文件的几个系统调用

尽管目前设计暂不要求改部分内容，如有时间可以实现。

目前这些涉及文件的系统调用是基于FILESYS\_STUB定义的方法实现的；

理论上应该基于FILESYS而不是基于FILESYS\_STUB定义的方法实现。

尽管基于的方法不同，但在体现系统调用的设计与实现机制上，原理是相同的，不必过于深究。

### 9.4.1 Create()

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

//void Create(char \*name);

case SC\_Create: {

int base=machine->ReadRegister(4);

int value;

int count=0;

char \*FileName= new char[128];

do{

machine->ReadMem(base+count,1,&value);

FileName[count]=\*(char\*)&value;

count++;

} while(\*(char\*)&value!='\0'&&count<128);

int fileDescriptor = OpenForWrite(FileName);

if (fileDescriptor == -1)

printf("create file %s failed!\n",FileName);

else

printf("create file %s succeed!, the file id is %d\n",FileName,fileDescriptor);

Close(fileDescriptor);

//machine->WriteRegister(2,fileDescriptor);

AdvancePC();

break;

}

### 9.4.2 Open()

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

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

\*/

//OpenFileId Open(char \*name); //int OpenFileId

case SC\_Open: {

int base=machine->ReadRegister(4);

int value;

int count=0;

char \*FileName= new char[128];

do{

machine->ReadMem(base+count,1,&value);

FileName[count]=\*(char\*)&value;

count++;

} while(\*(char\*)&value!='\0'&&count<128);

int fileDescriptor = OpenForReadWrite(FileName, FALSE);

if (fileDescriptor == -1)

printf("Open file %s failed!\n",FileName);

else

printf("Open file %s succeed!, the file id is %d\n",FileName,fileDescriptor);

machine->WriteRegister(2,fileDescriptor);

AdvancePC();

break;

}

### 9.4.3 Write()

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

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

case SC\_Write: {

int base =machine->ReadRegister(4); //buffer

int size=machine->ReadRegister(5); //bytes written to file

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

int value;

int count=0;

// printf("base=%d, size=%d, fileId=%d \n",base,size,fileId );

OpenFile\* openfile =new OpenFile (fileId);

ASSERT(openfile != NULL);

char\* buffer= new char[128];

do{

machine->ReadMem(base+count,1,&value);

buffer[count] = \*(char\*)&value;

count++;

} while((\*(char\*)&value!='\0') && (count<size));

buffer[size]='\0';

int WritePosition;

if (fileId==1)

WritePosition = 0;

else

WritePosition = openfile->Length();

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

if((writtenBytes)==0)

printf("write file failed!\n");

else

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

//machine->WriteRegister(2,size);

AdvancePC();

break;

}

### 9.4.4 Read()

/\* Read "size" bytes from the open file into "buffer".

\* Return the number of bytes actually read -- if the open file isn't

\* long enough, or if it is an I/O device, and there aren't enough

\* characters to read, return whatever is available (for I/O devices,

\* you should always wait until you can return at least one character).

\*/

//int Read(char \*buffer, int size, OpenFileId id);

case SC\_Read: {

int base =machine->ReadRegister(4);

int size = machine->ReadRegister(5);

int fileId=machine->ReadRegister(6);

OpenFile\* openfile=new OpenFile(fileId);

char buffer[size];

int readnum=0;

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

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

if(!machine->WriteMem(base,1,buffer[i]))

printf("This is something wrong.\n");

buffer[size]='\0';

printf("read succeed! the content is \"%s\" , the length is %d\n",buffer,size);

machine->WriteRegister(2,readnum);

AdvancePC();

break;

}

### 9.4.5 Close ()

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

//void Close(OpenFileId id);

case SC\_Close: {

int fileId =machine->ReadRegister(4);

//printf("SC\_Close: fileId in $4 = %d\n",fileId);

//void Close(int fd) in sysdep.cc

//OpenFile\* openfile=new OpenFile(fileId);

//delete openfile; //does not work well

Close(fileId);

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

AdvancePC();

break;

}

# 参考资料

[1] Peiyi Tang, Ron Addie, nachos\_introduction.pdf，University of Southern Queensland, 2002

[2] Peiyi Tang, Ron Addie, nachos\_study\_book.pdf，University of Southern Queensland, 2002

[3] Nachos-3.4（C++）源代码

[4] a road map through nachos.pdf

[5] gdb使用指南