



# 操作系统课程设计报告

题目： Nachos 分析与设计

计算机科学与技术学院

2019 级 1 班

XXXXXXXXXXXX XXX

2022 年 5 月 10 日

## 摘要

本课程设计报告的编写目的是为了全面地说明和分析本学期《操作系统课程设计》的情况和成果。操作系统课设以 Nachos 为例，Nachos 的全称是“Not Another Completely Heuristic Operating System”，它是一个可修改和跟踪的操作系统教学软件。它给出了一个支持多线程和虚拟存储的操作系统骨架，可在较短的时间内对操作系统中的基本原理和核心算法有一个全面和完整的了解。

本课设从 Nachos 的配置部署开始，涉及内核的线程管理、内存管理、系统调用、用户进程、文件系统等。课设报告从实验目的和任务出发，进行代码的阅读和内核原理的分析，然后做出抽象的设计架构和具体的代码实现，接着对实现的功能进行较为完备的测试，穿插调试过程以及问题解决方法，最后总结每个实验的工作，进一步反思归纳。

课程设计一共分 8 个子实验：实验一主要是 Nachos 安装配置，并且利用 GDB 调试理解 Nachos 工作原理，这是后续实验的基础；实验二是了解 Makefile 的编写，方便后续在工作目录下链接其他模块；实验三是在 Nachos 的信号量基础上模拟生产者消费者问题，进行线程同步控制；实验四与五是关于 Nachos 的文件系统的分析与改进，在分析并且深入理解文件系统不同模块之间的组织构建关系的基础上，对 Append 等命令进行完善，并且采用带模拟指针的多叉树数据结构来重新组织目录表，实现目录树以及扩展多级目录；实验六主要对 Nachos 的用户程序进行分析，理解交叉编译器工作原理、Nachos 可执行文件结构、页表设计与实现、Machine 类配合 Instruction 类来实现机器和指令的模拟等，了解用户进程的创建和执行过程、核心线程的映射原理，了解 Nachos 系统调用，为实验七和八奠定基础；实验七主要是扩展用户进程的地址空间使得 Nachos 能够支持多进程机制，完成伪进程号 SpaceId 的管理，并且初步完成了系统调用 Exec 的实现；实验八，在前面实验 5、6、7 的基础上，实现线程相关的 Join 等五个系统调用，理解系统调用的实现、参数传递、回传机制，此后分别基于 Linux 文件系统和 Nachos 文件系统实现两套文件相关的系统调用，对 Linux 和 Nachos 文件系统进行控制，最后基于实现的部分系统调用，实现了一个简易的 Nachos Shell，支持 echo、touch 等命令。

在该实验报告中，您将看到我在解决各个子实验时的方案，设计过程中的思路，相关操作系统的总结和面向对象设计模式的使用，以及最重要的我对整个操作系统课设的思考和收获。

**关键词：**Nachos、线程管理、系统调用、用户进程、文件系统

## 目录

摘要 .....	2
实验 1 Nachos 系统的安装与调试 .....	7
实验信息 .....	7
实验目的 .....	7
实验任务 .....	7
实验步骤 .....	8
安装 Nachos 与 gcc MIPs 交叉编译器 .....	8
设计与实现 .....	8
测试结果 .....	9
测试 Nachos .....	9
分析输出结果：考察 Nachos 的启动与退出过程 .....	10
其他模块测试 .....	12
利用 gdb 调试 Nachos C++代码的过程与方法 .....	12
结论体会 .....	18
课后作业 .....	18
实验收获 .....	21
实验 2 Nachos 的 Makefiles .....	22
实验信息 .....	22
实验目的与任务 .....	22
代码与原理分析 .....	22
测试结果 .....	25
结论体会 .....	27
实验 3 利用信号量实现线程同步 .....	28
实验信息 .....	28
实验目的 .....	28
实验任务 .....	28
代码与原理分析 .....	28
设计与实现 .....	31
测试结果 .....	35

---

结论体会 .....	38
实验 4 Nachos 的文件系统 .....	40
实验信息 .....	40
实验目的 .....	40
实验任务 .....	40
代码与原理分析 .....	41
设计与实现 .....	46
测试结果 .....	46
支持文件系统的 Nachos .....	46
Nachos 的硬盘以及文件系统 .....	48
Nachos 提供的三个测试文件 .....	51
Nachos 文件系统在硬盘上的布局 .....	56
结论体会 .....	57
实验 5 扩展 Nachos 的文件系统 .....	59
实验信息 .....	59
实验目的 .....	59
实验任务 .....	59
代码与原理分析 .....	60
阅读代码 .....	60
设计与实现 .....	63
Nachos -(h/n)ap 指令的实现 .....	63
CatalogTree 目录树的设计思想 .....	66
测试结果 .....	76
测试文件系统的命令 .....	76
测试文件系统的限制 .....	79
测试 CataLogTree .....	81
结论体会 .....	83
实验 6 Nachos 用户程序与系统调用 .....	84
实验信息 .....	84
实验目的 .....	84
实验任务 .....	84

---

代码与原理分析 .....	85
Nachos 应用程序与加载 .....	85
Nachos 分页管理方式 .....	87
应用程序进程的创建与启动 .....	90
Nachos 地址空间分配 .....	91
Nachos 核心线程执行用户程序的原理和方法 .....	93
Nachos 用户程序的创建与启动过程 .....	97
Nachos 进程控制块 PCB.....	97
Nachos 线程调度算法 .....	98
部分源码 .....	98
测试结果 .....	101
测试 Nachos 应用程序与执行过程 .....	101
测试 Nachos 分页 .....	104
结论体会 .....	105
实验问题回答 .....	105
实验 7 地址空间的拓展 .....	107
实验信息 .....	107
实验目的 .....	107
实验任务 .....	107
代码与原理分析 .....	108
Nachos 只支持单进程原因 .....	108
Nachos 系统调用流程 .....	108
Nachos 系统调用入口 .....	109
Nachos 系统调参数传递 .....	111
AdvancePC .....	111
SpaceID .....	112
Pid.....	112
设计与实现 .....	113
地址空间的拓展 .....	113
Exec 系统调用实现 .....	115
测试结果 .....	117

---

测试 Exec 以及地址空间拓展 .....	117
结论体会 .....	118
实验 8 系统调用 Exec() 与 Exit()	119
实验信息 .....	119
实验目的 .....	119
实验任务 .....	119
代码与原理分析 .....	119
编写自己的 Nachos 应用程序 .....	119
Nachos 系统调用参数传递 .....	120
用户进程的 Openfile .....	120
Advance PC .....	121
Pid 用户进程标识 .....	122
设计与实现 .....	122
Join() 系统调用的实现 .....	122
Exec() 系统调用的实现 .....	126
Exit() 系统调用的实现 .....	127
Yield() 系统调用的实现 .....	128
基于 FILESYS_STUB 实现文件相关系统调用 .....	129
基于 FILESYS 实现文件相关系统调用 .....	133
Nachos Shell 实现 .....	138
测试结果 .....	142
Join 测试 .....	142
Exit 测试 .....	143
Yield 测试 .....	145
线程综合测试 .....	146
FILE_STUB 系统调用综合测试 .....	147
FILESYS 系统调用综合测试 .....	149
Nachos Shell 测试 .....	153
结论体会 .....	154
参考文献 .....	156

# 实验 1 Nachos 系统的安装与调试

## 实验信息

日期: 2022.3.1

姓名: xxx

班级 19.1

学号: XXXXXXXXXXXX

## 实验目的

- (1) 安装编译 Nachos 系统，理解 Nachos 系统的组织结构与安装过程；
- (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) 安装成功后，根据 Nachos 的输出结果，分析分析跟踪 Nachos 的 C++程序及汇编代码，理解 Nachos 中线程的创建方法以及上下文切换的过程。
- (5) 阅读 Nachos 的相关源代码，理解 Nachos 内核的启动与停机过程。
- (6) 理解 Nachos 的运行参数的含义与使用。

## 实验任务

- (1) 安装 Linux 操作系统；
- (2) 安装 Nachos 及 gcc mips 交叉编译程序；
- (3) 编译测试 Nacho，并理解 Nachos 的运行参数的含义与使用；
- (4) 运行 Nachos，根据 Nachos 的输出，理解 Nachos 中第一个线程是如何产生的。理解并掌握 Nachos 中其它内核线程的创建方法；理解 idle 线程的创建与作用。  
进而理解一个实际的操作系统（如 Windows、Linux 等）的第一个进程是如何产生的，以及 ideler 进程的创建与使用。
- (5) 理解 Nachos 中的上下文切换过程；
- (6) 熟悉 gdb 调试工具；

# 实验步骤

## 安装 Nachos 与 gcc MIPS 交叉编译器

包括解压 Nachos-3.4.tar.gz 以及 gcc MIPS 交叉编译器的安装，同样 copy file “gcc-2.8.1-mips.tar.gz” to file fold “/usr/local” with cp command. 然后进行解压，即把 gcc 交叉编译器加入到了环境中。

**为什么 gcc-2.8.1-mips.tar.gz 需要安装到/usr/local 中？**

因为：在 Makefile.dep 中 GCCDIR 的目录就是 /usr/local 下图是 Liunx 下的 Makefile 程序

```

57
58 # 386, 386BSD Unix, or NetBSD Unix (available via anon ftp
59 #      from agate.berkeley.edu)
60 ifeq ($(uname),Linux)
61 HOST_LINUX=-linux
62 HOST = -DHOST_i386 -DHOST_LINUX
63 CPP=/lib/cpp
64 CPPFLAGS = $(INCDIR) -D HOST_i386 -D HOST_LINUX
65 arch = unknown-i386-linux
66 ifdef MAKEFILE_TEST
67 #GCCDIR = /usr/local/nachos/bin/decstation-ultrix-
68 GCCDIR = /usr/local/mips/bin/decstation-ultrix-
69 LDFLAGS = -T script -N
70 ASFLAGS = -mips2
71 endif
72 endif
73

```

然后可以进行相关测试了 见 测试结果

## 设计与实现

主要需要利用 gdb 的相关命令，进行调试

常用的 gdb 命令：

<b>gdb ./nachos</b>	# 打开可执行程序 nachos 开始调试
<b>gdb list</b>	# 显示代码
<b>gdb p currentThread</b>	# 查看主线程对象的地址
<b>gdb b function</b>	# 显示断点号及函数地址
<b>gdb b num</b>	# List 之后在具体行数上设置断点
<b>gdb b *SWITCH+44</b>	# 设置在 SWITCH 函数汇编代码的第 44 行
<b>gdb info b</b>	# 断点查看
<b>gdb run</b>	# 程序重新运行
<b>gdb c</b>	# 继续运行

```
gdb n # 单步运行
```

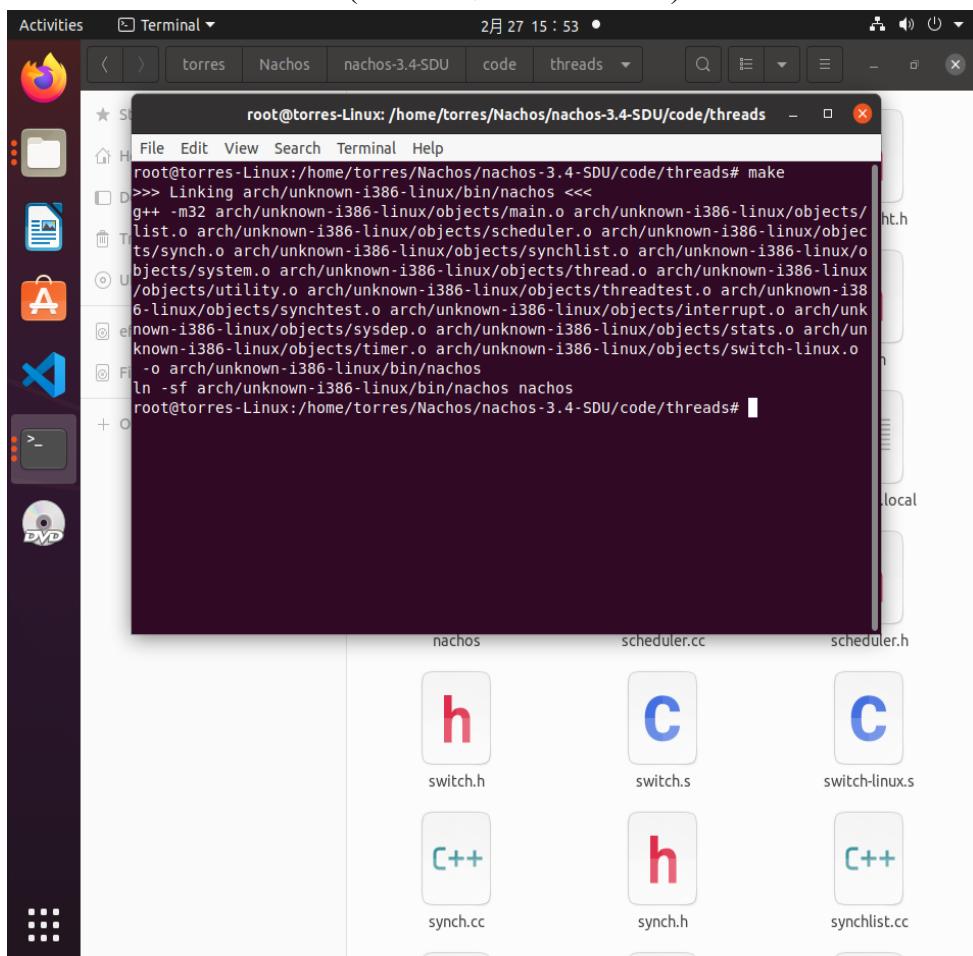
以及需要部分汇编知识

```
%ebp      # 栈底指针, 基地址  
%esp      # 栈顶指针
```

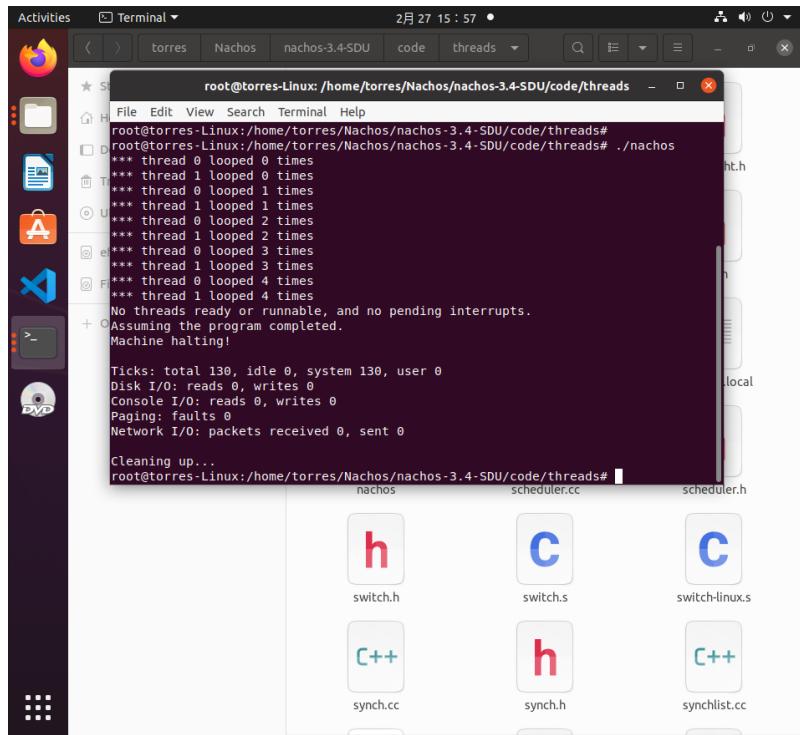
## 测试结果

### 测试 Nachos

在 code/thread 下执行 make (可能先需要 make clean)



在该目录下执行 ./nachos 命令，输出结果如下图所示。



## 分析输出结果：考察 Nachos 的启动与退出过程

### 1. 初始化 Nachos 的设备与内核

首先 main.cc 调用了

**(void) Initialize(argc, argv); //system.cc**

关键代码

```

DebugInit(debugArgs);           // initialize DEBUG messages
stats = new Statistics();      // collect statistics
interrupt = new Interrupt();   // start up interrupt handling
scheduler = new Scheduler();    // initialize the ready queue
if (randomYield)               // start the timer (if needed)
    timer = new Timer(TimerInterruptHandler, 0, randomYield);

threadToBeDestroyed = NULL;

// We didn't explicitly allocate the current thread we are running in
// But if it ever tries to give up the CPU, we better have a Thread
// object to save its state.
currentThread = new Thread("main");
currentThread->setStatus(RUNNING);

interrupt->Enable();
CallOnUserAbort(Cleanup);      // if user hits ctrl-C

```

创建了主线程 main、设置了线程状态 并且进行了断控制器，定时器、CPU、硬盘等初始化。

**讨论： main 线程创建后，后续的线程是如何创建的？**

是通过 main 线程 fork 创建子进程来实现的。

## 2. 对 Nachos 内核进行测试（线程的创建及并发执行）

在初始化结束之后，首先读入参数进行处理，然后进行 threadTest() 函数进行 simpleThreadTest() 而此函数的执行结果就是 ./nachos 的执行结果

Main 线程 fork 了一个子线程 SimpleThread，然后 main 线程执行

SimpleThread(0); 子线程执行 SimpleThread(1); 于是就是实现了上述的交替输出的结果。

## 3. 测试 Nachos 的锁机制及条件变量

运行 SynchTest() 可以看到如下的输出结果，是一个车过桥的同步测试问题。

```

Activities Terminal 3月 2 08:21
torres@torres-Linux: ~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/thread$ 
Direction [1], Car [1], Crossing...
Direction [1], Car [2], Crossing...
Direction [1], Car [0], Exiting...
Direction [1], Car [1], Exiting...
Direction [1], Car [2], Exiting...
Direction [0], Car [0], Arriving...
Direction [0], Car [1], Arriving...
Direction [0], Car [2], Arriving...
Direction [1], Car [3], Crossing...
Direction [1], Car [4], Crossing...
Direction [1], Car [5], Crossing...
Direction [1], Car [3], Exiting...
Direction [1], Car [4], Exiting...
Direction [1], Car [5], Exiting...
Direction [0], Car [3], Arriving...
Direction [0], Car [4], Arriving...
Direction [0], Car [5], Arriving...
Direction [1], Car [6], Crossing...
Direction [1], Car [6], Exiting...
Direction [0], Car [6], Arriving...
Direction [0], Car [0], Crossing...
Direction [0], Car [1], Crossing...
Direction [0], Car [2], Crossing...
Direction [0], Car [3], Crossing...
Direction [0], Car [4], Crossing...
Direction [0], Car [5], Crossing...
Direction [0], Car [3], Exiting...
Direction [0], Car [4], Exiting...
Direction [0], Car [5], Exiting...
Direction [0], Car [6], Crossing...
Direction [0], Car [6], Exiting...
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 3680, idle 0, system 3680, 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
torres@torres-Linux: ~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/thread$ 

```

## 4. 终止主线程，Nachos 退出

main() 函数退出之前，执行 currentThread->Finish() 终止主线程，

Thread::Finish() 调用了 Thread::Sleep()，因为需要首先回收子线程，让子线程执行完毕，然后最后再让 main 线程结束

当没有线程在执行程序时，会循环调用 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。  
这里和真实的操作系统比较相似。

## 其他模块测试

### 1. code/monitor 的测试

Monitor 实现了 nachos 使用的锁、条件变量、信号量、管程的等同步机制。

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/monitor$ ./nachos
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 590, idle 0, system 590, 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...
```

### 2. code/filesys 测试 nachos 的文件系统

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/filesys$ ./nachos
*** 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 1190, idle 1000, system 190, user 0
Disk I/O: reads 2, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

## 利用 gdb 调试 Nachos C++代码的过程与方法

可以借助于调试工具 gdb，来在终端对 nachos 代码进行分析或者对程序进行调试

### 1. gdb 命令 list 来查看代码

```
(gdb) list
73    //      (optionally) Call test procedure
74    //
75    //      "argc" is the number of command line arguments (including the name
76    //          of the command) -- ex: "nachos -d +" -> argc = 3
77    //      "argv" is an array of strings, one for each command line argument
78    //          ex: "nachos -d +" -> argv = {"nachos", "-d", "+"}
79    //-----
80
81    int
82    main(int argc, char **argv)
```

## 2. gdb 查看函数地址

```
(gdb) b InterruptEnable()
Breakpoint 1 at 0x3027: file thread.cc, line 242.
(gdb) b SimpleThread(int)
Breakpoint 2 at 0x3275: file threadtest.cc, line 26.
(gdb) b ThreadFinish()
Function "ThreadFinish()" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) b ThreadFinish()
Breakpoint 3 at 0x2ffc: file thread.cc, line 241.
(gdb) b ThreadRoot
Breakpoint 4 at 0x4e7c
(gdb) ■
```

## 3. gdb 查看给定线程的地址

```
(gdb) disass SimpleThread(int)
Dump of assembler code for function SimpleThread(int):
0x00003275 <+0>:    endbr32
0x00003279 <+4>:    push  %ebp
0x0000327a <+5>:    mov   %esp,%ebp
0x0000327c <+7>:    push  %ebx
0x0000327d <+8>:    sub   $0x14,%esp
0x00003280 <+11>:   call  0x12f0 <_x86.get_pc_thunk.bx>
0x00003285 <+16>:   add   $0x5ccb,%ebx
0x0000328b <+22>:   movl $0x0,-0xc(%ebp)
0x00003292 <+29>:   cmpl $0x4,-0xc(%ebp)
0x00003296 <+33>:   jg    0x32ca <SimpleThread(int)+85>
0x00003298 <+35>:   sub   $0x4,%esp
0x0000329b <+38>:   pushl -0xc(%ebp)
0x0000329e <+41>:   pushl 0x8(%ebp)
0x000032a1 <+44>:   lea    -0x3c90(%ebx),%eax
0x000032a7 <+50>:   push  %eax
0x000032a8 <+51>:   call  0x11c0 <printf@plt>
0x000032ad <+56>:   add   $0x10,%esp
0x000032b0 <+59>:   lea    0xf4(%ebx),%eax
0x000032b6 <+65>:   mov   (%eax),%eax
0x000032b8 <+67>:   sub   $0xc,%esp
0x000032bb <+70>:   push  %eax
0x000032bc <+71>:   call  0x2dbc <Thread::Yield()>
0x000032c1 <+76>:   add   $0x10,%esp
0x000032c4 <+79>:   addl $0x1,-0xc(%ebp)
0x000032c8 <+83>:   jmp   0x3292 <SimpleThread(int)+29>
0x000032ca <+85>:   nop
```

根据汇编代码，可以发现之前查看的 simplethread() 地址为 0x3275 也是正确的。而在 Initialize() 函数中，语句 currentThread = new Thread(“main” ) 创建了 Nachos 的主线程 “main”，并通过 current->setStatus(RUNNING) 将其状态设为就绪。因此要查看主线程的地址，需要 currentThread->setStatus(RUNNING) 上设置断点。

```
(gdb) s
44 }
(gdb) s
Initialize (argc=0, argv=0xffffcf28) at system.cc:150
150     currentThread->setStatus(RUNNING);
(gdb) s
Thread::setStatus (this=0x56563ca0, st=RUNNING) at thread.h:102
102     void setStatus(ThreadStatus st) { status = st; }
(gdb) p currentThread
$6 = (Thread *) 0x56563ca0
(gdb) display
(gdb) p *currentThread
$7 = {stackTop = 0x0, machineState = {0 <repeats 18 times>}, stack = 0x0,
      status = JUST_CREATED, name = 0x5655a1d6 "main"}
(gdb) 
```

然后查看子线程的地址

```
ThreadTest () at threadtest.cc:43
43 {
(gdb) l
38 //      to call SimpleThread, and then calling SimpleThread ourselves.
39 //-----
40
41 void
42 ThreadTest()
43 {
44     DEBUG('t', "Entering SimpleTest");
45
46     Thread *t = new Thread("forked thread");
47
(gdb) l
48     t->Fork(SimpleThread, 1);
49     SimpleThread(0);
50 }
51
(gdb) b 48
Breakpoint 2 at 0x56558327: file threadtest.cc, line 48.
(gdb) c
Continuing.

Breakpoint 2, ThreadTest () at threadtest.cc:48
48     t->Fork(SimpleThread, 1);
(gdb) n
49     SimpleThread(0);
(gdb) p t
$1 = (Thread *) 0x56563d00
(gdb) 
```

4. 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？

而 switch 函数的执行过程是

simpleThread()

thread->Yield()

nextThread = scheduler->FindNextToRun();

scheduler->Run(nextThread)

SWITCH(oldThread, nextThread);

然后调用汇编的 SWITCH 函数

所以

```
Breakpoint 5, Scheduler::Run (this=0x56563c80, nextThread=0x56563ca0)
  at scheduler.cc:116
116      SWITCH(oldThread, nextThread);
(gdb)
```

第一次进行 switch 的时候 可以发现:

oldThread 的地址 0x56563c80

nextThread 的地址 0x56563d00

对应于之前查看的线程地址 不难发现 旧线程是 main 线程 而 新线程是 fork 的子线程

```
Breakpoint 5, Scheduler::Run (this=0x56563c80, nextThread=0x56563d00)
  at scheduler.cc:116
116      SWITCH(oldThread, nextThread);
(gdb) info r
eax          0x0          0
ecx          0xc          12
edx          0x9          9
ebx          0x5655df40    1448468288
esp          0xfffffc80    0xfffffc80
ebp          0xfffffc98    0xfffffc98
esi          0x5655a2e3    1448452835
edi          0x56563d00    1448492288
eip          0x56556a18    0x56556a18 <Scheduler::Run(Thread*)+130>
eflags        0x282        [ SF IF ]
cs           0x23         35
ss           0x2b         43
ds           0x2b         43
es           0x2b         43
fs           0x0          0
gs           0x63         99
(gdb)
```

而进一步阅读 发现

```

2      movl    _EAX(%eax),%ebx          # get new value for eax into ebx
3      movl    %ebx,_eax_save          # save it
4      movl    _EBX(%eax),%ebx          # restore old registers
5      movl    _ECX(%eax),%ecx
6      movl    _EDX(%eax),%edx
7      movl    _ESI(%eax),%esi
8      movl    _EDI(%eax),%edi
9      movl    _EBP(%eax),%ebp
10     movl    _ESP(%eax),%esp          # restore stack pointer
11     movl    _PC(%eax),%eax          # restore return address into eax
12     movl    %eax,4(%esp)           # copy over the ret address on the stack
13     movl    %eax,0(%esp)
14     #This is a bug, the offset for return address should be 0, not 4.
15     # -- ptang, Sep/1/03, at UALR
16
17     movl    _eax_save,%eax
18
19     ret
20
21 #endif
22
23

```

当执行完 `movl %eax,4(%esp)` 寄存器 `eax` 中内容即为原线程 (`t1, oldThread`) 的地址

进行进一步调试

```
Breakpoint 1, 0x56559e86 in SWITCH ()
```

分别单步执行汇编程序 最后执行完 `ret` 到了 `ThreadRoot` 的汇编指令位置

```

0x56559eb2 in SWITCH ()
(gdb) ni
0x56559eb6 in SWITCH ()
(gdb) ni
0x56559eb9 in SWITCH ()
(gdb) ni
0x56559ebf in SWITCH ()
(gdb) x/i 0x56559ebf
=> 0x56559ebf <SWITCH+57>:    mov    0x8(%eax),%ebx
(gdb) print/x $ebx
$4 = 0x0
(gdb) print/x $eax
$5 = 0x56563d00
(gdb) stepi 8
0x56559ed6 in SWITCH ()
(gdb) x/i 0x56559ed6
=> 0x56559ed6 <SWITCH+80>:    mov    %eax,(%esp)
(gdb) ni
0x56559ed9 in SWITCH ()
(gdb) x/i 0x56559ed9
=> 0x56559ed9 <SWITCH+83>:    mov    0x5655e054,%eax
(gdb) ni
0x56559ede in SWITCH ()
(gdb) x/i 0x56559ede
=> 0x56559ede <SWITCH+88>:    ret
(gdb) print/x $ret
$6 = 0x0
(gdb) ni
0x56559e78 in ThreadRoot ()
(gdb) ni
0x56559e79 in ThreadRoot ()
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/torres/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/threads/nachos
*** thread 0 looped 0 times

Breakpoint 1, 0x56559e86 in SWITCH ()
(gdb) stepi 24
0x56559ed6 in SWITCH ()
(gdb) x/i 0x56559ed6
=> 0x56559ed6 <SWITCH+80>:    mov    %eax,(%esp)
(gdb) print/x $eax
$7 = 0x56559e78
(gdb) 
```

在执行到最后

```

        movl    _ESP(%eax),%esp      # restore stack pointer
        movl    PC(%eax),%eax       # restore return address into eax
#       movl    %eax,4(%esp)       # copy over the ret address on the stack
        movl    %eax,0(%esp)
#This is a bug, the offset for return address should be 0, not 4.
# -- ptang, Sep/1/03, at UALR

```

Print/x \$eax 查看发现 eax 就是 ThreadRoot 的入口地址!!!!!

进一步思考 ThreadRoot 干了什么呢？

```

/* void ThreadRoot( void )
*/
** expects the following registers to be initialized:
**     eax      points to startup function (interrupt enable)
**     edx      contains initial argument to thread function
**     esi      points to thread function
**     edi      point to Thread::Finish()
*/
ThreadRoot:
    pushl  %ebp
    movl   %esp,%ebp
    pushl  InitialArg
    call   StartupPC
    call   InitialPC
    call   WhenDonePC

    # NOT REACHED
    movl   %ebp,%esp
    popl   %ebp
    ret

```

关键的就是 call StartupPC 以及 Initial PC ，而在 fork 过程中，存在

StackAllocate() 时设置了

设置一些寄存器，初始化，用于后期 switch 之后的 ThreadRoot

```

machineState[PCState] = (_int) ThreadRoot;
machineState[StartupPCState] = (_int) InterruptEnable;
machineState[InitialPCState] = (_int) func;
machineState[InitialArgState] = arg;
machineState[WhenDonePCState] = (_int) ThreadFinish;

```

关键在于设置 machineState[InitialPCState] = (\_int) func;

而这个 func 就是

t->Fork(SimpleThread, 1); -> StackAllocate(func, arg); ->

machineState[InitialPCState] = (\_int) func;

所以当我们进行 switch 完成之后，是通过 ThreadRoot 来进行程序计数器的定位以及新的程序的执行的，在本实验中也就实现了两个进程交替执行同一个函数的线程调度!!!! 下面继续执行 输出也验证了这个想法！

```
(gdb) c
Continuing.
*** thread 1 looped 0 times
```

第二次进程切换的时候：

```
Breakpoint 1, 0x56559e86 in SWITCH ()
(gdb) stepi 24
0x56559ed6 in SWITCH ()
(gdb) x/i 0x56559ed6
=> 0x56559ed6 <SWITCH+80>:      mov    %eax,(%esp)
(gdb) print/x $eax
$8 = 0x56556a26
(gdb) ni
0x56559ed9 in SWITCH ()
(gdb) ni
0x56559ede in SWITCH ()
(gdb) ni
0x56556a26 in Scheduler::Run (this=0x56563c80, nextThread=0x56563d00) at scheduler.cc:116
116      SWITCH(oldThread, nextThread);
(gdb) ni
118      DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
(gdb) ni
0x56556a2f      118      DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());
(gdb) ni
```

Eax 是 SWITCH() 的返回地址 0x56556a26 也就是  
同时就对应了 原来主线程中断的地方 就是 SWITCH(oldThread,nextThread)  
也就是说 回到了之前主线程 SWITCH 的地方

```
SWITCH(oldThread, nextThread);

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

在进一步思考 按理说应该是执行 SWITCH(OLD,NEW)的下一条啊 为什么还是  
SWITCH 呢？

个人认为可以把 switch 进行切换子线程看作函数调用，执行了一个大函数，ni  
单步执行，会进入子线程，而当子线程 switch 完，才会完成最开始主进程的  
ni，所以还是会回到 switch 而不是 下一句的 Debug!!!!

## 结论体会

### 课后作业

#### 1. Nachos 的启动过程

- (1) 调用..//threads/system.cc 中的 Initialize(argc, argv) 初始化内核
  - (a) 处理核初始化内使用的一些命令行参数，如-d, -rs, -f, -s 等；
  - (b) 初始化系统统计数据，如寻道时间、系统开始计时等；
  - stats = new Statistics();
  - (c) 初始化中断控制器；
  - interrupt = new Interrupt;

(d) 初始化调度程序，创建就绪队列；

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

(e) 如果运行时携带参数`-rs`，初始化定时器，实现时间片抢先调度

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

(f) 创建主线程`main`，并作为当前运行的线程；

```
currentThread = new Thread("main");
```

```
currentThread->setStatus(RUNNING);
```

(g) 允许响应中断 `interrupt->Enable()`；

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

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

(i) 对于实验 6、7、8，初始化 CPU；

```
machine = new Machine(debugUserProg);
```

(j) 对于实验 4、5，初始化硬盘与文件系统；

```
synchDisk = new SynchDisk("DISK");
```

```
#ifdef FILESYS_NEEDED
```

```
    fileSystem = new FileSystem(format);
```

```
#endif
```

(2) 如果需要，运行线程测试程序与同步测试程序；

```
ThreadTest(); // 测试线程创建及多线程并发交替执行过程
// 这里输出屏幕的运行结果
```

```
#if 0
```

```
    SynchTest(); // 测试锁机制及条件变量
```

```
#endif
```

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

(4) 终止主线程

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

## 2. Nachos 的命令行参数及其处理

```
Usage: nachos -d <debugflags> -rs <random seed #>
//      -s -x <nachos file> -c <consoleIn> <consoleOut>
//      -f -cp <unix file> <nachos file>
//      -p <nachos file>
//      -r <nachos file>
//      -l
//      -D
//      -t
```

```

//      -n <network reliability> -e <network orderability>
//      -m <machine id>
//      -o <other machine id>
//      -z
//
//      -d causes certain debugging messages to be printed (cf. utility.h)
//      -rs causes Yield to occur at random (but repeatable) spots
//      -z prints the copyright message
//
// USER_PROGRAM
//      -s causes user programs to be executed in single-step mode
//      -x runs a user program
//      -c tests the console
//
// FILESYS
//      -f causes the physical disk to be formatted
//      -cp copies a file from UNIX to Nachos
//      -p prints a Nachos file to stdout
//      -r removes a Nachos file from the file system
//      -l lists the contents of the Nachos directory
//      -D prints the contents of the entire file system
//      -t tests the performance of the Nachos file system
//
// NETWORK
//      -n sets the network reliability
//      -e sets the network orderability
//      -m sets this machine's host id (needed for the network)
//      -o runs a simple test of the Nachos network software

```

### 3. 主线程（main）是如何创建的？

在启动 Nachos 初始化其内核时（参见.../threads/system.cc 中 Initialize(...)），Nachos 创建了它的第一个线程--主线程 main，该线程应该是 Nachos 所有其它线程的“始祖”

### 4. 如何创建线程；

通过 main 线程上 fork 或者 main 的子线程进行 fork，来创建线程

### 5. Nachos 是如何进行上下文切换的；

在分时系统下，需要正在执行的线程进行 Yield() 然后进行线程调度，找到下一个运行的线程，对这两个线程进行 SWITCH，使用汇编进行 SWITCH 是因

为要修改寄存器的值，而 switch 完，使用 ThreadRoot 来保存一些线程状态，然后返回新线程之前执行到的上下文或者是指定的函数入口，继续执行

## 实验收获

1. 遇到问题：报错 因为 gcc 版本过高

解决：更换 gcc 版本 或者 把一些报错代码换成最新的语法

2. 关于 SWITCH，再次总结一下上述实验就是：在本实验中，当 main 进行 fork 时，如果是分时系统，则会在创建子线程的 fork 里面的 stackallocate 里面分配好和 ThreadRoot 有关的状态信息，然后主线程获得时间片，执行 SimpleThread(0)，直到 yield()，然后进行 SWITCH，从 main 切换到别的线程。SWITCH 执行完 ret 时 ThreadRoot 的地址，本实验中也就是 fork 的子线程的 ThreadRoot 的地址，进一步查看，发现 call func 的 func 就是执行 SimpleThread(1) 函数；但是在 fork 的子线程切换到 main 线程的时候，ret 不返回 ThreadRoot，而是直接返回 main 线程上次执行到的上下文 SWITCH 语句中，继续执行。
3. 整体思路大致明确的情况下，最好使用 gdb 调试来查看某些地址和运行过程，才能验证自己的想法！

# 实验 2 Nachos 的 Makefiles

## 实验信息

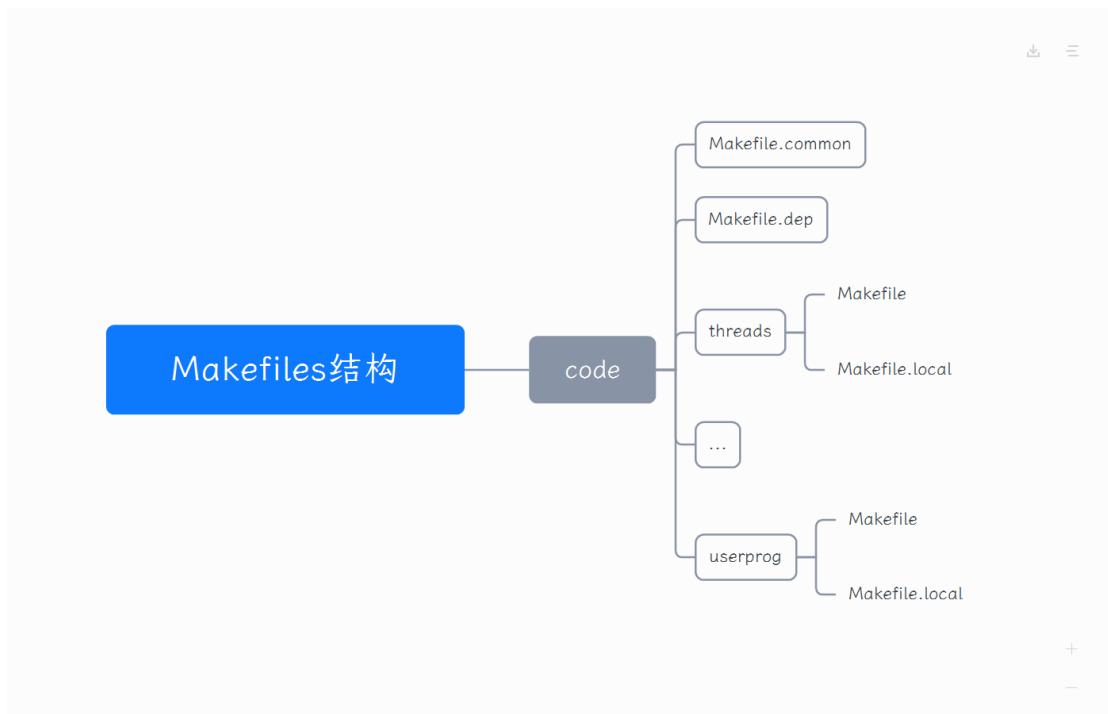
日期: 2022.3.8 姓名: XXX 学号: XXXXXXXXXX 班级: 19.1

## 实验目的与任务

- (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;

## 代码与原理分析

### 1. Makefiles 结构



Makefile 主要用于对 Nachos 系统进行编译和连接，整体结构是有一个大的 Makefile 和 Makefile.local 然后剩下的每个系统里面有一个单独的 makefile 和 makefile.local，之所以这样是因为可以针对不同的需求，编译链接形成不同的 nachos 内核。

## 2. Code 子目录下的 Makefile

只有 include 了本目录下的 makefile.local 以及父目录的 common makefile

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

### 3. Makefile.local

该文件主要是对一些编译、链接及运行时所使用的宏进行定义。

```
# CCFILES 用来指定本目录下的生成Nachos所需要到的C++源文件  
CCFILES = main.cc\  
        list.cc\  
        scheduler.cc\  
        synch.cc\  
        synclist.cc\  
        system.cc\  
        thread.cc\  
        utility.cc\  
        threadtest.cc\  
        synctest.cc\  
        interrupt.cc\  
        sysdep.cc\  
        stats.cc\  
        timer.cc  
  
# INC PATH 指明所涉及的C++源程序中的头文件(.h文件)所在的路径  
INCPATH += -I../threads -I../machine  
# 传递g++一些标号或者宏  
DEFINES += -DTHREADS  
  
endif # MAKEFILE_THREADS_LOCAL
```

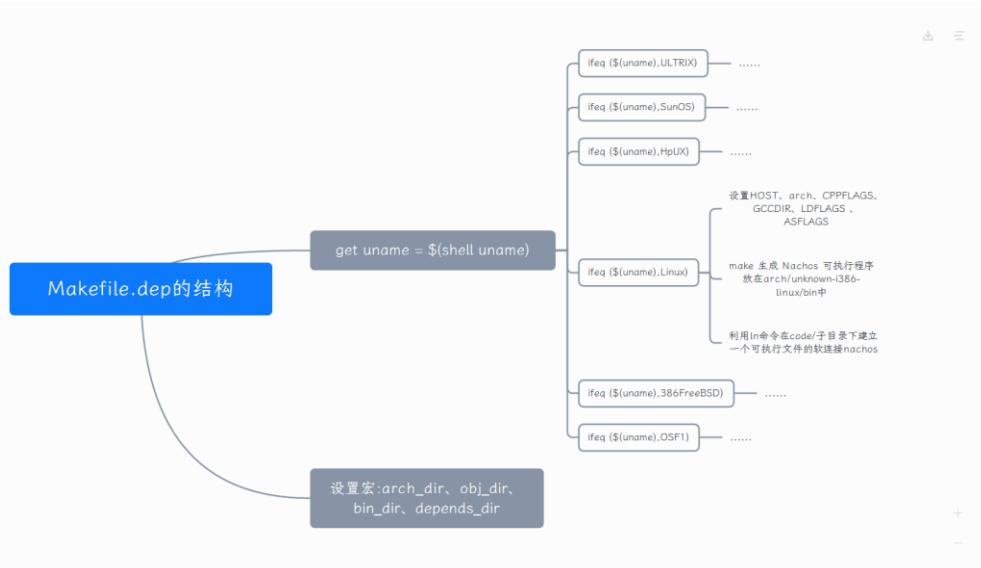
#### 4. Makefile.dep 文件

根据安装 Nachos 不同的操作系统环境，进行相应宏的定义等。

实现操作系统的宏定义：

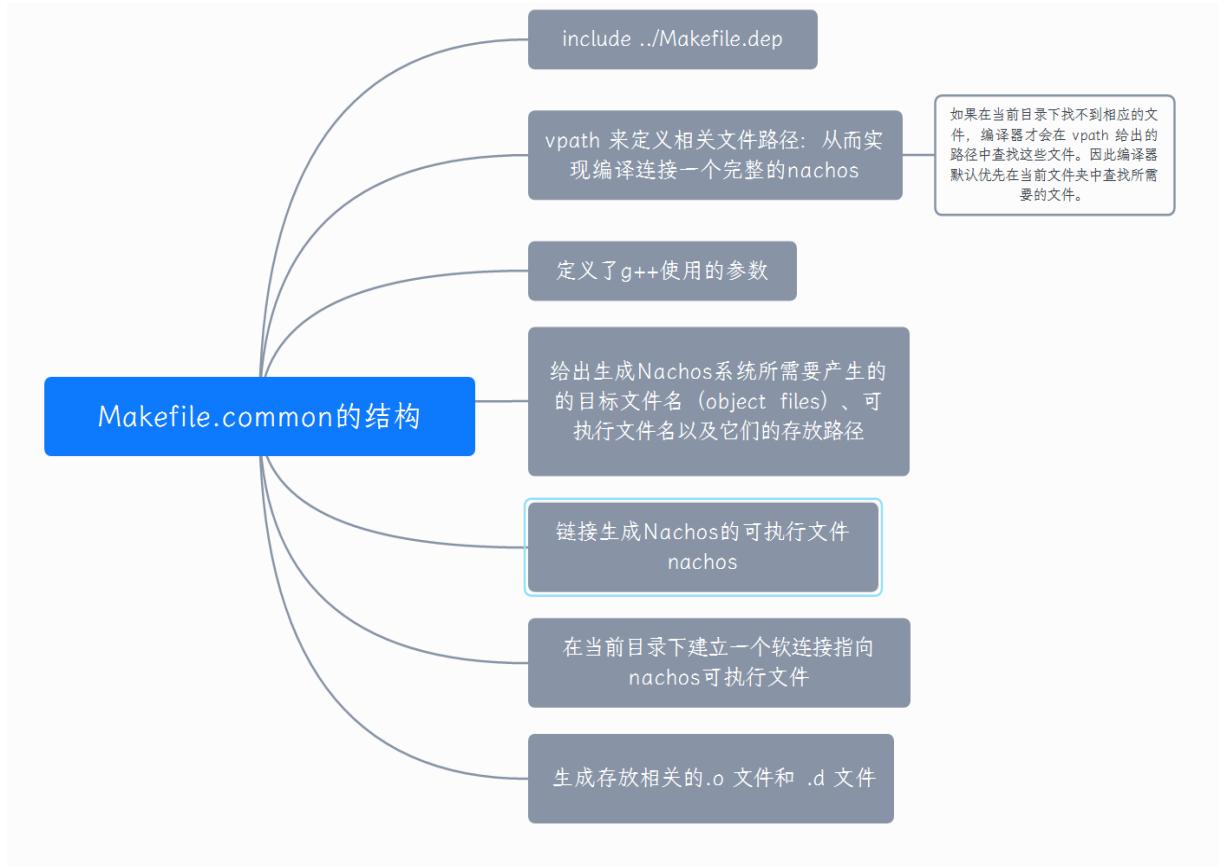
uname = \$(shell uname) 获取安装 Nachos 所使用的操作系统平台

大致结构如下



#### 5. 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。



## 测试结果

在其它目录中修改 Nachos 代码并生成修改后的 Nachos 系统然后进行测试

1. 按照指导书来移动文件到 lab2
2. 在 INCPATH 中将目录..../lab2 添加到..../threads 之前，如下：  
NCPATH += -I..../lab2 -I..../threads -I..../machine
3. **..../lab2\$ make**

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab2$ make
e
>>> Building dependency file for ./threads/switch-linux.s <<<
>>> Building dependency file for ./machine/timer.cc <<<
>>> Building dependency file for ./machine/stats.cc <<<
>>> Building dependency file for ./machine/sysdep.cc <<<
>>> Building dependency file for ./machine/interrupt.cc <<<
>>> Building dependency file for synctest.cc <<<
>>> Building dependency file for threadtest.cc <<<
>>> Building dependency file for ./threads/utility.cc <<<
>>> Building dependency file for thread.cc <<<
>>> Building dependency file for system.cc <<<
>>> Building dependency file for ./threads/synchlist.cc <<<
>>> Building dependency file for synch.cc <<<
>>> Building dependency file for scheduler.cc <<<
>>> Building dependency file for ./threads/list.cc <<<
>>> Building dependency file for main.cc <<<
>>> Compiling main.cc <<<
```

4. .. /lab2\$ touch scheduler.h

.. /lab2\$ make

```
>>> Compiling scheduler.cc <<<
g++ -m32 -g -Wall -Wshadow -I../lab2 -I../threads -I../machine -DTHREADS -DHOST_
i386 -DHOST_LINUX -DCHANGED -c -o arch/unknown-i386-linux/objects/scheduler.o sc
heduler.cc

>>> Linking arch/unknown-i386-linux/bin/nachos <<<
g++ -m32 arch/unknown-i386-linux/objects/main.o arch/unknown-i386-linux/objects
list.o arch/unknown-i386-linux/objects/scheduler.o arch/unknown-i386-linux/obje
cts/synch.o arch/unknown-i386-linux/objects/synchlist.o arch/unknown-i386-linux/
objects/system.o arch/unknown-i386-linux/objects/thread.o arch/unknown-i386-linu
m/objects/utility.o arch/unknown-i386-linux/objects/threadtest.o arch/unknown-i3
6-linux/objects/synctest.o arch/unknown-i386-linux/objects/interrupt.o arch/un
known-i386-linux/objects/sysdep.o arch/unknown-i386-linux/objects/stats.o arch/u
known-i386-linux/objects/timer.o arch/unknown-i386-linux/objects/switch-linux.o
-o arch/unknown-i386-linux/bin/nachos
```

发现只有目录 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, synch.cc, synctest.cc, system.cc, thread.cc 及 threadtest.cc 等）没有被重新处理。

5. .. /lab2\$ touch .. /threads/scheduler.h

.. /lab2\$ make

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。

6. 更新 lab2 目录下的文件

Ls

```
make: arch/unknown-i386-threads/bin/nachos: up to date.
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab2$ ls
... arch      Makefile.local    scheduler.h    system.cc    threadtest.cc
  main.cc   nachos           synch.cc     system.h
  Makefile  scheduler.cc    synctest.cc  thread.cc
```

Touch scheduler.h 之后 make

和 touch .. /threads/scheduler.h 后 make 对比

```

g++ -m32 arch/unknown-i386-linux/objects/main.o arch/unknown-i386-linux/objects/
list.o arch/unknown-i386-linux/objects/scheduler.o arch/unknown-i386-linux/objec
cts/synch.o arch/unknown-i386-linux/objects/synchlist.o arch/unknown-i386-linux/o
bjects/system.o arch/unknown-i386-linux/objects/thread.o arch/unknown-i386-linux
/objects/utility.o arch/unknown-i386-linux/objects/threadtest.o arch/unknown-i38
6-linux/objects/synctest.o arch/unknown-i386-linux/objects/interrupt.o arch/unk
nown-i386-linux/objects/sysdep.o arch/unknown-i386-linux/objects/stats.o arch/un
known-i386-linux/objects/timer.o arch/unknown-i386-linux/objects/switch-linux.o
-o arch/unknown-i386-linux/bin/nachos
ln -sf arch/unknown-i386-linux/bin/nachos nachos
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/threads$ 
cd ..
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code$ cd lab2
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab2$ mak
e
make: 'arch/unknown-i386-linux/bin/nachos' is up to date.
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab2$ 
syncntest.cc 21  llisttest.cc \n syncntest.cc

```

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

## 结论体会

1. g++ 的参数 -MM： 在编译时优先从.cc 所在的目录下找对应的头文件
2. INCPATH -I 参数 表示编译时不再按照默认的查找方式查找头文件，而是按照 INCPATH 所给的路径来查找。
3. touch .../threads/scheduler.h 后 make, INCPATH += -I.../threads -I.../machine 以及 INCPATH += -I.../lab2 -I.../threads -I.../machine 都会把所有与 scheduler.h 相关的 .cc 文件都会重新编译。  
而 “grep 语句将所有与 scheduler.h 的文件放入 lab2 文件夹中” +  
“INCPATH += -I.../lab2 -I.../threads -I.../machine” 以及 INCPATH += -I-  
I.../lab2 -I.../threads -I.../machine 只需 touch .../lab2/scheduler.h 再 make，所有  
相关.cc 都会重新编译

# 实验3 利用信号量实现线程同步

## 实验信息

姓名: XXX 学号:xxxxxxxxxxxxx 日期: 2022.3.15 班级: 19.1

## 实验目的

- (1) 进一步理解 Nachos 中如何创建线程;
- (2) 理解 Nachos 中信号量与 P、V 操作是如何实现的
- (3) 如何创建与使用 Nachos 的信号量
- (4) 理解 Nachos 中是如何利用信号量实现 producer/consumer problem;
- (5) 理解 Nachos 中如何测试与调试程序;
- (6) 理解 Nachos 中轮转法 (RR) 线程调度的实现;

## 实验任务

1. 在 code/ lab3 目录下, 详细阅读并深刻理解 ring.cc ring.h main.cc prodcons++.cc 代码
2. 在 prodcons++.cc 中添加或修改相应的代码, 满足设计要求。
3. 利用 make 编译生成新的 Nachos, 并测试其功能是否满足设计要求

## 代码与原理分析

### 1. 信号量机制

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, 并且在此基础上实现了 Lock 以及 Condition 等同步机制。

### 2. 生产者消费者问题

即生产者和消费者共享一个环形缓冲区, 生产者可以 put 到缓冲区里面, 而消费者则需要取出产品消费。当环形缓冲区满的时候生产者阻塞, 直到缓冲区有空才唤醒继续执行。同样, 当缓冲区为空时, 消费者阻塞, 直到缓冲区有物品时才唤醒消费者。

### 3. Initialize 与 命令行参数

```
for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {  
    argCount = 1;
```

```

if (!strcmp(*argv, "-d")) {
    if (argc == 1)
        debugArgs = "+"; // turn on all debug flags
    else {
        debugArgs = *(argv + 1);
        argCount = 2;
    }
} else if (!strcmp(*argv, "-rs")) {
    ASSERT(argc > 1);
    RandomInit(atoi(*(argv + 1))); // 设置随机数种子
                                    // number generator
    randomYield = TRUE;
    argCount = 2;
}

```

当传入 -rs 命令会设置随机数种子来实现轮转的时间片

传入 -d 命令时，则是打开 Debug 标记，如果的 -d \* (t 等参数) 则只打开部分 debug 标记

#### 4. 有关线程的生命周期以及调度的代码阅读

- 线程的创建

除了 main 线程之外，使用

`Thread::Fork(VoidFunctionPtr func, _int arg)`

fork() 函数进行创建新线程，而在 fork 需要提供函数指针以及 arg 代表线程编号

fork() 函数里面进行

`StackAllocate(func, arg);`

// 线程::StackAllocate

// 分配并初始化一个执行栈。堆栈是

// 使用 ThreadRoot 的初始堆栈帧进行初始化，其中：

// 启用中断

// 调用 (\*func)(arg)

// 调用 Thread::Finish

//

// "func" 是要分叉的过程

// "arg" 是要传递给过程的参数

或者可以通过 new Thread 来创建一个线程

- `Thread::Yield()`

等待调度，获得时间片，等待 currentThread yield 之后 进行调度，来获取时间片。

而在 Yield() 里面

`scheduler->ReadyToRun(this);`

把当前 this 线程 置为就绪态

然后运行调度选择的下一个线程

```
scheduler->Run(nextThread);
```

- Thread::Sleep()

首先需要保证断言是关中断状态，然后把当前线程置为阻塞态，执行 Idle 线程

```

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
```

- Thread::Finish()

需要关中断，然后设置当前的线程 ToBeDestoryed

然后进行 Thread::sleep() 从而触发线程切换

- Thread \* Scheduler::FindNextToRun ()

取就绪队列的 top 的指针，然后再从队列中 remove 对应的线程

- Scheduler::ReadyToRun (Thread \*thread)

这是线程为可运行态，然后加入就绪队列，一般就是把 old thread 重新加入就绪队列

- Scheduler::Run (Thread \*nextThread)

执行调度的线程。首先检查是否旧线程有未检测到的堆栈溢出，修改 currentThread 指针，然后设置为 RUNNING 状态，进行 SWITCH，切换到新的线程中进行执行。

而如果再经过调度，会到此处，会首先完成 SWITCH，然后判断 old thread 是否要 finish，如果是则要删除他的 carcass，**这里需要注意不能直接删除该线程因为当前正在该线程的堆栈环境中运行！**

## 5. 轮转调度的实现

使用 timer 来模拟硬件时钟中断。如果 Nachos 运行时带有 -rs 参数并提供一个随机数种子，即 nachos -rs randomseeds，则在 system.cc 中 Initialize() 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

通过 Interrupt 和 timer 的配合实现

# 设计与实现

补充完成 prodcons++.cc 代码

1. 首先是一些宏定义 无需修改 只需理解

```
// 设置生产者消费者的宏
#define BUFF_SIZE 3 // 环形缓冲区的大小
#define N_PROD    2 // 生产者的数量
#define N_CONS    2 // 消费者的数量
#define N_MESSG   4 // 每个生产者生产的消息的数量
#define MAX_NAME  16 // 名字的最大长度

#define MAXLEN  48
#define LINELEN 24

Thread *producers[N_PROD]; // 生产者队列
Thread *consumers[N_CONS]; // 消费者队列

char prod_names[N_PROD][MAX_NAME]; //array of character string for prod
names
char cons_names[N_CONS][MAX_NAME]; //array of character string for cons
names

Semaphore *nempty, *nfull; //标志空和满的信号量
Semaphore *mutex; //互斥信号量

Ring *ring;
```

2. 完善 Producer()

```
-----  
-  
// Producer  
// Loop N_MESSG times and produce a message and put it in the  
//      shared ring buffer each time.  
// "which" is simply a number identifying the producer thread.  
// 循环N次  
-----  
-  
  
void  
Producer(_int which)  
{
```

```

int num;
slot *message = new slot(0,0);

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

    // Put the code for synchronization before ring->Put(message)
here.
    // 先同步
    nempty->P();
    // 后互斥
    mutex->P();

    // 放入 message
    ring->Put(message);

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

}

```

### 3. 完善消费者

```

-----
-
// Consumer
// endless loop to fetch messages from the ring buffer and
//      record these message in the corresponding file.
//
-----

void
Consumer(_int which)
{
    char str[MAXLEN];
    char fname[LINELEN];

```

```

int fd;

slot *message = new slot(0,0);

// to form a output file name for this consumer thread.
// all the messages received by this consumer will be recorded in
// this file.
sprintf(fname, "tmp_%d", which);

// create a file. Note that this is a UNIX system call.
if ( (fd = creat(fname, 0600) ) == -1)
{
    perror("creat: file create failed");
    exit(1);
}

for ( ; ; ) {

    // Put the code for synchronization before ring->Get(message)
here.
    // 先同步后互斥
    nfull->P();
    mutex->P();

    // 获取 message
    ring->Get(message);

    // Put the code for synchronization after ring->Get(message) here.
    // 唤醒生产者
    mutex->V();
    nempty->V();

    // form a string to record the message
    sprintf(str,"producer id --> %d; Message number --> %d;\n",
    message->thread_id,
    message->value);
    // write this string into the output file of this consumer.
    // note that this is another UNIX system call.
    if ( write(fd, str, strlen(str)) == -1 ) {
        perror("write: write failed");
        exit(1);
    }
}

```

```

    }
}
```

#### 4. 完成 ProdCons

```

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

    // Put the code to construct all the semaphores here.
    // 构造信号量
    mutex = new Semaphore("mutex_sem", 1);           // 互斥信号量
    nempty = new Semaphore("empty_sem", BUFF_SIZE);   // 缓冲区是否为空的
    // 信号量
    nfull = new Semaphore("full_sem", 0);             // 缓冲区是否已满的
    // 信号量

    // Put the code to construct a ring buffer object with size
    // BUFF_SIZE here.
    // 定义环形缓冲区
    ring = new Ring(BUFF_SIZE);

    // 创建以及 fork N_PROD 个生产者线程
    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);

        // 创建新线程 并且 fork 到对应函数
        producers[i] = new Thread(prod_names[i]);
        producers[i]->Fork(Producer, i);
    };

    // 创建以及 fork N_CONS 个消费者线程
    for (i=0; i < N_CONS; i++)
    {
        // this statemet is to form a string to be used as the name for
        // consumer i.
        sprintf(cons_names[i], "consumer_%d", i);
    };
}
```

```

    // Put the code to create and fork a new consumer thread using
    consumers[i] = new Thread(cons_names[i]);
    consumers[i]->Fork(Consumer,i);

};

}

```

## 测试结果

1. 测试传入 -d t 开启 Debug 看看效果:

```

torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/threads$ ./nachos -d t
Entering SimpleTestForking thread "forked thread" with func = 0x5656b275, arg = 1
Putting thread forked thread on ready list.
*** thread 0 looped 0 times
Yielding thread "main"
Putting thread main on ready list.
Switching from thread "main" to thread "forked thread"
*** thread 1 looped 0 times
Yielding thread "forked thread"
Putting thread forked thread on ready list.
Switching from thread "forked thread" to thread "main"
Now in thread "main"
*** thread 0 looped 1 times
Yielding thread "main"
Putting thread main on ready list.
Switching from thread "main" to thread "forked thread"
Now in thread "forked thread"
*** thread 1 looped 1 times
Yielding thread "forked thread"
Putting thread forked thread on ready list.
Switching from thread "forked thread" to thread "main"
Now in thread "main"
*** thread 0 looped 2 times
Yielding thread "main"
Putting thread main on ready list.
Switching from thread "main" to thread "forked thread"
Now in thread "forked thread"
*** thread 1 looped 2 times
Yielding thread "forked thread"
Putting thread forked thread on ready list.
Switching from thread "forked thread" to thread "main"
Now in thread "main"
*** thread 0 looped 3 times
Yielding thread "main"
Putting thread main on ready list.
Switching from thread "main" to thread "forked thread"
Now in thread "forked thread"
*** thread 1 looped 3 times
Yielding thread "forked thread"
Putting thread forked thread on ready list.
Switching from thread "forked thread" to thread "main"

```

这样 我们以后可以在代码里面添加 Debug 来输出信息，更能够理清内核的运行过程

2. 补充完代码，make 后 ./nachos 执行结果如下

```

code > lab3 > tmp_0
1 producer id --> 0; Message number --> 0;
2 producer id --> 0; Message number --> 1;
3 producer id --> 0; Message number --> 2;
4 producer id --> 0; Message number --> 3;
5 producer id --> 1; Message number --> 0;
6 producer id --> 1; Message number --> 1;
7 producer id --> 1; Message number --> 2;
8 producer id --> 1; Message number --> 3;
9

code > lab3 > tmp_1
1

```

发现 0 号消费者 全消费了，1 号消费者没能消费  
输出 Debug 信息

```

Entering ProdConsForking thread "producer_0" with func = 0x565e5d8f, arg =
Putting thread producer_0 on ready list.
Forking thread "producer_1" with func = 0x565e5d8f, arg = 1
Putting thread producer_1 on ready list.
Forking thread "consumer_0" with func = 0x565e5e8d, arg = 0
Putting thread consumer_0 on ready list.
Forking thread "consumer_1" with func = 0x565e5e8d, arg = 1
Putting thread consumer_1 on ready list.
Finishing thread "main"
Sleeping thread "main"
Switching from thread "main" to thread "producer_0"
Sleeping thread "producer_0"
Switching from thread "producer_0" to thread "producer_1"
Sleeping thread "producer_1"
Switching from thread "producer_1" to thread "consumer_0"
↳ Putting thread producer_0 on ready list.
Putting thread producer_1 on ready list.
Sleeping thread "consumer_0"
Switching from thread "consumer_0" to thread "consumer_1"
Sleeping thread "consumer_1"
Switching from thread "consumer_1" to thread "producer_0"
Now in thread "producer_0"
Deleting thread "main"
Putting thread consumer_0 on ready list.
Finishing thread "producer_0"
Sleeping thread "producer_0"
Switching from thread "producer_0" to thread "producer_1"
Now in thread "producer_1"
Deleting thread "producer_0"
Putting thread consumer_1 on ready list.
Sleeping thread "producer_1"
Switching from thread "producer_1" to thread "consumer_0"
Now in thread "consumer_0"
Putting thread producer_1 on ready list.
Sleeping thread "consumer_0"
Switching from thread "consumer_0" to thread "consumer_1"
Now in thread "consumer_1"
Sleeping thread "consumer_1"
Switching from thread "consumer_1" to thread "producer_1"

```

可以发现原因：

- 从 main 线程 SWITCH 到 producer 0 生成三个 message 后，缓冲区已满，producer 1 未生产
- 执行 consumer 0，消费 0、1、2 缓冲区，唤醒 producer 0 1 然后阻塞自己
- 执行 consumer 1，缓冲区空，不能消费
- 执行 producer0，只能生产一个消息了，然后唤醒 consumer 0，结束自己

5. 执行 producer1, 生产两个消息, 唤醒 consumer1, 并且阻塞自己
6. 执行 consumer0, 消费 3 个信息, 唤醒 producer, 阻塞自己
7. 执行 consumer1, 没的消费
8. 执行 producer1, 生产最后两个, 结束自己, 唤醒 consumer0 和 consumer1
9. 执行 consumer0, 消费两个, 结束自己。
10. 执行 consumer1, 没有可消费的, 结束自己。

故: 因为设置的生产个数以及消费个数的原因, 导致不能够两个消费者都消费。

2. 解决: 采用分时系统, RR 调度, 定中断

执行: ./nachos -rs 2

结果:

```
code > lab3 > E tmp_0
1 producer id --> 0; Message number --> 0;
2 producer id --> 1; Message number --> 0;
3 producer id --> 0; Message number --> 1;
4 producer id --> 0; Message number --> 2;
5 producer id --> 0; Message number --> 3;
6
```

```
code > lab3 > E tmp_1
1 producer id --> 1; Message number --> 1;
2 producer id --> 1; Message number --> 2;
3 producer id --> 1; Message number --> 3;
4
```

还可以使用 ./nachos -rs 2 -d

输出时钟信息

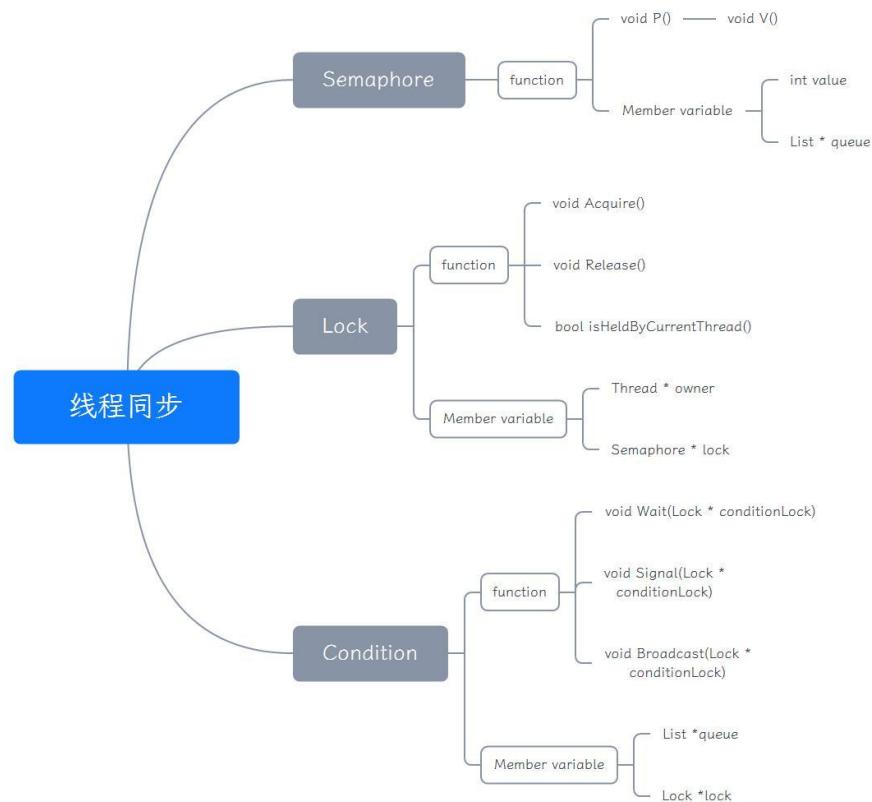
```
== Tick 810 ==
    interrupts: on -> off
Time: 810, interrupts off
Pending interrupts:
In mapcar, about to invoke 5663ffca(56800cb0)
Interrupt handler timer, scheduled at 828
End of pending interrupts
    interrupts: off -> on
    interrupts: on -> off
Sleeping thread "consumer_1"
Switching from thread "consumer_1" to thread "consumer_0"
Now in thread "consumer_0"
Sleeping thread "consumer_0"
Machine idling; checking for interrupts.
Time: 810, interrupts off
Pending interrupts:
In mapcar, about to invoke 5663ffca(56800cb0)
Interrupt handler timer, scheduled at 828
End of pending interrupts
Machine idle. No interrupts to do.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 828, idle 18, system 810, 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
```

## 结论体会

1. Nachos 的线程同步机制：主要包括信号量机制、锁机制、以及条件变量，而他们之间则是层层递进关系。

```
// Three kinds of synchronization are defined here: semaphores,  
// locks, and condition variables. The implementation for  
// semaphores is given; for the latter two, only the procedure  
// interface is given -- they are to be implemented as part of  
// the first assignment.
```



创作于 Effie (白朴版)

2. 时间片 + FCFS 调度算法 即 RR 调度算法的实现是依靠 timer 来模拟时钟中断 + Interrupt 中断机制配合实现的
  3. 消费者得不到消费的根本原因在于 **没有限制单个消费者，导致可以无限消费。**  
加时间片轮转就是**从时间片角度上限制**其不能一直消费，这样也能体现出调度

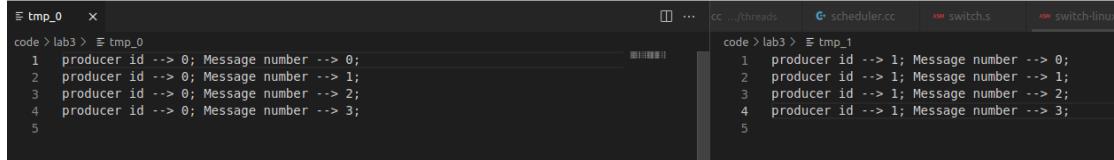
除此之外，还可以加一个每个消费者最大消费限度，从个数角度加以限制

```
#define N_CONS_MESSG 4 // 每个消费者最多消费的信息数量
```

修改消费者线程的代码，只需修改此处即可

```
int num;
for (num = 0; num < N_CONS_MESSG; num++)
```

然后不加时间片，可以看到，也能实现类似的效果，但是不能体现出调度的特点，更像顺序执行了，不能体现并发



```
code > lab3 > tmp_0
code > lab3 > tmp_1
1 producer id --> 0; Message number --> 0;
2 producer id --> 0; Message number --> 1;
3 producer id --> 0; Message number --> 2;
4 producer id --> 0; Message number --> 3;
5
1 producer id --> 1; Message number --> 0;
2 producer id --> 1; Message number --> 1;
3 producer id --> 1; Message number --> 2;
4 producer id --> 1; Message number --> 3;
```

当然，也可以在设置这个宏的基础上，进行 RR 调度

- 对于时间片：系统程序只有从关中断变为开中断才算消耗了时间片，这个宏可以自己修改。对于用户进程：执行完一条用户指令就算消耗了一个时间片。

# 实验 4 Nachos 的文件系统

## 实验信息

姓名: XXX 学号:xxxxxxxxxxxxx 日期: 2022.3.26 班级: 19.1

## 实验目的

- (1) 理解 Nachos 硬盘是如何创建的;
- (2) 熟悉查看 Nachos 硬盘上的内容的方法;
- (3) 理解硬盘初始化的过程 (如何在硬盘上创建一个文件系统);
- (4) 了解 Nachos 文件系统提供了哪些命令, 哪些命令已经实现, 哪些需要你自己实现;
- (5) 理解已经实现的文件系统命令的实现原理;
- (6) 理解硬盘空闲块的管理方法;
- (7) 理解目录文件的结构与管理;
- (8) 理解文件的结构与文件数据块的分配方法;
- (9) 了解一个文件系统命令执行后, 硬盘的布局;
- (10) 分析目前 Nachos 不能对文件进行扩展的原因, 考虑解决方案;

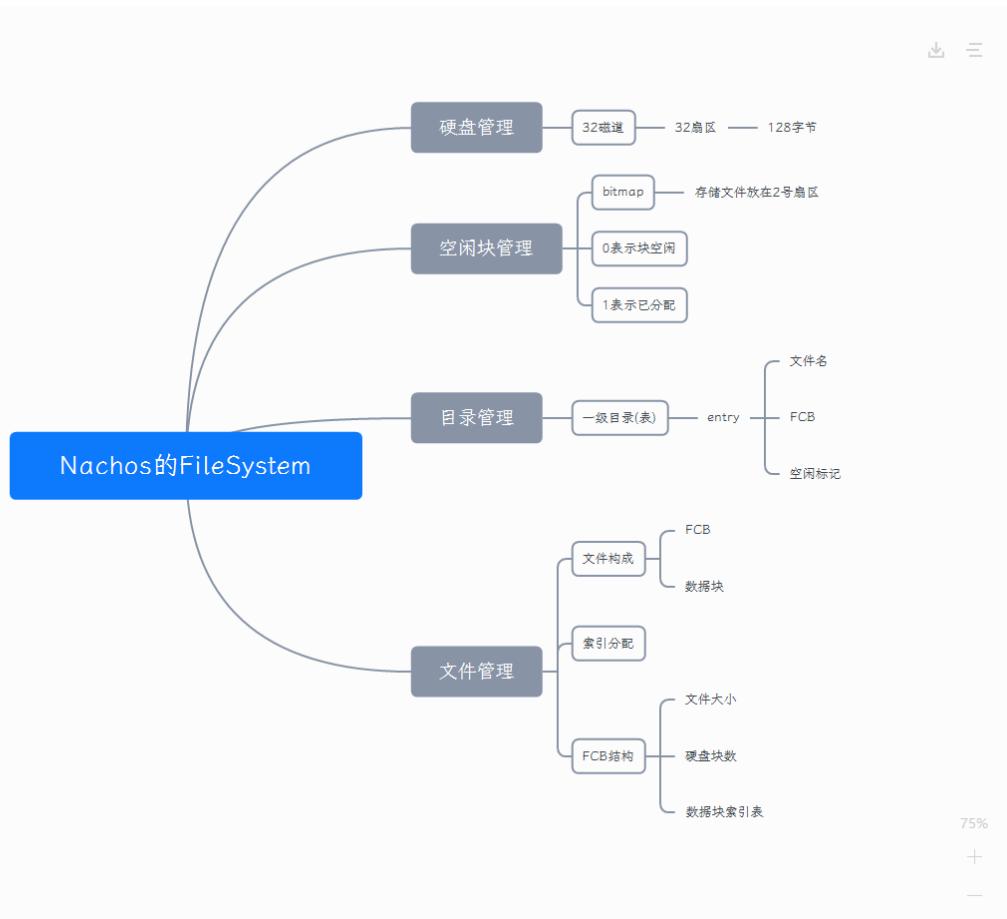
## 实验任务

1. 分析`./threads/ synchdisk.cc` 及`./machine/disk.cc`, 理解 Nachos 创建硬盘的过程与方法;
2. 分析`./lab5/main.cc`, 了解 Nachos 文件系统提供了哪些命令, 对每个命令进行测试, 根据执行结果观察哪些命令已经实现 (正确运行), 哪些无法正确运行 (尚未完全实现, 需要你自己完善);
3. 分析`./lab5/fstest.cc` 及`./filesys/filessys.cc`, 理解 Nachos 对这些命令的处理过程与方法;
4. 分析`./filesys/filessys.cc`, 特别是构造函数 `FileSystem::FileSystem(..)`, 理解 Nachos 硬盘“DISK”的创建及硬盘格式化 (创建文件系统) 的处理过程;
5. 利用命令 `hexdump -C DISK` 查看硬盘格式化后硬盘的布局, 理解格式化硬盘所完成的工作, 以及文件系统管理涉及到的一些数据结构组织与使用, 如文件头 (FCB)、目录表与目录项、空闲块管理位示图等;  
**理解文件头 (FCB) 的结构与组织、硬盘空闲块管理使用的位示图文件、目录表文件及目录下的组织与结构, 以及它们在硬盘上的位置;**
6. 利用命令 `nachos -cp ../test/small samll` 复制文件`..test/small` 到硬盘 DISK 中;

7. 利用命令 hexdump -C DISK 查看硬盘格式化后硬盘的布局，理解创建一个文件后相关的结构在硬盘上的存储布局；
8. 复制更多的文件到 DISK 中，然后删除一个文件，利用 hexdump -C DISK 查看文件的布局，分析文件系统的管理策略

## 代码与原理分析

文件系统结构示意图



主要代码文件、结构及其功能：

1. machine/disk.cc 中

```
#define MagicNumber      0x456789ab
magicNum = MagicNumber;
WriteFile(fileno, (char *) &magicNum, MagicSize); // write magic number
```

所以从 0x456789ab 开始写的 DISK 硬盘，开始的四个字节是硬盘标识地址

0x456789ab

2. disk.h 里定义了磁盘扇区的字节数

```
#define SectorSize 128 // 每个磁盘扇区的字节数
```

```
#define SectorsPerTrack 32 // 每个磁盘磁道的扇区数
#define NumTracks 32 // 每个磁盘的磁道数
#define NumSectors (SectorsPerTrack * NumTracks)
// 每个磁盘的总扇区数
```

所以 每个扇区 128 字节，

硬盘的存储分为四级：disk -> 磁道 -> 扇区 -> 字节

### 创建硬盘的过程和方法（任务一）：

一共五步

```
FileSystem::FileSystem(bool format)
{
    // 初始化文件系统
    DEBUG('f', "Initializing the file system.\n");
    // 是否需要初始化磁盘
    if (format) {
        // bitmap 初始化
        BitMap *freeMap = new BitMap(NumSectors);
        // 目录表初始化
        Directory *directory = new Directory(NumDirEntries);
        // bitmap 的文件头
        FileHeader *mapHdr = new FileHeader;
        // 文件目录表的文件头
        FileHeader *dirHdr = new FileHeader;
        DEBUG('f', "Formatting the file system.\n");

        // -----第一步-----
        // 首先，为目录和位图的 FileHeaders 分配空间
        // (确保没有人抢占这些块!) 先用 bitmap 进行标记
        freeMap->Mark(FreeMapSector);
        freeMap->Mark(DirectorySector);

        // -----第二步-----
        // 第二，为目录和位图文件的数据块分配空间
        ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
        ASSERT(dirHdr->Allocate(freeMap, DirectoryFileSize));

        // -----第三步-----
        // 刷新位图和目录 FileHeaders 回磁盘
        // 我们需要在“打开”文件之前做这个，因为打开
        // 需要从磁盘读取文件头(当前磁盘有垃圾 ? ? ?
        DEBUG('f', "Writing headers back to disk.\n");
        mapHdr->WriteBack(FreeMapSector);
        dirHdr->WriteBack(DirectorySector);
```

```

// -----第四步-----
// 现在打开位图和目录文件
// 当 Nachos 运行时, OS 系统假设这两个文件处于打开状态
freeMapFile = new OpenFile(FreeMapSector);
directoryFile = new OpenFile(DirectorySector);

// -----第五步-----
// 一旦文件“打开”，我们可以编写初始版本的每个文件到磁盘
// 此时目录是完全空的，但是位图已经被修改
// 将位图文件信息、文件目录项信息传入对应 Openfile 中
DEBUG('f', "Writing bitmap and directory back to disk.\n");
freeMap->WriteBack(freeMapFile); // flush changes to disk
directory->WriteBack(directoryFile);

if (DebugIsEnabled('f')) {
    freeMap->Print();
    directory->Print();
    delete freeMap;
    delete directory;
    delete mapHdr;
    delete dirHdr;
}
} else {
    // 非初始化操作，则根据原有位图文件头、文件目录表文件头信息初始化
    Openfile
        freeMapFile = new OpenFile(FreeMapSector);
        directoryFile = new OpenFile(DirectorySector);
    }
}

```

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；

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

这些文件头被放在众所周知的扇区，以便它们可以在启动时定位，即方便系统访问，文件头就是 FCB 或者叫 i-node。

```

// 包含空闲扇区位图的文件头的扇区,
// 和文件目录。这些文件头被放在众所周知的
// 扇区，以便它们可以在启动时定位。
#define FreeMapSector 0
#define DirectorySector 1

```

#### 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，目录就是一个 pair<文件名、扇区>的表，给出文件名就能找到对应的 FCB 在磁盘上的位置。

```
class DirectoryEntry {
public:
    bool inUse;          // Is this directory entry in use?
    int sector;          // Location on disk to find the
                         // FileHeader for this file
    char name[FileNameMaxLen + 1]; // Text name for file, with +1 for
                                  // the trailing '\0'
};
```

每个表项：inuse(是否使用) + sector + filename

#### 5. 文件系统的命令

在 lab5/fstest.cc 里面实现了各个操作的定义，包括 Copy、Append、NAppend、Print、以及 FileWrite、FileRead、PerformanceTest。

在 filesys/filesys.cc 里面定义了文件系统所需要的基础操作：list、Print、Create、Open、Remove

#### 6. Nachos 的删除操作

Nachos 删除一个文件需要：从目录删除、删除文件头空间、删除数据块空间、更改目录和 bitmap 写回磁盘，删除成功返回 TRUE，如果没有该文件返回 FALSE。

```
bool
FileSystem::Remove(char *name)
{
    Directory *directory;
    BitMap *freeMap;
    FileHeader *fileHdr;
    int sector;

    // 从磁盘取目录表
    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);
    // 找到删除文件所在的扇区
    sector = directory->Find(name);
    if (sector == -1) {
        delete directory;
        return FALSE;           // 文件未找到
    }
    // 从磁盘取文件头
    fileHdr = new FileHeader;
    fileHdr->FetchFrom(sector);
```

```

// 取 BITMAP
freeMap = new BitMap(NumSectors);
freeMap->FetchFrom(freeMapFile);
// 删除数据快、bitmap 以及目录表项
fileHdr->Deallocate(freeMap);           // remove data blocks
freeMap->Clear(sector);                 // remove header block
directory->Remove(name);
// 写回磁盘
freeMap->WriteBack(freeMapFile);         // flush to disk
directory->WriteBack(directoryFile);      // flush to disk
delete fileHdr;
delete directory;
delete freeMap;
return TRUE;
}

```

## 7. DISK 文件的创建

在 system.cc 里面的 Initialize 函数中，进行了系统的初始化，其中包括磁盘的初始化

```

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

```

在 SynchDisk 的构造函数里面

```

SynchDisk::SynchDisk(char* name)
{
    semaphore = new Semaphore("synch disk", 0);
    lock = new Lock("synch disk lock");
    disk = new Disk(name, DiskRequestDone, (_int) this);
}

```

New 了一个 Disk 对象，而 Disk 的构造函数如下

```

Disk::Disk(char* name, VoidFunctionPtr callWhenDone, _int callArg)
{
    // DISK 标识地址
    int magicNum;
    int tmp = 0;

    DEBUG('d', "Initializing the disk, 0x%x 0x%x\n", callWhenDone,
callArg);
    handler = callWhenDone;
    handlerArg = callArg;
    lastSector = 0;
    bufferInit = 0;
}

```

```

// OpenForReadWrite 中 int fd = open(name, O_RDWR, 0); 进行了 Open
fileno = OpenForReadWrite(name, FALSE);
if (fileno >= 0) {           // file exists, check magic number
    // 读对应地址 int retVal = read(fd, buffer, nBytes);
    Read(fileno, (char *) &magicNum, MagicSize);
    // 能读到 说明 DISK 已经存在了
    ASSERT(magicNum == MagicNumber);
}
else { // file doesn't exist, create it
    // 读不到 DISK, 则 open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);
    fileno = OpenForWrite(name);
    magicNum = MagicNumber;
    // 写入 DISK
    WriteFile(fileno, (char *) &magicNum, MagicSize); // write magic
number

    // 从 DiskSize - 4 写一个 int 作为 DISK 文件的末尾
    // 需要在文件末尾写入, 这样读取就不会返回 EOF
    Lseek(fileno, DiskSize - sizeof(int), 0);
    WriteFile(fileno, (char *)&tmp, sizeof(int));
}
active = FALSE;
}

```

## 8. 打开 Nachos 文件的过程

打开一个 Nachos 文件的大致过程为

- 读目录表的文件头到内存中
- 找到目录表的位置，读入内存
- 根据文件名查找目录项、找到文件头；不存在则返回错误
- 把该文件的文件头读入内存，即活结点
- 返回打开文件的文件描述符（句柄）

---

# 设计与实现

## 测试结果

### 支持文件系统的 Nachos

1. 在 code/filesys 下 make，可以生成支持文件系统的 Nachos 系统，Makefile 有：

```
include ../../threads/Makefile.local  
include ../../filesys/Makefile.local
```

能支持 threads 和 userprg 下的 C++ 文件

2. `./nachos -f` 生成模拟硬盘 DISK, `./nachos-D` 显示 DISK 的文件系统

3. od -c DISK 以八进制，输出二进制文件 DISK 的 ASCII 信息

4. **hexdump -c DISK** hexdump 是 Linux 下的一个二进制文件查看工具，它将二进制文件转换为 ASCII、八进制、十进制、十六进制格式进行查看。

5. **hexdump -C DISK** 以十六进制输出 ASCII 码

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/filesys$ hexdump
p -C DISK
00000000 ab 89 67 45 80 00 00 00 01 00 00 00 02 00 00 00 |...gE....|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080 00 00 00 00 c8 00 00 00 02 00 00 00 03 00 00 00 |.....|
00000090 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000100 00 00 00 00 1f 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00020004
```

## Nachos 的硬盘以及文件系统

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。

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；

```
#define NumDirEntries      10
Directory::Directory(int size)
{
    table = new DirectoryEntry[size];
    tableSize = size;
    for (int i = 0; i < tableSize; i++)
        table[i].inUse = FALSE;
}
```

文件大小：一级目录，然后  $\text{NumDirect} = (\text{SectorSize} - 2 * \text{sizeof}(\text{int})) / \text{sizeof}(\text{int})$   
因此每个文件最大 3780byte 也就是  $30 * 128\text{byte}$

### 测试 Nachos 文件系统命令（任务 2）

1. 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

这个上面已经测试了

2. 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 nachos\_filename;

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/labs
$ ./nachos -d -cp unixfile.txt ncsf1.txt
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1070, idle 1000, system 70, user 0
Disk I/O: reads 2, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

因为还没有实现 creat 的系统调用，所以不能执行此命令

3. nachos [-d f] -p nachos\_filename 表示输出 nachos 中文件 nachos\_filename 内容。

暂不支持

4. nachos [-d f] -r nachos\_filename 表示删除 nachos 中文件 nachos\_filename  
暂不支持

5. nachos [-d f] -l 表示输出 nachos 当前的文件目录。

暂不支持

6. nachos [-d f] -t 表示测试 nachos 文件系统的性能。

```
$ ./nachos [-d f] -t
Starting file system performance test:
Ticks: total 1070, idle 1000, system 70, user 0
Disk I/O: reads 2, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
Sequential write of 50000 byte file, in 10 byte chunks
Perf test: unable to open TestFile
Sequential read of 50000 byte file, in 10 byte chunks
Perf test: unable to open file TestFile
Perf test: unable to remove TestFile
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 16520, idle 16420, system 100, user 0
Disk I/O: reads 2, writes 1
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

7. nachos [-d f] -D 表示输出整个 nachos 文件系统的信息，包括位图文件、文件头、目录文件和普通文件。

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab5
./nachos [-d f] -D
Bit map file header:
FileHeader contents. File size: 0. File blocks:

File contents:
Directory file header:
FileHeader contents. File size: 0. File blocks:

File contents:
Bitmap set:

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

Ticks: total 2130, idle 2000, system 130, user 0
Disk I/O: reads 4, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

代码如下

```
#ifdef FILESYS
    if (!strcmp(*argv, "-cp")) {          // copy from UNIX to Nachos
        ASSERT(argc > 2);
        Copy(*(argv + 1), *(argv + 2));
        argCount = 3;
    }
```

```

else if (!strcmp(*argv, "-ap")) { // append from UNIX to Nachos
    ASSERT(argc > 2);
    Append(*(argv + 1), *(argv + 2), 0);
    argCount = 3;
}
else if (!strcmp(*argv, "-hap")) {
    // cut half and append from UNIX to Nachos
    ASSERT(argc > 2);
    Append(*(argv + 1), *(argv + 2), 1);
    argCount = 3;
}
else if (!strcmp(*argv, "-nap")) { // append from Nachos to Nachos
    ASSERT(argc > 2);
    NAppend(*(argv + 1), *(argv + 2));
    argCount = 3;
} else if (!strcmp(*argv, "-p")) { // print a Nachos file
    ASSERT(argc > 1);
    Print(*(argv + 1));
    argCount = 2;
} else if (!strcmp(*argv, "-r")) { // remove Nachos file
    ASSERT(argc > 1);
    fileSystem->Remove(*(argv + 1));
    argCount = 2;
} else if (!strcmp(*argv, "-l")) { // list Nachos directory
    fileSystem->List();
} else if (!strcmp(*argv, "-D")) { // print entire filesystem
    fileSystem->Print();
} else if (!strcmp(*argv, "-t")) { // performance test
    PerformanceTest();
}
#endif // FILESYS

```

所以可能需要自行实现-ap、-hap、-nap 命令

1. nachos [-d f] -ap UNIX\_filename nachos\_filename 表示将一个 UNIX 文件内容添加到 nachos 文件的末尾。
2. nachos [-d f] -hap UNIX\_filename nachos\_filename 表示将一个 UNIX 文件内容从 nachos 文件的中间部分开始向后添加并覆盖 nachos 文件的后半部分。
3. nachos [-d f] -nap nachos\_filename1 nachos\_filename2 表示将 nachos 中的 file1 中的内容添加到 file2 的文件末尾。

## Nachos 提供的三个测试文件

1. ./nachos -cp test/small small

然后 ./nachos -D

```
Directory contents:
Name: small, Sector: 5
FileHeader contents. File size: 38. File blocks:
6
File contents:
This is the spring of our discontent.\a
```

发现已经 cp 到 nachos 的 disk 中了

2. 分别 cp big 和 medium 到 nachos disk 里面 然后输出

```
Directory contents:
Name: small, Sector: 5
FileHeader contents. File size: 38. File blocks:
6
File contents:
This is the spring of our discontent.\a
Name: big, Sector: 7
FileHeader contents. File size: 608. File blocks:
8 9 10 11 12
File contents:
This is the spring of our discontent.\aThis is the sp
ring of our discontent.\aThis is the spring of our discontent.\aThis is the
spring of our discontent.\aThis is the spring of our di
scontent.\aThis is the spring of our discontent.\aThis is the spring of our
discontent.\aThis is the spring of our discontent.\aThis
is the spring of our discontent.\aThis is the spring of our discontent.\aTh
is is the spring of our discontent.\aThis is the spring
of our discontent.\aThis is the spring of our discontent.\aThis is the spri
ng of our discontent.\a
Name: medium, Sector: 13
FileHeader contents. File size: 152. File blocks:
14 15
File contents:
This is the spring of our discontent.\aThis is the spring of our discontent.
\This is the spring of our discontent.\aThis is the sp
ring of our discontent.\a
```

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
0x484~0x503	9 号扇区	big 文件第 2 块数据
0x504~0x583	10 号扇区	big 文件第 3 块数据
0x584~0x603	11 号扇区	big 文件第 4 块数据

0x604~0x683	12号扇区	big 文件第 5 块数据
0x684~0x703	13号扇区	medium 文件头
0x704~0x783	14号扇区	medium 文件第 1 块数据
0x804~0x883	15号扇区	medium 文件第 2 块数据

### 3. 测试删除 ./nachos -r big

删除前查看磁盘

```
00000000 ab 89 67 45 80 00 00 00 00 01 00 00 00 02 00 00 00 |...gE....|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080 00 00 00 00 c8 00 00 00 02 00 00 00 03 00 00 00 |.....|
00000090 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000100 00 00 00 00 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000180 00 00 00 00 01 00 00 00 05 00 00 00 73 6d 61 6c |.....small|
00000190 6c 00 00 00 00 00 00 00 01 00 00 00 07 00 00 00 |l.....|
000001a0 6d 65 64 69 75 6d 00 00 00 00 00 00 01 00 00 00 |medium....|
000001b0 0a 00 00 00 62 69 67 00 00 00 00 00 00 00 00 00 00 00 |...big....|
000001c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
Name: big, Sector: 10
FileHeader contents. File size: 608. File blocks:
11 12 13 14 15
```

删除

```
Directory contents:
Name: small, Sector: 5
FileHeader contents. File size: 38. File blocks:
6
File contents:
This is the spring of our discontent.\a
Name: medium, Sector: 13
FileHeader contents. File size: 152. File blocks:
14 15
File contents:
This is the spring of our discontent.\aThis is the spring of our discontent.
\atThis is the spring of our discontent.\atThis is the sp
ring of our discontent.\a
```

删除后，查看对应的磁盘块

```
00000000 ab 89 67 45 80 00 00 00 00 01 00 00 00 02 00 00 00 |...gE....|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080 00 00 00 00 c8 00 00 00 02 00 00 00 03 00 00 00 |.....|
00000090 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000100 00 00 00 00 ff 03 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000180 00 00 00 00 01 00 00 00 05 00 00 00 73 6d 61 6c |.....small|
00000190 6c 00 00 00 00 00 00 00 01 00 00 00 07 00 00 00 |l.....|
000001a0 6d 65 64 69 75 6d 00 00 00 00 00 00 00 00 00 00 00 00 |medium....|
000001b0 0a 00 00 00 62 69 67 00 00 00 00 00 00 00 00 00 00 00 |...big...|
000001c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000280 00 00 00 00 26 00 00 00 01 00 00 00 06 00 00 00 |....&....|
00000290 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000300 00 00 00 00 54 68 69 73 20 69 73 20 74 68 65 20 |....This is the |
00000310 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
00000320 73 63 6f 6e 74 65 6e 74 2e 0a 00 00 00 00 00 00 |scontent.....|
00000330 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
```

```

00000580 00 00 00 00 54 68 69 73 20 69 73 20 74 68 65 20 |....This is the |
00000590 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
000005a0 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 |scontent..This i|
000005b0 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 |s the spring of |
000005c0 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a |our discontent..|
000005d0 54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69 |This is the spri|
000005e0 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e |ng of our discon|
000005f0 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 |tent..This is th|
00000600 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 |e spring of our |
00000610 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 |discontent..This|
00000620 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f | is the spring o|
00000630 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 |f our discontent|
00000640 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 73 70 |..This is the sp|
00000650 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 |ring of our disc|
00000660 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 |ontent..This is |
00000670 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 |the spring of ou|
00000680 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 |r discontent..Th|
00000690 69 73 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 |is is the spring|
000006a0 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 | of our discontent|
000006b0 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 |nt..This is the |
000006c0 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
000006d0 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 |scontent..This i|
000006e0 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 |s the spring of |
000006f0 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a |our discontent..|
00000700 54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69 |This is the spri|
00000710 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e |ng of our discon|
00000720 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 |tent..This is th|
00000730 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 |e spring of our |
00000740 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 |discontent..This|
00000750 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f | is the spring o|
00000760 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 |f our discontent|
00000770 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 73 70 |..This is the sp|
00000780 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 |ring of our disc|
00000790 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 |ontent..This is |
000007a0 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 |the spring of ou|
000007b0 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 |r discontent..Th|
000007c0 69 73 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 |is is the spring|
000007d0 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 | of our discontent|
000007e0 6e 74 2e 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 |nt.....|
000007f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00020004

```

可以看出，删除前 big 占据 11 12 13 14 15, bit map 是 ff ff

经过 remove big 之后：

Bitmap 从原来的 ff ff 变成了 00 ff 03, ff(1111 1111) -> 03(0000 0011)

即对应于第 10 11 12 13 14 15 扇区变空

可以看出只是删除了 bitmap、然后改变了目录项 0x1ab 的值 inUse 位从 1 编程了 0，而目录表里面的文件名以及磁盘上文件头的数据以及文件数据都未删除。

这样做方便了恢复数据：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；

注：如何看 bitmap？

bitmap = ff 03, 读取单位是 1byte = 8bit 位

首先 0-7 扇区是使用的，03(0000 0011)，11 分别对应 8 9 扇区，然后从右向左 10 11 12 13 14 15 扇区是空的！

也就是 每八位从右向左读

b[0]	0	1	0	1	0	1	1	0
	7	6	5	4	3	2	1	0
b[1]	1	0	1	1	0	0	0	0
	15	14	13	12	11	10	9	8

## 思考

(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？

回答：当把 small、medium、big 文件 cp 到 Nachos 文件系统后，通过 nachos-D 发现，disk 上有三个文件，通过 hexdump -C DISK 也能看出

```
*
00000300 00 00 00 00 54 68 69 73 20 69 73 20 74 68 65 20 |....This is the |
00000310 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
00000320 73 63 6f 6e 74 65 6e 74 2e 0a 00 00 00 00 00 00 |scontent.....|
00000330 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000430 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 |s the spring of |
00000440 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a |our discontent..|
00000450 54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69 |This is the spri|
00000460 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e |ng of our discon|
00000470 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 |tent..This is th|
00000480 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 |e spring of our |
00000490 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 |discontent..This |
000004a0 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f | is the spring o|
000004b0 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 |f our discontent|
000004c0 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 73 70 |..This is the sp|
000004d0 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 |ring of our disc|
000004e0 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 |ontent..This is |
000004f0 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 |the spring of ou|
00000500 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 |r discontent..Th|
00000510 69 73 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 |is is the spring|
00000520 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 | of our disconte|
00000530 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 |nt..This is the |
00000540 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
00000550 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 |scontent..This i|
00000560 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 |s the spring of |
00000570 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a |our discontent..|
00000580 54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69 |This is the spri|
00000590 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e |ng of our discon|
000005a0 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 |tent..This is th|
000005b0 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 |e spring of our |
000005c0 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 |discontent..This |
000005d0 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f | is the spring o|
000005e0 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 |f our discontent|
000005f0 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 73 70 |..This is the sp|
00000600 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 |ring of our disc|
00000610 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 |ontent..This is |
00000620 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 |the spring of ou|
00000630 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 |r discontent..Th|
00000640 69 73 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 |is is the spring|
00000650 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 | of our disconte|
00000660 6e 74 2e 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |nt.....|
00000670 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000680 00 00 00 00 98 00 00 00 02 00 00 00 0e 00 00 00 00 00 |.....|
00000690 0f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000006a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000700 00 00 00 00 54 68 69 73 20 69 73 20 74 68 65 20 |....This is the |
00000710 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
00000720 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 |scontent..This i|
00000730 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 |s the spring of |
```

(b) 文件 big 的数据块 (data blocks) 的扇区号是多少?

通过 nachos -D, 即数据块在 8 9 10 11 12sector(file block)

注: 因为 cp 进来的顺序是 small big medium 所以 big 在 medium 前面

```
Name: big, Sector: 7
FileHeader contents. File size: 608. File blocks:
8 9 10 11 12
File contents:
This is the spring of our discontent.\aThis is the spring of our discontent.\aThis is the spring of our d
iscontent.\aThis is the sp
```

(c) 文件 big 的文件头 (file header) 的扇区号是多少?

即 FCB 位置, 即 sector 7

(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?

```
00000000 ab 89 67 45 80 00 00 00 00 01 00 00 00 00 02 00 00 00 |..gE.....
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
*
00000080 00 00 00 00 c8 00 00 00 02 00 00 00 03 00 00 00 00 00 |.....
00000090 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
*
00000100 00 00 00 00 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
*
00000180 00 00 00 00 01 00 00 00 05 00 00 00 73 6d 61 6c |.....smal
00000190 6c 00 00 00 00 00 00 00 01 00 00 00 07 00 00 00 |l....
000001a0 62 69 67 00 00 00 00 00 00 00 00 00 01 00 00 00 |big.....
000001b0 0d 00 00 00 6d 65 64 69 75 6d 00 00 00 00 00 00 |...medium...
000001c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
*
00000280 00 00 00 00 26 00 00 00 01 00 00 00 06 00 00 00 00 00 |....&....
00000290 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
*
00000300 00 00 00 00 54 68 69 73 20 69 73 20 74 68 65 20 |....This is the
00000310 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di
00000320 73 63 6f 6e 74 65 6e 74 2e 0a 00 00 00 00 00 00 |scontent.....
00000330 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
*
```

```

00000380 00 00 00 00 60 02 00 00 05 00 00 00 08 00 00 00 |....|.....|
00000390 09 00 00 00 0a 00 00 00 0b 00 00 00 0c 00 00 00 |.....|.....|
000003a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|.....|
*
00000400 00 00 00 00 54 68 69 73 20 69 73 20 74 68 65 20 |....This is the |
00000410 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
00000420 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 |scontent..This i|
00000430 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 |s the spring of |
00000440 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a |our discontent..|
00000450 54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69 |This is the spri|
00000460 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e |ng of our discon|
00000470 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 |tent..This is th|
00000480 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 |e spring of our |
00000490 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 |discontent..This|
000004a0 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f | is the spring o|
000004b0 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 |f our discontent|
000004c0 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 73 70 |..This is the sp|
000004d0 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 |ring of our disc|
000004e0 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 |ontent..This is |
000004f0 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 |the spring of ou|
00000500 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 |r discontent..Th|
00000510 69 73 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 |is is the spring|
00000520 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 | of our discontent|
00000530 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 |nt..This is the |
00000540 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
00000550 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 |scontent..This i|
00000560 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 |s the spring of |
00000570 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a |our discontent..|
00000580 54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69 |This is the spri|
00000590 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e |ng of our discon|
000005a0 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 |tent..This is th|
000005b0 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 |e spring of our |
000005c0 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 |discontent..This|
000005d0 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f | is the spring o|
000005e0 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 |f our discontent|
000005f0 2e 0a 54 68 69 73 20 69 73 20 74 68 65 20 73 70 |..This is the sp|
00000600 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 |ring of our disc|
00000610 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 |ontent..This is |
00000620 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 |the spring of ou|
00000630 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 |r discontent..Th|
00000640 69 73 20 69 73 20 74 68 65 20 73 70 72 69 6e 67 |is is the spring|
*
00000650 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e 74 65 | of our discontent|
00000660 6e 74 2e 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |nt.....|
00000670 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000680 00 00 00 00 98 00 00 00 02 00 00 00 00 0e 00 00 00 00 |.....|
00000690 0f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000006a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000700 00 00 00 00 54 68 69 73 20 69 73 20 74 68 65 20 |....This is the |
00000710 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 64 69 |spring of our di|
00000720 73 63 6f 6e 74 65 6e 74 2e 0a 54 68 69 73 20 69 |scontent..This i|
00000730 73 20 74 68 65 20 73 70 72 69 6e 67 20 6f 66 20 |s the spring of |
00000740 6f 75 72 20 64 69 73 63 6f 6e 74 65 6e 74 2e 0a |our discontent..|
00000750 54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69 |This is the spri|
00000760 6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e |ng of our discon|
00000770 74 65 6e 74 2e 0a 54 68 69 73 20 69 73 20 74 68 |tent..This is th|
00000780 65 20 73 70 72 69 6e 67 20 6f 66 20 6f 75 72 20 |e spring of our |
00000790 64 69 73 63 6f 6e 74 65 6e 74 2e 0a 00 00 00 00 00 |discontent.....|
000007a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00020004

```

Big 的 datablocks 可以看出是从 0x404~0x683 即 8~12 号扇区进行的  
 Big 的文件头可以看出是在 0x384~0x403 7 号扇区

## Nachos 文件系统在硬盘上的布局

- (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。

```

              ...
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/filesys$ hexdump -C DISK
00000000  ab 89 67 45 80 00 00 00  01 00 00 00 00 02 00 00 00 |...gE.....|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 |.....|
*
00000080  00 00 00 00 c8 00 00 00  02 00 00 00 03 00 00 00 |.....|
00000090  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 |.....|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 |.....|
*
00000100  00 00 00 00 1f 00 00 00  00 00 00 00 00 00 00 00 00 |.....|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 |.....|
*
00020004

```

- 0x0 --- 0x3 是磁盘标识 ab896745, 0x4~0x83 存放的是 bitmap 的 FCB 文件
- 0x4~0x7: 4 个字节, 位示图文件大小, 为 0x80, 表示位示图文件大小为  $0x80=128$  字节, 也就是对应 bitmap 从 0x104 ~ 0x183
- 0x8~0xB 四个字节, 0x01 表示为 bitmap 文件数据所分配的扇区数, 只需要一个扇区。(一个扇区=128 字节=0x80)
- 0xC~0xF 值为 0x2, 表示 bitmap 文件数据块所在的扇区号, 即二号扇区 (0x104~0x183)
- 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;
- 0x104~0x183: 扇区 2, 128 字节; 位示图的数据块, 存储位示图文件内容;
- 0x184~0x203, 扇区 3,
- 0x204~0x283: 扇区 4: 这两个扇区是目录文件的数据块, 存放目录项; 目前没有任何文件, 值为 0;

## 结论体会

1. nachos 的文件系统支持了一部分文件的 shell 命令, 方便操作
2. 可以使用 od -c、hexdump 查看二进制文件 DISK 的内容, 以 8、16 进制等输出, 也可以查看 ASCII 码
3. nachos 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)；

4. Nachos 默认一个文件名最长是 9 byte，最多 30 个扇区组成，maxsize = 3KB

# 实验 5 扩展 Nachos 的文件系统

## 实验信息

姓名: XXX 学号:xxxxxxxxxxxxx 日期: 2022.4.5 班级: 19.1

## 实验目的

理解文件系统中文件操作的实现方法，如文件打开、读、写、扩展、定位、关闭等；

理解如何管理硬盘空闲块；

创建文件时，如何为文件分配目录项及文件头（FCB）；

理解文件扩展时，如何为要扩展的数据查找并分配空闲块；

理解文件扩展后，文件大小是如何记录与保存的；

文件被删除后，如何回收为其分配的资源，如文件头、目录项、硬盘块等；

**拓展：**尝试多级目录（目录树）的设计与实现方法。

根据上述工作，总结操作系统（如 Nachos）读写文件，以及对文件追加数据的过程与步骤（如操作系统需要操作哪些文件系统的控制信息，如何操作的等）。

## 实验任务

修改 Nachos 的文件系统，以满足：

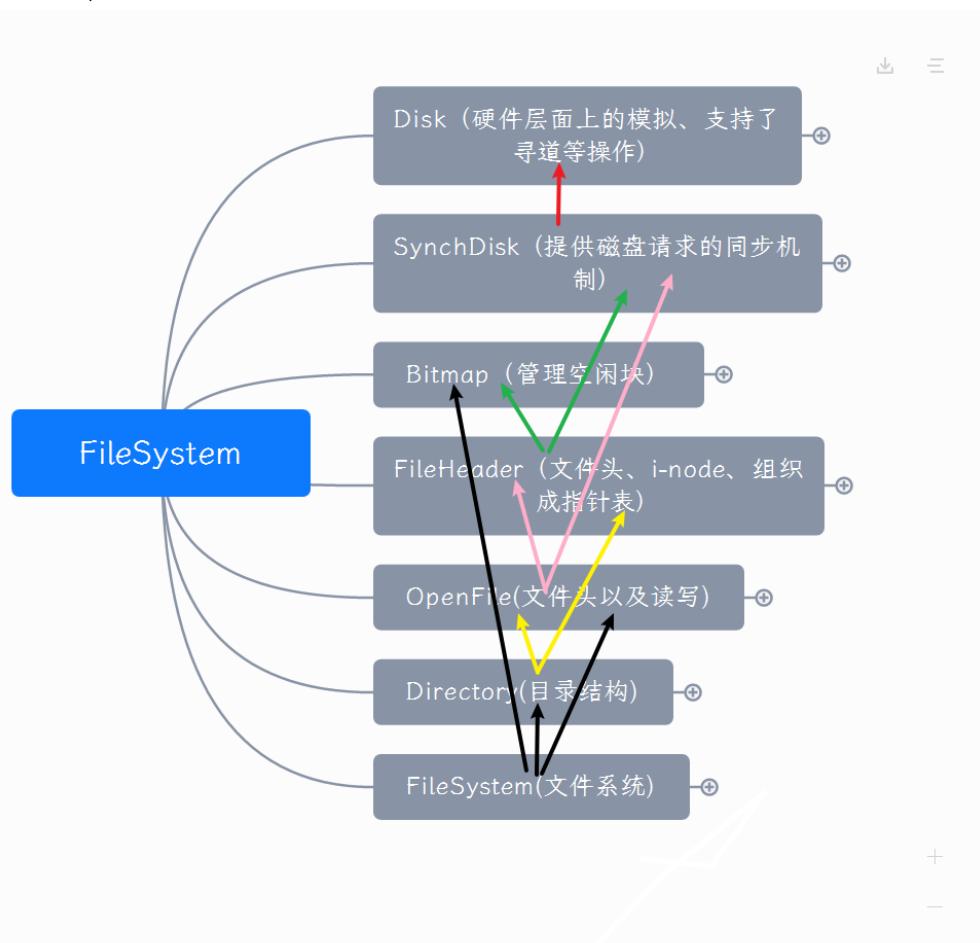
- (1) 文件创建时，其大小可初始化为 0；
- (2) 当一个文件写入更多的数据时，其大小可随之增大；
- (3) 要求能够在从一个文件的任何位置开始写入数据，即能够正确处理命令行参数 -ap, -hap, 及 -nap；

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

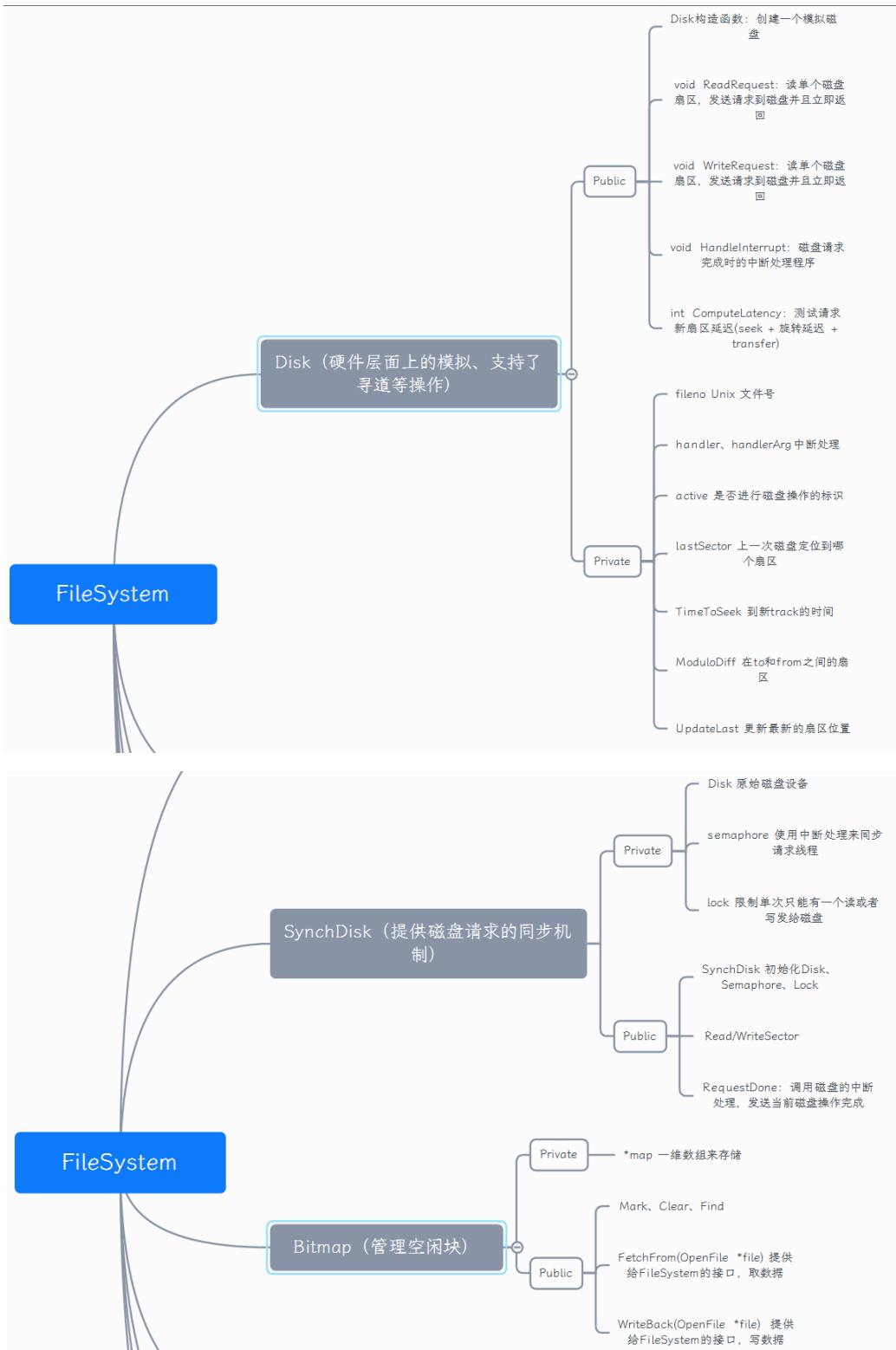
# 代码与原理分析

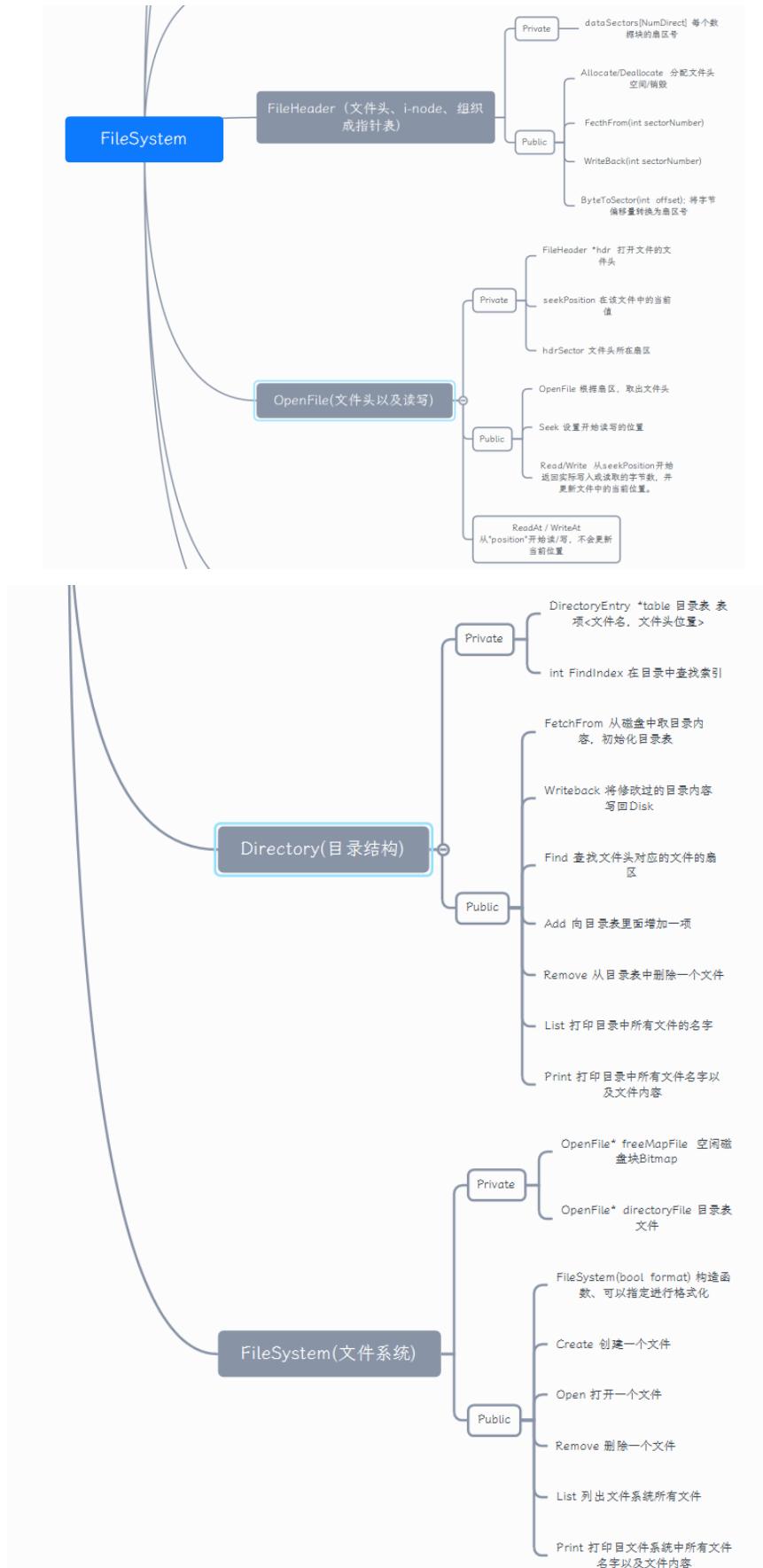
## 阅读代码

读 class Disk、class SynchDisk、class BitMap、class FileHeader、class OpenFile、class Directory 及 class FileSystem 等  
文件系统部分类的设计如下图  
各种类以及关系



具体：





回答问题：

- (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；
- 见设计与实现

## 设计与实现

### Nachos -(h/n)ap 指令的实现

#### 1. 修改 OpenFile::WriteAt()

```
// 修改之前
    // if ((numBytes <= 0) || (position >= fileLength))
    //     return 0;                                // check request
// 修改之后
    // 不符合条件的 不写入 或者 开始位置已经超过文件长度 都是异常行为
    if ((numBytes <= 0) || (position > fileLength))
        return 0;
    // 剩下可能是开始位置在文件末尾或者在中间
    // 需要写到其他扇区
    if ((position + numBytes) > fileLength){
        // 写在其他扇区的字节数
        int incrementBytes = position + numBytes - fileLength;
        BitMap *freeBitMap = fileSystem->getBitMap(); // 找空闲 bitmap
        // 分配 Bitmap
        bool hdrRet =
            hdr->Allocate(freeBitMap, fileLength, incrementBytes);
        //
        if (!hdrRet)
            // 分配失败 没有足够的磁盘空间 或者 文件太大了
            return -1;
        fileSystem->setBitMap(freeBitMap);
    }
```

这样就能够从文件末尾开始写数据了，而不是只能从文件头，并且可以写超过该扇区剩余字节数的文件。

注意，这里只是进行了 BitMap 的分配，还没有实际去写，调用 setBitMap 后才逐渐向下调用去写文件。

#### 2. 修改 FileSystem 类，增加 setBitMap() 与 getBitMap()

getBitmap 按从小到大的顺序，获得空扇区，然后进行 allocate 假分配，看能否分配，但不会进行实际的写操作，如果能够分配，再执行 setBitMap 将 Bitmap 文件写回磁盘。

```
-----
// FileSystem::getBitMap
//     获取空闲块位示图文件
-----

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

-----
// FileSystem::setBitMap
//     修改 bitmap 的状态
-----

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

### 3. 修改 OpenFile::OpenFile() 及 OpenFile::WriteBack()

将修改后的文件头写回硬盘

```
OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    seekPosition = 0;
    // 记录当前打开文件的扇区号
    hdrSector = sector;
}
```

### 4. 修改 FileHeader::Allocate()

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

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；

```
-----
// FileHeader::Allocate
// 重载，当写入的数据需要分配新的扇区的时候，为其分配
-----
bool
```

```

FileHeader::Allocate(BitMap *freeMap, int fileSize, int incrementBytes)
{
    // 修改位图文件信息以及文件头的信息，但修改结果均未写入磁盘中
    if(numSectors > 30) return false;
    // 超出限定大小 30 个扇区
    if((fileSize == 0) && (incrementBytes > 0)){
        // 在空文件后追加数据
        if(freeMap->NumClear() < 1)
            return false; // 空间不足
        dataSectors[0] = freeMap->Find();
    }
    // 先分配一个空闲磁盘块，并更新文件头信息
    numSectors = 1;
    // 初始化
    numBytes = 0;
}
numBytes = fileSize;
int offset = numSectors * SectorSize - numBytes;
// 原文件最后一个扇区块空闲空间
int newSectorBytes = incrementBytes-offset;
// 需要写到新扇区的数据 = 需要填的数据 - 最后一个扇区块空闲空间
// 最后一个扇区的空闲空间足够
if(newSectorBytes <= 0){
    numBytes = numBytes+incrementBytes;
}
// 更新文件头中的文件大小
return true;
}
// 最后一个扇区的空闲空间不足
int moreSectors = divRoundUp(newSectorBytes,SectorSize);
// 看看分多少个扇区
if(numSectors+moreSectors > 30)
    return false;
// 文件过大，超过 30 个磁盘块
if(freeMap->NumClear() < moreSectors)
    return false;
// 无足够扇区用于分配
for(int i = numSectors; i < numSectors+moreSectors; i++)
    dataSectors[i] = freeMap->Find();
numBytes = numBytes+incrementBytes;
// 更新文件大小
numSectors = numSectors+moreSectors;
// 更新文件扇区块数
return true;
}

```

## 5. 修改 ftest.cc 的 Append()

### (1) 修改写指针

对 ftest.cc 中的 Append() 函数， 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

ftest.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 ftest.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。

关键代码如下

```

while ((amountRead = fread(buffer, sizeof(char), TransferSize, fp)) > 0)
{
    int result;
    // printf("start value: %d, amountRead %d, ", start, amountRead);
    // result = openFile->WriteAt(buffer, amountRead, start);
    result = openFile->Write(buffer, amountRead);
    // printf("result of write: %d\n", result);
    if(result < 0){                                // 数据读取发生错误
        printf("\nERROR!!!\n");
        printf("Insufficient Disk Space, or File is too big!\nWriting Terminated!\n");
        break;
    }
    ASSERT(result == amountRead);
    // start += amountRead;
    // ASSERT(start == openFile->Length());
}
delete [] buffer;

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

```

## CatalogTree 目录树的设计思想

在数据结构课设中，实现过带父结点指针的兄弟链表所实现的目录树，但是阅读 Nachos 代码，发现目录节点是 DirectoryEntry 并且在一开始初始化目录的时

候，是以数组的形式初始化的，这样就不好进行像链表那样的动态新建目录或者文件的操作，也没法像链表一样索引关联节点。但是可以使用模拟指针的思想，以下标作为索引值，来找到某个文件/目录的父目录、左孩子、以及兄弟结点，从而能够遍历整颗树。

按照上午思路进行编程发现，不需要那么强的拓展功能，所以不采用兄弟链表作为数据结构了，而是采用带模拟指针的多叉树数据结构来实现目录树和多级目录，并且使用 graphviz+dot 可视化目录树。

1. 首先修改 DirectoryEntry，加入两个模拟指针以及孩子数的记录如下

```
class DirectoryEntry {
public:
    bool inUse;           // Is this directory entry in use?
    int sector;           // Location on disk to find the
                          // FileHeader for this file
    char name[FileNameMaxLen + 1]; // Text name for file, with +1 for
                                  // the trailing '\0'

    int parent;           // father dir 's index
    int Child_Num;        // dir's child num
    int children[MaxDirChildNUM]; // dir's children
    // bool filetype;      // 0 文件 1 目录

};
```

2. 在 Directory 的构造函数中，设置 0 目录项是根目录 Root，初始化的相关信息。他的父节点为-1，即没有，且 children 全为-1，即未分配。

```
Directory::Directory(int size)
{
    table = new DirectoryEntry[size];
    tableSize = size;
    for (int i = 0; i < tableSize; i++){
        table[i].inUse = FALSE;
        // update
        table[i].Child_Num = 0;
        table[i].parent = -1;
        for(int j = 0; j < MaxDirChildNUM; j ++){
            table[i].children[j] = -1;
        }
    }
    // set table[0] as root
    table[0].parent = -1;
    table[0].inUse = TRUE;
```

```
}
```

3. 阅读代码发现，在目录中，真正在 cp 命令时发挥作用的是 FindIndex 以及 Add 函数，首先修改 Add 函数

思路：

- 看文件是否存在
- 分割路径(因为可能是多级目录路径)
- 找到要添加的节点的父节点
- 加入新节点并且更新新节点信息
- 更新父节点信息

```
bool
Directory::Add(char *name, int newSector)
{
    // My Code
    // 1. Judge whether file is already exists
    if (FindIndex(name) != -1)
        return FALSE;
    // 2. Split the path
    string copy_name = name;
    // Split it from the last file
    // For example: dir1/dir2/dir3/file   cut into:<dir1/dir2/dir3,file>
    int cutpos = -1;
    // Seek Cut Position
    for(int i=0;i<copy_name.length();i++)
        if(copy_name[i] == '/')
            cutpos = i;
    string ParentDir = "";
    string ChildName = "";
    for(int i=0;i<cutpos;i++)
        ParentDir+=copy_name[i];
    for(int i=cutpos+1;i<copy_name.length();i++)
        ChildName += copy_name[i];
    // 3. Find ParentDir's ID
    // Initially set it = root
    int ParentDirId = 0;
    // Maybe ParentDir is empty
    if(ParentDir.length() > 0){
        // Limit 100 level
        char cstrParentName[1000];
        strcpy(cstrParentName , ParentDir.c_str());
        ParentDirId = FindIndex(cstrParentName);
```

```

    }

    // 4. Add new node in DirTable, and add info into Table Tree
    for(int i=1;i<tableSize;i++){
        // find a empty DirEntry
        if(!table[i].inUse){
            // Set attributes
            table[i].inUse = TRUE;
            table[i].sector = newSector;
            table[i].parent = ParentDirId;
            table[i].Child_Num = 0;
            // Rename
            for(int j=0;j<FileNameMaxLen;j++){
                if(j<ChildName.length())
                    table[i].name[j] = ChildName[j];
                else
                    table[i].name[j] = '\0';
            }
            // Update parent's info
            table[ParentDirId].children[table[ParentDirId].Child_Num] =
i;
            table[ParentDirId].Child_Num++;
            printf("Add File/Dir Successfully\n");
            return TRUE;
        }
    }
    return FALSE;
}

```

#### 4. 修改 FindIndex 命令 使其能够正确根据多级路径查找目录表项

思路

- 检查是否 DISK 初始化(即存在根目录)
- 分割路径
- 找到根目录下第一级目录
- 从上到下进行扫描，匹配路径名称直到找到最后一个文件/目录

代码

```

int
Directory::FindIndex(char *name)
{
    // origin code
    /*
    for (int i = 0; i < tableSize; i++)

```

```
    if (table[i].inUse && !strcmp(table[i].name, name,
FileNameMaxLen))
        return i;
    return -1;      // name not in directory
*/



// My updated code

// List();

// 1. Check whether there is dir or file from root
if(table[0].Child_Num <= 0){
    printf("No dir or file from root\n");
    return -1;
}

// 2. Split the multi path
vector<string> PathTable = SplitPath(name);

// 3. Find the first root dir ForExample : dir1/dir2/dir3/txt
//                               We need to find dir1 firstly
// to scan
string scandir = PathTable[0];
// index of the first dir
int firstDirIndex = -1;
for(int i = 0; i < table[0].Child_Num; i ++){
    int index = table[0].children[i];
    // judge inuse
    if(!table[index].inUse)
        continue;
    // judge Name Equal
    bool issame = JudgeNameEqual(scandir,index,table);
    if(issame){
        firstDirIndex = index;
        break;
    }
}

// check valid
if(firstDirIndex == -1)
    return -1;
```

```
// 4. From up to bottom scan the Tree and match
// To find final parent dir
// maybe just one dir
if(PathTable.size() == 1)
    return firstDirIndex;

// multi dir
// and parent Id is the parent of target file (which means target
dir)
int parentId = firstDirIndex;
// scan multi dir to match name
for(int i = 1; i < PathTable.size(); i++){
    // find flag
    bool isfind = FALSE;
    // scan this level's child to match
    for(int j = 0; j < table[parentId].Child_Num; j++){
        // In this level, current scan ID
        int currentId = table[parentId].children[j];
        // check inuse
        if(!table[currentId].inUse)
            continue;
        // check name
        bool issame = JudgeNameEqual(PathTable[i],currentId,table);
        if(issame){
            // we find the next dir
            isfind = true;
            // record the last dir in parentId
            parentId = currentId;
            break;
        }
    }
    // Find Failed
    if(!isfind)
        return -1;
}
printf("Successfully Find: %s",parentId);
// return the parent Dir
return parentId;
}
```

## 5. 为了展示目录树，修改 List 代码

## 思路

从 table[0] 即根目录开始向下遍历，迭代输出所有文件和目录的名称  
代码

```
void
Directory::List()
{
    // My codes:
    // List all the file names in dir
    printf("Directory Contents:\n");
    printf("/root\n");
    // from the root
    for(int i=1;i<tableSize;i++){
        // check ust
        if(table[i].inUse){
            // compose name and path
            vector<string> PathTable;
            int index = i;
            // find the parents
            while(table[index].parent != -1){
                string tmp = table[index].name;
                PathTable.push_back(tmp);
                index = table[index].parent;
            }
            // cout the path
            printf("/root");
            for(int j = PathTable.size() - 1; j >= 0; j --){
                printf( "/");
                cout << PathTable[j];
            }
            printf("\n");
        }
    }
}
```

## 6. 修改 Print 函数 使-D 指令能够正常执行

### 思路

- 从上到下进行层次遍历，名字记录在 vector<string>里，然后输出
- 代码

```
void
Directory::Print()
{
    // My Codes
    FileHeader *hdr = new FileHeader;
```

```

printf("Directory contents:\n");
for (int i = 0; i < tableSize; i++){
    if (table[i].inUse) {
        vector<string> PathTable;
        int index = i;
        // not to root
        while(table[index].parent != -1){
            string temp = table[index].name;
            PathTable.push_back(temp);
            index = table[index].parent;
        }
        printf("Name: /root");
        // the next path
        for(int j=PathTable.size()-1;j>=0;j--){
            printf("/");
            cout<<PathTable[j];
        }
        printf("\n");
        printf("Sector: %d\n",table[i].sector);
        hdr->FetchFrom(table[i].sector);
        hdr->Print();
    }
}
printf("\n");
delete hdr;
}

```

7. 重写 Remove 函数，使其能够正常删除多级目录，并且能够递归删除含有文件或者目录的目录

思路

- 根据多级目录 找到该文件
- 修改目录表项的 InUse 信息
- 更新有关父节点的信息
- 如果删除的是目录，递归删除该目录所有内容

代码

```

bool
Directory::Remove(char *name)
{
    // MY codes:
    int i = FindIndex(name);
    // name not in directory
    if (i == -1)

```

```

        return FALSE;
    // clean dir table set false
    table[i].inUse = FALSE;

    // Update parent
    int parentId = table[i].parent;
    // Until Root
    if(parentId != -1){
        // record other child except I
        vector<int> tempChild;
        for(int j=0;j

```

递归删除子目录或者子文件 DeteleChildren

```

//-----
// Directory::DeleteChildren
//
// "name" -- the file name to be removed
//-----
-
void
Directory::DeleteChildren(int i){
    for(int j = 0; j < table[i].Child_Num; j++){
        int index = table[i].children[j];

```

```

    if(table[index].inUse){
        table[index].inUse = FALSE;
        table[index].Child_Num = 0;
        DeleteChildren(index);
    }
}
}

```

## 8. 可视化

- (1) 增加 -V 参数 代表 visualize Catalog Tree
- (2) 在 filesys 添加相应成员函数

```
void VisualTree(); // Visualize CataLog Tree
```

以及

```

//-----
// FileSystem::VisualTree
// Visual CataLog TRee
//-----
void
FileSystem::VisualTree()
{
    Directory *directory = new Directory(NumDirEntries);

    directory->FetchFrom(directoryFile);
    directory->VisualCataLogTree();
    delete directory;
}

```

- (3) 在目录类中加入可视化函数:

思路

- 使用 graphviz + dot 作图
- ofstream 写出到文件 dot 代码
- 通过层次遍历添加 dot 代码
- 系统执行进行作图

代码

```

//-----
// Directory::VisualCataLogTree
// VisualCataLogTree
//-----
void Directory::VisualCataLogTree(){
    ofstream fout;
    fout.open("VisualCataLogTree.dot");
    fout<<"strict digraph s{\n";

```

```

fout<<"root[shape=record,color=green];\n";
for(int i=0;i<tableSize;i++){
    if(table[i].inUse){
        int index = i;
        vector<string> PathTable;
        while(table[index].parent != -1){
            string s = table[index].name;
            PathTable.push_back(s);
            index = table[index].parent;
        }
        PathTable.push_back("root");
        for(int j=PathTable.size()-1;j>=1;j--)
            fout<<PathTable[j] << " -> "<<PathTable[j-1]<<";\n";
    }
}
fout<<"}\n";
fout.close();
system("dot -Tpng VisualCataLogTree.dot -o VisualCataLogTree.png");
return ;
}

```

完成！

## 测试结果

### 测试文件系统的命令

#### 1. 测试 ap 命令

```

rm DISK
./nachos -f
./nachos -cp test/small small
./nachos -ap test/small small
./nachos -ap test/big small
./nachos -ap test/medium medium
./nachos -ap test/big small
./nachos -cp test/small small
./nachos -D

```

加入 small 文件后

```
Directory contents:  
Name: small, Sector: 5  
FileHeader contents. File size: 22. File blocks:  
6  
File contents:  
This is a small file.\a
```

进行 ap small small 操作后

```
Directory contents:  
Name: small, Sector: 5  
FileHeader contents. File size: 44. File blocks:  
6  
File contents:  
This is a small file.\aThis is a small file.\a
```

./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；

可以看出分配新扇区成功

./ nachos -ap test/medium medium, 测试给一个空文件追加数据的功能；

```
Name: medium, Sector: 12
FileHeader contents. File size: 92. File blocks:
13
File contents:
This is a medium file.\aThis is a medium file.\aThis is a medium file.\aThis is a medium file.\a
```

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；

./ 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；

Ap 之前

Ap 之后

发现在执行(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:

./将 test 目录下的 UNIX 文件 small 附加到 Nachos 文件 small 中；测试给一个已存在的文件追加数据

发现不会覆盖已经存在的文件，而是什么都不做

## 2. 测试 hap 指令

测试前后的文件内容如图，发现进行了从中间位置的写入和覆盖，总的文件 size 没变

./nachos -nap medium small, 测试将一个 nachos 文件附加到另一个 nachos 文件的功能；

写前后内容如下图， nap 成功

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

发现与上一个实验一样，small 文件虽然被删除了，但是 small 的文件头和数据都没有完全删除，而是 bitmap 里面 small 文件对应的扇区清空了，以及 small 文件头的 inuse 变量被置为 0 了。

用 hexdump -C DISK 具体查看

```
00000180  00 00 00 00 01 00 00 00  05 00 00 00 00 6d 65 64 69 | .....medi|
00000190  75 6d 00 00 00 00 00 00  00 00 00 00 00 07 00 00 00 |um.....|
000001a0  73 6d 61 6c 6c 00 00 00  00 00 00 00 00 00 00 00 00 |small....|
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 |.....|
*
```

00000400 00 00 00 00 54 68 69 73 20 69 73 20 61 20 73 6d |....This is a sm  
00000410 61 6c 6c 20 66 69 6c 65 2e 0a 54 68 69 73 20 69 |all file..This i  
00000420 73 20 61 20 62 69 67 20 66 69 6c 65 2e 0a 54 68 |s a big file..Th  
00000430 69 73 20 69 73 20 61 20 62 69 67 20 66 69 6c 65 |is is a big file  
00000440 2e 0a 54 68 69 73 20 69 73 20 61 20 62 69 67 20 |..This is a big  
00000450 66 69 6c 65 2e 0a 54 68 69 73 20 69 73 20 61 20 |file..This is a  
00000460 62 69 67 20 66 69 6c 65 2e 0a 54 68 69 73 20 69 |big file..This i  
00000470 73 20 61 20 62 69 67 20 66 69 6c 65 2e 0a 54 68 |s a big file..Th  
00000480 69 73 20 69 73 20 61 20 62 69 67 20 66 69 6c 65 |is is a big file  
00000490 2e 0a 54 68 69 73 20 69 73 20 61 20 62 69 67 20 |..This is a big  
000004a0 66 69 6c 65 2e 0a 54 68 69 73 20 69 73 20 61 20 |file..This is a  
000004b0 62 69 67 20 66 69 6c 65 2e 0a 54 68 69 73 20 69 |big file..This i

测试 nachos -l (列目录), nachos -p small (显示 small 的内容) 等命令

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab5$ ./nachos -m medium  
No threads ready or runnable, and no pending interrupts.
```

-p

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab5$ ./nachos -p small  
Print: unable to open file small
```

## 测试文件系统的限制

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

发现 此时已经达到一个文件的最大 size 了，不能写入。但是为什么 filesize 不是  $30 * 128B = 3840B$  呢。

首先 hexdump -C DISK 查看内容，发现确实后面没写入

```

0000011e0 54 68 69 73 20 69 73 20 61 20 62 69 67 20 66 69 | This is a big fi|
0000011f0 6c 65 2e 0a 54 68 69 73 20 69 73 20 61 20 00 00 |le..This is a ...|
000001200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
```

观察代码发现，每次都是从文件读取了 TransferSize 个 Byte 进行写，而 TransferSize = 10，所以最后会有 0--9byte 的空间被浪费掉，也就是内碎片

```

// Append the data in TransferSize chunks
buffer = new char[TransferSize];
while ((amountRead = fread(buffer, sizeof(char), TransferSize, fp)) > 0)
{
    int result;
    // printf("start value: %d, amountRead %d, ", start, amountRead);
    // result = openFile->WriteAt(buffer, amountRead, start);
    result = openFile->Write(buffer, amountRead);
    // printf("result of write: %d\n", result);
    if(result < 0){                                // 数据读取发生错误
        printf("\nERROR!!!\n");
        printf("Insufficient Disk Space, or File is too big!\nWriting Terminated!\n");
        break;
    }
    ASSERT(result == amountRead);
    // start += amountRead;
    // ASSERT(start == openFile->Length());
}

```

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

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab5$ ./nachos -cp test/small small10
Copy: couldn't create output file small10
```

总共 10 个文件

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab5$ ./nachos -l
small
small1
small2
small3
small4
small5
small6
small7
small8
small9
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

./nachos -cp test/empty empty

生成了空文件

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab5$ ./nachos -l
empty
small
```

文件 size 为 0 并且没有分配文件块

```
Directory contents:
Name: empty, Sector: 5
FileHeader contents. File size: 0. File blocks:
```

## 测试 CataLogTree

### 修改宏定义

在 directory.h

```
#define MaxDirChildNUM    20 // the number of child that a dir can have
使得最多有 20 个孩子
```

在 fileys.cc 中

```
#define NumDirEntries     20
使得目录表可以有 20 项，最多 20 个文件+目录
```

### 测试 cp 命令

执行如下代码

```
rm DISK
./nachos -f
./nachos -cp test/small /file1
./nachos -cp test/empty /dir1
./nachos -cp test/empty /dir12
./nachos -cp test/medium /file2
./nachos -cp test/medium /dir1/file21
./nachos -cp test/empty /dir1/dir2
./nachos -cp test/small /dir1/file22
./nachos -cp test/small /dir12/file121
./nachos -cp test/empty /dir12/dir21
./nachos -cp test/small /dir12/file122
```

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab5MultiDir$ ./nachos -cp test/empty /dir12
Creating file /dir12, size 0
This is in Find now
Add File/Dir Successly
Opening file /dir12
This is in Find now
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

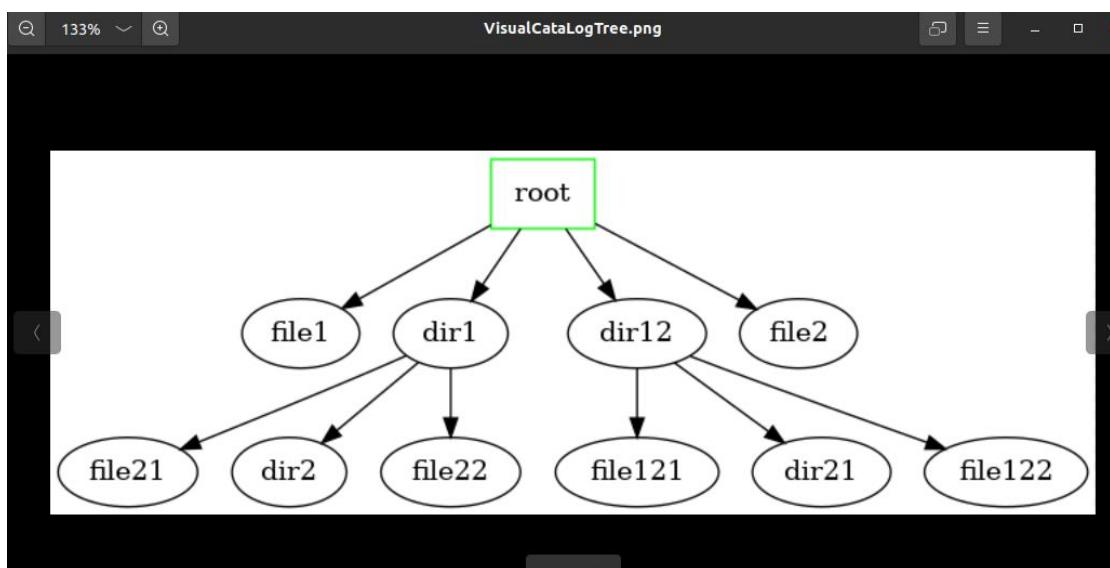
Ticks: total 199880, idle 198670, system 1210, user 0
Disk I/O: reads 27, writes 13
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

## 测试-1 命令

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab5MultiDir$ ./nachos -l
Directory Contents:
/root
/root/file1
/root/dir1
/root/dir12
/root/file2
/root/dir1/file21
/root/dir1/dir2
/root/dir1/file22
/root/dir12/file121
/root/dir12/dir21
/root/dir12/file122
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

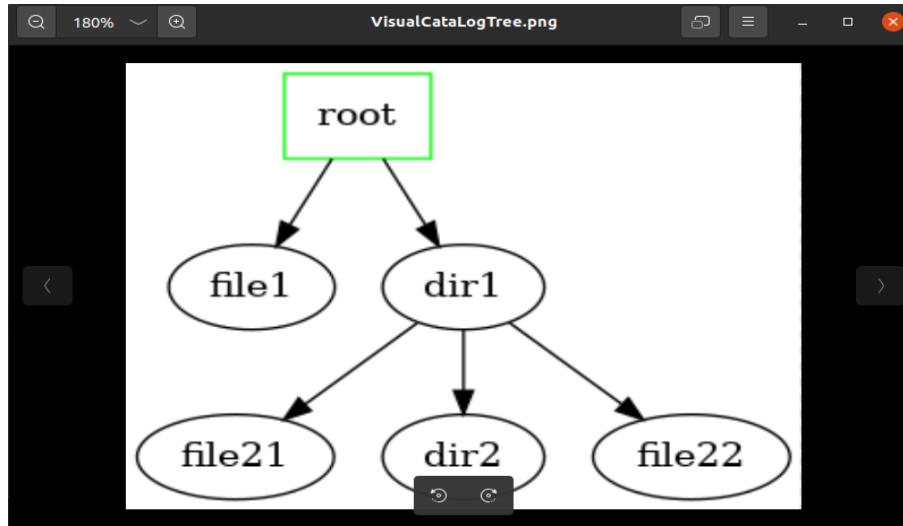
## 测试-V 命令

输出图片



## 测试-r 命令

```
./nachos -r /dir12
./nachos -r /file2
./nachos -V
```



## 结论体会

1. 在目录树的设计中，使用了 m 叉树，首先肯定是把这个数据结构应用到目录表上的，但是同时不仅仅需要修改目录表类的函数，很多其他函数也会修改，体现了操作系统紧密连接的
2. 实现目录表时，用了一些 STL 等工具，会方便很多，但是也出现了 gcc 版本不一致导致的错误，即在 `utils.h` 和 `sysdep.cc` 里面定义了一些 c++ 的工具时基于 C++98 和 C++11 差距较大，会报错，其中一种解决方案是把他们注释掉使用 `stdlib.h` 库
3. 在实现文件 `append` 时，基本按照了指导书的参考去写的，其中 `hap` 从中间续写，查了一下 Linux 时没有类似的命令的，而这个命令的初衷好像就是从中间开始写然后进行覆盖，而不是插入，所以不需要移动文件后面的内容。
4. 在 nachos 文件系统中，复制一个文件时所采用的策略是将文件按 10 字节大小的缓冲区进行划分，每次仅传输 10 字节大小的数据写入 DISK 中。这样做的目的是将一个大文件拆成多个小文件进行依次传输，降低传输过程中出错的概率。
5. 使用多叉树复杂度大致为  $O(\log_m(n))$ ，比单单一个线性表的目录表的查找更加高效，真实的操作系统应该需要这种高效。

# 实验 6 Nachos 用户程序与系统调用

## 实验信息

姓名: XXX 学号:xxxxxxxxxxxxx 日期: 2022.4.12 班级: 19.1

## 实验目的

为后续实验中实现系统调用 Exec() 与 Exit() 奠定基础  
理解 Nachos 可执行文件的格式与结构;  
掌握 Nachos 应用程序的编程语法, 了解用户进程是如何通过系统调用与操作系统内核进行交互的;  
掌握如何利用交叉编译生成 Nachos 的可执行程序;  
理解系统如何为应用程序创建进程, 并启动进程;  
理解如何将用户线程映射到核心线程, 核心线程执行用户程序的原理与方法;  
理解当前进程的页表是如何与 CPU 使用的页表进行关联的;

## 实验任务

1. 体验 Nachos 的用户程序、应用进程及 Nachos 系统调用的相关概念;
2. 阅读 ./bin/noff.h, 分析 Nachos 可执行程序.noff 文件的格式组成;
3. 阅读 ./test 目录下的几个 Nachos 应用程序, 理解 Nachos 应用程序的编程语法, 了解用户进程是如何通过系统调用与操作系统内核进行交互的;
4. 阅读 ./test/Makefile, 掌握如何利用交叉编译生成 Nachos 的可执行程序;
5. 阅读 ./threads/main.cc, ./userprog/progtest.cc, 根据对命令行参数-x 的处理过程, 理解系统如何为应用程序创建进程, 并启动进程的;
6. 阅读 ./userprog/progtest.cc, ./threads/scheduler.cc (Run()), 理解如何将用户线程映射到核心线程, 以及核心线程执行用户程序的原理与方法;
7. 阅读 ./userprog/progtest.cc, ./machine/translate.cc, 理解当前进程的页表是如何与 CPU 使用的页表进行关联的;

# 代码与原理分析

## Nachos 应用程序与加载

Nachos 是一个操作系统，可以运行 Nachos 应用程序。Nachos 应用程序采用类 C 语言语法，并通过 Nachos 提供的系统调用进行编写。

Nachos 模拟了一个执行 MIPS 指令的 CPU，因此需要利用 Nachos 提供的交叉编译程序 `gcc-2.8.1-mips.tar.gz` 将用户编写的 Nachos 应用程序编译成 MIPS 框架的可执行程序。

`gcc` MIPS 交叉编译器将 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。

注：这里在 `userprog` 目录下 输入`./nachos -x ../test/halt.noff` 命令才能正确找到 `noff` 文件

## Nachos 应用程序与可执行程序

### Nachos 应用程序的产生流程以及运行方法

查看 `test` 目录下的 `Makefile` 文件

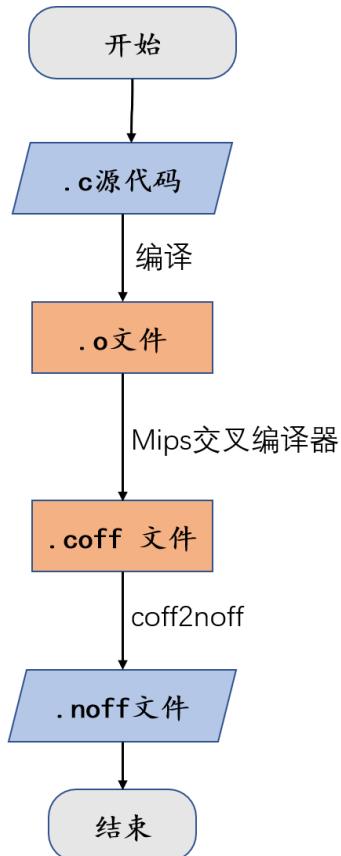
```
CC = $(GCCDIR)gcc
AS = $(GCCDIR)as
LD = $(GCCDIR)ld
targets = halt shell matmult sort
INCDIR = -I../userprog -I../threads
CFLAGS = -G 0 -c $(INCDIR)
coff2noff = ../bin/$(real_bin_dir)/coff2noff
coff2flat = ../bin/$(real_bin_dir)/coff2flat
$(all_coff): $(obj_dir)/%.coff: $(obj_dir)/start.o $(obj_dir)/%.o
    @echo ">>> Linking" $(obj_dir)/$(notdir $@) "<<<"
    $(LD) $(LDFLAGS) $^ -o $(obj_dir)/$(notdir $@)
```

```

$(all_noff): $(bin_dir)/%.noff: $(obj_dir)/%.coff
    @echo ">>> Converting to noff file:" $@ "<<<"
    $(coff2noff) $^ $@
    ln -sf $@ $(notdir $@)
# 将 coff 文件转变为 noff 文件
%.s: %.c
    @echo ">>> Compiling .s file for" $< "<<<"
    $(CC) $(CFLAGS) -S -c -o $@ $<

```

大致流程如下图



## Nachos 可执行程序格式

```

#define NOFFMAGIC 0xbadfad /* magic number denoting Nachos
                           * object code file
                           */
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;

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;

```

1 NOFFMAGIC 魔数 定义为 0xbadfad 用来表示 noff 文件

## 2. segment 段结构

VitrualAddr	段在虚拟地址空间中的地址
inFileAddr	段在文件(物理空间)中的地址
int size	段的大小

## 3. noffHeader noff 头结构

noffMagic	noff 文件的魔数, 用来标识文件
code	可执行代码段
initData	已经初始化的数据段
uninitData	未初始化的数据段在使用之前应空

## Nachos 分页管理方式

首先 用户进程的创建: Nachos 中的进程是通过形成一个地址空间从线程演化而来的。

Nachos 的存储管理采用分页管理方式

在 Machine/translate.h 里面声明了 Nachos 的页表项

```

class TranslationEntry {
public:
    int virtualPage; // 虚页号。
    int physicalPage; // 物理内存页号
    bool valid; // 有效标志位, 标识是否被初始化
    bool readOnly; // 只读标志位 标识对该页表项的修改权限
    bool use; // 使用标志位, 标识该页表项是否被使用
    bool dirty; // 脏位, 标识该页表项是否被修改过
};

```

## 地址转换过程解析

阅读 machine / translate.cc 里面 Translate 函数来体会地址转化过程

```
//-----
--  

// 将虚拟地址转换为物理地址，使用  

// 页表或 TLB。检查对齐和各种  

// 其他错误，如果一切正常，设置使用/脏位  

// 转换表条目，并存储转换后的物理  

// “physAddr”中的地址。如果有错误，返回类型  

// 异常。  

//  

// "virtAddr" -- 要转换的虚拟地址  

// "physAddr" -- 存放物理地址的地方  

// "size" -- 正在读取或写入的内存量  

// "writing" -- 如果为 TRUE，检查 TLB 中的 "read-only" 位  

//-----  

--  

ExceptionType  

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

{  

    int i;  

    unsigned int vpn, offset;  

    TranslationEntry *entry;  

    unsigned int pageFrame;  

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

"read");  

    // 检查是否对齐  

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

0x1))) {  

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

size);  

        return AddressErrorException;  

    }  

    // 检查 tlb 以及页表是否同时存在，如果同时存在则异常退出  

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

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

    // 根据虚拟地址和页面大小计算虚拟页号以及偏移量
```

```

vpn = (unsigned) virtAddr / PageSize;
offset = (unsigned) virtAddr % PageSize;

// 若 TLB 不存在 则检查页表
if (tlb == NULL) {
    // 如果虚页号大于页表长度，则抛出地址错误的异常
    if (vpn >= pageTableSize) {
        DEBUG('a', "virtual page # %d too large for page table
size %d!\n",
              virtAddr, pageTableSize);
        return AddressErrorException;
    } // 如果虚页号对应的页表项无效，则抛出 PageFaultException 异常
    else if (!pageTable[vpn].valid) {
        DEBUG('a', "virtual page # %d too large for page table
size %d!\n",
              virtAddr, pageTableSize);
        return PageFaultException;
    }
    // 令 entry 为对应的页表项
    entry = &pageTable[vpn];
} else {
    // 如果 TLB 存在 遍历 TLB 查找对应的页表项 如果页表项存在记录到 entry
    for (entry = NULL, i = 0; i < TLBSIZE; i++)
        if (tlb[i].valid && ((unsigned int)tlb[i].virtualPage ==
vpn)) {
            entry = &tlb[i];           // FOUND!
            break;
        }
    // 如果没找到 抛出 PageFaultException
    if (entry == NULL) {           // not found
        DEBUG('a', "*** no valid TLB entry found for this virtual
page!\n");
        return PageFaultException; // really, this is a TLB
fault,
                // the page may be in memory,
                // but not in the TLB
    }
}

// 如果要 write 写页表项 要检查 readOnly 抛出 ReadOnlyException
if (entry->readOnly && writing) { // trying to write to a read-
only page
    DEBUG('a', "%d mapped read-only at %d in TLB!\n", virtAddr, i);
}

```

```

        return ReadOnlyException;
    }
    // 获得实页号
    pageFrame = entry->physicalPage;

    // 检查 PageFrame 是否越界
    if (pageFrame >= NumPhysPages) {
        DEBUG('a', "*** frame %d > %d!\n", pageFrame, NumPhysPages);
        return BusErrorException;
    }
    // 修改标志位
    entry->use = TRUE;      // set the use, dirty bits
    if (writing)
        entry->dirty = TRUE;
    // 得到物理地址
    *physAddr = pageFrame * PageSize + offset;
    ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
    DEBUG('a', "phys addr = 0x%x\n", *physAddr);
    return NoException;
}

```

1. 根据 size 与 virtAddr 的值检查对齐错误，若没有对齐则抛出 AddressErrorException 异常；
2. 检查 TLB 或页表是否同时存在，若同时存在则异常退出；
3. 根据虚拟地址与页面大小计算虚拟页号及偏移量
4. 若 ‘TLB 不存在则检查页表，若虚拟页号大于页表长度，则抛出 AddressErrorException 异常，若虚拟页号所对应的页表项无效，则抛出 PageFaultException 异常，置 entry 为对应的页表项；若 TLB 存在，则遍历 TLB 来查找对应的页表项，若页表项存在则记录到 entry 中，不存在则抛出 PageFaultException 异常；
5. 若 writing 为 TRUE，则要检查对应的页表项是否支持写操作，若该页表项的权限为只读，即 readOnly 位为 TRUE，则抛出 ReadOnlyException 异常；
6. 由 entry 的 physicalPage 获得实页号 pageFrame，然后进一步检查 pageFrame 是否越界，若越界则抛出 BusErrorException 异常；
7. 修改相应标志位，最后计算物理地址完成转换，返回 NoException，没有发生异常，转换完成。

## 应用程序进程的创建与启动

1. 首先 Nachos -x 参数 调用 startProcess 函数，为用户程序创建相应的进程，并且启动该进程的执行。、

2. 阅读..//userprog/ progtest.cc 的 void StartProcess(char \*filename)函数：

1) 打开.noff 可执行文件

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

2) 为程序分配内存空间，将用户程序装入所分配的内存空间，创建页表并且建立虚页和实页的映射关系

```
space = new AddrSpace(executable);
```

3) 将用户进程 映射到 一个核心进程

```
currentThread->space = space;
```

4) 初始化寄存器

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

5) 将用户进程的页表 装载进内存中

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

6) 开始用户进程的执行

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

系统要运行一个应用程序，需要：

1. 为该程序创建一个用户进程，
2. 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；
3. 用户进程映射到核心线程
4. 调度执行该线程，根据用户进程页表读取用户进程指令

下面详细说明和分析

## Nachos 地址空间分配

Nachos 进程地址空间分配的实现主要在 AddressSpace::AddressSpace()中,该过程主要包括: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，建立虚页与实页的映射关系。

AddressSpace 执行过程

1. 读入可执行文件的文件头 noff
2. 判断是不是 noff 文件 (看魔数)
3. 计算地址空间的大小=code + initData + uninitData + UserStackSize
4. 根据 size 计算所需的页数
5. 根据页 重新计算 size (内碎片)
6. 检查越界 不能太大、
7. 首先新建页表 进行转化
8. 将整个地址空间清零

## 9. 然后 复制代码和数据段载入内存

代码

```
// -----
// addrSpace :: addrSpace
// 创建一个地址空间以运行用户程序。
// 从文件中加载程序“可执行”，并设置所有内容
// up 以便我们可以开始执行用户说明。
//
// 假设对象代码文件处于 noff 格式。
//
// 首先，设置从程序内存到物理的翻译
// 记忆。现在，这真的很简单（1: 1），因为我们是
// 只有 uniproal mangling，我们有一个单一的未分段页表
//
// “可执行”是包含要加载到内存的对象代码的文件
// -----
AddrSpace::AddrSpace(OpenFile *executable)
{
    NoffHeader noffH;
    unsigned int i, size;

    // 读入可执行文件 的文件头 noff
    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    // 判断是不是 noff 文件 (看魔数)
    if ((noffH.noffMagic != NOFFMAGIC) &&
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);

    // 计算地址空间的大小=code + initData + uninitData + UserStackSize
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
          + UserStackSize;      // we need to increase the size
                                // to leave room for the stack
    // 根据 size 计算所需的页数
    numPages = divRoundUp(size, PageSize);
    // 根据页 重新计算 size (内碎片)
    size = numPages * PageSize;
    // 检查越界 不能太大
    ASSERT(numPages <= NumPhysPages);      // check we're not trying
                                            // to run anything too big --
                                            // at least until we have
                                            // virtual memory
```

```

DEBUG('a', "Initializing address space, num pages %d, size %d\n",
      numPages, size);
// 首先新建页表 进行转化
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; // 如果代码段在一个单独分开的 page,
可以设置为 ReadOnly
}

// 将整个地址空间清 0
bzero(machine->mainMemory, size);

// 然后 复制代码和数据段载入内存
if (noffH.code.size > 0) {
    DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
          noffH.code.virtualAddr, noffH.code.size);
    executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]
),
                      noffH.code.size, noffH.code.inFileAddr);
}
if (noffH.initData.size > 0) {
    DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
          noffH.initData.virtualAddr, noffH.initData.size);
    executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualA
ddr]),
                      noffH.initData.size, noffH.initData.inFileAddr);
}

}

```

## Nachos 核心线程执行用户程序的原理和方法

在 startProcess 里面，分配完地址空间之后，将用户进程映射到一个核心线程

```

space = new AddrSpace(executable);
currentThread->space = space;

```

为使该核心线程能够执行用户进程的代码，需要核心在调度执行该线程时，根据用户进程的页表读取用户进程指令；因此需要将用户页表首地址传递给核心的地址变换机构；即

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

如下所示，kernel 在调度执行该线程的时候，首先根据用户进程的页表读取用户进程指令，然后对指令进行译码并且执行，为此 Nachos 封装了 instruction 类，

核心线程执行用户程序具体流程如下：

1. 传递用户进程页表给 Machine

```
void AddrSpace::RestoreState()
{
    machine->pageTable = pageTable;
    machine->pageTableSize = numPages;
}
```

2. 初始化寄存器

- 1) 首先 Nachos 里面有两种寄存器，第一种是 Machine 里面定义的 CPU 使用的寄存器

```
int registers[NumTotalRegs]; // CPU registers, for executing user
programs
```

- 2) 在 thread.h 里面有为用户进程设计的

```
int userRegisters[NumTotalRegs]; // user-level CPU register state
```

- 3) 运行用户程序的线程实际上有 2 组 CPU 寄存器 一个用于执行用户代码时的状态，一个用于其状态在执行内核代码时。

并且因为 Nachos 只有一个 CPU，所以只有一套 CPU 寄存器，但是每个核心线程都可能执行用户进程，因此相应的对每个核心线程都设置了一组用户寄存器，用于保存与回复用户程序指令的执行状态。

```
#ifdef USER_PROGRAM
// A thread running a user program actually has *two* sets of CPU
registers --
// one for its state while executing user code, one for its state
// while executing kernel code.

    int userRegisters[NumTotalRegs]; // user-level CPU register state

public:
    void SaveUserState(); // save user-level register state
    void RestoreUserState(); // restore user-level register state

    AddrSpace *space; // User code this thread is running.
#endif
```

4) 如果涉及到用户进程进行进程调度时，上下文切换中，会把旧进程的 CPU 寄存器状态保存在用户寄存器中，新的用户进程的寄存器状态保存在 CPU 寄存器中，下次 CPU 就直接能够执行用户程序。(可能是上次终端的用户进程) 正如 InitRegister 的注释所说

```
// AddrSpace::InitRegisters
// 设置用户级寄存器集的初始值。
//
// 我们将这些直接写入“机器”寄存器，所以
// 我们可以立即跳转到用户代码。请注意，这些
// 将被保存/恢复到 currentThread->userRegisters
// 当这个线程被上下文切换出去时。

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

3. Instruction 类封装了一条 Nachos 机器指令

```
class Instruction {
public:
    void Decode(); // 解码指令
    unsigned int value; // 指令的二进制表达值
    char opCode; // 指令类型
    char rs, rt, rd; // 指令的三个寄存器
    int extra; // 立即数或者目标或者偏移量
};
```

4. Run, 中读取指令

在 Machine run 里面调用

```
OneInstruction(instr);
```

在 Machine OneInstruction 里面读取内存来得到指令

```
machine->ReadMem(registers[PCReg], 4, &raw)
```

```
bool Machine::ReadMem(int addr, int size, int *value)
```

1) 调用 Translate 将虚拟地址转化为物理地址

- 2) 判断异常  
 3) size 可能为 1 2 4 如果时 1 直接读入 data 赋值 value, 否则需要进行大小端的转化后才能赋值。  
 4) WordToHost ShortToHost, 转化成 Nachos 所需的小端格式, 如果时 DEC 和 Intel 则 同样是小端。  
 5. 指令译码执行  
 1) 读出指令
- ```
// Fetch instruction
if (!machine->ReadMem(registers[PCReg], 4, &raw))
    return; // exception occurred
```
- 2) 对指令译码
- ```
instr->value = raw;
instr->Decode();
```
- 3) 计算下一个 PC 值
- ```
// Compute next pc, but don't install in case there's an error or
branch.
int pcAfter = registers[NextPCReg] + 4;
```
- 4) 根据译码的操作码来执行具体指令

```
// Execute the instruction (cf. Kane's book)
switch (instr->opCode) {

    case OP_ADD:
        sum = registers[instr->rs] + registers[instr->rt];
        if (!(registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT) &&
            ((registers[instr->rs] ^ sum) & SIGN_BIT)) {
            RaiseException(OverflowException, 0);
            return;
        }
        registers[instr->rd] = sum;
        break;

    case OP_ADDI:
        sum = registers[instr->rs] + instr->extra;
        if (!(registers[instr->rs] ^ instr->extra) & SIGN_BIT) &&
            ((instr->extra ^ sum) & SIGN_BIT)) {
            RaiseException(OverflowException, 0);
            return;
        }
}
```

- 5) 执行完指令, 进行延迟加载
- ```
// Do any delayed load operation
DelayedLoad(nextLoadReg, nextLoadValue);
```
- 6) 更新 PC 等寄存器的值, 为取下一条指令准备
- ```
// Advance program counters.
```

```

    registers[PrevPCReg] = registers[PCReg]; // for debugging, in
case we
                                // are jumping into lala-land
    registers[PCReg] = registers[NextPCReg];
    registers[NextPCReg] = pcAfter;

```

6. 一直循环执行程序指令，更新时钟周期，

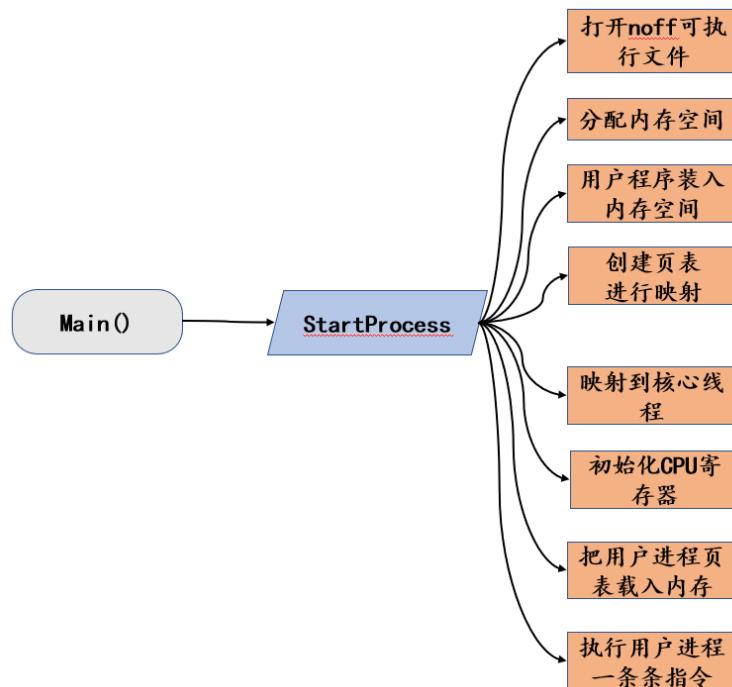
```

for (;;) {
    OneInstruction(instr);
    interrupt->OneTick();
    if (singleStep && (runUntilTime <= stats->totalTicks))
        Debugger();
}

```

直到执行完毕 或者在 OneInstruction 中抛出异常

## Nachos 用户程序的创建与启动过程



## Nachos 进程控制块 PCB

1. Kernel 线程不需要单独分配内存
2. Fork 的子线程 需要调用 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()）；

3. 至于 PCB，真实的操作系统首先为应用程序分配一个 PCB，存放应用程序进程的相应信息：包括进程号 pid、页表、堆栈、文件描述符等
4. Nachos 没用显式定义 PCB，ddrSpace 类体现了 PCB 的概念，但信息不是很完整。但是有类似的思想：利用所分配内存空间的对象指针来标识一个进程，该对象含有进程的页表、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，依据线程所保存的上下文来执行对应的用户进程。
5. 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；

## Nachos 线程调度算法

Nachos 默认的线程调度算法是 FCFS。可以使用 Yield 模拟抢先机制。使用 RS 进行定时中断实现轮转法 RR 调度。

## 部分源码

1. 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%d\n", pageTable[i].virtualPage,
               pageTable[i].physicalPage);
    }
    printf("=====\n\n");
}
```

2. 分配更大的地址空间

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()
{
    int i,j,k;
    k=3;
    i=2;
    j=j-1;
    k=i-j+k;
    Halt();
    /* not reached */
}
```

### 3. 大小端转换函数

为了适应不同机型的存储方式，Nachos 提供了大小端转换的函数，支持 word 和 short

```
unsigned int
WordToHost(unsigned int word) {
#ifdef HOST_IS_BIG_ENDIAN
    register unsigned long result;
    result = (word >> 24) & 0x000000ff;
    result |= (word >> 8) & 0x0000ff00;
    result |= (word << 8) & 0x00ff0000;
    result |= (word << 24) & 0xff000000;
    return result;
#else
    return word;
#endif /* HOST_IS_BIG_ENDIAN */
}

unsigned short
ShortToHost(unsigned short shortword) {
#ifdef HOST_IS_BIG_ENDIAN
    register unsigned short result;
    result = (shortword << 8) & 0xff00;
    result |= (shortword >> 8) & 0x00ff;
    return result;
#else
    return shortword;
#endif /* HOST_IS_BIG_ENDIAN */
}
```

### 4. ReadMem

首先根据页表进行虚实地址转换，然后根据物理地址读取一条指令

```
bool
Machine::ReadMem(int addr, int size, int *value)
```

```

{
    int data;
    ExceptionType exception;
    int physicalAddress;

    DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);

    exception = Translate(addr, &physicalAddress, size, FALSE);
    if (exception != NoException) {
        machine->RaiseException(exception, addr);
        return FALSE;
    }
    switch (size) {
        case 1:
            data = machine->mainMemory[physicalAddress];
            *value = data;
            break;

        case 2:
            data = *(unsigned short *) &machine->mainMemory[physicalAddress];
            *value = ShortToHost(data);
            break;

        case 4:
            data = *(unsigned int *) &machine->mainMemory[physicalAddress];
            *value = WordToHost(data);
            break;

        default: ASSERT(FALSE);
    }

    DEBUG('a', "\tvalue read = %8.8x\n", *value);
    return (TRUE);
}

```

## 5. MIPS 指令格式

R型，寄存器型

| opcode | rs    | rt    | rd    | shamt | funct |
|--------|-------|-------|-------|-------|-------|
| 6-bit  | 5-bit | 5-bit | 5-bit | 5-bit | 6-bit |

R型指令

R型指令格式包含6个域，最高位的opcode域，是6个比特，最低位的funct域也是6个比特，中间rs、rt、rd、shamt均为5个比特。

opcode用于指定指令的类型，对于所有R型指令，该域的值均为0，但这并不是说明R型指令只有一种，它还需要用funct域来更为精确的指定指令的类型，所以说对于R型指令，实际上一共有12个比特操作码。

rs Source Register 通常用于指定第一个源操作数所在的寄存器编号

rt Target Register 通常用于指定第二个源操作数所在的寄存器编号

rd Destination Register 通常用于指定目的操作数（保存运算结果）的寄存器编号

5-bit的域可表示0~31，对应32个通用寄存器

shamt shift amount 用于指定移位指令进行移位操作的位数，对于非移位指令，该域设为0。

## I型指令 条件分支

I 格式

|    |    |    |       |
|----|----|----|-------|
| 6  | 5  | 5  | 16    |
| op | rs | rt | 立即数操作 |

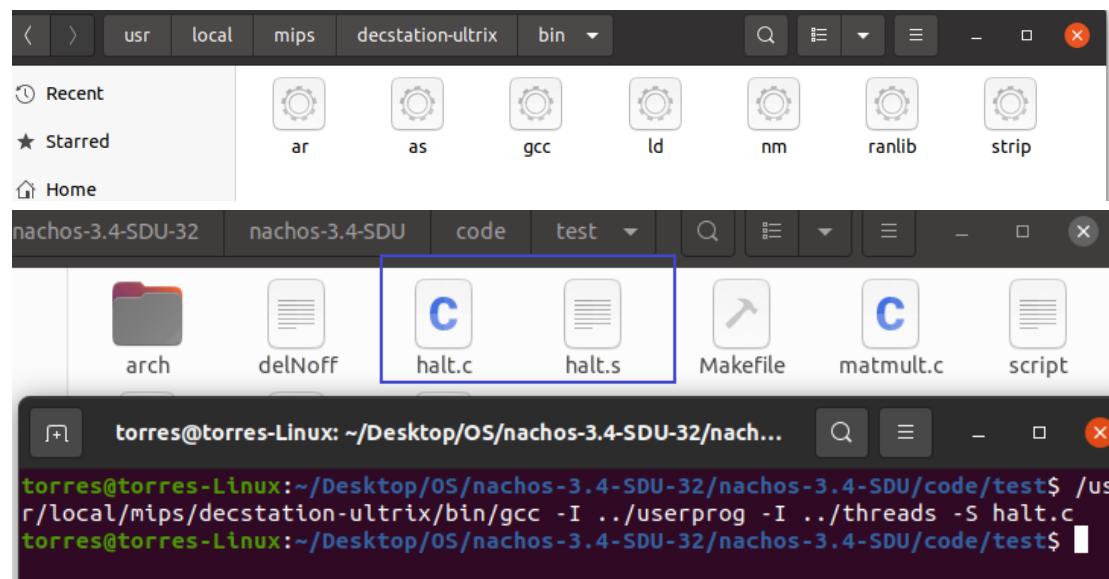
## J型 跳转指令

| opcode | address |
|--------|---------|
| 6-bit  | 26-bit  |

## 测试结果

### 测试 Nachos 应用程序与执行过程

- 在./test 目录中将文件 halt.c 的修改
- 在./test 目录中重新编译生成新的 halt
- ./usr/local/mips/decstation-ultrix/bin/gcc -I ..//userprog -I ..//threads -S halt.c



生成了 half.s

#### 4. 查看 half.s

main:

```
.frame $fp,40,$31      # vars= 16, regs= 2/0, args= 16, extra= 0
.mask 0xc0000000,-4
.fmask 0x00000000,0
subu $sp,$sp,40  # 创建栈指针
sw $31,36($sp)
sw $fp,32($sp)
move $fp,$sp
jal __main
li $2,3          # 0x00000003
sw $2,24($fp)
li $2,2          # 0x00000002
sw $2,16($fp)
lw $2,20($fp)
addu $3,$2,-1
sw $3,20($fp)
lw $2,16($fp)
lw $3,20($fp)
subu $2,$2,$3
lw $3,24($fp)
addu $2,$3,$2
sw $2,24($fp)
jal Halt
```

\$L1:

```
move $sp,$fp
lw $31,36($sp)
lw $fp,32($sp)
addu $sp,$sp,40  # 撤销栈指针
j $31
.endmain
```

#### 5. 进入目录..../userprog

- 1). Make 编译生成 Nachos 内核
- 2). 命令 ./nachos -x .../test/half.noff

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/userprog$ ./nachos -x ../test/halt.noff
*** 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
Machine halting!
```

3). ./nachos -d m -x .../test/halt.noff

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/userprog$ ./nachos -d m -x ../test/halt.noff
*** 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
Starting thread "main" at time 120
At PC = 0x0: JAL 52
At PC = 0x4: SLL r0,r0,0
At PC = 0xd0: ADDIU r29,r29,-40
At PC = 0xd4: SW r31,36(r29)
At PC = 0xd8: SW r30,32(r29)
At PC = 0xdc: JAL 48
At PC = 0xe0: ADDU r30,r29,r0
At PC = 0xc0: JR r0,r31
At PC = 0xc4: SLL r0,r0,0
At PC = 0xe4: ADDIU r2,r0,3
At PC = 0xe8: SW r2,24(r30)
At PC = 0xec: ADDIU r2,r0,2
At PC = 0xf0: SW r2,16(r30)
At PC = 0xf4: LW r2,20(r30)
At PC = 0xfb: SLL r0,r0,0
At PC = 0xfc: ADDIU r3,r2,-1
At PC = 0x100: SW r3,20(r30)
At PC = 0x104: LW r2,16(r30)
At PC = 0x108: LW r3,20(r30)
At PC = 0x10c: SLL r0,r0,0
At PC = 0x110: SUBU r2,r2,r3
At PC = 0x114: LW r3,24(r30)
At PC = 0x118: SLL r0,r0,0
At PC = 0x11c: ADDU r2,r3,r2
At PC = 0x120: JAL 4
At PC = 0x124: SW r2,24(r30)
At PC = 0x10: ADDIU r2,r0,0
At PC = 0x14: SYSCALL
Exception: syscall
Machine halting!
```

输出了每一步执行的机器指令

4). ./nachos -d m -s -x .../test/halt.noff 该命令实现了单步调试，每执行一步就会暂停并输出当前所有寄存器的结果

```

Machine registers:
 0: 0x0   1: 0x0   2: 0x0   3: 0x0
 4: 0x0   5: 0x0   6: 0x0   7: 0x0
 8: 0x0   9: 0x0  10: 0x0  11: 0x0
12: 0x0  13: 0x0  14: 0x0  15: 0x0
16: 0x0  17: 0x0  18: 0x0  19: 0x0
20: 0x0  21: 0x0  22: 0x0  23: 0x0
24: 0x0  25: 0x0  26: 0x0  27: 0x0
28: 0x0 SP(29): 0x570 30: 0x0 RA(31): 0x8
Hi: 0x0 Lo: 0x0
PC: 0x4 NextPC: 0xd0 PrevPC: 0x0
Load: 0x0 LoadV: 0x0

121>
At PC = 0x4: SLL r0,r0,0
Time: 122, interrupts on
Pending interrupts:
End of pending interrupts
Machine registers:
 0: 0x0   1: 0x0   2: 0x0   3: 0x0
 4: 0x0   5: 0x0   6: 0x0   7: 0x0
 8: 0x0   9: 0x0  10: 0x0  11: 0x0
12: 0x0  13: 0x0  14: 0x0  15: 0x0
16: 0x0  17: 0x0  18: 0x0  19: 0x0
20: 0x0  21: 0x0  22: 0x0  23: 0x0
24: 0x0  25: 0x0  26: 0x0  27: 0x0
28: 0x0 SP(29): 0x570 30: 0x0 RA(31): 0x8
Hi: 0x0 Lo: 0x0
PC: 0xd0 NextPC: 0xd4 PrevPC: 0x4
Load: 0x0 LoadV: 0x0

122>

```

## 测试 Nachos 分页

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，0-10）

```

torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/userprog$ ./nachos -x ./test/halt.noff
*** 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
page table dump: 11 pages in total
=====
  VirtPage,      PhysPage
    0,          0
    1,          1
    2,          2
    3,          3
    4,          4
    5,          5
    6,          6
    7,          7
    8,          8
    9,          9
   10,         10
=====
Machine halting!

```

## 结论体会

1. 理解用户页表与 CPU 页表的关系，虚拟地址与物理地址如何进行转换，如何标记一个用户程序，如何用核心线程去映射用户进程以及 CPU 执行指令的具体过程
2. 用户程序的创建：Nachos 中用 AddrSpace 类来标记一个用户进程，在该类初始化的过程中，先是对于类中页表进行一个虚页与实页映射，然后清空 Machine 中原有物理内存数据，最后将 noff 文件的内容拷贝到 Machine 物理内存中。至此即完成了用户程序地址空间的创建
3. Machine 初始化：创建完用户程序的地址空间之后，就先通过给核心线程的地址空间赋值来将用户进程映射到核心线程上，之后再给 Machine 类的 PC、栈指针寄存器赋值完成 Machine 的初始化。
4. 用户进程执行：程序的执行过程即是 Machine 根据其页表不断将 PC 寄存器中所存储的虚拟地址转换为 Machine 的物理地址，然后从物理地址中取出指令，最后执行指令并且  $PC += 4$ 。W

## 实验问题回答

1. Nachos 可执行程序的结构是怎样的？

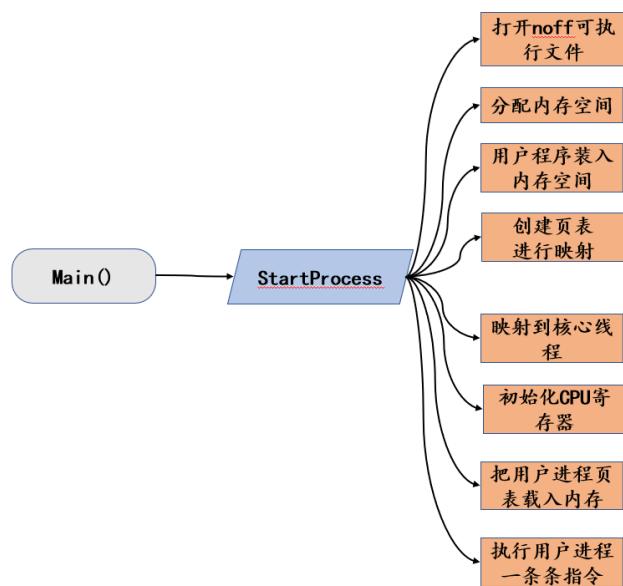
```
#define NOFFMAGIC 0xbadfad /* magic number denoting Nachos
                           * object code file
                           */

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;

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;
```

Nachos 可执行文件的文件头使用魔数 0xbadfad 来进行标识，然后又存储了代码段、初始化数据段、未初始化数据段的大小以及地址信息，Nachos 系统根据该文件头的内容来进行后续的进程创建与执行。

2. Nachos 用户进程的创建与执行过程是怎样的？



# 实验 7 地址空间的拓展

## 实验信息

姓名: XXX 学号:xxxxxxxxxxxxx 日期: 2022.4.27 班级: 19.1

## 实验目的

通过考察系统加载应用程序过程，如何为其分配内存空间、创建页表并建立虚页与实页帧的映射关系，理解 Nachos 的内存管理方法；  
理解如何系统对空闲帧的管理；  
理解如何加载另一个应用程序并为其分配地址空间，以支持多进程机制；  
理解进程的 pid；  
理解进程退出所要完成的工作；

## 实验任务

该实验中，你需要完成：

- (1) 阅读..//prog/progtest.cc，深入理解 Nachos 创建应用程序进程的详细过程
- (2) 阅读理解类 AddrSpace，然后对其进行修改，使 Nachos 能够支持多进程机制，允许 Nachos 同时运行多个用户线程；
- (3) 在类 AddrSpace 中添加完善 Print() 函数（在实验 6 中已经给出）
- (4) 在类 AddrSpace 中实例化类 Bitmap 的一个全局对象，用于管理空闲帧；
- (5) 如果将 SpaceId 直接作为进程号 Pid 是否合适？如果感觉不是很合适，应该如何为进程分配相应的 pid？
- (6) 为实现 Join(pid)，考虑如何在该进程相关联的核心线程中保存进程号；
- (7) 根据进程创建时系统为其所做的工作，考虑进程退出时应该做哪些工作；
- (8) 考虑系统调用 Exec() 与 Exit() 的设计实现方案；
- (9) 拓展：可以进一步考虑如何添加自己所需要的系统调用，即..//userprog/syscall.h 中没有定义的系统调用，如 Time，以获取当前的系统时间。

# 代码与原理分析

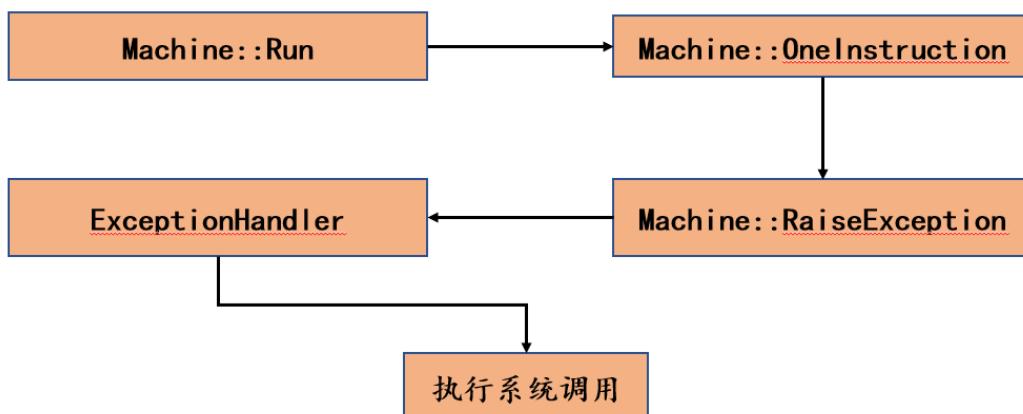
## Nachos 只支持单进程原因

如果执行一个 bar.off 应用程序，里面进行了系统调用，exec (./test/exec.noff)，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，但是因为在 AddrSpace::AddrSpace(OpenFile \*executable)中的程序片段中，Nachos 系统加载一个应用程序并为其分配内存时，总是将程序的第 0 号页面分配到内存的第 0 号帧，第 1 号页面分配到内存的第 1 号帧，...，以此类推。

因此，Nachos 在运行其应用程序./test/bar.noff 时，内存从的第 0 号开始的几个帧已经分配给程序./test/bar.noff，如果执行到语句 Exec("./test/exec.noff") 加载./test/exec.noff 时，Nachos 仍然将从第 0 号开始的几个帧分配给./test/exec.noff，导致系统为./test/bar.noff 所分配的内存空间被./test/exec.noff 覆盖，程序显然无法正确执行。

## Nachos 系统调用流程

如图所示：



1. Nachos 的程序执行过程，即 `Machine::Run`，会把程序的指令一条条取出并执行，也就是 `Machine::OneInstruction`。
2. 在 `Machine::OneInstruction` 里面，Nachos 对指令进行译码，如果发现是 `OP_SYSCALL`，系统调用时，会抛出一个 `SyscallException` 的异常，然后陷入内核态进行处理。
3. 即使用 `ExceptionHandler` 进行异常处理，里面应该定义对于不同的系统调用不同的处理方式。

## Nachos 系统调用入口

分析/test/start.s，来分析 Nachos 的系统调用，以 exec 为例

```
Exec:
    addiu $2,$0,SC_Exec
    syscall
    j    $31
    .end Exec
```

1. 将系统调用类型放入二号寄存器
2. 执行系统调用
3. 返回到 31 寄存器存放的地址(原程序的返回地址)

对 exec.s 进行分析

```
.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                      # 用于 2 字节对齐
    .globl main                   # main 全局变量
    .ent   main
main:
    # frame 用来声明堆栈布局:
    # $fp: 用于访问局部堆栈读寄存器
    # 32: 该函数已分配的堆栈读大小
    # $31: 用来保存返回地址的寄存器
    .frame $fp,32,$31      # vars= 8, regs= 2/0, args= 16, extra= 0
    .mask  0xc0000000,-4
    .fmask 0x00000000,0

    subu $sp,$sp,32      # $sp - 32 -> $sp, 构造 main 的栈, 栈指针指向栈最小的地址
                            # $sp 的原值应该是执行 main()之前的栈
                            # 上一函数对应栈 frame 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 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, main()函数返回
# $31: Return address for subrouting

.end    main

```

也就是寄存器 R2 用于系统调用，保存系统调用代码区地址，然后四个参数 arg1-4 分别放在寄存器 R4—R7 中。

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。

## Nachos 系统调参数传递

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。

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 $4,$LCO` 将 `Exec("../test/halt.noff")` 中的参数 `"../test/halt.noff"` 在内存中的地址 `$LCO` 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。

1. 在 `exec` 代码中，利用 `Machine::ReadRegister(4)` 从 `$4` 中获取 `"../test/halt.noff"` 的内存地址，利用 `Machine::ReadMem(...)` 获取 `FileName` `"../test/halt.noff"`
2. 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。
3. 当 `"../test/halt.noff"` 执行结束后，需要返回该线程的 `pid`

## AdvancePC

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。

而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。

## SpaceID

首先，Exec()返回类型为 SpaceId 的值，这里就是把 SpaceId 作为 Pid 来对待，唯一标识进程，需要解决两个问题

- (1) 如何产生 SpaceId；
- (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；
- (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。

## Pid

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。

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。

---

# 设计与实现

## 地址空间的拓展

根据实验要求，通过在 AddrSpace 里面引入 BitMap 类来实现地址空间的拓展方法如下

1. 引入 BitMap 来管理内存中的空闲帧，在 AddrSpace 里面加入私有成员，因为要求全局变量，所以设为 static

Private:

```
static BitMap *PageBitmap;
```

2. 在构造函数: AddrSpace::AddrSpace(OpenFile \*executable) 里面进行 PagebitMap 的初始化，要控制虚拟地址向物理地址的映射，所以 Bitmap 来管理物理分页。

```
// add codes to Init PageBitmap
PageBitmap = new BitMap(NumPhysPages);
```

3. 在进行 translation 时候，找到空闲物理页进行分配。

```
pageTable = new TranslationEntry[numPages];
for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
    pageTable[i].physicalPage = PageBitmap->Find(); // Find Empty
    Page to Process
    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
}
```

4. 修改主存寻址方式，原来 ReadAt 直接访问的 MainMemory 是按照虚拟地址访存的，因为之前 PageTable 默认 VA = PA，但是现在 VA != PA，所以需要按照实际的物理地址访存主存

原来：

```
// then, copy in the code and data segments into memory
if (noffH.code.size > 0) {
    DEBUG('a', "Initializing code segment, at 0x%0x, size %d\n",
          noffH.code.virtualAddr, noffH.code.size);
    // Because VA = PA, so it use VA to find PA
```

```

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

if (noffH.initData.size > 0) {
    DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
          noffH.initData.virtualAddr, noffH.initData.size);

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

```

修改之后，现在

```

// then, copy in the code and data segments into memory
if (noffH.code.size > 0) {
    // the num in PageTable
    int phynum = pageTable[noffH.code.virtualAddr/PageSize];
    int offset = noffH.code.virtualAddr % PageSize;
    DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
          phynum + offset, noffH.code.size);
    executable->ReadAt(&(machine->mainMemory[phynum + offset]),
                        noffH.code.size, noffH.code.inFileAddr);
}

if (noffH.initData.size > 0) {
    int phynum = pageTable[noffH.initData.virtualAddr/PageSize];
    int offset = noffH.initData.virtualAddr % PageSize;
    DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
          phynum + offset, noffH.initData.size);
    executable->ReadAt(&(machine->mainMemory[phynum + offset]),
                        noffH.initData.size, noffH.initData.inFileAddr);
}

```

5. AddrSpace 析构函数中，释放空闲帧

```

AddrSpace::~AddrSpace(){
    for(int i=0;i<numPages;i++)
        PageBitmap->Clear(pageTable[i].physicalPage);
    delete [] pageTable;
}

```

## Exec 系统调用实现

仅进行地址空间的扩展，当然还是无法进行 exec 系统调用的，还需要实现 Exec 系统调用。

- 在 userprog/exception.cc 里面的 void ExceptionHandler(ExceptionType which) 中添加 Exec() 的系统调用处理代码，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 args
                char filename[128];
                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');
                // open file
                OpenFile *executable = fileSystem->Open(filename);
                if(!executable){
                    printf("Unable to open file %s\n",filename);
                    return ;
                }
                // Apply for address space
                AddrSpace *space = new AddrSpace(executable);
                // print
                space->Print();
                delete executable;
                // create kernel process
                Thread *thread = new Thread(filename);
            }
        }
    }
}
```

```

        printf("New Thread SpaceID: %d,
Name: %s\n",space->getSpaceId(),filename);
        thread->Fork(StartProcess,(int)space->getSpaceId());
        // user process -> kernel process
        thread->space = space;
        machine->WriteRegister(2,space->getSpaceId());
        AdvancePC();
        break;
    }
    default:{
        printf("Unexpected user mode exception %d %d\n", which,
type);
        ASSERT(FALSE);
        break;
    }
}
} else {
    printf("Unexpected user mode exception %d %d\n", which, type);
    ASSERT(FALSE);
}
}

```

2. 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()

```

void AdvancePC(){
    machine->WriteRegister(PrevPCReg, machine->ReadRegister(PCReg));
    machine->WriteRegister(PCReg, machine->ReadRegister(NextPCReg));
    machine->WriteRegister(NextPCReg, machine->ReadRegister(NextPCReg) +
4);
}

```

3. 重载 Startprocess(char\* filename)使得当调度到用户进程所关联的核心线程时，能够 exec 中的 filename 对应的进程可以立即执行

```

void StartProcess(int spaceId){
    currentThread->space->InitRegisters();
    currentThread->space->RestoreState();
    machine->Run();
    ASSERT(FALSE);
}

```

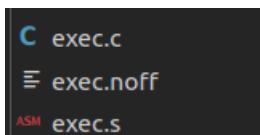
# 测试结果

## 测试 Exec 以及地址空间拓展

- 先添加 exec.c 以及 编译连接成 exec.noff

```
/usr/local/mips/decstation-ultrix/bin/gcc -I .. /userprog -I .. /threads -S exec.c
```

- 或者 Makefile 里面修改 targets targets = halt shell matmult sort exec



- 执行 exec.noff 里面会进行 exec 的系统调用

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/userprog$ ./nachos -x .. /test/exec.noff
*** 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
page table dump: 11 pages in total
=====
    VirtPage,      PhysPage
    0,          0
    1,          1
    2,          2
    3,          3
    4,          4
    5,          5
    6,          6
    7,          7
    8,          8
    9,          9
    10,         10
=====
```

```
Execute system call of Exec()
page table dump: 11 pages in total
=====
    VirtPage,      PhysPage
    0,          11
    1,          12
    2,          13
    3,          14
    4,          15
    5,          16
    6,          17
    7,          18
    8,          19
    9,          20
    10,         21
=====

New Thread SpaceID: 101, Name: .. /test/halt.noff
```

发现成功分配用户进程空间，成功执行了系统调用

## 结论体会

1. 本实验以及实验 8，对于 pid 的解决方案是使用 SpaceID 来唯一标识一个进程。
2. 实验七主要是对于地址空间类的扩展，改进了原有的一对一直接分配的虚实页关系，而是引入了一个静态全局位图来标记维护哪些页被存储了哪些页空闲，进而在页面分配上实现了多用户程序运行的条件。
3. 在进行页面分配之前要确认位图中是否有足够的页面用于页面分配，在地址空间的析构函数中将分配好的页面清空，避免资源浪费。
4. exec() 系统调用的实现：采取 StartProcess 的实现思路来将用户进程映射到核心线程并分配 SpaceId 来完成该系统调用，引入了 SpaceId 来标记一个地址空间，并用该地址空间来标记一个线程，类似于 pid 的功能。

# 实验 8 系统调用 Exec() 与 Exit()

## 实验信息

姓名: XXX 学号:XXXXXXXXXXXX 日期: 2022.5.10 班级: 19.1

## 实验目的

理解用户进程是如何通过系统调用与操作系统内核进行交互的；  
理解系统调用是如何实现的；  
理解系统调用参数传递与返回数据的回传机制；  
理解核心进程如何调度执行应用程序进程；  
理解进程退出后如何释放内存等为其分配的资源；  
理解进程号 pid 的含义与使用；

## 实验任务

- (1) 阅读..../userprog/exception.cc，理解系统调用 Halt() 的实现原理；
- (2) 基于实现 6、7 中所完成的工作，利用 Nachos 提供的文件管理、内存管理及线程管理等功能，编程实现系统调用 Exec() 与 Exit()（至少实现这两个）。
- (3) 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）。

## 代码与原理分析

### 编写自己的 Nachos 应用程序

- (1) 按照类 C 语言的语法以及 Nachos 所提供的系统调用，在目录..../test 中编写程序 exec.c 并保存到..../test 目录中。
- (2) 将 exec.c 交叉编译并转换成 Nachos 模拟的 CPU 可执行程序 exec.noff
  - a) 将 exec 添加到..../test/Makefile 文件的 targets 列表中
  - b) 运行 make，生成 exec.noff

## Nachos 系统调用参数传递

以 exit 为例：

```
#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 Exec() 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。

## 用户进程的 Openfile

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。

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：

1. 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。
2. 将该进程映射到一个核心线程 currentThread->space = space;，初始化寄存器，运行它。

这里需要注意

1. 进程的 PCB 比较简单，主要包括进程 pid (SpaceID)、页表（代码、数据、栈）.(实验 7 中已经初步实现了 PCB)
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。(在实验 7 已经通过地址空间的拓展解决该问题)
3. 用户线程与核心线程采用 one-to-one 线程映射模型。

当执行 pid=Exec("../test/halt.noff"); 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)。

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；

## Advance PC

如果不在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。

```
./test/exec.noff
```

## Pid 用户进程标识

进一步思考，如何在内核中记录 pid 呢？才能够在 join() 系统调用能够通过该值找到对应的进程？

Thread::Fork(VoidFunctionPtr func, \_int arg) 有两个参数，一个是线程要运行的函数指针(入口)func，另外一个是参数 arg int 类型，则可以考虑使用 arg 在用户进程映射到核心进程的时候将 pid 传入到内核中。

## 设计与实现

### Join()系统调用的实现

Join 系统调用 主要通过 int Join(SpaceId id) 来实现，调用 join 的用户进程等待 SpaceId = id 的进程结束后，才会结束本进程，而 Join() 返回的是进程 id 的退出码。

#### 具体设计与实现

当子线程执行结束后，父线程执行 Join() 时应该直接返回，不需等待；

由于目前 Nachos 没有实现线程家族树的概念，给 Join() 的实现带来一些困难；

前面已经提到，AddrSpace 类代替了 PCB 的功能，因此可以在 AddrSpace 中设法建立进程之间的家族关系；

1. 在 thread.cc 中添加函数 Join(int spaceId)，系统调用 int Join(int spaceId) 通过调用 Thread::Join() 实现
2. Thread::Join() 将当前线程睡眠，由于当前线程负责执行当前用户进程，因此效果是当前进程进入睡眠；

(a) 在 Scheduler::Scheduler() 中初始化一个线程等待队列及线程终止队列；

```
class Scheduler {
public:
    Scheduler();           // Initialize list of ready threads
    ~Scheduler();          // De-allocate ready list

    void ReadyToRun(Thread* thread); // Thread can be dispatched.
    Thread* FindNextToRun();        // Dequeue first thread on the ready
                                    // list, if any, and return thread.
    void Run(Thread* nextThread);   // Cause nextThread to start running
    void Print();                 // Print contents of ready list

private:
    List *readyList;            // queue of threads that are ready to run,
                                // but not running
```

```

List *waitingList; // queue of threads that are waiting
List *terminatedList; // queue of threads that are ready to be
terminated
};

```

并且在构造函数中初始化这两个队列

```

Scheduler::Scheduler()
{
    readyList = new List;
#ifndef USER_PROGRAM
    waitingList = new List;
    terminatedList = new List;
#endif
}

```

(b) Thread::Join(spaceId)中依据所等待进程的 spaceId 检查其对应的线程是否已经执行完毕

- 如果尚未退出，则将当前线程进入等待队列，然后调用 Thread::Sleep() 使当前线程睡眠（会引起线程调度），被唤醒后返回进程的退出码；
- 如果 Join(spaceId) 所等待的进程已经结束，Join() 应该直接返回，不需要等待；

如何检查所等待的进程（所对应的线程）是否已经退出？

为线程增加一个 TERMINATED 状态，相应地增加一个 terminated 队列，将所有的线程在调用 Finish() 后先进入该队列，再伺机销毁；这样我们可以通过检查 terminated 队列以确定一个线程是否已经终止

**Joiner** 是 执行 join 的进程(等待别的结束的), **Joinee** 是 SpaceId=id 的线程(被等待结束的)

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

为了销毁那些没有被 Join() 等待的线程 并且 从 terminated 队列移除 Joinee，这里采用了临时解决方案，如下

- 为父进程设置特殊的退出码，在 Exit() 中识别父进程，由父进程负责销毁 terminated 队列中的所有线程。

但该方案还是存在问题，即 terminated 队列中可能存在不属于该父进程的子进程，但被该父进程一并销毁。导致其真正的父进程在 Join() 这个子进程时无法找到该子进程。

(c) 当被等待的进程 Joinee 结束时，其对应的核心线程会自动执行 Thread::Finish()，因此需要修改 Thread::Finish() 函数，当被等待的进程 Joinee 退出时，唤醒线程 Joiner。

## 代码

### 1. 修改 exception.c 添加对 Join 系统调用的处理代码

```
case SC_Join:{
    printf("Execute system call of
Join(),CurThreadId:%d\n", (currentThread->space)->getSpaceId());
    int SpaceId = machine->ReadRegister(4);
    currentThread->Join(SpaceId);
    machine->WriteRegister(2,currentThread->waitExitCode());
    AdvancePC();
    break;
}
```

### 2. 在 thread.cc 里面添加 Join(int spaceID)，系统调用 Join 是在此基础上实现的

```
void Thread::Join(int SpaceId){
    // Disable Interrupt
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    // 设置父进程等待的子进程的 SpaceId
    currentThread->setWaitProcessSpaceId(SpaceId);
    // get TerminatedList
    List *terminatedList = scheduler->getTerminatedList();
    // check if joinee in TerminatedList
    bool interminatedList = FALSE;
    ListElement *p = terminatedList->getFirstElement();
    // 在终止队列中查找 Joinee
    while(p != NULL){
        Thread *thread = (Thread *)p->item;
        if(thread->userProgramId() == SpaceId){ // is in List
            interminatedList = TRUE;           // change Flag
            currentThread->setWaitExitCode(thread->ExitCode()); // set
            wait exitcode
            break;
        }
        p = p->next;
    }
    // Joinee is not in TerminatedList，则将 Joiner 放入等待队列中，并进入
    // 睡眠等待状态
    if(!interminatedList){
        List *waitingList = scheduler->getWaitingList();
```

```

        waitingList->Append((void *)this);
        currentThread->Sleep();      // Block Joiner
    }
    // wake up joiner and 在终止队列中删除 Joinee
    scheduler->deleteTerminatedThread(SpaceId);
    (void) interrupt->SetLevel(oldLevel); // open interrupt
}

```

3. Join 需要把当前线程 sleep，需要在 scheduler 里面添加等待队列以及终止队列，并初始化。

```

#ifndef USER_PROGRAM
private:
    List *waitingList; // queue of threads that are waiting
    List *terminatedList; // queue of threads that are ready to be
terminated
public:
    List* getReadyList(){return readyList;}
    List* getWaitingList(){return waitingList;}
    List* getTerminatedList(){return terminatedList;}
    void deleteTerminatedThread(int deleteSpaceId);
    void emptyList(List *tmpList) { delete tmpList; }
#endif

```

```

Scheduler::Scheduler()
{
    readyList = new List;
#ifndef USER_PROGRAM
    waitingList = new List;
    terminatedList = new List;
#endif
}

```

4. Thread 的终止函数,把线程加入终止队列并且切换新的线程

```

void
Thread::Terminated(){
    List *terminatedList = scheduler->getTerminatedList();
    ASSERT(this == currentThread);
    ASSERT(interrupt->getLevel() == IntOff);

    // set status as terminated and put in List
    status = TERMINATED;
    terminatedList->Append((void *)this);
    Thread *nextThread = scheduler->FindNextToRun();
}

```

```

while(nextThread == NULL){
    interrupt->Idle();
    nextThread = scheduler->FindNextToRun();
}
scheduler->Run(nextThread);
}

```

5. Thread::finish 处理等待队列线程，并且最后结束当前线程的执行。

```

void
Thread::Finish ()
{
    (void) interrupt->SetLevel(IntOff); // off
    ASSERT(this == currentThread);
#ifndef USER_PROGRAM
    List* waitingList = scheduler->getWaitingList();
    ListElement *p = waitingList->getFirstElement();
    // joinee finish and wake up joiner
    while(p!=NULL){
        Thread *waitingThread = (Thread *)p->item;
        if(waitingThread->waitProcessSpaceId ==
currentThread->userProgramId()){
            waitingThread->setWaitExitCode(currentThread->ExitCode());
            scheduler->ReadyToRun((Thread *)waitingThread);
            waitingList->RemoveItem(p);
            break;
        }
        p = p->next;
    }
    // end joinee process
    Terminated();
#else
    DEBUG('t', "Finishing thread \"%s\"\n", getName());
    threadToBeDestroyed = currentThread;
    Sleep();           // invokes SWITCH
    // not reached
#endif
}

```

## Exec()系统调用的实现

已在实验 7 实现

## Exit()系统调用的实现

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页表，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。

## 代码

1. exception 里面处理 exit 系统调用的代码，这里 exitcode=99 表示父线程退出，然后把终止队列里面的线程全部结束

- 从 4 号寄存器读出退出码
- 释放该进程的内存空间以及页表
- 释放该进程的实页
- 释放 pid
- 调用 currentThread->Finish 结束该进程对应的线程

```
case SC_Exit:{
    printf("Execute system call of
Exit(),CurThreadId:%d\n", (currentThread->space)->getSpaceId());
    int exitCode = machine->ReadRegister(4);
    machine->WriteRegister(2,exitCode);
    currentThread->setExitCode(exitCode);
    // a speical target when father process exit, and clear List
    if(exitCode==99)
        scheduler->emptyList(scheduler->getTerminatedList());
    delete currentThread->space;
    currentThread->Finish();
    AdvancePC();
    break;
}
```

## Yield() 系统调用的实现

功能：

不管当前线程是否在地址空间中，都将 CPU 移交给另外一个可运行线程

### 代码

直接调用线程的 Yield 函数，并且修改 PC 寄存器的值

```
case SC_Yield:{  
    printf("SC_Yield,  
CurrentThreadId: %d\n", (currentThread->space)->getSpaceId());  
    currentThread->Yield();  
    AdvancePC();  
    break;  
}
```

而 Thread::Yield() 里面实现了进行的切换即 SWITCH，实验一中已经分析过了

```
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);  
}
```

注意，如果使用参数 -rs 实现分时，或利用系统调用 Yield() 主动释放 CPU，Thread::getName() 的输出的线程名可能有问题，但不影响线程的执行，这时由于 Thread 类的构造函数传入线程名时使用指针导致的问题。

主要是因为当创建线程的函数退出时，字符指针 char \*thname 也随之释放，Thread 类中的 name 会指向一个未知的位置，线程名就不是我们所期望的内容：

```
New Thread SpaceID: 101, Name: ../test/exit.noff
Forking thread "../test/exit.noff" with func = 0x565d76c9, arg = 101
Execute system call of Yield, CurrentThreadId: 100 Name:main
Execute system call of Exit(), CurrentThreadId: 101 Name:a../test/exit 'X[K]V++]V+
Execute system call of Exit(), CurrentThreadId: 100 Name:main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

所以需要修改 Thread 的构造函数，通过 new 来避免 name 的释放

```
Thread::Thread(char* threadName)
{
    // name = threadName;
    name = new char[50];
    strcpy(name,threadName);
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifndef USER_PROGRAM
    space = NULL;
#endif
}
```

## 基于 FILESYS\_STUB 实现文件相关系统调用

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

基于 FILESYS\_STUB 对 Nachos 进行编译，实现文件系统的部分功能，是直接使用 Linux 系统调用实现的对文件的操作，访问的是 Linux 文件系统的文件，而不是 Nachos 磁盘上的文件。

在 sysdep.h 封装了一些 Linux 文件系统调用实现

```
// File operations: open/read/write/lseek/close, and check for error
// For simulating the disk and the console devices.
extern int OpenForWrite(char *name);
extern int OpenForReadWrite(char *name, bool crashOnError);
extern void Read(int fd, char *buffer, int nBytes);
extern int ReadPartial(int fd, char *buffer, int nBytes);
extern void WriteFile(int fd, char *buffer, int nBytes);
extern void Lseek(int fd, int offset, int whence);
extern int Tell(int fd);
extern void Close(int fd);
//extern bool Unlink(char *name);
extern int Unlink(char *name);
```

## Create 系统调用

功能：创建一个 Nachos 文件

实现：

```
case SC_Create:{
    printf("FILESYS_STUB_SC_Create, CurrentThreadId: %d Name:%s
\n", (currentThread->space)->getSpaceId(), currentThread->getName());
    int addr = machine->ReadRegister(4);
    char filename[128];
    for(int i=0;i<128;i++){
        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
    int fileDescriptor = OpenForWrite(filename);
    if(fileDescriptor == -1)
        printf("create file %s failed!\n",filename);
    else
        printf("create file %s succeed,the file id
is %d\n",filename,fileDescriptor);
    Close(fileDescriptor);
    AdvancePC();
    break;
}
```

## Open 系统调用

功能：打开一个文件并且返回文件的 OpenFileId(文件句柄)。

实现：从寄存器读入传入的文件名，然后调用文件系统的 Open 函数，打开文件即可。

```
case SC_Open:{
    printf("FILESYS_STUB_SC_Open, CurrentThreadId: %d Name:%s
\n", (currentThread->space)->getSpaceId(), currentThread->getName());
    // read address
    int addr = machine->ReadRegister(4);
    char filename[128];
    for(int i=0;i<128;i++){
        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
}
```

```

int fileDescriptor = OpenForWrite(filename);
if(fileDescriptor == -1)
    printf("Open file %s failed!\n",filename);
else
    printf("Open file %s succeed, the file id is %d\n",filename,
fileDescriptor);
    machine->WriteRegister(2, fileDescriptor);
    AdvancePC();
    break;
}

```

## Write 系统调用

功能：向文件 OpenFileId 为 id 的文件中写入 buffer 里 size 字节的数据。

实现：从寄存器读出 buffer、fileid、size，然后调用 WriteAt 函数将数据写入一个文件。

```

case SC_Write:{
    printf("FILESYS_STUB_SC_Write, CurrentThreadId: %d Name:%s
\n", (currentThread->space)->getSpaceId(), currentThread->getName());
    int addr = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);
    int fileid = machine->ReadRegister(6);
    // openfile
    OpenFile *openfile = new OpenFile(fileid);
    ASSERT(openfile != NULL);
    // read data
    char buffer[128];
    for(int i=0;i<size;i++){
        machine->ReadMem(addr+i,1,(int*)&buffer[i]);
        if(buffer[i] == '\0') break;
    }
    buffer[size] = '\0';
    // write data
    int writePosition;
    if(fileid == 1) writePosition = 0;
    else writePosition = openfile->Length();
    // begin at writeposition
    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);
    AdvancePC();
}

```

```

        break;
}

```

## Read 系统调用

功能：从打开的 Nachos 文件中读取 size 字节的数据到 buffer 并返回实际读取到的字节数。

实现：和都一样，利用 Openfile->Read 进行读

```

case SC_Read:{
    printf("FILESYS_STUB_SC_Read, CurrentThreadId: %d Name:%s
\n", (currentThread->space)->getSpaceId(), currentThread->getName());
    int addr = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);
    int fileid = machine->ReadRegister(6);

    // open file and read
    char buffer[size+1];
    OpenFile *openfile = new OpenFile(fileid);
    int readbytes = openfile->Read(buffer, size);

    for(int i =0;i<size;i++)
        if(!machine->WriteMem(addr,1,buffer[i])) printf("This is
something Wrong\n");
    buffer[size] = '\0';
    printf("Read succeed,the content is \"%s\",the length is %d
\n",buffer,size);
    machine->WriteRegister(2,readbytes);
    AdvancePC();
    break;
}

```

## Close 系统调用

功能：关闭打开的 Nachos 文件

实现：利用 Linux 系统调用 Close 关闭对应的文件句柄的文件

```

case SC_Close:{
    printf("FILESYS_STUB_SC_Close, CurrentThreadId: %d Name:%s
\n", (currentThread->space)->getSpaceId(), currentThread->getName());
    int fileid = machine->ReadRegister(4);

```

```

        Close(fileid);
        printf("File %d closed succeed!\n",fileid);
        AdvancePC();
        break;
}

```

## 基于 FILESYS 实现文件相关系统调用

下述文件的系统调用是基于 FILESYS 定义的方法实现的（即基于实验 4、5 中实现的 Nachos 文件系统），即文件操作是利用 Nachos 的 Openfile 类实现的，访问的是 Nachos 的磁盘 DISK 上的文件！

### 实现类文件句柄 FileId

1. Nachos 文件系统中没有实现 FileId，首先在 AddrSpace 类里面添加如下成员变量和成员函数

```

#ifndef FILESYS
    // 0 stdin 1 stdout 2 stderr
    OpenFile *fileDescriptor[UserProgramNum]; // Just Like Open file
Table
    int getFileDescriptor(OpenFile *openfile);
    OpenFile *getFileId(int fd);
    void releaseFileDescriptor(int fd);
#endif

```

```

#ifndef FILESYS
//-----
// AddrSpace::getFileDescriptor
//-----
int AddrSpace::getFileDescriptor(OpenFile *openfile) {
    for(int i = 3; i < 10; i++)
        if(fileDescriptor[i] == NULL){
            fileDescriptor[i] = openfile;
            printf("in getFileDescriptor set
FileDescriptor[%d]=%d",i,openfile);
            return i;
        }
    return -1;
}

```

```

//-----
// AddrSpace::releaseFileDescriptor
//-----

void AddrSpace::releaseFileDescriptor(int fd) {
    ASSERT((fd >= 0) && (fd < UserProgramNum));
    fileDescriptor[fd] = NULL;
}

//-----
// AddrSpace::get fileId
//-----

OpenFile* AddrSpace::get fileId(int fd) {
    ASSERT((fd >= 0) && (fd < UserProgramNum));
    return fileDescriptor[fd];
}

#endif

```

## 2. 初始化当前进程打开文件的文件描述符

```

#ifndef FILESYS
    for(int i = 3; i < 10; i++)
        fileDescriptor[i] = NULL;
    OpenFile *StdinFile = new OpenFile("stdin");
    OpenFile *StdoutFile = new OpenFile("stdout");
    OpenFile *StderrFile = new OpenFile("stderr");
    fileDescriptor[0] = StdinFile;
    fileDescriptor[1] = StdoutFile;
    fileDescriptor[2] = StderrFile;
#endif

```

## 3. 重载 OpenFile 类的构造函数

```
OpenFile(char *type) {}
```

## 4. 在 OpenFile 类添加两个标准的 I/O 设备输入输出

```

int WriteStdout(char *from, int numBytes);
    int ReadStdin(char *into, int numBytes);

```

## Create 系统调用

```

case SC_Create:{
    printf("FILESYS_SC_Create, CurrentThreadId: %d Name:%s
\n", (currentThread->space)->getSpaceId(), currentThread->getName());
    int addr = machine->ReadRegister(4);
    char filename[128];
    for(int i=0;i<128;i++){

```

```

        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
    // differernce in here
    if(!fileSystem->Create(filename,0))
        printf("Create file %s failed!\n",filename);
    else
        printf("Create file %s succeed!\n",filename);
    AdvancePC();
    break;
}

```

## Open 系统调用

```

case SC_Open:{ 
    printf("FILESYS_SC_Open, CurrentThreadId: %d Name:%s
\n", (currentThread->space)->getSpaceId(), currentThread->getName());
    // read address
    int addr = machine->ReadRegister(4);
    int fileid;
    char filename[128];
    for(int i=0;i<128;i++){
        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
    //different
    OpenFile *openfile = fileSystem->Open(filename);
    if(!openfile){
        printf("File \"%s\" not exists,could not open!\n",filename);
        fileid = -1;
    }else{
        fileid = currentThread->space->getFileDescriptor(openfile);
        if(fileid < 0)
            printf("Too many files have been Open\n");
        else
            printf("File \"%s\" open succeed!File id
= %d.\n",filename,fileid);
    }
    machine->WriteRegister(2, fileid);
    AdvancePC();
    break;
}

```

## Write 系统调用

```

case SC_Write:{

    printf("FILESYS_SC_Write, CurrentThreadId: %d Name:%s
\n", (currentThread->space)->getSpaceId(), currentThread->getName());
    int addr = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);
    int fileid = machine->ReadRegister(6);

    // create a openfile just like 文件句柄
    OpenFile *openfile = new OpenFile(fileid);
    ASSERT(openfile != NULL);

    // read data
    char buffer[128];
    for(int i=0;i<size;i++){
        machine->ReadMem(addr+i,1,(int*)&buffer[i]);
        if(buffer[i] == '\0') break;
    }
    buffer[size] = '\0';

    // open Nachos file
    openfile = currentThread->space->get fileId(fileid);
    if(!openfile){
        // no file
        printf("Fail to Open File Fileid = %d\n", fileid);
        AdvancePC();
        break;
    }
    if(fileid == 1 || fileid == 2){
        // stdout or stderr
        openfile->WriteStdout(buffer, size);
        AdvancePC();
        break;
    }

    // write data
    int writePosition = openfile->Length();
    openfile->Seek(writePosition);

    // begin at writeposition
    int WrittenBytes = openfile->Write(buffer, size);
}

```

```

    if(WrittenBytes == 0) printf("write file failed!\n");
    else if(fileid!=1 || fileid!= 2) printf("\"%s\" has wrote in file id
= %d succeed!Size = %d\n",buffer,fileid,size);
        AdvancePC();
        break;
}

```

## Read 系统调用

```

case SC_Read:{
    printf("FILESYS_SC_Read, CurrentThreadId: %d Name:%s
\n",(currentThread->space)->getSpaceId(),currentThread->getName());
    int addr = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);
    int fileid = machine->ReadRegister(6);

    // open file and read
    OpenFile *openfile = currentThread->space->get fileId(fileid);
    // read
    char buffer[size+1];
    int readbytes = 0;
    if(fileid == 0) readbytes = openfile->ReadStdin(buffer,size);
    else readbytes = openfile->Read(buffer,size);

    for(int i=0;i<readbytes;i++)
        machine->WriteMem(addr,1,buffer[i]);
    buffer[readbytes] = '\0';

    for(int i =0;i<readbytes;i++)
        if(buffer[i]>0 && buffer[i] <= 9)
            buffer[i] += 0x30;
    char *bufcopy = buffer;
    if(readbytes > 0){
        if(fileid != 0)
            printf("Read file %d succeed! the content is \"%s\", the
length = %d\n",fileid,bufcopy,readbytes);
    }
    else printf("Read file failed!\n");
    machine->WriteRegister(2,readbytes);
    AdvancePC();
    break;
}

```

```
}
```

## Close 系统调用

```
case SC_Close:{  
    printf("FILESYS_SC_Close, CurrentThreadId: %d Name:%s  
\n", (currentThread->space)->getSpaceId(), currentThread->getName());  
    int fileid = machine->ReadRegister(4);  
    OpenFile *openfile = currentThread->space->getFileId(fileid);  
    if(openfile!=NULL){  
        // write back to disk  
        openfile->WriteBack();  
        delete openfile;  
        currentThread->space->releaseFileDescriptor(fileid);  
        printf("File %d closed succeed!\n",fileid);  
    }else printf("Failed to Close File %d.\n",fileid);  
    AdvancePC();  
    break;  
}
```

## Nachos Shell 实现

### 原理

编写 Shell 作为一个用户程序，加载进 Nachos 文件系统，运行，并且不断读取输入，执行其他的命令；其他的命令也是通过实现.noff 文件，然后加载进 Nachos 内核再进行调用的。

### 代码

1. 编写 shell.c 使其输出提示信息，并且不断读取输入，利用了 ConsoleInput 和 ConsoleOutput

```
#include "syscall.h"  
  
int  
main()  
{  
    SpaceId newProc;
```

```
OpenFileId input = ConsoleInput;
OpenFileId output = ConsoleOutput;
char prompt[7], ch, buffer[60];
char Hbuffer[80];
char Cbuffer[80];
int i,j,k,h,m;
char c;

prompt[0] = 'N';
prompt[1] = 'a';
prompt[2] = 'c';
prompt[3] = 'h';
prompt[4] = 'o';
prompt[5] = 's';
prompt[6] = '$';
prompt[7] = ':';

while( 1 )
{
    Write(prompt, 8, output);
    i = 0;
    k = 0;
    h = 0;
    m = 0;
    do {
        Read(&c, 1, input);
        buffer[i] = c;
        Hbuffer[h++] = buffer[i];
        k++;
    } while( buffer[i++] != '\n' );
    Hbuffer[-h] = '\0';
    for (m=0;j<80;j++)
    {
        Cbuffer[m] = buffer[m];
    }
    buffer[--i] = '.';
    buffer[i++] = '.';
    buffer[i++] = '\n';
    buffer[i++] = 'o';
    buffer[i++] = 'f';
    buffer[i++] = 'f';
    buffer[i] = '\0';
}
```

```

if (k == 1)
    continue;

if( h > 0 ) {
    newProc = Exec(buffer);

    if (newProc == -1)
        //Write("Invalid Command, Enter again.", 30, output);
        Write("Invalid Command, Enter again, or try \"help\"\n",
44, output);
    else if (newProc != 127)
    {
        Join(newProc);
        Write("Command \\"", 9, output);
        Write(Hbuffer,k, output);
        Write("\\" Execute Completely.\n", 22, output);
    }
}
i = 0;
}
}

```

## 2. 编写其他命令的.c 文件，例如 echo

```

#include "syscall.h"

int
main()
{
    char c;
    OpenFileId input = ConsoleInput;
    OpenFileId output = ConsoleOutput;
    char buffer[60];
    int k = 0;
    int i = 0;
    int h = 0;
    int flag = 0;
    int j = 0;
    OpenFileId fp;
    do {
        Read(&c, 1, input);
        buffer[i] = c;
    } while( buffer[i++] != '\n' );
    buffer[i] = '\0';
}

```

```

for(;j<i;j++){
    if(buffer[j] == '>'){
        flag = 1;
        break;
    }
}
if(flag){
    char contents[60];
    char filename[60];
    for(k=0;k<j-1;k++)
        contents[k] = buffer[k];
    contents[j-1] = '\0';
    for(h = j+2;h<i;h++){
        if(h == '\n')
            break;
        if(h != ' ')
            filename[h-j-2] = buffer[h];
    }
    filename[i] = '\0';
    fp = Open(filename);
    Write(contents,j-1,fp);
    Close(fp);
}
else
    Write(buffer, i, output);
Exit(0);
}

```

### 3. 编写 touch.c 实现 touch 命令

```

#include "syscall.h"
// create a new file
int main(){
    OpenFileId fp;
    char buffer[50];
    char filename[50];
    int sz;

    // read name
    char c;
    OpenFileId input = ConsoleInput;
    OpenFileId output = ConsoleOutput;
    int k = 0;
    int i = 0;

```

```

int h = 0;
do {
    Read(&c, 1, input);
    filename[i] = c;
} while(filename[i++] != '\n');
filename[i] = '\0';
// read name end

// hint the user
Write("Create New File ", 17, output);
Write(filename, i, output);

//create
Create(filename);
// open file
fp = Open(filename);
Close(fp);
// exit
Exit(0);
}

```

4. 编译链接这些 C 文件，并且加载进 Nachos 文件系统

编写了 init.bash 来批处理这一过程，加载了 shell、halt、echo、exit、touch

```

lab10 > $ init.sh
1   rm DISK
2   ./nachos -f
3   ./nachos -cp ../test/shell.noff shell
4   ./nachos -cp ../test/halt.noff halt.noff
5   ./nachos -cp ../test/echo.noff echo.noff
6   ./nachos -cp ../test/exit.noff exit.noff
7   ./nachos -cp ../test/touch.noff touch.noff
8   ./nachos -x shell
9
10

```

## 测试结果

### Join 测试

Join.c 代码

```
#include "syscall.h"
```

```

int main(){
    SpaceId joinee = Exec("../test/exit.noff");
    Join(joinee);
    Exit(0);
}

```

执行

./nachos -x ../test/join.noff

结果如下：

可以看出首先启动了 main 线程，ID 为 100，然后系统调用 exec 执行的 exit.noff 的 ID 为 101，main 线程在等待 ../test/exit.noff 运行结束之后，才会执行 exit 系统调用，然后退出 main 线程，说明 Join 实现正确！

```

torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/userprog$ ./nachos -x ../test/join.noff
page table dump: 11 pages in total
=====
  VirtPage,      PhysPage
  0,          0
  1,          1
  2,          2
  3,          3
  4,          4
  5,          5
  6,          6
  7,          7
  8,          8
  9,          9
 10,         10
=====
Execute system call of Exec(), CurrentThreadID = 100
page table dump: 10 pages in total
=====
  VirtPage,      PhysPage
  0,          11
  1,          12
  2,          13
  3,          14
  4,          15
  5,          16
  6,          17
  7,          18
  8,          19
  9,          20
=====
New Thread SpaceID: 101, Name: ../test/exit.noff
Forking thread "../test/exit.noff" with func = 0x565f86c9, arg = 101
Execute system call of Join(), curThreadId:100
Execute system call of Exit(), curThreadId:101
Execute system call of Exit(), curThreadId:100
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 79, idle 0, system 40, user 39
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0

```

## Exit 测试

Exit.c 代码

```

#include "syscall.h"

int main(){
    Exit(1);
}

```

修改 exec 代码

```
#include "syscall.h"
int
main()
{
    Exec("../test/exit.noff");
    Exec("../test/exit.noff");
    Exec("../test/halt.noff");
    Exit(0);
}
```

进行测试

结果如下，可以看出：首先运行 main 线程，然后通过 exec 系统调用 exec 执行 exit、exit、halt，然后执行 exit 系统调用，退出 main 线程。与此同时，把 exit、exit、halt 线程加入了就绪队列，因此当 main exit 之后，Exit 线程执行系统调用 exit，第二个 exit 线程执行系统调用 exit，然后 halt 线程执行 halt 系统调用。

Exit 系统调用实现正确！

```
Execute system call of Exec(),CurrentThreadID = 100
page table dump: 10 pages in total
New Thread SpaceID: 101, Name: ../test/exit.noff
Forking thread "../test/exit.noff" with func = 0x565ee6c9, arg = 101
Execute system call of Exec(),CurrentThreadID = 100
page table dump: 10 pages in total
=====
VirtPage,      PhysPage
 0,          21
```

```

New Thread SpaceID: 102, Name: ../test/exit.noff
Forking thread "../test/exit.noff" with func = 0x565ee6c9, arg = 102
Execute system call of Exec(),CurrentThreadID = 100
page table dump: 11 pages in total
=====
    VirtPage,      PhysPage
    0,            31
    1,            32
    2,            33
    3,            34
    4,            35
    5,            36
    6,            37
    7,            38
    8,            39
    9,            40
   10,            41
=====

New Thread SpaceID: 103, Name: ../test/halt.noff
Forking thread "../test/halt.noff" with func = 0x565ee6c9, arg = 103
Execute system call of Exit(),CurThreadId:100
Execute system call of Exit(),CurThreadId:101
Execute system call of Exit(),CurThreadId:102
Execute system call of Halt(),CurrentThreadID = 103
Machine halting!

Ticks: total 154, idle 0, system 70, user 84
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

```

## Yield 测试

测试代码

Yield.c

```
#include "syscall.h"
int main(){
    Exec("../test/exit.noff");
    Yield();
    Exit(0);
}
```

然后 ./nachos -x ../test/yield.noff

执行结果如下，发现首先执行了系统调用 exec，产生 ../test/exit.noff 然后执行 Yield 系统调用，切换进程到 exit 进程，然后 ../test/exit.noff 进程执行 exit 系统调用结束，最后回到 main 线程执行 exit 系统调用退出。  
如下也解决了线程 getName 的问题。

```

Execute system call of Exec(),CurrentThreadID = 100
page table dump: 10 pages in total
=====
    VirtPage,      PhysPage
    0,           11
    1,           12
    2,           13
    3,           14
    4,           15
    5,           16
    6,           17
    7,           18
    8,           19
    9,           20
=====

New Thread SpaceID: 101, Name: ../test/exit.noff
Forking thread "../test/exit.noff" with func = 0x565de6f1, arg = 101
Execute system call of Yield, CurrentThreadId: 100 Name:main
Execute system call of Exit(),CurrentThreadId: 101 Name:../test/exit.noff
Execute system call of Exit(),CurrentThreadId: 100 Name:main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 77, idle 0, system 40, user 37
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...

```

## 线程综合测试

Test1.c 代码

```

#include "syscall.h"
int main(){
    SpaceId prog = Exec("../test/yield.noff");
    Join(prog);
    Exit(0);
}

```

进行了 exec 系统调用，调用了 yield.noff，然后等待 prog 完成，最后 exit。

yield.noff 如下

```

#include "syscall.h"
int main(){
    Exec("../test/exit.noff");
    Yield();
    Exit(0);
}

```

测试结果如下。

0. test1 用户程序映射到 main 线程，分配用户进程空间，spaceid = 100。
1. 首先 main 线程执行了系统调用 exec，执行“..../test/yield.noff”，生成新的用户线程，spaceid = 101
  2. main 线程执行 join 系统调用，等待“..../test/yield.noff”线程执行完成，因为此时是 101 在执行
  3. “..../test/yield.noff”执行 exec(“..../test/exit.noff”), 产生新的用户线程..../test/exit.noff spaceid = 102，加入就绪队列
  4. ..../test/yield.noff 继续执行，进行 yield 系统调用，此时调度到..../test/exit.noff 上继续执行，而不是 main 线程，因为 main 线程在等待队列而不是就绪队列
  5. ..../test/exit.noff 执行 exit 系统调用，成为第一个结束的用户线程，然后调度执行..../test/yield.noff
  6. ..../test/ yield.noff 执行 exit 系统调用，成为第二个结束的用户线程，然后调度执行 main 线程(..../test/test1.noff 用户线程)
  7. main 线程执行 exit 系统调用，成为第三个结束的用户线程，整个程序结束。  
全部正确！

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/userprog$ ./nachos -x ..../test/test1.noff
Execute system call of Exec(), CurrentThreadId: 100 Name:main
New Thread SpaceID: 101, Name: ..../test/yield.noff
Execute system call of Join(), CurrentThreadId: 100 Name:main
Execute system call of Exec(), CurrentThreadId: 101 Name:..../test/yield.noff
New Thread SpaceID: 102, Name: ..../test/exit.noff
Execute system call of Yield, CurrentThreadId: 101 Name:..../test/yield.noff
Execute system call of Exit(), CurrentThreadId: 102 Name:..../test/exit.noff
Execute system call of Exit(), CurrentThreadId: 101 Name:..../test/yield.noff
Execute system call of Exit(), CurrentThreadId: 100 Name:main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 134, idle 0, system 70, user 64
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...
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/userprog$
```

## FILE\_STUB 系统调用综合测试

1. 修改 userprog 里面的 MAKEFILE 如下

```

ifndef MAKEFILE_USERPROG
define MAKEFILE_USERPROG
yes
endif

ifdef MAKEFILE_FILESYS_LOCAL
DEFINES += -DUSER_PROGRAM
else
DEFINES += -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS_STUB
endif

# If the filesystem assignment is done before userprog, then
# uncomment the include below

include ../threads/Makefile.local
# include ../filesys/Makefile.local
include ../userprog/Makefile.local
include ../Makefile.dep
include ../Makefile.common

endif # MAKEFILE_USERPROG

```

## 2. 测试.c 文件代码如下

```

#include "syscall.h"
int main(){
    OpenFileId fp;
    char buffer[50];
    int size;
    //create
    Create("FileStubTest");
    // open file
    fp = Open("FileStubTest");
    //write
    Write("Test System Call base on FileStub!!!",34,fp);
    // read
    size = Read(buffer,16,fp);
    // close
    Close(fp);
    // exit
    Exit(0);
}

```

## 3. 执行

- 1) 首先进行了 Create 系统调用，创建了文件 FileStubTest 并且返回了文件句柄为 3
- 2) 执行系统调用 Open，打开文件成功。
- 3) 执行系统调用 Write，然后写入 34 个 char 到文件中

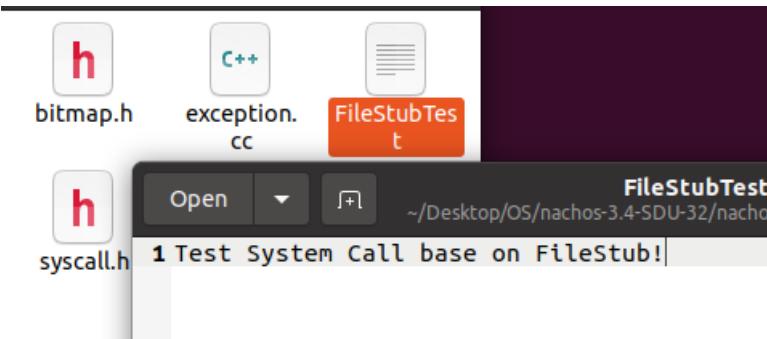
- 4) 执行系统调用 Read, 读出了 16 个字符, 就是文件前 16 个字符
- 5) 最后 close 文件 正确!

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/userprog$ ./nachos -x ../test/testfilestub.noff
FILESYS_STUB_SC_Create, CurrentThreadId: 100 Name:main
create_file FileStubTest succeed, the file id is 3
FILESYS_STUB_SC_Open, CurrentThreadId: 100 Name:main
Open file FileStubTest succeed, the file id is 3
FILESYS_STUB_SC_Write, CurrentThreadId: 100 Name:main
"Test System call base on FileStub!" has wrote in file 3 succeed!
FILESYS_STUB_SC_Read, CurrentThreadId: 100 Name:main
Read succeed, the content is "Test System Call", the length is 16
FILESYS_STUB_SC_Close, CurrentThreadId: 100 Name:main
File 3 closed succeed!
Execute system call of Exit(), CurrentThreadId: 100 Name:main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 65, idle 0, system 10, user 55
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...
```

查看 userprog 下面的 FileStubTest 文件的内容



测试成功！

## FILESYS 系统调用综合测试

1. 在测试阶段, 参考 lab7-8, 新建一个 lab9 目录, 文件内容如下:

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab9$ ls
addrspace.cc    dump           Makefile      switch.s      system.h
addrspace.h     exception.cc   Makefile.local  synch.cc      thread.cc
arch           filehdr.cc     nachos       synchdisk.cc  thread.h
bitmap.cc       filehdr.h     openfile.cc   synchdisk.h  threadtest.cc
bitmap.h        filesys.cc    openfile.h    synch.h       utility.cc
bool.h          filesys.h     progtest.cc  synchlist.cc utility.h
copyright.h     ftest.cc      scheduler.cc  synchlist.h
directory.cc    list.cc       scheduler.h   synctest.cc
directory.h     list.h        switch.h     syscall.h
DISK           main.cc       switch-linux.s svstem.cc
```

2. 编写 makefile 以及 makefile.local, 如下

```
ifndef MAKEFILE_USERPROG
define MAKEFILE_USERPROG
yes
endif

# If the filesystem assignment is done before userprog, then
# uncomment the include below

include ../lab9/Makefile.local
include ../Makefile.dep
include ../Makefile.common

endif # MAKEFILE_USERPROG
```

```
ifndef MAKEFILE_USERPROG_LOCAL
define MAKEFILE_USERPROG_LOCAL
yes
endif

# ifndef MAKEFILE_THREADS_LOCAL
# define MAKEFILE_THREADS_LOCAL
# yes
# endif

SFILES = switch$(HOST_LINUX).s

# If you add new files, you need to add them to CCFILES,
# you can define CFILES if you choose to make .c files instead.
#
# Make sure you use += and not = here.

CCFILES += main.cc\
list.cc\
scheduler.cc\
synch.cc\
synchlist.cc\
system.cc\
thread.cc\
utility.cc\
threadtest.cc\
synctest.cc\
interrupt.cc\
sysdep.cc\
```

```

stats.cc\
timer.cc\
addrspace.cc\
bitmap.cc\
exception.cc\
progtest.cc\
console.cc\
machine.cc\
mipssim.cc\
translate.cc\
directory.cc\
filehdr.cc\
filesys.cc\
fstest.cc\
openfile.cc\
synchdisk.cc\
disk.cc

```

INCPATH += -I- -I..../lab7-8 -I..../threads -I..../machine -I..../bin -I..../filesys -I..../monitor -I..../network

DEFINES += -DTHREADS

```

ifdef MAKE_FILE_FILESYS_LOCAL
DEFINES += -DUSER_PROGRAM
else
#DEFINES += -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS_STUB
DEFINES += -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS
endif

```

endif# MAKEFILE\_USERPROG\_LOCAL

3. 进行测试的 C 文件源码如下， filetest.c

```

#include "syscall.h"
int main(){
    OpenFileId fp;
    char buffer[50];
    int sz;
    //create
    Create("FTest");
    // open file

```

```
fp = Open("FTest");
//write
Write("Test System Call base on FileSystem!!!",39,fp);
// read
sz = Read(buffer,16,fp);
// close
Close(fp);
// exit
Exit(0);
}
```

4. ./nachos -f 初始化 Nachos 磁盘

5. ./nachos -cp ..//test/filetest.noff filetest 使用 cp 命令，把可执行 noff 文件，复制到 Nachos 的磁盘中

6. ./nachos -x filetest 运行用户程序

```
torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab9$ ./nachos -x filetest
FILESYS_SC_Create, CurrentThreadId: 100 Name:main
Create file FTest succeed!
FILESYS_SC_Open, CurrentThreadId: 100 Name:main
in getFileDescriptor set FileDescriptor[3]=1474890624File "FTest" open succeed!File id = 3.
FILESYS_SC_Write, CurrentThreadId: 100 Name:main
"Test System Call base on FileSystem!!!" has wrote in file id = 3 succeed!Size = 39
FILESYS_SC_Read, CurrentThreadId: 100 Name:main
Read file 3 succeed! the content is "Test System Call", the length = 16
FILESYS_SC_Close, CurrentThreadId: 100 Name:main
File 3 closed succeed!
Execute system call of Exit(),CurrentThreadId: 100 Name:main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 101526, idle 100591, system 880, user 55
Disk I/O: reads 22, writes 7
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

## 7. 分析结果:

- 1) 首先进行 Create 系统调用，在 Nachos 的文件系统中创建一个新文件 FTest，分配文件句柄号为 3
  - 2) 进行系统调用 Open，打开文件 FTest
  - 3) 进行系统调用 Write，写入内容 Test System Call base on FileSystem!!! 此时数据已经写入到了磁盘上

- 4) 进行系统调用 Read, 尝试读取文件内容, 16 个字节, 读取到了 Test System Call
  - 5) 执行系统调用 Close, 关闭文件, 并且把文件头相关信息写回到磁盘上。
  - 6) 执行 Exit 系统调用, 执行结束!

8. 查看磁盘 `./nachos -D` 以及 `hexdump -C DISK`, 发现文件 FTest 已经写入到了 Nachos 的磁盘中

9. 测试完毕！全部正确！

## Nachos Shell 测试

## 1. bash init.sh 初始化 Nachos 的 shell 环境

```
Cleaning up...
No threads ready or runnable, and no pending interrupt
Assuming the program completed.
Machine halting!

Ticks: total 1076020, idle 1071960, system 4060, user
Disk I/O: reads 70, writes 65
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Nachos$ echo
```

## 2. 测试 echo、touch、以及 echo >(写入文件)、exit 命令

```

Cleaning up...
Nachos$:echo
hello shell
hello shell
Command "echo" Execute Completely.
Nachos$:touch
tctest
Create New File tctest
Command "touch" Execute Completely.
Nachos$:echo
Testing touch > tctest
Command "echo" Execute Completely.
Nachos$:exit
Machine halting!

Ticks: total 278755, idle 270224, system 3000, user 5531
Disk I/O: reads 88, writes 8
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

```

测试了 echo、touch、echo 写入，最后使用 exit 退出 shell 正确！

### 3. 查看 touch 创建的文件

```

Name: tctest
, Sector: 40
FileHeader contents. File size: 12. File blocks:
41
File contents:
TestingTouch

```

```

torres@torres-Linux:~/Desktop/OS/nachos-3.4-SDU-32/nachos-3.4-SDU/code/lab10$ ./nachos -l
shell
halt.noff
echo.noff
exit.noff
touch.nof
tctest

```

说明测试成功！

## 结论体会

1. 在实现基于 Nachos 自己的文件系统的系统调用时，发现如果文件不能正常 Close，会导致使用 hexdump -C 查看磁盘时 发现文件内容写入，但是使用 nachos -D 发现输出的文件大小为 0，内容为空。原因就在于：Close 没有正常执行 openfile 没有 WriteBack()，导致没有把文件头等信息写入磁盘，不能找到对应的 Size 以及磁盘位置。

2. Nachos 系统调用如何进行参数传递

在 MIPS 架构中，对于一般的函数调用，一般利用 S4-S7 传递函数的前四个参数给子程序，参数多于 4 个时，其余参数使用堆栈进行传递。一般来说，Nachos 将要传递的参数依次保存到寄存器 S4-\$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，当字符串作为参数时，相应寄存器中保存的是字符串在内存中的地址。

### 3. Nahcos 用户进程标识的产生及维护方法？

为每一个地址空间 或者 该地址空间对应的现场分配一个唯一的整数作为用户进程标识，0~99 是 预留给核心进程使用的，用户进程从 100 开始使用。由于目前 Nachos 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 Spaceld 即 Pid 设为 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，需回收为其分配的 pid 号以分配给后续的应用程序，同时释放其所占用的内存空间及页表等信息，供后续进程使用。

### 4. 在 Join 的实现中如何检查所等待的进程已经退出？

为线程增加一个 TERMINATED 状态，相应地增加一个 terminated 队列，所有的线程在调用 FinishO 后先加入该队列，再被伺机销毁，具体方法如下：

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

### 5. 本次实验，也是最后一个实验，首先进行了进程地址空间的拓展，使得 Nachos 系统能够支持多进程机制，然后又基于对 Nachos 系统调用的执行流程、核心映射、参数传递等代码，对源码进行修改，实现了 Join、Exec、Exit、Yield 以及基于两种文件系统的 Create、Open、Write、Read、Close 总计 10 种系统调用。

## 参考文献

- [1] Abraham Silberschatz. 操作系统概念（第七版）[M].北京：高等教育出版社，2010.10
- [2] Nachos-3.4（C++）源代码
- [3] Peiyi Tang, Ron Addie, nachos\_study\_book.pdf, University of Southern Queensland, 2002