



山东大学
SHANDONG UNIVERSITY

操作系统课程设计

——山东大学（青岛）

学院：计算机科学与技术学院

姓名：牛彤

学号：201622130157

任课教师：杨兴强

摘要

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，可让学生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。

实验从系统安装与调试、利用信号量实现线程同步、文件系统、用户程序及系统调用这四个大实验，让学生对操作系统中的线程、文件管理、存储管理、系统调用等方面有了深入而全面的了解。学生通过对原系统代码的理解和更改，使系统更加完善，也使思维从理论知识走到实践中来，锻炼了理解代码能力，独立思考能力，动手能力。

目录

(一) 实验 1-2 Nachos 系统的安装与调试.....	4
一. 目的与要求.....	4
二. 源代码分析.....	4
三. 调试与运行.....	9
四. 收获与心得.....	11
(二) 实验 3 利用信号量实现线程同步.....	11
一. 目的与要求.....	11
二. 源代码分析.....	12
三. 代码实现.....	15
四. 运行结果及分析.....	16
五. 收获与心得.....	17
(三) 实验 4-5 Nachos 文件系统.....	17
一. 目的与要求.....	17
二. 源代码分析.....	19
三. 代码实现.....	30
四. 运行结果及分析.....	37
五. 收获与心得.....	44
(四) 实验 6-8 Nachos 用户程序及系统调用.....	44
一. 目的与要求.....	44
二. 源代码分析.....	46
三. 代码实现.....	53
四. 运行结果及分析.....	61
五. 收获与心得.....	63
(五) 总结.....	64
(六) 参考文献.....	65

(一) 实验 1-2 Nachos 系统的安装与调试

一. 目的与要求

- 正确安装编译 Nachos 系统，理解 Nachos 系统的组织结构，熟悉 C++编程语言；
- 安装测试 gcc MIPS 交叉编译器；
- 掌握利用 Linux 调试工具 GDB 调试跟踪 Nachos 的执行过程；
- 通过跟踪 Nachos 的 C++程序及汇编代码，理解 Nachos 中线程上下文切换的过程；
- 阅读 Nachos 的相关源代码，理解 Nachos 内核的工作原理及其测试过程；
- 熟悉 Makefile 的使用
- 实行 Nachos 中的 Makefiles 的结构

二. 源代码分析

- 1.正确安装编译 Nachos 系统，理解 Nachos 系统的组织结构，熟悉 C++编程语言；
- 2.安装测试 gcc MIPS 交叉编译器；

Q: 为什么 nachos-3.4tar.gz 一定要安装在/user/local 目录中

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

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

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

A: 首先介绍一下我常用的几个 gdb 调试指令：

b:break point 设置断点。

l:list 输出后续 10 行代码

r: run 运行 stack

n: next 逐条跟踪语句

s: step 键入单步执行命令，查看函数的内部指令

print: 输出程序中变量的值

B: 跟踪 GDB 的调试过程:

①程序自 main 开始执行

②调用 Initialize 函数初始化 Nachos 的设备与内核。

处理 Nachos 内核使用的一些命令行参数;

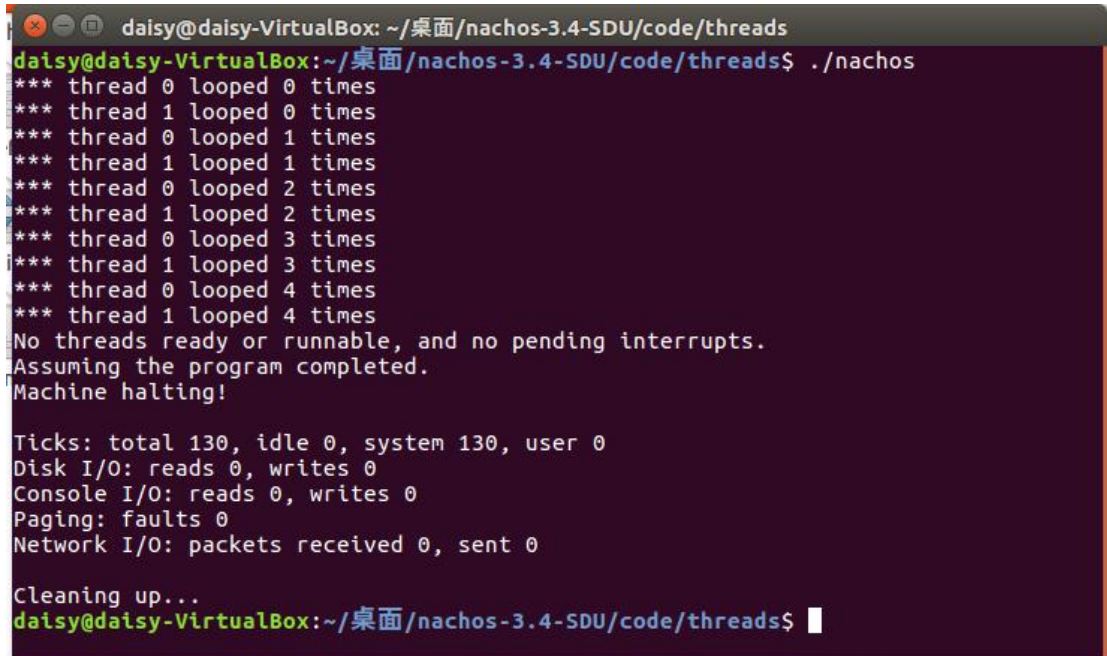
根据需要创建 Nachos 相应的硬件设备, 如中断控制器, 定时器, CPU, 硬盘等;

基于这些设备初始化一个 Nachos 的内核: 如初始化了一个线程调度程序, 在硬盘上创建了文件系统, 创建了 Nachos 的主线程 (Nachos 的第一个线程), 以及网络通信使用的邮箱。

③调用 ThreadTest 函数对 Nachos 内核进行测试 (线程的创建及并发执行)

命令行中的输出结果主要是该函数的执行结果:

主线程 “main” 创建了一个子线程 “forked Thread”, 并将 SimpleThread () 的函数指针传给子线程。随后主线程执行 SimpleThread(0), 子线程执行 SimpleThread(1), 由于 SimpleThread () 调用了 Thread::Yield(), 因此两个线程会交替执行, 输出如下结果。



```
daisy@daisy-VirtualBox: ~/桌面/nachos-3.4-SDU/code/threads
daisy@daisy-VirtualBox:~/桌面/nachos-3.4-SDU/code/threads$ ./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 130, idle 0, system 130, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
daisy@daisy-VirtualBox:~/桌面/nachos-3.4-SDU/code/threads$
```

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

本程序的运行关键是 ThreadTest 函数

```

41 void
42 ThreadTest()
43 {
44     DEBUG('t', "Entering SimpleTest");
45
46     Thread *t = new Thread("forked thread");
47
48     t->Fork(SimpleThread, 1);
49     SimpleThread(0);
50 }
51

```

进入 ThreadTest 函数，如上，函数完成了创建新的线程对象 forked thread，调用对象的 fork 方法，绑定 SimpleThread 这个函数，主进程执行函数 Simple Thread。

SimpleThread 函数中做的工作是打印出线程 n 的循环次数 i。接下来分析一下 Yield 函数和 Fork 函数。

Fork() :

①为线程分配栈空间，在 StackAllocate 函数中主要进行以下数据的初始化可以看出它为线程定义了整个生命周期各个阶段的入口地址和参数，包括线程初始化函数 ThreadRoot，fork 函数中传入的函数指针，线程结束函数 ThreadFinish。

②将线程本身添加到 scheduler 的就绪队列中等待调度。本步操作是原子操作不允许被中断。

Yield() :

①将当前线程添加到就绪队列

②scheduler 切换至下一个可以运行的线程。综上，两步操作中途不允许打断，且完成的工作是打断当前线程，允许就绪队列中下一个线程被调度运行，自身加入就绪队列。

接下来观察 ThreadRoot 和 SWITCH 的执行，switch 函数完成的操作很简单：将当前正在执行的状态存入“旧线程”对象中，从“新线程”对象中取出运行状态加载到寄存器，并跳转到 ra 即将要执行的代码地址。

5. nachos 的 Makefile 文件

code/的子目录下一般都有 Makefile 与 Makefile.local 两个工程文件，用于对包含该功能的 Nachos 系统进行编译与链接。例如基于 code/thread 目录下的这两个 makefile 文件，可以生成一个 Nachos 的最小内核，包含 Nachos 线程的创建、调度、撤销等功能；基于 code/filesys 目录下的两个 makefile 文件，可以生成包含文件系统的 Nachos 内核。code/目录下还有两个 makefile 文件：Makefile.common 与 Makefile.dep，包含编译、

链接 Nachos 系统所需的 makefile 公共语句，被 code 下子目录中的 Makefile 与 Makefile.local 所共享。

其大致结构入下：

```

../code/Makefile.common
/Makefile.dep
|
|
/threads/Makefile
/Makefile.local
|
|
/filesystem/Makefile
/Makefile.local
|
|
..
    
```

6. code/下子目录中的 Makefile 文件

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

其中的内容非常简单，重点代码只有这两句：

```

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

7. code/下子目录中的 Makefile.local 文件

该文件的作用主要是对一些编译、链接及运行时所使用的宏进行定义。使用 vi 命令在终端打开 Makefile.local 文件，如下：

```

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.

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

INCPATH += -I../threads -I../machine

DEFINES += -DTHREADS

endif # MAKEFILE_THREADS_LOCAL

```

其中，CCFILES 指定在该目录下生成 Nachos 时所涉及到的 C++源文件；INCPATH 指明所涉及的 C++源程序中的头文件所在的路径，以便利用 g++ 进行编译链接时通过这路径查找这些头文件；DEFINES 传递 g++ 的一些标号或者宏。

8. code/目录下的 Makefile.dep 文件

Makefile.dep 文件根据安装 Nachos 时所使用的操作系统环境，定义一些相应的宏，供 g++ 使用。在该文件中，语句 `uname = $(shell uname)` 就是获取安装 Nachos 所使用的操作系统平台，然后利用语句 `ifeq ($(uname),xxxx)` 根据所使用的平台定义相应的宏，为 g++ 所使用（xxxx 是相应的平台名，如 Linux）。

9. code/目录下的 Makefile.common 文件

Makefile.common 文件是最复杂的，它定义了编译链接生成一个完整的 Nachos 可执行文件所需要的所有规则。文件 Makefile.common 首先利用 `include` 语句把文件 Makefile.dep 包含进去（`include ../Makefile.dep`），然后利用 `vpath` 定义了一些编译时查找相关文件的路径，当利用 `make` 命令编译生成 Nachos 系统时，如果在当前目录下找不到相应的文件，会在 `vpath` 给出的路径中查找这些文件。

三. 调试与运行

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

- i. InterruptEnable()
- ii. SimpleThread()
- iii. ThreadFinish()
- iv. ThreadRoot()

答：找到地址的过程及方法：在 gdb 中在上述函数上设置断点，断点上显示的即为答案。答案在下图中给出。

```
(gdb) b InterruptEnable
Breakpoint 1 at 0x804a2e2: file thread.cc, line 242.
(gdb) b SimpleThread
Breakpoint 2 at 0x804a49b: file threadtest.cc, line 29.
(gdb) b ThreadFinish
Breakpoint 3 at 0x804a2c8: file thread.cc, line 241.
(gdb) b ThreadRoot
Breakpoint 4 at 0x804bad8
```

2. 下述线程对象的地址是多少？并说明找到这些对象地址的过程及方法。

- i. the main thread of the Nachos
- ii. the forked thread created by the main thread

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

ii. 要查看由主线程创建的线程对象的地址，在 `ThreadTest()` 函数中的 `SimpleThread(0)` 那一行上设置断点，程序运行到这个断点暂停之后，利用 `(gdb) p t` 给出主线程对象的地址。

```
Breakpoint 2, Initialize (argc=0, argv=0xbffff098) at system.cc:150
150      currentThread->setStatus(RUNNING);
(gdb) p t
No symbol "t" in current context.
(gdb) p currentThread
$1 = (Thread *) 0x8054af0
(gdb)
```

主线程对象的地址是 0x8054af0

```
Breakpoint 4, ThreadTest () at threadtest.cc:49
49      SimpleThread(0);
(gdb) p t
$2 = (Thread *) 0x8054b50
(gdb)
```

新线程对象的地址是 0x8054b50

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

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

单条指令跟踪 SWITCH() 执行，当执行完最后一条指令 0x0804bb3a<+88>ret 后，看看下条指令在何处执行，该指令地址就是 SWITCH() 的返回地址。

通过实验，下条执行的指令的地址是 0x0804bad4，对应程序中的 ThreadRoot 函数

```
Breakpoint 5, 0x0804bae2 in SWITCH ()
(gdb) s
Single stepping until exit from function SWITCH,
which has no line number information.
0x0804bad4 in ThreadRoot ()
```

continue 继续执行，第二次在 SWITCH 暂停执行，利用 s 或 n 单步执行，返回地址在 Scheduler::Run() 中，

```
Breakpoint 5, 0x0804bae2 in SWITCH ()
(gdb) s
Single stepping until exit from function SWITCH,
which has no line number information.
Scheduler::Run (this=0x8054ad0, nextThread=0x8054b50) at scheduler.cc:118
118      DEBUG('t', "Now in thread \"%s\\n", currentThread->getName());
(gdb)
```

重复上述过程，会发现 SWITCH 函数后续的返回位置与第二次返回的位置相同。

为什么会出现这种情况：

从代码中可以看出，Nachos 的主线程 “main” 中的 t->Fork(SimpleThread, 1) 语句调用了 Fork 函数，创建了一个子线程，并且其所执行的代码是 SimpleThread(1)，主线程 main 被调度是所执行的代码为 SimpleThread(0)；不管采用 RR 还是 FCFS 线程调度算法，都是首先调度执行主线程 “main”，因为在 Initialize 函数中就已经进入了就绪队列。主线程输出相关信息之后就会调用 Yield()，该函数会将子线程从就绪队列中取出，并将主线程的状态从执行转到就休并放入就绪队列尾部，将子线程设为执行状态，然后调用 SWITCH 将主线的上下文切换到子线程的上下文。

以上是 SWITCH 第一次被调用，这次 SWITCH 的返回值是 ThreadRoot 的第一条指令开始执行，由于子线程是从头开始执行，因此 ThreadRoot 是所有利用 Fork 创建的线程的入口。

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

四. 收获与心得

1. 熟悉了 gdb 的调试命令。对 Nachos 有了一个比较宏观的认识，其中对该系统中线程的创建的线程之间的切换机制有了深刻的理解，为后面的实验三打下了坚实的基础。了解了 Nachos 的大体架构和 Nachos 线程的创建、切换等方式。深入理解了 ThreadRoot(), ThreadTest() 等函数，了解到 ThreadRoot() 是压入栈底的函数，每个线程对象在创建之初已经存在，Thread() 方法只是对其属性进行初始化，赋予其一个数据结构。同时实验一还熟悉了 gdb 的使用，通过各种指令可以利用 gdb 逐步对程序进行调试，了解程序的运行过程，深入每一个函数进行查看，查看各种对象的创建和地址等等。

2. 通过这次的实验，我对 makefile 的结构有了大致的了解。对 makefile 的一些相关文件的宏定义有了较为深刻的认识。知道了如何对外部的文件进行链接等内容。

(二) 实验 3 利用信号量实现线程同步

一. 目的与要求

1. 目的

- 理解生产者-消费者模型；
- 理解 Nachos 中如何创建并发线程；
- 理解 Nachos 中信号量是如何实现的；
- 理解如何创建与使用 Nachos 的信号量
- 理解 Nachos 中是如何利用信号量实现 producer/consumer problem；
- 理解 Nachos 中如何测试与调试程序

- 理解 Nachos 中轮转法（RR）线程调度的实现

2. 要求

- 根据../lab3/prodcons++.cc 中的提示完善代码，运行 Nachos，查看输出结果；
- 根据生产者/消费者问题的功能定义，你的实现应该满足如下条件：
- 生产者线程所产生的所有的消息，都应该被消费者接收并保存到输出文件中(temp_0, temp_1, ...)
- 每个消息只能被接收一次且在文件保存一次
- 来自于同一个生产者的消息，以及被同一个消费者接收到的消息，在文件保存的顺序应该按其序号升序排列；

二. 源代码分析

1. 线程的创建

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

2. 信号量是如何实现的

到在 synch.cc 中定义了信号量，其中如下的 P, V 操作的定义使得信号量在改变时实现原子性，即在对信号量操作之前关中断，操作结束后再关中断。

```

64 void
65 Semaphore::P()
66 {
67     IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts
68
69     while (value == 0) { // semaphore not available
70         queue->Append((void *)currentThread); // so go to sleep
71         currentThread->Sleep();
72     }
73     value--; // semaphore available,
74             // consume its value
75
76     (void) interrupt->SetLevel(oldLevel); // re-enable interrupts
77 }
78
79 //-----
80 // Semaphore::V
81 //     Increment semaphore value, waking up a waiter if necessary.
82 //     As with P(), this operation must be atomic, so we need to disable
83 //     interrupts. Scheduler::ReadyToRun() assumes that threads
84 //     are disabled when it is called.
85 //-----
86
87 void
88 Semaphore::V()
89 {
90     Thread *thread;
91     IntStatus oldLevel = interrupt->SetLevel(IntOff);
92
93     thread = (Thread *)queue->Remove();
94     if (thread != NULL) // make thread ready, consuming the V immediately
95         scheduler->ReadyToRun(thread);
96     value++;
97     (void) interrupt->SetLevel(oldLevel);
98 }
99
100

```

3. 分析 Thread 中 fork, yield, sleep, finish 等函数，理解线程调度过程。

Fork:

```

87 void
88 Thread::Fork(VoidFunctionPtr func, _int arg)
89 {
90     #ifdef HOST_ALPHA
91     DEBUG('t', "Forking thread \"%s\" with func = 0x%lx, arg = %ld\n",
92         name, (long) func, arg);
93     #else
94     DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",
95         name, (int) func, arg);
96     #endif
97
98     StackAllocate(func, arg);
99
100     IntStatus oldLevel = interrupt->SetLevel(IntOff);
101     scheduler->ReadyToRun(this); // ReadyToRun assumes that interrupts
102                                 // are disabled!
103     (void) interrupt->SetLevel(oldLevel);
104 }
105

```

首先 Thread 的构造方法创建了一个新的线程，随后调用在 fork 函数，在 fork 函数中。

①为线程分配栈空间，在 StackAllocate 函数中主要进行以下数据的初始化 可以看出它为线程定义了整个生命周期各个阶段的入口地址和参数，包括线程初始化函数 ThreadRoot, fork 函数中传入的函数指针，线程结束函数 ThreadFinish。

②将线程本身添加到 scheduler 的就绪队列中等待调度。本步操作是原子操作不允许被中断。

Yield:

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

    ASSERT(this == currentThread);

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

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

①将当前线程添加到就绪队列

②scheduler 切换至下一个可以运行的线程。 综上，两步操作中途不允许打断，且完成的工作是打断当前线程，允许就绪队列中下一个线程被调度运行，自身加入就绪队列。

接下来观察 ThreadRoot 和 SWITCH 的执行，switch 函数完成的操作很简单：将当前正在执行的状态存入“旧线程”对象中，从“新线程”对象中取出运行状态加载到寄存器，并跳转到 ra 即将要执行的代码地址。

Sleep:

```
217 Thread::Sleep ()
218 {
219     Thread *nextThread;
220
221     ASSERT(this == currentThread);
222     ASSERT(interrupt->getLevel() == IntOff);
223
224     DEBUG('t', "Sleeping thread \"%s\"\n", getName());
225
226     status = BLOCKED;
227     while ((nextThread = scheduler->FindNextToRun()) == NULL)
228         interrupt->Idle(); // no one to run, wait for an interrupt
229
230     scheduler->Run(nextThread); // returns when we've been signalled
231 }
???
```

可以看出当线程调用 sleep 时，首先将状态设为阻塞态，然后从就绪队列中选取第一个线程运行。如果就绪队列为空，就等待。

三. 代码实现

1. 创建信号量

```
Semaphore *nempty, *nfull; //two semaphores for empty and full slots
Semaphore *mutex;          //semaphore for the mutual exclusion
// ....
mutex=new Semaphore("mutex",1);
nempty=new Semaphore("nempty",BUFF_SIZE);
nfull=new Semaphore("nfull",0);
```

mutex 为互斥信号量，nempty 和 nfull 为同步信号量。

2. Producer:

```
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.
        // ...
        message->value=num;
        message->thread_id=which;

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

        ring->Put(message);

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

注意一定要同步在前，互斥在后。

3. Consumer 核心代码:

```
for ( ; ; ) {

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

    ring->Get(message);

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

四. 运行结果及分析

1. 在不考虑 rr 调度算法的情况下，即线程调度按照 FCFS 方式会出现如下结果。

第一个消费者进程，取出如下：

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

第二个消费者进程，没有取出任何东西。

说明，第二个消费者进程根本没有得到过 CPU，没有执行过。且从上图可以看出，是一个消费者进程执行完所有指令后，第二个消费者进程才会有机会执行。

2. 如果在执行 nachos 文件时候后面加指令 -rs, 说明系统的线程调度采用时间片抢先式。

以 -rs 5 为例，两个消费者进程分别得出如下文件。

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



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

五. 收获与心得

通过实验三，进一步加深并巩固了对于 Nachos 中线程的创建、切换等一些列功能和状态的理解，并实现了 Nachos 下的线程同步。同步问题是上学期操作系统学习中接触到的极其重要的问题之一，在本实验中依旧通过信号量的方式解决同步问题，模拟了较为简单和经典的生产者-消费者模型，实验结果符合预期设想。

本次实验难度适中，结合实验指导书与 PPT 中的内容，以及之前同学的上台讲解，基本已经扫除了实验中遇到的难点，实验的代码量也并不大。通过这次实验，对于 Nachos 的程序架构和工作方式有了更为具体的体会，希望能够对后续的实验起到帮助。

（三）实验 4-5 Nachos 文件系统

一. 目的与要求

1. 实验四

- Nachos 模拟了一个硬盘（../lab5/DISK，或../filesystem/DISK），实现的文件系统比较简单，该实验将熟悉一些文件系统的操作命令，观察这些命令对硬盘（DISK）的影响，根据结果分析理解 Nachos 文件系统的实现原理，为下一个实验（实验 5）实现 Nachos 文件的扩展功能奠定基础；
- 实验 5 要求能够在从一个文件的任何位置开始写入数据，即能够正确处理命令行参数 -ap, -hap, -nap；
- 该实验中需要理解一些 Nachos 文件的基本知识，特别是文件头（FCB 或索引节点）的结构与作用，空闲块的标识方法，空闲块的分配与回收过程等；
- 文件的扩展实质上是从一个给定的位置开始对文件进行写操作，涉及到文件的打开、定位，空闲块的分配等操作，写操作结束后还需要将文件头、空闲块位示图写到硬盘中，以保存修改后的信息。
- 该实验完成后，需要你：
 - （1）理解 Nachos 硬盘是如何创建的；

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

2. 实验 5

- 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;
- 目前 Nachos 实现的文件系统存在诸多限制, 其中之一是文件大小不能扩展, 即无法在已经存在的文件尾部追加数据; 该实验的任务就是让你修改 Nachos 的文件系统, 以满足:

(1) 文件创建时, 其大小可初始化为 0;

(2) 当一个文件写入更多的数据时, 其大小可随之增大;

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

(4) 需要理解 `../machine` 及 `../filesystem` 目录下的相关源代码 (`class Disk`、`class SynchDisk`、`class BitMap`、`class FileHeader`、`class OpenFile`、`class Directory` 及 `class FileSystem` 等) 以及 `../lab5` 下的两个模块 (`mian.cc` 及 `fstest.cc`), 分析 Nachos 不能对文件进行扩展的原因;

二. 源代码分析

1、Nachos 文件系统初探

通过../filesystems/filesys.h 中的代码（如图 1）可知，Nachos 实现的文件系统实现了两个版本，依据宏 FILESYS_STUB 与 FILESYS 条件编译产生两个不同的实现。宏 FILESYS_STUB 实现的文件操作直接利用 UNIX 所提供的系统调用实现，操作的不是硬盘 DISK 上的文件；宏 FILESYS 实现的文件系统是通过 OpenFile 类对 DISK 上的文件进行操作，最终也是使用 UNIX 的系统调用实现。考察../filesystems/makefile 及 makefile.local 可以看出，实验 4 与 5 默认是使用宏 FILESYS 所定义的实现，即在硬盘 DISK 上对文件进行操作。

```
#include "copyright.h"
#include "openfile.h"

#ifdef FILESYS_STUB
    // Temporarily implement file system calls as
    // calls to UNIX, until the real file system
    // implementation is available

class FileSystem {
public:
    FileSystem(bool format) {}

    bool Create(char *name, int initialSize) {
        int fileDescriptor = OpenForWrite(name);

        if (fileDescriptor == -1) return FALSE;
        Close(fileDescriptor);
        return TRUE;
    }

    OpenFile* Open(char *name) {
        int fileDescriptor = OpenForReadWrite(name, FALSE);

        if (fileDescriptor == -1) return NULL;
        return new OpenFile(fileDescriptor);
    }

    bool Remove(char *name) { return (bool)(Unlink(name) == 0); }
};

#else // FILESYS
class Filesystem {
public:
    Filesystem(bool format);

    // Initialize the file system.
    // Must be called *after* "synchDisk"
    // has been initialized.
    // If "format", there is nothing on
    // the disk, so initialize the directory
    // and the bitmap of free blocks.

    bool Create(char *name, int initialSize);
};
```

```

// Create a file (UNIX creat)
OpenFile* Open(char *name); // Open a file (UNIX open)
bool Remove(char *name); // Delete a file (UNIX unlink)
void List(); // List all the files in the file sy
void Print(); // List all the files and their cont

private:
OpenFile* freeMapFile; // Bit map of free disk blocks,
// represented as a file
OpenFile* directoryFile; // "Root" directory -- list of
// file names, represented as a file
};
#endif // FILESYS

```

2、编译 Nachos 的文件系统

在目录 code/filesys 下进入终端，运行 make，则会正确编译生成一个支持文件系统功能的 Nachos 系统。分析 code/filesys 中的 Makefile 文件可知，该文件包含了目录 code/threads 以及 code/filessys 中的 Makefile.local 文件。查看 code/filesys 中的 Makefile.local 文件可知，支持文件系统的 Nachos 系统除了使用 code/filesys 目录下的 C++文件外，还使用了 code/threads 以及 code/userprog 目录下的 C++文件。Make 工具通过 code 目录下 Makefile.common 中 VPATH 所定义的路径从这些目录中自动搜寻所需要的文件，以此编译生成支持文件系统功能的 Nachos 系统。

3、Nachos 的硬盘及文件系统

Nachos 利用 UNIX 的系统调用 open() 创建了一个名为“DISK”的文件作为 Nachso 的模拟硬盘。硬盘及文件系统具有以下特点：

- (1) 磁盘开始的 4 个字节（0~3 号字节）是硬盘标识（MagicNumber），其值为 0x456789ab，
指明该硬盘是一个 Nachos 硬盘；
- (2) 硬盘的第 4 个字节(序号从 0 字节开始)至第 131 字节为其第 0 号扇区(128 字节)，其后的 128 个字节为其第 1 号扇区，以此类推。即，如果字节序号从 0 字节开始，则每个扇区对应的字节序如表 1 所示。

表 1 Nachos 文件系统硬盘布局

扇区号	起止字节号	存储内容	大小
0	0x4~0x83	空闲块管理使用的位示图文件头	128 字节
1	0x84~0x103	目录文件头	128 字节

2	0x104~0x183	位示图文件数据块	128 字节
3	0x184~0x203	根目录表（目录文件）	128 字节
4	0x204~0x283	根目录表（目录文件）	128 字节
5	0x284~0x303	第一个文件的文件头	128 字节
6	0x304~0x383	第一个文件的数据块	128 字节
7	0x384~0x403	第一个文件的数据块	128 字节
8	

(3) 硬盘包括 32 个道,每个道包括 32 个扇区,每个扇区大小是 128 字节。故硬盘 DISK 共有 $32 \times 32 = 1024$ 个扇区,硬盘大小 ($\text{DiskSize} = (\text{MagicSize} + (\text{NumSectors} * \text{SectorSize})) = (4 + 32 \times 32 \times 128) \text{B} / 1024 = 0\text{x}80\text{KB}$ 。

(4) 每个逻辑块大小为 128 字节,与一个扇区对应。

(5) 采用一级目录结构（单级目录结构），最多可创建 10 个文件。

(6) 一个目录文件由“文件头+目录表”组成,目录文件 Directory 中的每个文件目录项包含三项。即:

`inUser;`//该目录项是否已经分配

`int sector;`//文件头所在的扇区号,这里文件头是 FCB 或者是 i-node

`char name[FileNameMaxLen+1];`//文件名,定义最长为 9 个字节,+1: 末尾 ‘\0’

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

2、系统初始化（创建）文件系统时,在目录文件中初始化了 10 个目录项,也就是说该文件系统中目前最多只能创建 10 个文件。

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

(7) 一个文件由“文件头+ 数据块组成”,每个文件最多包括 30 个扇区 ($\text{NumDirect} = (\text{SectorSize} - 2 * \text{sizeof(int)}) / \text{sizeof(int)}$) 因此每个文件最大为 3780 字节 (3KB) (30*128 字节)

(8) 文件在硬盘上的分配方法及文件头文件头相当于 FCB,说明文件的属性,以及文件的数据块在硬盘上的位置,在磁盘其结构如下:

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

文件在磁盘上的分配方法一般有三种：连续文件、链接文件及索引文件；

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

Nachos 一个文件头大小为 128 字节，恰好占用一个扇区（Nachos 有意将一个文件头存储到一个扇区中），由于 `int numBytes` 与 `int numSectors` 已经占用两个整数，因此直接块 `dataSectors[NumDirect]` 数组的最大项数由下式确定：

$$\text{NumDirect} = (\text{SectorSize} - 2 * \text{sizeof(int)}) / \text{sizeof(int)} = (128 - 2 * 4) / 4 = 30$$
 块，即每个文件的数据最多存储到 30 个扇区中，因此每个文件最大为 $30 * 128$ 字节 = 3KB，一个文件头大小 = $4 + 4 + 30 * 4 = 128$ 字节。

（9） 磁盘空闲块的管理

硬盘采用位示图（BitMap）的思想管理硬盘的空闲块，即根据硬盘的扇区数建立一个位置图，当一个扇区空闲，位示图中对应的位为 0，否则为 1；位示图也是一个文件，由文件头+数据块组成，文件头保存在第 0 号扇区中；由于硬盘共有 1024 个扇区，需要 1024 位表示每个扇区的状态，因此位示图文件大小应为 $1024 / 8 = 128$ 字节，故硬盘的位示图文件也可以恰好存储在一个扇区中。成员函数 `BitMap::FetchFrom()` 与 `BitMap::WriteBack()` 可以将位示图文件读入内存及写入硬盘。

（10） 目录文件的文件头存储在第 1 号扇区

在初始化文件系统时（`FileSystem` 类的构造函数），将 0 号和 1 号这两个特殊的扇区置位（已使用），然后将硬盘位示图文件头与目录表文件头写入到这两个特殊的扇区中；对于一个真正的操作系统，由于系统启动时需要根据目录文件的文件头访问根目录，因此为这两个特殊的结构分配到 0 号扇区与 1 号扇区这两个特殊的扇区中，是为了便于系统启动时从已知的、固定的位置访问它们。

（11） 当文件创建后，其大小不能改变；例如当复制一个文件到 DISK 中，该文件大小将无法改变。综上所述，Nachos 文件系统在硬盘 DISK 的布局如表 2 所示。

表 2 Nachos 的磁盘布局

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

4、Nachos 的文件系统命令

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

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

nachos [-d f] -cp UNIX_filename nachos_filename: 将一个 Unix 文件系统中的文件 UNIX_filename 复制到 Nachos 文件系统中,重新命名为 nachos_filename ; 目前实现的 Nachos 文件系统尚未提供 creat()系统调用,也就没有提供创建文件的命令。如果要在 Nachos 的硬盘中创建文件,目前只能通过该命令从 UNIX 系统中复制一个文件到 Nachos 硬盘中;

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

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

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

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

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

5、测试文件

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

5.1 UNIX 命令 od

在 UNIX 下的 `../filesystems` 目录下，运行 `od -c test/small`，屏幕将输出：

```
artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ od -c test/small
00000000  T  h  i  s  i  s  t  h  e  s  p  r  i
00000020  n  g  o  f  o  u  r  d  i  s  c  o  n
00000040  t  e  n  t  .  \n
00000046
```

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

5.2 UNIX 命令 hexdump

在 UNIX 目录 `../filesystems` 下，运行 `hexdump -c test/small`，屏幕将输出：

```
artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ hexdump -c test/small
00000000  T  h  i  s  i  s  t  h  e  s  p  r  i
00000010  n  g  o  f  o  u  r  d  i  s  c  o  n
00000020  t  e  n  t  .  \n
00000026
artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ hexdump -C test/small
00000000  54 68 69 73 20 69 73 20 74 68 65 20 73 70 72 69  |This is the spri|
00000010  6e 67 20 6f 66 20 6f 75 72 20 64 69 73 63 6f 6e  |ng of our discon|
00000020  74 65 6e 74 2e 0a                                |tent..|
00000026
```

偏移量以 16 进制表示

若使用命令 `hexdump -C test/small`，则会以 ASCII 形式输出文件内容

5.3 测试 Nachos 文件系统

linux 命令终端中进入目录 `../filesystems`，键入 `make` 生成支持文件系统的 Nachos，运行以下 `nachos` 命令，查看输出结果。

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

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

[illegible]

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

(c) 运行 `od -c DISK`，屏幕将输出以下转储信息：

```
artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ hexdump -c DI
SK
00000000  211  g  E 200  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
0000010  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000080  \0  \0  \0  \0  200  \0  \0  \0 002  \0  \0  \0 003  \0  \0  \0
0000090  004  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00000a0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000100  \0  \0  \0  \0  037  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000110  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0020004
```

(d) 运行 `hexdump -c DISK`, 屏幕将输出以下转储信息:

```
artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ hexdump -c DI
SK
00000000  211  g  E 200  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
0000010  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000080  \0  \0  \0  \0  200  \0  \0  \0 002  \0  \0  \0 003  \0  \0  \0
0000090  004  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00000a0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0000100  \0  \0  \0  \0  037  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000110  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0020004
```

(e) 运行 `hexdump -C DISK`, 屏幕将输出以下转储信息:

```

artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ hexdump -C DI
SK
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

```

(f) 运行 `nachos -cp test/small small`，将 `test` 目录下的 UNIX 文件 `small` 复制制到 Nachos 文件系统中；运行命令 `nachos -l`，`nachos -p small` 以及 `nachos -D`，根据输

出结果检查 Nachos 文件系统中是否存在文件 `small`（在硬盘 DISK 中存储）。

```

artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ ./nachos -cp
test/small small
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 131520, idle 130880, system 640, user 0
Disk I/O: reads 13, writes 8
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ ./nachos -l
small
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2550, idle 2420, 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...

artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ ./nachos -p s
mall
This is the spring of our discontent.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 5200, idle 4920, system 280, user 0
Disk I/O: reads 9, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

```


[illegible]

(g) 利用命令 `od -c DISK`, `hexdump -c DISK` 及 `hexdump -C DISK` 根据转储内容查看硬盘 DISK 有何变化

[illegible]


```

artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ hexdump -c DISK
00000000  211 g E 200 \0 \0 \0 001 \0 \0 \0 002 \0 \0 \0
00000010  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
00000080  \0 \0 \0 \0 211 \0 \0 \0 002 \0 \0 \0 003 \0 \0 \0
00000090  004 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
000000a0  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
00000100  \0 \0 \0 \0 177 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00000110  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
00000180  \0 \0 \0 \0 001 \0 \0 \0 005 \0 \0 \0 s m a l
00000190  l \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
000001a0  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
00000280  \0 \0 \0 \0 & \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
00000290  q \0 \0 \0 211 w 211 211 211 w 211 211 \0 \0 \0 \0
000002a0  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
000002f0  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 200 \0 \0 \0
00000300  020 \0 \0 \0 T h i s i s t h e
00000310  s p r i n g o f o u r d i
00000320  s c o n t e n t . \n \0 \0 \0 \0 \0 \0 \0
00000330  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
00200004
artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ hexdump -C DISK
00000000  ab 89 67 45 80 00 00 00 01 00 00 00 02 00 00 00 |..gE.....|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080  00 00 00 00 c8 00 00 00 02 00 00 00 03 00 00 00 |.....|
00000090  04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000100  00 00 00 00 7f 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000180  00 00 00 00 01 00 00 00 05 00 00 00 73 6d 61 6c |.....small|
00000190  6c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |l.....|
000001a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000280  00 00 00 00 26 00 00 00 01 00 00 00 06 00 00 00 |....&.....|
00000290  71 00 00 00 b0 77 da b7 b0 77 da b7 00 00 00 00 |q....W...W....|
000002a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
000002f0  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 |.....|
00000300  10 00 00 00 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 |.....|
*
00200004

```

根据 hexdump -C DISK 命令输出的屏幕内容可清楚地看到硬盘 DISK 的变化，即硬盘上多了 small 这个文件。

(h) 利用 nachos -cp test/medium medium，及 nachos -cp test/big big 将 UNIX 文件复制到 Nachos 文件系统中，使用 hexdump -C DISK 查看文件转储信息，通过 ./nachos -l 列出当前目录中的文件名

```
*
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  |....big.....|
000001c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
```

```
artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ ./nachos -l
small
medium
big
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

(i) 利用 `nachos -r` 命令, 删除 Nachos 硬盘上的 big 文件, 通过 `./nachos -l` 列出当前目录中的文件名, 可知文件 big 被成功删除。

```
artemis2@artemis2-VirtualBox:~/nachos/nachos-3.4-SDU/code/filesys$ ./nachos -l
small
medium
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2550, idle 2420, system 130, user 0
Disk I/O: reads 4, writes 0
Console I/O: reads 0, writes 0
```

这里需要再补充一下的内容就是文件的删除过程, 以删除 small 文件的过程为例:


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

①依据 small 文件头所提供的 small 文件数据块所在的扇区号,在位示图中清除 samll 文件数据块所占用的扇区,然后清除 samll 文件头所占用的扇区;

②清除目录表中为 samll 文件所分配的目录项中的 inUse 位

③可以看出,删除一个文件后,该文件在目录表中文件名、文件头所占的扇区号均未清除,只是将该目录项变为空闲;文件头中的信息(文件大小、文件所占用的扇区数以及为文件数据分配的扇区列表)也未清除;文件的内容也未清除;

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

三. 代码实现

1. Makefile 更改说明

首先先更改一下 makefile 和 makefile.local

makefile:

```
# NOTE: this is a GNU Makefile.  You must use "gmake" rather than "make".
#
# Makefile for the file system assignment
#   Defines set up assuming multiprogramming and virtual memory done first.
#   If not, use the "bare bones" defines below.
#
# Copyright (c) 1992 The Regents of the University of California.
# All rights reserved.  See copyright.h for copyright notice and limitation
# of liability and disclaimer of warranty provisions.

ifndef MAKEFILE_FILESYS
define MAKEFILE_FILESYS|
yes
endif

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

include ../threads/Makefile.local
include ../lab5/Makefile.local
#include ../userprog/Makefile.local
#include ../vm/Makefile.local
#include ../fileSYS/Makefile.local

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

endif # MAKEFILE_FILESYS
```

makefile.local

```

ifndef MAKEFILE_FILESYS_LOCAL
define MAKEFILE_FILESYS_LOCAL
yes
endif

# Add new sourcefiles here.

CCFILES +=bitmap.cc\
        directory.cc\
        filehdr.cc\
        filesys.cc\
        fstest.cc\
        openfile.cc\
        synchdisk.cc\
        disk.cc\
main.cc\
fstest.cc

ifdef MAKEFILE_USERPROG_LOCAL
DEFINES := $(DEFINES:FILESYS_STUB=FILESYS)
else
INCPATH += -I../userprog -I../lab5
DEFINES += -DFILESYS_NEEDED -DFILESYS
endif

endif # MAKEFILE_FILESYS_LOCAL

```

接下来对文件系统简单进行理解:

[illegible]

根据上述系统转储信息，我们可以看出：

(1) bitmap 文件在 Nachos 硬盘的第一个扇区中，大小是 128 字节（一个扇区），从中可以看出文件 bitmap 的 i-node 默认存储在硬盘的第 0 号扇区中。

(2) 目录文件（文件目录表）占用了 200 字节，是两个扇区，被安排在 3,4 扇区中，其 i-node 处于 1 号扇区中。

(3) 文件 small 大小是 42 字节，占用一个扇区，它的 i-node 在 5 号扇区，数据块存储在 6 号扇区中。

`./nachos -ap` 实现将 Unix 文件附加到 Nachos 文件之后

`./nachos -hap` 实现将 Unix 文件从 Nachos 文件的中间开始写入，并且覆盖掉 NachosFile 后半部分的内容。

`./nachos -nap` 实现将 FromNachos 文件附加到 ToNachos 文件尾部。

3.1 nachos -ap 与 nachos -hap 命令的实现

`fstest.cc` 中的 `Append()` 调用了 `OpenFile::Write()`, `Write()` 又调用了 `OpenFile::WriteAt()`, `WriteAt()` 函数就是试图在文件尾追加另一个文件内容。

分析原文件系统给的 `WriteAt()` 函数，此函数有两个约束，第一是如果开始写入的位置是文件尾，则函数会退出；第二是如果写入的数据过多，超过了原文件剩余的长度，那么超出部分也不会再写入，同样会使文件扩展操作无法进行。

如果取消上述的两个约束，数据就可以在文件尾部追加，但是由于文件数据扩展之后，文件头三元组<文件长度，占用扇区数，扇区列表>中的数据没有更新，所以这种扩展是无意义的。且超出文件最后一个扇区的位置的字节也不知道写在何处，会产生较为严重的后果。

因此，我们将要做的工作有如下几步：

修改 `OpenFile::WriteAt()` 函数：

(1) 修改两个约束

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

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

上述步骤涉及到的内容由 `OpenFile` 类，`FileHeader` 类，`BitMap` 类，`FileSystem` 类以及 `fstest.cc` 中的 `Append()` 和 `Nappend()`；

具体的修改如下：

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

(2) 修改 `FileSystem` 类，添加空闲块位示图文件的硬盘读写操作。

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

(4) 修改 `FileHeader::Allocate()`，为添加的数据分配扇区；

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

接下来给出具体的实现方案：

1. 修改 `OpenFile::WriteAt()`

目的：允许从文件尾开始写数据

修改后的代码如下图：

```

    if ((numBytes <= 0) || (position > fileLength)) //约束1，将等于号去掉
        return 0; // check request
    //约束2如果条件 (coition>fileLength)成立，说明文件需要扩展
    if ((position + numBytes) > fileLength)
    {
        int incrementBytes=(position+numBytes)-fileLength; //多出来需放在新扇区
        BitMap *freeBitMap=fileSystem->getBitMap(); //读取位视图，新的空闲磁盘块
        hdr->Allocate(freeBitMap, fileLength, incrementBytes); //分配新的磁盘块
        fileSystem->setBitMap(freeBitMap); //写回位视图
    }

```

其余部分不需要修改。

而且中的 `getBitMap()`, `Allocate()`, `setBitMap()` 三个函数需要继续修改或重新创建。

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

目的：从硬盘读取空闲快位示图文件，内容被修改后再将其写会硬盘；

类 `FileSystem` 在其构造函数中，维护了两个一直处于打开的状态的文件句柄，`OpenFile* freeMapFile, ,directoryFile`，前者是硬盘上的位示图文件，一个是硬盘上的目录文件，可以直接使用它们对位示图文件和目录文件进行读写操作（读写函数是 `FetchFrom(OpenFile*)`, `WriteBack(OpenFile*)`），因为这两个 `OpenFile` 对象是 `FileSystem` 类的私有变量，因此需要在那个 `FileSystem` 类中定义实现这 `getBitMap()`，和 `setBitMap()`；

其中代码部分如下图。

```

1  }
2  Bitmap* FileSystem::getBitMap(){
3      //读取位视图文件
4      Bitmap* freeBitMap=new Bitmap(NumSectors);
5      freeBitMap->FetchFrom(freeMapFile);
6      return freeBitMap;
7  }
8  void FileSystem::setBitMap(Bitmap* freeMap){
9      freeMap->WriteBack(freeMapFile); //回写位视图
10 }
11

```

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

目的：将修改后的文件头写会硬盘。

原文件系统中, FetchFrom(int x) 是从硬盘的扇区 x 中读取一个文件的头文件信息, WriteBack(int x) 将一个文件的头文件写到硬盘的扇区 x 中。

OpenFile 类维护了一个 FileHeader 类对象 hdr, 原文件系统的构造函数从硬盘的扇区 sector 读取该文件的文件头 (FCB), 并将读写指针 (偏移量) 设置为开始位置 (0)。此时我们需要定一个 hdrSector 整数, 记录该文件的文件头所在的扇区号, 便于 FerchFrom 和 WriteBack 函数的调用。具体的代码如下:

```

OpenFile::OpenFile(int sector)
{
    hdr = new FileHeader;
    hdr->FetchFrom(sector);
    seekPosition = 0;
    hdrSector=sector; //记录该文件头所在扇区号
}

```

```

void OpenFile::WriteBack(){
    hdr->WriteBack(hdrSector); //在文件头改写后将其回写到硬盘的扇区
}

```

4. 修改 FileHeader::Allocate()

目的：为要写入的文件数据分配硬盘空间, 写入的数据要可以利用文件的最后的一个扇区的剩余空间, 也可以为其新分配扇区 (硬盘块)

(1) 重写 FileHeader 构造函数

原默认的构造函数有些无效数据, 可能会对后面的输出有影响, 所以需要重构构造函数, 其中文件头需要包含以下的内容, 文件的大小, 文件所占的扇区数, 文件扇区索引表。具体如下:

```
FileHeader::FileHeader(){
//构造函数
//默认下, 为一个文件所分配的文件头在有效数据之后可能含有一些无用的数据, 影响数据分析
//在此对文件头内容进行清除
numBytes=0;//文件大小
numSectors=0;//文件扇区数
for(int i=0;i<NumDirect;i++){
dataSectors[i]=0;//文件扇区索引表清零
}
}
```

(2) 重写 Allocate 函数:

这个函数会根据文件的大小为文件分配所需要的所有扇区块, 并在位示图中标记所分配的扇区块, 设置头文件三元组<numBytes, numSectors, dataSectors[30]>, 重载 FileHeader 函数后, 增加输入参数 incrementByte, 意为要扩展的数据大小, 判断是否需要分配新的扇区块, 并更新文件头的三元组。

具体的代码如下图:

```
//进行重载
bool FileHeader::Allocate(BitMap *freeMap, int fileSize, int incrementBytes){
if((fileSize==0)&&(incrementBytes>0)){
//在一个空文件后追加数据
if(freeMap->NumClear()<1){
//判断是否有空闲块
printf("Insufficient Disk Space.\n");
return false;}
dataSectors[0]=freeMap->Find();
numSectors=1;
}
numBytes=fileSize;
int offsetr=numBytes%SectorSize;//原文件最后一个扇区块数据偏移量
int newSectorBytes=incrementBytes-(SectorSize-offsetr);//判断是否需要分配新的扇区
if(newSectorBytes<=0){
numBytes=numBytes+incrementBytes;//更新文件大小
return true;}
int moreSectors=divRoundUp(newSectorBytes, SectorSize);//新加扇区数? divRoundUp
if(freeMap->NumClear()<1){
//是否有空闲扇区
printf("Insufficient Disk Space.\n");
return false;}
for(int i=numSectors;i<numSectors+moreSectors;i++){
dataSectors[i]=freeMap->Find();//更新扇区索引表
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 fstest.cc 中的 Append()

(1) 修改写指针


```

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);
    ASSERT(result == amountRead);
    start += amountRead; //使每次操作都是从上次写入的数据之后的位置开始进行
    // ASSERT(start == openFile->Length());
}

```

取消 start 那句的注释, 是的每次操作都是从上次写入的数据之后的位置开始进行。

(2) 将文件写回硬盘

在 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。

```

154
155 // Write the inode back to the disk, because we have
156     openFile->WriteBack();
157 // printf("inodes have been written back\n");
158

```

6. ./nachos -nap 的实现

NAppend 函数也是调用了 OpenFile::Write 实现文件的写操作, 由于已经修改了其相应的函数, 所以已经实现了可以从 nachos 文件的任务位置开始写入数据。

因此只需要做和 Append 一样的操作就可以了。

```

226     openFileFrom->Seek(0);
227     while ( (amountRead = openFileFrom->Read(buffer, TransferSize)) > 0)
228     {
229         int result;
230         // printf("start value: %d, amountRead %d, ", start, amountRead);
231         // result = openFile->WriteAt(buffer, amountRead, start);
232         result = openFileTo->Write(buffer, amountRead);
233         // printf("result of write: %d\n", result);
234         ASSERT(result == amountRead);
235         start += amountRead;
236         // ASSERT(start == openFile->Length());
237     }
238     delete [] buffer;
239
240 // Write the inode back to the disk, because we have changed it
241 openFileTo->WriteBack();
242 // printf("inodes have been written back\n");

```

四.运行结果及分析

(1) 为了便于测试, 我们将 test 下的 small, medium 和 big 文件进行一定的修改。修改后的样子如下:

(3) nachos -D 及 hexdump -C DISK 考察 Nachos 在硬盘 DISK 上初始化的文件系统情况；包括空闲快位示图的头文件 (0)，空闲快位示图文件数据块 (2)，目录表头文件 (1)，目录表数据块 (3, 4)

[illegible]

(4) `./nachos -cp test/small small` 复制 test 目录下的 UNIX 文件 small 到 DISK 中。

[illegible]

此时可以看出已经将 small 文件从 Unix 下复制到 Nachos 系统下了。small 的头占 5 号扇区，文件占 6 号扇区。位示图表示 0, 1, 2, 3, 4, 5, 6 已经被占了。

(5) `./nachos -ap test/small small` 测试给一个已存在的文件追加数据。即把 unix 文件附加到一个 nachos 文件后。

[illegible]

可以看出，已经将文件附加成功，由于文件较小，所以附加之后不需要新增扇区。

(6) `./nachos -ap test/big small`: 将 big 文件附加到一个 small 中, 测试为文件分配新的扇区的功能

[illegible]

可以看出扩展了 7, 8, 9, 10, 11 这些扇区

(7) `./nachos -ap test/medium medium` 测试给一个空文件追加数据的功能，此时 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：

[illegible]

(8) `./nachos -hap test/medium small`, 测试从 `small` 中间写入文件的功能。

[illegible]

可以看出 medium 直接插入到 small 文件的后面，并没有新的文件头等。

(9) `./nachos -nap medium small` 测试将一个 nachos 文件附加到另一个 nachos 文件的功能。

[illegible]

可以看出在 medium 附加到了 small 的尾部。

(10) `./nachos -r small` 测试文件删除的功能

```
File contents:  
|1|\0\0\0\5|\0\medium\0\0\0\0\0\0\8\0\0\small\0\0\0\0\0\0\0\0\0\0\0\0  
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0  
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0  
\0\0\0
```

```
Bitmap set:  
0, 1, 2, 3, 4, 5, 6, 7,  
Directory contents:  
Name: medium, Sector: 5  
FileHeader contents. File size: 177. File blocks:  
6 7  
File contents:  
This is a medium file\aThis is a medium file\nThis is a medium file\nThis is a m-  
edium file\nThis is a medium file\nThis is a medium f  
ile\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file\nThis is a medium file  
No threads ready or runnable, and no pending interrupts.  
Assuming the program completed.
```

```
Ticks: total 6900. idle 6500. system 400. user 0
```

与上图对比，可以看出 small 文件被删除了。包括对应的位示图都已经变了。

(11) 反复运行 `nachos -ap` 或者 `nachos -cp` 在硬盘上 DISK 新建文件，测试 nachos 文件系统最多可创建多少文件。因为 nachos 采用一级目录，最多有 10 个目录项，因此最多存储 10 个文件。

(12) nap 运行结果

./nachos -nap test/medium small

测试结果如下：

```
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
Directory contents:
Name: small, Sector: 5
FileHeader contents. File size: 424. File blocks:
6 7 8 9
File contents:
small file small file small file\asmall file small file small file\***end of fi
le***\abig file big file big file big file big file
\abig file big file big file big file big file \abig file big file big file big
file medium file medium file medium file\amedium f
ile medium file medium file\***end of file***\ae big file \abig file big file b
ig file big file big file \abig file big file big fi
le big file big file \***end of file***\a
Name: medium, Sector: 10
FileHeader contents. File size: 90. File blocks:
11
File contents:
medium file medium file medium file\amedium file medium file medium file\***end
of file***\a
```

可以看到 Linux 文件 medium 的内容成功插入文件 small 中。

最后测试-nap 功能，该功能将 nachos 的文件附加到 nachos 文件尾部，输入如下指令：

./nachos -nap medium small

测试结果如下：

```
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
Directory contents:
Name: small, Sector: 5
FileHeader contents. File size: 514. File blocks:
6 7 8 9 12
File contents:
small file small file small file\asmall file small file small file\***end of fi
le***\abig file big file big file big file big file
\abig file big file big file big file big file \abig file big file big file big
file medium file medium file medium file\amedium f
ile medium file medium file\***end of file***\ae big file \abig file big file b
ig file big file big file \abig file big file big fi
le big file big file \***end of file***\amedium file medium file medium file\am
edium file medium file medium file\***end of file**
*\a
Name: medium, Sector: 10
FileHeader contents. File size: 90. File blocks:
11
File contents:
medium file medium file medium file\amedium file medium file medium file\***end
of file***\a
```

可以看到，由于在 small 后面附加 medium 需要开辟新的扇区，small 当前已经占用的最大扇区号是 9 号扇区，而 10 号和 11 号扇区已经被 medium 占用，因此 nachos 文件系统为 small 新附加的内容分配了 12 号扇区。这也体现了 nachos 的扇区分配策略。

五. 收获与心得

实验四是对 nachos 文件系统的初探，整个实验过程中需要对大量的代码进行阅读并理解，实验难度较大，进行时间也较长。最后对 nachos 的文件系统有了一个较为全面的理解，理解了一个文件系统如何通过空闲块管理、目录索引等多项机制来组织起当前系统内的文件。nachos 通过一个虚拟磁盘向我们展示了一个简单的文件系统，通过实验四为下面实验五的顺利进行奠定了基础。

通过实验五，对文件系统的读写有了进一步的理解，首先仔细阅读代码发现当前文件系统为何无法进行扩展，之后通过编程实现文件的扩展，了解到文件扩展需要位图、目录、文件头等多方面的同步更新。在实验过程中遇到测试时出现“段转储错误”的问题，发现问题出现在写指针出现在扇区边界时接下来无法正确定位，会回到 0 号扇区，此时继续写操作会覆盖位图的头文件，导致-D 命令无法正确执行。修改方法已在报告中指出，需要添加关于边界处理的判断。

实验五使自己对文件系统的理解进一步加深，了解了一个简单的文件系统如何通过各种数据结构来规划文件的读写组织等各项操作，收获颇丰。

（四）实验 6-8 Nachos 用户程序及系统调用

一. 目的与要求

1. 实验六：

- 分析 Nachos 可执行程序.noff 文件的格式组成
- 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。
- 掌握如何利用交叉编译生成 Nachos 的可执行程序
- 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。

- 阅读 `userprog/progtest.cc`, `threads/scheduler.cc` 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和方法。
- 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。

2.实验七:

- 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

① 阅读 `./prog/protest.cc`，深入理解 Nachos 创建应用程序进程的详细过程

② 阅读理解类 `AddrSpace`，然后对其进行修改，使 Nachos 能够支持多进程机制，

允许 Nachos 同时运行多个用户线程；

③ 在类 `AddrSpace` 中添加完善 `Print()`函数（在实验 6 中已经给出）

④ 在类 `AddrSpace` 中实例化类 `Bitmap` 的一个全局对象，用于管理空闲帧；

⑤ 如果将 `SpaceId` 直接作为进程号 `Pid` 是否合适？如果感觉不是很合适，应该如何为进程分配相应的 `pid`？

⑥ 为实现 `Join(pid)`，考虑如何在该进程相关联的核心线程中保存进程号；

⑦ 根据进程创建时系统为其所做的工作，考虑进程退出时应该做哪些工作；

⑧ 考虑系统调用 `Exec()`与 `Exit()`的设计实施方案；

3.实验八:

- 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）

二. 源代码分析

0.前期工作

通过../filesystem/filesys.h 中的代码（如图 1）可知，Nachos 实现的文件系统实现了两个版

本，依据宏 FILESYS_STUB 与 FILESYS 条件编译产生两个不同的实现。宏 FILESYS_STUB 实现的文件操作直接利用 UNIX 所提供的系统调用实现，操作的不是硬盘 DISK 上的文件；宏 FILESYS 实现的文件系统是通过 OpenFile 类对 DISK 上的文件进行操作，最终也是使用 UNIX 的系统调用实现。考察../userprog/makefile 与 makefile.local 的内容可以看出，实验 6.7.8 默认使用的是 FILESYS_STUB 定义的相关实现，是对 UNIX 文件进行操作：


```

#include "copyright.h"
#include "openfile.h"

#ifdef FILESYS_STUB

// Temporarily implement file system calls as
// calls to UNIX, until the real file system
// implementation is available

class FileSystem {
public:
    FileSystem(bool format) {}

    bool Create(char *name, int initialSize) {
        int fileDescriptor = OpenForWrite(name);

        if (fileDescriptor == -1) return FALSE;
        Close(fileDescriptor);
        return TRUE;
    }

    OpenFile* Open(char *name) {
        int fileDescriptor = OpenForReadWrite(name, FALSE);

        if (fileDescriptor == -1) return NULL;
        return new OpenFile(fileDescriptor);
    }

    bool Remove(char *name) { return (bool)(Unlink(name) == 0); }
};

#else // FILESYS
class FileSystem {
public:
    FileSystem(bool format);

    // Initialize the file system.
    // Must be called *after* "synchDisk"
    // has been initialized.
    // If "format", there is nothing on
    // the disk, so initialize the directory
    // and the bitmap of free blocks.

    bool Create(char *name, int initialSize);

    // Create a file (UNIX creat)

    OpenFile* Open(char *name);

    // Open a file (UNIX open)

    bool Remove(char *name);

    // Delete a file (UNIX unlink)

    void List();

    // List all the files in the file sy

    void Print();

    // List all the files and their cont

private:
    OpenFile* freeMapFile;

    // Bit map of free disk blocks,
    // represented as a file

    OpenFile* directoryFile;

    // "Root" directory -- list of
    // file names, represented as a file
};

#endif // FILESYS

```

1. Nachos 可执行程序格式

```
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 */
}
```

Nachos 的应用程序其实是一种文件类型。一个可执行程序由三个段组成，分别是代码段，初始数据段（一般为初始化的全局变量等），未初始化的数据段，一般是指程序运行是才会分配的动态内存，静态变量和未初始化的全局变量等。

```
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;
```

每个段都定义了大小（size），在文件中的位置（inFileAddr），程序的入口地址（virtualAddr）。

AddrSpace 的构造函数在将 NOFF 文件装入内存之前，先打开文件，读入文件头，然后根据文件头信息确定程序所占用的空间，将相应的段装入内存，将程序的入口地址，给 PC 赋值。

其中在构造函数中，将应用程序装入内存后，就建立了这个用户程序的页表，实现虚页与实页的对应关系。页表实现了续页与事业的对应关系，系统根据页表实现存储保护，页面置换算法根据页表信息进行页面置换等操作。

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

每一页表项有虚拟页号，物理页号，空标志位，只读位，是否被引用或修改位和是

否被修改位。

2. 理解 Nachos 应用程序的启动与执行过程

(1) Nachos 的进程是通过形成一个地址空间从线程演化而来的，且其存储管理采用分页管理方式，所以当 Nachos 为一个应用程序新建一个地址空间后，打印一下这个程序的页表（页面与帧的映射关系）将会有助于后续程序的调试。

所以在类 AddrSpace 中定义一个 Print 成员函数，如下：

```
void AddrSpace::Print() {
    printf("page table dump:  %d pages  in total\n", numPages);
    printf("=====\n"); printf("\tVirtPage, \tPhysPage\n");

    for (int i=0; i < numPages;  i++) {
        printf("\t %d, \t\t%d\n", pageTable[i].virtualPage, pageTable[i].physicalPage);
    }
    printf("=====\n\n");
}
```

每当为一个新的应用程序新建一个空间的时候，就调用 Print() 函数，输出页表信息。

如果想为页表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。如下：

```
static int a[40];
int
main()
{
    Halt();
    /*  not reached */
}
```

(2) 阅读与理解 progtest.cc 中的 StartProcess 函数

```

void
StartProcess(char *filename)
{
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;

    if (executable == NULL) {
        printf("Unable to open file %s\n", filename);
        return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;

    delete executable;          // close file

    space->InitRegisters();      // set the initial reg
    space->RestoreState();       // load page table reg

    machine->Run();              // jump to the user program
    ASSERT(FALSE);              // machine->Run never returns
    // the address space exits
    // by doing the syscall "exit"
}
    
```

①系统要运行一个应用程序，需要为该程序创建一个用户进程，为程序分配内存空间。首先将执行文件从外存中读入，然后将用户程序（三个段）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）。此时的 `space` 就是该进程的一个标识。

②为便于上下文切换时保存与恢复寄存器状态，Nachos 设置了两组寄存器，一组是 CPU 用的 registers，另一组是用户寄存器 userRegisters，由于 CPU 只有一个，因此 CPU 寄存器也只有一套，因此每个核心线程都需要设置一组用户寄存器，用于保存和恢复相应的用户程序指令的执行状态。当用户进程进行上下文切换时，老进程的 CPU 寄存器状态保存为用户寄存器中，并将新用户进程的寄存器状态恢复到 CPU 寄存器中。

其中 `space->InitRegisters` 用于初始化 CPU 寄存器，`space->RestoreState` 用于将用户进程的页表传递给系统核心，以便 CPU 能从用户进程的地址空间中读取到应用程序指令。

以上就是将用户进程映射到一个核心进程（将用户进程的页表给核心进程的页表，将用户进程的寄存器内容给核心进程的寄存器内容），由于 Nachos 只有一个核心进程，所以核心进程就是 CPU 进程。

当前进程的页表如何与 CPU 使用的页表进行关联，详细内容可以看本节的 4.

③machine->Run():程序正式开始执行：从程序入口开始，完成取指令，译码执行的过程，知道进程遇到 Exit 语句或者异常才退出。

```
run(mipssip)
```

```
<-oneInstruction(machine/mipssim)//对读出的指令进行译码并且执行
```

```
<-readmem(translate.cc)//根据物理地址从内存中读出一条指令
```

```
<-translate(machine/translate.cc)//将虚拟地址进行转换为物理地址
```

其中，一个进程有一个页表（对应进程用到的全部内存）页表中的某些虚拟页号和对应的帧号存在快表中，如果快表中没有的话，不代表内存中没有。

translate 中，先检查 TLB 是否为空，如果为空就返回页错误，让系统去页表中查，查到的话就可以将虚拟地址转换为物理地址。

Nachos 目前不支持多进程。

3. 如何进行系统调用

执行时返回 SystemcallException 到 RaiseException(machine.cc) 中，然后调用 ExceptionHandler(exception.cc) 对系统调用进行处理。

```
break,
.....
case OP_SYSCALL:
    RaiseException(SyscallException, 0);
return;
.....
default:
    ASSERT(FALSE);
```

```
void
Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG('m', "Exception: %s\n", exceptionNames[which]);
    // ASSERT(interrupt->getStatus() == UserMode);
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0); // finish anything in progress
    interrupt->setStatus(SystemMode);
    ExceptionHandler(which); // interrupts are enabled at this point
    interrupt->setStatus(UserMode);
}
```

```
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 the file to be executed
                char filename[50];
                int addr=machine->ReadRegister(4);
                int i=0;
                do{
                    //read filename from mainMemory
                    machine->ReadMem(addr+i, (int*) &filename[i])
                } while (filename[i] != '\0');
            }
        }
    }
}
```

4. Nachos 进程的相关信息：

Nachos 没有显示的实现 PCB，而是将进程的信息分散到相应的类对象中，例如利用所分配内存空间的对象指针标识一个进程，该对象中含有进程的页表，栈指针，与核心线程的映射等信息。

进程的上下文保存在核心线程中，当一个线程被调度执行后，依据线程所保存的进程上下文中执行所对应的用户进程。

如果遇到进程之间的切换，

```
Scheduler::Run (Thread *nextThread)
{
    Thread *oldThread = currentThread;

#ifdef USER_PROGRAM // ignore until running user programs
    if (currentThread->space != NULL) { // if this thread is a user program,
        currentThread->SaveUserState(); // save the user's CPU registers
        currentThread->space->SaveState();
    }
}
#endif
```

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

④程序之间进行切换，首先现将进程放到就绪队列里面，run 函数进行上下文切换（在 scheduler 里面），在 threads.cc 中有将 CPU 寄存器内容存到用户寄存器中的操作

saveuserstate()，有将用户寄存器中的内容重新载入 CPU 中的函数 restoreuserstate()；在 addrSpace 里面有将用户页表载入内核 restorestate()，并且将内核中的页表存回内存的函数操作 savestate()。

三. 代码实现

1.在类 AddrSpace 中添加完善 Print()函数，便于后续查看也表分配的情况。Print()函数如下所示

```
void AddrSpace::Print(){
    printf("page table dump: %d pages in total\n",numPages);
    printf("=====\n");
    printf("\tVirtPage,\tPhysPage\n");
    for(int i=0;i<numPages;i++){
        printf("\t%d,\t\t%d\n",pageTable[i].virtualPage,pageTable[i].physicalPage);
    }
    printf("=====\n\n");
}
```

打印的结果如下图所示

```
page table dump: 12 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
   11,         11
=====
```

2.理解系统如何管理空闲帧

```
// first, set up the translation
pageTable = new TranslationEntry[numPages];
for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE; // if the code segment was entirely on
                                    // a separate page, we could set its
                                    // pages to be read-only
```

根据上述分配规则，发现 Nachos 为一个应用程序分配内存时，总是将程序的 0 号页面分配给 0 号帧，1 号页面分配给 1 号帧，以此类推。因此如果执行 exec 这样的系统调用时，新的程序仍然会被从 0 号开始分配内存，这样的话原程序的内存空间就会被覆盖，程序就无法正常执行。

3.建立 BitMap 全局变量对空闲帧进行管理。

如果想让 Nachos 同时运行多个用户线程，就需要让 Nachos 为新的进程分配空间的时候不要再从头开始分配。这时在 addrspace 中建立一个 BitMap 全局变量对 Nachos 进行空闲帧的管理。

```
Bitmap *pageMap;
```

在一开始初始化一个 pageMap，让其数量为内存中物理帧的数量。

```
//ASSERT(flag);
if(pageMap==NULL) {
    pageMap=new Bitmap(NumPhysPages);
    printf("pageMap was cleared\n");
}
```

当为新进程分配帧的时候，从尚未被分配的帧中选择新帧进行分配，这样就不会有每次都从头开始分配的结果了。如下所示。

```

first, set up the translation
pageTable = new TranslationEntry[numPages];
for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i; // for now, virtual page =
    pageTable[i].physicalPage = pageMap->Find();
    printf("page %d\n", pageTable[i].physicalPage);
    ASSERT(pageTable[i].physicalPage!=-1);
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE; // if the code segment was
    // a separate page, we could set its

```

4.为进程分配相应的 pid，对 SpaceID 的修改

在系统调用中，Exec（）返回类型是 SpaceID，是一个类似于 Pid 可以表示一个进程的数。StartProcess()中可以看出，加载运行一个应用程序的过程就是首先打开这个程序文件，为该程序分配一个新的内存空间，并将改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。

也就是说，为进程创建的地址空间的首地址是唯一的，理论上可以作为 pid 唯一标识当前进程，但是该值不连续，且值过大，所以这样做比较麻烦。

我们可以为一个地址空间或地址空间对应的进程分配一个唯一的整数，叫 spaceID，比如用户进程号从 100 开始。每当创建一个新进程，就为执行文件依次分配一个进程号。当系统调用 Exit 退出是，应当收回为其分配的进程号。具体 spaceID 的定义方式如下代码。

```

AddrSpace::AddrSpace(OpenFile *executable)
{
    for(int i=100;i<MAX_USERPROCESSES;i++){
        if(!ThreadMap[i]){
            ThreadMap[i]=1;
            //flag=true;
            spaceID=i;
            break;
        }
    }
}

```

定义一个全局的 ThreadMap 数组，用来管理进程号，如果这个 ID 已经被分配出去，就将其变为 1，每次从 0 的数里面依次选为新进程分配 spaceId,并将 ThreadMap 对应的数也置 1.

5.如何保存进程号

在 SpaceAdress 类中，写一个 getSpaceID()的函数，用来返回当前进程的进程号。如下：

```

void Print(); //print the page of current program
int getSpaceID(){return spaceID;}
private:

```

6.进程退出时的工作

在析构方法中要释放空间。并且把 ThreadMap 对应的进程号设为 0.

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

7. 实现系统调用的前期工作

当 Nachos 模拟的 CPU 检测到该条指令是执行一个 Nachos 的系统调用时，则抛出一个异常 `SystemException` 以便从用户态陷入到核心态去处理这个系统调用。但是一条指令正常执行结束之后，需要将 PC 推进，指向下一条指令，但后不是一条 `break` 语句，而是 `return`，原因是通常情况下，处理完异常之后需要重启这条指令，但是系统调用异常是个特例，所以异常处理结束之后指令不需要重启。

避免重启的方法是在系统调用处理程序中添加 PC 的推进操作。具体代码如下：

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

通过分析，Mips 的架构是将寄存器 2 和寄存器 3 存放返回值，寄存器 4-7 用来传递参数，如果存的是字符串的话，存字符串的地址。

系统调用号存在 2 号寄存器中（具体的 Nachos 如何实现系统调用的过程见实验六报告）

8.SC_Exec()的实现

（1）首先获取系统调用参数，即 `Exec()` 的参数 `filename` 在内存中的地址存到 4 号寄存器中获取。

（2）利用 `Machine::ReadMem()` 从该地址读取应用程序文件名 `filename`。

（3）打开应用程序并为其分配内存空间，创建页表，分配 `spaceID`，就为应用程序创建了一个进程。

（4）创建一个核心线程，将该应用程序映射到核心线程。

系统将 `main()` 对应主程序映射到核心线程 `main`，让主线程执行主程序。对于应用程序，需要自己调用 `Thread::Fork()` 创建一个核心线程。并将应用程序映射到该核心线程，以执行应用程序

(5) 最后的返回值 spaceID 返回到二号寄存器中。

(6) PC 向后加。

```

}
case SC_Exec:{
    printf("Execute system call of Exec()\n");
    //read the file to be executed
    char filename[50];
    int addr=machine->ReadRegister(4);
    int i=0;
    do{
        //read filename from mainMemory
        machine->ReadMem(addr+i,1,(int*)&filename[i]);
    }while(filename[i++]!='\0');
    printf("Exec(%s):\n",filename);
    //openFile
    OpenFile *executable=fileSystem->Open(filename);
    if(executable==NULL){
        printf("Unable to openFile%s\n",filename);
        return;
    }
    space=new AddrSpace(executable);
    delete executable;
    //new and fork thread
    char* forkedThreadName=filename;
    thread=new Thread(forkedThreadName);
    thread->Fork(StartProcess,space->getSpaceID());
    thread->space=space->getSpaceID();

    //run the new thread
    //currentThread->Yield();

    //return spaceID
    machine->WriteRegister(2,space->getSpaceID());
    AdvancePC();
    break;
}

```

9.SC_Exit()的实现

首先从 4 号寄存器中读出参数，释放对应的 pid，调用 currentThread->Finish 结束该进程对应的线程。释放对应的程序地址空间。

但是空闲帧的位示图以及 Threadmap 结构不能释放，因为他们是全局的。具体的代码如下：

```

case SC_Exit:{
    printf("Execute system call of Exit()\n");
    delete currentThread->space;
    currentThread->Finish();
    AdvancePC();
    break;
}

```

10.SC_Yield () 的实现

```

}
case SC_Yield:{
    currentThread->Yield();
    AdvancePC();
    break;
}

```

11. SC_Join () 的实现

父进程依据所等待的 spaceId 检查其对应的线程是否已经执行完毕,如果尚未退出,则将当前进程进入等待队列,然后调用 Thread::Sleep()使当前进程睡眠,被唤醒后返回进程的退出码,如果所等待的进程已经结束,Join 就直接返回,不需要等待。

(1) 如何检查所等待的进程是否已经退出

当一个进程结束的时候,会进行 Exit() 系统调用,而根据上文可知,此系统调用会使线程进入调用 Finish 函数,因此,我们只需要给进程多加一个状态——Terminated 状态,在 Finish() 中,将进程的状态置为 Terminated。如下图:

```

9 void
10 Thread::Finish ()
11 {
12     (void) interrupt->SetLevel(IntOff);
13     ASSERT(this == currentThread);
14
15     DEBUG('t', "Finishing thread \"%s\"\n", getName());
16
17     //if there some thread wait for wakeup!
18     Thread* t;
19     int i=scheduler->waitList->GetSize();
20     while(i!=0){
21         t=(Thread *)scheduler->waitList->Remove();
22
23         if(t->waitThread==this)
24             scheduler->Run(t);
25         else
26             scheduler->waitList->Append((void *)t);
27         i--;
28     }
29
30     threadToBeDestroyed = currentThread;
31     currentThread->setStatus(TEMINATED);
32     Sleep(); // invokes SWITCH
33     // not reached
34 }
35
36 //-----
37
38

```

(2) Join 函数

首先判断,等待结束的程序是否已经结束,如果结束了,则将主进程的等待进程置为空,如果没有结束,则将这个进程置为主进程的等待进程,并且进入循环判断,等待进程是否结束。(判断结束的标志是进程的状态是否为 Terminated) 具体代码如下:

```

void
Thread::Join(Thread* t)
{
    ASSERT(this!=t);
    Thread *nextThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    if(!t->isTerminated())
    {
        scheduler->AppendJoinThread(this);
        waitThread=t;
        while(!t->isTerminated()){
            while ((nextThread = scheduler->FindNextToRun()) == NULL)
                interrupt->Idle(); // no one to run, wait for an interrupt
            scheduler->Run(nextThread);
        }
    }
    //wait
    delete currentThread->waitThread;
    currentThread->waitThread=NULL;
    (void) interrupt->SetLevel(oldLevel);
}

```

(3) SC_Join 的实现

输入一个参数，就是要等待的进程号。从 readyList 中找到对应进程号的进程，然后调用 Join 函数。Join 结束之后，PC 继续向后走。

```

}
else if((which == SyscallException) && (type == SC_Join)){
    printf("Expected Join exception\n");
    int pid=machine->ReadRegister(4);
    int i=scheduler->readyList->GetSize();
    Thread* t;
    while(i!=0)
    {
        t=(Thread *)scheduler->readyList->Remove();
        scheduler->readyList->Append((void *)t);
        i--;
        if(t->GetPid()==pid)
        {
            currentThread->Join(t);
        }
    }
    printf("%d has been joined\n",pid);
    AdvancePC();
}

```

12. SC_Write () 的实现

Write 函数，实现 Nachos 系统向文件中写的功能。

Write 函数传入三个参数，第一个是写入内容的首地址，第二个要写入的内容大小，第三个是写入的文件 ID。将内容依次写到文件中，如果文件 ID 为 1，说明是将内容写到输出到控制台上。如果文件 ID 是别的，就写到对应的文件中。

```

else if((which == SyscallException) && (type == SC_Write)){
    printf("Expected Write exception\n");
    int base =machine->ReadRegister(4);
    int size=machine->ReadRegister(5);
    int fileId=machine->ReadRegister(6);
    int value;
    int count=0;

    OpenFile* openfile =new OpenFile (fileId);
    char* paramStr= new char[128];
    do{
        machine->ReadMem(base+count,1,&value);
        paramStr[count]=*(char*)&value;
        count++;
    }while(*(char*)&value!='\0'&&count<size);
    paramStr[count]='\0';
    if(fileId==1){
        printf("%s\n",paramStr);
    }
    else{
        if((openfile->Write(paramStr,size))==0)
            printf("write file fialed!");
    }
    delete openfile;
    AdvancePC();
}

```

13. SC_Read () 的实现

Read() 函数，实现 Nachos 系统从文件中读的功能。

Read 函数传入三个参数，第一个是读的 buffer 的首地址，第二个要读的内容大小，第三个是读的文件的文件 ID。如果文件 ID 是 0，则说明是从控制台上读取内容，如果是别的书，则说明是从别文件中读取内容，并将读取的内容放到 buffer 中。


```

    }
    else if((which == SyscallException) && (type == SC_Read)) {
        int base =machine->ReadRegister(4);
        int size = machine->ReadRegister(5);
        int fileId=machine->ReadRegister(6);

        OpenFile* openfile=new OpenFile(fileId);
        int readnum=0;
        if(fileId==0){
            char str[60];
            scanf("%s",str);
            int i;
            for(i=0;i<strlen(str);i++){
                machine->WriteMem(base+i,1,str[i]);
            }
            machine->WriteMem(base+i,1,'\0');
        }
        else{
            char temp[size];
            readnum=openfile->Read(temp,size);
            int i;
            for(i = 0;i<size;i++)
                if(!machine->WriteMem(base,1,temp[i]))
                    printf("This is something worry!");
            temp[readnum]='\0';
        }

        machine->WriteRegister(2,readnum);
        AdvancePC();
    }
}

```

四. 运行结果及分析

shell.cc 内容:

```

#include "syscall.h"

int
main()
{
    int newProc;
    OpenFileId input = ConsoleInput;
    OpenFileId output = ConsoleOutput;
    char prompt[2], ch, buffer[60];

    prompt[0] = '-';
    prompt[1] = '-';
    Write(prompt, 2, output);
    Read(&buffer, 60, input);
    newProc = Exec(buffer);
    Join(newProc);
    Halt();
}

```

mytest.cc 内容:

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

```
int main(){
    Exit(0);
}
```

```
daisy@daisy-VirtualBox:~/桌面/final/code/userprog$ make
make: 'arch/unknown-i386-linux/bin/nachos' is up to date.
daisy@daisy-VirtualBox:~/桌面/final/code/userprog$ ./nachos -x ../test/shell.
Pid is 100
Page table ( 11 pages in total):
=====
      Vitural      Physics
      0             0
      1             1
      2             2
      3             3
      4             4
      5             5
      6             6
      7             7
      8             8
      9             9
     10            10
=====
Expected Write exception
--
../test/mytest.noff
Expected Exec exception
Exec file is ../test/mytest.noff
Pid is 101
Page table ( 10 pages in total):
=====
      Vitural      Physics
      0             11
      1             12
      2             13
      3             14
      4             15
      5             16
```

```

5          16
6          17
7          18
8          19
9          20
=====
process 101 begin to run
Expected Exit exception
Expected Join exception
101 has been joined
Expected Halt exception
Machine halting!

Ticks: total 106, idle 0, system 40, user 66
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...

```

由输出内容可以看出，shell 运行，该进程 pid 为 100，遇到 Write 系统调用，在输入我要执行的文件路径之后，Read 系统调用，对该执行文件进行执行，并且分配相应的页表，该进程 pid 为 101。随后 101 号子进程运行结束，并被父进程 100 回收。

综上，测试实现了 Exec(), Exit(), Join(), Write(), Read(), Yield();

五. 收获与心得

通过这次实验,我了解到了 Nachos 可执行程序.noff 文件的格式组成,理解了 Nachos 应用程序的编程语法, 了解了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。掌握了如何利用交叉编译生成 Nachos 的可执行程序, 理解了系统如何为应用程序创建进程, 并启动进程的。理解了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和方法。理解了当前进程的页表是如何与 CPU 使用的页表进行关联的。理解了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; 理解了如何系统对空闲帧的管理; 理解了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; 理解了进程的 pid, 理解了进程退出所要完成的工作; 使 Nachos 能够支持多进程机制, 允许 Nachos 同时运行多个用户线程; 理解了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;

理解了系统调用是如何实现的; 理解了系统调用参数传递与返回数据的回传机制; 理解了核心进程如何调度执行应用程序进程; 理解了进程退出后如何释放内存等为其分

配的资源；理解了进程号 pid 的含义与使用；，测试实现了 Exec(),Exit(),Join(),Write(), Read(),Yield()等系统调用。

(五) 总结

通过这个学期的操作系统课设，我对操作系统的理解从理论层面升级到实践层面，在实验的过程中，我对对应知识的理解和掌握也更加透彻，接下来我就分别总结一下四个大实验带给我的收获。

第一个实验，Nachos 系统的安装与调试，我理解了 Nachos 系统的组织结构，熟悉 C++编程语言，掌握利用 Linux 调试工具 GDB 调试跟踪 Nachos 的执行过程，并通过 GDB 调试理解 Nachos 中线程上下文切换的过程，对 Makefile 的使用和结构也更加熟悉。

第二个实验，利用信号量实现线程同步，进一步加深并巩固了对于 Nachos 中线程的创建、切换等一些列功能和状态的理解，并实现了 Nachos 下的线程同步。同步问题是上学期操作系统学习中接触到的极其重要的问题之一，在本实验中依旧通过信号量的方式解决同步问题，模拟了较为简单和经典的生产者-消费者模型。

第三个实验，改写 Nachos 的文件系统，整个实验过程中需要对大量的代码进行阅读并理解，实验难度较大，进行时间也较长。最后对 nachos 的文件系统有了一个较为全面的理解，理解了一个文件系统如何通过空闲块管理、目录索引等多项机制来组织起当前系统内的文件。nachos 通过一个虚拟磁盘向我们展示了一个简单的文件系统，通过对文件系统的读写有了进一步的理解，首先仔细阅读代码发现当前文件系统为何无法进行扩展，之后通过编程实现文件的扩展，了解到文件扩展需要位图、目录、文件头等多方面的同步更新。在实验过程中遇到测试时出现“段转储错误”的问题，发现问题出现在写指针出现在扇区边界时接下来无法正确定位，会回到 0 号扇区，此时继续写操作会覆盖位图的头文件，导致-D 命令无法正确执行。修改方法已在报告中指出，需要添加关于边界处理的判断。这个实验使得自己对文件系统的理解进一步加深，了解了一个简单的文件系统如何通过各种数据结构来规划文件的读写组织等各项操作，收获颇丰。

第四个实验，Nachos 用户程序与系统调用，我了解到了 Nachos 可执行程序.noff 文件的格式组成，理解了 Nachos 应用程序的编程语法，了解了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。掌握了如何利用交叉编译生成 Nachos 的可执行程序，理解了系统如何为应用程序创建进程，并启动进程的。理解了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和方法。理解了当前进程的页表是如何与 CPU 使用的页表进行关联的。理解了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；理解了如何系统对空闲帧的管理；理解了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；理解了进程的 pid，理解了进程退出所要完成的工作；使 Nachos 能够支持多进程机制，允许 Nachos 同时运行多个用户线程；理解了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；理解了系统调用是如何实现的；理解了系统调用参数传递与返回数据的回传机制；理解了核心进程如何调度执行应用程序进程；理解了进程退出后如何释放内存等为其分配的资源；理解了进程号 pid 的含义与使用；，测试实现了 Exec(),Exit(),Join(),Write()，Read(),Yield() 等系统调用。

(六) 参考文献

- [1] 韩芳溪. OS 课程设计指南 (C++)
- [2] Abraham Silberschztz. 操作系统概念 (第七版) [M], 北京: 高等教育出版社, 2007. 3
- [3] Peiyi Tang, Ron Addie, nachos_introduction.pdf, University of Southern Queensland, 2002