操作系统课程设计

实验教程

张鸿烈编著

目录

教学指导型操作系统-Nachos 概述

计算机操作系统是当今计算机科学中最重要的基础课程之一，并且它是一门实践性极强的课程，需要学生亲自动手分析、编写和调试一个操作系统内核源代码才能真正掌握它的基本概念和原理。但是由于操作系统的复杂性，不可能让学生从 0 开始做起，也不可能让学生对一个庞大而又完整的真实操作系统进行二次开发。因此针对操作系统实践教学的要求出现了许多教学指导性操作系统，它们提供一个简单的但易于扩展的基本操作系统内核平台，使得学生可以在此基础上很容易的亲自动手分析、编写、调试和扩充操作系统内核的各种功能。Nachos 就是这样的一个系统。它由美国加州大学 Tom Adnerson 教授编写。是美国加州大学伯克莱分校在操作系统课程中已多次使用的操作系统课程设计平台，目前在国外许多著名大学中被采用为操作系统课程设计的实验教学平台，它在操作系统教学方面具有一下几个突出的特点：

Nachos 是建立在一个软件模拟的虚拟机之上的，模拟了 MIPS R2/3000 的指令集、主存、中断系统、网络以及磁盘系统等操作系统所必须的硬件系统。许多现代操作系统大多是先在用软件模拟的硬件上建立并调试，最后才在真正的硬件上运行。用软件模拟硬件的可靠性比真实硬件高得多，不会因为硬件故障而导致系统出错，便于调试。虚拟机可以在运行时报告详尽的出错信息，更重要的是采用虚拟机使 Nachos 的移植变得非常容易，在不同机器上移植 Nachos，只需对虚拟机部分作移植即可。

采用 R2/3000 指令集的原因是该指令集为 RISC 指令集，其指令数目比较少。Nachos 虚拟机模拟了其中的 63 条指令。由于 R2/3000 指令集是一个比较常用的指令集，许多现有的编译器如 g++能够直接将 C 或 C++源程序编译成该指令集的目标代码，于是就不必编写编译器，读者就可以直接用 C/C++语言编写应用程序，使得在 Nachos 上开发大型的应用程序也成为可能。

使用并实现了现代操作系统中的一些新的概念。Nachos 将这些新概念融入操作系统教学中，包括网络、线程和分布式应用。而且 Nachos 以线程作为一个基本概念讲述，取代了进程在以前操作系统教学中的地位。

Nachos 的虚拟机使得网络的实现相当简单。Nachos 只是一个在宿主机上运行的一个进程。在同一个宿主机上可以运行多个 Nachos 进程，各个进程可以相互通讯，作为一个全互连网络的一个节点；进程之间通过 Socket 进行通讯，模拟了一个全互连网络.

确定性调试比较方便；随机因素使系统运行更加真实因为操作系统的不确定性，所以在一个实际的系统中进行多线程调试是比较困难的。由于 Nachos 是在宿主机上运行的进程，它提供了确定性调试的手段。所谓确定性调试，就是在同样的输入顺序、输入参数的情况下，Nachos 运行的结果是完全一样的。在多线程调试中，可以将注意力集中在某一个实际问题上，而不受操作系统不确定性的干扰。当然，不确定性是操作系统所必须具有的特征，但 Nachos 可采用了随机因子模拟了真实操作系统的不确定性。

简单而易于扩展。Nachos 是一个教学用操作系统平台，它必须简单而且有一定的扩展余地。Nachos 不是向学生展示一个成功的操作系统，而是让学生在一个框架下发挥自己创造性的扩展。例如一个完整的类似于 UNIX 的文件系统是很复杂的，但是对于文件系统来说，无非是需要实现文件的逻辑地址到物理地址的映射以及实现文件 inode、打开文件结构、线程打开文件表等重要的数据结构以及维护它们之间的关系。基本 Nachos 中具有所有这些内容，但是在很多方面需要扩展，比如只有一级索引结构限制了系统中最大文件的大小。学生可以应用学到的各种知识对文件系统进行扩展，以实现自己的设计。

面向对象性 Nachos 的主体是用 C++或 JAVA 的一个子集来实现的。目前面向对象语言日渐流行，它能够清楚地描述操作系统各个部分的接口。Nachos 没有用到面向对象语言的所有特征，如继承性、多态性等，所以它的代码就更容易阅读和理解。

本课程设计实验教材结合 Nachos 系统分析和安排了操作系统中最基本的几个方面的设计和实验：

第一章讲述和分析了怎样利用 Nachos 系统进行内核线程管理的设计。包括线程的创建、切换、撤销、调度等原语的编程方法。安排了扩展线程调度、控制有关的实验内容。

第二章讲述和分析了怎样利用 Nachos 系统进行内核线程同步与互斥的设计。包括信号量、锁、管程等同步机制的编程方法。安排了扩展线程同步与互斥制机制有关的实验内容。

第三章讲述和分析了怎样利用 Nachos 系统进行内存管理设计。包括进程的装入、内存空间的页表分配、用户系统调用和虚拟内存等的编程方法。安排了扩充内存管理有关的实验内容。

第四章讲述和分析了怎样利用 Nachos 系统进行文件系统设计。包括 I/O 设备的驱动与同步访问控制、磁盘空闲空间的分配、i 节点的构造、打开文件结构、目录结构、逻辑文件系统的编程方法。安排了扩充文件系统有关的实验内容。

第 1 章 操作系统内核线程管理

操作系统设计的核心任务之一是进程和线程的管理，而现代操作系统进程和线程管理的首要任务是进程和线程的并发执行，其中线程又是并发执行最小的单位。本章通过对 Nachos 系统线程类的分析和实验来实现可并发执行的内核线程的设计和构造。

1.1 内核线程设计分析

操作系统中对处理器的管理即对系统中进程和线程的管理。简单的讲，进程是程序的执行.执行一个程序需要许多资源:CPU 周期、内存、文件、和 I／O 设备。但是关键是怎样使多个进程能共享单一的 CPU 以及其他的系统资源。

在一个程序编译和连接后，它的可执行的二进制文件通常是存储在外存文件中的。执行这个程序的进程是怎样启动的呢？当一个进程不能向前推进时系统又是怎样切换到另一个进程的呢？怎样使一个挂起的进程又重新继续执行呢？这些问题我们将通过使用 NACHOS 源代码，构造和跟踪其执行来了解和领会。

1.1.1 进程-Process

进程是执行中的程序。当程序执行时是什么样子呢？执行中的程序的地址空间表现在存储它的内存空间中并且它由以下部分组成：

- TEXT：从外存中装入的可执行文件的二进制代码。
- DATA：由全局的和静态变量组成的程序的数据区。它分为初始化的和未初始化的两部分。初始化的数据是从二进制可执行文件中装入的。
- STACK：函数的局部变量存储区，它由函数的调用而增长，由函数的返回而缩小。
- HEAP:动态分配变量存储区。例如 C 中的 malloc()或 C++中的 new。它随动态变量或结构的分配而增长，随变量或结构的回收而缩小。

除此之外，进程还有：寄存器集。最值得注意的寄存器有：

- PC：包含者可执行代码下一条指令地址的指令寄存器。
- SP：指向当前栈的栈指针寄存器。

一个 CPU 仅具有一组寄存器集合，它们可以由运行进程在任一特别的时间所占有。然而，当进程由运行态切换到另一状态(block 或 ready)，这些寄存器中的内容就必须保存到该进程的进程控制块中，否则进程就无法重新执行。保存和恢复这些寄存器的操作叫做上下文切换—Context Switch。

- 关联一个进程的标识信息，如进程标识号 PID 是 OS 中表示进程的唯一的整数。
- 打开或激活资源，如进程打开文件读 / 写，或连接一个套接口通过网络收 / 发数据。

1.1.2 线程-Thread

什么是线程?什么是线程和进程之间的不同?线程包含于进程中，线程实际上是一个抽象的并发程序执行顺序.属于同一进程的多个线程共享着进程的正文和数据部分、标识以及进程资源。但是每个线程具有各自的寄存器和栈空间。为什么我们需要把线程的栈和寄存器分开呢？因为栈和寄存器集决定了程序执行中动态上下文的内容。栈保存了函数调用的返回点和传递的参数，而寄存器组保存了当前指令执行后的结果、状态和下条要执行指令的地址。

现在我们就有了一个分级的程序执行的结构：一个系统中可以具有多个进程而且每个进程可以具有多个线程，它们共享着进程的代码、数据、堆、标识和资源。

但进程和线程共享许多相同的概念：

- 状态转换
- 控制块
- 上下文切换
- 调度

以下我们主要通过 NACHOS 来讨论线程的实现和控制。

1.1.3 线程控制块的定义和线程对象的建立

NACHOS 中的线程是由类 Thread 定义的。线程控制块是作为线程类中的一部分数据成员来说明的。

在文件 threads/thread.h 中的 77—113 行可以看到线程控制块数据成员的说明。

```
77  class Thread {
78    private:
```

```
79      // NOTE: DO NOT CHANGE the order of these first two members.
80      // THEY MUST be in this position for SWITCH to work.
81      int* stackTop;                          // the current stack
pointer
82      _int machineState[MachineStateSize];  // all registers
except for stackTop
83
84    public:
85      Thread(char* debugName);              // initialize a
Thread
86      ~Thread();                            // deallocate a
Thread
87                                            // NOTE -- thread
being deleted
88                                            // must not be running
when delete
89                                            // is called
90
91      // basic thread operations
92
93      void Fork(VoidFunctionPtr func, _int arg);  // Make thread
run (*func)(arg)
94      void Yield();                                // Relinquish
the CPU if any
95                                                   // other
thread is runnable
96      void Sleep();                                // Put the
thread to sleep and
97                                                   // relinquish
the processor
98      void Finish();                               // The thread
is done xecuting
99
100      void CheckOverflow();                       // Check if
thread has
101                                                  // overflowed
its stack
102      void setStatus(ThreadStatus st) { status = st; }
```

```
103      char* getName() { return (name); }
104      void Print() { printf("%s, ", name); }
105
106    private:
107      // some of the private data for this class is listed above
108
109      int* stack;                          // Bottom of the
stack
110                                           // NULL if this is
the main thread
111                                           // (If NULL, don't
deallocate stack)
112      ThreadStatus status;                 // ready, running or
blocked
113      char* name;
114
115      void StackAllocate(VoidFunctionPtr func, _int arg);
116                                           // Allocate a stack
for thread.
117                                           // Used internally
by Fork()
```

首先线程状态被存储在成员变量 status 中，它的类型是：ThreadStatus，该类型定义在同一文件的 61 行中。

```
60  // Thread state
61  enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED };
```

线程必须具有这些状态之一。当线程状态改变时 status 的值也对应的改变。线程具有自己的栈和寄存器组。在 NACHOS 中当状态从 JUST__CREATED 变为 READY 时则分配线程栈空间。类构造函数 Thread(见 Thread.cc)简单的建立线程名(为了调试的目的)并且设置线程态为 JUST__CREATED。指向整数的指针变量 stack 用于存储栈底(对栈溢出做检查)指向整数的指针变量 stackTop 是当前栈顶指针 SP。其他的寄存器包括 PC 都被存储在数组元素类型为宿主机机器字长的 machineState[MachineStateSize]数组中。该数组的的长度为 MachineStateSize，它定义在 Thread.h 中的 52 行。

```
49  // CPU register state to be saved on context switch.
```

```
50  // The SPARC and MIPS only need 10 registers, but the Snake needs
18.
51  // For simplicity, this is just the max over all architectures.
52  #define MachineStateSize 18
```

我们注意到一个用户进程控制块被从内核线程块中划分出去。在同一文件 thread.h 中可以看到 119—131 行上有 Thread 类附加的用户进程控制块数据成员

```
119  #ifdef USER_PROGRAM
120  // A thread running a user program actually has *two* sets of
CPU registers --
121  // one for its state while executing user code, one for its state
122  // while executing kernel code.
123
124      int userRegisters[NumTotalRegs];    // user-level CPU
register state
125
126    public:
127      void SaveUserState();               // save user-level
register state
128      void RestoreUserState();            // restore
user-level register state
129
130      AddrSpace *space;                   // User code this
thread is running.
131  #endif
132  };
133
134  // Magical machine-dependent routines, defined in switch.s
135
136  extern "C" {
137  // First frame on thread execution stack;
138  //      enable interrupts
139  //      call "func"
140  //      (when func returns, if ever) call ThreadFinish()
141  void ThreadRoot();
142
```

```
143  // Stop running oldThread and start running newThread
144  void SWITCH(Thread *oldThread, Thread *newThread);
145  }
```

其中在 124 行上说明的 userRegisters[NumTotalRegs]数组，用于保存用户寄存器的内容。可以看到 NACHOS 中的模拟机有两组寄存器组：

- 系统寄存器组用于运行内核线程。它与宿主机硬件寄存器有关。
- 用户寄存器组用于运行用户进程。它与宿主机硬件寄存器无关。

另外在 130 行上说明了一个指向 AddrSpace 类的指针 space。AddrSpace 类定义了用户地址空间，通过 space 我们可以访问用户地址空间。

每个 NACHOS 中的用户进程开始都是一个内核线程，通过内核线程构建用户进程的工作空间并装入用户程序后，该内核线程即转变为一个用户的进程。现在内核和用户进程的关系我们就有了一个基本的了解了。

在一个线程对象刚由 Thread()构造出来时，线程并没有完全具备可以运行的条件。即它仅生成了线程控制块，并没有为线程分配栈空间，也没有初始化寄存器组，即如果此时要调度它运行还不知道从哪儿启动。

1.1.4 线程状态转换控制原语

定义在 Thread 类中的 5 个线程控制原语可以在 threads/thread.h 文件的 84-98 行中看到。它们是函数：

```
84    public:
85      Thread(char* debugName);            // initialize a
Thread
86      ~Thread();                          // deallocate a
Thread
87                                          // NOTE -- thread
being deleted
88                                          // must not be running
when delete
89                                          // is called
90
91      // basic thread operations
92
93      void Fork(VoidFunctionPtr func, _int arg);  // Make thread
run (*func)(arg)
```

```
94      void Yield();                   // Relinquish
the CPU if any
95                                      // other
thread is runnable
96      void Sleep();                   // Put the
thread to sleep and
97                                      // relinquish
the processor
98      void Finish();                  // The thread
is done xecuting
```

- Thread()是线程对象的构造函数。它仅仅是建立对象的数据结构和将对象状态设置为 JUST_CREATED。
- Fork()用于产生从 JUST_CREATE 到 READY 的状态转换，并生成线程实例可运行的环境。
- Yield()用于当就绪队列非空时将线程状态从 RUNNING 转换为 READFY。它将当前进程（即调用 Yield 的线程）放入就绪队列尾部并且通过上下文切换将就绪队列中的一个线程变为运行状态。如果就绪队列为空，它没有任何作用并且继续运行当前线程。
- Sleep()用于将调用者线程从 RUNNING 转变为 BLOCKED,并从就绪队列中切换一个线程为运行。如果就绪队列为空，CPU 状态将变为空闲，直到有一个就绪线程要运行。Sleep 通常用于当线程开始 I/O 请求或要等待某个事件,它不能继续向前推进需要等待 I/O 完成或事件发生。在调用这个函数之前，线程通常将自己放入对应的 I/O 等待或事件有关的队列中。
- Finish()用于终止一个线程。

读一下函数文件 thread.cc 中的 Yiled 和 Sleep 的源代码可以理解 Nachos 中的线程状态是如何转换的。

栈的分配和线程控制块的初始化工作是由函数 Fork()完成的。这样做的原因是因为我们可以更加灵活的控制线程的并发执行。Fork 携带的第一个参数 func 是一个指向将要执行的线程函数的指针,第二个参数 arg 是线程函数所携带的参数。Arg 可以是一个和宿主机字长等长的整数(可从其类型说明为_int 看出)，如果线程函数要携带多个参数也可以是指向一组参数起始地址的指针。看一下 Fork 函数在 thread.cc 的 87-104 行上的源代码就可以了解 Fork 的主要工作过程。

```
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  }
```

可以看到 Fork 调用了函数 StackAllocate 为线程分配内存栈空间，并且初始化数组 machineState[]。然后由调度对象将自己推入就绪队列。

让我们来看一下说明在 thread.cc 中第 258-289 行上的 StackAllocate 函数（Nachos 支持多种硬件架构的机器，以下讨论涉及到硬件架构的地方均假设 Nachos 运行在 MIPS 机器上）。

```
246  // Thread::StackAllocate
247  //      Allocate and initialize an execution stack.  The stack
is
248  //      initialized with an initial stack frame for ThreadRoot,
which:
249  //              enables interrupts
250  //              calls (*func)(arg)
251  //              calls Thread::Finish
252  //
253  //      "func" is the procedure to be forked
254  //      "arg" is the parameter to be passed to the procedure
```

```
255
//----------------------------------------------------------------
--------
256
257  void
258  Thread::StackAllocate (VoidFunctionPtr func, _int arg)
259  {
260      stack = (int *) AllocBoundedArray(StackSize *
sizeof(_int));
261
262  #ifdef HOST_SNAKE
263      // HP stack works from low addresses to high addresses
264      stackTop = stack + 16;      // HP requires 64-byte frame
marker
265      stack[StackSize - 1] = STACK_FENCEPOST;
266  #else
267      // i386 & MIPS & SPARC & ALPHA stack works from high
addresses to low addresses
268  #ifdef HOST_SPARC
269      // SPARC stack must contains at least 1 activation record
to start with.
270      stackTop = stack + StackSize - 96;
271  #else  // HOST_MIPS  || HOST_i386 || HOST_ALPHA
272      stackTop = stack + StackSize - 4;    // -4 to be on the safe
side!
273  #ifdef HOST_i386
274      // the 80386 passes the return address on the stack.  In
order for
275      // SWITCH() to go to ThreadRoot when we switch to this
thread, the
276      // return addres used in SWITCH() must be the starting
address of
277      // ThreadRoot.
278      *(--stackTop) = (int)ThreadRoot;
279  #endif
280  #endif  // HOST_SPARC
281      *stack = STACK_FENCEPOST;
282  #endif  // HOST_SNAKE
```

```
283
284     machineState[PCState] = (_int) ThreadRoot;
285     machineState[StartupPCState] = (_int) InterruptEnable;
286     machineState[InitialPCState] = (_int) func;
287     machineState[InitialArgState] = arg;
288     machineState[WhenDonePCState] = (_int) ThreadFinish;
289 }
```

machineState 数组的 5 个下标宏，对于 MIPS 机在 switch.h 中定义为：

```
29  #define SP 0
30  #define S0 4
31  #define S1 8
32  #define S2 12
33  #define S3 16
34  #define S4 20
35  #define S5 24
36  #define S6 28
37  #define S7 32
38  #define FP 36
39  #define PC 40

51  #define InitialPC       s0
52  #define InitialArg      s1
53  #define WhenDonePC      s2
54  #define StartupPC       s3

56  #define PCState         (PC/4-1)  // s9
57  #define FPState         (FP/4-1)  // s8
58  #define InitialPCState  (S0/4-1)  // s0
59  #define InitialArgState (S1/4-1)  // s1
60  #define WhenDonePCState (S2/4-1)  // s2
61  #define StartupPCState  (S3/4-1)  // s3
```

对于 i386 系列在 switch.h 中定义为：

```
132  #define _ESP  0
133  #define _EAX    4
134  #define _EBX    8
135  #define _ECX    12
```

```
136  #define _EDX     16
137  #define _EBP     20
138  #define _ESI     24
139  #define _EDI     28
140  #define _PC      32
141
142  /* These definitions are used in Thread::AllocateStack(). */
143  #define PCState         (_PC/4-1)
144  #define FPState         (_EBP/4-1)
145  #define InitialPCState  (_ESI/4-1)  // %esi
146  #define InitialArgState (_EDX/4-1)  // %edx
147  #define WhenDonePCState (_EDI/4-1)  // %edi
148  #define StartupPCState  (_ECX/4-1)  // %ecx
149
150  #define InitialPC       %esi
151  #define InitialArg      %edx
152  #define WhenDonePC      %edi
153  #define StartupPC       %ecx
```

它们分别代表了机器寄存器存放在 machineState 数组中的偏量。

在 284 行上的 ThreadRoot 是一个定义在 switch.s 中的汇编语言函数，它是每个线程首次执行时调用的过程。285 行的 InterruptEnable 和 288 行的 ThreadFinish 是定义在 thread.cc 中的两个静态函数，InterruptEnable 用于打开中断，ThreadFinish 用于终止线程的执行。286 行上的 func 是线程执行函数入口地址，287 行上的 arg 是 func 所携带的参数，它俩都是由 Fork 函数的参数传递过来的。

当一个线程第一次被调度时，上下文切换例程总是将 machineState[PCState]中的值装入返回地址寄存器（对于 MIPS 机这个寄存器称为 ra 寄存器）。它恰恰是 ThreadRoot 的第一条可执行指令的地址。因此一个新线程执行的第一个例程总是 ThreadRoot。其代码如下：

```
69          .globl ThreadRoot
70          .ent    ThreadRoot,0
71  ThreadRoot:
72          or      fp,z,z          # Clearing the frame pointer
here
73                                  # makes gdb backtraces of
thread stacks
```

```
74                                      # end here (I hope!)
75
76          jal     StartupPC           # call startup procedure
77          move    a0, InitialArg
78          jal     InitialPC           # call main procedure
79          jal     WhenDonePC          # when we are done, call clean
up procedure
80
81          # NEVER REACHED
82          .end ThreadRoot
```

MIPS 计算机的指令：jal Reg
称作跳转连接(jump and link 类似于 i386 的 call)指令。它将当前下一条指令的地址存入 ra 寄存器并跳转到 Reg 寄存器所指向的程序入口执行。当该程序返回时，自动从保存在 ra 中的地址继续执行。

函数 ThreadRoot 实际上执行了三个顺序的子程序：StartupPC，InitialPC，WhenDonePC
它们分别对应 InterruptEnable func，ThreadFinish 三个函数。可以看出 ThreadRoot 包含了一个线程从启动到终止的整个生命期的活动。

1.1.5 线程和进程调度

一个线程或进程在他们的生命期间将通过许多次状态切换。在所有这些状态中就绪队列用于放置所有就绪状态的线程或进程。其他队列对应的放置在因为申请不同 I/O 设备而处于阻塞状态的进程或线程，它们等待响应 I/O 请求的完成。线程或进程由作业调度者在队列中按调度策略移动。

在 Nachos 中，线程调度是由定义在 Threads/scheduler.h 和 scheduler.cc 的 Scheduler 类的一个全局对象来完成的。这个类的方法提供了线程和进程的所有调度功能。当 Nachos 首次启动时，首先建立一个 Scheduler 类的全局实例对象的引用 scheduler，由它负责完成线程或进程的调度任务。这个类的定义见 scheduler.h 文件的 20-34 行。

```
16  // The following class defines the scheduler/dispatcher
abstraction --
17  // the data structures and operations needed to keep track of
which
```

```
18  // thread is running, and which threads are ready but not
running.
19
20  class Scheduler {
21    public:
22      Scheduler();                          // Initialize list of
ready threads
23      ~Scheduler();                         // De-allocate ready
list
24
25      void ReadyToRun(Thread* thread);     // Thread can be
dispatched.
26      Thread* FindNextToRun();              // Dequeue first
thread on the ready
27                                            // list, if any, and
return thread.
28      void Run(Thread* nextThread);        // Cause nextThread to
start running
29      void Print();                         // Print contents of
ready list
30
31    private:
32      List *readyList;              // queue of threads that are
ready to run,
33                                    // but not running
34  };
```

Schedulerl 类仅有一个私有对象它就是指向 list 对象的一个指针（见 list.h 和 list.cc）。这个 list 对象是放置 READY 状态的就绪队列。函数 ReadyToRun(Thread *thread)将一个线程推入该队列尾，同时函数 FindNextToRun()从队列返回出队线程的指针（或 NULL,如果队列为空）。可能在 Nachos 中最有趣的函数是该类中的 Run(Thread*)。这个函数调用汇编语言函数 SWITCH(Thread*,Thread*)将当前线程切换到由第二参数指向的另一线程。Scheduler 类的这三个函数用于线程对象的状态转换。图 1.1 说明了 thread,scheduler,list 三个对象的关系。

图中 ClassA 到 ClassB 的箭头代表 ClassA 的函数将调用 ClassB 对象。在我们这种情况中 scheduler 对象包括了 list 对象。读一下 Scheduler 类对应的源代码，我们会明白 scheduler 是怎样工作的。

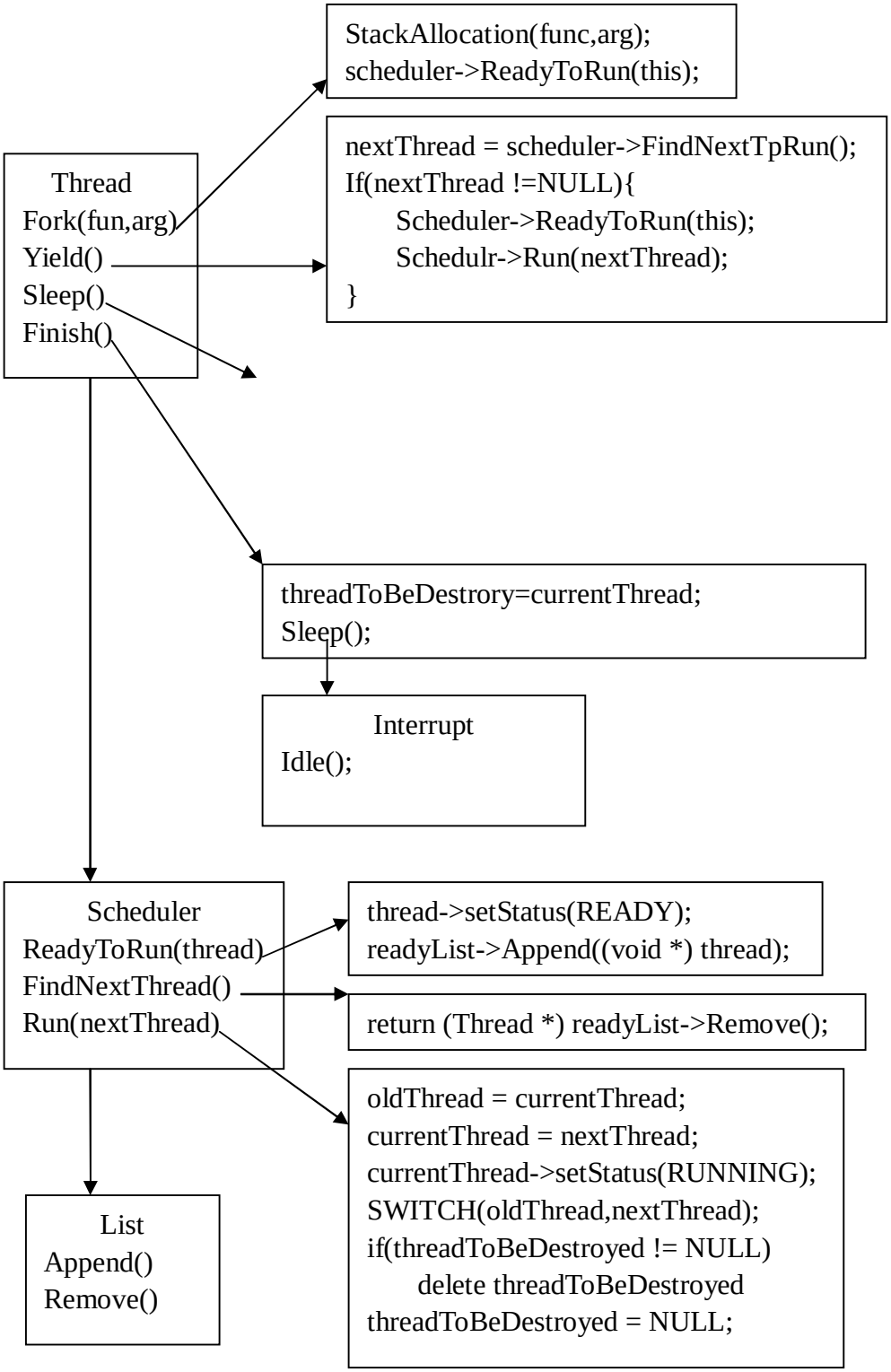

图 1.1 Nachos 中线程类、调度类、队列类之间的关系图

1.1.6 线程/进程的上下文切换

实际的操作系统中上下文切换是怎样完成的呢？请看 Nachos 中的实现过程。在 Nachos 中上下文切换是由调用 Scheduler 类的 Run(Thread *) 函数开始的。在 scheduler.cc 文件中的 90-135 行上，Run 源代码如下：

```
90  void
91  Scheduler::Run (Thread *nextThread)
92  {
93      Thread *oldThread = currentThread;
94
95  #ifdef USER_PROGRAM                    // ignore until
running user programs
96      if (currentThread->space != NULL) { // if this thread is
a user program,
97          currentThread->SaveUserState(); // save the user's CPU
registers
98          currentThread->space->SaveState();
99      }
100  #endif
101
102      oldThread->CheckOverflow();            // check if the
old thread
103                                             // had an
undetected stack overflow
104
105      currentThread = nextThread;            // switch to the
next thread
106      currentThread->setStatus(RUNNING);     // nextThread is
now running
107
108      DEBUG('t', "Switching from thread \"%s\" to thread
\"%s\"\n",
109            oldThread->getName(), nextThread->getName());
110
```

```
111     // This is a machine-dependent assembly language routine
defined
112     // in switch.s.  You may have to think
113     // a bit to figure out what happens after this, both from
the point
114     // of view of the thread and from the perspective of the
"outside world".
115
116     SWITCH(oldThread, nextThread);
117
118     DEBUG('t', "Now in thread \"%s\"\n",
currentThread->getName());
119
120     // If the old thread gave up the processor because it was
finishing,
121     // we need to delete its carcass.  Note we cannot delete
the thread
122     // before now (for example, in Thread::Finish()), because
up to this
123     // point, we were still running on the old thread's stack!
124     if (threadToBeDestroyed != NULL) {
125         delete threadToBeDestroyed;
126         threadToBeDestroyed = NULL;
127     }
128
129 #ifdef USER_PROGRAM
130     if (currentThread->space != NULL) {         // if there
is an address space
131         currentThread->RestoreUserState();     // to restore,
do it.
132         currentThread->space->RestoreState();
133     }
134 #endif
135 }
136
```

在讨论这个函数的细节之前，我们需要说明 Nachos 内核中一组重要的全局变量。在 threads/system.cc 文件中定义了 6 个全局变量，它们是：

```
14  Thread *currentThread;                  // the thread we are
running now
15  Thread *threadToBeDestroyed;            // the thread that
just finished
16  Scheduler *scheduler;                   // the ready list
17  Interrupt *interrupt;                   // interrupt status
18  Statistics *stats;                      // performance
metrics
19  Timer *timer;                           // the hardware timer
device,
```

全局变量 currentThread 是指向当前运行中的线程的指针。全局变量 scheduler 是指向负责调度、分派就绪线程的调度对象的指针。当 Nachos 内核启动时将建立这个调度对象及其他一些对象。建立和初始化这些全局对象的例程是 system.cc 文件中的函数 Initialize(int argc,char **argv)。其他的全局对象我们在讨论到用户进程和文件系统时就会清楚了。

函数 Scheduler::Run(Thread *nextThread)首先将 currentThread 保存到变量 oldThread 中并将 currentThread 指向 nextThread 所指向的线程对象。然后在 116 行上调用汇编函数 SWITCH(oldThread, nextThread)。注意 111-114 行上的注释，搞清在 SWITCH 调用期间发生了什么是领会上下文切换概念的基础。

汇编函数 SWITCH 在文件 threads/switch.h 和 switch.cc 中说明，这些文件包括了许多主机如：,MIPS , Sparc, i386, alpha, …。正如我们前面约定的，我们现在的讨论假设 Nachos 运行在 MIPS 硬件体系上。这样我们就先看一下与 MIPS 机有关的汇编代码。在 switch.h 的 20-39 行上定义了与 MIPS 机寄存器有关的常量：

```
25  /* Registers that must be saved during a context switch.
26   * These are the offsets from the beginning of the Thread object,
27   * in bytes, used in switch.s
28   */
29  #define SP 0
30  #define S0 4
31  #define S1 8
32  #define S2 12
33  #define S3 16
34  #define S4 20
```

```
35  #define S5 24
36  #define S6 28
37  #define S7 32
38  #define FP 36
39  #define PC 40
```

在文件 switch.s 中的 51-64 行上定义了一组 MIPS 机寄存器名：

```
51  #define z       $0      /* zero register */
52  #define a0      $4      /* argument registers */
53  #define a1      $5
54  #define s0      $16     /* callee saved */
55  #define s1      $17
56  #define s2      $18
57  #define s3      $19
58  #define s4      $20
59  #define s5      $21
60  #define s6      $22
61  #define s7      $23
62  #define sp      $29     /* stack pointer */
63  #define fp      $30     /* frame pointer */
64  #define ra      $31     /* return address */
```

汇编函数 SWITCH(…)的定义在文件 switch.s 的 86-113 行上：

```
84          # a0 -- pointer to old Thread
85          # a1 -- pointer to new Thread
86          .globl SWITCH
87          .ent    SWITCH,0
88  SWITCH:
89          sw      sp, SP(a0)              # save new stack
pointer
90          sw      s0, S0(a0)              # save all the
callee-save registers
91          sw      s1, S1(a0)
92          sw      s2, S2(a0)
93          sw      s3, S3(a0)
94          sw      s4, S4(a0)
95          sw      s5, S5(a0)
```

```
96          sw      s6, S6(a0)
97          sw      s7, S7(a0)
98          sw      fp, FP(a0)              # save frame pointer
99          sw      ra, PC(a0)              # save return address
100
101          lw      sp, SP(a1)              # load the new stack
pointer
102          lw      s0, S0(a1)              # load the
callee-save registers
103          lw      s1, S1(a1)
104          lw      s2, S2(a1)
105          lw      s3, S3(a1)
106          lw      s4, S4(a1)
107          lw      s5, S5(a1)
108          lw      s6, S6(a1)
109          lw      s7, S7(a1)
110          lw      fp, FP(a1)
111          lw      ra, PC(a1)              # load the return
address
112
113          j       ra
114          .end SWITCH
```

这里的 a0 和 a1 就是 MIPS 机中的$4 和$5 寄存器的宏定义。$4 和$5 寄存器分别用于存放函数的第一参数和第二参数。对于函数 SWITCH(oldThread,nextThread)，a0 就是第一参数 oldTread,a1 就是第二参数 nextThread。

MIPS 机汇编指令：sw　　Rsrc,const(Rindex)

将 Rsrc 寄存器中的字存入由 Rindex 寄存器的内容加上一个 const 常数所确定的内存单元中去。

MIPS 机汇编指令：lw　　Rsrc,const(Rindex)

从 Rindex 寄存器的内容加上一个 const 常数所确定的内存单元中取出一个字存入 Rsrc 寄存器中。

SWITCH 首先保存所有重要的寄存器的值到当前线程的线程控制块中。即第一个私有的线程类成员 stackTop 及其后的 machineState[machineStatesize]数组中。换句话说，引用一个 Thread 对象实际上就是对该对象的 stackTop 的引用。

注意常数变量 sp，s0，s1,…具有固定值 0，1，2…，即 stackTop 和 machineState[]数组元素的摆列位置要和对应的寄存器的编排位置严格对应。

在保存的所有这些寄存器中，ra 寄存器用于存放调用函数的返回地址。对于当前的 ra，它包含了调用 SWITCH 后函数应返回的正确的地址。

放弃了 CPU 的当前线程将会由其他上下文切换事件再次获得 CPU。当它被切换回来时，所有保存在 stackTop 和 machineState[]数组中的内容都将恢复到对应的寄存器中，包括返回地址寄存器 ra，第 113 行上的指令使得控制跳转到 ra 所保存的地址上，则当前线程又重新获得执行。

也应注意行 95-100 和 129-134 行上的代码。这些代码用于保存和恢复用户进程的用户寄存器和用户地址空间。整个 Run 函数运行于内核，因为它属于 Nachos 内核进程。

注意调用 Run 函数的当前线程它不会立即返回，实际上它将不会自动返回，直到有系统调度事件发生并被选中后才可能被切换回来再次成为当前线程继续运行。

1.1.7 线程的终止

我们已经在 ThreadRoot 中看到了线程的终止函数 WhenDonePC，它将使控制进入定义在 thread.cc 第 241 行上的静态函数 ThreadFinish：

```
241  static void ThreadFinish()     { currentThread->Finish(); }
```

ThreadFinish 简单的调用当前进程的中止函数 Finish。由于线程不应自身析构，所以 Finish 仅是设置全局变量 ThreadToBeDestroyed 为当前进程，并调用 Sleep 函数将自身状态置为阻塞。该线程的真正终止实际上是由下一次线程上下文切换时完成的。再次注意一下函数 Run 在调用了 SWITCH 之后的代码：

```
120      // If the old thread gave up the processor because it was
finishing,
121      // we need to delete its carcass.  Note we cannot delete
the thread
122      // before now (for example, in Thread::Finish()), because
up to this
123      // point, we were still running on the old thread's stack!
124      if (threadToBeDestroyed != NULL) {
125          delete threadToBeDestroyed;
126          threadToBeDestroyed = NULL;
```

```
127         }
```

可以看出，无论哪个线程它在上下文切换之后重新执行时都将去检测是否有将要终止的线程，如果有则将其删除。删除运算将析构该线程对象，回收为该线程分配的内存栈空间。

1.1.8 Nachos 基本内核的工作过程

我们以已经概括了 Nachos 系统管理线程及进程的情况。看到了线程在 Nachos 中是怎样建立、运行、终止的。我们也分析了线程的上下文切换是怎样在 Nachos 中完成的，以及 Nachos 系统的调度对象是怎样管理多个线程并发执行的。

现在我们看一下 Nachos 内核本身。如其他操作系统一样，Nachos 内核也是操作系统的一部分。最小的 Nachos 内核仅包含 Nachos 线程管理，可以在 threads 目录中编译生成。Nachos 内核 main()函数是内核程序的启动入口。可以在 threads/main.cc 中看到。

main() 的主要过程为：

```
Initialize(argc,argv);
ThreadTest();
currentThread->Finish();
```

函数 Initialize 定义在 threads/system.cc 文件中。它建立并初始化 Nachos 中的全局对象，如调度对象、定时器对象、第一个线程对象"main"等。

函数 threadTest 定义在 threads/threadtest.cc 文件中，它是一个内核测试函数。

```
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  }
```

可以看出这个函数生成了一个叫"forked thread"的线程对象，并通过指向该对象的指针引用该对象的 Fork 函数，启动了一个新的线程函数 SimpleThread，新的线程函数 SimpleThread 带有参数 1，它将与下一行上的属于"main"线程对象的另一带有参数 0 的 SimpleThread 函数并发执行。

函数 SimpleThread 也定义在 ThreadTest.cc 中。它简单的打印出当前线程函数的参数后即调用线程的 Yield 函数放弃 CPU，如此循环 5 次。

```
24  void
25  SimpleThread(_int which)
26  {
27      int num;
28
29      for (num = 0; num < 5; num++) {
30          printf("*** thread %d looped %d times\n", (int) which,
num);
31          currentThread->Yield();
32      }
33  }
```

在函数 TreadTest 中我们启动了两个 SimpleThread 线程，因此这两线程将反复切换直到它们终止。

在 Unix/Linux 系统中我们可以将 Nachos 内核作为 Unix/Linux 下的一个进程启动。

```
$ ./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...

$

以上输出显示了 Nachos 内核中两并发线程的执行情况，它们轮流交替的打印各自的信息。

在 main 函数中最后调用的函数是 currentThread->Finish()。这个函数决不会返回，因为这个线程将在上下文切换之后被下一线程删除。当所有线程都终止之后，Nachos 内核将从 Unix/Linux 系统中退出。

1.2 内核线程控制实验

通过上节的分析我们已经了解了基本 Nachos 系统线程类的构造原理和设计的方法。本节我们将通过实验来体验操作系统内核线程的设计和开发，扩充基本 Nachos 系统线程类的构造和功能，实现自主设计的操作系统内核线程管理。

1.2.1 实验目的

- 练习安装、编译和测试基本 Nachos 系统
- 了解基本 Nachos 系统组织结构
- 熟悉 C++语言和系统开发调试工具
- 理解 Nachos 系统的 Makefile
- 掌握重构 Nachos 系统的方法

1.2.2 安装 Nachos 和 MIPS gcc 交叉编译

进行 Nachos 操作系统实验需要两个软件包：

- Nachos-3.4.tgz 它是 Nachos 系统开发软件包
- Gcc-2.8.1-mips.tar.gz 它是 gcc 的 MIPS 交叉编译软件包

Nachos 系统开发软件包的安装

进入您的工作目录,将文件 nachos-3.4.tgz 复制到当前目录中：

- 使用 shell 命令安装：$tar xzvf nachos-3.4.tgz
- 使用图形窗体安装：在文件窗体中选中 nachos-3.4.tgz 点击解压

安装成功后在当前工作目录中都会生成目录 nachos-3.4，该目录中应生成以下图 1.2 所示的目录结构：

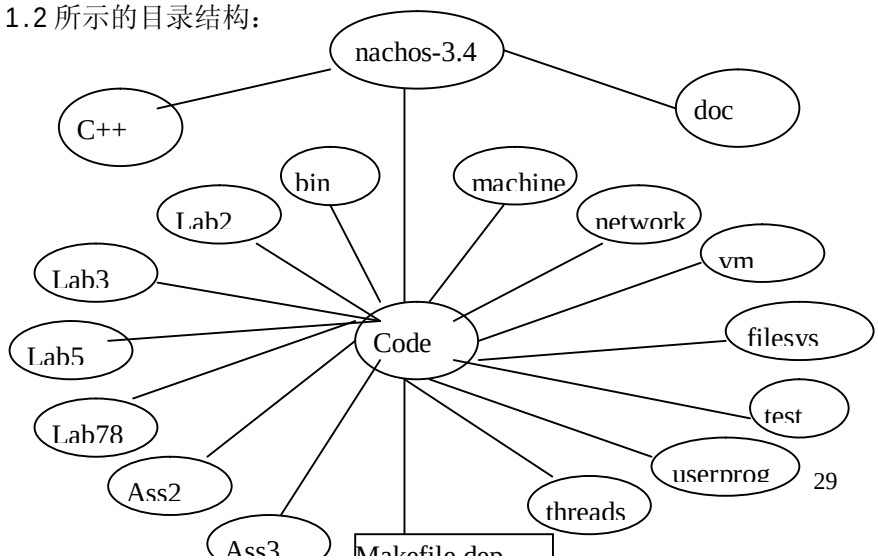

图 1.2 Nachos 系统的基本结构

gcc 的 MIPS 交叉编译软件包的安装：

以 root 身份进入系统目录/usr/local，将文件 Gcc-2.8.1-mips.tar.gz 复制到/usr/local 目录中：

- 使用 shell 命令安装：$tar xzvf Gcc-2.8.1-mips.tar.gz
- 使用图形窗体安装：在文件窗体中选中 Gcc-2.8.1-mips.tar.gz 点击解压

安装成功后在/usr/local 目录中都会生成目录 Gcc-2.8.1-mips

1.2.3 编译和测试基本 Nachos 系统

在 code 目录下的 lab2/,lab3,lab5/,lab78/和 ass2/,ass3/子目录是我们开发实验用的工作目录。

machine/,threads/,userprog/,filesys/,network/,vm/,test/子目录存放基本 Nachos 内核源代码。

为了测试 Nachos 系统最基本的功能是否正确，应首先进入 threads/目录编译生成 Nachos 基本内核。输入命令：

```
$ make
   ....>>> Linking arch/unknown-i386-linux/bin/nachos <<<
   g++ arch/unknown-i386-linux/objects/main.o .....................
   .....................................................................
   .....................................................................
   ln unknown-i386-linux/bin/nachos nachos
$
```

如果你成功的完成了以上编译工作，你已经有了一个最小的 Nachos 操作系统内核。你应看到在当前动作目录中生成一个链接到 Nachos 内核可执行文件 arch/unknown-i386-linux/bin/nachos 的连接文件 nachos。现在你可以测试 Nachos 系统的工作情况。输入：

```
$ ./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...
$
```

如果看到以上输出信息，说明最小的 Nachos 内核工作正常，可以在此基础上展开您的 Nachos 操作系统实验了。

1.2.4 C++程序设计语言和 gdb 调试程序简介

Nachos3.4 版源代码是用 C++语言编写的。但是进行 Nachos 实验并不要求预先的 C++语言知识。当然如果你以前已学过面向对象的程序设计课程则将对你会很有帮助的。Nachos 源代码的编写仅使用了 C++中有关抽象数据类型和封装的功能，没有使用 C++中继承、重载等晦涩难懂的功能。为了方便 Nachos 的实验的开展 nachos-3.4 软件包中附带了一份由 Dr.Tom Anderson 编写的 C++语言快速简介文档”quick introduction of C++”并附有该文档中介绍的例题程序。它们都放在 c++exampl 目录中。如果你对 C++语言不是很熟悉可先阅读一下该文档及其附带例题程序。

在开发实验 Nachos 系统时可能需要使用 GNU 的 gdb 调试程序对我们开发的程序进行跟踪调试。如果你对 gdb 的使用不太熟悉的话，可以先利用 c++exampl 目录中的 stack.cc 作一下练习。练习步骤如下：

进入 c++exampl 目录中输入 make 命令编译并生成可执行程序 stack

利用编辑程序 emacs 启动 gdb
先输入 emacs & 在后台打开 emacs 视窗
在 emacs 视窗的中输入 Alt-x 键并输入：gdb ./stack 之后，emacs 将启动 gdb 并装入程序 stack, 出现以下提示：

```
Current directory is /home/staff/ptang/units/204/98/linux/nachos3.4/c++example/
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386redhatlinux), Copyright 1996 Free Software Foundation, Inc...
(gdb)
```

现在你就可以输入调试命令跟踪 stack 的执行了。首先列出 stack.cc 源代码前 10 行。输入命令：

```
(gdb)list
120 }
121
122 //
123 // main
124 // Run the test code for the stack implementation.
125 //
126
127 int
128 main() {
129 Stack *stack = new Stack(10); // Constructor with an argument.
(gdb)
```

在 129 行上设立断点

```
(gdb) break 129

Breakpoint 1 at 0x80488b6: file stack.cc, line 129.
(gdb)
```

输入程序执行命令。

```
(gdb) run
Starting program: nachos3.4/c++example/stack
```

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

当程序执行到断点时，执行暂停，emace 将其视窗分成两部分，上方视窗仍保留为命令输入和信息提示窗，如上所示；下方视窗显示程序断点处上下 10 行程序源代码，并有一箭头指示当前断点行，如下所示：

```
}
//
// main
// Run the test code for the stack implementation.
//
int
main() {
=> Stack *stack = new Stack(10); // Constructor with an argument.
stack>SelfTest();
delete stack; // always delete what you allocate
return 0;
}
```

此时你可以在命令提示窗继续输入调试命令。如逐行跟踪执行命令：

```
(gdb) run
Starting program: nachos3.4/c++example/stack
Breakpoint 1, main () at stack.cc:129
(gdb) next
--------------------------------------------------------------------------
int
main() {
Stack *stack = new Stack(10); // Constructor with an argument.
=> stack>SelfTest();
delete stack; // always delete what you allocate
return 0;
```

或输入进入函数逐行跟踪命令：

```
(gdb) run
Starting program: nachos3.4/c++example/stack
Breakpoint 1, main () at stack.cc:129
(gdb) next
(gdb) step
--------------------------------------------------------------------------
```

```
void
Stack::SelfTest() {
=> int count = 17;
// Put a bunch of stuff in the stack...
while (!Full()) {
cout << "pushing " << count << "\n";
Push(count++);
}
```

你也可以输入打印命令，观察程序执行中变量的值：

```
(gdb) run
Starting program: nachos3.4/c++example/stack
Breakpoint 1, main () at stack.cc:129
(gdb) next
(gdb) step
(gdb) print count
```

有关 gdb 的调试命令的细节请通过 man gdb 查阅。

清除以前的编译结果，重新编译
可以通过输入命令：

```
$make clean
```

这将删除掉所有编译的中间结果文件，当再次执行 make 命令时编译重新开始。

1.2.5 make 命令与 Makefiles 结构

make 是一种控制编译或重复编译软件的工具软件，make 可以自动管理软件的编译内容、
编译方式和编译时机。使用 make 需要你为你所编写的软件的开发过程和组织结构编写一个 Makefile 文件。make 将根据 MAkefile 中的说明去自动管理你的软件的开发过程。Makefile 是一个文本形式的数据库文件。可应包含以下目标软件的生成规则：

- 目标体（target），即 make 要建立的目标文件。
- 目标的依赖体（dependency）列表，通常为要编译的源文件或要连接的浮动目标代码文件。

- 从目标依赖体创建目标体的命令（command）列表，通常为编译或连接命令。

以上叙述在 Makefile 中用以下规则形式表示

```
target：        dependency […]
                command1
                command2
                […]
```

例如我们编写了一个 C 程序存放在 hello.c 和一个 hello.h 文件中，为了使用 make 自动管理这个 C 程序的开发，可以编写以下 Makefile 文件：

```
hell.o:  hello.c hello.h
         gcc  -c hello.c hello.h
hello:   hello.o
         gcc hello.o  -o hello
clean:
         rm  -f *.o
```

这样我们就可以使用 make 按我们说明在 Makefile 中的编译规则编译我们的程序了：

```
$make                          生成可执行文件 hello
$make hello.o                  生成浮动模块文件 hello.o
$make clean                    清除所有.o 文件
```

make 怎样知道什么时候需要重新编译或无需重新编译或编译部分文件呢？答案是：

- 如果指定的目标体 make 找不到，make 就根据该目标体在 Makefile 中说明的生成规则建立它。
- 如果目标体存在，make 就对目标体和依赖体的时间戳进行比较，若有一个或多个依赖体比目标体新，make 就根据生成命令重新生成目标体。这意味着每个依赖体的改动都将使目标体重新生成。

在 Makefile 中可以说明伪目标。上例的 clean 目标体就是一个伪目标。即伪目标物依赖体，它仅指定了要执行的命令。

make 中的宏变量

在 Makefile 中可以定义宏变量。变量的定义格式为：变量名=字符串 1 字符串 2 ….

变量的引用格式为：$(变量名)

如上例可改写为：

```
obj=hello.o
hello:   $(obj)
         gcc $(obj) –o hello
```

make 中的自动变量

make 中提供了一组元字符用来表示自动变量，自动变量用来匹配某种规则，它们有：

$@	规则的目标体所对应的文件名
$<	规则中第一个相关文件名
$^	规则中所有相关文件名的列表
$?	规则中所有日期新于目标文件名的列表
$(@D)	目标文件的目录部分
$(@F)	目标文件的文件名部分

make 中的预定义变量

▌	AR	归档维护程序，默认值=ar
▌	AS	汇编程序，默认值=as
▌	CC	C 编译程序，默认值=gcc
▌	CPP	C++编译程序，默认值=cpp
▌	RM	删除程序，默认值=rm -f
▌	ARFIAGS	归档选项开关，默认值=rv
▌	ASFLAGE	汇编选项开关
▌	CFLAGS	C 编译选项开关
▌	CPPFLAGS	C++编译选项开关
▌	LDFILAGS	链接选项开关

make 中隐式规则（静态规则）

编译过程中一些固定的规则可以省略说明，称为隐式规则。如上例中目标体 hello.o 的规则隐含在目标体 hello 的规则中，就属于隐式规则，可以省略为：

```
obj=hello.c
hello:   $(obj)
         gcc $(obj) –o hello
```

make 中的模式规则

```
%        用于匹配目标体和依赖体中任意非空字符串
例如：%.o:       %.c
                 $(cc) –c  $^ -o $@
```

以上的模式规则表示，用 g++编译器编译依赖体中所有的.c 文件，生成.o 浮动目标模块，目标文件名采用目标体文件名。

前缀#符号的行为注释行。

1.2.6 Nachos 系统的 Makefiles 结构

在 code 目录中有两个为其子目录所公用的 Makefile 文件：
Makefile.common
Makefile.local
在 code/的每个子目录中各自都有两个 Makefile 文件：
Makefile
Makefile.local

即 Nachos 系统的 Makefile 结构为：

```
../code/Makefile.common,Makefile.dep
    |____threads/Makefile,Makefile.local
    |____userprog/ Makefile,Makefile.local
    .
    .
    .
    |____filesys/Makefile,Makefile.local
```

code/下的每个子目录中的 Makefile 都括入

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

1.2.7 Makefile.local 文件的说明

Makefile.local 每个子目录中都不同，主要用于说明本目录中文件特有的依赖关系。其中预定义变量的值为：

- CCFILE　　构造本目录中 Nachos 系统所用到的 C++源文件的文件名串

- INCPATH　　指示 g++编译器查找 C++源程序中括入的.h 文件的路径名串
- DEFINES　　传递给 g++编译器的标号串

例如threads/目录下的Makefile.local的定义为：

```
CCFILES = main.cc\
    list.cc\
    scheduler.cc\
    synch.cc\
    synchlist.cc\
    system.cc\
    thread.cc\
    utility.cc\
    threadtest.cc\
    synchtest.cc\
    interrupt.cc\
    sysdep.cc\
    stats.cc\
    timer.cc
INCPATH += -I../threads -I../machine
DEFINES += -DTHREADS
```

1.2.8 Makefile.dep 文件的说明

在 code/目录中的 Makefile.dep 文件用于定义由 g++使用的系统依赖关系的宏。它被括入在 code/Makefile.common 文件中。当前发行的 Nachos 可以在 4 种不同的 unix/linux 系统中编译并生成可执行的二进制文件 nachos。可执行文件统一放在 arch 目录的特定目录下。例如在 i386 的 linux 系统中可执行的nachos 程序应放在arch/unknown-i386-linux/bin/目录中。

这些在 Makefile.dep 定义的依赖系统的宏有：

- HOST　　主机系统架构
- arch　　文档存放路径
- CPP　　C++编译器的名字
- CPPFLAGS　　C++编译开关
- GCCDIR　　g++安装路径
- LDFLAGS　　程序链接开关

- ASFLAGS 汇编开关

例如：当前系统为 i386 架构，linux 操作系统，则以上的宏定义为：

```
HOST_LINUX=-linux
HOST = -DHOST_i386 -DHOST_LINUX
CPP=/lib/cpp
CPPFLAGS = $(INCDIR) -D HOST_i386 -D HOST_LINUX
arch = unknown-i386-linux
```

在这个文件中还定义了其他一些依赖系统的宏，它们是：

```
arch_dir = arch/$(arch)                 归档文件目录
obj_dir = $(arch_dir)/objects           存放目标文件的目录
bin_dir = $(arch_dir)/bin               存放可执行文件的目录
depends_dir = $(arch_dir)/depends       存放依赖关系文件的目录
```

例如在i386/linux系统中最后3个目录为：

```
arch/unknown-i386-linux/objects
arch/unknown-i386-linux/bin
arch/unknown-i386-linux/depends
```

1.2.9 Makefile.common 文件的说明

code/目录中的 Makefile.common 首先括入 Makefile.dep，然后用 vpath 定义各类文件搜索路径。

```
include ../Makefile.dep

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

vpath 定义告诉 make 到哪儿去查找在当前目录中找不到的文件。这就是为什么我们在一个新的目录中构造一个新的 Nachos 系统时不必复制那些我们不作修改的文件的原因。

这个文件中还定义了编译开关宏 CFLAGS、目标文件规则宏 ofile ，和最终目标程序规则宏 program。

```
CFLAGS = -g -Wall -Wshadow -fwritable-strings $(INCPATH) $(DEFINES)
$(HOST) -DCHANGED

s_ofiles = $(SFILES:%.s=$(obj_dir)/%.o)
c_ofiles = $(CFILES:%.c=$(obj_dir)/%.o)
cc_ofiles = $(CCFILES:%.cc=$(obj_dir)/%.o)

ofiles = $(cc_ofiles) $(c_ofiles) $(s_ofiles)

program = $(bin_dir)/nachos

$(program): $(ofiles)
```

这些规则说明了我们将根据所有的.o 文件在目录 unknown-i386-linux/bin 中（对于 linux 系统）构造二进制可执行文件 nachos。使用以上规则构造这个目标的定义为：

```
$(bin_dir)/% :
    @echo ">>> Linking" $@ "<<<"
    $(LD) $^ $(LDFLAGS) -o $@
    ln -sf $@ $(notdir $@)
```

以上定义的规则隐式的联合了$(program): $(ofiles)规则。目标体的%号将通配其他目标体中任意非空字符串规则。

第一条命令使用g++（这里用$(LD)代表）链接所有的.o文件（这里用$^代表），生成可执行目标文件nachos（这里用$@代表）。

第二条命令将在当前目录中产生一个与可执行文件nachos同名的链接文件。即命令：

```
    ln -sf $@ $(notdir $@)
```

实际扩展为 shell 命令：

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

从C++源代码文件make浮动目标模块.o文件的规则定义为：

```
$(obj_dir)/%.o: %.cc
    @echo ">>> Compiling" $< "<<<"
    $(CC) $(CFLAGS) -c -o $@ $<
```

这儿又一次使用了隐式规则。%号再次用于通配任意非空字符串。例如，这一规则说明了怎样从main.cc产生出arch/unknown-i386-linux/objects/main.o。不过目标代码也依赖许多括入在.cc文件中的.h文件。对于这些.h文件的依赖关系实际上是由另外的规则自动产生的。

首先，我们需要知道.h文件是在C++源文件编译时直接或间接括入的。g++可以自动为你搜索这些.h文件。但你需要使用g++编译选项MM。我们可以做一个试验，在threads目录中使用带MM选项的g++编译一下main.cc。输入：

```
$g++ MM –g Wall Wshadow –fwritable –string –I../threads –I../machine
–DTHREADS –DHOST_i386 –DHOST_LINUX –DCHANGED main.cc
```

你应当看到以下的输出：

```
  main.o: main.cc copyright.h utility.h ../threads/bool.h \
  ../machine/sysdep.h ../threads/copyright.h system.h thread.h \
  scheduler.h list.h ../machine/interrupt.h ../threads/list.h \
  ../machine/stats.h ../machine/timer.h ../threads/utility.h
```

g++列出了所有与main.o有依赖关系得.h文件。如果这些.h文件中的任何一个有修改，都将引发main.cc被重新编译产生新的main.o文件。

这一规则也应包括在Makefile.common文件中,以便对任何一个.o文件都可以自动的生成对应的依赖规则。首先Makefile.common应当生成一个存放每个源文件依赖关系文件的目录。我们规定在linux系统中这个目录是：arch/unknown-i386-linux/depends/。生成这些规则的定义为(仅以.cc文件的为例)：

```
$(depends_dir)/%.d: %.cc
    @echo ">>> Building dependency file for " $< "<<<"
    @$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \
    | sed '\''s@$*.o[ ]*:@$(depends_dir)/$(notdir $@)
$(obj_dir)/&@g'\'' > $@'
```

以上定义通过g++ 的MM开关列出对应.cc文件的依赖关系清单并通过管道将清单发送给sed命令，sed命令在清单的前面插入规定的目录路径后在指定的目录中生成对应的依赖关系文件.d。例如对于main.cc的依赖关系文件main.d其文件内容为：

```
arch/unknown-i386-linux/depends/main.d
arch/unknown-i386-linux/objects/main.o: main.cc copyright.h
```

```
utility.h ../threads/bool.h \
  ../machine/sysdep.h ../threads/copyright.h system.h thread.h \
  scheduler.h list.h ../machine/interrupt.h ../threads/list.h \
  ../machine/stats.h ../machine/timer.h ../threads/utility.h
```

然后Makefile.common将所有文件的依赖关系定义在变量dfile中，通过语句：include $(dfile)将所有它所建立的依赖关系文件都括进来。这些文件的内容就变为了Makefile关系规则的一部分。编译将结合这些规则产生目标代码。使用这一技术的重要性在于我们可以通过检测对应得依赖关系文件，知道编译的源代码使用了哪些.h文件。这一技术对于你在另一目录中构造一个新版的Nachos是非常有帮助的，在这个新目录中你仅需放入你要修改的源代码和头文件并且可以保证仅有这些文件被重新编译。

1.2.10 在另目录中构造一个改进的新内核

Nachos允许你在code/下任建的一个新目录中利用原有的内核源代码扩充和修改后重新构造。在这个新目录中可以仅有你想改变的源代码文件或增加一些你为内核源代码新增的文件。

例如，我们要在空目录../lab2/目录中重新构造一个仅改变了调度算法的新版Nachos内核。 假设这需要改变Scheduler类，使用新的scheduler.h和scheduler.cc文件。而其他所有的文件仍然使用
在../threads/，../machine/等目录中原有的文件。

为了这样做，首先你需要在../lab2/目录中重建或从../threads/目录中拷贝scheduler.h和scheduler.cc文件，从../threads/目录中递归的拷贝../arch/目录和Makefile,Makefile.local文件。接下来的工作是修改../lab2/中的Makefile.local文件，以便能在../lab2/中正确的构建新的Nachos（../lab2/中的Makefile文件无需修改）。在Makefile.local文件中定义了基本的CCFILE宏和重定义的INCPATH宏。如果新增了.cc文件你需要在CCFILE中声明。本例中CCFILE宏无需改变，因为你没有增加新的.cc文件，make会沿着vpaths定义的路径顺序查找所有不在当前目录中.cc文件。重定义的INCPATH需要修改。首先要把当前目录添加到INCPATH中：

```
INCPATH += -I../labe –I../threads –I../machine
```

这样做仅是声明了.cc文件中直接扩入的.h文件的查找路径，但是一些.cc文件中间接括入的.h文件的查找路径并不是按照INCPATH定义的路径查找的，它们是按照由g++ MM产生的依赖关系来查找的。因此一些不在当前目录中而又间接括入了当前目录中.h文件的.cc文件不会

随着当前目录中.h文件的修改而重新编译。例如main.cc文件括入了system.h文件，而system.h文件又括入了scheduler.h文件，现在的main.cc不会随着scheduler.h的改变而重新编译。解决这个问题的第一种方法是：查出不在本目录中所有与要修改的.h文件有间接关系的文件，将它们拷贝到当前目录中。但这种方法比较麻烦。解决这个问题的第二种方法是：利用-I-编译开关。-I-开关禁止处理与.cc文件在同一个目录中的.h文件，即关闭由g++ MM产生的依赖关系，让每个.cc文件按INCPATH定义的路径查找.h文件。我们可以在INCPATH中加入-I-开关：

```
INCPATH += -I- -I../labe –I../threads –I../machine
```

现在不需要查找和拷贝不在本目录中所有与要修改的.h文件有间接关系的文件了，make会根据我们在当前目录中所作的修改正确的重构新的系统。

第 2 章 操作系统内核线程通信设计

操作系统中多数进程和线程间都有协作关系，它们能够正确地并发执行需要不断的相互通信。解决进程间通信问题的一个最基本的方法就是由操作系统内核提供同步机制，通过同步机制实现进程和线程间的同步。本章通过对 Nachos 系统中几种同步机制的分析和实验来实现内核同步机制的设计和构造。

2.1 内核线程通信设计分析

2.1.1 进程同步 synchronization

对于协同工作的并发进程或线程，如果它们访问共享数据就需要同步或互斥的执行。有三种基本的方法控制并发进程或线程访问共享数据。

- Semaphore
- Critical Regins
- Monitor

而经常用来构造进程同步的是一些高级机制，例如：Monitor、Critical Regions。Monitor 是较广泛使用的同步机制，它具有模块化的结构，并且在语义上更利于理解。Critical Regions 用的较少，我们在此暂不讨论它。

2.1.2 临界区问题

每个并发进程或线程访问共享资源（或称为临界资源）的程序段称为临界区（critical section）。进程或线程要执行临界区内的代码时必须满足以下 3 个基本条件：

- Mutual exclusion　　互斥执行
- Progress　　空闲让进
- Bounded waiting　　有限等待

在仅有一个 CPU 的系统中，进程在临界区执行期间的同步问题可用禁止中断的方法近似解决。不过这种近似的解决方法不能解决在多处理器上并发

执行的问题。这是使用硬件解决同步问题的主要原因。硬件解决方案的主要问题是它们依赖于忙等待。忙等待使用循环重复测试，浪费了 CPU 的时间。由于这个原因，在多处理器的系统中常联合使用硬件和软件方法以完成高级的多处理器的同步问题。

信号量 semaphore

Semaphore 是使用最广且最重要的软件同步机制。Semaphore 上有两种操作：semaphore.wait()和 semaphore.signal()。最早也称为 semaphore.P()和 semaphore.V()。为了不与我们后面要讨论的 Monitor 的操作混肴，我们在讨论 semaphore 的操作时仍采用 P()和 V()操作的说法。

Nachos 信号量的实现是用了与经典教材中稍微不同的算法。从资源竞争的观点看上去这种描述更加自然。其算法可以描述为：

```
.P(){
      While( 信号量的值 V=0)
          将调用者线程推入阻塞队列 B，调用者线程阻塞；
      V = V-1
}
.V(){
    If (   阻塞队列 B 非空 ){
            从阻塞队列 B 中取出一个线程；
            把它推入系统就绪队列 R；
    }
    V   =   V+1;
}
```

2.1.3 Nachos 中信号量的实现

一个 Nachos 中的信号量是作为 Semaphore 类的对象实现的。Semaphore 类的定义可以在 threads/synch.h 中找到。

```
40  class Semaphore {
41    public:
42      Semaphore(char* debugName, int initialValue);       //
set initial value
43      ~Semaphore();                                       //
de-allocate semaphore
44      char* getName() { return name;}                     //
debugging assist
45
```

```
46      void P();     // these are the only operations on a semaphore
47      void V();     // they are both *atomic*
48
49    private:
50      char* name;         // useful for debugging
51      int value;          // semaphore value, always >= 0
52      List *queue;        // threads waiting in P() for the value
to be > 0
53  };
```

注意私有变量 queue 是一个指向所有在该信号量上阻塞的线程队列指针。

P()操作函数和 V()操作函数的定义在 threads/sync.h.cc 文件中：

```
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



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  }
```

注意，为保证这两个函数的操作都是原子操作，这两个函数都由 interrupt->SetLevel(IntOff)关中断和 interrupt->SetLevel(oldLevel)开中断栝起。

Nachos 中的 P()、V()操作关键的概念是保持信号量的值始终大于等于 0。即信号量的值代表了资源可利用量，当资源量等于 0 时说明线程无资源可用必需等待可用资源的释放。为了保持信号量的值始终大于等于 0,我们必须在 P()操作中使用 while 语句判断信号量的值是否等于 0。如果在使用 while 语句的地方使用了 if 语句，一些竟态条件可能引起信号量的值小于 0，从而发生错误。例如，有 3 个并发线程 A、B、C 它们并发执行，共享同一资源。信号量 S 负责管理对该资源的访问。假设此时 S 的值 V 为 0，之后有如下的一系列的操作：

1. 线程 A 因请求该资源，引用 P()操作而阻塞；
2. 线程 B 释放该资源,引用 V()操作唤醒了线程 A,将 A 推入了就绪队列，使 V=1；
3. 线程 C 首先从就绪队列中被选中执行,C 也请求该资源,引用 P()操作，使 V=0；开始访问该资源；
4. 线程 A 从就绪队列中被选中执行，如果这里使用 if 语句，线程 A 不会再去判断 V 是否等于 0,而是使 V=-1，也开始访问该资源，从而发生了与线程 C 非互斥的使用同一资源的错误。而如果这里使用 while 语句，线程 A 会发现 V 再次等于 0,而再次进入阻塞队列，保证了 V 的值始终大于等于 0，从而避免了与线程 C 同时使用同一资源的错误。

由此可见，那些由 V()操作唤醒刚进入就绪队列的线程仍然被当作阻塞态线程，因为它们还未完成它们调用 P()操作中递减 V 值的工作。

2.1.4 信号量的应用

信号量属于一种低级进程或线程通信原语。信号量可以用于并发进程或线程的临界区管理，使并发进程或线程互斥的访问资源。信号量也可以用于协作进程或线程的同步，例如使用信号量实现经典的生产者/消费者在有界缓冲上同步的算法。但不恰当的使用信号量会产生死锁，在阻塞队列中进/线程较多时也可能引发饥饿现象。

2.1.5 锁 Lock

锁是另一类低级通信原语。它类似于二值信号量（但有所不同）。一个二值信号量其值为一个不超过 1 的整数。锁 L 有两种操作：

- 获取锁 L.Acquire()和
- 释放锁 L.Release()

它们类似于（但有所不同）一个二值信号量上的 P()和 V()操作。

锁具有两种状态：acquired 和 released。初始态必须是 released。在一个 released 锁上的 Release 操作会产生一个错误。仅允许获取到锁的进程或线程执行 Released 操作，否则也会产生错误。这是它与二值信号量不同的地方。在一个 acquired 态锁上的 Acquire 操作会把调用者推入锁等待队列。在一个 acquired 态锁上的 Release 操作会唤醒所队列中的一个等待者，使它进入就绪队列。像信号量上的 P()、V()操作一样，Acquire()和 Release()操作也应当是原子操作。

显然，锁可以使用二值信号量来实现。Nachos 中的锁的实现正是这样做的。在 Nachos 中锁类 Class Lock 定义在 threads/synch,h 文件的 67-85 行上：

```
67  class Lock {
 68    public:
 69      Lock(char* debugName);              // initialize lock
to be FREE
 70      ~Lock();                            // deallocate lock
 71      char* getName() { return name; }    // debugging assist
 72
 73      void Acquire(); // these are the only operations on a lock
 74      void Release(); // they are both *atomic*
 75
 76      bool isHeldByCurrentThread();       // true if the
current thread
```

```
 77                                              // holds this lock.
Useful for
 78                                              // checking in
Release, and in
 79                                              // Condition
variable ops below.
 80
 81    private:
 82      char* name;                             // for debugging
 83      Thread *owner;                          // remember who
acquired the lock
 84      Semaphore *lock;                        // use semaphore for
the actual lock
 85  };
```

在 84 行上可以看到 Nachos 中锁的是通过一个私有信号量对象指针 lock 实现的。在 83 行上说明了一个私有的线程对象指针 owner，通过它可以知道谁是该锁当前的拥有者。76 行上的 isHeldByCurrentThread()函数用于判断当前线程是否是锁的拥有者,这对于避免发生锁操作错误和后面讨论的条件变量的操作很有用。

锁操作 Acquire()和 Release()的代码定义在 threads/synch.cc 文件 133-156 行：

```
133  void Lock::Acquire()
134  {
135      IntStatus oldLevel = interrupt->SetLevel(IntOff);  //
disable interrupts
136
137      lock->P();                                 // procure the
semaphore
138      owner = currentThread;                     // record the new
owner of the lock
139      (void) interrupt->SetLevel(oldLevel); // re-enable
interrupts
140  }
141
```

```
142
//----------------------------------------------------------------
--------
143  // Lock::Release
144  //      Set the lock to be free (i.e. vanquish the semaphore).
Check
145  //      that the currentThread is allowed to release this lock.
146
//----------------------------------------------------------------
--------
147  void Lock::Release()
148  {
149      IntStatus oldLevel = interrupt->SetLevel(IntOff);  //
disable interrupts
150
151      // Ensure: a) lock is BUSY  b) this thread is the same one
that acquired it.
152      ASSERT(currentThread == owner);
153      owner = NULL;                              // clear the owner
154      lock->V();                                  // vanquish the
semaphore
155      (void) interrupt->SetLevel(oldLevel);
156  }
```

可以看到获取锁的操作 Acruire()主要是调用了信号量的 P()操作,释放锁的操作 Release()主要是调用了信号量的 V()操作。但在第 138 行和 152 行上的语句，确保了调用 Release()操作的线程与锁的获取者是同一个线程。这样任何一个线程试图在 released 态的锁上执行 Release()操作都将在 152 行上引发一个错误，从而打断它的执行。
当锁处于 released 态时，变量 owner 就设置为 NULL，此时任何一个线程使用 Acquire()获取该锁时都会成为该锁的拥有者。

2.1.6 管程 Monitor

管程是一种高级抽象数据类型，它支持在它的函数中隐含互斥操作。结合条件变量和其他一些低级通信原语，管程可以解决许多仅用低级原语不能

解决的同步问题。利用管程提供一个不会发生死锁的哲学家就餐问题就是一个很好的例子。

2.1.7 互斥执行 Mutal

管程封装了并发进程或线程要互斥执行的函数。为了让这些并发进程或线程在管程内互斥的执行，管程的实现必须隐含的具有锁或二值信号量。

2.1.8 条件变量 Condition Variables

如果没有条件变量，管程就不会很有用。多数同步问题要求在管程中说明条件变量。条件变量提供了一种对管程内并发协作进程的同步机制。条件变量代表了管程中一些并发进程或线程可能要等待的条件。一个条件变量管理着管程内的一个等待队列。如果管程内某个进程或线程发现其执行条件为假，则该进程或线程就会被条件变量挂入管程内等待该条件的队列。如果管程内另外的进程或线程满足了这个条件，则它会通过条件变量再次唤醒等待该条件的进程或线程，从而避免了死锁的产生。所以，一个条件变量 C 应具有两种操作 C.wait()和 C.signal()。
当管程内同时出现唤醒者和被唤醒者时，由于要求管程内的进程或线程必须互斥执行，因此就出现了两种样式的条件变量：

- Mesa Style(signal-and-continue)：唤醒者进程或线程继续执行，被唤醒者进程或线程等到唤醒者进程或线程阻塞或离开管程后再执行。
- Hoare Style(signal-and-wait)：被唤醒者进程或线程立即执行，唤醒者进程或线程阻塞，直道被唤醒者阻塞或离开管程后再执行。

2.1.9 Nachos 系统中条件变量的实现

Nachos 使用 Condition 类定义和实现了一个 Mesa 样式的条件变量。其类定义在 threads/synch.h 文件的第 119-139 行：

```
119  class Condition {
120    public:
121      Condition(char* debugName);          // initialize
condition to
122                                           // "no one waiting"
```

```
123      ~Condition();                          // deallocate the
condition
124      char* getName() { return (name); }
125
126      void Wait(Lock *conditionLock);        // these are the 3
operations on
127                                             // condition
variables; releasing the
128                                             // lock and going to
sleep are
129                                             // *atomic* in
Wait()
130      void Signal(Lock *conditionLock);      // conditionLock
must be held by
131      void Broadcast(Lock *conditionLock);// the currentThread
for all of
132                                             // these operations
133
134    private:
135      char* name;
136      List* queue;   // threads waiting on the condition
137      Lock* lock;    // debugging aid:  used to check correctness
of
138                     // arguments to Wait, Signal and Broacast
139  };
```

有一个等待队列指针 queue，用于排放等待该条件的进程或线程。
类构造函数 Condition 所携带的参数 conditionLock 是一个用于互斥进入管程的锁指针。它将被传送给主要的成员函数以判断该锁是否属于条件变量的调用者。条件变量主要的操作 Wait()和 Signal()定义在 threads/synch.cc 文件的 206-241 行：

```
206  void Condition::Wait(Lock* conditionLock)
207  {
208      IntStatus oldLevel = interrupt->SetLevel(IntOff);
209
210      ASSERT(conditionLock->isHeldByCurrentThread());  //
check pre-condition
211      if(queue->IsEmpty()) {
```

```
212          lock = conditionLock;  // helps to enforce
pre-condition
213      }
214      ASSERT(lock == conditionLock); // another pre-condition
215      queue->Append(currentThread);  // add this thread to the
waiting list
216      conditionLock->Release();      // release the lock
217      currentThread->Sleep();        // goto sleep
218      conditionLock->Acquire();      // awaken: re-acquire the
lock
219      (void) interrupt->SetLevel(oldLevel);
220  }
```

对于 Wait 函数可正确执行有两个重要的先决条件，一是调用线程所传递的锁必须是它自己获取的锁。210 行就是对这个先决条件的检查。另外一个先决条件是调用线程所传递的锁必须与条件变量对象中存放的锁是同一把锁，对于这个条件的检查是由 211 和 214 行联合完成的。可以看到在先决条件满足后调用 Wait 的线程在 216-217 行上被推入等待队列，释放锁并进入阻塞睡眠状态。当条件满足被唤醒后，该线程将在 218 行处重新获取锁后继续执行。

```
229  void Condition::Signal(Lock* conditionLock)
230  {
231      Thread *nextThread;
232      IntStatus oldLevel = interrupt->SetLevel(IntOff);
233
234      ASSERT(conditionLock->isHeldByCurrentThread());
235      if(!queue->IsEmpty()) {
236          ASSERT(lock == conditionLock);
237          nextThread = (Thread *)queue->Remove();
238          scheduler->ReadyToRun(nextThread);      // wake up
the thread
239      }
240      (void) interrupt->SetLevel(oldLevel);
241  }
```

注意对唤醒操作 Signal 仅检查其调用者线程所传递的锁是否是该线程自己获取的锁这一先决条件。如果等待队列不空，调用者线程必须从等待

队列中取出一等待线程，并将其放入就绪队列。注意，此时这个唤醒者并没有释放它所拥有的锁，这反映出它是一种 Mesa 样式的条件变量的操作。另外可以看出 Mesa 样式的条件变量仅需一个等待队列就可实现对于线程的同步，而 Hoare 样式的条件变量则不行。
同以上介绍的其他低级通信函数一样，Wait 函数和 Signal 函数也是原子操作。

2.1.10 Nachos 中管程的实现

如果仅使用 Mesa 样式的条件变量，管程的实现是较为简单的。这种管程仅需要一个锁，其中的每个条件变量仅需一个条件等待队列就可以了。对于它的每一个成员函数，在函数的开始处必须调用 Acquire 获取锁，在函数的结束处必须调用 Release 释放锁。这样管程内的每个线程就可以排他的互斥执行，同时通过 Mesa 样式的条件变量相互通信。

下面我们分析一个 Nachos 中可由多线程共享的叫做 SynchList 的同步链表类。这个类实际上就是一个由 Mesa 样式的条件变量构造的管程。
SynchList 的定义在文件 threads/synchlist.h 文件的 24-40 行：

```
24  class SynchList {
25    public:
26      SynchList();                  // initialize a synchronized
list
27      ~SynchList();                 // de-allocate a synchronized
list
28
29      void Append(void *item);      // append item to the end of
the list,
30                                    // and wake up any thread
waiting in remove
31      void *Remove();               // remove the first item from
the front of
32                                    // the list, waiting if the
list is empty
33                                    // apply function to every
item in the list
34      void Mapcar(VoidFunctionPtr func);
35
```

```
36    private:
37      List *list;                 // the unsynchronized list
38      Lock *lock;                 // enforce mutual exclusive
access to the list
39      Condition *listEmpty;       // wait in Remove if the list
is empty
40  };
```

注意 SynchList 类的私有数据中有一个指向非同步化链表对象的 list 指针，一个指向锁对象的 lock 指针和一个指向条件变量对象的 listEmpty 指针。它的两个主要的成员函数 Append 负责将 item 所指对象追加到 list 所指链表的队列中，函数 Remove 负责从 list 所指链表的队列中取出队列首项并返回出队对象的指针.

函数 Append 和 Remove 定义在文件 threads/synchlist.cc 文件的 53-82 行：

```
53  void
54  SynchList::Append(void *item)
55  {
56      lock->Acquire();            // enforce mutual exclusive
access to the list
57      list->Append(item);
58      listEmpty->Signal(lock);    // wake up a waiter, if any
59      lock->Release();
60  }

70  void *
71  SynchList::Remove()
72  {
73      void *item;
74
75      lock->Acquire();                    // enforce mutual
exclusion
76      while (list->IsEmpty())
77          listEmpty->Wait(lock);          // wait until list
isn't empty
78      item = list->Remove();
79      ASSERT(item != NULL);
```

```
80      lock->Release();
81      return item;
82  }
```

注意，这两个函数在开始处都调用了 Acquire()以获取锁，在结束处调用了 Release()以释放锁，从而保证线程的互斥执行。在函数 Remove 中，如果队列为空，调用线程会通过条件变量 listEmpty 的 Wait 操作进入 lock 所指等待队列阻塞。在函数 Append 中，调用线程将一项加入项目队列从而满足了队列项目不为空的条件，并且它主动使用 listEmpty 的 Signal 操作去唤醒等在 lock 锁队列上的线程，这样因调用 Remove 而阻塞的线程又会从 78 行上继续运行，从队列中取出一项。

2.2 进程和线程通信实验内容

通过上节的分析我们已经了解了基本 Nachos 系统几种同步对象的构造原理和设计的方法。本节我们将通过实验来体验操作系统内核同步机制的设计和开发，扩充基本 Nachos 系统同步机制的构造和功能，实现自主设计的操作系统内核同步机制。

2.2.1 实验目的

- 理解 Nachos 系统是怎样实现的信号量机制
- 怎样使用 Nachos 系统提供的信号量解决生产/消费者问题
- 进一步练习怎样在 Nachos 中建立多线程
- 调试和分析多线程工作环境中出现的死锁和饥饿现象
- 构造一个能够解决生产/消费者问题的 Hoare 样式的管程
- 应用 Hoare 样式的管程解决生产/消费者问题

2.2.1 生产/消费者问题

生产消费者问题是操作系统设计中经常遇到的问题。多个生产者和消费者线程访问在共享内存中的环形缓冲。生产者生产产品并将它放入环形缓冲，同时消费者从缓冲中取出产品并消费。当缓冲区满时生产者阻塞并且当缓冲区有空时生产者又重新工作。类似的，消费者当缓冲区空时阻塞并且当缓冲区有产品时又重新工作。显然，生产者和消费者需要一种同步机制以

协调它们的工作。

2.2.3 Nachos 中的 main 程序

当你启动 Nachos 系统时，第一个被执行的程序模块就是 main。Nachos 中每个子目录中都有一个内容不尽相同的 main.cc。请查看一下../threads//main.cc，研究其以下的功能：

- Nachos 是怎样解释命令行的。
- Nachos 的内核是怎样初始化的。
- main 程序的线程是怎样建立的执行 SimpleThread 函数的另一个线程的。

2.2.4 使用信号量解决生产/消费者同步问题

在../lab3 目录中，你可以找到文件：mian.cc，prodcons++.cc，ring.cc 和 ring.h。

文件 ring.cc 和 ring.h 定义和实现了一个为生产者和消费者使用的环形缓冲类 Ring。这两个文件已经完成了，不需要我们作任何改动。在这个目录中的 main.cc 是../threads 目录中 main.cc 的改进版，我们也不必改动了。

在新的 main.cc 中调用了函数 ProdCons()而不是函数 ThreadTest()。函数 ProdCons()定义在文件 prodcons++.cc 中。假设这个文件包含着建立两个生产者和两个消费者线程以及实现两个生产者消/者问题算法的代码。不过，这个文件并不完整，我们的实验任务就是完成这个文件，构造一个能工作的生产者/消费者问题算法。

文件 prodcons++.cc 文件包含所有的数据结构和函数接口，但涉及到生产者和消费者线程的建立以及生产者/消费者问题算法的代码没有给出，只在对应位置通过注释给出了要补充完成的代码的要求。可以在../lab3/中执行 make 构造一个带有生产者/费者问题解法的新的 Nachos 系统。但是因为 prodcons.++.cc 并没完成，所以它并不能正确地工作。我们的实验任务是：

a) 读文件 ring.h 和 ring.h，了解类 Ring 的各个属性和方法的实现细节
b) 读文件 main.cc，了解它的主要工作过程。
c) 读文件 prodcons++.cc，了解：
    1. 程序的结构。
    2. 程序要完成的任务。
d) 完成文件 prodcons++.cc 中所有的程序设计
e) 使用命令 make 编译新的 Nachos 并且测试新的 Nachos 工作是否正确。

这个例子中两个生产者和两个消费者，每个生产者都产生了四条信息，由消费者写入两个文件 tmp_0 和 tmp_1 中。例如，输出可能为：

文件 tmp 0 中的内容为：

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

文件 tmp 1 中的内容为：

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;

以上结果正确吗？测试这个程序正确的标准是什么？对应生产者/费者问题的概念，一个正确的实现应当保证以下标准：

- 所有由生产者线程产生的信息都被消费者接受并写入到输出文件中。
- 输出文件没有丢失的信息和多余的信息
- 从同一个生产者生产出的信息和由同一个消费者接受到的信息的顺序应当是递增的。
- 重复运行带有不同随机数种子的 Nachos 系统，输出总能满足以上情况。

2.2.5 使用管程解决生产/消费者同步问题

管程是一种高级同步结构。管程实质上是带有同步语义的抽象数据类型 ADT。面向对象的程序设计语言像 C++、Java 等都通过类定义来实现 ADT。在这个实验中要求你为 Nachos 内核添加一个能够解决生产者/消费者问题的管程机制 Ring，并且这个管程采用 Hoare 样式的条件变量。为了实现这个管程需要定义一个 Ring 类，在这个类中封装上生产者/消费者问题中所有与同步和互斥操作有关的属性和方法。例如我们可以定义：。

```
class Ring {
  public:
    Ring(int sz);            // Constructor: initialize variables, allocate space.
    Ring();                  // Destructor: deallocate space allocated above.
```

```
    void Put(slot *message); // Put a message the next empty slot.
    void Get(slot *message); // Get a message from the next full slot.
    int Full();              // Returns non0 if the ring is full, 0 otherwise.
    int Empty();             // Returns non0 if the ring is empty, 0 otherwise.
  private:
    int size;                // The size of the ring buffer.
    int in, out;             // Index of Put and Get
    slot *buffer;            // A pointer to an array for the ring buffer.
    int current;             // the current number of full slots in the buffer
    Condition_H *notfull;    // condition variable to wait until not full
    Condition_H *notempty; // condition variable to wait until not empty
    Semaphore *mutex;        //semaphore for the mutual exclusion
    Semaphore *next;         //semaphore for "next" queue
    int next_count;          // the number of threads in "next" queue
};
```

有了以上的类定义，对于解决像生产者/消费者这类问题就不必再使用像信号量一类的低级同步机制了。相应的在以上实验中的 prodcons++.cc 就可以简化为：

对于生产者的过程：

```
  void   Producer(_int which)
  {
          int num;
          slot *message = new slot(0,0);
          for (num = 0; num < N_MESSG ; num++) {
          // the code to prepare the message goes here.
          // ..
          ring>Put(message);
    }
  }
```

对于消费者的过程：

```
  void   Consumer(_int which)
  {
       char str[MAXLEN];
       char fname[LINELEN];
       int fd;
       slot *message = new slot(0,0);
```

```
        sprintf(fname, "tmp_%d", which);
        printf("file name is %s \n", fname);
        if ( (fd = creat(fname, 0600) ) == 1)
        {
            perror("creat: file create failed");
            exit(1);
        }
        for (; ; ) {
            ring>Get(message);
            sprintf(str,"producer id > %d; Message number > %d;\n",
                message>thread_id,
                message>value);
        if ( write(fd, str, strlen(str)) == 1 ) {
            perror("write: write failed");
            exit(1);
        }
    }
}
```

可以看到在生产者/消费者过程中 ring 对象的 Put 操作和 Get 操作已经代替信号量来完成同步操作了。这样就避免了由于不恰当的使用信号量而引起的死锁和饥饿问题。

下面我们的实验所要完成的任务就是：

- 定义和实现 Ring 类中 Hoare 样式的条件变量 Condition_H 类
- 实现 Ring 类中定义的各个方法
- 使用 Ring 类实现生产者/消费者算法

第 3 章 操作系统内存管理设计

内存管理是操作系统设计的另一核心任务，现代操作系统进程和线程的并发执行性要求多道程序必须同时驻留在内存中。同时为了能使程序局部装入即可执行而构造了虚拟内存。本章通过对 Nachos 系统内存管理体系的分析和实验来实现多道程序同时驻留在内存和虚拟内存的设计和构造。

3.1 内存管理设计分析

我们前面提已经提到每个用户程序有一个地址空间。当进程运行时这个空间至少有一部分驻留在计算机的内存。这一部分讨论在单一的物理空间中怎样实现多个并发进程的内存地址空间。我们仍采用 Nachos 系统来说明怎样实现由 MIPS 模拟机模拟的 MIPS 物理内存。

3.1.1 从程序空间到物理空间

为了在计算机上运行一个应用程序，我们应当进行一些什么工作呢？主要涉及到三个步骤：

1. 编译和转换源代码到它的目标代码
2. 连接程序将所有的标代码模块链接成一个单一的可执行模块，即众所周知的可装入模块。
3. 装入程序将可装入模块装入计算机内存。

每个模块都有属于它的正文段，初始的数据段，符号表和重定位信息。链接程序的任务是合并所有的模块为一个带有各个目标模块的交叉引用和未解决的库引用的单一的可装入模块。这就要求改变每一个目标模块内部的地址引用，解决它们之间的外部引用来重新定位目标模块。

装入模块提供程序的逻辑地址空间。这个地址空间的开始地址是 0，并且模块所有的地址都参考这个开始的 0 地址。

在程序运行之前，所有的逻辑地址空间都需要变换为物理地址空间。这

一变换通常都由硬件的内存管理部件 MMU 完成。操作系统可以由改变重定位寄存器的值来重定位程序的装入模块到内存不同的段上。

3.1.2 内存管理机制

有四种逻辑地址到物理地址的变换机制：

1. 连续内存分配方式 Contiguous Allocation
2. 页式内存分配方式 Paging
3. 段式内存分配方式 Segmentation
4. 段页式内存分配方式 Segmentation with Paging

基本的 Nachos 系统是采用页式内存分配方式管理用户内存空间的

3.1.3 MIPS 模拟机

在讨论 Nachos 中的内存地址变换之前，我们需要讨论 Nachos 中的 MIPS 模拟器。这是因为此时由 Nachos 支持的用户程序是一个 MIPS 指令的二进制可执行文件。为了能使用户程序在非 MIPS 机上的 Nachos 内核上工作，Nachos 发行版都带有一个 MIP 模拟器。一个 MIPS 机的模拟器用于在 Nachos 中运行用户程序。它是通过定义在 machine/machine.h 第 107-190 行的一个 Machine 类实现的。这个模拟的 MIPS 机的主要部件是：

用户寄存器组： Int registers[NumTotlRegs];

- 主存： char *mainMemory;
- 当前用户的页表： TranslationEntry *pageTable;
- 块表： TranslationEntry *tlb;

机器的操作由以下各种函数来模拟：

- 内存读写：
  - 从 addr 单元中读长度为 size 字节的数据到 value 指向的单元中
    bool ReadMem(int addr,int size,int *value);
  - 向 addr 单元中写入 value 单元中长度为 size 字节的数据
    bool WriteMem(int addr,int size,int value);
- 寄存器读写：
  - 从编号为 num 的 CPU 寄存器中读出数据
    int ReadRegister(int num);
  - 将 value 单元中的数据写入编号为 num 的 CPU 寄存器中
    void WriteRegister(int num,int value);

- 指令的解释执行：
  - 运行用户程序：void Run();
  - 执行一条 MIPS 指令：void OneInstuction(Instruction *instr);
- 异常控制:
  - 处理一个类型为 which 的异常
    void RaiseException(ExceptionType which,int badVaddr);
- 地址变换:
  - 将单元长度为 size 的逻辑地址 virAddr 变换为物理地址 physAddr。该单元是否可写由 writing 决定。
    ExceptionType Translate(int VirAddr,int *physAddr,int siae,bool writing);

3.1.4 MIPS 指令的解释执行

在文件 machine/mips.cc 第 30-45 行上的 Run()函数负责设置机器进入用户态，并且在一个无限循环中模拟取 CPU 指令和执行 CPU 指令的过程。

```
30  void
31  Machine::Run()
32  {
33      Instruction *instr = new Instruction;  // storage for
decoded instruction
34
35      if(DebugIsEnabled('m'))
36          printf("Starting thread \"%s\" at time %d\n",
37                  currentThread->getName(), stats->totalTicks);
38      interrupt->setStatus(UserMode);
39      for (;;) {
40          OneInstruction(instr);
41          interrupt->OneTick();
42          if (singleStep && (runUntilTime <= stats->totalTicks))
43            Debugger();
44      }
45  }
```

函数 OneInstuction()模拟取下一条指令和执行一条指令的周期。

```
101     // Fetch instruction
102     if (!machine->ReadMem(registers[PCReg], 4, &raw))
103         return;                 // exception occurred
104     instr->value = raw;
105     instr->Decode();
106
……
117     // Compute next pc, but don't install in case there's an
error or branch.
118     int pcAfter = registers[NextPCReg] + 4;
119     int sum, diff, tmp, value;
120     unsigned int rs, rt, imm;
121
122     // Execute the instruction (cf. Kane's book)
123     switch (instr->opCode) {
124
125       case OP_ADD:
126         sum = registers[instr->rs] + registers[instr->rt];
127         if (!((registers[instr->rs] ^ registers[instr->rt])
& SIGN_BIT) &&
128             ((registers[instr->rs] ^ sum) & SIGN_BIT)) {
129             RaiseException(OverflowException, 0);
130             return;
131         }
132         registers[instr->rd] = sum;
133         break;
……
546       case OP_RES:
547       case OP_UNIMP:
548         RaiseException(IllegalInstrException, 0);
549         return;
550
551       default:
552         ASSERT(FALSE);
553     }
554
555     // Now we have successfully executed the instruction.
556
```

```
557       // Do any delayed load operation
558       DelayedLoad(nextLoadReg, nextLoadValue);
559
560       // Advance program counters.
561       registers[PrevPCReg] = registers[PCReg];    // for
debugging, in case we
562                                                   // are
jumping into lala-land
563       registers[PCReg] = registers[NextPCReg];
564       registers[NextPCReg] = pcAfter;
565   }
```

在 102 行取出一条指令和在 105 行指令解码之后，CPU 的处理进入 123-554 行间一个大的 switch 语句中执行对应的操作。如果该指令执行成功，在 558-564 行上 CPU 指令计数器被向前推进，准备执行下一条指令。
以上这个大的 switch 语句解释执行了多数 MIPS 计算机的指令。有关 MIPS 指令集的说明可见文档“SPIM S20 A MIPS R2000 Simulator”

3.1.5 Nachos 系统中的用户程序

在 Nachos 系统中用户的的 MIPS 的可执行二进制程序是由以下两个步骤产生的：

- 用 gcc MIPS 交叉编译程序编译用户的.c 源代码文件，产生通常在 Uinx 系统中可执行的 Coff 格式文件。
- 使用 Coff2noff 程序转换以上 Coff 文件，在../bin 目录中产生在 Nachos 系统中可执行的 Noff 格式文件。

Noff 格式是 Nachos 系统规定的可执行文件格式。它类似于 Coff 格式。Noff 格式在 bin/noff.h 文件中定义如下：

```
8 #define NOFFMAGIC  0xbadfad     /* magic number denoting Nachos
9                     * object code file
10                    */
11
12 typedef struct segment {
13   int virtualAddr;      /* location of segment in virt addr space */
14   int inFileAddr;       /* location of segment in this file */
15   int size;             /* size of segment */
```

```
16 } Segment;
17
18 typedef struct noffHeader {
19  int noffMagic;        /* should be NOFFMAGIC */
20   Segment code;        /* executable code segment */
21   Segment initData;        /* initialized data segment */
22   Segment uninitData;      /* uninitialized data segment --
23                * should be zero'ed before use
24                */
25 } NoffHeader;
```

可以看出 Noff 格式头主要定义了 Nachos 系统中用户可执行文件的段表格式以及段的类型。Nachos 系统将根据这一定义将 Noff 格式的文件加载到 Nachos 的用户内存解释执行。

所有用于测试的用户 C 程序源代码文件都在目录 test/中。其中有：

Makefile arch/ manult.c shell.c start.s
Makefile.orig halt.c
script sort.c

为了产生其中 C 程序的 Noff 格式文件，如果已安装好 gcc MIPS 交叉编译程序，就可以在 test/目录中执行 make。test/目录中的 Makefile 文件声明将用 coff2noff 和 coff2float 程序转换 Coff 格式的 MIPS 可执行文件到 Noff 和 flat 格式的可执行文件。make 完成后 test/目录中将产生与.c 文件相对应的一系列.noff 和.flat 文件：halt.noff halt.flat manult.noff manult.flat …

3.1.6 用户进程的地址空间

用户程序的地址空间是由类 AddrSpace 定义的。这个类的定义在 userprog/addrspace.h 文件中。

```
21  class AddrSpace {
22    public:
23      AddrSpace(OpenFile *executable);    // Create an address
space,
24                                          // initializing it
with the program
25                                          // stored in the file
"executable"
```

```
26      ~AddrSpace();                       // De-allocate an
address space
27
28      void InitRegisters();               // Initialize
user-level CPU registers,
29                                          // before jumping to
user code
30
31      void SaveState();                   // Save/restore
address space-specific
32      void RestoreState();                // info on a context
switch
33
34    private:
35      TranslationEntry *pageTable;        // Assume linear page
table translation
36                                          // for now!
37      unsigned int numPages;              // Number of pages in
the virtual
38                                          // address space
39  };
```

可以看到 AddrSpace 中包括了初始化和管理用户进程空间的一些方法：

- AddrSpace() 根据打开的用户可执行文件构造用户内存空间。
- InitRegisters() 初始化用户 CPU 寄存器。
- SaveState() 保存用户空间现场。
- RestoreState() 恢复用户空间现场。

一个指向 TranslationEntry 类的指针 pageTable 给出了页表数组的起始地址。而整数 numPages 给出了页表数。

Nachos 系统中的页表结构是由 TranslationEntry 类定义的，该定义在文件 machine/translation.h 的 30-43 行：

```
30  class TranslationEntry {
31    public:
32      int virtualPage;    // The page number in virtual memory.
33      int physicalPage;   // The page number in real memory
(relative to the
34                          //  start of "mainMemory"
```

```
35      bool valid;         // If this bit is set, the translation
is ignored.
36                          // (In other words, the entry hasn't
been initialized.)
37      bool readOnly;      // If this bit is set, the user program
is not allowed
38                          // to modify the contents of the page.
39      bool use;           // This bit is set by the hardware every
time the
40                          // page is referenced or modified.
41      bool dirty;         // This bit is set by the hardware every
time the
42                          // page is modified.
43  };
```

virtualPage 是用户逻辑页号，physicalPage 是对应的物理块号。valid 指示该页是否有效，readOnly 指示该页是否可写。use 是引用位，dirty 是改写位，利用这两位信息可以构造虚拟内存。

用户地址空间的格式化是由调用 AddrSpace 类的构造函数完成的。构造函数的参数是一个打开的 Noff 格式的二进制可执行文件。在此时 Nachos 内核还不具有自己的文件系统，因此它利用了 Unix/Linux 的文件系统来打开一个可执行文件。在有了 Nachos 自己的文件系统之后它将使用自己的文件系统打开一个可执行文件。可以从 userprog/目录中的 Makefile 文件中看到对于文件系统引用的定义：

```
22  ifdef MAKE_FILE_FILESYS_LOCAL
23  DEFINES += -DUSER_PROGRAM
24  else
25  DEFINES += -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS_STUB
```

22-23 行说明如果 Nachos 本地文件系统建立则无需定义使用 Unix 文件系统的标志 FILESYS_STUB；否则在 25 行通过-DFILESYS_STUB 加入使用 Unix 文件系统的标志 FILESYS_STUB。这个 FILESYS_STUB 标志在../filesys/filesys.h 中被引用：

```
41  #ifdef FILESYS_STUB             // Temporarily implement file
system calls as
42                                  // calls to UNIX, until the
real file system
```

```
43                                      // implementation is
available
44  class FileSystem {
45    public:
46      FileSystem(bool format) {}
47
48      bool Create(char *name, int initialSize) {
49          int fileDescriptor = OpenForWrite(name);
50
51          if (fileDescriptor == -1) return FALSE;
52          Close(fileDescriptor);
53          return TRUE;
54          }
55
56      OpenFile* Open(char *name) {
57            int fileDescriptor = OpenForReadWrite(name, FALSE);
58
59            if (fileDescriptor == -1) return NULL;
60            return new OpenFile(fileDescriptor);
61        }
62
63      bool Remove(char *name) { return (bool)(Unlink(name) ==
0); }
64
65  };
66
67  #else // FILESYS
```

为了与 Nachos 文件系统调用名称统一，当前的 Nachos 打开文件系统调用函数 OpenForWrite 和 OpenForReadWrite 封装了对应的 Unix 文件系统调用。打开文件的一些操作如：读、写、寻道也是封装的相应的 Unix 文件系统调用。

现在可以看一下用户进程的地址空间在 AddrSpace 构造函数中是怎样被格式化的了：

```
60  AddrSpace::AddrSpace(OpenFile *executable)
61  {
62      NoffHeader noffH;
```

```
63      unsigned int i, size;
64
65      executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
66      if ((noffH.noffMagic != NOFFMAGIC) &&
67                  (WordToHost(noffH.noffMagic) == NOFFMAGIC))
68          SwapHeader(&noffH);
69      ASSERT(noffH.noffMagic == NOFFMAGIC);
70
71  // how big is address space?
72      size = noffH.code.size + noffH.initData.size +
noffH.uninitData.size
73                              + UserStackSize;        // we need to
increase the size
74                                                      // to leave
room for the stack
75      numPages = divRoundUp(size, PageSize);
76      size = numPages * PageSize;
77
78      ASSERT(numPages <= NumPhysPages);           // check we're
not trying
79                                                      // to run
anything too big --
80                                                      // at least
until we have
81                                                      // virtual
memory
82
83      DEBUG('a', "Initializing address space, num pages %d, size
%d\n",
84                                              numPages, size);
85  // first, set up the translation
86      pageTable = new TranslationEntry[numPages];
87      for (i = 0; i < numPages; i++) {
88          pageTable[i].virtualPage = i;   // for now, virtual
page # = phys page #
89          pageTable[i].physicalPage = i;
90          pageTable[i].valid = TRUE;
91          pageTable[i].use = FALSE;
```

```
92          pageTable[i].dirty = FALSE;
93          pageTable[i].readOnly = FALSE;  // if the code segment
was entirely on
94                                          // a separate page, we
could set its
95                                          // pages to be
read-only
96      }
97
98  // zero out the entire address space, to zero the unitialized
data segment
99  // and the stack segment
100     bzero(machine->mainMemory, size);
101
102  // then, copy in the code and data segments into memory
103     if (noffH.code.size > 0) {
104         DEBUG('a', "Initializing code segment, at 0x%x, size
%d\n",
105                         noffH.code.virtualAddr,
noffH.code.size);
106
executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]
),
107                         noffH.code.size,
noffH.code.inFileAddr);
108     }
109     if (noffH.initData.size > 0) {
110         DEBUG('a', "Initializing data segment, at 0x%x, size
%d\n",
111                         noffH.initData.virtualAddr,
noffH.initData.size);
112
executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualA
ddr]),
113                         noffH.initData.size,
noffH.initData.inFileAddr);
114     }
115
```

```
116   }
```

在第 65 行上，首先从可执行文件中读出 Noff 头部信息。然后再 72-73 行上计算地址空间的长度。之后计算出地址空间需要多少页，86 行上分配页表，87-96 行上这个页表被初始化，这样页号和帧号就都填写好了。

第 100 行上清理有 machine->mainMemory 模拟的物理内存。注意 machine 是一个全局变量，当 Nachos 内核创建时它被初始化指向一个模拟的 MIPS 计算机。

第 103-114 行上，代码段和初始化的数据段从可执行文件中被装载到分配给它的物理地址空间中。非初始化的数据段不必装入，因为它的对应段已包含了初始化的 0 值。

3.1.7 Nachos 中的内存管理

你已经看到了用户的可执行文件是怎样装入到模拟的 MIPS 机的物理内存中去的了。装入程序的映像格式化了用户进程地址空间。当然，这一地址空间是一个逻辑的地址空间。

在程序执行时逻辑地址需要需要变换为物理空间。在真实的计算机中，这一工作是由 MMU 硬件完成的。当变换发生错误时 MMU 会自动发出各种异常中断。

Nachos 模拟带有 TLB 的页式内存管理。MMU 由函数 Translate 模拟。两个函数 ReadMem 和 WriteMem 在访问物理内存之前都要调用函数 Translate 将要访问的逻辑地址变换为物理地址。Translate 函数的代码可以在 machine/translate.cc 中找到。如果变换发生错误函数 Translate 会返回一个异常。Translate 函数的代码如下：

```
186   ExceptionType
187   Machine::Translate(int virtAddr, int* physAddr, int size,
bool writing)
188   {
189       int i;
190       unsigned int vpn, offset;
191       TranslationEntry *entry;
192       unsigned int pageFrame;
193
194       DEBUG('a', "\tTranslate 0x%x, %s: ", virtAddr, writing ?
"write" : "read");
195
```

```
196  // check for alignment errors
197      if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) &&
(virtAddr & 0x1))){
198          DEBUG('a', "alignment problem at %d, size %d!\n",
virtAddr, size);
199          return AddressErrorException;
200      }
201
202      // we must have either a TLB or a page table, but not both!
203      ASSERT(tlb == NULL || pageTable == NULL);
204      ASSERT(tlb != NULL || pageTable != NULL);
205
206  // calculate the virtual page number, and offset within the
page,
207  // from the virtual address
208      vpn = (unsigned) virtAddr / PageSize;
209      offset = (unsigned) virtAddr % PageSize;
210
211      if (tlb == NULL) {          // => page table => vpn is index
into table
212          if (vpn >= pageTableSize) {
213              DEBUG('a', "virtual page # %d too large for page
table size %d!\n",
214                              virtAddr, pageTableSize);
215              return AddressErrorException;
216          } else if (!pageTable[vpn].valid) {
217              DEBUG('a', "virtual page # %d too large for page
table size %d!\n",
218                              virtAddr, pageTableSize);
219              return PageFaultException;
220          }
221          entry = &pageTable[vpn];
222      } else {
223          for (entry = NULL, i = 0; i < TLBSize; i++)
224              if (tlb[i].valid && ((unsigned
int)tlb[i].virtualPage == vpn)) {
225                  entry = &tlb[i];                        //
FOUND!
```

```
226                 break;
227             }
228         if (entry == NULL) {                            //
not found
229             DEBUG('a', "*** no valid TLB entry found for this
virtual page!\n");
230             return PageFaultException;          // really,
this is a TLB fault,
231                                                 // the page
may be in memory,
232                                                 // but not
in the TLB
233         }
234     }
235
236     if (entry->readOnly && writing) {   // trying to write to
a read-only page
237         DEBUG('a', "%d mapped read-only at %d in TLB!\n",
virtAddr, i);
238         return ReadOnlyException;
239     }
240     pageFrame = entry->physicalPage;
241
242     // if the pageFrame is too big, there is something really
wrong!
243     // An invalid translation was loaded into the page table
or TLB.
244     if (pageFrame >= NumPhysPages) {
245         DEBUG('a', "*** frame %d > %d!\n", pageFrame,
NumPhysPages);
246         return BusErrorException;
247     }
248     entry->use = TRUE;          // set the use, dirty bits
249     if (writing)
250         entry->dirty = TRUE;
251     *physAddr = pageFrame * PageSize + offset;
252     ASSERT((*physAddr >= 0) && ((*physAddr + size) <=
MemorySize));
```

```
253       DEBUG('a', "phys addr = 0x%x\n", *physAddr);
254       return NoException;
255   }
```

当前的 MIPS 模拟器允许页表和快表两种地址变换机制。这个函数假设我们可用页式和快表两种变换方式，但当前的实现假设我们要么使用页表要么使用快表，但不能两者同时都用。为了模拟带有快表的真正的页式变换，这个函数需要改进.。可以在 Machine 类中看到，Machine 类已经提供了一个指向当前用户进程的指针和一个指向系统 TLB 的指针。

3.1.8 从内核线程到用户进程

以前我们在介绍 Thread 类时已看到 Nachos 用户进程是构建在 Nachos 的线程之上的。我们可以再回顾一下 Thread 类中有关用户进程的定义：

```
119   #ifdef USER_PROGRAM
120   // A thread running a user program actually has *two* sets of
CPU registers --
121   // one for its state while executing user code, one for its state
122   // while executing kernel code.
123
124       int userRegisters[NumTotalRegs];    // user-level CPU
register state
125
126     public:
127       void SaveUserState();               // save user-level
register state
128       void RestoreUserState();            // restore
user-level register state
129
130       AddrSpace *space;                   // User code this
thread is running.
131   #endif
132   };
```

以上行显示了用户进程用于保存用户寄存器的数组和为了生成用户进程

所添加给一个内核线程的用户地址空间指针。

当你在 userprog 目录中编译时，其中 Makefile 文件定义了 USER_PROGRAM 标志。此时以上 124-132 行的代码将会被编译进内核中去。

定义在../userprog/progtest.cc 文件中的函数 StartProcess 说明了如何由一个内核线程构造并启动一个用户进程的过程：

```
23  void
24  StartProcess(char *filename)
25  {
26      OpenFile *executable = fileSystem->Open(filename);
27      AddrSpace *space;
28
29      if (executable == NULL) {
30          printf("Unable to open file %s\n", filename);
31          return;
32      }
33      space = new AddrSpace(executable);
34      currentThread->space = space;
35
36      delete executable;                  // close file
37
38      space->InitRegisters();             // set the initial
register values
39      space->RestoreState();              // load page table
register
40
41      machine->Run();                     // jump to the user
progam
42      ASSERT(FALSE);                      // machine->Run never
returns;
43                                          // the address space
exits
44                                          // by doing the
syscall "exit"
45  }
```

StartProcess 的参数是在 Nachos 系统启动时由命令行参数传入的 Noff 格

式的用户可执行文件名字符串。文件系统调用函数 Open 根据它打开要装入的文件，AddrSpace 再使用这个打开的文件建立和初始化好一个用户进程空间并且返回一个指向该进程空间的指针。当调用 machine->Run()函数后，当前线程就变成了运行在 MIPS 模拟机上的用户进程，当然之后的机器工作状态就由系统的核心态转变为用户态。

3.1.9 系统调用的实现

在本章中，我们看一下 Nachos 的系统调用和它的实现。
系统调用是用户程序和操作系统内核的接口。用户程序从系统调用函数取得系统服务。
当 CPU 控制从用户程序切换到系统态时，CPU 的工作方式由用户态改变为系统态。而当内核完成系统调用功能时，CPU 工作状态又从系统态改变回用户态并且将控制再次返回给用户程序。两种不同的 CPU 工作状态提供了操作系统基本的保护方式。
我们已经实验过 Nachos 的线程管理和同步，在前一节中，我们又饰演了一个用户程序是怎样被格式化为一个可运行于 Nachos 模拟的 MIPS 虚拟机上的进程的。现在我们实验 Nachos 系统调用的实现。

3.1.10 Nachos 用户程序的机构

在 Nachos 系统中用户程序是怎样编译的呢？实际这些 C 语言编写的用户程序在由 gcc MIPS 交叉编译后都在前面连接上一个由 MIPS 汇编程序 start.s 生成的叫 start.o 的目标模块。实际上 start 是用户程序真正的启动入口，由它来调用 C 程序的 main 函数。所以不要求用户编程时一定要把 main 函数作为第一个函数。这个汇编程序也为 Nachos 系统调用提供了一个汇编语言的存根（stub）。以下是汇编程序 start.s 的汇编清单：

```
9  #define IN_ASM
10  #include "syscall.h"
11
12          .text
13          .align  2
14
15  /*
-------------------------------------------------------------
```

```
16   * __start
17   *      Initialize running a C program, by calling "main".
18   *
19   *      NOTE: This has to be first, so that it gets loaded at
location 0.
20   *      The Nachos kernel always starts a program by jumping to
location 0.
21   *
-----------------------------------------------------------
22   */
23
24          .globl __start
25          .ent    __start
26   __start:
27          jal     main
28          move    $4,$0
29          jal     Exit     /* if we return from main, exit(0) */
30          .end __start
31
32   /*
-----------------------------------------------------------
33   * System call stubs:
34   *      Assembly language assist to make system calls to the
Nachos kernel.
35   *      There is one stub per system call, that places the code
for the
36   *      system call into register r2, and leaves the arguments
to the
37   *      system call alone (in other words, arg1 is in r4, arg2
is
38   *      in r5, arg3 is in r6, arg4 is in r7)
39   *
40   *      The return value is in r2. This follows the standard C
calling
41   *      convention on the MIPS.
42   *
-----------------------------------------------------------
43   */
```

```
44
45          .globl Halt
46          .ent    Halt
47  Halt:
48          addiu $2,$0,SC_Halt
49          syscall
50          j       $31
51          .end Halt
52
53          .globl Exit
54          .ent    Exit
55  Exit:
56          addiu $2,$0,SC_Exit
57          syscall
58          j       $31
59          .end Exit
60
61          .globl Exec
62          .ent    Exec
63  Exec:
64          addiu $2,$0,SC_Exec
65          syscall
66          j       $31
67          .end Exec
68
69          .globl Join
70          .ent    Join
71  Join:
72          addiu $2,$0,SC_Join
73          syscall
74          j       $31
75          .end Join
76
77          .globl Create
78          .ent    Create
79  Create:
80          addiu $2,$0,SC_Create
81          syscall
```

```
82          j       $31
83          .end Create
84
85          .globl Open
86          .ent    Open
87  Open:
88          addiu $2,$0,SC_Open
89          syscall
90          j       $31
91          .end Open
92
93          .globl Read
94          .ent    Read
95  Read:
96          addiu $2,$0,SC_Read
97          syscall
98          j       $31
99          .end Read
100
101          .globl Write
102          .ent    Write
103  Write:
104          addiu $2,$0,SC_Write
105          syscall
106          j       $31
107          .end Write
108
109          .globl Close
110          .ent    Close
111  Close:
112          addiu $2,$0,SC_Close
113          syscall
114          j       $31
115          .end Close
116
117          .globl Fork
118          .ent    Fork
119  Fork:
```

```
120         addiu $2,$0,SC_Fork
121         syscall
122         j       $31
123         .end Fork
124
125         .globl Yield
126         .ent    Yield
127 Yield:
128         addiu $2,$0,SC_Yield
129         syscall
130         j       $31
131         .end Yield
132
133 /* dummy function to keep gcc happy */
134         .globl  __main
135         .ent    __main
136 __main:
137         j       $31
138         .end    __main
```

例如 C 程序 halt.c 被编译为 haltt.o，同时 start.s 也被汇编为 start.o。之后两个目标模块被连接成可执行的 Coff 格式的可执行文件，最后这个 Coff 文件又被转换为 Noff 格式的 Nachos 可执行文件。

我们看一下 halt.c 和它生成的汇编语言 halt.s 的清单就可以比较清楚 start.s 和 helt.s 之间的关系以及系统调用的定义和执行的过程了。halt.c 程序非常简单，它只有一个 main 函数且仅执行了一句 Nachos 的停机系统调用 Halt()：

```
13 #include "syscall.h"
14
15 int
16 main()
17 {
18         int i,j,k;
19         k=3;
20         i=2;
21         j = i-1;
22         k = i - j+k;
```

```
23          Halt();
24      /* not reached */
25  }
```

由 halt.c 生成的汇编语言 halt.s 的清单

```
1           .file   1 "halt.c"
2   gcc2_compiled.:
3   __gnu_compiled_c:
4           .text
5           .align  2
6           .globl  main
7           .ent    main
8   main:
9           .frame  $fp,40,$31              # vars= 16, regs= 2/0,
args= 16, extra= 0
10          .mask   0xc0000000,-4
11          .fmask  0x00000000,0
12          subu    $sp,$sp,40
13          sw      $31,36($sp)
14          sw      $fp,32($sp)
15          move    $fp,$sp
16          jal     __main
17          li      $2,3                    # 0x00000003
18          sw      $2,24($fp)
19          li      $2,2                    # 0x00000002
20          sw      $2,16($fp)
21          lw      $2,16($fp)
22          addu    $3,$2,-1
23          sw      $3,20($fp)
24          lw      $2,16($fp)
25          lw      $3,20($fp)
26          subu    $2,$2,$3
27          lw      $3,24($fp)
28          addu    $2,$3,$2
29          sw      $2,24($fp)
30          jal     Halt
31  $L1:
32          move    $sp,$fp
```

```
33          lw      $31,36($sp)
34          lw      $fp,32($sp)
35          addu    $sp,$sp,40
36          j       $31
37          .end    main
```

可以看到 main 做的第一件事情就是栈顶指针减去一个帧长度以建立一个新的栈帧。然后在这一帧中保存返回地址寄存器($31)和帧指针寄存器($fp)。（line20-30）。

主例程的主体简单的调用汇编例程 Halt(jal Halt)。这个 Halt 定义在 start.s 的 47-54 行。(__main 子例程是一个钩子，gcc 产生这个钩子程序是为了在 main 函数主体开始之前提供一个做一些其它事情的机会。它定义在 start.s 的 136-138 行，此时它什么事情也没做，是个虚拟例程)。如果主函数正常退出， 27-31 行上的指令将取消这一帧。这几行恰为 20-23 行上的逆操作。最后 CPU 控制将由 31 行上的”j $31”指令汗回到调用这个主函数的例程中去。在此例中，如果控制到达这条指令，它将返回到 start.s 的第 28 行。

3.1.11 系统调用接口

现在我们已经看到了系统调用 Halt 的汇编代码，它没有携带任何要传送到内核中的参数。如果系统调用携带参数，编译又怎样来生成传送参数的指令呢？

所有 Nachos 系统调用的接口原型都定义在文件 userprog/syscall.h 中。当编译用户程序时编译器会括入这个文件并取得这些系统调用接口原型的信息（这就是为什么当你编译 halt.c 是 Makefile 文件中必须加入-I../userprog 定义）。

当前， Nachos 支持 11 个系统调用。这些系统调用接口原型在 userprog/syscall.h 文件的 45-125 行上。

```
18  /* system call codes -- used by the stubs to tell the kernel which
system call
19   * is being asked for
20   */
21  #define SC_Halt         0
22  #define SC_Exit         1
23  #define SC_Exec         2
24  #define SC_Join         3
25  #define SC_Create       4
```

```
26  #define SC_Open         5
27  #define SC_Read         6
28  #define SC_Write        7
29  #define SC_Close        8
30  #define SC_Fork         9
31  #define SC_Yield        10
32
33  #ifndef IN_ASM
34
35  /* The system call interface.  These are the operations the
Nachos
36   * kernel needs to support, to be able to run user programs.
37   *
38   * Each of these is invoked by a user program by simply calling
the
39   * procedure; an assembly language stub stuffs the system call
code
40   * into a register, and traps to the kernel.  The kernel
procedures
41   * are then invoked in the Nachos kernel, after appropriate error
checking,
42   * from the system call entry point in exception.cc.
43   */
44
45  /* Stop Nachos, and print out performance stats */
46  void Halt();
47
48
49  /* Address space control operations: Exit, Exec, and Join */
50
51  /* This user program is done (status = 0 means exited normally).
*/
52  void Exit(int status);
53
54  /* A unique identifier for an executing user program (address
space) */
55  typedef int SpaceId;
56
```

```
57  /* Run the executable, stored in the Nachos file "name", and
return the
58   * address space identifier
59   */
60  SpaceId Exec(char *name);
61
62  /* Only return once the the user program "id" has finished.
63   * Return the exit status.
64   */
65  int Join(SpaceId id);
```

……

对应的系统调用的汇编语言存根在 test/start.s 文件中的 45-131 行。如果你要添加你自己的系统调用，就应当首先在 syscall.h 和 start.s 中声明你的系统调用原型和存根。

当一个系统调用由一个用户进程发出时，由汇编语言编写的对应于存根的程序就被执行。然后，这个存根程序会由执行一个系统调用指令而引发一个异常或自陷。

在 start.s 中的这些系统调用的接口程序代码都是一样的。即：

- 将对应的系统调用的编码送$2 寄存器
- 执行系统调用指令 SYSCALL 且
- 返回到用户程序

3.1.12 异常和自陷

模拟 MIPS 计算机的异常和自陷管理的是 Machine 类中的函数 RaiseException(ExceptionType which, int badVAddr)。其中的第一个参数 which 是一个 ExceptionType 枚举类型的变量。ExceptionType 类型的定义也在 machine/machine.h 文件中：

```
39   enum ExceptionType { NoException,           // Everything ok!
40                        SyscallException,      // A program
executed a system call.
41                        PageFaultException,    // No valid
translation found
```

```
42                      ReadOnlyException,      // Write
attempted to page marked
43                                              // "read-only"
44                      BusErrorException,      // Translation
resulted in an
45                                              // invalid
physical address
46                      AddressErrorException, // Unaligned
reference or one that
47                                              // was beyond the
end of the
48                                              // address space
49                      OverflowException,      // Integer
overflow in add or sub.
50                      IllegalInstrException, // Unimplemented
or reserved instr.
51
52                      NumExceptionTypes
53  };
```

系统调用是这些异常中的一个。MIPS 计算机的”SYSCALL”指令在 Nachos 中是由 machine/mipssim.cc 中 534-536 行上的通过发系统调用异常模拟的：

注意在系统调用异常处理之后的下一条语句是一条 return 返回语句，而不是 break 语句。这一点很重要，return 语句不会使程序计数器 PC 向前推进，从而在异常处理之后同一条指令将会再次被启动。

函数 RaiseException(ExceptionType which, **int** badVAddr) 的代码在 machine/nachine.cc 文件中：

```
100  void
101  Machine::RaiseException(ExceptionType which, int badVAddr)
102  {
103      DEBUG('m', "Exception: %s\n", exceptionNames[which]);
104
105  //  ASSERT(interrupt->getStatus() == UserMode);
106      registers[BadVAddrReg] = badVAddr;
107      DelayedLoad(0, 0);                  // finish anything in
progress
```

```
108       interrupt->setStatus(SystemMode);
109       ExceptionHandler(which);            // interrupts are
enabled at this point
110       interrupt->setStatus(UserMode);
111   }
```

这个函数模拟硬件的动作，切换到系统态并且在异常处理完成后返回到用户态。108-110 行代表了 Nachos 内核和用户程序的一个界面接口。110 行上的 ExceptionHandler(which)函数调用模拟硬件的动作发发一个异常中断到对应的异常处理程序。这个函数定在 userprog/execution.cc 中：

```
51  void
52  ExceptionHandler(ExceptionType which)
53  {
54      int type = machine->ReadRegister(2);
55
56      if ((which == SyscallException) && (type == SC_Halt)) {
57          DEBUG('a', "Shutdown, initiated by user program.\n");
58          interrupt->Halt();
59      } else {
60          printf("Unexpected user mode exception %d %d\n", which,
type);
61          ASSERT(FALSE);
62      }
63  }
```

在此时，Nachos 仅能处理带有 SC__HALT 代码的系统调用。寄存器$2 包含系统调用代码（如果异常是一个系统调用异常）且寄存器$4-$7 包含着当系统调用开始处理时的前 4 个参数。系统调用的返回值，如果有，都将返回$2。对于系统调用 Halt 的异常处理只是简单的模拟了 Interrupt 类指向的中断函数 Halt()。

3.1.13 虚拟内存

前面我们讨论的用户进程需要整个地址空间都进入物理内存后才能开始运行。这种进程全部进入内存后再运行的策略是很受限制和浪费内存空间的。实际上多数程序的工作方式是符合”局部性原理”的，即当前要执行的

程序的和当前要访问的数据往往是集中在某些内存页面上，而当前执行不到程序和访问不到的数据完全可以先放在第二存储器中，等到用到时再从第二存储器中装入物理内存。

虚拟内存技术就是解决程序部分装入物理内存也能执行的技术。由于允许进程部分驻留物理内存，所以一个进程的逻辑空间可以远远大于分配给进程工作的物理空间。

3.1.14 请求式分页技术

虚拟内存技术可以在带有分页式内存管理的系统中，通过采用请求式分页存储管理技术实现。请求式分页内存管理技术的意思是，装入物理内存的内存逻辑页仅在访问到它时才被分配到物理内存帧。其关键的技术是，当请求的页不在物理内存时使用缺页异常或自陷进入内核，由内核缺页异常处理程序从外存将该页装入一空闲内存帧中，如果无空闲帧，将选择已在物理内存中的一页将其置换。在这一异常被处理完成之后，发出对该内存访问的同一条指令将再次被执行。

Nachos 提供了实现虚拟内存技术的基本机制，你可以通过这些基本机制来实现 Nachos 中的虚拟内存。其中缺页异常是由 MIPS 机模拟函数 translate 产生的。这个函数是由函数 ReadMem 和 WriteMem 调用的。让我们来看一下在文件 machine/translate.cc 中的 ReadMem 函数的有关执行过程：

```
87  bool
88  Machine::ReadMem(int addr, int size, int *value)
89  {
90      int data;
91      ExceptionType exception;
92      int physicalAddress;
93
94      DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);
95
96      exception = Translate(addr, &physicalAddress, size,
FALSE);
97      if (exception != NoException) {
98          machine->RaiseException(exception, addr);
99          return FALSE;
100      }
```

……

参数 addr 是一条指令发出的要寻找的逻辑地址。这个地址通过调用函数 translate 被转换为物理地址。如果发生缺页异常，则会通过调用函数 RaiseException 将该异常分派到对应的异常处理函数处理。如果你想实现 Nachos 的虚拟内存，这儿就是你的设计起点。注意函数 ReadMem 在完成缺页异常处理之后将返回一个 FALSE(97-99 行)，这一点很重要，它将使调用 ReadMem 的指令重新执行。

在文件 machine/mips.cc 中定义的 MIPS 模拟器在执行以下操作时都有可能引发缺页异常处理，调用函数 RaiseException：

- 取一条指令(102)行或
- 执行 234、252、274、288、319、484、514 行上的指令

所有这些情况，如果相关函数返回值为 FALSE，在函数 Run 的 for 循环中的函数 OneInstuction 就不会向前推进程序计数器，而是立即返回被再次执行。即 Nachos 中 MIPS 机模拟器正确的模拟了请求式分页算法的第六步-Restart Instruction。

3.1.15 页置换

页置换是解决当物理内存已满且有一新页需要装入时怎样选择被淘汰页的问题。常用的页置换算法有：

- FIFO 算法
- Optimal 算法
- LRU 算法

要做好这方面的设计，你还需要明白 Belady's 异常和栈算法。除了 Optimal 算法外，应当搞清 FIFO、LRU 算法是怎样实现的。这些算法的实现经常是考虑化肥和实用而将 FIFO、LUR 算法结合起来的一些近似算法。页缓冲技术可以用来改进置换算法的执行效率。

Nachos 对于页置换的硬件模拟的支持反映在 machine/translate.h 文件中定义的页表结构 TranslationEntry 中：

```
30  class TranslationEntry {
31    public:
32      int virtualPage;    // The page number in virtual memory.
33      int physicalPage;   // The page number in real memory
(relative to the
34                          //  start of "mainMemory"
```

```
35      bool valid;         // If this bit is set, the translation
is ignored.
36                          // (In other words, the entry hasn't
been initialized.)
37      bool readOnly;      // If this bit is set, the user program
is not allowed
38                          // to modify the contents of the page.
39      bool use;           // This bit is set by the hardware every
time the
40                          // page is referenced or modified.
41      bool dirty;         // This bit is set by the hardware every
time the
42                          // page is modified.
43  };
```

变量 valid 和 readonly 对应 MMU 硬件的 valid 和 read-only 位，是用于内存管理和保护的。变量 use 和 dirty 对应硬件的 use 位和 dirty 位，是用于虚拟内存页置换的。考虑一下你应当怎样利用一下这两个变量为 Nachos 的虚拟内存实现页置换。

3.1.16 帧的分配

在 Nachos 的 MIPS 模拟器中的物理内存是通过一个字节型的数组模拟的。在文件 machine/machine.cc 中定义的 Machine 类的构造函数初始化了这个基本内存(61-63 行)，其大小为 Memeorysiae 字节或 NumphysPages 帧。

```
55  Machine::Machine(bool debug)
56  {
57      int i;
58
59      for (i = 0; i < NumTotalRegs; i++)
60          registers[i] = 0;
61      mainMemory = new char[MemorySize];
62      for (i = 0; i < MemorySize; i++)
63          mainMemory[i] = 0;
64  #ifdef USE_TLB
65      tlb = new TranslationEntry[TLBSize];
66      for (i = 0; i < TLBSize; i++)
```

```
67          tlb[i].valid = FALSE;
68      pageTable = NULL;
69  #else   // use linear page table
70      tlb = NULL;
71      pageTable = NULL;
72  #endif
73
74      singleStep = debug;
75      CheckEndian();
76  }
```

NumPhysPages()（在文件 machine.h 中）当前定义的大小是：

```
35  #define NumPhysPages     32
```

当你为 Nachos 实现一个虚拟内存时，你需要考虑：

- 每个进程最小的页帧数是多少？
- 使用什么样的帧分配算法？
- 提供的帧算法是全局的还是局部的？

3.2 内存管理实验

正如我们在本章实验分析部分所说明的，Nachos 使用 MIPS 模拟机来运行用户程序。在../test 目录中，Nachos 系统已经为我们准备好了几个 C 语言用户程序。Nachos 用户程序的二进制文件通过 gcc MIPS 编译后生成 Coff 格式的文件，再由 coff2noff 命令转换为 Noff 格式。

能够运行 Nachos 用户程序的 Nachos 内核在../userprog 目录中。我们可以在../userprog 目录中进行我们的内存管理有关的内核设计，然后利用..test/目录中的 C 语言用户程序来验证我们设计的新内核的内存管理功能。

3.2.1 实验目的

- 实现 Nachos 中用户程序的装入和内存页式转换机制。
- 实现带有 TLB 机制的的内存管理机制。
- 实现几种基本的系统调用功能。
- 实现虚拟内存管理机制
- 通过多用户程序并发执行验证以上你的设计是否成功的实现了。

3.2.2 Nachos 用户可执行文件的生成和执行

首先，我们应当研究一下../test 目录中 Makefile 文件的内容，搞清 Nachos 用户可执行文件是怎样生成的。../test 中现有 5 个 C 语言用户源程序，可以通过 make 命令一次性编译连接生成它们的可执行文件和其在该目录中的符号链接。

然后，我们可以进入../userprog 目录，使用 make 命令生成带有基本内存管理功能的 Nachos 内核。现在我们就可以使用新的内核执行用户程序了。例如，为了执行../test 目录中的 halt.noff 程序，可以输入命令：

$./nachos –x ../test/halt.noff

此时，显示的输出结果仍然同../threads 目录中 nachos 的执行结果相同，为了能观察 MIPS 模拟器执行的每条用户指令，你可以在以上命令行中增加选项参数-d 。也可以使用 gdb 启动以上命令跟踪调试用户程序的启动和执行情况。

为了能够了解 Nachos 中多用户程序驻留内存的情况，可以在 AssSpace

类中增加以下打印成员函数 Print：

```
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"
);
}
```

在装入用户程序后你可以利用这个函数显示出该用户程序在内存中页分配的情况。例如最小的用户程序 halt.c 会占用 10 页的内存空间。如果你想验证系统能否分配更多的内存页，可以在用户程序中说明一些较大的数组。例如，修改 halt.c，在其中说明一个静态数组 a：

```
#include "syscall.h"
static int a[40];
int
main()
{
Halt();
/* not reached */
}
```

再次执行../test 目录中的 make，并再次进入../userprog 目录，使用 make 命令生成带有输出用户程序页分配信息的 Nachos 内核。输入命令：

```
$./nachos –x ../test/halt.noff
……
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
========================================
  ……
```

现在可以看到用户程序的页表情况，修改后的 halt.c 分配的内存增加到了 12 页。

若想查看用户程序汇编指令清单，了解函数栈帧的建立与删除过程以及汇编指令与 C 语句的对应关系，可以使用命令：

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

这个命令将会生成一个 halt.c 的汇编语言程序 start.s。

3.2.3 用户地址空间的扩充

在了解了 Nachos 装入并执行单个用户进程的情况后，我们就需要进一步完成用户内存空间的扩充以便多用户程序同时驻留内存，进而使多用户进程并发执行。

要在 Nachos 中实现多用户程序同时驻留内存并发执行，首先涉及到 Nachos 的两个系统调用：Exec()和 Exit()。这两个系统调用也是构造父子进程并发执行的基础。假设我们有以下两个用户程序：../test/exec.c 和../test/halt.c

../test/halt.c

```
1  #include "syscall.h"
2  int
3  main()
4  {
5  Halt()
6  }
```

../test/exec.c

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

```
2  int
3  main()
4  {
5  SpacId pid;
6  pid = Exec("../test/halt.noff");
7  Halt()
8  }
```

在文件../test/exec.c 第 5 行上的语句 Exec 是一条 Nachos 的系统功能调用，它的功能为装入并执行以其参数为名的可执行文件，即创建一个新的用户进程。假设我们先执行../test/exec.noff 程序，则../test/exec.noff 会在它还没有执行结束时又装入并执行另外一个程序 halt.noff，并与它同时驻留内存。当前的 Nachos 能实现这一功能吗？

首先看一下当前的 Nachos 有关用户内存页表的初始化功能，即 AddrSpace.cc 中的有关片段：

```
86      pageTable = new TranslationEntry[numPages];
87      for (i = 0; i < numPages; i++) {
88          pageTable[i].virtualPage = i;   // for now, virtual
page # = phys page #
89          pageTable[i].physicalPage = i;
90          pageTable[i].valid = TRUE;
91          pageTable[i].use = FALSE;
92          pageTable[i].dirty = FALSE;
93          pageTable[i].readOnly = FALSE;  // if the code segment
was entirely on
94                                          // a separate page, we
could set its
95                                          // pages to be
read-only
96      }
```

注意 89 行物理帧的分配总是循环变量 i 的值。可以想象，当两个程序同时驻留内存时后一个程序会装入到前一个程序的物理地址中，从而将先前已装入的程序覆盖。可见基本的 Nachos 并不具有多个程序同时驻留内存的功能。

为了实现多个程序同时驻留内存的功能需要我们改进 Nachos 的内存分配算法的设计。一个比较简单的设计就是利用 Nachos 在../userprog/bitmap.h 中文件定义的 Bitmap 类。利用 bitmap 记录和申请内存物理帧，使不同的

程序装入到不同的物理空间中去。请仔细阅读一下 bitmap.h 和 bitmap.cc，了解 Bitmap 类的定义和实现，以便完成我们的设计。

3.2.4 系统调用 Exec、Join 和 Exit 的实现

在我们完成了多个程序同时驻留内存的内存分配算法后，我们就应当考虑用户父子进程并发执行的问题了。为了完成这一功能首先可以准备一个作为父进程的用户程序 exec.c:

```
1  #include "syscall.h"
2  int
3  main()
4  {
5  SpacId pid;
6  pid = Exec("../test/halt.noff");
7  Halt()
8  }
```

为了能和已有的用户 C 程序一起生成可执行文件，可以修改../test/Makefile 文件，将 exec 加入到 targets 定义中：targets = halt shell matmult sort exec。在../test 中重新 make 生成 exec.noff 可执行文件。

在 exec.c 中为了生成子进程 halt.noff，使用了 Nachos 的系统调用 Exec。它带有一个字符串参数，是一个可执行文件名。在发生系统调用时系统内核怎样得到这个参数并根据它建立子进程呢？这需要我们看一下对应于 exec.c 的汇编代码，了解一下 MIPS 机指令系统对于参数传递是如何安排的：

```
1          .file   1 "exec.c"
2  gcc2_compiled.:
3  __gnu_compiled_c:
4          .rdata
5          .align  2
6  $LC0:
7          .ascii  "../test/halt.noff\000"
8          .text
9          .align  2
10          .globl  main
11          .ent    main
12  main:
```

```
13          .frame  $fp,32,$31              # vars= 8, regs= 2/0,
args= 16, extra= 0
14          .mask   0xc0000000,-4
15          .fmask  0x00000000,0
16          subu    $sp,$sp,32
17          sw      $31,28($sp)
18          sw      $fp,24($sp)
19          move    $fp,$sp
20          jal     __main
21          la      $4,$LC0
22          jal     Exec
23          sw      $2,16($fp)
24          jal     Halt
25  $L1:
26          move    $sp,$fp
27          lw      $31,28($sp)
28          lw      $fp,24($sp)
29          addu    $sp,$sp,32
30          j       $31
31          .end    main
```

可以看到要传递的文件名字符串被汇编安排在逻辑地址$LC0 处，在转向系统调用之前在 21 行处这个逻辑地址被放入 4 号寄存器（la $4,$LC0）。

下面我们就可以考虑怎样实现 Exec 系统调用的异常处理函数 Exec 了。由于异常属于一种中断处理，因此通常把这类处理函数都封装到 Interrupt 类中，作为 Interrupt 类的成员函数。

对于异常处理函数 Exec 的编码可以参考../userprog/progtest.cc 文件中 StartProcess 函数。主要的不同处在于 StartProcess 中要打开的文件名是从命令行传递过来的，而 Exec 要打开的文件名是由$4 寄存器传递过来的。

另外系统调用异常处理的接口函数 ExceptionHandler 中当前仅有 Halt 系统调用一项的入口，在实现了相应的处理函数后就应当在这里声明它，以便响应对应的系统调用请求。

还要注意在文件../machine/mipssim.cc 中系统调用模拟指令的操作是以 return 返回的：

```
534         case OP_SYSCALL:
535           RaiseException(SyscallException, 0);
536           return;
```

这意味着在执行完系统调用后是否令程序计数器向前推进的工作交给了对应的系统调用处理函数去决定。所以你还应当准备一个函数 AdvancePC 以便当系统调用成功后向前推进程序计数器：

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

Exec 系统调用的返回值是一个 SpaceId 类型的进程标识，它将作为 Join 系统调用的参数。Join 系统调用可用于返回以 SpaceId 为标识的进程结束时的退出状态。有关 SpaceId 的值需要解决两个问题：

- 怎样在内核中产生这个值
- 怎样在内核中记录这个值，以便 Join 系统调用通过这个值找到对应的用户进程。

有关用户进程控制的系统调用还有 Exit。Exit 的参数是一个整数，它代表了用户进程退出时的状态（通常约定：0 值代表正常退出，非 0 值代表出错代码）。一些子进程可能在其父进程调用 Join 来返回其退出状态之前就已经退出了。因此，这个退出值应当在内核中保存起来，以便在子进程退出后其父进程仍可通过 Join 系统调用取到子进程的退出状态。

第 4 章 操作系统的文件系统设计

文件系统是操作系统管理外部设备中信息的软件集合。通过文件系统，操作系统隐藏了不同设备物理控制的复杂细节，提供了简单一致的信息访问方法，保障了系统信息的安全。本章通过对 Nachos 文件系统各个层次上类的分析和实验来实现文件系统的设计和构造。

4.1 文件系统设计分析

本章主要讨论的主题是：

- 文件系统的结构
- 文件空间分配的算法
- 自由空间的管理
- 文件系统的执行效率
- 文件系统的修复

有关这些主题概念和技术的实现我们将通过 Nachos 文件系统来说明和实现。

文件系统的组织结构

经典的文件系统包括 5 个层次：

- 逻辑文件系统层
- 文件模块组织层
- 基本文件系统层
- I/O 控制系统层
- 物理设备

4.1.1 Nachos 文件系统的组织结构

Nachos 文件系统使用 7 个模块实现了文件系统的 5 个层次的基本功能，它们是：

- Filesys
- Directory
- OpenFile

- FileHeader
- BitMap
- SynchDisk
- Disk

这些模块之间的关系见图 4.1，它们和经典文件系统中文件层次的对应关系见图 4.1 中虚线部分。

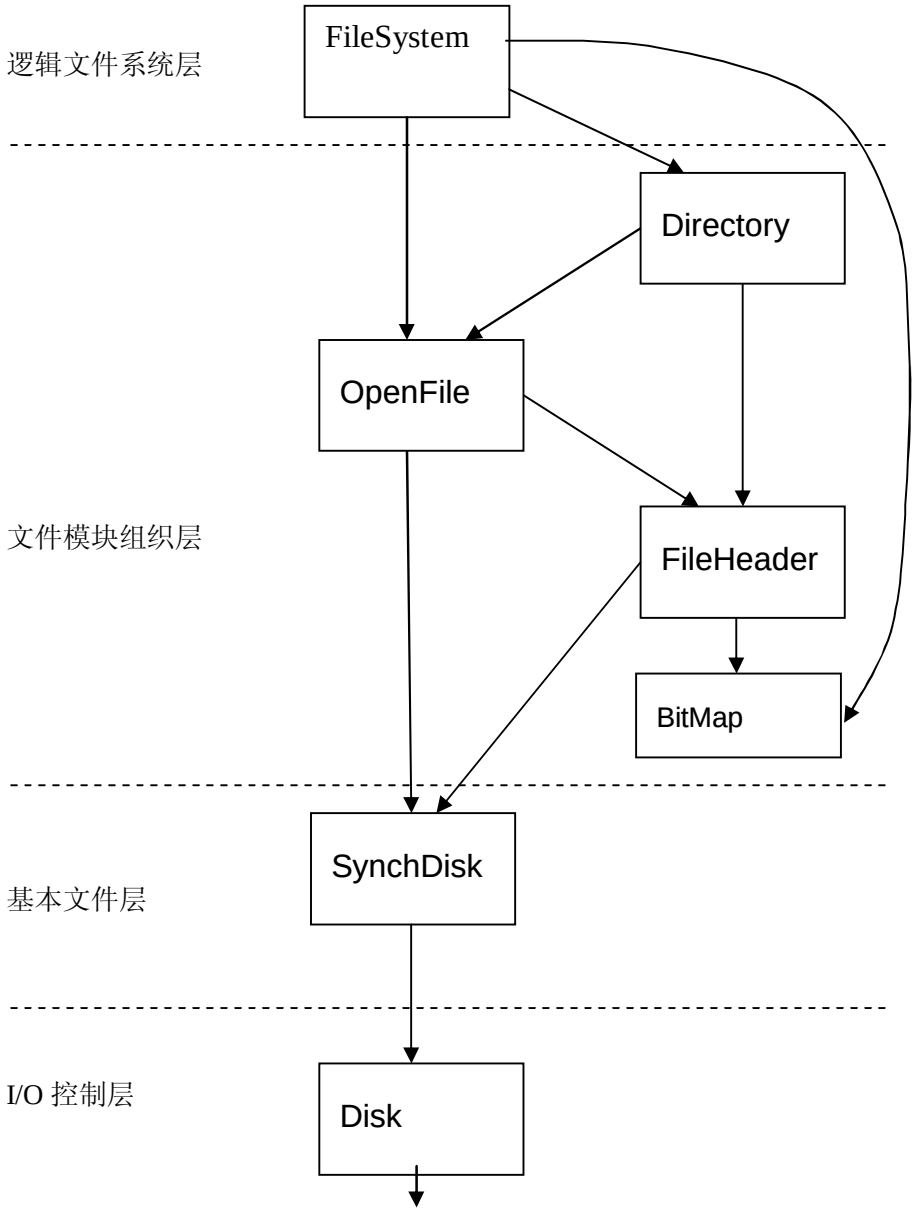

图 4.1　Nachos 文件系统结构层次图

4.1.2 I/O 控制和设备

有 3 种 I/O 方式可以控制 I/O 设备和设备驱动来完成 I/O:

- 轮询 I/O　Polling I/O
- 中断 I/O　Interupt-Driver I/O
- 直接内存访问　DMA I/O

在当前的 Nacho 文件系统中磁盘的 I/O 中断驱动是由两个模块模拟的：Disk、SynchDisk。

Disk 模拟了设备控制和磁盘的自身驱动，SynchDisk 模拟了系统内核 I/O 系统的一部分。模拟硬盘驱动的源代码文件是：machine/disk.h 和 machine/disk.cc。

让我们来看一下模拟磁盘的物理参数：

```
49  #define SectorSize              128     // number of bytes per
disk sector
50  #define SectorsPerTrack         32      // number of sectors
per disk track
51  #define NumTracks               32      // number of tracks
per disk
52  #define NumSectors              (SectorsPerTrack * NumTracks)
53                                          // total # of sectors
per disk
```

可以看出该磁盘预定义的容量 = SectorSize* SectorsPerTrack* NumTracks=128KB

类对象 Disk 模拟了硬磁盘及其操作，它定义在文件 machine/disk.h 中：

```
55  class Disk {
56    public:
57      Disk(char* name, VoidFunctionPtr callWhenDone, _int
callArg);
58                                          // Create a simulated
disk.
59                                          // Invoke
(*callWhenDone)(callArg)
```

```
60                                          // every time a
request completes.
61      ~Disk();                            // Deallocate the
disk.
62
63      void ReadRequest(int sectorNumber, char* data);
64                                          // Read/write an
single disk sector.
65                                          // These routines
send a request to
66                                          // the disk and return
immediately.
67                                          // Only one request
allowed at a time!
68      void WriteRequest(int sectorNumber, char* data);
69
70      void HandleInterrupt();             // Interrupt handler,
invoked when
71                                          // disk request
finishes.
72
73      int ComputeLatency(int newSector, bool writing);
74                                          // Return how long a
request to
75                                          // newSector will
take:
76                                          // (seek + rotational
delay + transfer)
77
78    private:
79      int fileno;                         // UNIX file number
for simulated disk
80      VoidFunctionPtr handler;            // Interrupt handler,
to be invoked
81                                          // when any disk
request finishes
82      _int handlerArg;                    // Argument to
interrupt handler
```

```
83      bool active;                        // Is a disk operation
in progress?
84      int lastSector;                     // The previous disk
request
85      int bufferInit;                     // When the track
buffer started
86                                          // being loaded
87
88      int TimeToSeek(int newSector, int *rotate); // time to get
to the new track
89      int ModuloDiff(int to, int from);        // # sectors
between to and from
90      void UpdateLast(int newSector);
91  };
92
93  #endif // DISK_H
```

在硬盘由 Disk(**char*** name, VoidFunctionPtr callWhenDone, _int callArg) 构造时，私有的类成员函数 handler 被设置为指向基本文件层磁盘管理程序 callWhenDone，其参数 callArg 也被保存在私有类成员变量 handlerArg 中。这样当一次硬盘操作完成时可以回调该函数完成硬盘中断处理。
构造硬盘的源代码在文件 disk.cc 的 43-68 行：

```
43  Disk::Disk(char* name, VoidFunctionPtr callWhenDone, _int
callArg)
 44  {
 45      int magicNum;
 46      int tmp = 0;
 47
 48      DEBUG('d', "Initializing the disk, 0x%x 0x%x\n",
callWhenDone, callArg);
 49      handler = callWhenDone;
 50      handlerArg = callArg;
 51      lastSector = 0;
 52      bufferInit = 0;
 53
 54      fileno = OpenForReadWrite(name, FALSE);
```

```
 55      if (fileno >= 0) {                  // file exists, check
magic number
 56          Read(fileno, (char *) &magicNum, MagicSize);
 57          ASSERT(magicNum == MagicNumber);
 58      } else {                            // file doesn't exist,
create it
 59          fileno = OpenForWrite(name);
 60          magicNum = MagicNumber;
 61          WriteFile(fileno, (char *) &magicNum, MagicSize); //
write magic number
 62
 63          // need to write at end of file, so that reads will not
return EOF
 64          Lseek(fileno, DiskSize - sizeof(int), 0);
 65          WriteFile(fileno, (char *)&tmp, sizeof(int));
 66      }
 67      active = FALSE;
 68  }
```

注意函数 OpenForReadWrite(name, FALSE)是怎样打开名为 DISK 的文件的，这个文件用来作为 Nachos 文件系统的裸盘。

为硬盘提供的 I/O 驱动函数是：ReadRequest(**int** sectorNumber, **char*** data)和 WriteRequest(**int** sectorNumber, **char*** data)。这两个函数模拟设备驱动程序控制硬盘的命令。它们的源代码在文件 disk.cc 中。在此主要分析一下 ReadRequest 的实现实现，WriteRequest 的实现与其基本类似：

```
115  void
116  Disk::ReadRequest(int sectorNumber, char* data)
117  {
118      int ticks = ComputeLatency(sectorNumber, FALSE);
119
120      ASSERT(!active);                    // only one
request at a time
121      ASSERT((sectorNumber >= 0) && (sectorNumber <
NumSectors));
122
123      DEBUG('d', "Reading from sector %d\n", sectorNumber);
```

```
124     Lseek(fileno, SectorSize * sectorNumber + MagicSize, 0);
125     Read(fileno, data, SectorSize);
126     if (DebugIsEnabled('d'))
127         PrintSector(FALSE, sectorNumber, data);
128
129     active = TRUE;
130     UpdateLast(sectorNumber);
131     stats->numDiskReads++;
132     interrupt->Schedule(DiskDone, (_int) this, ticks,
DiskInt);
133 }
```

在说明的扇被从 UNIX 文件中读出后，active 被设置为 TRUE，并且模拟系统中断，注册一个在将来 tick 时间后将要发生的事件。这个事件到达的时间代表了磁盘操作完成的时间。当事件到达时，硬中断模拟器将通过中断处理程序 DiskDone，用事先传递给它的参数完成对硬盘操作的中断处理 DiskDone 是定义在 disk.cc 文件 29 行上的一个静态函数：

```
28  // dummy procedure because we can't take a pointer of a member
function
29  static void DiskDone(_int arg) { ((Disk
*)arg)->HandleInterrupt(); }
```

它简单的使用了一个参数 arg 指向一个 Disk 对象，并且调用该对象的>HandleInterrupt()函数。这个函数在 disk.cc 文件的 161-166 行：

```
161  void
162  Disk::HandleInterrupt ()
163  {
164      active = FALSE;
165      (*handler)(handlerArg);
166  }
```

可以看到该函数重新将变量 active 设置为 FALSE，并调用基本文件层在构造硬盘对象时安排的中断管理函数完成最终的硬盘中断处理。

4.1.3 内核 I/O 子系统

这一层也叫”基本文件系统”。关于硬盘 I/O 中断驱动，我们应当提供一种机制去阻止进程在没有完成 I/O 处理就向硬盘连续发 I/O 访问请求。硬盘管理程序也应提供在磁盘 I/O 完成后的中断处理。磁盘管理程序的另一项任务是按多进程或线程去同步当前磁盘的访问。所有这些对磁盘同步的管理任务在 Nachos 文件系统中都是由文件 synchdisk.h 中定义的 SynchDisk 类完成的。

```
27  class SynchDisk {
28    public:
29      SynchDisk(char* name);              // Initialize a
synchronous disk,
30                                          // by initializing
the raw Disk.
31      ~SynchDisk();                       // De-allocate the
synch disk data
32
33      void ReadSector(int sectorNumber, char* data);
34                                          // Read/write a disk
sector, returning
35                                          // only once the data
is actually read
36                                          // or written.  These
call
37                                          //
Disk::ReadRequest/WriteRequest and
38                                          // then wait until the
request is done.
39      void WriteSector(int sectorNumber, char* data);
40
41      void RequestDone();                 // Called by the disk
device interrupt
42                                          // handler, to signal
that the
43                                          // current disk
operation is complete.
```

```
44
45    private:
46      Disk *disk;                          // Raw disk device
47      Semaphore *semaphore;                // To synchronize
requesting thread
48                                           // with the interrupt
handler
49      Lock *lock;                          // Only one read/write
request
50                                           // can be sent to the
disk at a time
51  };
```

这里的信号量显然是用来控制被阻塞进程的,同时锁被用来提供互斥的访问磁盘。当然这个类必须要访问的是硬盘 DISK。函数 RequestDone()用于当 I/O 请求完成后释放磁盘中断处理程序。事实上这个函数被包装在一个静态函数 DiskRequestDone 中，这个函数在 synchdisk.cc 文件 26-32 行上：

```
26  static void
27  DiskRequestDone (_int arg)
28  {
29      SynchDisk* dsk = (SynchDisk *)arg;  // disk -> dsk
30
31      dsk->RequestDone();                          //
disk -> dsk
32  }
```

这样做的原因是因为 C++语言不允许类成员函数作为函数的参数被传递。

同步化的 I/O 操作函数：ReadSector(**int** sectorNumber, **char*** data)和 WriteSector(**int** sectorNumber, **char*** data)是在附加了同步操作再调用对应的裸盘上的 I/O 操作实现的。其中它们的源代码在 synchdisk.cc 文件中。仅分析一下在 72-79 行上的 ReadSector，WriteSector 与其类似。

```
72  void
73  SynchDisk::ReadSector(int sectorNumber, char* data)
74  {
```

```
 75      lock->Acquire();                    // only one disk I/O
at a time
 76      disk->ReadRequest(sectorNumber, data);
 77      semaphore->P();                     // wait for
interrupt
 78      lock->Release();
 79  }
```

lock 用于互斥多线程对于磁盘的访问，等待访问磁盘的线程被放置在 lock 等待队列中。信号量 semaphore 用于同步正在访问磁盘的线程对于磁盘的操作，一个线程在发出了磁盘操作命令后将被放置在中断事件队列中直到裸盘访问操作完成后再被唤醒。当磁盘 I/O 完成后，裸盘的中断处理程序将重新调用带有一个整形参数的函数 DiskRequestDone，而这个函数实际上又调用了磁盘同步函数 RequestDone。函数 RequestDone 的工作是调用 semaphore 信号量上的 V()操作，因此唤醒了阻塞在中断事件队列中的磁盘访问线程。

4.1.4 自由空间管理

正如以上我们所看到的，扇是磁盘中最小的存储单位，也是磁盘 I/O 操作最基本的单位。在 Nachos 中以扇区为基本单位的自由存储空间是由位示图类 BitMap 管理的。BitMap 类的定义可以在文件../userprog/bitmap.h 中找到：

```
34  class BitMap {
35    public:
36      BitMap(int nitems);          // Initialize a bitmap, with
"nitems" bits
37                                   // initially, all bits are
cleared.
38      ~BitMap();                   // De-allocate bitmap
39
40      void Mark(int which);        // Set the "nth" bit
41      void Clear(int which);       // Clear the "nth" bit
42      bool Test(int which);        // Is the "nth" bit set?
43      int Find();                  // Return the # of a clear bit,
and as a side
44                                   // effect, set the bit.
```

```
45                                  // If no bits are clear, return
-1.
46      int NumClear();             // Return the number of clear
bits
47
48      void Print();               // Print contents of bitmap
49
50      // These aren't needed until FILESYS, when we will need to
read and
51      // write the bitmap to a file
52      void FetchFrom(OpenFile *file);     // fetch contents from
disk
53      void WriteBack(OpenFile *file);     // write contents to
disk
54
55    private:
56      int numBits;                        // number of bits in
the bitmap
57      int numWords;                       // number of words of
bitmap storage
58                                          // (rounded up if
numBits is not a
59                                          //  multiple of the
number of bits in
60                                          //  a word)
61      unsigned int *map;                  // bit storage
62  };
```

私有成员变量 map 指向一个保存位示图的内存,变量 numBits 保存了能表示扇号的位数，numWords 保存了组成位示图的字数。如果要占用一个扇则与其扇号对应的位就置”1”,如果要释放一个扇则与其扇号对应的位就置”0”。注意函数 find()的作用，它返回找到的第一个空闲位的索引同时将该位置”1”。因为内存是易失性的，所以对应硬盘的位示图需要作为一个文件保存到磁盘上。它作为内核一个特殊文件被管理。函数 FetchFrom(OpenFile *file)和 WriteBack(OpenFile *file)用于完成这一目的。BitMap 类的实现在文件 userprog/bitmap.cc 中。

4.1.5 文件头(I-Node)

在 Nachos 中的文件头 Fie Header 也就是众所周知的 Unix 中的 I-node。I-node 是文件系统中重要的数据结构。它保存了除文件名以外的一个文件所用到的所有的管理信息。在基本 Nachos 系统中使用 File Header 类仅实现了简单的一级索引分配方法及其结构。File Header 类的定义在 filesys/filehdr.h 文件 38-64 行：

```
38  class FileHeader {
39    public:
40      bool Allocate(BitMap *bitMap, int fileSize);// Initialize
a file header,
41                                                  //
including allocating space
42                                                  //  on disk
for the file data
43      void Deallocate(BitMap *bitMap);            //
De-allocate this file's
44                                                  //  data
blocks
45
46      void FetchFrom(int sectorNumber);   // Initialize file
header from disk
47      void WriteBack(int sectorNumber);   // Write
modifications to file header
48                                          //  back to disk
49
50      int ByteToSector(int offset);       // Convert a byte
offset into the file
51                                          // to the disk sector
containing
52                                          // the byte
53
54      int FileLength();                   // Return the length
of the file
55                                          // in bytes
56
```

```
57      void Print();                       // Print the contents
of the file.
58
59    private:
60      int numBytes;                       // Number of bytes in
the file
61      int numSectors;                     // Number of data
sectors in the file
62      int dataSectors[NumDirect];         // Disk sector
numbers for each data
63                                          // block in the file
64  };
```

这里的 dataSectors[NumDirect]是一个扇区号的索引表，它记录了一个文件所占用的扇区号。numBytes 记录了文件的字节长度，numSectors 记录了文件占用的扇区数。这三个私有数据成员组成了 Nachos 文件的一个 I-node。

在该文件的 20-21 行有两个重要的常量定义：

```
20  #define NumDirect       ((SectorSize - 2 * sizeof(int)) /
sizeof(int))
21  #define MaxFileSize     (NumDirect * SectorSize)
```

NumDirect 是索引表 dataSectors 的长度，它是一个 Nachos 文件所能占用的数据块的最大数字。它也决定了一个 Nachos 文件的最大容量 MaxFileSize 的值。

可以看出 NumDirect 的最大值受到了扇区长度的限制，因为 Nachos 中限定一个扇区中仅能容纳一个 I-node。

在 Nachos 文件系统中当要建立一个新文件时首先调用 FileHeader 类的成员函数 Allocate(BitMap *bitMap, **int** fileSize)。它的实现在 filehdr.cc 文件 41-52 行：

```
41  bool
42  FileHeader::Allocate(BitMap *freeMap, int fileSize)
43  {
44      numBytes = fileSize;
45      numSectors  = divRoundUp(fileSize, SectorSize);
46      if (freeMap->NumClear() < numSectors)
47          return FALSE;           // not enough space
```

```
48
49      for (int i = 0; i < numSectors; i++)
50          dataSectors[i] = freeMap->Find();
51      return TRUE;
52  }
```

这个函数根据指定的文件长度 fileSize 换算出文件数据要占用的磁盘扇区数，并将申请到的自由扇区号登记到扇区索引表 dataSectors 中。即它完成了一个 Nachos 文件 I-node 的初始化工作。

成员函数 FetchFrom(**int** sectorNumber)和 WriteBack(**int** sectorNumber)的功能是获取和保存一个文件自身的 I-node。

4.1.6 打开文件

一个打开的文件是文件系统中的一个层次。正像 Unix 系统中一样，Nachos 系统中的每个进程都具有一个打开文件表。打开文件的数据结构应提供：

- 打开文件当前的位移位置
- 打开文件 I-node 的一个引用
- 打开文件的访问函数如 read 或 write

在 Nachos 系统中，这些数据结构是通过定义在 openfile.h 中的 OpenFile 类提供的：

```
64  class OpenFile {
65    public:
66      OpenFile(int sector);               // Open a file whose
header is located
67                                          // at "sector" on the
disk
68      ~OpenFile();                        // Close the file
69
70      void Seek(int position);            // Set the position
from which to
71                                          // start
reading/writing -- UNIX lseek
72
```

```
73      int Read(char *into, int numBytes); // Read/write bytes
from the file,
74                                          // starting at the
implicit position.
75                                          // Return the #
actually read/written,
76                                          // and increment
position in file.
77      int Write(char *from, int numBytes);
78
79      int ReadAt(char *into, int numBytes, int position);
80                                          // Read/write bytes
from the file,
81                                          // bypassing the
implicit position.
82      int WriteAt(char *from, int numBytes, int position);
83
84      int Length();                       // Return the number
of bytes in the
85                                          // file (this
interface is simpler
86                                          // than the UNIX idiom
-- lseek to
87                                          // end of file, tell,
lseek back
88
89    private:
90      FileHeader *hdr;                    // Header for this
file
91      int seekPosition;                   // Current position
within the file
92  };
```

可以看到在类 OpenFile 中除了说明了一系列的读写函数外，seekPosition 说明了打开文件的位移位置，hdr 说明了打开文件的 I-node。

在文件 openfile.cc 中定义了这个类中成员函数的实现。我们可以看一下这个类的构造函数怎样从磁盘上装入一个 I-node 和设置一个位移量为零的 OpenFile 的：

```
27  OpenFile::OpenFile(int sector)
28  {
29      hdr = new FileHeader;
30      hdr->FetchFrom(sector);
31      seekPosition = 0;
32  }
```

存放着 I-node 的扇号 sector 是由更高一层的模块-目录类提供的。我们将在下一部分讨论它。Openfile 类中的成员函数 Seek(..)和 Length()是很直观的。函数 Read(…)和 Write(…)分别是由调用下层的 ReadAt(…)和 WriteAt(…)实现的，在此就不一一分析了。

4.1.7 文件目录

基本 Nachos 文件系统的目录是非常简单的一级目录，所有文件都属于这个目录。

Nachos 的目录属性及其操作方法是由一个 Directory 类定义的。该定义在 directory.h 文件的 51-79 行：

```
51  class Directory {
52    public:
53      Directory(int size);                // Initialize an
empty directory
54                                          // with space for
"size" files
55      ~Directory();                       // De-allocate the
directory
56
57      void FetchFrom(OpenFile *file);     // Init directory
contents from disk
58      void WriteBack(OpenFile *file);     // Write
modifications to
59                                          // directory contents
back to disk
60
```

```
61      int Find(char *name);               // Find the sector
number of the
62                                          // FileHeader for
file: "name"
63
64      bool Add(char *name, int newSector);  // Add a file name
into the directory
65
66      bool Remove(char *name);            // Remove a file from
the directory
67
68      void List();                        // Print the names of
all the files
69                                          //  in the directory
70      void Print();                       // Verbose print of
the contents
71                                          //  of the directory
-- all the file
72                                          //  names and their
contents.
73
74    private:
75      int tableSize;                      // Number of
directory entries
76      DirectoryEntry *table;              // Table of pairs:
77                                          // <file name, file
header location>
78
79      int FindIndex(char *name);          // Find the index into
the directory
80                                          //  table
corresponding to "name"
81  };
```

其中的类 DirectoryEntry 定义了 Nachos 文件系统的目录结构，这个定义在 directory.h 文件的 32-39 行：

```
32  class DirectoryEntry {
```

```
33    public:
34      bool inUse;                         // Is this directory
entry in use?
35      int sector;                         // Location on disk to
find the
36                                          //   FileHeader for
this file
37      char name[FileNameMaxLen + 1];      // Text name for file,
with +1 for
38                                          // the trailing '\0'
39  };
```

DirectoryEntry 类中的布尔变量 inUse 用于指示这个目录项是否已占用。字符数组 name 是一个文件名串。变量 sector 指示该文件的 I-node 所在的扇号。

Nachos 中的每个文件都有一个 DirectoryEntry 类，多个文件就构成了 Nachos 的文件目录表，Director 类中的私有数据成员 table 指向了这个表的首项。

目录本身也是一个文件，它需要经常从磁盘中取出和写回。Director 类中成员函数 FetchFrom(OpenFile *file)和 WriteBack(OpenFile *file)用于此目的。函数 Add(char *name, int newSector)和 Remove(char *name)则用于从目录表中加入或删除一条目录项。所有这些函数的实现都在可在文件 directory.cc 中找到。请看在该文件 129-141 行上 Add(…)的实现：

```
129  bool
130  Directory::Add(char *name, int newSector)
131  {
132      if (FindIndex(name) != -1)
133          return FALSE;
134
135      for (int i = 0; i < tableSize; i++)
136          if (!table[i].inUse) {
137              table[i].inUse = TRUE;
138              strncpy(table[i].name, name, FileNameMaxLen);
139              table[i].sector = newSector;
140          return TRUE;
141          }
142      return FALSE;       // no space.  Fix when we have
extensible files.
```

```
143  }
```

基本 Nachos 文件中的目录表的长度是固定的，因此目录文件的长度也被固定了，所以这个函数仅简单的在表中查找第一个未用的目录入口并向里放一个文件名和 I-node 扇号。函数 List()类似 Unix 系统的”ls”命令，列出文件目录中所有的文件名。其他目录操作函数的实现都相当直观，在此就不一一分析了。

4.1.8 逻辑文件系统

这是文件系统的最高层，在这一层中应实现文件的逻辑分区和每个逻辑分区的初始和格式化。需要特别注意在文件 filesys.cc 中第 80-143 行有关文件系统构造的源代码：

```
80  FileSystem::FileSystem(bool format)
81  {
82      DEBUG('f', "Initializing the file system.\n");
83      if (format) {
84          BitMap *freeMap = new BitMap(NumSectors);
85          Directory *directory = new Directory(NumDirEntries);
86          FileHeader *mapHdr = new FileHeader;
87          FileHeader *dirHdr = new FileHeader;
88
89          DEBUG('f', "Formatting the file system.\n");
90
91      // First, allocate space for FileHeaders for the directory
and bitmap
92      // (make sure no one else grabs these!)
93          freeMap->Mark(FreeMapSector);
94          freeMap->Mark(DirectorySector);
95
96      // Second, allocate space for the data blocks containing the
contents
97      // of the directory and bitmap files.  There better be
enough space!
98
99          ASSERT(mapHdr->Allocate(freeMap, FreeMapFileSize));
```

```
100           ASSERT(dirHdr->Allocate(freeMap,
DirectoryFileSize));
101
102       // Flush the bitmap and directory FileHeaders back to disk
103       // We need to do this before we can "Open" the file, since
open
104       // reads the file header off of disk (and currently the disk
has garbage
105       // on it!).
106
107           DEBUG('f', "Writing headers back to disk.\n");
108           mapHdr->WriteBack(FreeMapSector);
109           dirHdr->WriteBack(DirectorySector);
110
111       // OK to open the bitmap and directory files now
112       // The file system operations assume these two files are
left open
113       // while Nachos is running.
114
115           freeMapFile = new OpenFile(FreeMapSector);
116           directoryFile = new OpenFile(DirectorySector);
117
118       // Once we have the files "open", we can write the initial
version
119       // of each file back to disk.  The directory at this point
is completely
120       // empty; but the bitmap has been changed to reflect the
fact that
121       // sectors on the disk have been allocated for the file
headers and
122       // to hold the file data for the directory and bitmap.
123
124           DEBUG('f', "Writing bitmap and directory back to
disk.\n");
125           freeMap->WriteBack(freeMapFile);         // flush
changes to disk
126           directory->WriteBack(directoryFile);
127
```

```
128          if (DebugIsEnabled('f')) {
129              freeMap->Print();
130              directory->Print();
131
132          delete freeMap;
133          delete directory;
134          delete mapHdr;
135          delete dirHdr;
136          }
137      } else {
138      // if we are not formatting the disk, just open the files
representing
139      // the bitmap and directory; these are left open while
Nachos is running
140          freeMapFile = new OpenFile(FreeMapSector);
141          directoryFile = new OpenFile(DirectorySector);
142      }
143  }
```

在这个文件系统构造函数中生成了目录和位示图文件的空间，并打开了这两个文件准备好对于用户文件的操作。在 filesys.cc 文件 63-65 行对于目录和位示图文件的长度进行了定义：

```
63  #define FreeMapFileSize         (NumSectors / BitsInByte)
64  #define NumDirEntries           10
65  #define DirectoryFileSize       (sizeof(DirectoryEntry) *
NumDirEntries)
```

NumDirEntries 定义了目录表项数，它是可建立的最大文件数。FreeMapFileSize 定义了 BitMap 文件的长度，DirectoryFileSize 定义了目录文件的长度。

分析一下文件系统构造函数首次工作的过程。首先它建立一个位示图和一个目录对象(84-85)，接着为这两个对象建立对应的文件头对象(86-87)，并在位示图中置位它们所占用的扇号（位示图文件为 0，目录文件为 1）。接下来要把这两对象的 I-node 保存到磁盘上(108-109)。现在可以打开这两个文件了(115-116)，这两个文件在内存中已经有了新的的内容，将这些新内容写入其对应的文件中(125-126)。现在内存中的对象 freeMap 、directory、mapHdr、dirHdr 已经不再有用，可以将其从内存中删除了(132-135)。

4.2 文件系统实验内容

这部分的实验室是通过 Nachos 操作系统研究文件系统的功能。为了能在较短的时间周期内读通文件系统的功能并展开文件系统的开发，基本 Nachos 系统的文件系统设计非常短小和简单。为了搞清最基本的 Nachos 文件功能，我们需要运行一下 Nachos 有关文件系统的命令，观察一下 Nachos 模拟硬盘操作的效果。以便展开对于文件系统功能的扩充。

4.2.1 实验目的

- 知道怎样生成 Nachos 文件系统。
- 知道怎样测试 Nachos 文件系统的基本功能。
- 知道怎样检查 Nachos 文件系统中模拟硬盘中的内容。
- 完善 Nachos 文件建立的过程。
- 使 Nachos 文件的长度可以扩展。
- 扩充 Nachos 文件的最大容量。

4.2.2 编译 Nachos 文件系统

为了编译带有文件系统的 Nachos 系统，请进入../filesys 目录。这个目录中的 Makdefile 包括两个 Makefile.local 文件：../threads/Makefile.local 和../filesys/Makefile.local。

文件../filesys/Makefile.local 的内容如下：

```
ifndef MAKEFILE_FILESYS_LOCAL
define MAKEFILE_FILESYS_LOCAL
yes
endef

# Add new sourcefiles here.

CCFILES +=bitmap.cc\
          directory.cc\
```

```
        filehdr.cc\
        filesys.cc\
        fstest.cc\
        openfile.cc\
        synchdisk.cc\
        disk.cc

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

endif # MAKEFILE_FILESYS_LOCAL
```

可以看到这个版本的 Nachos 在使用了../threads 和../userprog 的 C++文件的基础上又附加了以上 CCFILES 命令中列出的 C++文件。执行 make 命令。一个带有文件系统功能的新版的 Nachos 系统 nachos 就会在该目录中生成。

4.2.3 Nachos 文件命令的用法

有关 Nachos 命令的完整用法定义在../threads/main.cc 和../threads/system.cc 文件中。这里仅列出与文件系统有关的命令。可选标识符 d 和 f 用于显示有关文件系统的调试信息。

- nachos [d f] –f
  这个命令用于在任何其它文件系统命令使用之前格式化一个模拟的叫 DISK 的硬盘。
- nachos [d f] –cp Unix 文件名 Nachos 文件名
  从 Unix 系统中拷贝一个文件到 Nachos 系统中。这是在当前 Nachos 文件系统建立一个文件的唯一的方法。
- nachos [d f] –p 文件名
  显示 Nachos 系统中一个文件的内容（类似 Unix 系统中的 cat 命令）
- nachos [d f] –d 文件名
  删除 Nachos 系统中一个文件（类似 Unix 系统中的 rm 命令）
- nachos [d f] –l
  列出 Nachos 系统中所有的文件名（类似 Unix 系统中的 ls 命令）

- nachos [d f] –D
  转储 Nachos 整个文件系统的信息到屏幕，这包括位示图、文件头、目录和文件内容。
- nachos [d f] –t
  测试 Nachos 文件系统是否工作正常。

在../filesys/test 目录中有三个用于测试 Nachos 文件系统的文件：small、medium、big。它们是一些文本文件，你可以先用 Unix 的 od 或 cat 命令查看一下它们的内容以便利用它们检测 Nachos 文件系统的工作情况。

4.2.4 检测基本 Nachos 文件系统的工作情况

执行以下命令并且检查执行结果：

$ nachos –f

Nachos 现在应在当前目录中建立一个名为 DISK 的模拟硬盘

$ nachos –D

转储模拟硬盘 DISK 的整个内容到屏幕。你应当看到：

```
...
FileHeader contents. File size: 128. File blocks:
2
File contents:
\1f\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Directory file header:
FileHeader contents. File size: 200. File blocks:
3    4
File contents:
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Bitmap set:
0, 1, 2, 3, 4,
```

Directory contents:
Directory contents:
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
Ticks: total 5500, idle 5030, system 470, user
0
Disk I/O: reads 10, writes
0
Console I/O: reads 0, writes
0
Paging: faults
0
Network I/O: packets received 0, sent
0
Cleaning up..

从以上输出信息可以看出Nachos中一个刚格式化的硬盘扇区分配的情况。第0，1扇存放了BitMap和Directory文件的文件头，长度分别为128和200字节。第2扇存放了Bitmap文件的内容、十六进制的数\1f说明该盘的第0，1，2，3，4扇已经占用。第3，4扇用于存放目录信息，当前没有内容（全0）。

$ nachos –cp test/small small

该命令将Unix系统中的test/small文件拷贝到Nachos系统的small文件。你可以使用命令：

$ nachos –l small

$ nachos –p small

$ nachos –D

观察和确定在Nachos系统中已经建立了一个叫small的文件。

4.2.5 Nachos 文件长度的扩展

当前 Nachos 文件的长度不能够扩展，既不能对现有文件追加新的内容。我们的第一项任务就是设计和实现一个能够扩展文件长度的新的 Nachos 文件系统。在开始你的新的设计之前应首先阅读一下 Nachos 文件系统中有关的源代码，分析为什么当前 Nachos 不能扩展文件的长度。考虑以下的一些问题：

- 哪些模块需要改变？ 哪些模块不需要改变？

- 在需要改变的模块中需要改变哪些函数怎样改变？
- 在需要改变的模块中需要添加新的函数和变量吗？
- 需要改变的模块中要删除某些变量吗？需要交叉引用其它模块吗？

在做了以上的分析和考虑后，就可以开始我们的设计与实现了。为了保留原始版本我们可以进入../lab5 目录，在此目录中已包含了一些测试文件。同前面做过的开发类似，我们可以从../filesys 目录中将 arch 子目录递归的拷贝到../lab5 中。在../lab5 中 main.cc 和 fstest.cc 文件有新的内容。它们包含了为了测试新增功能而准备的新的文件系统命令。文件 main.cc 是完整的无需改变。文件 fstest.cc 也是完整的的但需要将其中 4 个注释行打开，这 4 行在函数 Append(..)和 NAppend(…)中，如下所示：

```
// openFile->WriteBack();
// printf("inodes have been written back\n");
```

将它们前面的//注释符删除打开着 4 行语句，这样就为类 OpenFile 添加了 Writeback()函数。

可以采用以下的步骤编码和实现你的设计：

- 接口：第一步在模块和你要改变的文件系统之间建立一个新的接口。仅改动.h 文件中的类定义和.cc 文件中的函数头，不改动函数体。使用 make 编译和连接一次改动后的系统，确保改动没有引发语法和连接的错误。
- 编码和测试：下一步改动函数体并且测试结果代码。可以将整个过程分为若干步，在每一步达到了预期效果后再进行下一步。

4.2.6 测试新的文件系统

我们需要一些命令去测试 Nachos 文件系统的新功能。这些命令在../lab5 目录中的 main.cc 和 fstest.cc 文件中已经准备好了,它们是：

- nachos [d f] –ap Unix 文件名 Nachos 文件名
  这个命令追加一个 Unix 文件的内容到一个已存在的 Nachos 的文件中。它用于测试是否能扩展一个已经存在的 Nachos 文件的长度。
- nachos [d f] –hap Unix 文件名 Nachos 文件名
  这个命令重写一个 Unix 文件的内容到一个已存在的 Nachos 的文件中。它用于测试当一个 Unix 文件的长度超过现有 Nachos 文件长度的一半时，Nachos 文件的长度是否能被扩展。

应当阅读一下../lab5 目录中的 main.cc 和 fstest.cc 文件，了解这些命令是如何实现的。

为了测试新功能，需要使用命令 nachos – f 刷新模拟硬盘 DISK，首先可以顺序输入以下命令，测试追加文件内容的功能：

nachos -cp test/small small
nachos -ap test/small small
nachos -cp test/empty empty
nachos -ap test/medium emtpy

然后可以使用 nachos –D 转储 DISK 的映像，检查以上命令操作的结果：

Bit map file header:
FileHeader contents. File size: 128. File blocks:
File contents:
\ff\7\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Directory file header:
FileHeader contents. File size: 200. File blocks:
3 4
File contents:
\1\12\11@\5\0\0\0small\0\0\0\0G\5\8\1H\5\8\8\0\0\0empty\0\0\0\0\0\0\0\0G
\5\8\0\0\0\0\0\0\0\00\0\0\0\e0\12\11@\0\12\11@\10G\5\8X\13\0\0\d0\15\11@
\18\0\0\0\0\12\11@\c8\12\11@\80G\5\8\18\0\0\0‘\0\0\0\0\0\0\0\0\0\0\0\2\0
\0\0\a0F\5\8\0\0\0\0\0\0\0\0\18\0\0\0
\0\0\0\0\1\0\0\0\a8D\5\8\0\11\0\0\c8\15\11@H\0\0\0\f8\12\11@\f8\12\11@\0F
\5\8\0\0\0\0\0\0\0\00\0\0\0\e0\12\11@\0\12\11@\90D\5\8\c8\15\11@HH\5\8\1
8
\0\0\0
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
Directory contents:
Name: small, Sector:
5
FileHeader contents. File size: 168. File blocks:
6 7
File contents:
small file small file small file\asmall file small file small
file\a***end of file***\asmall file small file small file\asm
all file small file small file\a***end of file***\
a

Name: empty, Sector:
8
FileHeader contents. File size: 162. File blocks:
9 10
File contents:
medium file medium file medium file\amedium file medium file med
ium file\amedium file medium file medium file\amedium file medi
um file medium file\a***end of file***\a

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
Ticks: total 8490, idle 8000, system 490, user
0
Disk I/O: reads 16, writes
0
Console I/O: reads 0, writes
0
Paging: faults
0
51
Network I/O: packets received 0, sent
0
Cleaning up..

可以看出现在 Nachos 中硬盘信息存放的情况。第 0，1 扇存放了 BitMap 和 Directory 文件的文件头，长度分别为 128 和 200 字节。第 2 扇存放了 Bitmap 文件的内容、十六进制的数\1f 说明该盘的第 0，1，2，3，4 扇已经占用。第 3，4 扇用于存放目录信息。
现在文件目录中有了一个叫 small 的文件，它的长度为 168 字节（占 2 扇），它的 i-node 在第 5 扇，数据存放在第 6，7 扇。
文件目录中还有了一个叫 empty 的文件，它的长度为 162 字节（占 2 扇），它的 i-node 在第 8 扇，数据存放在第 9，10 扇。

我们还需要完成测试文件重写的功能。这需要我们自己去设计一个测试程序以确保我们新设计的文件系统在各种不同的情况下都能正常的工作。

4.2.7 扩充 Nachos 文件的最大容量

在 Nachos 当前的设计中文件数据空间的分配采用了一级索引方式。文件头(inode)仅能记录 NumDirect（这个值等于 30）个扇号。因此一个 Nachos 文件的最大容量仅有 NumDirect*SectorSize=3840 字节。虽然我们在../lab5 改进了 Nachos 文件系统的功能使它具有了扩展文件长度的功能，但文件扩展的最大长度不能超出 3840 这个最大容量。现在我们的任务就是要在 Nachos 文件系统中增加类似 Unix 文件系统的多级索引方式，扩充 Nachos 能够记录的扇号，从而增大 Nachos 文件的最大容量。图 4.2 表示了一个多级索引方式的结构

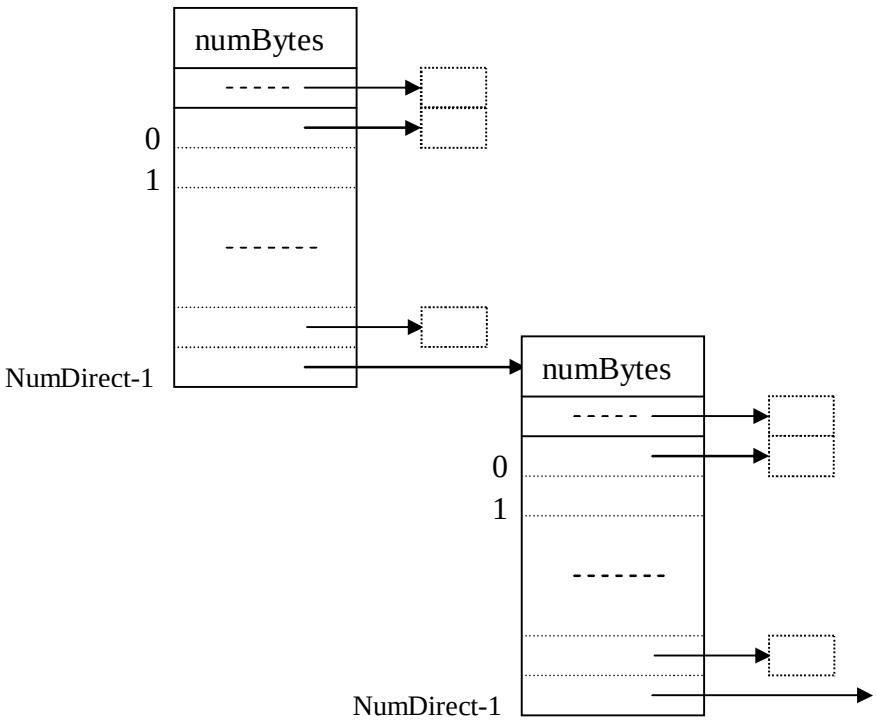

图 4.2 具有多级索引结构的 I-node

例如，当文件要求的扇数超出 NumDirect 时可以使用 dataSectors[]的最后一项存放一个二级间接索引节点的扇号，在二级扇区中存放超出的扇号。这样一个具有二级索引结构的 I-node 的定义应为：

```
    private:
      int numBytes;                 // Number of bytes in the file
```

```
    int numSectors;              // Number of data sectors in the
file
    FileHeader *indirect;        // pointer to indirect header
    int dataSectors[NumDirect];  // Disk sector numbers for each
data
```

因此 NumDirect 的定义就应改为：

```
20  #define NumDirect      ((SectorSize - 3 * sizeof(int)) /
sizeof(int))
```

现在文件扇区的分配方法就应根据文件实际长度的大小来确定了：当文件长度小于等于(NumDirect-1)*SectorSize 字节时我们仍采用一级索引的分配方法；当文件长度大于 (NumDirect-1)*SectorSize 字节时我们则采用二级索引的分配方法，使用 dataSectors[NumDirect-1]存储二级索引扇号，再在二级索引扇区的 dataSectors[]中存放文件数据扇号；这样一个二级索引结构能够存放的文件容量就可以达到：(NumDirect-1)*SectorSize+NumDirect*SectorSize=7296 字节

4.2.8 进一步的开发与测试

本任务的开发可以在../ass3 目录中展开。原始的../ass3 目录中仅有 Makefile 和 Makefile.local 两个文件。我们可以将../lab5 目录中已开发好的内容全部拷贝到该目录中在../lab5 的基础上展开工作。要做的工作都是围绕 FileHeader 类进行的。

注意，在文件 fstest.cc 中，宏 Tranfersiae 定义了每次向一个 Nachos 文件追加的字节个数为 10。你设计的程序应对于任何一个追加值，例如 100，250，400，1000，2000 都应当是正确的

为了能够验证我们的设计达到了预期的效果，在目录../ass2 中准备了几个测试用的文件：

rwrr	1	ptang	337	Oct 2 17:15	big
rwrr	1	ptang	3804	Oct 2 17:13	huge
rwrr	1	ptang	3573	Oct 2 17:12	huge1
rwrr	1	ptang	3678	Oct 2 17:13	huge2
rwrr	1	ptang	169	Oct 2 17:14	medium
rwrr	1	ptang	1	Sep 28 18:07	onebyte
rwrr	1	ptang	90	Oct 2 17:14	small

在我们新设计的 Nachos 文件系统中第一级 i-node 能记录的扇号为 28 个，即能存放的文件内容为 3584 字节。如果文件长度超出了 3584 字节就应当使用二级间接索引存放超出的数据。

我们可以使用以下的步骤检测是否能实现要求的功能：

首先拷贝文件 huge1 到 Nachos 中：nachos –cp ../test/hug1 huge1

将模拟盘映像转储显示，正确的内容应为：

Bit map file header:

FileHeader contents. File size: 128. File blocks:

2

File contents:

\ff\ff\ff\ff\0\0\0\3\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

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

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

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

Directory file header:

FileHeader contents. File size: 200. File blocks:

3 4

File contents:

\0\0\0\1\0\0\0\5huge1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

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

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

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

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

\0\0\0\0\0\0\0\0\0\0\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, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,

Directory contents:

Name: huge1, Sector: 5

FileHeader contents. File size: 3573. File blocks:

6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33

File contents:

This is a huge file.\aThis is a huge file.\aThis is a huge file.\aThis is a huge file.\aThis is a huge file.\aThis is a huge file.\aTh

......

可以看出由于现在文件 huge1 的长度为 3573 字节，占用了从第 6 扇区

到第 33 扇区共 28 个扇区，它还没有超出 Nachos 文件系统一级索引所能容纳的数量。

然后追加文件 small 到 Nachos 的 huge1 文件中：nachos –ap ../test/small huge1。再看模拟盘映像转储显示，扇号占用的情况应为：

Bitmap set:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,

Directory contents:

Name: huge1, Sector: 5

FileHeader contents. File size: 3663. File blocks:

6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 35

File contents:

.....

可以看出由于现在文件 huge1 的长度为 3663 字节，它的长度已经超出了一级索引的 28 扇即 3584 字节的限制，需要 29 扇的存储空间。因此它占用了从第 6 扇区到第 35 扇区共 30 个扇区，其中第 34 扇用于存放二级节点，所以在文件块号中没有出现。

最后我们可以再追加 medium 到 Nachos 的 huge1 文件中：nachos –ap ../test/medium huge1。现在 Nachos 中的 huge1 文件的长度应为 3832 字节，应占有 28+2=30 扇。下面是利用 Unix 的 od 命令转储的其中的 33 扇（从 10204 字节开始）到 36 扇（到 11203 字节结束）的内容，可以看出其中文件的存放跳过了第 34 扇。

```
0010200    a h u g e f i l e . \n T h i
0010220    s I s a h u g e f i l e
……
0010340    \n * * * e n d o f h u g e 1
0010360    f i l e * * * \n s m a l l f
0010400    i l e \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
\0
0010420     \0 \0 \0 # \0 \0 \0 $ \0 \0 \0 \0 \0 \0
\0 \0
0010440    \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
\0 \0
*
0010600    \0 \0 \0 \0 s m a l l f i l e s
0010620    m a l l f i l e \n s m a l l
0010640    f i l e s m a l l f i l e
```

附录 A 操作系统课程设计学习建议

A.1 认真阅读分析源代码

在开始我们操作系统课程设计的编码和实验之前，应首先认真阅读每一单元基本 NachOS 系统内核源代码，在理解 NachOS 系统的基本构造的基础上注意引用现代操作系统的一些先进技术和高级概念展开我们的设计和实验。

A.2 设计时间安排

可以考虑以下的时间进度完成设计任务，其中分析和设计共 32 学时，其中代码分析可有指导教师来讲解。实验 36 学时：

次序	学时	方 式	内 容	实验报告	难度系数
1	4	代码分析	分析基本 Nachos 系统线程管理		
2	4	系统设计	扩充基本 Nachos 系统线程管理功能		
3	2	实验调试	熟悉 Nachos 操作系统开发技术		1
4	6	实验调试	实现扩充的 Nachos 系统线程管理功能	提交报告和结果	2
5	4	代码分析	分析基本 Nachos 系统的同步机制		
6	4	系统设计	扩充基本 Nachos 系统		

			线程同步机制		
7	2	实验调试	实验 Nachos 基本同步机制功能		1
8	8	实验调试	实现扩充的 Nachos 高级同步机制的设计	提交报告和结果	3
9	4	代码分析	分析多道程序设计的内存管理方法和系统调用机理		
10	4	系统设计	扩充 Nachos 系统内存管理功能和系统调用		
11	2	实验调试	实现扩充的 Nachos 多道程序设计内存管理功能		1
12	8	实验调试	实现扩充的 Nachos 系统调用功能的设计	提交报告和结果	4
13	4	代码分析	分析 Nachos 文件系统		
14	4	系统设计	扩充文件系统功能		
15	2	实验调试	Nachos 基本 I/O 系统功能		1
16	8	实验调试	实现扩充的 Nachos 文件系统的功能	提交报告和结果	4

A.3 设计过程的组织

- 可以组成 2-4 人的设计小组，以便于交流讨论。
- 控制设计进度，确保按时完成设计。

- 及时与指导教师交流，定期汇报设计情况。
- 按时提交设计实验报告

A.4 设计实验的检查

检查可以有设计检查和实验检查。

设计检查可以通过学生提交到网上的设计报告和实验结果考察

实验检查可以通过上机检查：包括编辑过程、编译过程、调试过程、结果测试过程。

在全部设计都完成后可以组织一次答辩会让学生自己来介绍自己的设计。

A.5 设计成绩的评定

设计成绩的评定应从多方面考察。

- 设计题目的难度系数
- 设计实验报告的质量
- 设计实验结果的性能
- 设计实验完成的时间
- 设计实验的创新性

附录 B 操作系统课程设计实验报告

在完成了一个阶段的工作后，写好设计实验报告是很重要的，它可以帮助我们：

- 总结阶段性成果，启发创新思维。
- 锻炼科研精神，训练写作能力。
- 及时反馈实验效果，便于交流讨论
- 作为成绩评定的主要参考

指导教师应事先给出设计实验报告的标准格式，规定好报告的提交日期和提交方式。

- 对于设计实验报告的检查应注意：
- 是否具有独到的创新设计
- 是否达到了设计的基本要求
- 是否按规定的格式编写的报告
- 结构是否严谨，清晰
- 内容是否完整，全面
- 语言文字是否流畅，表达是否准确

建议的设计实验报告基本内容：

操作系统课程设计实验报告

一、 基本信息
- 设计实验题目
- 小组编号
- 完成人
- 报告日期

二、 设计和实验内容简要描述
- 设计目标
- 设计要求
- 实验的软硬件环境

三、 报告的主要内容
- 总体设计思路

- 主要对象的模型描述
- 主要类属性代码的分析说明
- 主要类方法代码的分析说明
- 项目管理文件的说明

四、 实验过程和结果

- 设计和实验投入的学时数
- 调试排错过程的记录
- 多种方式测试结果的记录
- 实验结果的分析综合

五、 设计和实验的总结

- 实验中遇到的问题和解决的方法
- 实验结果达到设计目标的程度
- 设计还可以进行哪些改进
- 设计和实验得到哪些收获和启发

附录 1：参考文献

附录 2：程序源代码

参考文献:

1. Abraham Silberschatz, Peter Galvin, and Greg Gagne, Applied Operating system concept, John Wiley & Sons, Inc, Sixth Edition ,2002
2. Gary Nutt. Operating Systems: A Modern Perspective. Addison Wesley, 2002
3. Andrew S. Tanenbaum, Modern Operating Systems, Prentice Hall, Second Edition, 2001
4. Thomas Narten ,A Road Map Through Nachos ,1997