目　　录

第一章 绪论

第一节 Nachos概述

一、引言

计算机操作系统是一门实践性很强的课程。一般地阐述其工作原理，很可能使本来具体生动的内容变得十分抽象、枯燥并难以理解。解决好理论与实践相结合的问题是提高操作系统教学质量的关键。一门好的操作系统实践课将使读者更加形象和深刻地理解课堂中讲述的概念、原理和它们的应用。

国内外许多著名的大学都在操作系统教学实践方面作了大量研究，比较突出的有著名计算机专家 A.S.Tanenbaum 设计和实现的 MINIX。MINIX 是一个比较完整的操作系统，它的用户界面类似于 UNIX。说它比较完整，是因为它包括了进程管理、文件系统管理、存储管理、设备管理以及 I/O 管理等操作系统的所有重要内容，而且还包含了系统启动和 Shell 等实际操作系统不可缺少的部分。由于 MINIX 较 UNIX 的出现晚十年，所以在程序风格上较原来的 UNIX 要好得多，更加结构化和模块化。包含有 3000 行注释的 12000 行源代码使整个系统较为容易阅读和理解。但是 MINIX 作为教学用操作系统有它的不足之处，就是由于它的目标是一个完整的操作系统，必然要和具体的设备打交道；而且不同的机器指令集需要有不同的编译器，所以 MINIX 的移植性并不令人满意。一个 MINIX 操作系统需要占据一台独立的主机，所以在网络的配置和实现上比较复杂，读者需要有一定的实践经验才能完成。

上海交通大学开发的 MOS 操作系统是另一个较成功的教学用操作系统。它是一个小型而功能较齐全的多道程序的操作系统，主要包括作业调度管理和文件系统管理，建立在一个只包含十几条指令的指令集虚拟机基础之上。由于 MOS 比较简单，读者可以非常容易地理解操作系统课程中讲述的进程调度和文件系统等部分原理。MOS 的不足是过于简单，不能涵盖操作系统的大部分功能。MOS 的虚拟机指令集是自定义的，没有现成的编译器，所以读者必须直接编写汇编程序才能在 MOS 虚拟机上运行。这样就缺乏开发大型应用程序的能力。但是 MOS 毕竟给了读者一个自由发挥的空间，在 MOS 的基础上衍生出 TOS 等学生自己定义和实现的相对完整的操作系统。

二、Nachos教学用操作系统

作为教学用操作系统，需要实现简单并且尽量缩小与实际操作系统之间的差距，所以我们采用 Nachos 作为操作系统课程的教学实践平台。Nachos 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，在美国很多大学中得到了应用，它在操作系统教学方面具有一下几个突出的优点：

- 采用通用虚拟机
  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 gc++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。

- 使用并实现了操作系统中的一些新的概念
  随着计算机技术和操作系统技术的不断发展，产生了很多新的概念。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。与 MINIX 不同，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。

  另外，不确定性是操作系统所必须具有的特征，Nachos 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 不是向读者展示一个成功的操作系统，而是让读者在一个框架下发挥自己的创造性进行扩展。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。Nachos 中具有所有这些内容，但是在很多方面作了一定的限制，比如只有一级索引结构限制了系统中最大文件的大小。读者可以应用学到的各种知识对文件系统进行扩展，逐步消除这些限制。Nachos 在每一部分给出很多课程作业，作为读者进行系统扩展的提示和检查对系统扩展的结果。

- 面向对象性
  Nachos 的主体是用 C++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。

以下各章分五个部分讲述 Nachos 的各个部分以及它们的功能。它们是机器模拟、线程管理、文件系统管理、用户程序和虚拟存储以及网络系统。各章的安排是：

第二章分析 Nachos 虚拟机的各个部分，包括中断系统、定时器、以及一些外部设备，如磁盘、键盘和显示器。Nachos 的应用程序将在这个虚拟机上运行。

第三章分析 Nachos 如何实现多线程机制以及 Nachos 的线程管理方法。Nachos 没有借助于

属主 UNIX 操作系统的多进程机制，而是通过编写自己的进程图象切换函数来实现多线程。该部分对 Nachos 的进程图象切换函数作了详细介绍。

第四章分析 Nachos 的文件系统。Nachos 原有的文件系统非常简单，该部分在分析原有文件系统的基础上提出了对文件系统的扩展要求。

第五章介绍用户程序和虚拟存储。该部分补充介绍了 Nachos 对虚拟机内存、寄存器以及 CPU 的模拟。现有的 Nachos 系统没有实现虚拟内存，当一个用户进程的逻辑地址空间较大时，就不能在现有 Nachos 上运行。该部分提出了虚拟内存的概念，并且给出了一个实例。

第六章论述了 Nachos 的网络系统，Nachos 的网络部分实现了不可靠的定长报文传送，在此之上需要建立可靠的网络，并实现网络应用程序。

第二节 Nachos的实验环境

一、Nachos的安装

本书的实际实验环境是 Linux，Nachos 可以运行在内核版本 1.2.13 以上的各种 Linux 版本，包括 Slackware 和 Redhat。编译器的版本是 gcc2.7.2 版本以上。

本书附有一张软盘，磁盘的格式为 DOS 格式，磁盘上有一个名为“nachos-3.4.tgz”的压缩文件。学生需要将此文件拷贝到自己的工作目录下：

```
~/$ mcopy a:nachos-3.4.tgz .
```

并将其解开：

```
~/$ gzip -dc nachos-3.4.tgz | tar xf -
```

二、Nachos的目录结构

以上操作系统可以发现在工作目录下生成一个名为 nachos-3.4 的目录。该目录中含有：

copyright	文件	Nachos 的版权信息
readme	文件	Nachos 的 readme 信息
nachos.ps	文件	Nachos 的介绍文档（Postscript 格式）
c++example	目录	有关 C++介绍和实例
doc	目录	Nachos 各个部分介绍和原有的作业要求
code	目录	Nachos 各个部分的源代码

最主要的部分是 Nachos 的源代码部分。它的目录结构是：

Makefile	文件	Nachos 的 Makefile 文件。当 Nachos 需要移植到其它系统时，
Makefile.common	文件	可以修改 Makefile.dep 中的 HOST 参数
Makefile.dep	文件	
machine	目录	Nachos 虚拟机模拟部分源代码
threads	目录	Nachos 线程管理部分源代码
filesys	目录	Nachos 文件系统管理部分源代码
userprog	目录	Nachos 用户程序部分源代码
network	目录	Nachos 网络管理部分源代码
vm	目录	Nachos 虚拟内存管理部分源代码
test	目录	一些测试用应用程序
bin	目录	包含有用户程序目标码变换的程序

三、各个部分的编译运行

Nachos的各个部分都可以独立编译运行，也可以同时编译各个部分。全部编译可以采用如下命令：

```
~/nachos-3.4$   make
```

当需要单独编译线程管理部分时，先进入 threads 目录，然后采用如下命令：

```
~/nachos-3.4/threads$   make  depend
~/nachos-3.4/threads$   make  nachos
```

实际上，各部分目录下都有一个 Makefile 文件，内容大体相同，区别在于一些条件编译的参数。比如在单独编译线程管理部分时，文件管理部分就被屏蔽了，这样读者读者就可以专心于线程管理部分的调试。

四、应用程序的编译

由于 Linux 指令集和 R2/3000 指令集不同，用户编写的应用程序用 Linux 系统中标准 gcc 编译后，不能直接在 Nachos 虚拟机环境下运行。所以需要采用交叉编译技术。所谓交叉编译技术是在一个操作系统下将源码编译成另一个操作系统的目标码，这里就是在 Linux 下通过 gcc 交叉编译版本将用户程序的源码编译成 R2/3000 指令集的目标码。

在 Linux 中，没有缺省的交叉编译工具。读者可以到上海交通大学计算机系 FTP 服务器上下载，URL 为:

ftp://donkey.cs.sjtu.edu.cn/linux/cross-compiler.tgz

该文件的解开需要有超级用户的权利，将解开至/usr/local/目录下：

```
/# gzip -dc cross-compiler.tgz | tar xf -
```

在编译用户程序时，用交叉编译器将源码编译成 R2/3000 指令集的目标代码，再经过一个简单的转换就可以在 Nachos 虚拟机上运行。注意，在读者实现虚拟存储之前，有些应用程序可能会因为使用过多的内存而不能运行。

第二章 机器模拟

第一节　概述

Nachos 是建立在一个软件模拟的虚拟机上的。该虚拟机包括计算机的基本部分：如 CPU、主存、寄存器、中断系统，还包括一些外部设备，如终端设备、网络以及磁盘系统。现代许多操作系统都是先在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。软件模拟的硬件可靠性比真实的硬件高的多，不会因为硬件故障而导致系统出错，因而便于调试。模拟的硬件还可以监视程序对硬件的操作，并加以严格的限制，在程序误操作时报告详尽的出错信息。这些都是真实硬件难以做到的。

用软件来模拟硬件另一个优点是充分利用了宿主机操作系统的软件资源，避免了编写复杂的硬件控制程序。更重要的是提高了程序的可移植性，只要在不同硬件上实现 Nachos 虚拟机就完成了 Nachos 的大部分移植工作。我们将 Nachos 移植到 Linux 上的工作就受益于这种设计。

下面对 Nachos 的机器模拟部分作概要说明。Nachos 是用 C++语言中的类来表示各个对象的。

其中 Machine 类用来模拟计算机主机。它提供的功能有：读写寄存器。读写主存、运行一条用户程序的汇编指令、运行用户程序、单步调试用户程序、显示主存和寄存器状态、将虚拟内存地址转换为物理内存地址、陷入 Nachos 内核等等。

Machine 类实现方法是在宿主机上分配两块内存分别作为虚拟机的寄存器和物理内存。运行用户程序时，先将用户程序从 Nachos 文件系统中读出，写入模拟的物理内存中，然后调用指令模拟模块对每一条用户指令解释执行。将用户程序的读写内存要求，转变为对物理内存地址的读写。Machine 类提供了单步调试用户程序的功能，执行一条指令后会自动停下来，让用户查看系统状态，不过这里的单步调试是汇编指令级的，需要读者对 R2/3000 指令比较熟悉。如果用户程序想使用操作系统提供的功能或者发出异常信号时，Machine 调用系统异常陷入功能，进入 Nachos 的核心部分。

Interrupt 类用来模拟硬件中断系统。在这个中断系统中，中断状态有开，关两种，中断类型有时钟中断、磁盘中断、控制台写中断、控制台读中断、网络发送中断以及网络接收中断。机器状态有用户态，核心态和空闲态。中断系统提供的功能有开/关中断，读/写机器状态，将一个即将发生中断放入中断队列，以及使机器时钟前进一步。

在 Interrupt 类中有一个记录即将发生中断的队列，称为中断等待队列。中断等待队列中每个等待处理的中断包含中断类型、中断处理程序的地址及参数、中断应当发生的时间等信息。一般是由硬件设备模拟程序把将要发生的中断放入中断队列。中断系统提供了一个模拟机器时钟，机器时钟在下列情况下前进（详见第二节对中断模块的分析）：

- 用户程序打开中断
- 执行一条用户指令
- 处理机没有进程正在运行

机器时钟前进时，中断处理的过程如下图 2.1 所示:

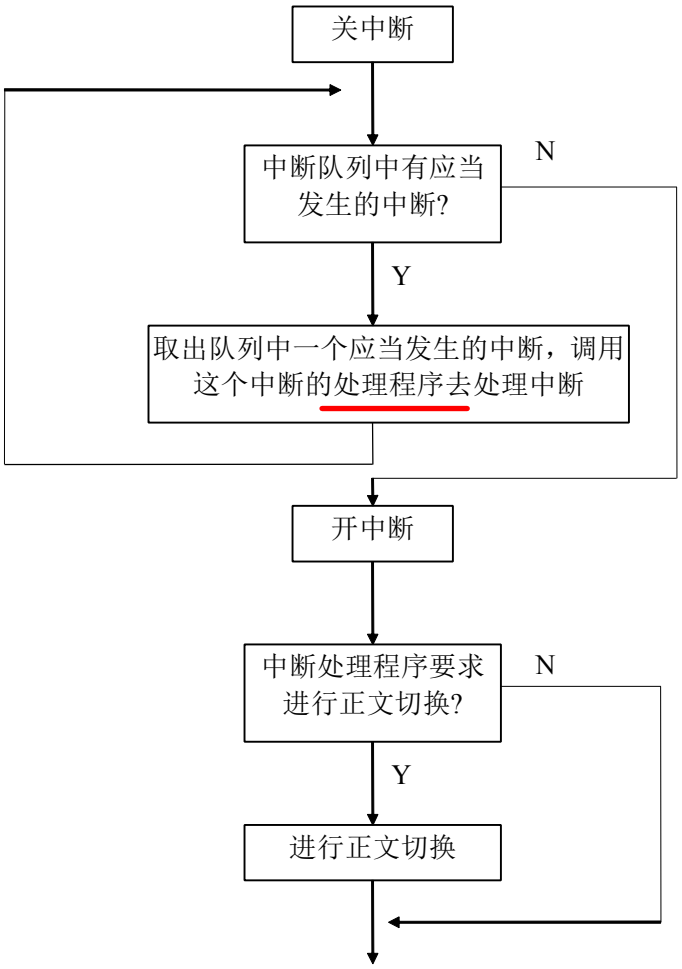


图 2.1　Nachos 中断处理时机

所以，在 Nachos 中只有在时钟前进时，才会检查是否有中断会发生，而 Nachos 模拟时钟前进的时机不是任意的，这样即使打开了中断，中断也不能在任意时刻发生。只有在模拟时钟前进的时候才能处理等待着的中断。通过以后的叙述我们可以看到，在执行非用户代码的大部分时间里，系统不会被中断。这意味着不正确的同步代码可能在这个硬件模拟环境下工作正常，而实际上在真正的硬件上是无法正确运行的。

在有些中断处理程序的最后可能要进行正文切换，可以通过调用 Interrupt 类的一个成员函数来要求时钟前进的时候进行正文切换。中断系统还提供关机的功能，当系统中没有正在运行的进程同时系统中没有除了时钟中断以外的其它中断时，Nachos 结束运行。

在这个中断系统基础上，Nachos 模拟了各种硬件设备，这些设备都是异步设备，依靠中断来与主机通信。

Timer 类模拟定时器。定时器每隔 X 个时钟周期就向 CPU 发一个时钟中断。它是时间片管理必不可少的硬件基础。它的实现方法是将一个即将发生的时钟中断放入中断队列，到了时钟中断应发生的时候，中断系统将处理这个中断，在中断处理的过程中又将下一个即将发生的时钟中断放入中断队列，这样每隔 X 个时钟周期，就有一个时钟中断发生。

由于 Nachos 是一个软件模拟的系统，有很多的随机事件需要通过一定的控制来实现。所以系统中在计算下一个时钟中断应发生的时间时，还加入了一些随机值，使得中断发生的时间间隔不确定，这样就与现实的定时器更相似。

Console 类模拟的是控制台设备。当用户程序向控制台写一个字符时，写程序立即返回，过了给定的时钟周期后 I/O 操作完成，控制台向 CPU 发一个控制台写中断。但是控制台是否有用户输入可供读取是随机的，所以控制台每隔给定的时钟周期向 CPU 发一个控制台读中断，周期性地发中断的方法与定时器的类似，即先计算下一个控制台读中断将发生的时间，然后将读中断放入中断队列，等待读中断的发生。读中断发生后，如果有用户输入的话，控制台读中断处理过程将控制台输入的字符放入字符缓冲区。当用户从控制台读字符时，把字符缓冲区的内容传给用户。控制台的读/写分别用两个文件来模拟。

Disk 类模拟了物理磁盘，它一次只能接受一个读写请求，当读写操作完成后向 CPU 发一个磁盘中断。该物理磁盘只有一个面，分为几个磁道，每道又分为几个扇区。每道的扇区数，每个扇区的存储容量都是固定的。磁盘的使用者可以读写指定的扇区，读写单位是一个扇区。模拟磁盘用宿主机文件系统中一个文件来实现，当用户发出读写请求时，Nachos 的处理过程如下:

1. 从模拟文件中读出数据或向模拟文件写入数据。
2. 计算磁盘操作需要的时间。磁盘操作时间 = 移动磁头寻道的时间 + 旋转到读写扇区的时间 + 数据传送的时间。
3. 将一个磁盘读/写中断放入中断队列，因为中断是在操作完成后发生的。所以，中断发生时间 = 当前时间 + 磁盘操作时间。

每个 Nachos 运行时是宿主机上的一个进程，如果在宿主机上运行多个 Nachos 进程，这些 Nachos 进程可以组成一个网络，而每个 Nachos 进程就是一个网络节点。Network 类模拟了一个网络节点。这个网络节点可以把报文发送到网络的其他节点上。报文的长度固定，Nachos 模拟了在现实网络中时常发生的报文丢失的情况；但是报文中的内容不会在网络传送中被修改破坏。每个网络节点都有全网络唯一的“地址”，报文传送的起始节点、目的节点都是由这个“地址”表示。

报文在网络中的传递是用通过 Socket（套接口）来实现的。每个节点还有一个可靠性系数，用来模拟报文从这个节点发出后丢失的概率。Network 的实现与控制台类似，每隔一定的时钟周期，就产生一个网络接收中断，网络接收中断处理过程是:

1. 将下一个网络接收中断放入中断队列以实现中断的周期性发生。
2. 如果报文缓冲区中已有报文，则返回。
3. 读取套接口，如果没有报文，则返回。
4. 读取报文，把它放入报文缓冲区。
5. 调用本节点自定义的接收处理函数。

在现有实现中，报文缓冲区只能存放一个报文，有可能因为报文缓冲区满而造成报文丢失（上面第 2 行），可以多设几个报文缓冲区来减少丢失的可能性。

Network 类提供了让网络用户读取已经收到的报文的成员函数，当报文缓冲区为空时，它返回空，否则从报文缓冲区读出报文，并将报文缓冲区清空，返回刚读出的报文。

报文发送的过程是:

1. 将网络发送中断放入中断队列。

2. 产生一个随机数。
3. 如果这个随机数大于网络的可靠性系数，则不发送报文 (用来模拟报文丢失)，否则通过套接口将报文发送出去。

从以上的叙述中可以看出，中断系统成为整个 Nachos 虚拟机的基础，其它的模拟硬件设备都是建立在中断系统之上的。在此之上，加上 Machine 类模拟的指令解释器，可以实现 Nachos 的线程管理、文件系统管理、虚拟内存、用户程序和网络管理等所有操作系统功能。图 2.2 展示了 Nachos 系统的整体结构。

图 2.2　Nachos 系统的整体结构

第二节　机器模拟的实现

1. Sysdep模块分析（文件sysdep.cc sysdep.h）

Nachos 的运行环境可以是多种操作系统，由于每种操作系统所提供的系统调用或函数调用在形式和内容上可能有细微的差别。sysdep 模块的作用是屏蔽掉这些差别。

1.1 PoolFile 函数

语法：　bool PoolFile (int fd)
参数：　fd:　文件描述符，也可以是一个套接字 (socket)
功能：　测试一个打开文件 fd 是否有内容可以读，如果有则返回 TRUE，否则返回 FALSE。当 Nachos 系统处于 IDLE 状态时，测试过程有一个延时，也就是在一定时间范围内如果有内容可读的话，同样返回 TRUE。
实现：　通过 select 系统调用。
返回：　打开文件是否有内容供读取。

1.2 OpenForWrite 函数

语法：　int OpenForWrite (char *name)
参数：　name:　文件名
功能：　为写操作打开一个文件。如果该文件不存在，产生该文件；如果该文件已经存在，则将该文件原有的内容删除。
实现：　通过 open 系统调用。
返回：　打开的文件描述符。

1.3 OpenForReadWrite 函数

语法：　int OpenForReadWrite (char *name, bool crashOnError)
参数：　name:　文件名
crashOnError: crash 标志
功能：　为读写操作打开一个文件。当 crashOnError 标志设置而文件不能读写打开时，系统出错退出。
实现：　通过 open 系统调用。
返回：　打开的文件描述符。

1.4 Read 函数

语法：　void Read (int fd, char *buffer, int nBytes)
参数：　fd:　打开文件描述符
buffer:　读取内容的缓冲区
nBytes:　需要读取的字节数
功能：　从一个打开文件 fd 中读取 nBytes 的内容到 buffer 缓冲区。如果读取失败，系统退出。
实现：　通过 read 系统调用。
返回：　无。
注意：这和系统调用 read 不完全一样。read 系统调用返回的是实际读出的字节数，而 Read 函数则必须读出 nBytes，否则系统将退出。如果需要使用同 read 系统调用相对应的函数，请用 ReadPartial。

1.5 ReadPartial 函数

语法：　int ReadPartial (int fd, char *buffer, int nBytes)

参数：　fd:　　打开文件描述符
　　buffer:　读取内容的缓冲区
　　nBytes:　需要读取的最大字节数

功能：　从一个打开文件 fd 中读取 nBytes 的内容到 buffer 缓冲区。

实现：　通过 read 系统调用。

返回：　实际读出的字节数。

1.6 WriteFile 函数

语法：　void WriteFile (int fd, char *buffer, int nBytes)

参数：　fd:　　打开文件描述符
　　buffer:　需要写的内容所在的缓冲区
　　nBytes:　需要写的内容最大字节数

功能：　将 buffer 缓冲区中的内容写 nBytes 到一个打开文件 fd 中。

实现：　通过 write 系统调用。

返回：　无。
　　注意：这和系统调用 write 不完全一样。write 系统调用返回的是实际写入的字节数，而 WriteFile 函数则必须写入 nBytes，否则系统将退出。

1.7 Lseek 函数

语法：　void Lseek (int fd, int offset, int whence)

参数：　fd:　文件描述符
　　offset:　偏移量
　　whence:　指针移动的起始点

功能：　移动一个打开文件的读写指针，含义同 lseek 系统调用；出错则退出系统。

实现：　通过 lseek 系统调用。

返回：　无。

1.8 Tell 函数

语法：　int Tell (int fd)

参数：　fd:　文件描述符

功能：　指出当前读写指针位置

实现：　通过 lseek 系统调用。

返回：　返回当前指针位置。

1.9 Close 函数

语法：　void Close (int fd)

参数：　fd:　文件描述符

功能：　关闭当前打开文件 fd，如果出错则退出系统。

实现：　通过 close 系统调用。

返回：　无。

1.10 Unlink 函数

语法：　bool Unlink (char *name)

参数：　name:　文件名

功能：　删除文件。
实现：　通过 unlink 系统调用。
返回：　删除成功，返回 TRUE；否则返回 FALSE。

1.11 OpenSocket 函数

语法：　int OpenSocket ()
参数：　无
功能：　申请一个 socket。
实现：　通过 socket 系统调用。
其中 AF_UNIX 参数说明使用 UNIX 内部协议。（Nachos 是用 SOCKET 文件来模拟网络节点，所以采用 UNIX 内部协议。向该文件读写内容分别代表从该节点读取内容和向该网络节点发送内容）
SOCK_DGRAM 参数说明采用无连接定长数据包型的数据链路。
返回：　申请到的 socket ID。

1.12 CloseSocket 函数

语法：　void CloseSocket (int sockID)
参数：　sockID:　socket 标识
功能：　释放一个 socket。
实现：　通过 close 系统调用。
返回：　无。

1.13 AssignNameToSocket 函数

语法：　void AssignNameToSocket(char *socketName, int sockID)
参数：　socketName:　socket 文件名
sockID:　socket 标识
功能：　将一个文件名和一个 socket 标识联系起来，于是将一个 SOCKET 文件同一个 Nachos 进程连接起来，使宿主机上该 Nachos 进程成为一个网络节点。
实现：　通过 bind 系统调用。
返回：　无。

1.14 DeAssignNameToSocket 函数

语法：　void DeAssignNameToSocket(char *socketName)
参数：　socketName:　socket 文件名
功能：　将一个文件名删除，实际上是和相应的 socket 标识脱离关系。
实现：　通过 unlink 系统调用。
返回：　无。

1.15 PoolSocket 函数

语法：　bool PoolSocket (int sockID)
参数：　socketID:　socket 标识
功能：　查询一个 socket 是否有内容可以读取。
实现：　调用 PoolFile。在 UNIX 中 socket 标识和普通的文件标识没有本质的区别，可以采用

相同的方式操作；Nachos 中的网络收发信息的模拟实际上是文件操作。
返回：　socket 中有内容，返回 TRUE；否则返回 FALSE。

1.16 ReadFromSocket 函数

语法：　void ReadFromSocket (int sockID, char *buffer, int packetSize)
参数：　socketID:　　socket 标识
　　buffer:　　读取内容的暂存空间
　　packetSize:　　读取数据包的大小
功能：　从一个 socket 标识中读取 packetSize 大小的数据包，放在 buffer 缓冲中。
实现：　通过 recvfrom 系统调用。
返回：　无。

1.17 SendToSocket 函数

语法：　void SendToSocket (int sockID, char *buffer, int packetSize, char *toName)
参数：　socketID:　　socket 标识
　　buffer:　　发送内容的暂存空间
　　packetSize:　　发送数据包的大小
　　toName:　　要接收数据包的 Nachos 虚拟机模拟网络文件的文件名
功能：　向 socket 标识中发送 packetSize 大小的数据包。
实现：　通过 sendto 系统调用。
　　Nachos 的网络处理中断程序会检查和自己相连的模拟网络 SOCKET 文件中是否有内容可读。
　　当 Nachos 需要向其它节点发送信息时，需要指明其它节点的地址，实际上就是和其它节点相连的模拟网络 SOCKET 文件名。
返回：　无。

1.18 CallOnUserAbort 函数

语法：　void CallOnUserAbort (VoidNoArgFunctionPtr func)
参数：　func:　　函数指针
功能：　设定一个函数，在用户强制退出系统时调用。
实现：　通过 signal 系统调用。
返回：　无。

1.19 Delay 函数

语法：　void Delay (int seconds)
参数：　seconds:　需要延迟的秒数
功能：　系统延迟一定的时间。
实现：　通过 sleep 系统调用。
返回：　无。

1.20 Abort 函数

语法：　void Abort ()
参数：　无
功能：　退出系统 (非正常退出)。
实现：　通过 abort 系统调用。
返回：　无。

1.21 Exit 函数

语法：　void Exit (int exitCode)
参数：　exitCode: 向系统的返回值
功能：　退出系统。
实现：　通过 exit 系统调用。
返回：　无。

1.22 RandomInit 函数

语法：　void RandomInit (unsigned seed)
参数：　seed:　　随机数产生魔数
功能：　初始化随机数发生器。
实现：　通过 srand 系统调用。
返回：　无。

1.23 Random 函数

语法：　int RandomInit ()
参数：　无
功能：　产生一个随机整数。
实现：　通过 rand 系统调用。
返回：　产生的随机整数。

1.24 AllocBoundedArray 函数

语法：　char * AllocBoundedArray (int size)
参数：　size: 需要申请的空间大小
功能：　申请一个受保护的存储空间。
实现：　通过 mprotect 的系统调用，申请一块比 size 较大的空间，并且在要申请空间两头区域的属性设置成不可访问；当用户使用不当时（使用到受保护范围之外时），系统会接收到 SIGSEGV 信号。不是每个操作系统都支持这样的内存申请，如果支持的话，对监测内存的使用是否恰当非常有用。
返回：　申请成功后指针，该指针指向可以访问的申请空间，而不是指向受限区域的开始。

1.25 DeallocBoundedArray 函数

语法：　void DeallocBoundedArray (char *ptr, int size)
参数：　ptr:　要释放空间的指针
　　size: 申请的空间大小
功能：　将受保护的存储空间释放。
实现：　通过 mprotect 系统调用；释放的空间包括头尾受限区域，所以必须知道原来申请区域的大小。
返回：　无。

2. 中断模块分析（文件interrupt.cc interrupt.h）

中断模块的主要作用是模拟底层的中断机制。可以通过该模拟机制来启动和禁止中断(SetLevel)；该中断机制模拟了 Nachos 系统需要处理的所有的中断，包括时钟中断、磁盘中断、终端读/终端写中断以及网络接收/网络发送中断。

中断的发生总是有一定的时间。比如当向硬盘发出读请求，硬盘处理请求完毕后会发生中断；在请求和处理完毕之间需要经过一定的时间。所以在该模块中，模拟了时钟的前进。为了实

现简单和便于统计各种活动所占用的时间起见，Nachos 规定系统时间在以下三种情况下前进：

- 执行用户态指令
  执行用户态指令，时钟前进是显而易见的。我们认为，Nachos 执行每条指令所需时间是固定的，为一个时钟单位(Tick)。
- 重新打开中断
  一般系统态在进行中断处理程序时，需要关中断。但是中断处理程序本身也需要消耗时间，而在关闭中断到重新打开中断之间无法非常准确地计算时间，所以当中断重新打开的时候，加上一个中断处理所需时间的平均值。
- 就绪队列中没有进程
  当系统中没有就绪进程时，系统处于 Idle 状态。这种状态可能是系统中所有的进程都在等待各自的某种操作完成。也就是说，系统将在未来某个时间发生中断，到中断发生的时候中断处理程序将进行中断处理。在系统模拟中，有一个中断等待队列，专门存放将来发生的中断。在这种情况下，可以将系统时间直接跳到中断等待队列第一项所对应的时间，以免不必要的等待。

当前面两种情况需要时钟前进时，调用 OneTick 方法。OneTick 方法将系统态和用户态的时间分开进行处理，这是因为用户态的时间计算是根据用户指令为单位的；而在系统态，没有办法进行指令的计算，所以将系统态的一次中断调用或其它需要进行时间计算的单位设置为一个固定值，假设为一条用户指令执行时间的 10 倍。

虽然 Nachos 模拟了中断的发生，但是毕竟不能与实际硬件一样，中断发生的时机可以是任意的。比如当系统中没有就绪进程时，时钟直接跳到未处理中断队列的第一项的时间。这实际情况下，系统处于 Idel 状态到中断等待队列第一项发生时间之间，完全有可能有其它中断发生。由于中断发生的时机不是完全随机的，所以在 Nachos 系统中运行的程序，不正确的同步程序也可能正常运行，我们在此需要密切注意。

时钟中断每隔一段时间就会出现。

Nachos 线程运行有三种状态：

- Idle 状态
  系统 CPU 处于空闲状态，没有就绪线程可以运行。如果中断等待队列中有需要处理的除了时钟中断以外的中断，说明系统还没有结束，将时钟调整到发生中断的时间，进行中断处理；否则认为系统结束所有的工作，退出。
- 系统态
  Nachos 执行系统程序。Nachos 虽然模拟了虚拟机的内存，但是 Nachos 系统程序本身的运行不是在该模拟内存中，而是利用宿主机的存储资源。这是 Nachos 操作系统同真正操作系统的重要区别。
- 用户态
  系统执行用户程序。当执行用户程序时，每条指令占用空间是 Nachos 的模拟内存（见第五章）。

Nachos 需要处理的中断种类有：

TimerInt:	时钟中断
DiskInt:	磁盘（读/写）中断
ConsoleWriteInt:	终端写中断
ConsoleReadInt:	终端读终端
NetworkSentInt:	网络发送中断
NetworkRecvInt:	网络接收中断

中断等待队列是 Nachos 虚拟机最重要的数据结构之一，它记录了当前虚拟机可以预测的将在未来发生的所有中断。当系统进行了某种操作可能引起未来发生的中断时，如磁盘的写入、向网络写入数据等都会将中断插入到中断等待队列中；对于一些定期需要发生的中断，如时钟中断、终端读取中断等，系统会在中断处理后将下一次要发生的中断插入到中断等待队列中。中断的插入过程是一个优先队列的插入过程，其优先级是中断发生的时间，也就是说，先发生的中断将优先得到处理。

当时钟前进或者系统处于 Idle 状态时，Nachos 会判断中断等待队列中是否有要发生的中断，如果有中断需要发生，则将该中断从中断等待队列中删除，调用相应的中断处理程序进行处理。图 2.3 是中断等待队列的操作示意图。

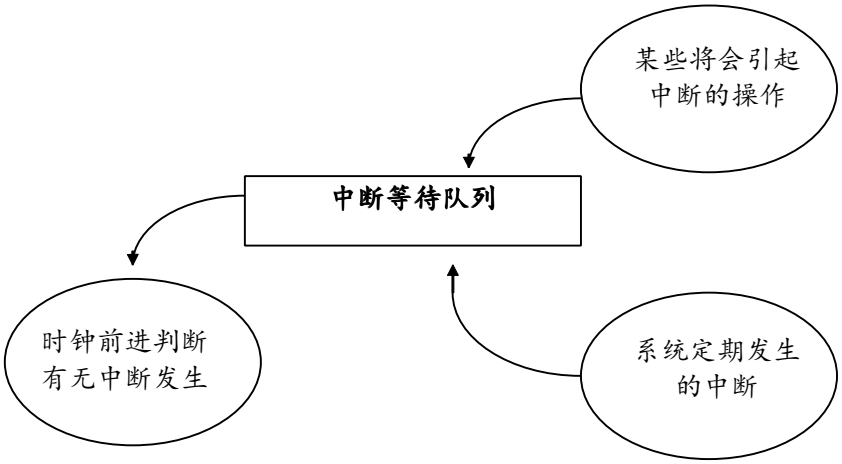


图 2.3　对中断等待队列的操作

中断处理程序是在某种特定的中断发生时被调用，中断处理程序的作用包括可以在现有的模拟硬件的基础上建立更高层次的抽象。比如现有的模拟网络是有丢失帧的不安全网络，在中断处理程序中可以加入请求重发机制来实现一个安全网络。

2.1 PendingInterrupt类

```
class PendingInterrupt {
  public:
    PendingInterrupt (VoidFunctionPtr func, int param, int time, IntType kind);
                    VoidFunctionPtr handler;      // 中断对应的中断处理程序
    int arg;                                      // 中断处理程序的参数
    int when;                                     // 中断发生的时机
    IntType type;                                 // 中断的类型，供调试用
};
```

这个类定义了一个中断等待队列中需要处理的中断。为了方便起见，所有类的数据和成员函数都设置为 public 的，不需要其它的 Get 和 Set 等存取内部数据的函数。初始化函数就是为对应的参数赋值。

2.2 Interrupt类

```
class Interrupt {
  public:                   // 以下函数是 Interrupt 的对外接口
    Interrupt();            // 初始化中断模拟
    ~Interrupt();               // 终止中断模拟

    IntStatus SetLevel(IntStatus level);
                            // 开关中断，并且返回之前的状态
    void Enable();              // 开中断
    IntStatus getLevel() {return level;}
                            // 取回当前中断的开关状态
    void Idle();            // 当进程就绪队列为空时，执行该函数
    void Halt();                // 退出系统，并打印状态
    void YieldOnReturn();           // 设置中断结束后要进行进程切换的标志
    MachineStatus getStatus() { return status; }
                            // 返回系统当前的状态
  VOID SETSTATUS(MACHINESTATUS ST) { STATUS = ST; }
                            // 设置系统当前的状态
    void DumpState();           // 调试当前中断队列状态用

    void Schedule(VoidFunctionPtr handler, int arg, int when, IntType type);
                            // 在中断等待队列中，增加一个等待中断
    void OneTick();             // 模拟时钟前进
  private:                  // 以下是内部数据和内部处理方法
    IntStatus level;            // 中断的开关状态
    List *pending;              // 当前系统中等待中断队列
    bool inHandler;             // 是否正在进行中断处理标志
    bool yieldOnReturn;         // 中断处理后是否需要正文切换标志
    MachineStatus status;           // 当前虚拟机运行状态
    bool CheckIfDue(bool advanceClock);
                            // 检查当前时刻是否有要处理的中断
    void ChangeLevel(IntStatus old, IntStatus now);
                            // 改变当前中断的开关状态，但是不前进模拟时钟
};
```

其中，Schedule 和 OneTick 两个方法虽然标明是 public 的，但是除了虚拟机模拟部分以外的其它类方法是不能调用这两个方法的。将它们设置成 public 的原因是因为虚拟机模拟的其它类方法需要直接调用这两个方法。

2.2.1 内部使用方法

2.2.1.1 CheckIfDue 方法

语法：　bool CheckIfDue (bool advanceClock)
参数：　advanceClock: 时钟前进标志
在前面时钟前进时机的论述时说到，当系统处于 Idle 状态时，时钟直接跳到等待中断队列第一项规定的时间。是否需要这样做，由 advanceClock 标志来决定。
功能：　测试当前等待中断队列中是否要有中断发生，并根据不同情况作出不同处理。
实现：　1　在等待处理的中断队列中取出第一项（最早会发生的中断）
2　如果不存在任何中断，返回 FALSE。
3　如果该中断的发生时机没有到：
3.1　如果 advanceClock 没有设置，将取出的中断放回原处，等待将来处理。
3.2　如果 advanceClock 设置了，系统时间 totalTicks 跳到中断将要发生的时间。说明中断马上就要发生。
4　如果当前的状态是 Idle 态，而且取出的中断是时钟中断，同时等待中断队列中没有其它的中断，意味着系统将退出。但是系统的退出不在这里处理，而是将该中断放回原处，等待以后处理；并返回 FALSE。
5　中断发生
5.1　inHandler 标志设置，说明正在进行中断处理程序。
5.2　status 设置成系统态；很显然，中断处理程序是系统态的。
5.3　进行中断处理程序的处理，直到处理结束
5.4　恢复虚拟机的 status 和 inHandler 标志。
返回：　如果有需要处理的中断，返回 TRUE；否则返回 FALSE。

2.2.1.2 ChangeLevel 方法

语法：　void ChangeLevel (IntStatus old, IntStatus now)
参数：　old:　原有的中断开关状态
now:　需要设置的中断开关状态
功能：　将当前的中断开关状态设置成 now，不对系统时钟产生任何影响。
实现：　赋值实现。
返回：　无。

2.2.2 内部使用函数

2.2.2.1 PrintPending 函数

语法：　void PrintPending (int arg)
参数：　arg: 未处理的中断指针（被强制类型转换成整数表示）
功能：　调试时打印出当前中断的一些信息，包括中断类型和发生的时机。
实现：　打印信息。
返回：　无。

2.2.3 对外接口

2.2.3.1 构造方法

语法：　Interrupt ()
参数：　无
功能：　构造一个中断机制，设置关中断状态、清除 inHandle 标志、系统初始为系统态；同时初始化等待中断队列。

实现：　赋值实现。
返回：　无。

2.2.3.2 析构方法

语法：　~Interrupt ()
参数：　无
功能：　释放等待中断队列的空间。
实现：　循环释放等待中断队列中每个中断的空间。
返回：　无。

2.2.3.3 SetLevel方法

语法：　Intstatus SetLevel(IntStatus now)
参数：　now:　　需要设置的中断开关状态
功能：　将当前的中断开关状态设置成 now；当开关状态从 IntOff 转变为 IntOn 时，时钟前进。
实现：　调用 ChangeLevel 内部方法。
返回：　机器原有的中断开关状态。

2.2.3.4 Enable方法

语法：　void Enable ()
参数：　无
功能：　开中断。用在 ThreadRoot (switch.s) 中，在启动一个线程之前首先需要开中断。
实现：　调用 ChangeLevel 内部方法。
返回：　无。

2.2.3.5 OneTick方法

语法：　void OneTick ()
参数：　无
功能：　时钟前进一个单位（系统态时间单位是用户态时间单位的 10 倍。用户态执行一条用户指令花费一个用户态时间单位；当开中断时前进一个系统态时间单位，系统态时间单位是一个平均值）
实现：
1. 根据当前状态为用户态或是系统态时钟分别前进一个用户态时间单位或系统态时间单位，并且对 Nachos 运行的各项时间（用户态时间、系统态时间）进行统计。
2. 检查当前时刻是否有中断发生；如果有，进行中断处理
3. 如果 yieldOnReturn 标志设置，作进程切换

返回：　无。

2.2.3.6 YieldOnReturn方法

语法：　void YieldOnReturn ()
参数：　无
功能：　设置 yieldOnReturn 标志
该方法必须在中断处理程序中调用，比如时钟中断处理的最后可能会引起进程切换就需要调用该方法。
实现：　赋值实现。
返回：　无。

2.2.3.7 Idle方法

语法：　void Idle ()

参数：　无
功能：　系统处于 Idle 状态时调用该方法 (见 Thread 类 Sleep 方法)，检查当前等待中断队列中是否有待处理的中断，如果有处理该中断。
实现：　检查等待中断队列是否有中断要发生
1. 如果有
1.1. 将系统时钟调整到第一个待处理中断的发生时间，处理该中断。（调用 CheckIfDue，并将 advanceClock 设置为 TRUE）
1.2. 处理在新的时刻其它需要发生的中断。
2. 如果没有，退出系统。
（当系统中除了时钟中断之外没有其它中断时，退出 Nachos。但是如果启动了终端设备或者网络，这样的情况不会发生，因为系统会一直等待终端输入或网络数据的到达）
返回：　无。

2.2.3.8 Halt方法

语法：　Void Halt ()
参数：　无
功能：　系统退出。
实现：　打印系统各项统计信息，释放占用空间后退出。
返回：　无。

2.2.3.9 Schedule方法

语法：　void Schedule (VoidFunctionPtr handler, int arg, int fromNow, IntType type)
参数：　handler:　中断处理函数
arg:　中断处理函数的参数
fromNow:　中断发生的时刻和现在时刻的差值
type:　中断的类型
功能：　将一个中断插入等待处理中断队列。
实现：　调用 List.SoftInsert()方法。（见第三章第三节第一部分）
返回：　无。

2.2.3.10 DumpState方法

语法：　void DumpState ()
参数：　无
功能：　打印出当前等待中断队列中的所有中断的状态。
实现：　调用 PrintPending 函数。
返回：　无

3. 时钟中断模块分析（文件timer.cc timer.h）

该模块的作用是模拟时钟中断。Nachos 虚拟机可以如同实际的硬件一样，每隔一定的时间会发生一次时钟中断。这是一个可选项，目前 Nachos 还没有充分发挥时钟中断的作用，只有在 Nachos 指定线程随机切换时，（Nachos -rs 参数，见线程管理部分 Nachos 主控模块分析）启动时钟中断，在每次的时钟中断处理的最后，加入了线程的切换。实际上，时钟中断在线程管理中的作用远不止这些，时钟中断还可以用作：

- 线程管理中的时间片轮转法的时钟控制，（详见线程管理系统中的实现实例中，对线程调度的改进部分）不一定每次时钟中断都会引起线程的切换，而是由该线程是否的时间片是否已经用完来决定。
- 分时系统线程优先级的计算（详见线程管理系统中的实现实例中，对线程调度的改进部

分）

- 线程进入睡眠状态时的时间计算
  可以通过时钟中断机制来实现 sleep 系统调用，在时钟中断处理程序中，每隔一定的时间对定时睡眠线程的时间进行一次评估，判断是否需要唤醒它们。

Nachos 利用其模拟的中断机制来模拟时钟中断。时钟中断间隔由 TimerTicks 宏决定（100 倍 Tick 的时间）。在系统模拟时有一个缺陷，如果系统就绪进程不止一个的话，每次时钟中断都一定会发生进程的切换（见 system.cc 中 TimerInterruptHandler 函数）。所以运行 Nachos 时，如果以同样的方式提交进程，系统的结果将是一样的。这不符合操作系统的运行不确定性的特性。所以在模拟时钟中断的时候，加入了一个随机因子，如果该因子设置的话，时钟中断发生的时机将在一定范围内是随机的。这样有些用户程序在同步方面的错误就比较容易发现。但是这样的时钟中断和真正操作系统中的时钟中断将有不同的含义。不能象真正的操作系统那样通过时钟中断来计算时间等等。是否需要随机时钟中断可以通过设置选项(-rs)来实现。

Timer 类定义和实现如下所示：

```
class Timer {
  public:
    Timer (VoidFunctionPtr timerHandler, int callArg, bool doRandom);
                                  // 初始化方法
    ~Timer() {}                   // 析构方法
    void TimerExpired();          // 当时钟中断发生时调用
  private:
    bool randomize;               // 是否需要随机时钟中断标志
    VoidFunctionPtr handler;      // 时钟中断处理函数
    int arg;                      // 时钟中断处理函数参数
    int TimeOfNextInterrupt();    // 计算下一次时钟中断的发生时间
};
```

```
1.  static void TimerHandler(int arg)
2.  { Timer *p = (Timer *)arg; p->TimerExpired(); }

3.  Timer::Timer(VoidFunctionPtr timerHandler, int callArg, bool doRandom)
4.  {
5.      randomize = doRandom;
6.      handler = timerHandler;
7.      arg = callArg;
8.      interrupt->Schedule(TimerHandler, (int) this, TimeOfNextInterrupt(),
    TimerInt);
9.  }
```

```
10. void Timer::TimerExpired()
11. {
12.     interrupt->Schedule(TimerHandler, (int) this, TimeOfNextInterrupt(),
    TimerInt);
13.     (*handler)(arg);
14. }

15. int  Timer::TimeOfNextInterrupt()
16. {
17.     if (randomize)
18.         return 1 + (Random() % (TimerTicks * 2));
19.     else
20.         return TimerTicks;
21. }
```

Timer 类的实现很简单，当生成出一个 Timer 类的实例时，就设计了一个模拟的时钟中断。这里考虑的问题是：怎样实现定期发生时钟中断？

在 Timer 的初始化函数中，借用 TimerHandler 内部函数（见第 1 行）。为什么不直接用初始化函数中的 timerHandler 参数作为中断处理函数呢？因为如果直接使用 timerHandler 作为时钟中断处理函数，第 8 行是将一个时钟中断插入等待处理中断队列，一旦中断时刻到来，立即进行中断处理，处理结束后并没有机会将下一个时钟中断插入到等待处理中断队列。TimerHandler 内部函数正是处理这个问题。当时钟中断时刻到来时，调用 TimerHandler 函数，其调用 TimerExpired 方法，该方法将新的时钟中断插入到等待处理中断队列中，然后再调用真正的时钟中断处理函数。这样 Nachos 就可以定时的收到时钟中断。

那么为什么不将 TimerExpired 方法作为时钟中断在 Timer 的初始化函数中调用呢？这是由于 C++语言不能直接引用一个类内部方法的指针，所以借用 TimerHandler 内部函数。这也是 TimerExpired 必须设计成 public 的方法的原因，因为它要被 TimerHandler 调用。

这样的方法不仅仅在 Timer 模拟时钟中断中用到，所有需要定期发生的中断都可以采用这样的方法，如 Nachos 需要定期地检查是否有终端的输入、网络是否有发给自己的报文等都是用这种方式实现。详见 network.cc 以及 console.cc。

TimeOfextInterrupt()方法的作用是计算下一次时钟中断发生的时机，如果需要时钟中断发生的时机是随机的，可以在 Nachos 命令行中设置 –rs 选项。这样，Nachos 的线程切换的时机将会是随机的。但是此时时钟中断则不能作为系统计时的标准了。

4. 终端设备模块分析（文件console.cc console.h）

该模块的作用是模拟实现终端的输入和输出。包括两个部分，即键盘的输入和显示输出。终端输入输出的模拟是异步的，也就是说当发出终端的输入输出请求后系统即返回，需要等待中断发生后才是真正完成了整个过程。

Console 类定义和实现如下所示：

```
class Console {
  public:
    Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail,
            VoidFunctionPtr writeDone, int callArg);
                                  // 初始化方法
                                  // readAvail:     键盘读入中断处理函数
                                  // WriteDonel: 显示输出中断处理函数
    ~Console();                   // 析构方法
    void PutChar(char ch);              // 将字符 ch 向终端上输出
    char GetChar();               // 从终端上读取一个字符
    void WriteDone();             // 写终端中断时调用
    void CheckCharAvail();              // 读终端中断时调用
  private:
    int readFileNo;               // 模拟键盘输入的文件标识符
    int writeFileNo;              // 模拟显示器的文件标识符
    VoidFunctionPtr writeHandler;       // 写中断处理函数
    VoidFunctionPtr readHandler;        // 读中断处理函数
    int handlerArg;               // 中断处理函数参数
    bool putBusy;                 // 正在写终端标志
    char incoming;                // 读取终端字符的暂存空间
};
```

Nachos 的终端模拟借助了两个文件，即在生成函数中的 readFile 和 writeFile。这两个文件分别模拟键盘输入和屏幕显示。当 readFile 为 NULL 时，Nachos 以标准输入作为终端输入；当 writeFile 为 NULL 时，Nachos 以标准输出作为终端输出。

两个内部函数 ConsoleReadPoll 以及 ConsoleWriteDone 的作用同 Timer 模拟中的 TimerHandler 内部函数。Nachos 需要以一定的时间间隔检查终端是否有字符供读取；当然在这些中断处理程序中，还包括了一些统计的工作。

系统的终端操作有严格的工作顺序，对读终端来说：

CheckCharAvail -> GetChar -> CheckCharAvail -> GetChar ->...

系统通过定期的读终端中断来判断终端是否有内容供读取，如果有则读出；如果没有，下一次读终端中断继续判断。读出的内容将一直保留到 GetChar 将其读走。

对写终端来说：

PutChar -> WriteDone -> PutChar -> WriteDone -> ...

系统发出一个写终端命令 PutChar，模拟系统将直接向终端输出文件写入要写的内容，但是对 Nachos 来说，整个写的过程并没有结束，只有当写终端中断来到后整个写过程才算结束。

5. 磁盘设备模块分析（文件disk.cc disk.h）

磁盘设备模拟了一个物理磁盘。Nachos 用宿主机中的一个文件来模拟一个单面物理磁盘，该磁盘由道组成，每个道由扇区组成，而每个扇区的大小是固定的。和实际的物理磁盘一样，Nachos 以扇区为物理读取/写入的最小单位，每个扇区有唯一的扇区地址，具体的计算方法是：

```
track * SectorsPerTrack + offset
```

该物理磁盘是一个异步的物理磁盘，同终端设备和网络设备一样，当系统发出读磁盘的请求，立即返回，只有具体的磁盘终端到来的时候，整个过程才算结束。

Disk 类的定义和实现如下所示：

```
class Disk {
  public:
    Disk(char* name, VoidFunctionPtr callWhenDone, int callArg);
                          // 初始化方法，生成一个物理磁盘
                          // callWhenDone：磁盘终端调用函数
    ~Disk();              // 析构方法
    void ReadRequest(int sectorNumber, char* data);
                          // 读取一个扇区的内容，放入 data 缓冲区
    void WriteRequest(int sectorNumber, char* data);
                          // 将 data 缓冲区中的内容写入一个扇区
    void HandleInterrupt();       // 当磁盘中断时调用
  private:
    int fileno;                   // 模拟磁盘的文件标识符
    VoidFunctionPtr handler;      // 磁盘中断处理函数
    int handlerArg;               // 磁盘中断处理函数参数
    bool active;          // 磁盘正在传送数据标志
    int lastSector;               // 最后一次磁盘动作所在的扇区
    int bufferInit;               // trackbuffer 中内容的调用时间
    int ComputeLatency(int newSector, bool writing);
                          // 计算磁头读写需要的延迟时间
    int TimeToSeek(int newSector, int *rotate);
                          // 计算磁头从旧扇区移动新扇区所需要的时间
    int ModuloDiff(int to, int from);//  计算两个磁道的距离
    void UpdateLast(int newSector); //  修改最后访问的磁道号
};
```

Nachos 对物理磁盘的模拟和对网络、终端等的模拟非常类似，所采用的手段也很类似。这里就不详细叙述，需要说明的有以下几点：

1. 和其它的模拟不同的是，每次磁盘请求到磁盘中断发生之间的时间间隔是不一样的，这取决于两次磁盘访问磁道和扇区的距离。
2. 为了和实际情况更加接近，Nachos 的物理磁盘设置有 trackbuffer 高速缓冲区。其中存放的是最后一次访问的磁道中的所有内容。这样如果相邻的两次磁盘读访问是同一个磁道，可以直接从 trackbuffer 中读取扇区的内容，而不必要进行真正的磁盘读访问，当然写磁盘则多了向 trackbuffer 写入的步骤。
3. 磁盘模拟文件开头四个字节的值为 MagicNumber，其作用是为了不让 Nachos 磁盘模拟文件同其它文件混淆。

对于操作系统的实现来说，这部分内容并不是很重要。但是当我们认识到对于磁盘的访问会影响到系统的效率时，可能会重新设计文件系统，让系统不要在磁头的移动中无谓地浪费时间。比如在有些文件系统的设计中，采用了多 inode 区。目的是使一个文件的 inode 同文件内容所在的磁盘扇区比较接近，从而减少磁头移动的时间。

6. Nachos运行情况统计（文件stats.cc stats.h）

在本章的最后部分，我们要说明的是对 Nachos 运行情况进行统计的类 Statistics。这并不属于机器模拟的一部分，但是为了帮助读者了解自己设计的操作系统的各种运行情况。Statistics 类中包含的各种统计项是非常有价值的。

Statistics 类的定义和实现如下：

```
class Statistics {
  public:
    int totalTicks;                 // Nachos 运行的时间
    int idleTicks;                  // Nachos 在 Idle 态的时间
    int systemTicks;                // Nachos 在系统态运行的时间
    int userTicks;                  // Nachos 在用户态运行的时间
    int numDiskReads;               // Nachos 发出的读磁盘请求次数
    int numDiskWrites;              // Nachos 发出的写磁盘请求次数
    int numConsoleCharsRead;        // Nachos 读取的终端字符数
    int numConsoleCharsWritten;     // Nachos 输出的字符数
    int numPageFaults;              // 页转换出错陷入次数
    int numPacketsSent;             // 向网络发送的数据包数
    int numPacketsRecvd;            // 从网络接收的数据包数

    Statistics();                   // 初始化方法，将所有的值都置 0
    void Print();                   // 打印统计信息
};
```

第三章 线程管理系统

第一节　进程与线程

一、进程

1. 进程概念

进程是操作系统中最基本也是最重要的概念之一，它表示程序在给定数据集上的一次执行活动。通常情况下，一个计算机系统（无论是单机还是多机系统）同时存在的进程数大于计算机系统所有的 CPU 数。（由于 Nachos 模拟的是单机系统，所以以下都以单机系统为例）但是每个 CPU 在一个瞬时只能运行一个进程。所以 CPU 会在多个进程之间切换。在任一时刻，系统中有若干进程处在起点和终点之间，这些进程被认为是在运行着，这就是我们所说的**并发处理**。

系统运行过程中，通常有多个进程同时存在，它们各自执行的指令序列，对系统资源和服务的需求以及状态的变化往往互不相同、千变万化而难以预测。同时还可能接收到需要立即处理的中断信号。而中断信号发生的时间以及频繁程度与系统中许多经常变换着的不确定因素有关。所以每个进程在一种不可预测的次序中交替前进。操作系统内部动作的不可预测、不可重复就是**操作系统的不确定性**。

2. 进程的状态及状态变化

进程是程序的一次执行活动，它是一种动态的概念，而这种动态在宏观上表现为状态的变化。进程在运行中，有三种基本状态：

- 运行态：进程分配到处理机运行。
- 就绪态：进程已经可以在处理机上运行，只是暂时没有分配到处理机。
- 阻塞态：进程因等待某一个事件发生而暂时不能调度上处理机运行。

一个系统中的进程在一定条件下可以在这三种状态之间转换。一般有四种类型的转换。如图 3.1 所示：

- 运行态 -> 就绪态
  进程占用 CPU 运行了一段时间，但是没有运行结束。为使各就绪进程能比较平衡地共享 CPU，此时调度程序需要将其它就绪进程调度上处理机运行，于是原来占据处理机的进程成为就绪态，等待下一次被调度上处理机运行。
- 就绪态 -> 运行态
  进程处于就绪态，调度程序总是有机会将其调度上处理机，于是该进程从就绪态转为运行态，并从上一次运行的中断点继续运行。
- 运行态 -> 阻塞态
  进程运行过程中可能因等待某种事件发生而暂时停止，比如等待一次键盘事件或者磁盘输入输出。进程进入阻塞态时，调度程序会调度一个就绪态进程上处理机运行。
- 阻塞态 -> 就绪态
  当进程进入阻塞态之前等待发生的事件业已发生，则该进程从阻塞态转为就绪态，于是它可以再被调度上处理机继续运行。

除了个别进程外，一般进程都需要经历这三种状态，并在这三种状态中反复变换直至运行终止。

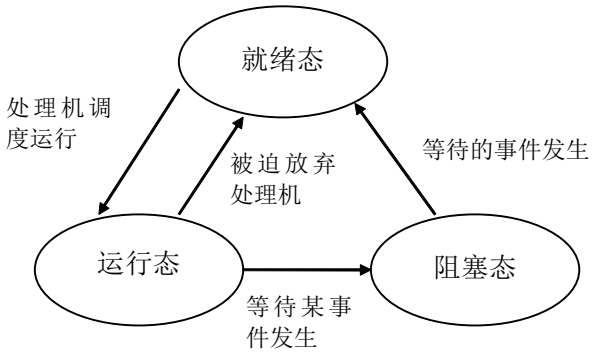


图 3.1　进程的状态转换

3. 进程调度

进程调度程序的优劣取决于其调度算法，在不同应用的操作系统中，调度算法的重点各不相同，这里给出一般进程调度的原则：

- 调度程序必须公平，确保每个进程都有机会使用系统的资源，不会出现由于一直分配不到资源而出现进程“饿死”的现象。
- 充分利用系统的各项资源，尤其是 CPU 资源。不会出现有进程处于就绪态，而 CPU 处于空闲状态的现象。
- 一定的系统吞吐率，在单位时间内执行完成的进程数越大越好。
- 合理的系统响应时间。对于有交互功能的进程，用户的等待时间要短。
- 能够反映用户的不同类型以及他们对有关作业运行优先程度的要求。
- 合理的系统开销。

进程调度分为非抢占式调度和抢占式调度。**非抢占式调度**就是进程在运行过程中不会切换到其它进程运行，除非其主动放弃处理机或者运行结束。由于进程的运行有不可预见性，有可能一个进程会占用处理机达几个小时，甚至一个编写错误的进程会一直占用处理机不放。为了确保没有一个进程单独运行的时间太长，几乎所有的计算机系统都内置了时钟，周期性地进行时钟中断，在每一次时钟中断时，由进程调度程序负责判断是否有就绪进程比正在运行的进程更加适合占用 CPU。如果有这类进程则进行进程调度，这样的调度称为**抢占式调度**。通用操作系统一般采用抢占式调度，这样才能体现以上所说的进程调度原则。

进程调度的算法一般有以下几种：

- **循环轮转调度**

  在循环轮转调度中，每个进程都被安排了一个运行时间段限制，叫做**时间片**。如果一个进程在其时间片结束时仍没有运行结束，CPU 便被抢占并被调度执行其它进程。如果进程在其时间片结束之前已经阻塞或完成，则操作系统当即切换 CPU 到其它进程运行。轮转调度比较简单，在系统中只需要维护一个就绪进程的列表。如图 3.2 所示：

  在图 3.2(a)中，当前运行的进程是 B 进程，当 B 进程运行的时间片用完，调度程序将紧接的 F 进程调度上处理机，成为当前运行进程，而 B 进程被调度到就绪进程表的最后。

(a)　　(b)

图 3.2　进程循环轮转调度

循环轮转调度中的一个重要问题是时间片的长短。由于进程切换需要一定的系统开销，包括保护和恢复现场等，将时间片设置过短会导致引起过多的进程切换而降低处理机效率；但是如果时间片设置过长，将引起用户响应时间比较长。在不同要求的操作系统中实现的时间片长短是不一样的。某些系统还实现了多时间片调度，以满足不同的需要

- **优先权调度**

  优先权调度法按照进程执行任务的轻重缓急，给每个进程一个调度优先权。系统在切换时，在所有就绪进程中选择优先权最高的进程调度上处理机运行。优先权调度法分成**静态优先权调度法**和**动态优先权调度法**。

  - 静态优先权调度法

    如果在创建进程时就确定了进程的优先权，而且在进程的运行过程中除设置外不会经常改变，那么这样的调度方法称为静态优先权调度法。静态优先权的确定方法有如下几种：

    1) 按进程的类型确定：一般系统进程的优先权高于用户进程；进程在核心态下运行时的优先权高于在用户态下运行的优先权；前台进程的优先权高于后台进程；实时性要求高的进程的优先权高于实时性要求低的进程的优先权。
    2) 按进程提交的时间次序确定：一般先提交的进程优先权较高。
    3) 按作业要求的资源类型和数量确定：一般要求资源较少的进程有较高的优先权，这样有助于提高系统的吞吐量。

  - 动态优先权调度法

    进程优先权在进程创建后如果不是固定的，而是根据系统中的运行状态不断变化的。这样的优先调度称为动态优先权调度。一般动态优先权调度算法的优先权计算的原则是：

    4) 连续占用处理机时间长的进程，优先权相应降低。在进程切换调度时这种进程被调度上处理机的机会较少。
    5) 在较长时间未使用处理机或者虽然频繁使用处理机，但每次使用时间很短的进程，进程优先权较高。在进程切换调度时，这种进程被调度占用处理机的机会增加。

    动态优先权的系统开销比较大，系统开销一方面取决于动态优先数计算的复杂程度，另一方面也与计算的频繁度有关。在计算优先权时机的选择上，一般至少在一定的时间间隔重新计算当前运行进程的优先权以及一些有可能调度上处理机的进程的优先权，而不是将系统中所有的进程之优先权都重新计算。

4. 进程之间的同步和互斥

同一个计算机系统中的进程不是完全孤立的，它们之间存在着相互依赖，相互制约的关系。某些进程相互协作，共同完成某种任务；同时，它们又互相竞争使用系统的有限资源。进程的这些关系意味着进程之间需要相互通讯，这表现为同步和互斥两个方面。

两个进程之间的**同步**是指一个进程达到某一运行点后，除非另一进程完成了某些操作，否则就不得不停下来等待这些操作的完成。系统中存在有这样的资源，它一次只能分配给一个进程使用。这样的资源称为**临界资源**。当一个进程使用该资源时，其它进程必须等待该进程释放它之后，才会获得该资源的占有权。进程之间的这种关系叫做**互斥**。进程之间可以共享某些数据，对这些共享数据的操作往往也必须是互斥的。与互斥有关的程序段称为**临界区**，针对同一临界资源进行操作的程序段称为**同类临界区**。

锁机制、信号量以及条件变量是实现同类临界区同步和互斥的三种常用方法。

5. 进程的实施

进程由四个部分组成：程序（正文段）、数据（数据段）、进程的运行栈（栈段）以及进程控制块（PCB）。正文段描述了进程所要完成的功能，一般不能修改，所以正文段可以被多个进程共享，又称共享正文段；数据段中是要完成功能所需要的数据，而且这部分数据需要使用不随进程运行而变化的存储控件，一般而言，数据段不能被共享；栈段是进程运行的附加空间，记录了该进程运行的一部分状态，不能被共享；进程控制块包含了进程的描述信息和控制信息，是进程动态特性的集中反映。在一个最简单的操作系统中，进程控制块至少要保存以下信息：

- 正文段、数据段以及栈段的位置
- 进程的状态
- 进程的运行现场：
  在一个多进程的系统中，一个准备就绪状态的被选择切换上处理机运行时，应该从它上次运行的暂停点开始。这样才能保证进程运行的正确性。为此在进程控制块中设置运行现场部分，它保存进程上次退出处理机时的各硬件寄存器（一般包括程序计数器、程序状态字以及各种通用寄存器中）的值。

对于一个复杂的系统，进程控制块中还需要保存其它一些信息，包括进程运行的各项统计数据等。

6. 进程的创建

很多操作系统的进程结构如同树状。以 UNIX 为例，每个进程都可以创建若干子进程。整个操作系统中的进程树型结构理论上可以不断延伸。一个进程创建了子进程，它被称为该子进程的父亲。父亲创建子进程的基本任务是为新进程构造一个可以运行的环境，包括正文段、数据段和运行的初始栈，以及子进程的进程控制结构，并且保证能够使子进程作为一个可被独立调度的进程。UNIX 创建子进程的基本方式是：除了与进程状态、标识以及与时间有关的少数控制项外，子进程复制或共享父进程的图象。

子进程复制了父进程的进程图象后，就和父进程一样有了自己的正文段（虽然是与父进程共享）、数据段、一个独立的 PC 指针以及自己独立的运行栈，可以独立运行。我们将进程所运行的空间称为地址空间，进程 PC 指针和运行栈看作一个线索，那么子进程和父进程一样拥有各自的地址空间和而且各有一个运行线索。

二、线程

1. 线程概念

我们经常通过生成子进程的方式去完成相关而又独立的任务。比如一个文件服务器，它接收读写文件的请求并将所需数据传入或传出，为了提高性能，服务器将最近传送的数据放入缓存，以便在需要时重复使用。当有请求到达时，为了处理的独立性，该进程就要生成一个子进程去完成这一请求。当多个请求到达时，就需要生成多个子进程。这种处理方法比较简单明了，但是也有不少问题：

- 创建子进程以及在进程之间切换的开销比较大。
- 进程之间共享缓存和进程和进行通信虽然可以使用操作系统提供的很多进程通信机制，包括共享内存段等。但是也有系统开销大，效率偏低的问题。

为了解决这些问题，提出了线程的概念。见图 3.3；

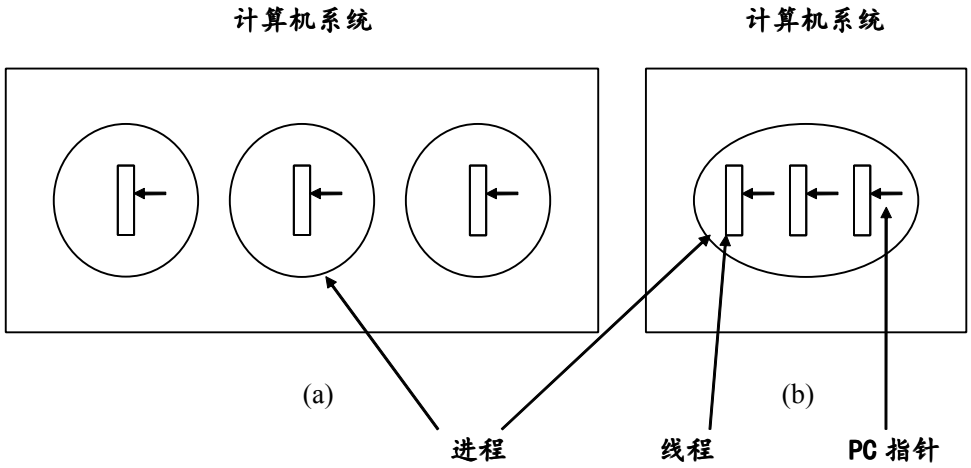


图 3.3　进程和线程

图 3.3(a)计算机系统中共运行了三个进程，各自有自己的地址空间和运行线索。如果这三个进程的地址空间是完全可以共享的，所不同的只是运行的线索（例如在文件服务器中），那是否可以将三个进程的地址空间完全合一呢？这就是图 3.3(b)所描绘的线程。图 3.3(b)中只有一个进程，但该进程中包含有三个运行线索。每个运行线索各有自己的 PC 指针和运行栈空间，严格按照自己的顺序进行。同进程调度一样，在某个时刻单机系统中只能有一个线程在运行。引入线程之后，线程即成为调度程序可以调度的最小单位，但是如果在同一进程中的不同线程之间进行调度切换，那么系统开销将显著降低。对线程的调度算法随操作系统的不同而异。

线程的引入还为同族线程的数据共享提供了便利。只要系统提供基本的同步互斥操作，例如锁和信号量，同族线程就可以方便地对数据区实施共享，相对于同族进程必须通过操作系统核心共享数据操作就简单得多了。同族线程间对共享数据的管理任务从系统内核移到了用户程序上，既减轻了核心的负担，又给程序员编写一些共享程序提供了方便。

现代操作系统使用线程并不仅仅针对用户程序，线程思想还可渗入到核心的各个部分。系统内核把每一个系统调用看作来自用户端的服务请求，经过一些必要的处理后，就产生一个新线程去执行具体的操作。这些线程共享了核心的地址空间，可以直接访问核心的公用数据。这样做有两个明显的好处：(1) 简化了系统内核，加快了内核对系统调用的响应速度；(2) 简化了核心态下的状态保存。这些思想是一些微内核操作系统的主体思想。

2. 进程和线程的关系

在引入线程机制后，进程不再是单一的动态实体，而是由两部分组成：

- 各线程活动的环境，包括：统一的地址控件、全局变量、打开文件和计时器等。
- 若干个线程，它们是进程中的活动部分，也是处理机的调度单位，而进程不再是处理机的最小调度单位。

一个进程中的所有线程在同一地址空间中活动，共享该地址空间中的全局变量，共享打开文件和计时器等。它们总是相互协作，各自承担一个作业中的某个部分。与传统的进程相似，线程具有状态的变化。通常，这些状态是：运行、阻塞、就绪或终止。

表 3.1　与进程和线程有关的主要信息表

线程控制信息	进程控制信息
程序计数器	地址空间
运行栈	全局变量
寄存器集	打开文件
子线程	子进程
运行状态	计时器
	线程

第二节　Nachos的线程管理

一、Nachos的线程管理

Nachos 的第一个需要扩充的部分是线程管理。Nachos 提供了一个基本的线程管理系统和一个同步互斥机制信号量的实现。读者在此基础上完成锁和条件变量等另两种同步互斥机制并对现有的线程机制进行一些加强。然后利用这些同步原语解决一些并发性问题。例如实现一个简单的生产者-消费者问题，在实现中使用条件变量来表示缓冲区空和缓冲区满两个状态。

对于刚刚学习编程的读者来说，理解并发性需要一个概念上的飞跃。Nachos 的设计者认为，教授并发性的最好方法是让读者能够直观地看到程序的并发行为，只有这样，读者才能学会实现并发性程序的正确方法。这一章的设计正体现了这个思想。

Nachos 中的线程管理设计有很多优点，读者可以一条指令一条指令地跟踪线程切换的过程，既可以从外界观察者的角度，也可以从相关线程的角度观察，在一个线程切换到另一个线程的过程中发生了什么事情。这个经历对帮助读者解开并发之谜是至关重要的。其次，Nachos 是一个可以实际工作的线程系统，可以在它上面编写并发性程序，并测试这些程序。在测试的过程中，将发现许多过去没有注意到的问题。事实证明，即使是有丰富经验的程序员也觉得考虑并发性问题是很困难的。在 Nachos 线程管理作业部分增加了许多实际问题的处理。同时，在现有的 Nachos 版本已实现的部分中，可能也会存在在并发问题上考虑不周全的部分，读者可以对其加以改进。但是正如机器模拟一章中提到的：在现有的系统中一个不正确的同步程序也可能正确地运行。所以读者对线程管理系统的修改之后，还需要对以前认为正确的同步互斥程序进行一些测试，以便尽早发现问题。

Nachos 广泛采用线程的概念，是多线程操作系统。线程是 Nachos 处理机调度的单位，在 Nachos 中线程分成两类，一类是系统线程。所谓系统线程是只运行核心代码的线程，它运行在核心态下，并且占用宿主机的资源，系统线程共享 Nachos 操作系统本身的正文段和数据段；一个系统线程完成一件独立的任务，比如在 Nachos 网络部分，有一个独立的线程一直监测有无发给自己的数据报。

Nachos 的另一类线程同 Nachos 中的用户进程有关。Nachos 中用户进程由两部分组成，核心代码部分和用户程序部分。用户进程的进程控制块是线程控制块基础上的扩充。每当系统接收到生成用户进程的请求时，首先生成一个系统线程，进程控制块中有保存线程运行现场的空间，保证线程切换时现场不会丢失。该线程的作用是给用户程序分配虚拟机内存空间，并把用户程序的代码段和数据段装入用户地址空间，然后调用解释器解释执行用户程序；由于 Nachos 模拟的是一个单机环境，多个用户进程会竞争使用 Nachos 唯一的处理机资源，所以在 Nachos 用户进程的进程控制块中增加有虚拟机运行现场空间以及进程的地址空间指针等内容，保证用户进程在虚拟机上的正常运行。

在图 3.4 中可以看出，系统线程竞争使用宿主机的 CPU 资源，而用户进程的用户程序部分竞争使用的是虚拟机的 CPU 和寄存器。所以用户进程在被切换下处理机时，需要保存其系统线程部分的现场，同时还需要保存虚拟机部分的现场。

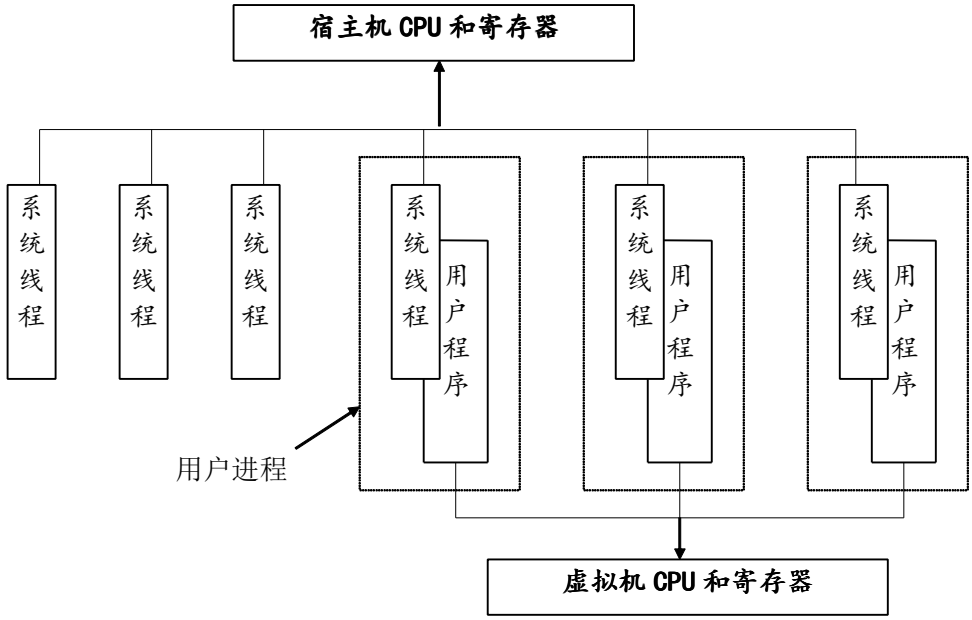


图 3.4　Nachos 中的系统线程和用户进程

当线程运行终止时，由于当前线程仍然运行在自己的栈空间上，所以不能直接释放空间，只有借助其他的线程释放自己，这点将在以后详细叙述。

Nachos 为线程提供的功能函数有:

1. 生成一个线程(Fork)
2. 使线程睡眠等待(Sleep)
3. 结束线程(Finish)
4. 设置线程状态(setStatus)
5. 放弃处理机(Yield)

线程系统的结构如图 3.5 所示:

图 3.5　Nachos 线程系统的结构

线程管理系统中，有两个与机器相关的函数，正文切换过程依赖于具体的机器，这是因为系统线程切换是借助于宿主机的正文切换，正文切换过程中的寄存器保护，建立初始调用框架等操作对不同的处理机结构是不一样的。其中一个函数是 ThreadRoot，它是所有线程运行的入口；另一个函数是 SWITCH，它负责线程之间的切换。

Scheduler 类用于实现线程的调度。它维护一个就绪线程队列，当一个线程可以占用处理机时，就可以调用 ReadyToRun 方法把这个线程放入就绪线程队列，并把线程状态改成就绪态。FindNextToRun 方法根据调度策略，取出下一个应运行的线程，并把这个线程从就绪线程队

列中删除。如果就绪线程队列为空，则此函数返回空(NULL)。现有的调度策略是先进先出策略(FIFO)，

Thread 类的对象既用作线程的控制块，相当于进程管理中的 PCB，作用是保存线程状态、进行一些统计，又是用户调用线程系统的界面。

用户生成一个新线程的方法是:

```
Thread* newThread = new Thread("New Thread"); // 生成一个线程类
newThread->Fork(ThreadFunc, ThreadFuncArg);    // 定义新线程的执行函数及其参数
```

Fork 方法分配一块固定大小的内存作为线程的堆栈，在栈顶放入 ThreadRoot 的地址。当新线程被调上 CPU 时，要用 SWITCH 函数切换线程图像，SWITCH 函数返回时，会从栈顶取出返回地址，于是将 ThreadRoot 放在栈顶，在 SWITCH 结束后就会立即执行 ThreadRoot 函数。ThreadRoot 是所有线程的入口，它会调用 Fork 的两个参数，运行用户指定的函数；Yield 方法用于本线程放弃处理机。Sleep 方法可以使当前线程转入阻塞态，并放弃 CPU，直到被另一个线程唤醒，把它放回就绪线程队列。在没有就绪线程时，就把时钟前进到一个中断发生的时刻，让中断发生并处理此中断，这是因为在没有线程占用 CPU 时，只有中断处理程序可能唤醒一个线程，并把它放入就绪线程队列。

线程要等到本线程被唤醒后，并且又被线程调度模块调上 CPU 时，才会从 Sleep 函数返回。有趣的是，新取出的就绪线程有可能就是这个要睡眠的线程。例如，如果系统中只有一个 A 线程，A 线程在读磁盘的时候会进入睡眠，等待磁盘操作完成。因为这时只有一个线程，所以 A 线程不会被调下 CPU，只是在循环语句中等待中断。当磁盘操作完成时，磁盘会发出一个磁盘读操作中断，此中断将唤醒 A 线程，把它放入就绪队列。这样，当 A 线程跳出循环时，取出的就绪线程就是自己。这就要求线程的正文切换程序可以将一个线程切换到自己，Nachos 的线程正文切换程序 SWITCH 可以做到这一点，于是 A 线程实际上并没有被调下 CPU，而是继续运行下去了。

二、Nachos线程管理同实际进程管理的不同

Nachos 除了在线程管理上作了一系列的简化外，和实际的进程管理还有以下的不同：

- 不存在系统中所有线程的列表
  在一般的操作系统中，进程的数目总是有限的，但是 Nachos 中的线程数目可以是无限的（当然，用户进程的数目应该也是有限的。当虚拟机内存以及虚拟内存都耗尽时，就不能产生新的用户线程）。这是因为，线程的控制结构和系统线程的运行是占用宿主机的。能够开多少线程完全由宿主机条件限制，理论上是无限的。
- 线程的调度比较简单
  在启动了时钟中断的情况下，当时钟中断到来时，如果就绪线程队列中有就绪线程，就必须进行线程切换；当没有启动时钟中断的情况下，Nachos 使用非抢占式调度。
- 没有实现父子线程的关系
  可以说，所有的 Nachos 线程都是 Nachos 的一个子线程。但是 Nachos 线程之间的父子关系没有实现。这样产生的混乱体现在线程的空间释放上，一个线程空间的释放是由下一个被切换的线程也即兄弟线程进行的，而这两个线程可以是没有任何关系的。这样的情况对以后进一步进行系统扩充是不利的。

第三节　Nachos线程管理系统的初步实现

1. 工具模块分析（文件list.cc list.h utility.cc utility.h）

工具模块定义了一些在 Nachos 设计中有关的工具函数，和整个系统的设计没有直接的联系，所以这里仅作一个简单的介绍。

List 类在 Nachos 中广泛使用，它定义了一个链表结构，有关 List 的数据结构和实现如下所示：

```
class ListElement {                          //  定义了 List 中的元素类型
   public:
     ListElement(void *itemPtr, int sortKey);//  初始化方法
     ListElement *next;                      //  指向下一个元素的指针
     int key;                                //  对应于优先队列的键值
     void *item;                             //  实际有效的元素指针
};
```

其中，实际有效元素指针是(void *)类型的，说明元素可以是任何类型。

```
class List {
  public:
    List();                          //  初始化方法
    ~List();                         //  析构方法
    void Prepend(void *item);              //  将新元素增加在链首
    void Append(void *item);               //  将新元素增加在链尾
    void *Remove();                  //  删除链首元素并返回该元素
    void Mapcar(VoidFunctionPtr func);     //  将函数 func 作用在链中每个元素上
    bool IsEmpty();                  //  判断链表是否为空
    void SortedInsert(void *item, int sortKey);   //  将元素根据 key 值优先权插入
到链中
    void *SortedRemove(int *keyPtr);       //  将 key 值最小的元素从链中删除，
                                     //  并返回该元素
  private:
    ListElement *first;              //  链表中的第一个元素
    ListElement *last;               //  链表中的最后一个元素
};
```

其它的工具函数如 min 和 max 以及一些同调试有关的函数，这里就不再赘述。

2. 线程启动和调度模块分析（文件switch.s switch.h）

线程启动和线程调度是线程管理的重点。在 Nachos 中，线程是最小的调度单位，在同一时间内，可以有几个线程处于就绪状态。Nachos 的线程切换借助于宿主机的正文切换，由于

这部分内容与机器密切相关，而且直接同宿主机的寄存器进行交道，所以这部分是用汇编来实现的。由于 Nachos 可以运行在多种机器上，不同机器的寄存器数目和作用不一定相同，所以在 switch.s 中针对不同的机器进行了不同的处理。读者如果需要将 Nachos 移植到其它机器上，就需要修改这部分的内容。

2.1 ThreadRoot函数

Nachos 中，除了 main 线程外，所有其它线程都是从 ThreadRoot 入口运行的。它的语法是：

```
ThreadRoot (int InitialPC, int InitialArg, int WhenDonePC, int StartupPC)
```

其中，InitialPC 指明新生成线程的入口函数地址，InitialArg 是该入口函数的参数；StartupPC 是在运行该线程是需要作的一些初始化工作，比如开中断；而 WhenDonePC 是当该线程运行结束时需要作的一些后续工作。在 Nachos 的源代码中，没有任何一个函数和方法显式地调用 ThreadRoot 函数，ThreadRoot 函数只有在线程切换时才被调用到。一个线程在其初始化的最后准备工作中调用 StackAllocate 方法（见本章 3.4），该方法设置了几个寄存器的值（InterruptEnable 函数指针，ThreadFinish 函数指针以及该线程需要运行函数的函数指针和运行函数的参数），该线程第一次被切换上处理机运行时调用的就是 ThreadRoot 函数。其工作过程是：

1. 调用 StartupPC 函数；
2. 调用 InitialPC 函数；
3. 调用 WhenDonePC 函数；

这里我们可以看到，由 ThreadRoot 入口可以转而运行线程所需要运行的函数，从而达到生成线程的目的。

2.2 SWITCH函数

Nachos 中系统线程的切换是借助宿主机的正文切换。SWITCH 函数就是完成线程切换的功能。SWITCH 的语法是这样的：

```
void SWITCH (Thread *t1, Thread *t2);
```

其中 t1 是原运行线程指针，t2 是需要切换到的线程指针。线程切换的三步曲是：

1. 保存原运行线程的状态
2. 恢复新运行线程的状态
3. 在新运行线程的栈空间上运行新线程

3. 线程模块分析（文件thread.cc thread.h）

Thread 类实现了操作系统的线程控制块，同操作系统课程中进程程管理中的 PCB (Process Control Block) 有相似之处。

Thread 线程控制类较 PCB 为简单的多，它没有线程标识 (pid)、实际用户标识 (uid)等和线程操作不是非常有联系的部分，也没有将 PCB 分成 proc 结构和 user 结构。这是因为一个 Nachos 线程是在宿主机上运行的。无论是系统线程和用户进程，Thread 线程控制类的实例都生成在宿主机而不是生成在虚拟机上。所以不存在实际操作系统中 proc 结构常驻内存，而 user 结构可以存放在盘交换区上的情况，将原有的两个结构合并是 Nachos 作的一种简化。

Nachos 对线程的另一个简化是每个线程栈段的大小是固定的，为 4096-5 个字 (word)，而且是不能动态扩展的。所以 Nachos 中的系统线程中不能使用很大的栈空间，比如：

```
void foo () { int buff[10000]; …}
```

可能会不能正常执行，如果需要使用很大空间，可以在 Nachos 的运行堆中申请：

```
void foo () { int *buf = new int[10000]; …}
```

如果系统线程需要使用的栈空间大于规定栈空间的大小，可以修改 StackSize 宏定义。

Thread 类的定义和实现如下所示：

```
class Thread {
  private:
    int* stackTop;                      //  当前堆栈指针
    int machineState[MachineStateSize];      //  宿主机的运行寄存器
  public:
    Thread(char* debugName);                 //  初始化线程
    ~Thread();                          //  析构方法
    void Fork(VoidFunctionPtr func, int arg); //  生成一个新线程，执行 func(arg)
    void Yield();                       //  切换到其它线程运行
    void Sleep();                       //  线程进入睡眠状态
    void Finish();                      //  线程结束时调用
    void CheckOverflow();               //  测试线程栈段是否溢出
    void setStatus(ThreadStatus st);         //  设置线程状态
    char* getName() { return (name); }       //  取得线程名（调试用）
    void Print() { printf("%s, ", name); }        //  打印当前线程名（调试用）
  private:
    int* stack;                         //  线程的栈底指针
    ThreadStatus status;                //  当前线程状态
    char* name;                         //  线程名（调试用）
    void StackAllocate(VoidFunctionPtr func, int arg);
                                        //  申请线程的栈空间
#ifdef USER_PROGRAM
    int userRegisters[NumTotalRegs];         //  虚拟机的寄存器组
  public:
    void SaveUserState();                    //  线程切换时保存虚拟机寄存器组
    void RestoreUserState();            //  线程切换时恢复虚拟机寄存器组
    AddrSpace *space;                   //  线程运行的用户程序
#endif
}
```

线程控制类中的 stackTop 栈指针变量，machineState 机器寄存器数组变量的位置必须是固定的，因为这两个变量和线程的切换有密切的关系，在 SWITCH 函数中：

```
    void SWITCH (Thread *t1, Thread *t2);
```

(int *)(*t1)实际上指向原有线程的栈空间，而 (int)(*(t1+1))开始则同宿主机的寄存器组一一对应。这样使 SWITCH 的实现比较简单。

线程状态有四种，状态之间的转换如图 3·6 所示：

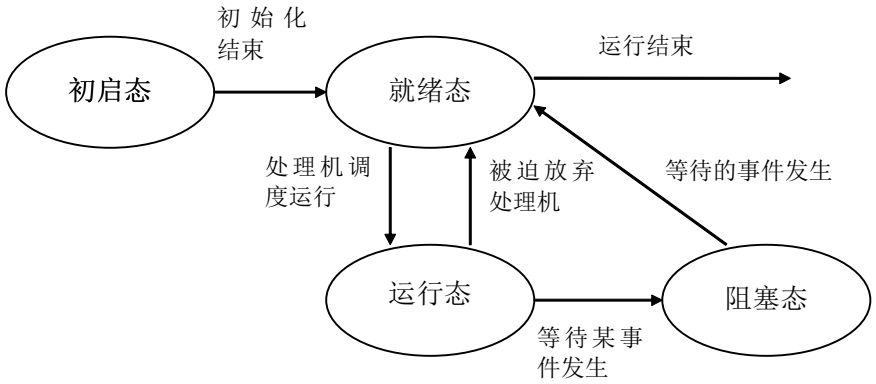


图 3.6　Nachos 线程的状态转换

JUST_CREATED	线程初始时的状态。此时线程控制块中没有任何内容
RUNNING	线程正在处理机上运行
READY	线程处理就绪态
BLOCKED	线程处于阻塞态

用户进程在线程切换的时候，除了需要保存宿主机的状态外，必须还要保存虚拟机的寄存器状态。UserRegisters[]数组变量和 SaveUserState(), RestoreUserState()方法就是为了用户进程的切换设计的。

在 UNIX 操作系统中，进程终止时释放大部分空间，有一部分工作留给父进程处理。（除了 0 号进程外，所有的进程都有父进程）在 Nachos 中，当一个线程运行结束时，同样需要将线程所占用的空间释放。但是 Nachos 线程不能释放自己的空间，因为此时它还运行在自己的栈段上。所以当线程结束时调用 Finish 方法，Finish 方法的作用是设置全局变量 threadToBeDestroyed，说明该线程已经运行结束，需要释放栈空间。Finish 紧接着切换到其它线程运行，该运行线程释放 threadToBeDestoryed 线程栈空间。Scheduler 类中的 Run 方法才有机会删除 threadToBeDestroyed 线程栈空间。当系统中没有就绪线程和中断等待处理时，系统会退出而不会切换到其它线程，只有借助于系统释放空间的机制来释放 threadToBeDestroyed 线程的空间。（不能在 Interrupt 类中的 Idle 方法加上：

```
if (threadToBeDestroyed != NULL)
{
    delete threadToBeDestroyed;
    threadToBeDestroyed = NULL;
}
```

试一试，想想这是为什么？）

3.1 Fork 方法

语法：　void Fork (VoidFunctionPtr func, int arg)

参数：　func:　　新线程运行的函数
　　arg:　　func 函数的参数
功能：　线程初始化之后将线程设置成可运行的。
实现：　1.　申请线程栈空间
　　2.　初始化该栈空间，使其满足 SWITCH 函数进行线程切换的条件
　　3.　将该线程放到就绪队列中。
返回：　无。

3.2 StackAllocate 方法

语法：　void StackAllocate (VoidFunctionPtr func, int arg)
参数：　func:　　新线程运行的函数
　　arg:　　func 函数的参数
功能：　为一个新线程申请栈空间，并设置好准备运行线程的条件。

实现：
```
void Thread::StackAllocate (VoidFunctionPtr func, int arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    //  申请线程的栈空间
    stackTop = stack + StackSize - 4;          //  设置栈首指针
    *(--stackTop) = (int)ThreadRoot;
    //  线程准备好运行后进行线程切换，会切换到 ThreadRoot 函数。ThreadRoot 函数
    //  将会开中断，并调用 func(arg)成为一个独立的调度单位。
    *stack = STACK_FENCEPOST;          //  设置栈溢出标志
    machineState[PCState] = (int) ThreadRoot;  //  设置 PC 指针，从 ThreadRoot 开始运行
    machineState[StartupPCState] = (int) InterruptEnable;
    machineState[InitialPCState] = (int) func;
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = (int) ThreadFinish;
    //  以上是为 ThreadRoot 作好准备，ThreadRoot 将分别调用 InterruptEnable，
    //  func(arg)和 ThreadFinish。
}
```
返回：　无

3.3 Yield 方法

语法：　void Yield ()
参数：　无
功能：　当前运行强制切换到另一个就绪线程运行
实现：

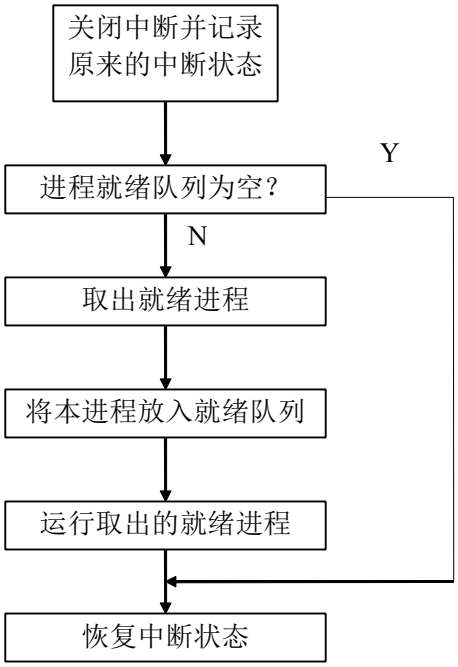


图 3.7　Yield 函数

注意：这些操作是原子操作，不允许中断。

返回：　无。

3.4 Sleep 方法

语法：　void Sleep ()

参数：　无

功能：　线程由于某种原因进入阻塞状态等待一个事件的发生（信号量的 V 操作、开锁或者条件变量的设置）。当这些条件得到满足，该线程又可以恢复就绪状态。

实现：

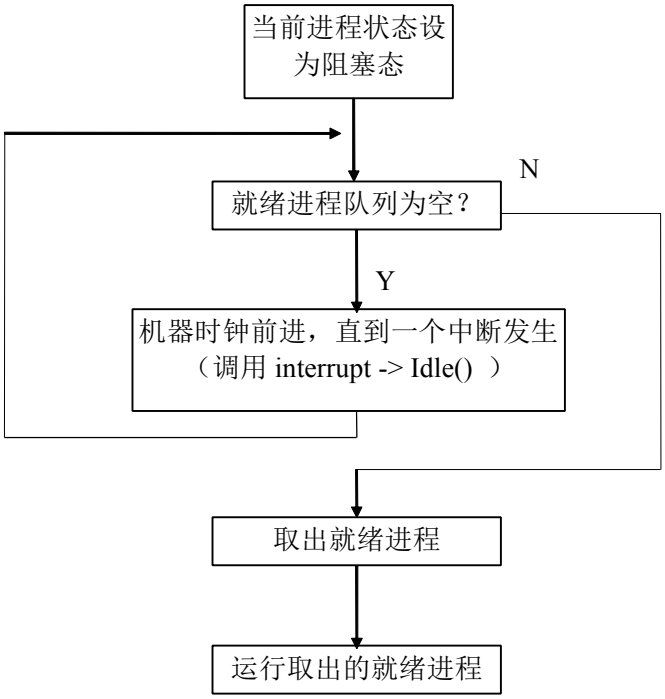


图 3.8　Sleep 函数

注意：这些操作是原子操作，不允许中断。

返回：　无。

4. 线程调度算法模块分析（文件scheduler.cc scheduler.h）

该模块的作用是进行线程的调度。在 Nachos 系统中，有一个线程就绪队列，其中是所有就绪线程。调度算法非常简单，就是取出第一个放在处理机运行即可。由于 Nachos 中线程没有优先级，所以线程就绪队列是没有优先级的。读者可以在这一点上进行加强，实现有优先级的线程调度。

Scheduler 类的定义和实现如下：

```
class Scheduler {
  public:
    Scheduler();                    //  初始化方法
    ~Scheduler();                   //  析构方法
    void ReadyToRun(Thread* thread);    //  设置一个线程为就绪态
    Thread* FindNextToRun();            //  找出下一个处于就绪态的线程
    void Run(Thread* nextThread);       //  切换到 nextThread 执行
    void Print();                   //  打印出处于就绪态的所有线程
  private:
    List *readyList;                //  线程就绪队列
};
```

所有 Scheduler 中定义的方法都有一个前提条件：必须是原子操作，不允许中断。

4.1 Run方法

语法：　void Run (Thread *nextThread)
参数：　nextThread:　需要切换运行的线程
功能：　当前运行强制切换到 nextThread 就绪线程运行
实现：
1. 如果是用户线程，保存当前虚拟机的状态
2. 检查当前运行线程栈段是否溢出。（由于不是每时每刻都检查栈段是否溢出，所以这时候线程的运行可能已经出错）
3. 将 nextThread 的状态设置成运行态，并作为 currentThread 现运行线程（在调用 Run 方法之前，当前运行线程已经放入就绪队列中，变成就绪态）

（以上是运行在现有的线程栈空间上，以下是运行在 nextThread 的栈空间上）

4. 切换到 nextThread 线程运行
5. 释放 threadToBeDestroyed 线程需要栈空间（如果有的话）
6. 如果是用户线程，恢复当前虚拟机的状态

返回：　无。

5.　Nachos主控模块分析（文件main.cc system.cc system.h）

该模块是整个 Nachos 系统的入口，它分析了 Nachos 的命令行参数，根据不同的选项进行不同功能的初始化设置。选项的设置如下所示：

一般选项：
-d:　显示特定的调试信息
-rs:　使得线程可以随机切换
-z:　打印版权信息
和用户进程有关的选项：
-s:　使用户进程进入单步调试模式
-x:　执行一个用户程序
-c:　测试终端输入输出
和文件系统有关的选项：
-f:　格式化模拟磁盘

-cp: 将一个文件从宿主机拷贝到 Nachos 模拟磁盘上
-p: 将 Nachos 磁盘上的文件显示出来
-r: 将一个文件从 Nachos 模拟磁盘上删除
-l: 列出 Nachos 模拟磁盘上的文件
-D: 打印出 Nachos 文件系统的内容
-t: 测试 Nachos 文件系统的效率
和网络有关的选项：
-n: 设置网络的可靠度（在 0-1 之间的一个小数）
-m: 设置自己的 HostID
-o: 执行网络测试程序

main 函数的处理逻辑：

```
int main (int argc, char **argv)
{
    对命令行参数进行处理，并且初始化相应的功能
    currentThread -> Finish ();
    return (0);              //  此行执行不到。
}
```

在 main 函数的最后，是 currentThread->Finish()语句。为什么不直接退出呢？这是因为 Nachos 是在宿主机上运行的一个普通的进程，当 main 函数退出时，整个占用的空间要释放，进程也相应的结束。但是实际上在 Nachos 中，main 函数的结束并不能代表系统的结束，因为可能还有其它的就绪线程。所以在这里我们只是将 main 函数作为 Nachos 中一个特殊线程进行处理，该线程结束只是作为一个线程的结束，系统并不会退出。

6. 同步机制模块分析（文件synch.cc synch.h）

线程的同步和互斥是多个线程协同工作的基础。Nachos 提供了三种同步和互斥的手段：信号量、锁机制以及条件变量机制，提供三种同步互斥机制是为了用户使用方便。

在同步互斥机制的实现中，很多操作都是原子操作。Nachos 是运行在单一处理器上的操作系统，在单一处理器上，实现原子操作只要在操作之前关中断即可，操作结束后恢复原来中断状态。

6.1 信号量（Semaphore）

Nachos 已经实现了 Semaphore，它的基本结构如下所示：

```
class Semaphore {
  public:
    void P();                    //  信号量的 P 操作
    void V();                    //  信号量的 V 操作
  private:
    int value;                   //  信号量值 ( >=0)
    List *queue;                 //  线程等待队列
};
```

信号量的私有属性有信号量的值，它是一个阀门。线程等待队列中存放所有等待该信号量的线程。信号量有两个操作：P 操作和 V 操作，这两个操作都是原子操作。

6.1.1 P 操作

1. 当 value 等于 0 时，
   1.1. 将当前运行线程放入线程等待队列。
   1.2. 当前运行线程进入睡眠状态，并切换到其它线程运行。
2. 当 value 大于 0 时，value--。

6.1.2 V 操作

1. 如果线程等待队列中有等待该信号量的线程，取出其中一个将其设置成就绪态，准备运行。

2.　value++;

6.2 锁机制

锁机制是线程进入临界区的工具。一个锁有两种状态，BUSY 和 FREE。当锁处于 FREE 态时，线程可以取得该锁后进入临界区，执行完临界区操作之后，释放锁；当锁处于 BUSY 态时，需要申请该锁的线程进入睡眠状态，等到锁为 FREE 态时，再取得该锁。

锁的基本结构如下所示：

```
class Lock {
  public:
    Lock(char* debugName);                  // 初始化方法
    ~Lock();                                // 析构方法
    char* getName() { return name; }        // 取出锁名（调试用）
    void Acquire();                         // 获得锁方法
    void Release();                         // 释放锁方法
    bool isHeldByCurrentThread();           // 判断锁是否为现运行线程拥有
  private:
    char* name;                             // 锁名（调试用）
};
```

在现有的 Nachos 中，没有给出锁机制的实现，锁的基本结构也只给出了部分内容，其它内容可以视实现决定。总体来说，锁有两个操作 Acquire 和 Release，它们都是原子操作。

Acquire:　申请锁：
- 当锁处于 BUSY 态，进入睡眠状态。
- 当锁处于 FREE 态，当前线程获得该锁，继续运行

Release:　释放锁（注意：只有拥有锁的线程才能释放锁）
将锁的状态设置成 FREE 态，如果有其它线程等待该锁，将其中的一个唤醒，进入就绪态。

6.3 条件变量

条件变量和信号量与锁机制不一样，它是没有值的。（实际上，锁机制是一个二值信号量，可以通过信号量来实现）当一个线程需要的某种条件没有得到满足时，可以将自己作为一个等待条件变量的线程插入所有等待该条件变量的队列；只要条件一旦满足，该线程就会被唤醒继续运行。条件变量总是和锁机制一同使用，它的基本结构如下：

```
class Condition {
  public:
    Condition(char* debugName);             // 初始化方法
    ~Condition();                           // 析构方法
    char* getName() { return (name); }      // 取出条件变量名（调试用）
    void Wait(Lock *conditionLock);         // 线程进入等待
    void Signal(Lock *conditionLock);       // 唤醒一个等待该条件变量的线
```

```
程
    void Broadcast(Lock *conditionLock);        // 唤醒所有等待该条件变量的线
程
  private:
    char* name;                                 // 条件变量名（调试用）
};
```

在现有的 Nachos 中，没有给出条件变量的实现，条件变量的基本结构也只给出了部分内容，其它内容可以视实现决定。总体来说，条件变量有三个操作 Wait、Signal 以及 BroadCast，所有的这些操作必须在当前线程获得一个锁的前提下，而且所有对一个条件变量进行的操作必须建立在同一个锁的前提下。

void Wait (Lock *conditionLock)	线程等待条件变量 1. 释放该锁 2. 进入睡眠状态 3. 重新申请该锁
void Signal (Lock *conditionLock)	唤醒一个等待该条件变量的线程（如果存在的话）
void BroadCast (Lock *conditionLock)	唤醒所有等待该条件变量的线程（如果存在的话）

第四节　线程管理系统作业

Nachos 的线程管理还有一些不完善的地方，这部分的作业是完善 Nachos 的线程管理系统，读者可以根据自己的理解去完善，然后在自己的线程管理系统的基础上完成一些同步互斥问题。

1. 实现锁机制和条件变量。你可以使用系统提供的 P/V 操作或者甚至使用更加底层的函数。我们在 synch.h 中提供了有关锁和条件变量的 public 的接口。需要读者定义 private 的数据和实现。

2. 在自己实现的锁机制和条件变量的基础上实现有限大小缓冲区的生产者/消费者问题。生产者每次向缓冲区放入一个字符，如果缓冲区满了，它必须等待；消费者每次从缓冲区中读出一个字符写在屏幕上。请考虑存在多个生产者和多个消费者的情况。

3. 实现一个 “alarm clock” 的类。线程通过调用
   Alarm::GoToSleepFor (int howLong)
   的方法进入睡眠状态。alarm clock 类可以通过时钟硬件来实现 (timer.h)。当时钟中断发生时，需要检测是否有线程需要被唤醒。并不需要线程一旦被唤醒时就立刻运行（当然也可以这样做），但是必须将其放入就绪队列中。

4. 实现有优先级的线程调度。修改现有的线程调度机制，使得每次总是调度优先级最高的线程上处理机。读者需要实现一个新的线程构造函数（静态优先调度），保留原有的构造函数以保持向下兼容性。你可以假设系统有固定数目的优先级，当然至少有两个。然后在此基础上再考虑生成者/消费者问题。在这个时候，每个生产者和消费者有不同的优先级，请给出在不同的优先级情况下对输出（其实是线程的调度）有什么具体的影响。

5. 用条件变量实现一个具有预读性质的 cache。这个 cache 可以嵌入在 threads 类中，当一个 thread 提出需要从物理内存或磁盘上读出或者写入数据的时候，如果要读出的部分已经在 cache 中了就立刻返回。如果要读出的数据并不在 cache 中，必须将 cache 中的有些内容从 cache 中剔除，（注意：如果 cache 中的内容被修改过，需要将 cache 中的内容写回磁盘）然后从内存中将要读的内容读入 cache。当然，这里 cache 的数目也是有限的。

   需要注意的是，我们很多工作是为了提高整个系统的效率。但是由于 Nachos 本身是一个在宿主机上的进程。在外部表现上，每增加一个层次却会使 Nachos 系统在宿主机上运行的速度下降。这和提高系统效率是两个不同的问题。

6. 一个电脑管理的投币洗衣房中，有 NMACHINES 台洗衣机，顾客来投币洗衣，顾客投币的数量表示他一次需要几台洗衣机，请编写一个洗衣房管理系统。

7. 交大研究生楼有 18 楼，有一部电梯，每部电梯可以载无数学生。电梯和学生用不同的线程表示。当有学生需要使用电梯时（M 楼到 N 楼）时，唤醒电梯工作，否则电梯处于等待状态；当电梯正在忙时，需要使用电梯的学生必须等待。请编写一个电梯管理系统。

8. 有一座窄桥，桥上同时只能有一个方向的汽车行驶，而且如果桥上的汽车超过两辆，桥就会坍塌。在这个系统中，每辆汽车用一个线程表示，请实现一个汽车过桥管理系统。

9. 用线程的同步互斥机制实现五个哲学家吃通心面问题。五个哲学家围坐在圆桌周围，每个哲学家面前都有一盆意大利面条。面条非常滑，需要两把叉子才能夹住。在两盆面条之间都有一把叉子。

   哲学家不是在思考就是在进食。当哲学家觉得饿了时，他就试图去取他左边与右边叉子，每次动作取一把。无论是怎样的次序，如果顺利地取得两把叉子，就开始进食；吃完以后放下叉子继续思考。（注意不要出现死锁的现象）

10. 在 Nachos 中线程，除了就绪线程队列外没有全局的线程队列，给对所有线程进行统一的计算带来了一定的麻烦。实现这样的全局线程队列，并且改变系统结束条件：当全局线程队列中没有线程存在时，系统退出。

11. 目前 Nachos 系统中没有用户的概念，所以无论是系统线程还是用户进程，进程本身没有属主。在线程结构中增加用户 uid。

12. Nachos 中没有父子线程的概念，建立这样的概念并且维护父子线程之间的关系。

13. 死锁的检测和预防
    操作系统管理着计算机系统的一大批软硬件资源，如果管理不当，在一定条件下会使得一组进程中的每一个进程都占用着一些资源，同时又企图占用已被该组进程的其它进程占用且不能被强迫释放再分配的资源。于是造成进程处于封锁状态，即死锁。

    产生死锁的充要条件有四：
    - 涉及到的资源是不能共享的
    - 一个进程占有一定的资源，又请求分配另一个
    - 进程在对不可共享资源的请求不能得到满足而进入封锁态时，并不释放自己已经占用的资源。
    - 一组进程中存在一个进程对资源需求的循环链。

如果系统中存在死锁，对系统的影响将是显而易见的。这预示着操作系统的宝贵资源被无法完成的任务占据。所以检测和预防死锁是非常必要的。但是由于操作系统的不确定性，死锁的检测和预防将是非常复杂的工作。

读者可以采用 NACHOS 提供的同步互斥机制（如条件变量）来：

- 模拟死锁的发生
- 对系统中存在的死锁进行检测
- 在分配不可共享资源时，增加死锁预防算法。

第五节　实现实例

这个实现实例对线程及调度的基本设计仍然承袭原有的思想，在类 **Thread** 和 **Scheduler** 的基础上进行修改和扩充。

改进的首要任务是决定如何管理系统中的线程，对多线程的管理包括两部分的内容：(1) 如何安排系统中的线程；(2) 线程的调度算法。

4.1 对线程的改进

原有 Nachos 系统的线程没有一个统一的管理。创建线程之后，线程就游离在整个系统中，只有线程就绪队列维护着所有就绪态的线程，在任何时刻，我们都没有办法知道系统中到底有多少线程。正因为这样，当线程就绪队列为空并且中断队列中除了时钟中断之外没有其它中断时，Nachos 就退出。而实际上，可能确实有线程进入睡眠。所以这里对线程的控制结构进行了改善。

系统主要通过各种线程控制结构队列（以下简称线程队列）对线程进行管理。线程队列的作用是加速各种需要对线程进行检索的算法。系统中的线程队列主要有：

1) 系统的线程队列，线程一经创建就插入该队列，直到线程的控制结构被释放前才从该队列中移出，该队列提供了遍历所有线程的手段，比如说每次时钟中断时，可以通过全局的线程队列对所有的定时线程进行一次时间检查以判断是否需要唤醒它们。该队列的队首和队尾指针定义为全局变量 **globalThreadHead** 和 **globalThreadTail**，线程结构中有两个指针 **nextThread** 和 **prevThread** 用来维护这个双向队列；
2) 每个线程管理两个线程队列，该线程活动的子线程队列和终止的子线程队列，这两个队列用以加速父子线程间的通信算法，如 **Wait**。线程结构中指针 **Parent** 指向父线程，指针 **firstChild** 指向活动子线程队列，而指针 **exitChild** 指向终止的子线程队列，类中还定义了指针 **leftSibling** 和 **rightSibling** 用来维护父子线程队列；
3) 系统调度队列，该队列用于实现系统的线程调度算法。这个队列由类 **Scheduler** 进行管理；
4) 各种系统资源上的等待队列，当线程等待分配系统资源而进入睡眠状态时，由系统资源的管理部件对睡眠的线程进行管理，这些管理一般都是用线程队列实现。

修改后的类 **Thread** 的增加的函数成员和数据成员的说明如下：

```
class Thread {
  …                                  //  原有的 Thread 属性和方法
  public :
    int  ErrorNo;                    //  线程操作是的错误信息
    int  ThreadID;                   //  线程的线程号
    int  sleepReason;                //  睡眠的原因
    int  ExitStatus;                 //  线程的终止状态
    Thread  *Parent;                 //  父线程指针
    Thread  *firstChild;             //  指向活动子线程队列
    Thread  *exitChild;              //  指向终止子线程队列
```

```
    Thread  *leftSibling, *rightSibling;      //  维护父子线程双向队列的指针
    Thread  *prevThread, *nextThread;         //  维护系统线程双向队列的指针
    unsigned Priority;                        //  线程的优先级
    void BeforeDestroy() ;                    //  线程控制结构释放前调用，作一些必要
工作
    void SetSleepReason();                    //  设置线程的睡眠原因
}
```

线程的状态增加一个 ZOMBIE 态（僵尸态），当线程运行结束到空间释放之间的一段时间，线程处于该状态，新的状态转换图如下：

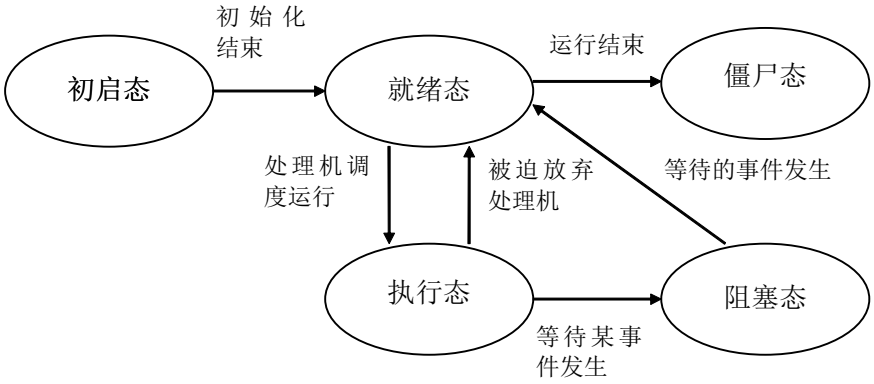


图 3.5　新 Nachos 线程的状态转换

ThreadID 记录了线程的线程号，系统的线程号从 **1** 开始递增。

对于线程在核心中的各种操作。仍然以原有的实现为主，在此基础上为新的管理要求增加一些操作。增加的主要操作是对于线程队列的要求，一个线程创建时，要把自己插入系统的线程队列，同时还要插入父线程的子线程队列；线程终止前必须从各种队列中移走，并有一些其他的扫尾工作，该工作由新的成员函数 **BeforeDestroy()**完成；线程终止时要把所以的子线程设置为孤儿线程，并释放所有终止子线程的控制结构。

一个重要的改动在原有的成员函数 **Finish()**中，线程父子关系的出现，将使终止线程的控制结构由它的父线程释放，子线程终止时把自己从父线程的子线程队列中移出，插入父线程的中止子线程队列，仅当父线程不存在是才采用原来的释放策略。

4.2 对线程调度的改进

线程调度决定当有多个线程处于就绪态时如何选择下一个运行的线程。原来 Nachos 系统的调度算法比较简单：当 Nachos 没有指定-rs 参数时，系统的时钟中断没有启动，所以 Nachos 中的线程必须自己指定放弃处理机；即使指定了-rs 参数，系统启动了时钟中断，每隔一个随机的时间系统中断一次，每次中断的任务就是进行一次线程的切换（见 Nachos 主控模块分析）。这样的线程调度太简单了，不符合一个复杂系统的要求。所以改进后的 Nachos 系统采用时间片轮转法，并采用了动态优先权调度。在每次时钟中断时，重新计算线程的优先级，以便进行线程的优先调度。

调度策略的选择取决于系统的类型，**Nachos** 作为一个分时系统，采用计算优先级的抢占式调度，其调度策略的主要目标是：

1) 加快系统和用户之间的交互速度；
2) 保证系统的运行平稳和一定的系统效率。

这两个目标之间是有冲突的，加快响应速度必然要求在短的时间内运行多道用户程序，而频繁的线程切换会增加系统开销，降低系统效率。因此实现调度算法的工作集中在三方面：

1) 选择合适的时间片大小。时间片大小的选择对系统性能是很重要的，也直接影响到一些算法的计算细节，如 **LRU** 刷新、线程动态优先级的计算。
2) 避免频繁的线程切换。一般的策略是保证运行线程占用处理机后在一段时间内不被强迫调度失去处理机，系统设置一个变量记录上次调度时间，调度时根据这个时间和当前时间的差决定是否调度，由于系统时间统计上的不精确性，其效果很难衡量；
3) 选择合适的优先级计算策略。优先级算法的具体实现在后面对类 **Scheduler** 的改进中作深入讨论。

对调度算法的改进也是在原有的类 **Scheduler** 的基础上进行的，除了在线程控制结构中增加优先级数据，其余所有的优先级控制都集中在 **Scheduler** 中，修改后的增加的常数和 **Scheduler** 中新增的函数和数据成员说明如下：

```
#define     TimerTicks          200             //  时间片大小
#define     CreatePriority      50          //  线程创建时的优先级
#define     BlockedPriority     80          //  睡眠唤醒线程的优先级
#define     AdaptPace           -5          //  优先级调整幅度
#define     MinSwitchPace       (TimerTicks/4)  //  两次抢占调度的最小时间间隔

class Scheduler  {
  …                             //  原有的属性和方法
  public :
    void  FlushPorioty();       //  时钟中断中调用，调整所有就绪线程 的优先级
  private :
    int  lastSwitchTick;        //  记录上次线程切换的时间
}
```

线程优先级的在以下几个时机下计算：

- 创建新的线程时，新线程的优先级为 *CreatePriority*
- 线程睡眠时，醒来的优先级为 *BlockPriority*
- 线程调度时，当前线程的优先级进行调整，计算公式为：
  *Priority = Priority - (*当前系统时间 *- lastSwitchTick) /20*
- 时钟中断中，对所有就绪线程的优先级进行调整，计算公式为：
  Priority = Priority－AdaptPace

成员函数 **FlushPriority()**刷新所有就绪线程的优先级，其算法比较简单，这里就不多作介绍了。原来的成员函数 **FindNextToRun()**担起了优先调度的责任，新的流程如下：

1. 计算当前进程的优先级
2. 计算最近调度与现在的间隔
3. 间隔太小，返回 NULL　　（避免过分频繁地调度）
4. 就绪队列为空，返回 NULL　　（没有就绪线程可以调度）
5. 找到优先级最高的就绪态线程
   5.1. 如果优先级高于当前运行线程，
      5.1.1.运行该线程
      5.1.2.设置最近调度事件
   5.2. 将该线程插回就绪队列，返回 NULL　（继续运行当前线程）

以上是对线程调度算法的简单改进。如果系统中线程运行的时间有很大的差别，可以采用多个时间片就绪队列来代替现有系统中的线程就绪队列。比如三个时间片分别定位 100 个 Tick、500 个 Tick、以及 2500 个 Tick。线程创建时进入 100 个 Tick 的就绪队列，如果 100 个 Tick 之后；线程没有运行结束，就自动转换到 500 个 Tick 的就绪队列，依次类推。对于 2500 个 Tick 的就绪队列采用简单轮循法。系统总是对需要时间片较小的线程优先进行处理，以提高系统的吞吐率，而长线程一旦在处理机上运行，就会占据较长时间，避免了多次线程切换而必须的现场保护的系统开销。但是这样做的缺点是有可能出现在系统较忙的情况下，一个较早提交的线程会一直得不到处理机。同样的，线程优先级的计算也可以进一步改进。

第四章 文件管理系统

第一节　文件管理系统概述

在操作系统中，文件管理系统（简称文件系统）是负责管理和存取文件信息的子系统，它是操作系统中与用户关系最为密切的部分。文件系统为用户提供了一种简便、统一的存取和管理信息的方法。用户可以通过文件名，简单直观地操作存取所需要的信息，而不必关心文件是如何在物理存储介质上存放的以及一些硬件输入输出的细节。

一、文件

1. 文件结构

文件系统的设计者应该从两种不同的观点去研究文件的结构类型和组织方式。

- 用户观点：其主要目的是研究文件的逻辑结构。在这种观点下，文件系统应该向用户提供一种结构清晰、使用简便的逻辑文件结构。
- 实现观点：其主要目的是研究存放在物理存储介质上的实际文件。在这种观点下，文件系统应该选择一些工作性能良好、设备利用率高的文件物理结构。

文件系统的重要作用之一，就是在文件的逻辑结构和物理结构之间建立映照关系，实现两者之间的相互转换。

1.1 文件的逻辑结构

比较常见的文件逻辑结构有以下三种：

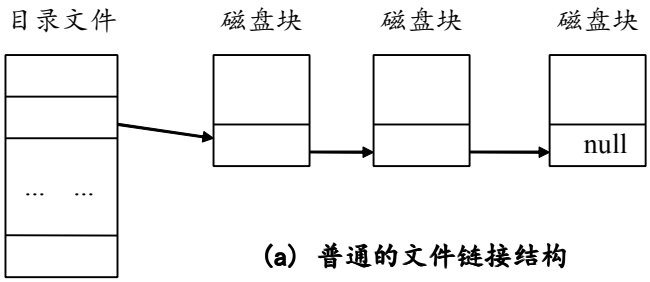
- 无结构的字节流：在这种结构下，操作系统并不知道也不关心文件的内容是什么，文件的涵义由使用该文件的应用程序来解释，这样的文件结构比较通用。
- 定长记录结构：在这种结构下，文件的组成是一个个定长记录。记录的涵义由具体的应用程序解释。操作系统以记录为单位读取、写入文件。
- 变长记录树状结构：在这种结构下，文件由许多不等长的记录组成。在记录的固定位置是该记录的关键值（Key）。通过关键值将文件组织成分类树。当需要对某个记录进行操作时，可以快速地找到该记录。目前的大型机仍然沿用该逻辑结构。

1.2 文件的物理结构

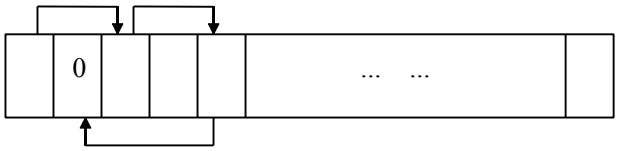
文件的物理结构是文件在物理介质上的存储结构。一般有连续结构、链接结构和索引结构。

- 连续结构
  一个逻辑文件的信息存放在物理介质中连续编号的物理块中，这样的文件是连续结构的文件。文件在磁带上的存放是连续结构的。
- 链接结构
  这是一种非连续的存储结构。存放文件的每一个盘块中有一个指针字，指向下一个盘块（见图 4.1a）。在相应文件目录项中则包含指向文件第一个物理盘块的指针以及文件长度的信息（可以了解共有多少盘块）。DOS 文件系统的 FAT 结构即源于这种结构，所不同的是 FAT 表将各个盘块中的链接指针组织在一起，这样可以比较方便地实现随机存储（见图 4.1b）。

- 索引结构
  这种结构是为每个文件建立逻辑块号到物理块号的对照表，这张表称为该文件的索引表，索引表的索引项按文件的逻辑块号顺序排列。



(a) 普通的文件链接结构



(b) DOS 中的 FAT 表结构

图 4.1　文件的链接结构

每种物理结构都有其优点、缺点及其适用环境，以下表 4.1 是对这三种结构的比较：

表 4.1　三种文件物理结构的比较

	连续结构	链接结构	索引结构
动态增加	不能	能够	能够
指定文件长度创建	是	否	否
充分利用空闲区	否	是	是
存取方式	顺序存取	主要是顺序存取。在 FAT 表结构中，随机存取也较方便	顺序存取和随机存取
访问速度	快	较慢	快
典型系统	磁带文件	MS-DOS	UNIX

2. 文件访问

对文件的访问形式分为顺序访问和随机访问两类。所谓**顺序访问**，就是进程必须以顺序的方式对文件进行字节或记录的读取或写入。比如进程需要读取第 1000 字节位置的内容，它首先需要读取前 999 个字节。

随机访问则可以从文件的任意指定位置开始读、写操作，于是也就消除了顺序存取这个限制。例如进程需要从第 1000 字节位置开始读写，可以直接将它的文件读写指针移至第 1000 字节，然后读取。随机访问可以用在对磁盘文件的访问上，当然磁盘文件也可以用顺序访问方式进行存取。

3. 文件类型

一般操作系统都支持多种文件类型，如普通文件、目录文件以及设备文件等。

- **普通文件**是存放用户信息的文件，它包括以行为单位组成的 **ASCII 文件**和**二进制文件**。ASCII 文件和二进制文件根据文件的不同格式还可以分成很多小类。
- **目录文件**是维持文件系统树状结构的系统文件。系统通过目录文件对文件进行组织和管理。
- **设备文件**是操作系统对设备实施组织管理的一种文件，它对用户屏蔽了设备的特殊性，于是使得用户对设备的操作如同对文件操作一样。设备文件又可以分成字符设备文件和块设备文件。

UNIX 系统中还有管道文件、链接文件等多种文件类型。

4. 文件属性

每个文件都有其文件名和数据，为了对文件进行各种操作和保护，各种操作系统又给每个文件赋予各种属性。以下是一个文件可能有的属性列表。

表 4.2　文件可能有的属性

属性名	含义
Protection	设置谁对该文件有怎样的访问权
Password	访问该文件时需要的口令
Creator	生成该文件的用户 ID
Owner	该文件的所有者
Read-only flag	文件的只读标志。如果该标志设置，该文件只读
Hidden flag	文件隐藏标志
System flag	系统文件标志
Archive flag	文件是否已经备份标志
ASCII/binary flag	ASCII/二进制文件标志
Random access flag	随机访问文件/顺序访问文件标志
Temporary flag	临时文件标志
Lock flags	文件加锁标志（可以有读锁标志和写锁标志）
Record length	文件记录长度
Key position	文件记录键值在记录中的位置
Key length	文件记录键的长度
Creation time	文件的创建时间
Time of last access	文件上次访问时间
Time of last change	文件上次修改时间
Current size	当前文件长度
Maximum size	文件可以增长的最大长度

不是所有的操作系统都支持以上所有文件属性。文件属性根据需要可以用一位、一个字节或一个字来表示，几个属性位可以组合成一个属性字。

5. 文件操作

文件是用来存放信息的，所以文件的基本操作是读和写。但是不同的操作系统对文件的读写方法不尽相同。以下是一些典型的文件操作之系统调用。

表 4.3　文件操作系统调用

系统调用名称	含义
CREATE	生成一个没有内容的文件
DELETE（UNLINK）	删除一个文件
OPEN	打开一个文件供读或写
CLOSE	关闭一个文件
READ	读取一个文件内容
WRITE	将内容写入文件
APPEND	在文件尾添加内容
SEEK	移动文件指针到文件中某一位置
GET ATTRIBUTES	取得文件属性
SET ATTRIBUTES	设置文件属性
RENAME	修改文件名
LINK	将两个文件名和同一个实际的物理文件联系起来

二、目录

用户一般按名字访问文件，文件系统要按名字对文件进行管理，这都要求实现从文件名到到文件物理实体的映照，实施这种功能的重要结构是文件目录。每个文件目录是由目录项组成，每个目录项对应一个文件。

1. 目录结构

1.1　一级目录结构

整个系统中只有一个目录，分成若干个目录项，在整个系统中每个文件的文件名是不同的。这样的目录结构用在比较简单的系统中。

1.2　两级目录结构

在早期的多用户系统中，采用两级目录结构，每个用户使用不同的目录。两级目录结构分成主文件目录（MFD）和用户文件目录（UFD）。它的优点是不同的用户目录下的文件可以同名。

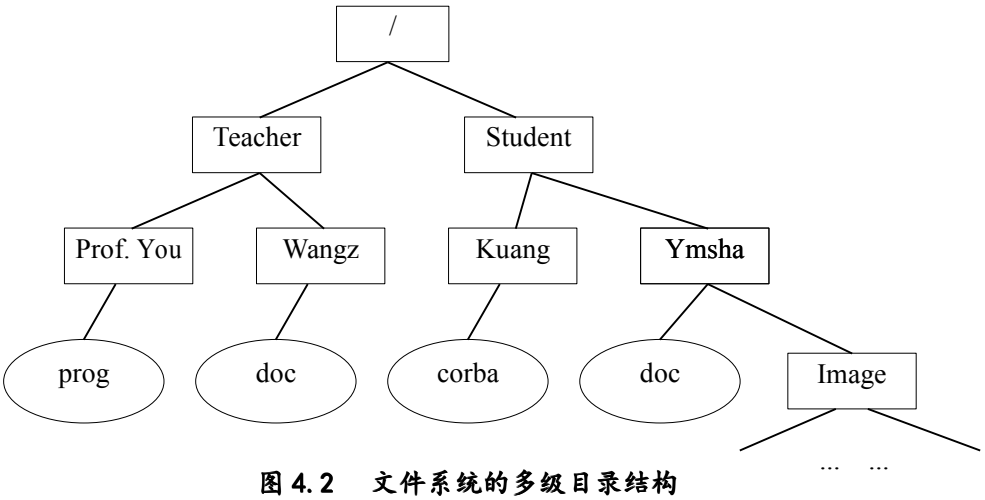


图 4.2　文件系统的多级目录结构

2. 多级目录结构

在一个系统中目录成树状结构。如图 4.2 所示。现在通用操作系统一般都采用多级目录结构，以下的描述也以多级目录为例。

在图中，整个文件系统的目录结构成倒树状，其中方块代表目录，椭圆代表文件。倒树状结构中根节点、中间节点是目录；叶节点则是一个个的文件。

3. 文件路径名

在多级目录结构中，同一个目录下的文件不能重名，但是不同目录下文件的文件名完全可以一样。那么如何来区分这些文件呢？可以通过文件的路径名。所谓文件的路径名可以看作文件在文件系统中的全名，比如上图中 corba 文件的路径名是“/Student/Kuang/corba”，虽然在 Ymsha 目录下和 Wangz 目录下都有以 doc 为文件名的文件，但是它们的路径名是不一样的，分别是“/Teacher/Wangz/doc”和“/Student/Ymsha/doc”。

4. 工作目录

我们可以通过文件路径名对系统中的文件进行操作，但是由于整个文件系统的树型结构是可以无限伸展的，如果完全通过路径名对文件进行操作，系统效率会很低。考虑到一个进程在一段时间存放的文件通常在一定范围的目录中，所以在该范围内指定一个目录为当前工作目录。以后的操作就针对以工作目录为根的子树进行。在 UNIX 中，一个文件的路径名是从根目录开始，必须以’/’开头，称为**绝对路径名**；从工作目录开始的路径名，称为**相对路径名**。

5. 目录结构的勾连

传统的多级目录结构成倒树状，这样的结构不便于实现文件的共享。目录结构的勾连就是将传统的树状结构变成图状。图 4.3 表示了带有勾连的多级目录结构。图中表达了两种勾连方式：一种是对目录项的勾连，如 Wangz 目录，既是 Teacher 目录中的一个目录项，也是 Student 目录中的目录项；另一种是对文件的勾连，如图中的 corba 文件。

对目录的勾连意味着可以共享该目录及其后续子目录所包含的所有文件，共享范围比较宽，但是这种勾连容易形成勾连环。如图 4.3 中虚线所示，造成目录结构的混乱。在分布式系统中更是如此，所以有的系统不允许对目录进行勾连。

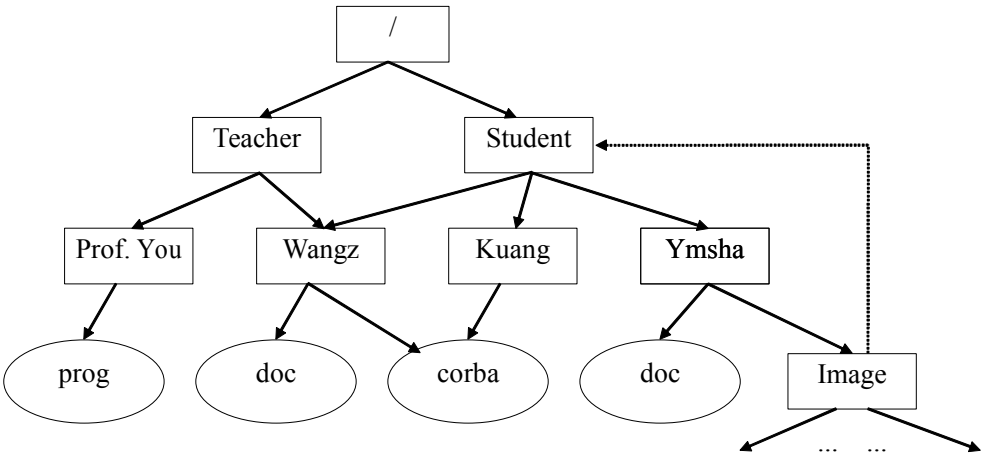


图 4.3　带有勾连的多级目录结构

对文件和目录进行勾连之后，一个文件在系统中就有多个绝对路径名。图中 corba 文件的绝

对路径名是“/Teacher/Wangz/corba”、“/Student/Kuang/corba”以及“/Student/Wangz/corba”等三种。

6. 目录项

一个目录文件是由若干个目录项组成的。目录项中包含有文件名和文件的可能属性（见表4.2）。在不同的操作系统，文件属性的存放位置是不一样的。在 DOS 操作系统中，目录项包含了 DOS 文件的所有属性，同时包含了文件在物理介质上的存放位置（实际上，DOS 目录项中保留的是文件的首簇在物理介质上的位置，通过 FAT 表的链接结构得到后续簇在物理介质上的位置）；而在 UNIX 中，文件的属性和物理存放位置放在一个独立的结构中，称为 inode 结构。系统中所有的 inode 结构集中在一起，形成一个 inode 区。目录项中包含的是文件名和文件 inode 在 inode 区中的位置。

三、UNIX文件系统的实现

1. UNIX文件系统中的主要结构

1.1 inode 结构和文件索引

在讲述目录项时提到，目录项中记录的是文件名和文件相对应的 inode 在整个 inode 区中的索引号，文件 inode 结构中除了存放文件的属性外，最主要的是文件的索引表。

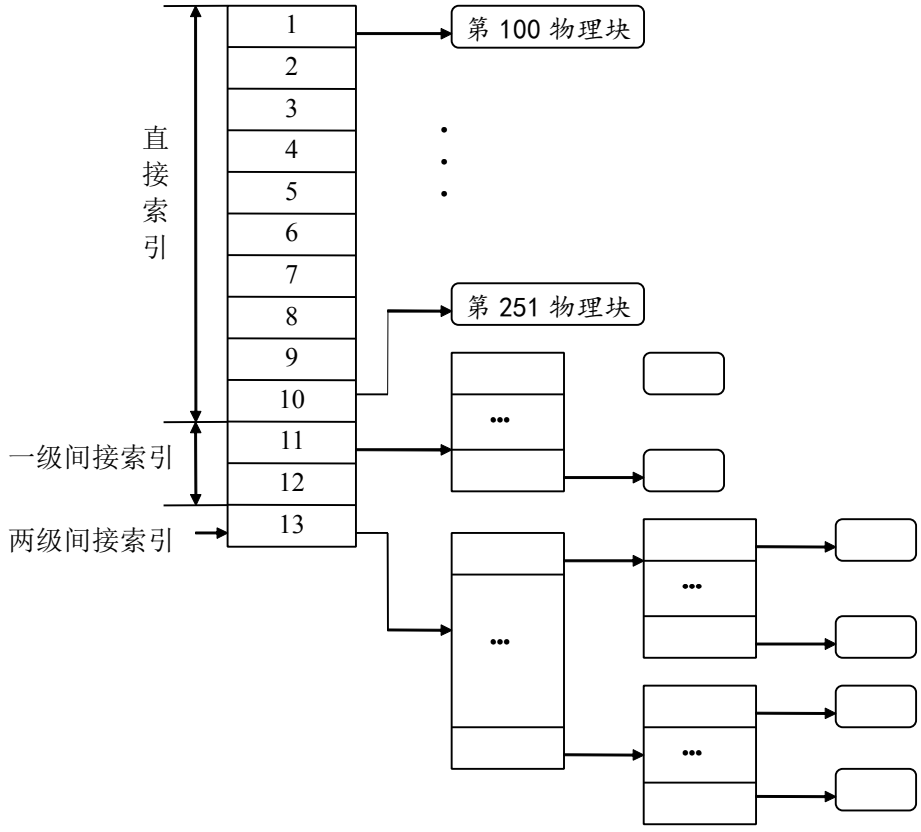


图 4.4　文件的直接索引和间接索引

所谓文件的索引表就是文件的逻辑结构同物理结构之间的转换关系，表的大小是固定的。如每个索引表有 13 项组成。整个表相当于一个变换函数，$f(i)\ (\ i \in [1, 13]\ )$ 代表文件中的

第 i 块在物理介质上的位置，这样的索引称为**直接索引**。文件系统中所能存放文件的最大长度有索引表来决定，如果索引表的每一项都是直接索引，无论是一个磁盘块的大小有多大，文件的最大长度总是有限的，所以引入了间接索引。

间接索引相当于一个多级函数，根据索引的间接层次可以分为一级间接索引、两级间接索引.....设索引表的第 m 项为一级间接索引，前(m-1)项都是直接索引，一个磁盘块的大小是 K 个索引字，则 f(m)不是文件第 m 块在物理介质上的位置，而是指向一个间接索引磁盘数据块，块中记录了一个索引函数，它负责将文件中第 m 到 m+K-1 块的逻辑位置映射到具体的物理位置。两级以上的索引依次类推。多级索引的引入使得文件系统中最大文件的大小可以扩展成无限的。但是为了算法简单，一般索引结构最多采用三级，并且多种索引联合使用，目的是为了提高整个文件系统的效率。

1.2 打开文件结构

一个文件可以被同一个进程多次打开，还能被不同进程同时访问。进程对文件的访问是动态的，具体体现在文件指针的移动上。由于进程之间有相互的独立性，各有自己的文件指针的移动策略，所以每个进程的文件指针应该是不同的。即使是同一个进程，多次打开的同一个文件的文件指针也应该不同。inode 结构并不能反映这样的动态性。

打开文件结构是用来描述进程访问文件的动态信息的，其基本内容如下：

```
struct file
{
    unsigned short      f_count; //  该结构可以被共享，记录共享进程数
    unsigned int      f_inode; //  文件对应的 inode 号
    unsigned long       f_offset;//  当前文件指针的位置
};
```

打开一个文件时，文件指针值为 0；每次读写后，文件指针会移动到读/写部分的下一字节。

1.3 进程打开文件表

每个进程都维护一张打开文件表，表中每一项都可以指向一个打开文件结构。进程打开文件时所作的操作有：分配一个打开文件结构、在内存中建立文件 inode 的备份，并且建立两者之间的关系；同时在进程打开文件表中分配一空闲项，将打开文件结构的指针填入其中，图 4.5 表示了进程打开文件后三个系统结构之间的关系。

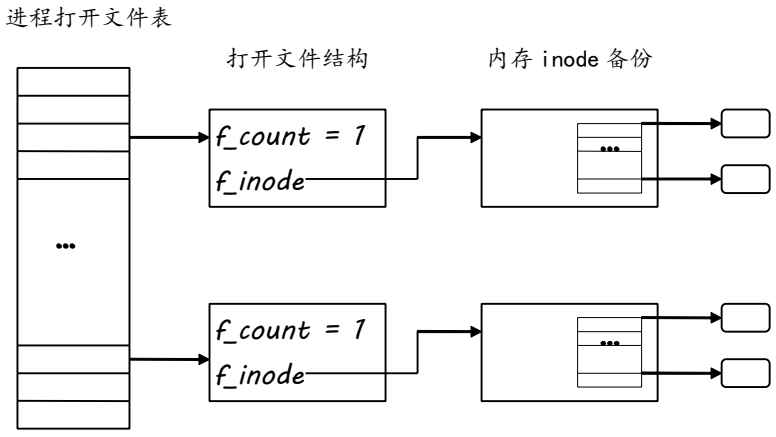


图 4.5　三个系统结构之间的关系

进程以某种形式打开文件后，就不在需要通过文件名对文件进行访问了，而是通过打开文件表中的表项编号来访问。该表项编号被称为**进程打开文件描述符**，简称文件描述符。

2. UNIX文件系统存储资源的分配和回收

2.1 磁盘块的大小

磁盘是一种块设备，对磁盘文件的访问都是以磁盘块为单位的。磁盘块划分得越小，则对文件的一次访问可能需要越多次的磁盘操作；反之，磁盘块越大，则对于小文件比较多的系统，存储空间浪费比较大。所以在一个系统中选择磁盘块的大小和系统应用有关，同时需要在使用效率和访问速率之间进行抉择，一般的文件系统以 1K~4K 的磁盘块为宜。

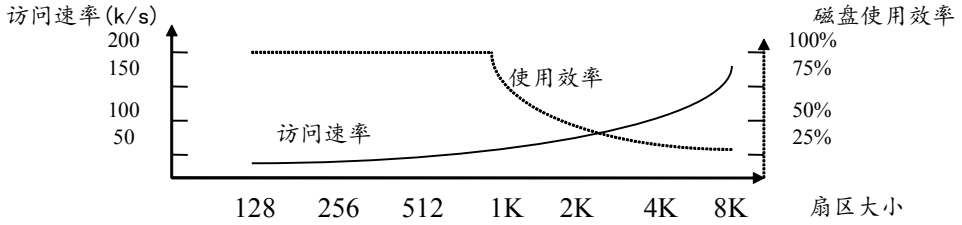


图 4.6　扇区大小同磁盘访问速率和磁盘使用效率之间的关系

2.2 磁盘存储空间的安排

在 UNIX 文件系统中，磁盘块的作用分成两类：一类存放文件的 inode，这一类磁盘块组织在一起，形成 inode 区；另一类存放文件内容本身，该类的集合形成存储数据区，如图 4.7。图中，0#块用来存放系统的自举程序；1#块为管理块，管理本文件系统中资源的申请和回收。主要内容有：

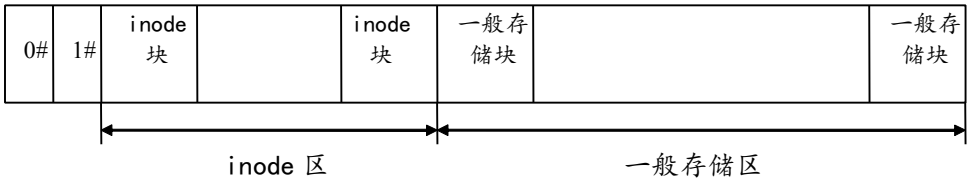


图 4.7　文件系统磁盘存储空间的安排

- 与 inode 管理有关的

```
short   s_isize;              // inode 区包含的磁盘块数
int  s_ninode, s_inode[100];  // 由 1#块直接管理的空闲 inode 数和空闲 inode 索引表
int  s_ilock;                 // 访问空闲 inode 索引表锁标志
```

- 与一般存储块管理有关的

```
short   s_fsize;              // 盘块总数
int  s_nfee, s_free[100];     // 由 1#块直接管理的空闲盘块数和空闲盘块索引表
int  s_flock;                 // 访问空闲盘块索引表锁标志
```

- 其它

在现代操作系统中，文件系统管理的磁盘都比较大。采用以上存储空间的安排会出现一个文件的 inode 同其实际存储区过远的情况，对一个文件的访问可能需要较多的时间。可以通过采用多 inode 区的方式。即整个系统中有多个 inode 区，目的是使系统中绝大多数文件同它实际的存储区较接近。如图 4.8 所示：

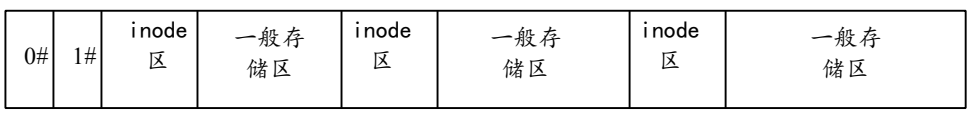


图 4.8　多 inode 区磁盘存储空间安排方式

在多 inode 区磁盘存储空间安排方式中还有固定位置 inode 区和浮动 inode 区两种实现方法。

2.3 空闲 inode 的申请和回收

在 inode 区中，空闲 inode 的数目是非常巨大的，而且在不断的变化。但是文件系统只对其中小于 100 个 inode 进行直接的管理，由 s_inode[100]中的前 s_ninode 项直接指向。

1. 当需要申请一个空闲 inode 时：
   1.1. 若 s_ninode 不为 0，则分配 s_inode[--s_ninode];
   1.2. 若 s_ninode 为 0，则目前文件系统直接管理的空闲 inode 数为 0。但是系统中的空闲 inode 数可能仍然很多。于是需要重新建立 s_inode 表，找出系统中的 100 个空闲 inode，填入表中。并且设置 s_ninode 为 100，然后再实施 1.1 的分配策略。
2. 当需要释放一个 inode 时：
   2.1. 若 s_ninode 不为 0，即空闲 inode 表未满，则填入 s_inode[s_ninode++];
   2.2. 若 s_ninode 为 0，即空闲 inode 表已满，则任其游离在整个系统中，不作任何处理。

2.4 一般存储块的申请和回收

由于系统对一般存储块的访问次数远远超过对 inode 的访问，对一般存储块的管理如果采用上述方法，可能会引起效率的急剧下降，所以改用分组链式索引法进行管理。

所谓**分组**就是将系统中每 100 个空闲磁盘块分成一组，第一组和最后一组除外。所谓**链式索引**就是整个空闲块组通过每组的第一块形成一个链。最后一组直接由空闲块索引表 s_free[100]进行管理，其余各组的索引表则分别存放在他们下一组第一个盘块的前 101 个字节，第一个字是上一组空闲块的个数，随后的 100 个字是索引表。如图中表示对 349 个空闲块的分组链式索引。

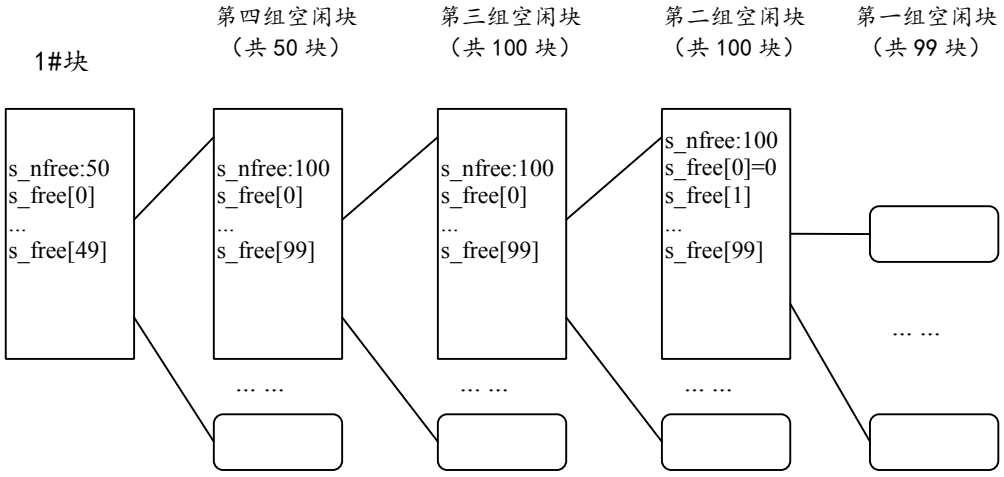


图 4.9　空闲盘块分组链式索引

1. 当需要申请一个空闲盘块时：
   1.1. 若 s_nfree 大于 1，则分配 s_free[--s_nfree];
   1.2. 若 s_nfree 等于 1 时，即分配到本组的最后一个盘块，该盘块上记录了和上一组的链接关系。所以在分配该磁盘块之前需要将前 101 个字读入 s_nfree 以及 s_free[100]中，使得间接管理的下一组变成直接管理的组，直到没有磁盘块可以分配。
2. 当需要释放一个盘块时：
   2.1. 当 s_nfree 不为 100 时，即 s_free 没有满，则将该磁盘块填入 s_free[s_nfree++];
   2.2. 当 s_nfree 为 100 时，即 s_free 已满，就需要重新组织一个组。将 s_nfree 和 s_free[100]

的 101 个字写入盘块的前 101 个字节；然后设置 s_nfree 为 1，并将此释放盘块号填入 s_free[0]。

从以上算法可以看出，UNIX 对空闲 inode 和一般存储块采用的是一种类似栈式的管理方法。这样的管理方法的优点是分配和回收的效率高，使得整个文件系统的效率也较高。但是其缺点在于对文件的误操作几乎没有办法恢复。比如用户误删除了一个文件，其所占用的磁盘块可能会立即被其它文件所占用。尤其对一个多用户多进程的系统来说，对磁盘的访问可以说无时无刻不在进行。更糟糕的情况是，一个文件被误删除之后，其 inode 被刚生成的文件所占用，这将是非常致命的，因为这样会导致文件的索引结构完全丢失。所以一些在 DOS 系统中所常用的命令，如 UNDELETE 命令等，在 UNIX 系统中比较难以实现。

第二节　Nachos文件管理系统

Nachos 是在其模拟磁盘上实现了文件系统。它包括一般文件系统的所有的特性，可以：

- 按照用户的要求创建文件和删除文件
- 按照用户要求对文件进行读写操作
- 对存放文件的存储空间进行管理，为各个文件自动分配必要的物理存储空间，并为文件的逻辑结构以及它在存储空间中的物理位置建立映照关系。
- 用户只需要通过文件名就可以对文件进行存放，文件的物理组织对用户是透明的。

但是应该强调的是，Nachos 只是提供了一个现代操作系统的实验平台和框架。目前实现的文件系统部分同其它部分一样，比较简单。但是 Nachos 为读者提供了进一步改进和发展的基础。它和 UNIX 操作系统课程中的文件系统有较大的区别。主要表现在：

1. 在磁盘的组织结构方面
   Nachos 中文件同样有其 inode 和一般存储磁盘块，即 Nachos 物理磁盘的扇区。但是它不象 UNIX 一样，将文件系统中所有文件的 inode 存放在一起，即 inode 区中。每个 Nachos 文件的 inode 占用一个单独的扇区，分散在物理磁盘的任何地方，同一般存储扇区用同样的方式进行申请和回收。它没有文件系统管理块，是通过位图（Bitmap）来管理整个磁盘上的空闲块。

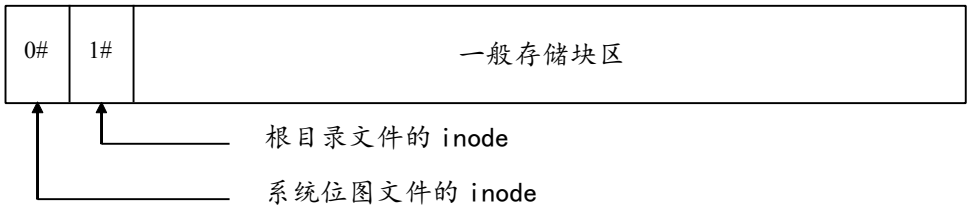


图 4.10　Nachos 文件系统磁盘分布

2. 在文件系统空闲磁盘数据块和 inode 块管理方面
   Nachos 中有一个特殊的文件，即位图文件。该文件存放的是整个文件系统的扇区使用情况的位图。如果一个扇区为空闲，则它在位图文件相应的位为 0，否则为 1。Nachos 中没有专门对 inode 扇区进行管理。

   当需要申请一个扇区时，根据位图文件寻找一个空闲的扇区，并将其相应的位置为 1。当释放一个扇区时，将位图中相应的位置为 0。位图文件是一个临界资源，应该互斥访问。现有的文件系统没有实现互斥访问，所以每次只允许一个线程访问文件系统。位图文件的 inode 占据 0 号扇区。

3. 在文件系统的目录管理方面
   一般的文件系统都采用树状目录结构，有的 UNIX 文件系统还有目录之间的勾连，形成图状文件系统结构。Nachos 则比较简单，只有一级目录，也就是只有根目录，所有的文件都在根目录下。而且根目录中可以存放的文件数是有限的。Nachos 文件系统的根目录同样也是通过文件方式存放的，它的 inode 占据了 1 号扇区。

4. 文件的索引结构上
   Nachos 同一般的 UNIX 一样，采用索引表进行物理地址和逻辑地址之间的转换，索引

表存放在文件的 inode 中。但是目前 Nachos 采用的索引都是直接索引，所以 Nachos 的最大文件长度不能大于 4K。

5. Nachos 文件系统除了在以上几点上有所不足外，还有以下一些不完善的地方：
   - 必须在文件生成时创建索引表。所以 Nachos 在创建一个文件时，必须给出文件的大小；而且当文件生成后，就不能改变文件的大小。
   - 目前该文件系统没有 Cache 机制
   - 目前文件系统的健壮性不够强。当正在使用文件系统时，如果突然系统中断，文件系统中的内容可能不保证正确。

第三节　Nachos文件系统的实现

Nachos 的文件系统是建立在 Nachos 的模拟物理磁盘上的，文件系统实现的结构如图 4.11 所示：

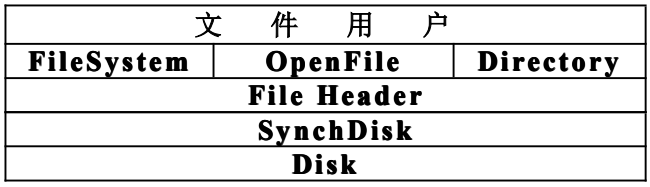


图 4.11　Nachos 的文件系统实现

在 Nachos 文件系统中，许多数据结构既可存放在宿主机内存里，又可存放在磁盘上。为了一致起见，文件系统中 Synchdisk 以上的类都有一个 FetchFrom 成员方法，它把数据结构从磁盘读到内存；还有一个 WriteBack 成员方法，与 FetchFrom 相反，它把数据结构从内存写回磁盘。在内存中的数据结构与磁盘上的完全一致，这给管理带来了不少方便。

1. 同步磁盘分析（文件synchdisk.cc 、synchdisk.h）

和其它设备一样，Nachos 模拟的磁盘是异步设备。当发出访问磁盘的请求后立刻返回，当从磁盘读出或写入数据结束后，发出磁盘中断，说明一次磁盘访问真正结束。

Nachos 是一个多线程的系统，如果多个线程同时对磁盘进行访问，会引起系统的混乱。所以必须作出这样的限制：

- 同时只能有一个线程访问磁盘
- 当发出磁盘访问请求后，必须等待访问的真正结束。

这两个限制就是实现同步磁盘的目的。

SynchDisk 的类定义和实现如下所示：

```
class SynchDisk {
  public:
    SynchDisk(char* name);                    // 生成一个同步磁盘
    ~SynchDisk();                             // 析构方法
    void ReadSector(int sectorNumber, char* data);  // 同步读写磁盘，只有当真正读写完毕
    void WriteSector(int sectorNumber, char* data); // 后返回
    void RequestDone();                       // 磁盘中断处理时调用
  private:
    Disk *disk;                               // 物理异步磁盘设备
    Semaphore *semaphore;                     // 控制读写磁盘返回的信号量
    Lock *lock;                               // 控制只有一个线程访问的锁
};
```

以 ReadSector 为例来说明同步磁盘的工作机制：

```
void SynchDisk::ReadSector(int sectorNumber, char* data)
{
    lock->Acquire();                    // 加锁（一次只允许一个线程访问磁盘）
    disk->ReadRequest(sectorNumber, data);      // 对磁盘进行读访问请求
    semaphore->P();                     // 等待磁盘中断的到来
    lock->Release();                    // 解锁（访问结束）
}
```

当线程向磁盘设备发出读访问请求后，等待磁盘中断的到来。一旦磁盘中断来到，中断处理程序执行 semaphore->V()操作，ReadSector 得以继续运行。对磁盘同步写也基于同样的原理。

2. 位图模块分析（文件bitmap.cc、bitmap.h）

在 Nachos 的文件系统中，是通过位图来管理空闲块的。Nachos 的物理磁盘是以扇区为访问单位的，将扇区从 0 开始编号。所谓位图管理，就是将这些编号填入一张表，表中为 0 的地方说明该扇区没有被占用，而非 0 位置说明该扇区已被占用。这部分内容是用 BitMap 类实现的。BitMap 类的实现比较简单，这里只是介绍其接口。

```
class BitMap {
  public:
    BitMap(int nitems);          // 初始化方法，给出位图的大小，将所有位标明未用
    ~BitMap();                   // 析构方法
    void Mark(int which);        // 标志第 which 位被占用
    void Clear(int which);       // 清除第 which 位
    bool Test(int which);        // 测试第 which 位是否被占用，若是，返回 TRUE
    int Find();                  // 找到第一个未被占用的位，标志其被占用；
                                 // 若没有找到，返回-1
    int NumClear();              // 返回多少位没有被占用
    void Print();                // 打印出整个位图（调试用）
    void FetchFrom(OpenFile *file);   // 从一个文件中读出位图
    void WriteBack(OpenFile *file);   // 将位图内容写入文件
  private:
    …                            // 内部实现属性
};
```

3. 文件系统模块分析（文件filesys.cc、filesys.h）

读者在增强了线程管理的功能后，可以同时开展文件系统部分功能的增强或实现虚拟内存两部分工作。在 Nachos 中，实现了两套文件系统，它们对外接口是完全一样的：一套称作为 FILESYS_STUB，它是建立在 UNIX 文件系统之上的，而不使用 Nachos 的模拟磁盘，它主要用于读者先实现了用户程序和虚拟内存，然后再着手增强文件系统的功能；另一套是 Nachos 的文件系统，它是实现在 Nachos 的虚拟磁盘上的。当整个系统完成之后，只能使用第二套文件系统的实现。

以下我们只分析在 Nachos 文件系统的实现：

```
class FileSystem {
  public:
    FileSystem(bool format);                    //  生成方法
    bool Create(char *name, int initialSize);   //  生成一个文件
    OpenFile* Open(char *name);                 //  打开一个文件
    bool Remove(char *name);                    //  删除一个文件
    void List();                                //  列出文件系统中所有的文件
                                                //  （实际上就是根目录中所有的文件）
    void Print();                               //  列出文件系统中所有的文件和它们的内
容
  private:
    OpenFile* freeMapFile;                      //  位图文件打开文件结构
    OpenFile* directoryFile;                    //  根目录打开文件结构
};
```

虽然说FileSystem中的Create、Open以及Remove方法类似于UNIX操作系统中的creat、open和unlink系统调用。但是在Nachos中，打开和创建文件没有给出打开和创建方式。这是因为在目前的Nachos文件系统中，没有用户分类的概念，也就没有同组用户和其它用户的概念。一个线程打开文件以后，就获得了对该文件操作所有的权利。大多数实用文件系统都提供对文件存取进行保护的功能，保护的一般方法是将用户分类、以及对不同类的用户规定不同的存取权。读者在Nachos提供的文件系统基础上，完全可以添加自己设计和实现的文件保护机制。

3.1 生成方法

语法：　FileSystem (bool format)

参数：　format:　是否进行格式化的标志

功能：　在同步磁盘的基础上建立一个文件系统。当 format 标志设置时，建立一个新的文件系统；否则使用原来文件系统中的内容。

实现：　1.　如果 format 标志没有设置，则使用原有的文件系统
　　打开位图文件和目录文件，返回

　　2　如果 format 标志设置，则生成一个新的文件系统

- 生成新的位图和空白的根目录
- 生成位图 FileHeader 和目录 FileHeader
- 在位图中标识 0 和 1 号扇区被占用（虽然此时还没有占用）
- 为位图文件和目录文件申请必要的空间，如果申请不到，系统出错返回
- 将位图 FileHeader 和目录 FileHeader 写回 0 和 1 号扇区（这时候位图文件和目录文件所在的位置已经确定）
- 打开位图文件和目录文件
- 将当前的位图和目录写入相应的文件中（位置确定，内容终于可以写入）而且这两个文件保持打开状态

返回：　无

3.2 Create方法

语法：　bool Create (char *name, int initialSize)
参数：　name:　　需要创建的文件名
　　initialSize:　　需要创建的文件的初始大小
功能：　在当前的文件系统中创建一个固定大小的文件
实现：　在根目录下搜寻该文件名
　　1.　如果搜索到，出错返回
　　2.　如果没有搜索到，
　　2.1. 申请文件 FileHeader 所需的空间，如果申请不到，出错返回
　　2.2. 将文件加入到目录文件中，如果失败，出错返回
　　2.3. 为新文件申请 FileHeader
　　2.4. 根据新文件的大小申请相应块数的扇区，如果申请不到，出错返回
　　2.5. 将所有有变化的数据结构写入磁盘
返回：　如果生成成功，返回 TRUE，否则返回 FALSE

显然，在 Nachos 的文件系统中，对目录对象和位图对象的操作应该是临界区操作。因为如果两个线程同时需要向同一个目录中写入一个文件，可能会出现两个线程同时申请到同一个目录项；在空闲块分配时，也会出现相类似的情况。但是目前 Nachos 没有对此进行处理。

3.3 Open方法

语法：　OpenFile * Open (char *name)
参数：　name:　　需要打开的文件名
功能：　在当前的文件系统中打开一个已有的文件
实现：　在根目录下搜寻该文件名
　　1.　如果没有搜索到，返回 NULL
　　2.　如果搜索到，打开该文件并返回打开文件结构
返回：　打开文件结构

3.4 Remove方法

语法：　bool Remove (char *name)
参数：　name:　　需要删除的文件名
功能：　在当前的文件系统中删除一个已有的文件
实现：　在根目录下搜寻该文件名
　　1.　如果没有搜索到，返回 FALSE
　　2.　如果搜索到，打开该文件并返回打开文件控制块
　　2.1. 将该文件从目录中删除
　　2.2. 释放 FileHeader 所占用的空间
　　2.3. 释放文件数据块占用的空间
　　2.4. 将对位图和目录的修改写回磁盘
返回：　如果删除成功，返回 TRUE；否则返回 FALSE

4. 文件头模块分析（文件filehdr.cc、filehdr.h）

文件头实际上就是 UNIX 文件系统中所说的 inode 结构，它给出一个文件除了文件名之外的所有属性，包括文件长度、地址索引表等等（文件名属性在目录中给出）。所谓索引表，就是文件的逻辑地址和实际的物理地址的对应关系。Nachos 的文件头可以存放在磁盘上，也可以存放在宿主机内存中。在磁盘上存放时一个文件头占用一个独立的扇区。Nachos 文件

头的索引表只有直接索引。

文件头的定义和实现如下所示，由于目前 Nachos 只支持直接索引，而且文件长度一旦固定，就不能变动。所以文件头的实现比较简单，这里不再赘述。

```
class FileHeader {
  public:
    bool Allocate(BitMap *bitMap, int fileSize);        // 通过文件大小初始化文件头
                                                        // 根据文件大小申请磁盘空间
    void Deallocate(BitMap *bitMap);                    // 将一个文件占用的数据空间释放
                                                        // （没有释放文件头占用的空间）
    void FetchFrom(int sectorNumber);                   // 从磁盘扇区中取出文件头
    void WriteBack(int sectorNumber);                   // 将文件头写入磁盘扇区
    int ByteToSector(int offset);                       // 文件逻辑地址向物理地址的转换
    int FileLength();                                   // 返回文件长度
    void Print();                                       // 打印文件头信息（调试用）
  private:
    int numBytes;                                       // 文件长度（字节数）
    int numSectors;                                     // 文件占用的扇区数
    int dataSectors[NumDirect];                         // 文件索引表
};
```

在 Nachos 中，每个扇区的大小为 128 个字节。每个 inode 占用一个扇区，共有 30 个直接索引。所以 Nachos 中最大的文件大小不能超过 3840 个字节。

5. 打开文件结构分析（文件openfile.cc、openfile.h）

该模块定义了一个打开文件控制结构。当用户打开了一个文件时，系统即为其产生一个打开文件控制结构，以后用户对该文件的访问都可以通过该结构。打开文件控制结构中的对文件操作的方法同 UNIX 操作系统中的系统调用。

针对 FileSystem 结构中的两套实现，这里的打开文件控制结构同样有两套实现。这里分析建立在 Nachos 上的一套实现：

```
class OpenFile {
  public:
    OpenFile(int sector);               // 打开一个文件，该文件的文件头在 sector 扇区
    ~OpenFile();                        // 关闭一个文件
    void Seek(int position);            // 移动文件位置指针（从文件头开始）
```

```
    int Read(char *into, int numBytes);  //  从文件中读出 numByte 到 into 缓冲
                                         //  同时移动文件位置指针（通过 ReadAt 实现）
    int Write(char *from, int numBytes);     //  将 from 缓冲中写入 numBytes 个字节到文件中
                                         //  同时移动文件位置指针（通过 WriteAt 实现）
    int ReadAt(char *into, int numBytes, int position);
                                         //  将从 position 开始的 numBytes 读入 into 缓冲
    int WriteAt(char *from, int numBytes, int position);
                                         //  将 from 缓冲中 numBytes 写入从 position 开始的区域
    int Length();                        //  返回文件的长度
  private:
    FileHeader *hdr;                     //  该文件对应的文件头（建立关系）
    int seekPosition;                    //  当前文件位置指针
};
```

5.1 ReadAt方法

语法：　int ReadAt (char *into, int numBytes, int position)
参数：　into:　　读出内容存放的缓冲
　　numBytes:　需要读出的字节数
　　position:　　需读出内容的开始位置
功能：　将从 position 开始的 numBytes 读入 into 缓冲
实现：　1.　计算实际需要读出的字节数
　　2.　计算出需要读出内容的扇区起始地址
　　3.　将这些扇区的内容读入一个内部缓冲
　　4.　将所需要的内容从缓冲中读出到 into 中
返回：　实际读出的字节数

5.2 WriteAt方法

语法：　int WriteAt (char *from, int numBytes, int position)
参数：　from:　　存放需写入内容的缓冲
　　numBytes:　需写入的字节数
　　position:　　需写入内容的开始位置
功能：　将 from 缓冲中的 numberBytes 字节从 position 开始的位置写入文件
实现：　1.　计算实际需要读出的字节数
　　2.　计算出需要读出内容的扇区起始地址
　　3.　申请一个内部缓冲
　　4.　将首尾扇区中不能修改内容先读入内部缓冲适当位置
　　5.　将需要写入文件的内容写入内部缓冲适当位置
　　6.　将内部缓冲中内容写入磁盘文件
返回：　实际写入的字节数

实际上，对文件的一次写操作应该是原子操作，否则会出现两个线程交叉写的状况。比如 A

线程和 B 线程都需要对扇区 a 和 b 进行修改。工作过程如下：

A（写 a）-> 线程切换 -> B（写 a） -> B（写 b）-> 线程切换 -> A（写 b）

这样扇区 b 中的内容是 A 线程写的，而扇区 a 的内容是 B 线程写的。这样，数据的一致性不能得到保证。但是目前 Nachos 没有对此进行处理。

6. 目录模块分析（文件directory.cc directory.h）

目录在文件系统中是一个很重要的部分，它实际上是一张表，将字符形式的文件名与实际文件的文件头相对应。这样用户就能方便地通过文件名来访问文件。

Nachos 中的目录结构非常简单，它只有一级目录，也就是只有根目录；而且根目录的大小是固定的，整个文件系统中只能存放有限个文件。这样的实现比较简单，这里只介绍目录的接口：

```
class DirectoryEntry {                  //  目录项结构
  public:
    bool inUse;                         //  该目录项是否在使用标志
    int sector;                         //  对应文件的文件头位置
    char name[FileNameMaxLen + 1];      //  对应文件的文件名
};

class Directory {
  public:
    Directory(int size);                //  初始化方法，size 规定了目录中可以放多少文件
    ~Directory();                       //  析构方法
    void FetchFrom(OpenFile *file);     //  从目录文件中读入目录结构
    void WriteBack(OpenFile *file);     //  将该目录结构写回目录文件
    int Find(char *name);               //  在目录中寻找文件名，返回文件头的物理位置
    bool Add(char *name, int newSector); // 将一个文件加入到目录中
    bool Remove(char *name);            //  将一个文件从目录中删除
    void List();                        //  列出目录中所有的文件
    void Print();                       //  打印出目录中所有的文件和内容（调试用）
  private:
    int tableSize;                      //  目录项数目
    DirectoryEntry *table;              //  目录项表
    int FindIndex(char *name);          //  根据文件名找出该文件在目录项表中的表项序号
};
```

第四节　文件管理系统作业

目前 Nachos 文件系统所提供的功能不强，所以，我们的任务是在若干方面尽可能加强其功能，预定的目标是：

1. 实现可以多线程同时访问的同步文件系统，所实现的文件系统必须符合以下的几点：
   1.1. 一个文件可以同时被多个线程访问。每个线程独自打开文件，独自拥有一个当前文件访问位置。不同的线程之间不会互相干扰。
   1.2. 所有对文件系统的操作必须是原子操作和序列化的。比如，当一个线程正在对一个文件进行修改而另一个线程正在读取文件的内容时。读取文件内容的线程要么读出修改过的文件，要么读出原来的文件，不存在不可预计的中间状态。另外，假设对一个文件的 OpenFile::Write 操作在 OpenFile::Read 操作之前完成，Read 操作必须反映 Write 操作的结果。
   1.3. 当某一线程欲删除一个文件时，如果另外一些线程正在访问该文件。那么这些线程仍然可以对该文件进行操作。直到所有这些线程关闭了这个文件。该文件才被删除。也就是说，只要有一个线程还打开着这个文件，该文件的物理空间就没有真正地被删除。

2. 修改现有的文件系统使得文件系统中的文件长度可以达到几乎无限大，例如在 1G 或 2G。在现有的系统中，一个文件的大小被限制在 4K 以内。为了实现非常大的文件，我们必须修改 FileHeader 中的关于索引文件表的部分。注意修改过的文件头不要超过一个扇区的大小。

3. 实现可以修改的文件长度机制。在现有的系统中，一个文件一旦被生成，即不能修改文件长度。在 UNIX 和其它的一些操作系统中，文件刚生成时，其文件长度为 0。以后可以不断地被修改。新文件系统也要有这样的功能。

4. 提高文件系统的性能。读者可以在宿主机内存中开辟一块独立的区域作为磁盘访问的缓冲区以提高文件系统的性能。

5. 实现文件系统的多级目录结构。当前的文件系统是一级目录结构，也即所有的文件都放在同一个目录之下，这种目录结构过于简单，不能满足实际应用的需要。我们的目标是实现类似 UNIX 的多级目录，并实现目录之间的勾连。

6. 修改当前空闲区管理方法，采用 UNIX 中通常用的栈式管理，并比较这两种空闲区管理方法对文件系统性能的影响。

7. 在现有的文件系统中增加灾难保护机制，当系统崩溃时，保证文件系统在下一次重新启动时是完整的（当然有些文件可能会因为崩溃而丢失）。这可以通过在目前文件系统上，编写一些应用程序来实现。（如 DOS 中的 defrag, chkdsk, scandisk 等等）

8. 为新文件系统增加用户保护机制。即将用户分类，每类用户有不同的文件访问权限。

第五章　用户程序和虚拟内存

第一节　Nachos 对内存、寄存器以及CPU的模拟

在第二章中，我们讲述了 Nachos 的机器模拟。这里补充讲述 Nachos 对内存、寄存器以及 CPU 指令解释器的分析，涉及到机器模拟部分的 machine.cc、machine.h、translate.cc、translate.h、mipssim.cc 和 mipssim.h 等文件。

Nachos 机器模拟很重要的部分是内存和寄存器的模拟。Nachos 寄存器组模拟了全部 32 个 MIPS 机（R2/3000）的寄存器，同时加上有关 Nachos 系统调试用的 8 个寄存器，以期让模拟更加真实化并易于调试，对于一些特殊的寄存器说明如下：

寄存器名	编号	描述
StackReg	29	用户程序的堆栈指针
RetAddrReg	31	存放过程调用的返回地址
HiReg	32	存放乘法结果的高32位
LoReg	33	存放乘法结果的低32位
PCReg	34	当前PC指针
NextPCReg	35	下一条执行语句的PC指针
PrevPCReg	36	上一条执行语句的PC指针（调试用）
LoadReg	37	需要延迟载入的寄存器编号
LoadValueReg	38	需要延迟载入的寄存器值
BadAddReg	39	当出错陷入（Exception）时用户程序的逻辑地址

Nachos 用宿主机的一块内存模拟自己的内存。为了简便起见，每个内存页的大小同磁盘扇区的大小相同，而整个内存的大小远远小于模拟磁盘的大小。由于 Nachos 是一个教学操作系统，在内存分配上和实际的操作系统是有区别的。事实上，Nachos 的内存只有当需要执行用户程序时用户程序的一个暂存地，而作为操作系统内部使用的数据结构不存放在 Nachos 的模拟内存中，而是申请 Nachos 所在宿主机的内存。所以 Nachos 的一些重要的数据结构如线程控制结构等的数目可以是无限的，不受 Nachos 模拟内存大小的限制。

这里需要强调的是，此处 Nachos 模拟的寄存器组同 Thread 类（第三章第三节）中的 machineState[]数组表示的寄存器组不同，后者代表的是宿主机的寄存器组，是实际存在的；而前者只是为了运行拥护程序模拟的。

在用户程序运行过程中，会有很多系统陷入核心的情况。系统陷入有两大类原因：进行系统调用陷入和系统出错陷入。系统调用陷入在用户程序进行系统调用时发生。系统调用可以看作是软件指令，它们有效地弥补了机器硬件指令不足；系统出错陷入在系统发生错误时发生，比如用户程序使用了非法指令以及用户程序逻辑地址同实际的物理地址映射出错等情况。不同的出错陷入会有不同的处理，比如用户程序逻辑地址映射出错会引起页面的重新调入，而用户程序使用了非法指令则需要向用户报告等等。Nachos 处理的陷入有：

SyscallException	系统调用陷入
PageFaultException	页面转换出错
ReadOnlyException	试图访问只读页面
BusErrorException	总线错，转换用户程序页面时出错
AddressErrorException	页面访问没有对齐，或者超出了页面的大小
OverflowException	加减法时整数溢出
IllegalInstruException	非法指令访问

模拟机的机器指令由操作代码和操作数组成的，其类定义和实现如下所示：

```
class Instruction {
  public:
    void Decode();          // 将指令的二进制表示转换成系统方便处理的表示
    unsigned int value;     // 指令的二进制表示
    char opCode;        // 分析出的操作代码
    char rs, rt, rd;        // 分析出的指令的三个寄存器的值
    int extra;              // 分析出的指令立即数
};
```

Machine 类的定义和实现如下所示：

```
class Machine {
  public:
    Machine(bool debug);                    // 初始化方法
    ~Machine();                         // 析构方法
    void Run();                         // 运行一个用户程序
    int ReadRegister(int num);                  // 读出寄存器中的内容
    void WriteRegister(int num, int value);         // 向一个寄存器赋值
    void OneInstruction(Instruction *instr);        // 执行一个用户程序指令
    void DelayedLoad(int nextReg, int nextVal);     // 执行一次延迟载入
    bool ReadMem(int addr, int size, int* value);   // 读出内存 addr 地址中的
内容
    bool WriteMem(int addr, int size, int value);   // 向内存 addr 地址中写入
内容
    ExceptionType Translate(int virtAddr, int* physAddr, int size,bool writing);
                                        // 将用户程序逻辑转换成物理地址
    void RaiseException(ExceptionType which, int badVAddr);
                                        // 执行出错陷入处理程序
    void Debugger();                    // 调用用户程序调试器
    void DumpState();                       // 打印机器寄存器和内存状态
    char *mainMemory;                       // 模拟内存
```

```
    int registers[NumTotalRegs];        // CPU寄存器模拟
    TranslationEntry *tlb;              // TLB页面转换表
    TranslationEntry *pageTable;        // 线性页面转换表
    unsigned int pageTableSize;         // 线性页面转换表大小
  private:
    bool singleStep;                    // 单步执行标志
    int runUntilTime;                   // 调试时钟
};
```

需要注意的是，虽然这里的很多方法和属性规定为 public 的，但是它们只能在系统核心内被调用。定义 Machine 类的目的是为了执行用户程序，如同许多其它系统一样，用户程序不直接使用内存的物理地址，而是使用自己的逻辑地址，在用户程序逻辑地址和实际物理地址之间，就需要一次转换，系统提供了两种转换方法的接口：**线性页面地址转换方法**和 **TLB 页面地址转换方法**。

无论是线性页面地址转换还是 TLB 页面地址转换，都需要一张地址转换表，地址转换表是由若干个表项（Entry）组成的。每个表项记录程序逻辑页到内存实页的映射关系，和实页的使用状态、访问权限信息。类 TranslationEntry 描述了表项的结构：

```
class TranslationEntry {
  public :
    int  virtualPage;     // 逻辑页号
    int  physicalPage;    // 内存物理页号
    boot valid;           // 该 Entry 是否使用，TRUE 表示在使用
    bool readOnly;        // 对应页的访问属性，TRUE 示只读，否则为读写
    bool use;             // 该 Entry 是否被使用过，每次访问后置为 TRUE
    bool dirty;           // 对应的物理页使用情况，TRUE 表示被写过
}
```

线性页面地址转换是一种较为简单的方式，即用户程序的逻辑地址同实际物理地址之间的关系是线性的。在作转换时，给出逻辑地址，计算出其所在的逻辑页号和页中偏移量，通过查询转换表（实际上在使用线性页面地址转换时，TranslationEntry 结构中的 virtualPage 是多余的，线性页面转换表的下标就是逻辑页号），即可以得到实际物理页号和其页中偏移量。在模拟机上保存有线性页面转换表，它记录的是当前运行用户程序的页面转换关系；在用户进程空间中，也需要保存线性页面转换表，保存有自己运行用户程序的页面转换关系。当其被切换上模拟处理机上运行时，需要将进程的线性页面转换表覆盖模拟处理机的线性页面转换表。线性页面转换的过程如图 5.1 所示。

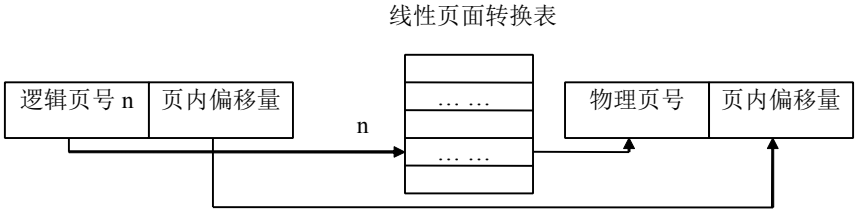


图 5.1 线性页面转换机制

TLB 页面转换则不同，TLB 转换页表是硬件来实现的，表的大小一般较实际的用户程序所占的页面数要小，所以一般 TLB 表中只存放一部分逻辑页到物理页的转换关系。这样就可能出现逻辑地址转换失败的现象，会发生 PageFaultException 异常。在该异常处理程序中，就需要借助用户进程空间的线性页面转换表来计算出物理页，同时 TLB 表中增加一项。如果 TLB 表已满，就需要对 TLB 表项做 LRU 替换。使用 TLB 页面转换表处理起来逻辑较线性表为复杂，但是速度要快得多。由于 TLB 转换页表是硬件实现的，所以指向 TLB 转换页表的指针应该是只读的，所以 Machine 类一旦实例化，TLB 指针值不能改动。

在实际的系统中，线性页面地址转换和 TLB 页面地址转换只能二者取一，目前为简便起见，Nachos 选择了前者，读者可以自行完成 TLB 页面地址转换的实现。

另外，由于 Nachos 可以在多种平台上运行，各种运行平台的数据表达方式不完全一样，有的是高位在前，有的是高位在后。为了解决跨平台的问题，特地给出了四个函数作数据转换，它们是 WordToHost、ShortToHost、WordToMachine 和 ShortToMachine，这里不作详细的解释。

1 RaiseException 方法

语法： void RaiseException (Exception which, int badVAddr)
参数： which: 系统出错陷入的类型
badVAddr: 系统出错陷入所在的位置
功能： 当系统检查到出错陷入时调用，通过调用 ExceptionHandler 来处理陷入。（见 exception.cc）ExceptionHandler 目前对所有的陷入都不进行处理，需要进一步加强。如上面提到的，如果使用 TLB 表时，应该对 PageFaultError 作出处理。
实现： 调用 ExceptionHandler。
返回： 无。

2 ReadMem 方法

语法： bool ReadMem (int addr, int size, int *value)
参数： addr: 用户程序逻辑地址
size: 需要读出的字节数
value: 读出的内容暂存地
功能： 从用户逻辑地址读出 size 个字节，转换成相应的类型，存放在 value 所指向的空间中。
实现： 调用 Translate 方法。
返回： 读取是否成功。

3 WriteMem 方法

语法： bool WriteMem (int addr, int size, int *value)
参数： addr: 用户程序逻辑地址
size: 需要写入的字节数
value: 需要写入的内容
功能： 将 value 表示的数值根据 size 大小转换成相应的机器类型存放在 add 用户逻辑地址中
实现： 调用 Translate 方法。
返回： 写入是否成功。

4 Translate方法

语法： Exception Translate (int virtAddr, int *physAddr, int size, bool writing)
参数： virtAddr: 用户程序的逻辑地址
physAddr: 转换后的实际地址
size: 数据类型的大小
writing: 读/写内存标志
功能： 将用户的逻辑地址转换成实际的物理地址，同时需要检查对齐。
实现：
1. 判断用户逻辑地址是否对齐
   1.1. 如果 size 是 2，virtAddr 必须是 2 的倍数
   1.2. 如果 size 是 4，virtAddr 必须是 4 的倍数
   没有对齐则返回 AddressErrorException
2. 计算出虚拟地址所在的页号 vpn 及其在页面中的偏移量
3. 根据采用不同的转换方法作不同的处理
   3.1. 如果采用的是线性转换表
      3.1.1.当 vpn>=pageTableSize 时，即虚拟页数过大，返回 AddressErrorException
      3.1.2.当页表中显示该页为无效时，返回 PageFaultException
      3.1.3.一切正常则得到相应的页表表项
   3.2. 如果采用的是 TLB 转换表，查找 TLB 表
      3.2.1.如果查找到，得到相应的页表表项
      3.2.2.如果没有查找到，返回 PageFaultException
4. 如果得到的页表表项是只读，writing 标志设置，返回 ReadOnlyException
5. 如果表项中相应的物理地址大于实际的内存物理地址，返回 BusErrorException
6. 设置表项正在使用标志，如果 writing 标志设置，设置表项中的 dirty 标志
7. 返回 NoException

返回： 某一种出错陷入，或一切正常标识。

5 Run 方法

语法： Void Run ()
参数： 无
功能： 执行在模拟内存内的用户程序。
实现：
1. 将系统当前状态设置为用户模式
2. 取出一条指令（调用 OneInstruction 方法）
   2.1 将该指令进行解码，分析出其中的操作代码、寄存器和立即数
   2.2 根据操作代码将该指令模拟执行
   2.3 时钟前进一个单位（Tick）
3. 转向 2，直到用户程序执行完毕。

返回： 无

第二节 Nachos用户进程运行机制

一、用户程序空间（文件address.cc, address.h）

在第三章第三节中 Nachos 线程管理中提到，Nachos 的用户进程由两部分组成：核心部分和用户程序部分。核心部分同一般的系统线程没有区别，它共用了 Nachos 的正文段和数据段，运行在宿主机上；而用户程序部分则有自己的正文段、数据段和栈段，它存储在 Nachos 的模拟内存中，运行在 Nachos 的模拟机上。在控制结构上，Nachos 的用户进程比系统线程多了以下内容：

```
int userRegisters[NumTotalRegs];     // 虚拟机的寄存器组
void SaveUserState();                // 线程切换时保存虚拟机寄存器组
void RestoreUserState();             // 线程切换时恢复虚拟机寄存器组
AddrSpace *space;                    // 线程运行的用户程序
```

其中，用户程序空间有 AddrSpace 类来描述：

```
class AddrSpace {
  public:
    AddrSpace(OpenFile *executable);   // 根据可执行文件构成用户程序空间
    ~AddrSpace();                      // 析构方法
    void InitRegisters();              // 初始化模拟机的寄存器组
    void SaveState();                  // 保存当前机器页表状态
    void RestoreState();               // 恢复机器页表状态
  private:
    TranslationEntry *pageTable;       // 用户程序页表
    unsigned int numPages;             // 用户程序的虚页数
};
```

本书所描述的 Nachos 系统是实现在 Linux 操作系统上的。在 Linux 系统中，使用 gcc 交叉编译技术将 C 程序编译成 R2/3000 可以执行的目标代码，通过 Nachos 提供的 **coff2noff** 工具将其转换成 Nachos 可以识别的可执行代码格式，拷贝到 Nachos 的文件系统中才能执行。

1.1 生成方法

语法： AddrSpace (OpenFile *executable)
参数： Executable: 需要执行代码的打开文件结构
功能： 初始化用户程序空间。
实现：
1. 判断打开文件是否符合可执行代码的格式，如果不符合，出错返回
2. 将用户程序的正文段、数据段以及栈段一起考虑，计算需要空间大小。如果大于整个模拟的物理内存空间，出错返回。
3. 生成用户程序线性页表。
4. 将用户程序的正文段和数据段依次调入内存，栈段记录的是用户程序的运行状态，它的位置紧接于数据段之后。

返回： 无。

目前 Nachos 在运行用户程序时，有如下的限制：

- 系统一次只能有运行一个用户程序，所以目前的线性转换页表比较简单，虚拟页号同物理页号完全一样。当读者需要加强这部分内容时，需要增加内存分配算法。
- 系统能够运行的用户程序大小是有限制的，必须小于模拟的物理内存空间大小，否则出错。在虚拟内存实现以后，这部分内容也将做改动。

1.2 InitRegisters方法

语法： Void InitRegisters ()
参数： 无。
功能： 初始化寄存器，让用户程序处于可以运行状态。
实现： 设置 PC 指针、栈指针的初值，并将其它寄存器的值设置为 0。
返回： 无。

1.3 SaveState方法

语法： Void SaveState ()
参数： 无。
功能： 存储用户程序空间的状态。
实现： 目前为空。
返回： 无。

1.4 RestoreState方法

语法： Void RestoreState ()
参数： 无。
功能： 恢复处理机用户程序空间的状态。
实现： 赋值实现。
返回： 无。

目前系统存储和恢复用户程序空间的实现非常简单，这是因为系统一次只能运行一个用户程序的局限和使用了线性页面转换表而决定的。

当用户程序空间初始化之后（设由 space 指针指向），真正的启动运行过程如下：

```
SPACE -> INITREGISTERS();        //   初始化模拟机寄存器组
space -> RestoreState();         //   恢复处理机用户程序空间的状态，实际上是将
                                 //   用户程序空间的转换页表覆盖模拟机的转换页表
machine -> Run();                //   运行用户程序
```

二、系统调用（文件exception.cc, syscall.h, start.s）

前面提到过，系统陷入有两大原因：即系统调用和系统出错陷入。虽然计算机硬件本身提供了硬件指令，用户可以通过组织这些硬件指令来开发所有的系统软件和应用软件。但是在软件开发过程中我们发现，有些指令序列需要重复使用，正如在结构化程序开发中，函数可以被重复调用一样。我们将其中一些同硬件或者和操作系统功能有密切联系的一部分常用指令序列抽取出来，使它们成为操作系统本身的一部分，这就是所谓的系统调用。所以系统调用

也被看作软件指令，它是在用户态下运行的程序和操作系统的界面。用户态程序可以通过系统调用获得操作系统提供的各种服务。

一般的 UNIX 操作系统提供几十条甚至上百条系统调用，系统调用的多少取决于操作系统的复杂程度及其主要的适用范围。只有用户态的程序才能进行系统调用，进行系统调用后的状态变成系统态，一般程序在系统态运行会有较高的优先级。系统调用作为操作系统的一部分，它们是长驻内存的。操作系统提供的系统调用太少，会影响用户程序的执行效率和用途；因为不是每种应用都会用到所有的系统调用，系统调用数目太多，则会使得操作系统的核心过于庞大而同样影响系统的效率。现代的很多操作系统提出了 mini-kernel 甚至 micro-kernel 的思想，在核心系统调用的选用方面比较谨慎，一般只选用最基本的部分，如网络、线程管理的与核心密切相关的部分，而将其它部分转化成开发系统的一部分，作为开发系统的标准函数提供，运行在用户态下。这样就可以减轻操作系统的负担，增加操作系统的灵活性、可配置性以及分布式应用开发的简便性等。

在 Nachos 中，系统调用用其它异常陷入的入口处理函数都是 ExceptionHandler 函数，只是陷入的类型为 SyscallException。同样的入口函数如何来确定具体是什么系统调用呢？在 test 目录下的 start.s 模块中描述了具体的系统调用过程，以 Halt 系统调用为例：

```
.GLOBL      HALT
.ENT        HALT
HALT:  ADDIN        $2, $0, SC_HALT
SYSCALL
J       $31
   .end     Halt
```

其中 addin $2, $0, SC_Halt 语句的作用是在 r2 寄存器中存放系统调用的类别码 SC_Halt，即 Halt 系统调用。start.s 同 ExceptionHandler()配合使用，完成整个调用过程：

```
void ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);                    // 取出系统调用代码
    if ((which == SyscallException) && (type == SC_Halt)) { // 进行 Halt 系统调用
        interrupt->Halt();
    } else {
        非法异常出错，返回。
    }
}
```

Halt 系统调用是没有参数的，对于有参数的系统调用，MIPS 编译决定了参数传递的以下规则：

参数 1： r4 寄存器
参数 2： r5 寄存器
参数 3： r6 寄存器
参数 4： r7 寄存器

如果系统调用有返回值，根据 MIPS 的标准 C 调用习惯，返回值存放在 r2 寄存器中。

第三节　虚存管理的设计和实现

一、Nachos存储管理的改进要求

作为现代操作系统，对其存储管理系统的主要要求有四：

1. 提供多道程序的运行环境，从而提高系统的使用效率；
2. 使程序的可使用空间不受物理内存大小的限制，即使系统具备运行大程序的能力，从理论上说一个程序的可用空间是无限的；
3. 提供存储访问保护功能，这主要是为了防止数据的越权读写；
4. 具有良好的性能

这样，就需要同学们在原有的存储系统的基础上实现虚存管理。所谓**虚存管理**，就是用一部分的磁盘作为物理内存的扩充。用户程序在执行的过程中，其图象可能在物理内存中，也可能在虚拟内存中。

二、一个虚拟存储管理实现的实例

2.1 虚拟存储系统的总体设计

采用一部分磁盘作为虚拟内存的使用方法有两种方式：一种是将物理内存和虚拟内存合并起来，作为一个整体使用；另一种是将实际的物理内存作为虚拟内存的 cache。前一种方式，系统可以运行的最大用户程序会比后一种大一些，但是后一种方式实现起来较为简单。该实例采用后一种方式。读者可以自行设计和实现前一种方式。

目前流行的操作系统开辟虚拟内存有三种方式：

1. 在物理磁盘上开辟单独的一块作为虚拟内存（有别于文件系统）
2. 将一个独立的磁盘分区作为虚拟内存
3. 将文件系统中的一个文件作为虚拟内存

这三种方式各有千秋，通过物理磁盘来实现一般效率较高，而第一种方式比第二种方式少占用磁盘分区，第二种方式实现较为简单。第三种方式灵活性较好，可以动态增加虚拟内存的大小。这三种方式在目前一些常用的操作系统，如下表：

表 5.1　一些常用操作系统的虚拟内存管理方式

操作系统	使用虚存管理方式
SCO UNIX	1
LINUX	2
SUN Solaris	1, 2, 3
SGI IRIX	1, 2, 3
MS-WINDOWS 系列	1, 3

需要说明的是 MS-WINDOWS 系列，采用的虚拟内存的方式表面上都借用了文件系统中的文件，但是采用了两种方式，以 WINDOWS 3.1 为例，分别采用 Permanent 和 Temporary 虚存文件两种方式。Temporary 虚存管理，虚存文件实际上是一个普通文件；而 Permanent 虚存文件一旦生成，则由操作系统使用另一种效率更高的管理方式，这里将这种方式归入 1。

这里 Nachos 的虚存管理改进采用第二种方式，这是因为；

1. 对于 swap 区的操作提供自己的处理方式要比使用文件系统的功能获得更好的性能；
2. 原始的 Nachos 本身提供的功能十分简单，其最初的文件系统也是需要改进的，将 swap 区和文件系统完全分离将使系统更加灵活和可伸缩，同时也可以并行地开发虚存管理系统和文件系统。

改进后的虚存管理系统使用分页管理算法，实页和虚页通过各自的页表进行管理。总的说来，系统中有三类页表，**实页页表**、**虚页页表**和**用户程序页表**。实页页表和虚页页表整个系统只有一张，由操作系统维护，在 Nachos 中，实页页表和虚页页表都占用宿主机的内存；每个用户进程执行的进程图象都有一张页表，该页表在其进程控制结构中，同样是占用宿主机的内存。实页页表中的每个页表项对应于一个实页，记录该实页的使用状态。虚页页表中的每个页表项对应于一个虚页，记录虚页的映射信息、使用状态等情况；每个虚页内容或者对应于一个实页内容，或者不在内存里；线程页表中的每一项都对应于一个虚页。系统中的共享层次位于虚页页表和线程页表之间，多个线程页表项可共享同一个虚页，而一个虚页最多只能对应于一个实页。系统页表的总体布局如图 5.1 。

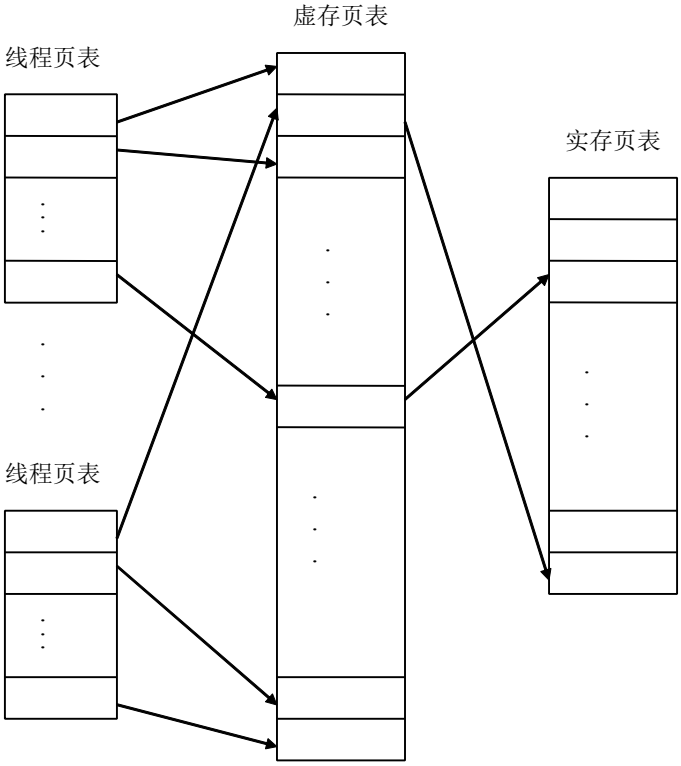


图 5.1　虚拟存储系统页表转换

程序运行时的地址变换就是通过系统的三级页表实现，变换过程如图 5.2。

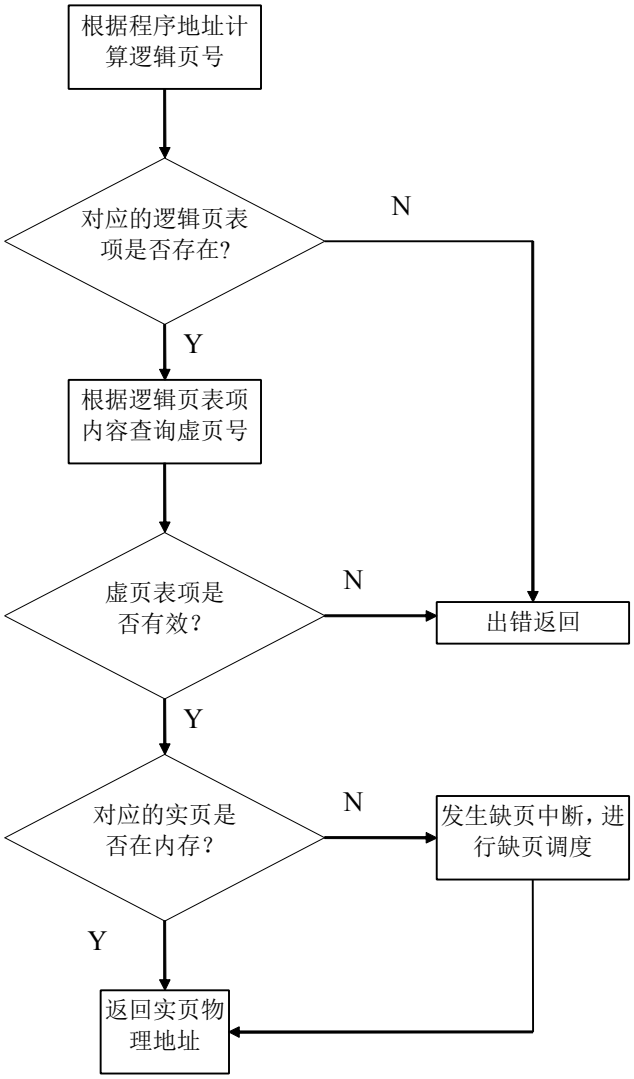


图 5.2　虚实地址变换

当用户程序运行时，发现有页面在虚存中而不在实存中，会发生缺页中断陷入，需要将虚存中的内容调入实存;但是由于实存要比虚存小,可能就有些内容必须从实存中调出到虚存中。这里可以采用的技术有全相联技术、组相联技术以及块组相联技术。

- 全相联技术

当一个页面需要从虚存调入实存时，从头扫描整个实存，看有无空闲实页。如果有，就将该虚页内容拷贝到该空闲实页中；如果没有，将某个实页调入对应的虚页。也就是说,虚拟内存中的任何一页可以被调入实存的任何一页。

- 组相联技术

将虚拟内存中的页分成同实存中页数相同的组，每一组同实存中的一页相对应。当某虚页需要调入实存中，它只能够调入和它所在组对应的实页中。

- 块组相联技术

将虚存和实存分成相同数目的组，当然虚存中一组中的页数会大于实存中一组的数目，每一组虚页同一组实页相对应。当某虚页需要调入实存中，它能够调入和它所在组对应

的实页组中任何一页中。

全相联技术在实存的查找方面效率较低；组相联技术可能会引起页面经常调入调出，同样影响效率，还可能出现执行一条指令需要相关的页总是不能同时调入实存（比如这两页属于同一个组）而不能正常运行的情况。所以该实例中虚存管理采用的是块组相联技术。实际上，实现了块组相联，通过适当的宏配置就可以实现全相联和组相联，因为全相联和组相联不过是块组相联的一个特例而已。读者可以自己实现这三种技术，并对它们作一个比较。

块组相联的技术要点在于如何选择组的大小，目前把每组中的实页数定为 4，整个系统共有 8 组实页。每组中的页数是一个经验值，取值的大小取决于系统实际运行的效率。系统对于同组的实页和虚页使用 Hash 表进行管理，通过 Hash 表来加速虚存的分配回收以及缺页调度算法。

2.2 缺页中断陷入及其调度算法

对于缺页中断陷入，首要解决的问题是确定陷入的发生时机。如果缺页中断陷入发生在指令执行前，这就需要在执行指令前对指令的访问进行预判；如果缺页中断陷入发生在指令执行中，则如何保存指令执行间的状态就是一个主要的问题。实现的虚存管理系统采用第二种方法，这是因为 Nachos 的每个线程都有独立的核心堆栈，因此指令执行时中间状态保存就比较容易。中断的具体发生时机在每次地址转换时，当发现访问目标不在内存中，就发生缺页中断陷入。

缺页调度算法在虚存管理系统中具有举足轻重的地位，虚拟存储管理系统的性能取决于系统的平均访问时间 T，该参数可以通过下式计算：

$$T = T1 \times H + T2 \times (1-H) \qquad (5\cdot1)$$

其中 T1 是命中的访问时间，H 是命中率，T2 是不命中的访问时间。对于虚拟存储管理来说，命中的访问时间接近于是物理内存的访问时间，而不命中时需要调页，访问时间就长得多，一般为命中访问时间的上百倍，现假设这个比例恰为 N (N>>1)，则命中率对系统平均访问时间 T 的影响基本上如图 5.3 所示。

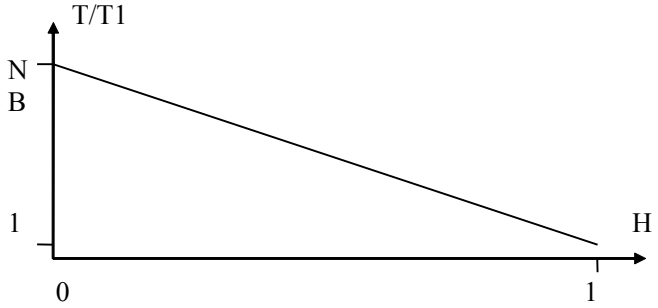


图 5.3　实页命中率对平均访问时间的影响

由上图可知，提高虚拟存储管理性能的关键在于提高系统的页访问命中率。提高页命中率包括两部分的内容：

1. 提高进程创建时的热启动页命中率，就是当一个进程创建时，如何为它分配存储空间以提高起始运行一段时间内的页命中率；
2. 选择比较好的页调度算法以提高系统的页命中率。

对于前者，当一个新进程创建时，只要有实页总是分配实页。若没有则分配虚页，调用通用的虚存分配函数而不进行特殊的处理。

虚存管理系统的页调度算法包括两方面的内容，预调算法和请调算法，这里的存储管理不考虑预调算法，而把注意力集中在请调算法，也就是缺页调度算法上。当前的缺页调度算法采用近似的 LRU 算法，实页的控制结构增加一个参数表示该实页的 LRU 值，作为访问计数。当发生缺页中断陷入时，将同组中 LRU 值最小的实页调出内存，把不命中的页调入内存，其处理流程如图 5.4。

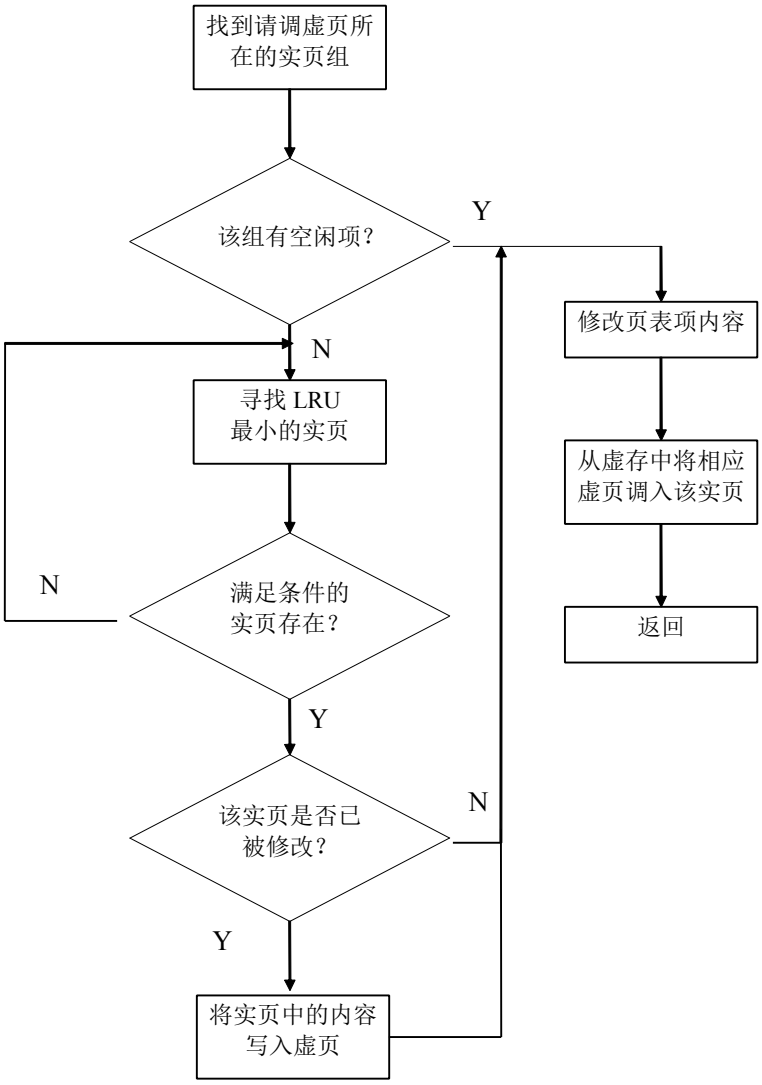


图 5.4　缺页中断陷入的处理过程

系统通过以下的机制对实页的 LRU 值进行管理，从而保证了近似 LRU 替换算法的实现：
LRU 值的初始化：

- 当一个实页被分配时，给该实页的 LRU 值赋一个初值
- 当一个虚页由于缺页中断陷入被调入内存时，给对应实页的 LRU 值置初值

LRU 值的更新：

- 当一个实页被访问时，该实页的 LRU 值加 1 操作
- 每次时钟中断时对所有实页的 LRU 值进行刷新，目前使用的除 2 操作

实页 LRU 值初始赋值的目的是为了防止新调入的页很快的被调出内存，从而引发系统性能的颠簸。LRU 值更新的目的是为了使实页的 LRU 值的变动能比较实际的反映实页的使用情况，从而使近似的算法能获得较好的实际效果。

2.3 虚存的存储分配

在页式管理系统中，系统存储资源的分配和释放都是以页为单位，存储分配和回收的算法比较简单。系统中选择 Bitmap 对实页表和虚页表进行管理，为了使分配的实页项和虚页项比较均匀地分布在整个表中，寻找空闲页表项时使用循环算法，而不是每次从头寻找。由于虚实页的映射采用块组相联技术，所以在分配虚存空间时就不能单纯的通过寻找一个空闲的虚页表项来进行。这是因为在系统尚有空闲实页时，有空闲的虚页表项并不说明对应于该虚页有空闲的实页。分配虚存时要充分利用空闲的实页以提高物理内存的利用率，再同时分配相应的虚页，从而提高系统性能。因此在分配虚存时，先寻找是否有空闲的实页，若有则优先分配实页，当没有空闲实页时才寻找空闲的虚页。

2.4 存储保护

作为操作系统，存储访问保护也是不可缺少的一部分，Nachos 的存储保护包括双重策略：一是程序地址空间的隔离，每个线程有自己的页表，当对地址进行访问时作的程序逻辑页到系统虚页的地址变换仅使用这张页表，访问的地址空间一旦超出线程的地址空间就会出错，这样除了线程共享的代码段和共用数据段，每个线程占用的存储区不会重叠；二是对访存实行存取控制，存取控制包括访问优先级控制和访问权限控制，实现时没有考虑访问页优先级的设定，只对每个页表表项规定该页的读写访问权利，目前规定的权利有只读和读写两种，控制层次在用户程序的程序页表上。

2.5 实现中的一些细节

对于虚拟存储管理系统的实现，除了在设计时需要精心分析，实际实现中还会遇到一些问题，需要使用各种技巧来解决。

问题一：如何保证系统正在使用的实页不被调出内存？

当系统对某些实页进行读写等操作时，由于等待操作完成进入睡眠状态，这时被操作的实页不能被调出内存，其理由是显而易见的。为了解决这个问题，在实页的控制结构中增加了一个参数表示实页的引用计数，每当系统使用某实页时将该实页的引用计数加 1，使用完毕将引用计数减 1，而缺页调度算法进行调度时不考虑引用计数非 0 的实页，从而使得系统正在使用的实页不被替换出内存。

问题二：如何保证在指令的执行中，该指令使用到的所有实页都在内存中？

对于缺页中断陷入发生在指令执行前的操作系统，系统在执行指令前必须判断指令引用到的所有实页是否在内存中，若缺页则发生缺页中断陷入，这时就需要保证该指令引用到的已在内存中的实页不被替换出去；对于缺页中断陷入发生在指令执行间的操作系统，当指令执行中发生缺页中断陷入时，需要保证已被指令引用到的实页不被替换出内存。对于上述两者，如果这个要求得不到保证，那么可能在缺页中断陷入完成后再次发生缺页中断陷入，造成系统性能的颠簸。解决这个问题用到了上面引入的实页的引用计

数，在指令的执行过程中，每次访问内存就对于访问的实页增加引用计数，在一条指令执行完毕后释放指令执行过程中增加的引用计数，这样就能保证指令执行过程中，所有引用的实页不被替换出内存。

第四节　用户程序和虚拟存储作业

1. 去除同时只能有一个用户程序运行的限制，使得系统中可以同时提交多个用户进程。

2. 实现 Exit、Exec、Join、Create、Open、Read、Write 和 Close 等系统调用，读者可以根据需要自行增加。

3. Exec 系统调用需要能够在新创建的地址空间上传递用户程序的参数，使得可以提交带有参数的用户程序。

4. 设计和实现虚拟内存，对设计和实现方法作详细说明，并同第三节给出的实例作比较，给出实页命中的分析结果和实际结果。

5. 在实现了系统和虚拟内存的基础上，实现较大型的用户程序，如

cat 文件名	将文件的内容显示在屏幕上
cp 源文件 目标文件	将文件系统的的一个文件拷贝成另一个文件
sh	操作系统的外壳程序，负责启动其它应用程序 注意：需要实现一定的内部命令才更加实用

6. 目前的用户程序是单线程的，在实现了线程的 Fork 和 Yield 系统调用的基础上，实现允许用户程序生成多线程，并在这些线程之间不断切换交替运行。

第六章 Nachos的网络系统

随着计算机技术的发展，网络在计算机领域扮演越来越重要的角色。从 1983 年加州大学伯克利分校首次推出了 TCP/IP 的第一个 BSD UNIX 开始，网络模块就成了操作系统不可分割的一部分。计算机网络在数据共享、设备共享、通信、安全性以及容错性等方面给操作系统带来新的课题，而操作系统内含网络机制使得拥护使用和进一步开发网络应用更加容易，同样促进了网络的发展。从共享和网络透明的层次上分，现代操作系统又可以分成网络操作系统和分布式操作系统。

Nachos 包含有网络操作系统。在硬件层次上，Nachos 通过 socket 的机制使得同一台宿主机上的多个 Nachos 进程模拟了一个工作站网络，每个工作站都运行 Nachos。Nachos 本身以及运行在其上的用户程序可以通过向模拟网络收/发消息同模拟网络上的其它模拟计算机进行通信。

ISO 在计算机网络中规定了七层协议，为了让读者清楚地认识到如何使用网络和网络协议的工作过程，Nachos 在物理网络的基础上实现了一个简单的邮局协议（Post Office Protocal）。在邮局层（Post Office Layer）提供了一组“邮箱”来接收和到达的邮件，发往“邮箱”的消息中包含了目的地址和返回地址以便对方发回确认。

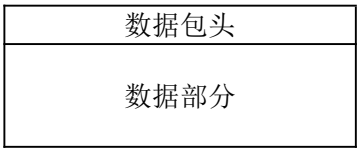
第一节 Nachos对物理网络的模拟

Nachos 的物理网络由 machine 目录下的 network.cc 和 network.h 文件进行模拟，该模块模拟了和一个 Nachos 虚拟机相连接的物理网络。该模拟网络有如下特点：

- 数据报发送是有序的
- 数据报的发送可能丢失，不是可靠的；但是数据报一旦到达目标节点，其内容一定是正确的，所以并不需要校验
- 网络之间的连接是全互连的

每个 Nachos 虚拟机有唯一的一个网络地址，该地址可以在 Nachos 启动时在命令行中给出。Nachos 实现时用一个 SOCKET 文件模拟了和自己相连接的网络部分。该文件名的格式是 SOCKET_网络地址。

一个网络数据包由数据包头和数据部分两部分组成。头部的内容是数据包发送者和接收者的地址信息和数据报文内容的长度。每个数据包的长度是固定的，为 64 个字节。




```
Class PacketHeader {
  public:
```

```
    NetworkAddress to;          // 数据报目标地址
    NetworkAddress from;        // 数据报源地址
    unsigned length;            // 数据报除了头之外的长度
};
```

Network 类定义和实现如下所示：

```
class Network {
  public:
    Network (NetworkAddress addr, double reliability,
            VoidFunctionPtr readAvail, VoidFunctionPtr writeDone, int callArg);
                                    // 网络初始化方法
                                    // readAvail:   网络读中断处理函数
                                    // writeDone:   网络写中断处理函数
    ~Network();                     // 析构方法
    void Send(PacketHeader hdr, char* data);    // 网络异步发送函数
    PacketHeader Receive(char* data);           // 网络接收数据函数
    void SendDone();                // 发送中断时调用
    void CheckPktAvail();           // 接收中断时调用
  private:
    NetworkAddress ident;           // 自身的网络地址
    double chanceToWork;            // 网络的可靠程度
    int sock;                       // 模拟网络的 socket 标识
    char sockName[32];              // 模拟网络的文件名
    VoidFunctionPtr writeHandler;   // 写中断处理函数
    VoidFunctionPtr readHandler;    // 读中断处理函数
    int handlerArg;                 // 中断处理函数参数
    bool sendBusy;                  // 数据包正在发送标志
    bool packetAvail;               // 有数据包到来标志
    PacketHeader inHdr;             // 到来数据包报头信息
    char inbox[MaxPacketSize];      // 到来数据包暂存区域
};
```

和机器模拟部分终端模拟实现相类似，两个内部函数 NetworkReadPoll 和 NetworkSendDone 作为网络读写的中断处理函数，这两个内部函数通过 CheckPktAvail 和 SendDone 两个内部方法调用真正的中断处理函数。这样，Nachos 可以以一定的时间间隔检查是否存在发给自己的数据包。当然在这些网络中断处理程序中，还包括了一些统计的工作。

系统的网络读写操作同样有严格的工作顺序，对网络读来说：

CheckPktAvail -> Receive -> CheckPktAvail -> Receive -> CheckPktAvail -> Receive ->...

系统通过定期的网络读中断来判断是否有发给自己的数据包，如果有则读出；如果没有，下一次读网络中断继续判断。读出的内容将一直保留到 Receive 将其读走。

对写网络来说：

Send -> SendDone -> Send -> SendDone -> Send -> SendDone -> Send -> SendDone -> ...

系统发出一个向网络发送数据包的指令 Send，模拟系统将直接向和目标机相连接的网络模拟文件发送数据包，但是对 Nachos 来说，整个写的过程并没有结束，只有当写网络中断来到后整个写过程才算结束。

语法： void Send (PacketHeader hdr, char * data)
参数： hdr: 发送数据包头
data: 发送数据包数据内容
功能： 向一个目标地址发送一个数据包。
实现： 1. 根据 hdr 的内容取出数据包的目标地址，得到与其相联系的目标 socket 文件
2. 设置 sendBusy 标志，在网络发送中断到来以前，不能再发送其它的数据包
3. 将网络发送中断放入等待处理中断队列中
4. 实际向目标文件写入数据包内容，表示数据报已经发送。
返回： 无。

在该模拟网络中，有可能发生这样的现象，就是发送方调用 Send 函数发送一个数据报，由于发送方此时只有等待到网络发送中断到达才可能确定此次发送成功。但是在调用 Send 函数到网络发送中断到达之前，与接收节点相连的网络 Socket 文件中已经有内容，接收节点完全有可能探测到数据报内容的存在并且将内容读走。所以目前的机制存在网络数据保送和接收同步问题的隐患。

需要说明为什么用 socket 机制来模拟网络，而不选用普通的文件？我们知道，通过 socket 和文件连接，这样的文件拥有这样的性质：已经读走的内容不可再现。于是对于这样的文件只需要不断地从文件头部读取内容即可。与此相类似，还可以通过使用有名管道文件达到同样的效果。图 6.1 是 Nachos 全互连网络的工作示意图：

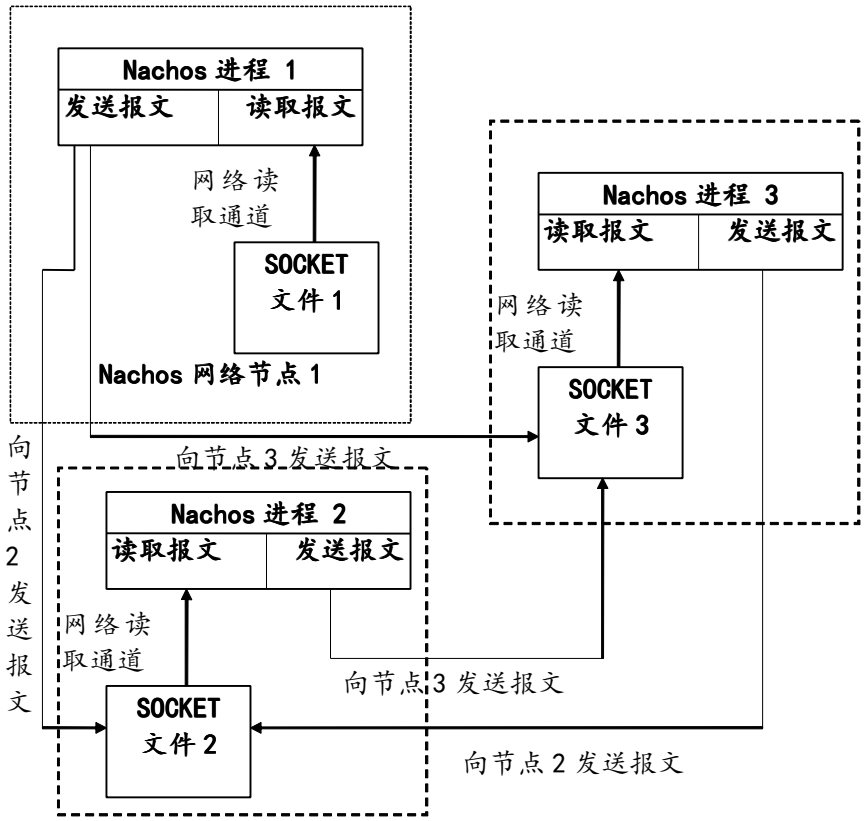


图 6.1 Nachos 全互连网络示意图

第二节 Nachos的邮局协议

Nachos 在上一节描述的物理网络的基础上建立了一个邮局协议。network 目录下的 post.cc 和 post.h 就是对该协议的实现。nettest.cc 文件对该协议的运作做了一个简单测试。

在每个 Nachos 模拟机上，有一个由邮箱组成的邮局，它负责监控与其相联的网络是否有可以接收的邮件。如果有，就接收下来分发给特定的邮箱；邮局对象同时还负责将本机的邮件发送给其它 Nachos 模拟机。读者可能会对邮箱的概念产生疑问，由于 Nachos 是一个多线程的系统，每个线程都可以独立地利用网络，这些线程在使用网络的时候如何不互相干扰呢？这里实际上借用了**网络端口**号的概念，每个线程需要进行网络通讯，就必须和一个网络端口建立联系，操作系统发现有网络数据包到达时，才知道发送给哪个线程。这里的网络端口就是邮箱。

上一节中提到，一个网络数据报有两个部分组成：数据报头和数据部分。报头部分包括数据报的收发地址，以及数据报的长度。实现 PostOffice 协议时，网络数据报又有两个部分组成：邮件的头部和邮件的内容。具体的邮件头部结构和邮件结构如下所示：

```
class MailHeader {
  public:
    MailBoxAddress to;              //  目标机的邮箱地址
    MailBoxAddress from;            //  需要回复的邮箱地址（即本机邮箱地址）
    unsigned length;                //  邮件的长度
};

class Mail {
  public:
    Mail (PacketHeader pktH, MailHeader mailH, char *msgData);
                                    //  生成方法
    PacketHeader pktHdr;                //  物理层加上的头部
    MailHeader mailHdr;             //  邮局层加上的头部
    char data[MaxMailSize];         //  实际数据部分
};
```

邮箱中存放的实际上是一个邮件的链表，邮局负责将接收的邮件放入邮箱，系统上特定的线程到邮局中特定的邮箱去取。对于邮箱的操作是互斥的。邮箱的结构和实现如下：

```
class MailBox {
  public:
    MailBox();                      //  初始化方法
    ~MailBox();                     //  析构方法
    void Put(PacketHeader pktHdr, MailHeader mailHdr, char *data);
```

```
                                    //  将一个邮件放入邮箱
    void Get(PacketHeader *pktHdr, MailHeader *mailHdr, char *data);
                                    //  从一个邮箱中取出邮件
  private:
    SynchList *messages;            //  同步邮件列表（互斥访问）
};
```

网络是一个异步设备，向网络发出请求后不需要等待其结束有可以返回，当网络处理的中断到来时，整个处理结束。在邮局对象生成方法中，生成了一个 Demon 线程，专门同步监测传给自己的邮件，分析邮件头部并将邮件放入特定的邮箱中。邮局的定义和实现如下所示：

```
class PostOffice {
  public:
    PostOffice(NetworkAddress addr, double reliability, int nBoxes);
                        //  邮局生成方法
    ~PostOffice();      //  析构方法
    void Send(PacketHeader pktHdr, MailHeader mailHdr, char *data);
                        //  发出一封邮件
    void Receive(int box, PacketHeader *pktHdr, MailHeader *mailHdr, char *data);
                        //  从邮箱中取出一封邮件
    void PostalDelivery();      //  监控网络上发给自己的邮件，放入特定的邮箱
    void PacketSent();  //  邮件发送时的中断处理函数
    void IncomingPacket();  //  邮件接收时的中断处理函数
  private:
    Network *network;       //  实现邮局的底层网络
    NetworkAddress netAddr;//  本机的网络地址
    MailBox *boxes;         //  本机的邮箱列表
    int numBoxes;           //  本机的邮箱数目
    Semaphore *messageAvailable;
                        //  邮件接收时的同步信号量
    Semaphore *messageSent;//  邮件发送时的同步信号量
    Lock *sendLock;         //  邮件发送锁（同时只能有一个线程发送邮件）
};
```

2.1 PostalDelivery方法

语法： void PostalDelivery ()

参数： 无

功能： 监测网络上是否有邮件到来，并将到来的邮件分发给各个邮箱。这就是邮局生成的 demon 线程执行的程序。

实现：
```
for (;;) {
    messageAvailable->P();              // 等待邮件的到来，
                                        // 网络有数据报到来时会执行 V 操作
    pktHdr = network->Receive(buffer);  // 将数据报接收到缓冲中
    mailHdr = *(MailHeader *)buffer;    // 取得邮件发送信息
    检查邮件的正确性，并将邮件放入相应的邮箱
}
```
返回： 无

2.2 Send方法

语法： void Send (PacketHeader pktHdr, MailHeader mailHdr, char *data)
参数： 无
功能： 将邮件发出去，接收邮件的地址由 pktHdr 中的接收地址和 mailHdr 中的接收邮箱决定
实现： 准备数据报

```
sendLock->Acquire();                    // 每次只能发送一个数据报
network->Send(pktHdr, buffer);          // 调用物理层的发送程序
messageSent->P();                       // 等待发送完毕的信号
                                        // 当发送完毕时调用 V操作
sendLock->Release();                    // 整个操作结束
```

返回： 无

第三节 网络部分作业

1. 目前的模拟网络是不可靠的，可以在命令行中设置网络可靠选项系数（-n #），其中#在0到1之间。在此基础上，需要实现一个可靠的网络协议，在此协议基础之上的网络没有丢失帧的现象。这里并不规定该网络协议的设计层次，读者可以实现在物理网络之上，也可以实现在邮局协议之上。

2. 在1的基础上，实现一些网络应用，比如可以实现多个用户可以通过网络进行交谈，可以设计一个网络四国大战游戏等。

3. 在安全可靠的网络基础上，实现带有cache机制的分布式文件系统、用户进程的迁移、分布式虚拟内存以及分布式共享内存等和分布式操作系统有关的部分。

4. 实现类似网络socket机制的系统调用，如Socket，Bind，Connect，Accept，Listen，Sendto，Recvfrom等，并在此基础上实现文件传输协议（FTP）等应用