目　录

一、概述

1.1 设计目标

1、基本框架：引导支持 FAT12 文件系统，内核采用 dos 下的 exe 格式，shell 独立成一个应用程序。

2、应用程序：支持 dos 下的 exe 和 com 文件的执行，如果有时间的话就多写几个小的示例程序。

3、用户界面：主要是字符界面，也会有一部分简单的图形界面（如果有时间做的话）。

4、内存管理：简单的内存管理。如果可能的话，添加实模式的虚拟内存支持，访问高端内存支持

5、文件系统：支持 FAT12 格式的文件系统

6、多任务：如果有时间的话尝试实模式的多任务机制

7、保护模式：如果有时间尝试进入保护模式

8、网络：如果有时间可以支持 ne2000 网卡，网络协议采用 WATTCP 实现的 TCP/IP 协议

9、驱动模型：不支持

1.2 设计目的

首先，设计 QUOS 的出发点不是为了创造商业价值。商业的软件需要的是更能完善强大的系统，这需要很多的人合作完成。我不具备这些条件。

其次，设计 QUOS 也不是为了在理论上都什么创新。在操作系统基础理论方面，已经比较完善了。对于一个专科生来说。研究理论也是不现实事情。

我设计它仅仅是为了实践，实践一些课堂上学到的知识。设计一个操作系统需要涉及到很多学科的知识。它不是学完一两门课程就可以做的，也不是仅仅学完课堂上的知识就可以完成的。设计它，不仅仅实践了所学，还获得了在设计一般应用程序时所无法获得的底层开发经验。通过设计它会加深我对计算机内部机制的理解，这也是从开发其他应用程序中获取不到的。

1.3 必备的基本知识

操作系统理论

基础理论是必须的。在操作系统理论中会讲到很多通用的策略，例如进程调度策略，内存管理策略等等。这些问题都是十分核心的问题。它们是基础，只有在理论基础上的实践才是可靠的。

汇编语言

汇编语言有两个优点使其难以被取代。首先是用汇编语言写出来的程序小巧。这个特点在编写引导程序的时候被用到。引导程序最多是 512 个字节，通常还好包含分区信息或文件系统信息。所以体积小巧是很重要的。汇编语言的另一个优点是底层操作很方便。比如调用 BIOS 的中断，端口读写等等，都需要用汇编语言来完成。这个方面可以在 C 语言里面使用嵌入式汇编来完成，但是，这依然是汇编。在我写的系统中用汇编还有一个理由那就是使用汇编文件写程序的开始部分可以使 VC 不再自动添加一些代码。

C 语言

C 语言有一些有点使其在写操作系统时比其他语言更方便一些。其实，任何一个可以编

译成本机代码的语言都可以用来编写操作系统。但是，C 语言比起其他语言更加的方便一些。首先，C 语言可以与汇编语言很容易结合在一起。前面我提到汇编语言无法替代的几个原因。C 语言可以跟汇编语言紧密的结合在一起使其比较容易的控制一些细节，比如端口读写等等。这个方面,C++也可以做到。但是，下面这个优点 C++有难以做到了。C 语言可以很容易的转化成汇编代码。而且转换后的代码可读性比 C++好的多。对于操作系统来讲，细节上的控制很重要。

计算机组成、微机原理和计算机体系结构

操作系统很多时候需要直接跟硬件打交道。熟悉计算机硬件的内部原理是必要的。比如 CPU 的指令集，工作模式等等。

数据结构

数据结构的知识不仅仅在写操作系统的时候被用到，在写其他软件的时候也会用到。但是，写操作系统的时候，数据结构的知识是必须的。没有数据结构的知识很多事情就难以完成。比如在我实现内存管理的时候就使用到了链表。

写操作系统不仅仅需要课堂上学过的知识。还需要很多课堂上没有学到的知识。这些知识一般都是比较具体的技术，与具体的实现有关。比如写一个运行于 X86CPU 上面的系统，就需要懂《intel 编程手册》上面的知识。而如果实现的系统需要在 arm 上面运行，《intel 编程手册》就没有什么用处。我实现的是运行在 X86 上面的系统，需要以下知识：

X86 硬件知识

介绍 X86 硬件知识最好的教科书莫过于 intel 的编程手册了。通常我们写操作系统的时候需要对 CPU 十分精通。这儿所说的精通不是指对 CPU 内部原理精通，而是指对 CPU 提供的软件接口精通。比如，我们需要将操作系统进入保护模式，就不需懂得怎样做才能让 CPU 从开机时候的实模式进入保护模式，进入保护模式与在实模式下有那些具体的不同。

BIOS

一般来说，提到 BIOS 的时候总是在说 BIOS 的设置。这儿不是。在这儿提到的是 BIOS 的中断。在 BIOS 里面有很多中断例程可以调用，这些例程的兼容性是不容置疑的。比如调显卡工作模式我们可以使用 BIOS 中断，也可以使用使用端口读写的方式。但是，不同的硬件对于端口的设置会有一些小的差别，这些小的差别往往造成很多兼容性问题。使用 BIOS 就不会有这些问题。

文件系统

在操作系统理论里面，有三个比较重要的部分，一个是内存管理，一个是进程管理，另一个就是文件系统。对于文件系统，仅仅知道理论是不够的。操作系统的一大作用就是资源管理。而数据管理是其中最重要的部分，对于数据的管理就是使用文件系统来完成的。在实现操作系统的时候，我们需要支持一个具体的操作系统。这就要求我们对于文件系统有很深刻的了解。我实现的系统中支持的是 FAT12 文件系统。在系统中会提供对于该操作系统的一些基本操作，比如文件的读写等等。

各种可执行文件的格式

要想在操作系统里面运行一个应用程序，首先需要对该应用程序的格式有所了解。现在的应用程序格式一般都不是纯粹的二进制格式，不是加载到内存中就可以直接运行的，它需要操作系统加载并解析才可以。常用的应用程序格式有 PE、COM 和 ELF 等等。我在操作系统中就采用了 COM 和 dos 下的 EXE 两种格式。其中，引导程序和内核加载程序采用的是 COM 格式，内核和 Shell 采用的是 EXE 格式。

1.4 实现的基本功能

对于现代操作系统来说至少应该包含以下几个部分：

1、 内存管理
2、 文件系统
3、 进程管理与调度
4、 设备管理
5、 中断和异常处理
6、 系统调用
7、 用户界面

这儿我不对每个概念进行详细的介绍了，在操作系统理论相关书籍里面都介绍的很详细。

由于我的这个操作系统是实模式下的操作系统，所以并不能称为是一个现代的操作系统。但是，大部分的功能还是实现了。我这个操作系统整体的结构是这样的：

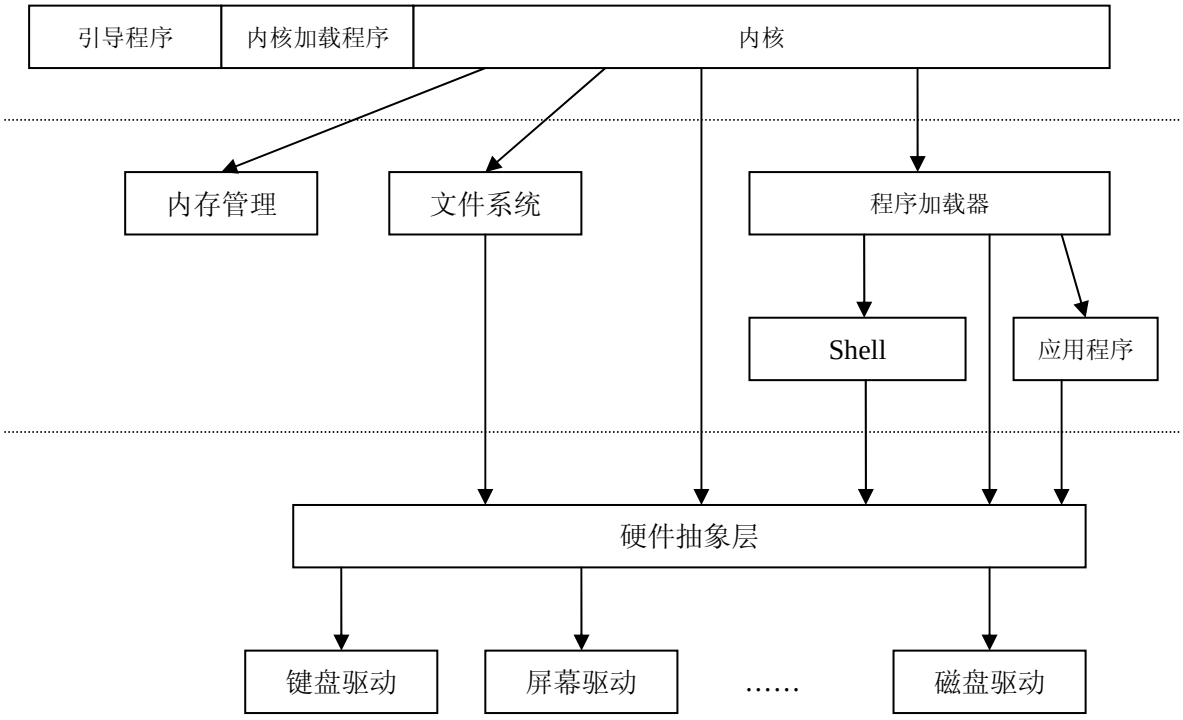

图 1.1

我在系统中实现功能并不是很多。我把主要的精力放在了系统的实现上，而不是应用上。具体来说有以下功能：

1、支持 FAT12 文件系统的引导程序。

这个引导程序可以支持 FAT12 文件系统。FAT12 文件系统是在 dos 下使用的文件系统。在 windows 下软盘就被格式化成这种格式。这使我们可以很容易的，在 windows 下面编辑我们的软盘镜像。因为在 windows 下面有很多程序可以编辑 FAT 文件系统类型的软盘镜像。我采用的就是 winimage。而且在虚拟机里面使用软盘镜像来运行这个操作系统也是十分方便的。

2、支持 FAT12 文件系统的加载程序

在内核运行之前，加载程序只能通过 BIOS 提供的读写硬盘的中断来读写硬盘。而 BIOS 提供的读写硬盘的功能是以扇区为单位的读写，它是不支持 FAT 文件系统的。我们必须在程序里面包含对文件系统的读操作。

3、EXE 格式内核

这儿的 EXE 格式不是 windows 下的 exe 文件格式。Windows 下的 exe 类型的应用程序采用的是 PE 格式。我们采用的是 dos 下的 EXE 格式。具体来说是 small 类型的 EXE 格式。关于为什么采用这种格式，我会在介绍内核的时候详细介绍。

采用 EXE 格式的内核的一个好处就是我们可以把我们的内核当作一个普通的 dos 下的应用程序来运行。这样做的好处就是调试起来很方便。我们甚至可以使用 debug 或 TDebug 来调试我们的系统。但是，我没有这样做，我使用的是 Bochs 这个工具来调试我的操作系统。我会在操作系统的调试里面详细的介绍。

4、简单的内存管理

由于我的操作系统是完全工作在实模式下，所以不存在操作系统理论里面讲到的分页管理、分段管理和段页式管理。在实模式下没有选择，只有分段管理。而且这个分段管理中段的大小是固定的。在实模式下，一个段是 64KB 大小，段的开始地址需要字节对其。

由于实模式下内存很少（实际可用内存不到 1MB），所以，不可以一个段一段的分配内存。那样的话很快就内存不足了。我的做法是用户要多少内存我就分配多少内存，我有一个链表记录内存使用情况，分配内存的时候就从链表中找，释放内存的时候也是对链表进行操作。这样做的好处就是可以最大限度的节省内存。

5、FAT12 文件系统

FAT12 文件系统是 DOS 使用的文件系统。这个文件系统比较简单，便于实现。我实现了对这个文件系统的简单读写支持。

6、DOS-EXE 格式程序加载器

现在，应用程序分为很多种格式，比如 Windows 下使用的 PE 格式、linux 下使用的 ELF、A.OUT 格式、DOS 下使用的 COM、EXE 格式。每种格式的实现都考虑到了其具体运行的操作系统环境，比如 PE 文件里面的存在资源节就是因为 Windows 是个以图形界面为主的操作系统，而 linux 之类的类 UNIX 操作系统里面采用的 ELF 就没有，那是因为它们主要以字符界面为主。

我在系统中采用的是 DOS-EXE 格式。这样做的原因就是这种格式是专门为了在实模式下运行而设计的。

7、简单的 shell

Shell 是 UNIX 下的概念。在 dos 下是 Command.com 这个程序，而在 Windows 是 CMD 这个程序。我沿用了 UNIX 下的叫法。其实，就是一个简单的字符类型的用户界面。

8、硬件抽象层

事实上我的硬件抽象层实现的并不是很完整，这儿的实现主要是通过重写 C 语言标准库函数来实现的。操作系统和用户程序通过调用我提供的标准库函数来访问硬件。

由于在实模式下，根本没有什么保护措施可以使用户程序不能直接访问硬件。所以，这个硬件抽象层并不是访问硬件的唯一途径。但是，通过硬件抽象层来访问硬件可以最大限度的保持程序的兼容性和健壮性。

9、键盘驱动

这儿仅仅是对 BIOS 的键盘中断做了封装，不能称为真正意义上的键盘驱动。因为不通过它也是完全可以实现键盘的访问的。

10、屏幕驱动

对 BIOS 的屏幕输入输出中断做了封装。

11、磁盘驱动

对 BIOS 的磁盘读写中断做了封装。

1.5 版权问题

对于软件来说总选择一个版权协议。对于商业软件来说一般都选择版权所有，来维护其商业利益。我们并非开发商业程序，也没有想过以此盈利。所以我们采用一个开源协议。

在开源协议里面也有很多中。常见的有 GUN v2、BSD 协议等等。我在源代码中采用的是 GUN v2 协议。我认为这种协议可以更好的帮助我体现我写这个系统的目地，并且可以更好的保护该系统。

对于文档，我保留一切版权。当然，只要不用于商业目地，我也是没有什么意见的。可以随意的拷贝和传播，但是不能修改，不能以此盈利。

当然，对于这个小操作系统来说版权其实没有多大关系。因为它实在太小，没有什么实际价值。但是，我依然对其做了版权说明。目地只有一个就是养成版权意识。

二、开发环境的建立和内核的调试

2.1 开发环境的建立

对于操作系统的开发来说，开发环境的建立是开发过程的第一步工作。一般来说，我们找不到一个专门的操作系统的开发工具。开发工具都是为了编写应用软件而设计的。所以，我需要专门的建立一个开发环境。

开发环境中主要一个编辑器、C 语言编译器、汇编编译器和连接器。可以选择的组合有很多种。编辑器就不用说了。任意一个可以编辑纯文本文件的编辑器就可以。因为源代码文件都是纯文本的。当然，如果可以支持代码着色就更好了。我使用的就是 editplus，它可以支持代码着色。而且还可以随意扩展，将其配置成一个 IDE。

C 语言编译器我采用的是 VC1.5。VC1.5 是我以前使用的编译器，对它比较熟悉了。虽然它有很多的不足，比如其嵌入式汇编不支持 32 位寄存器等等。但是，我这次还是选择了它。原因是熟悉。其实可以选择的 C 语言编译器是很多的。选择 GCC 的比较多一些。著名的 Linux 操作系统就是使用 Gcc 编译的。也有人选择 Bouland C。甚至有人使用 VC6.0。其实，选择的原则只有两个，一个是熟悉，一个是可以跟汇编编译器方便的协调工作。

汇编器我选择使用 MASM6.11。我选择这个编译器的原因是，它是最后一个可以生产 OMF 格式 OBJ 文件的 MASM 版本。因为 VC1.5 生成的就是这种格式的 OBJ 文件。要想把它们链接在一起就必须使它们的格式一样。对于汇编器，也有很多可以选择，比如 TASM,AS86 等等都可以。

链接器我使用的是 masm 里面代的 link。链接器一样有很多可以选择，比如 GUN 的 ld 就是很不错的链接器，功能很强大，还支持链接脚本。

2.2 内核的调试

在写程序的时候，经常出现的情况是程序的运行并不是我们所预知的情况。所以，往往需要一个调试的过程，来找到出问题的地方。很多时候问题并不是那么容易找到。有可能仅仅是因为一个指针应该是长指针却一不小心成了短指针，这么一点小小的错误就有可能造成系统崩溃。那么我们怎样找到错误的所在呢？在开发应用程序的时候我们有开发工具提供的调试器可以使用。我们可以单步执行，看一下程序的运行情况。但是，如何来调试一个操作

系统呢？它是运行在裸机上。是不是需要硬件调试工具呢？当然使用硬件调试工具是可以的。但是，现实情况是我们没有这么好的条件。是不是就没有办法调试了呢？当然不是。只是调试内核是一件比较困难的事情，但并不是说无法实现。

在 linux 里面有一个对于普通程序员来说比较神秘的内核调试器。虽然 linux 的创始者 linus 先生曾经撰文说不提倡使用调试器调试内核。但是，linux 内核依然提供了比较完善的调试器。它是基于 GDB 的调试器。功能很强大。但是，很不幸我却不能使用。因为我使用的是 VC1.5 开发的操作系统。GBD 的调试信息来自 GCC。

但是，我们依然可以通过使用 x86 模拟器来调试我们的系统。X86 模拟器，又称作虚拟机，它可以用来模拟一台真实的计算机。它和真实计算机一样有 CPU、硬盘、光驱等设备。只不过这些设备都是虚拟的。比如硬盘就是一个硬盘镜像，其实就是一个文件，只不过这个文件的组织格式和真机中硬盘的组织格式来说，也有扇区、磁道等概念，想读取其中的数据也是需要通过和真机中一样的方法（比如调用 BIOS）。我的操作系统就是放在一个软盘镜像上面。然后虚拟机像真机读取软驱一样的读取软盘镜像里面的内容，然后在虚拟机里面运行。

回到我们的话题里面，如何使用 X86 虚拟机来调试我们的操作系统呢？常见的虚拟机有很多，比如比较出名的 VMware 和 VPC。还有一些不是很有名的 qemu、Bochs 等等。其中 qemu 和 Bochs 是开源工具。它们提供了硬件级的调试功能。比如你可以在运行过程中暂停一下看一下内存中某个地方的数据等等。不过 qemu 本身提供的调试功能不是很全，它需要结合 GDB 来使用才能完成调试工作。Bochs 就不一样了。Bochs 是一个全模拟的虚拟机。它提供了比较全的调试功能。你甚至可以在里面单步执行或反汇编一些代码。这样一来就很方便了。我们可以添加在操作系统中添加断点，然后单步执行、反汇编我们的代码或查看内存中某个地方的值。

不过 Bochs 并不像普通的 Windows 应用程序那样有常规的图形界面。它提供的是字符界面，对于用惯了图形界面的我们来说一开始可能不是很习惯。但是，慢慢的熟悉了以后，你就会发现它的好处。Bochs 的用法比较复杂，我在这儿不再赘述了。在参考资料【14】里面有详细的介绍。这是我以前写的一篇关于如何使用 Bochs 的文章。

三、引导程序

3.1 简介

引导的功能很简单，就是完成 BIOS 没有完成的一些加载问题。由于 BIOS 不会支持文件系统，而我们的内核通常有被放在各种文件系统里面。所以，BIOS 是没有办法直接加载我们的内核的。也就是说，内核的加载其实需要我们自己在引导程序里面来完成。

3.2 相关原理

我们知道在操作系统里面，运行一个程序是由操作系统来加载的。那么，操作系统是如何运行起来的呢？很大一部分就是引导程序的功劳。

CPU 加电启动以后，会自动运行 BIOS 例程。BIOS 首先要做的就是检查硬件是否有问题。如果有问题会提出报警。比如内存没有插好的时候，主板就会不停的响。这个工作就是 BIOS 来做的。如果硬件没有问题的话，BIOS 就会根据用户的设置从磁盘中找操作系统了。实际上找的是操作系统的引导程序。它会扫描各个磁盘的引导扇区，比如软盘、光盘、硬盘等等。如果某个磁盘的引导扇区是以 0x55AA 结尾的。BIOS 就会把该磁盘的引导扇区加载到内存的 0X7C00 处，并跳转过去运行。

接下来的工作就是由我们自己写的引导程序来完成了。一般情况下，引导程序不会直接加载，操作系统内核。因为引导程序必须在引导扇区里面，而引导扇区只有 512 个字节。这 512 个字节还不仅仅是用来存放引导程序，对于硬盘来讲主引导扇区还要保存分区信息，分区的引导扇区还要保存该分区的文件系统信息。对于软盘来讲，也和硬盘分区的引导扇区一样需要保存其使用的文件系统信息。

引导程序又分为好几种。一种是多系统的引导程序。这种引导程序可以引导多个操作系统。比如 GRUB。它们其实不再是一个简单的引导扇区。它们可以算的上一个小的操作系统了。它们一般都支持多种文件系统，比如 FAT32/16/12、EXT2、NTFS、ISO9600 等等。另外一种是专有系统的引导程序。它们都是面向单一的一个操作系统的。比如 dos 的引导程序。

我的操作系统是保存在一个软盘镜像里面的。所以，引导程序是保存在软盘的引导扇区里面的。而且我的操作系统仅仅是为了实践，没有必要做的太复杂。所以，我的引导程序仅仅可以引导我自己的操作系统。

对于引导程序来说，其工作就是从硬盘上面读取一个程序到内存中运行。可以支持文件系统，也可以不知道文件系统。我在以前写的一个操作系统里面就没有支持文件系统，我把要加载的程序放在了一个固定的扇区里面。这样处理起来很简单。但是不是很灵活，因为我们没有办法在 windows 下使用软件来编辑其中的内容。我当时写了一个小程序 put，来完成对软盘镜像的读写工作。

在这个操作系统中，我实现了对文件系统的支持。这样做是我们可以很方便的编辑软盘镜像。因为我们的内核被保存在 FAT 系统的软盘镜像里面，也就可以很容易的使用类似与 winimage 的软件镜像编辑工具来编辑它。

3.3 具体实现

在讲解我如何实现的引导程序前，我觉得有必要看一下 linux 是怎么做的。下面这个图是我从《linux 内核完全注释》里面剪切出来的。

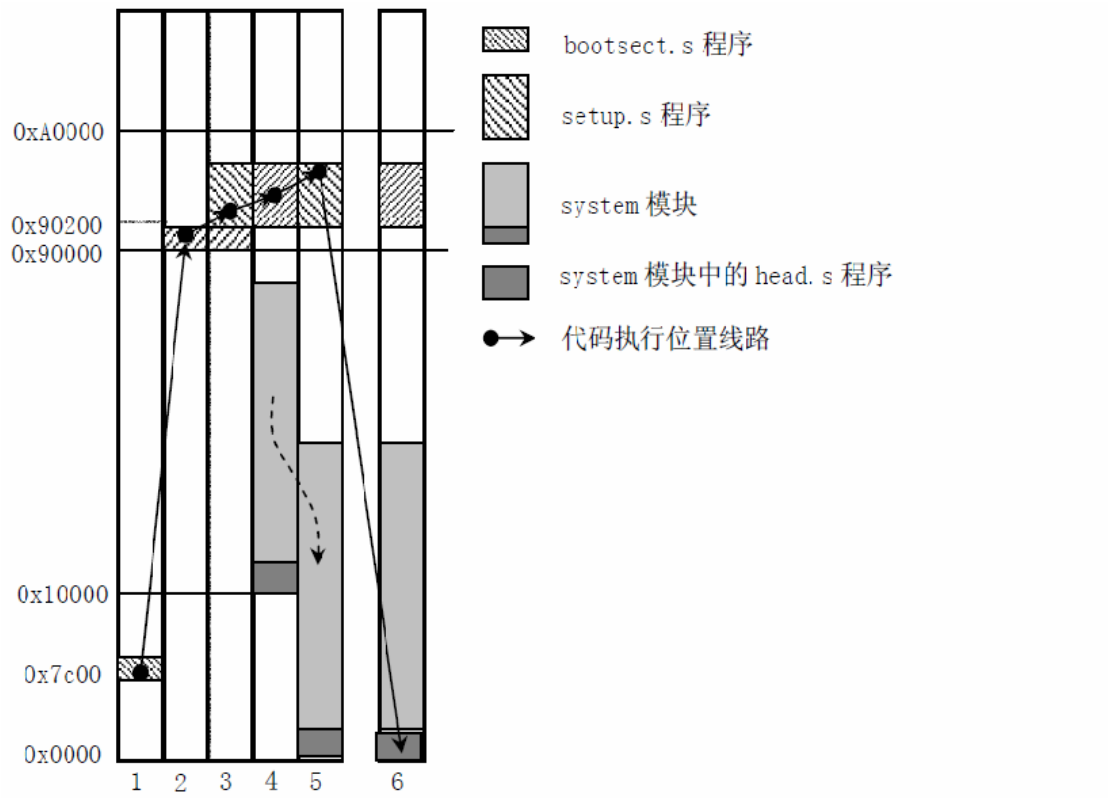

图 3.1 启动引导时内核在内存中的位置和移动后的位置情况

图 3.1

我们可以看到，处于0x7c00处的就是就是linux的引导程序bootsect.s。它首先把自己移到0x90000这个地方然后加载内核加载程序setup.s，最后跳转到内核加载程序setup.s来运行。在它的系统中并没有实现文件系统的解析，而是直接“*利用BIOS中断INT 0x13 将setup模块从磁盘第2个扇区开始读到0x90200 开始处，共读4个扇区*”【8】。它这样做原因是很多的，详细的介绍可以找相关的资料。

那么我是怎么做的呢？请看下图：

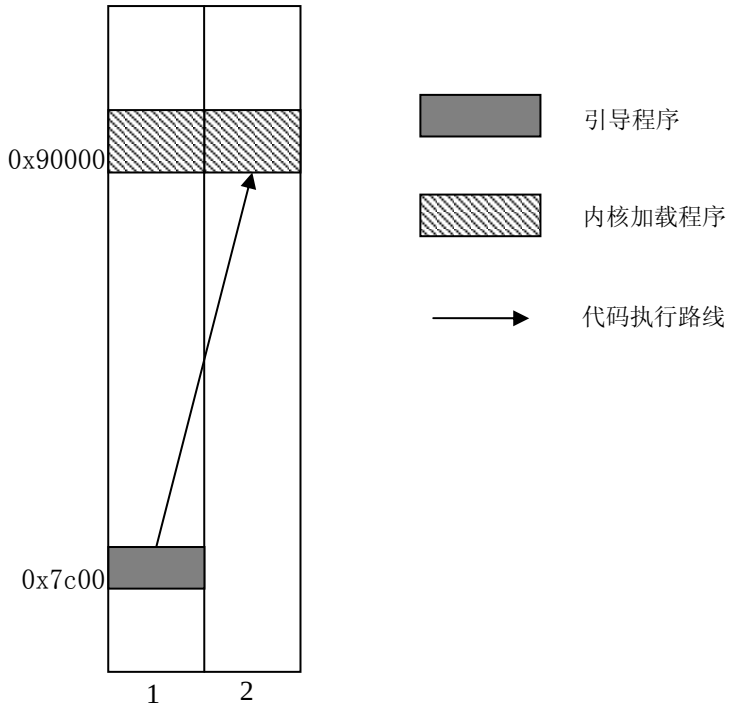

图3.2

在计算机加电以后 BIOS 会自动把我的引导程序从软盘的引导扇区里面读到内存的 0x7c00 处，然后跳转过来运行。

引导程序首先是初始化运行环境，比如设置堆栈、显示模式等等。接下来就是显示一个字符串“Booting ”。

接下来，引导程序会调用 BIOS 的中断 int13h 读取软盘的根目录所在的扇区，并分析读取到的数据，以从根目录里面找到内核加载程序 setup.bin。如果找不到就会显示提示信息“No SETUP.”然后死机。如果找到就会将内核加载程序加载到 0x90000 处，并显示提示信息“Ready.”。

最后，引导程序跳转到内核加载程序运行。到这儿引导程序的工作就完成了。

3.4 代码细节

首先，来看看引导程序详细的运行流程：

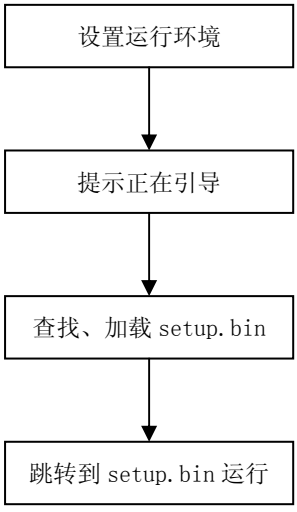

图 3.3

因为 booter 会被 BIOS 加载到 0x7c00 处。堆栈又是向下生长的，所以我将堆栈放到 0x7c00 处，这样就不会存在因为堆栈过小而冲掉我们自己代码的问题。

即便如此，堆栈和代码也并不是紧靠着的。引导程序虽然被 BIOS 加载到了 0x7C00 处。但是，引导程序的代码并不是从 0x7C00 开始的。在 0x7C00 处是一个跳转语句。它会跳过一段内存，然后进入引导程序的代码运行。那么，跳过的这部分数据是什么呢？

它们是定义与 boot.asm 开始的一些与 FAT12 文件系统有关的常量。我在前面提到过在软盘的引导程序里面会包含该软盘文件系统的一些信息，这儿就是那些信息。用 linux 的描述方法就是它是 FAT12 文件系统超级块【4】。我的操作系统支持的是 FAT12 文件系统，所以这儿描述的也是 FAT12 文件系统。

在引导程序的开始我们会重新设置一下显示模式。这样做的目的有两个。一个是清除 BIOS 的输出信息。重新设置显示模式是清屏的一种简单方法。这样做的目地就是保持代码短小。理由重新设置显示模式的方法清屏，代码才几个字节，而通过其他方法需要几十甚至上百字节。另一个目地就是统一显示方式，有可能不同机器的 BIOS 在引导时屏幕的显示方式是不一样的，而这种不同有可能会对我们的屏幕输出带来一些影响。虽然这种情况很少发生。设置然显示模式以后就是输出提示信息“Booting ”了

接下来还会复位一下软驱。这个操作并不是必须的。由于我使用的是软盘镜像，所以根本看不出来有什么效果。如果使用的是真正的软盘，通过真正的软驱读取软盘数据就会看到一点效果了。这样做的好处只有一个，那就是可以让软驱读取 setup.bin 的时候更快一点【6】。

然后就是在根目录里面查找 setup.bin。这个查找过程有点复杂。涉及到 FAT12 的原理。详细的原理会在《文件系统》那一章中提到。这儿我仅仅描述我是怎么做的。

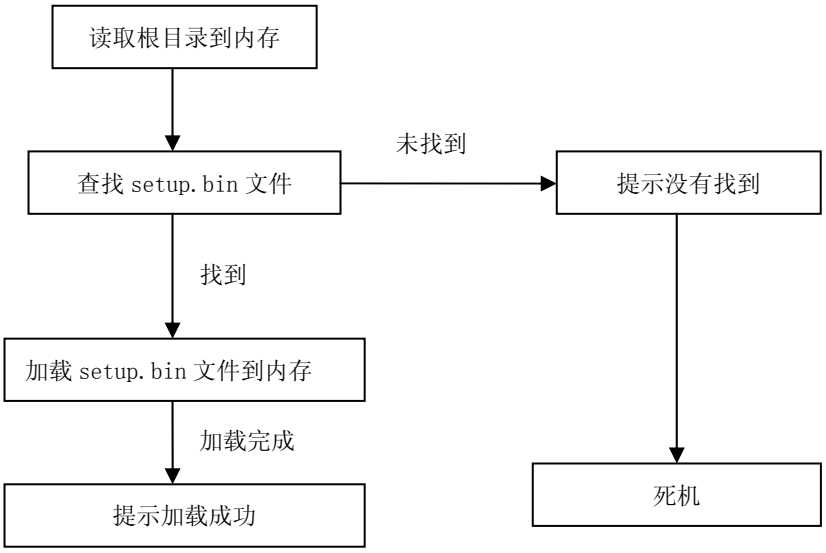

图 3.4

第一步：调用 BIOS 的中断 int13h 读取软盘的根目录所在的扇区。根目录所在扇区是固定的。有 FAT12 文件系统规定。

第二步：依次对比根目录里面的每个文件是不是我们要找的 setup.bin。如果对比完根

目录里面的所有文件还没有找到就是说明 setup.bin 不存在，我们就输出提示信息“No SETUP.”然后死机。如果找到，就读取该文件的 FAT 表，然后通过 FAT 表读取相应的扇区到内存中。

这样就把 setup.bin 加载到内存里面来了。引导程序的工作也就完成了。接下来要做的就是输出提示信息“Ready.”。然后跳转到内核引导程序 setup.bin 里面运行了。

那么内核加载程序运行完会不会返回呢？当然不会。首先，我采用的是 jmp 指令，而不是 call。内核加载程序也就没有跳转回来的可能。还有就是，内核加载程序的工作是加载内核，并运行。而内核不会返回到内核加载程序里面来。因为内核最终的执行结果是关机或重启。这有点像 Windows 下图形界面程序。在图形界面程序里面有一个消息循环。它几乎是一个死循环。只有接到退出消息时 Windows 才会结束掉它。其最后的执行结果就是所在进程被销毁。我的操作系统也是这样的，运行的最后结果就是关机或重启。

因此，引导程序在执行完以后，也就没有什么意义了。完全可以覆盖掉。

四、内核加载程序

4.1 简介

内核加载程序的任务就是把内核加载到内存中去并运行。

并不是每个操作系统都会有一个内核加载程序存在。像我的系统就完全可以不要内核加载程序。但是，我还是写了写个程序。具体原因我会在本章的具体实现一节中介绍。

在成熟的操作系统里面都会有一个内核加载程序存在。这是必需的。为什么说是必需的呢？

首先，内核完成的功能比较复杂，这就决定了它会比较大。像现在的 linux，在压缩了以后还大于 1MB 呢。也就说在实模式下是不能完成内核的加载任务的。因为在实模式下，总共可以访问的内存才 1MB，而且还有一些被显卡和 BIOS 用掉了。对于引导程序来说无法完成这么复杂的任务。引导程序不能超过 512 个字节。因此，没有办法在引导程序里面完成这些工作。

其次，内核不再是加载以后跳转过去就可以直接运行的了。像 linux，内核就被压缩了。而且内核也会有一些格式。比如可能是 PE 格式或 ELF 格式等等。这些格式都需要解析。这些工作也不是在 512 个字节大小的引导程序里面可以完成的。

在Windows的系统分区里面有一个250多KB的系统文件叫做ntldr。它其实就是Windows的内核加载程序。

有时候内核加载程序并不仅仅完成内核加载运行的任务。在 linux 中内核加载程序就会做很多其他的工作。比如搜集硬件信息，如内存信息、磁盘信息等等。

4.2 相关原理

首先看看我的内核加载程序的运行流程是什么样子的：

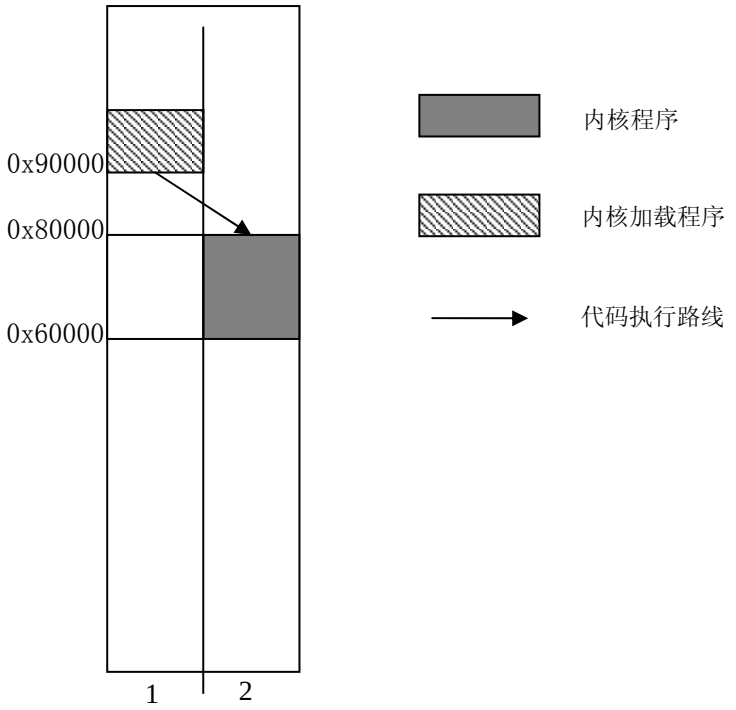

图 4.1

从图中我们可以看出内核加载程序的执行流程其实引导程序差不多。确实如此。内核加载程序之所以会存在就是因为引导程序不能做的太大，无法完成一些复杂的工作。这些复杂的工作就是由内核加载程序来完成的。那么，内核加载程序到底完成了哪些复杂的工作呢？一般有以下一个工作需要在内核加载程序里面完成：

1、加载内核程序到内存

有时候内核比较大，所以在实模式下是无法完成加载任务的。因为实模式下内存只能有1MB。如果内存大一点就必须先进入保护模式然后再加载内核了。但是，进入了保护模式有一点比较麻烦，那就是不能调用 BIOS 提供的中断。也就是说，原来简单的调用 BIOS 中断就可以完成的读写磁盘的工作在保护模式下需要费些心思才能实现。

2、搜集硬件信息

其中包括内存大小、磁盘信息、显卡信息等等。这些信息是通过调用 BIOS 中断获得的。对于现代操作系统来说，进入保护模式是理所当然的事情。我前面提到了，在保护模式下无法调用 BIOS 的中断。而这些信息又必须通过 BIOS 获得。因为 BIOS 在启动时会检测硬件，这些信息的可靠的，没有再重复 BIOS 的工作。

3、实现多系统引导

有时候我们会在硬盘上同时安装多个操作系统。比如在 C 盘安装了 Windows 2000 以后又在 D 盘安装了 Windows XP，甚至同时安装了 linux 和 Windows。这个时候就需要在引导的时候检测一下硬盘上到底有几个系统，然后让用户选择进入哪个系统。

4、解析内核的文件格式

内核一般不会是纯粹的二进制格式，而是比较复杂的格式。比如 linux 的内核就是经过压缩的。有可能内核采用的是 PE 或 ELF 等文件格式。这些解析工作就需要内核加载程序来完成。

对于我的操作系统来讲，没有这么复杂。首先，我的内核虽然是 exe 格式，但是在编译完以后就做了一些处理。这些处理使其可以像纯粹的二进制文件一样的直接加载到内存运行。我们也不需要在内核加载程序里面搜集硬件信息。因为我们的内核就是工作的实模式下，

想获取硬件信息随时可以调用 BIOS 中断获得。我也没有打算实现多系统引导。所以，我的内核加载程序仅仅完成第一项工作，那就是加载内核程序到内存。

4.3 具有实现

我在本章的开头就说过，对于我的这个操作系统来讲内核加载程序实际上可以不需要的。原因很简单，内核加载程序完成的这些功能都可以在引导程序里面完成。也就是说，我可以直接使用引导程序把内核加载到内存中运行。但是，我还是写了内核加载程序。理由有两个。

一、在引导程序中由于大小限制不能输出很多的提示信息。一旦出现什么问题，只能输出很简短的信息。我需要提供比较标准的规范的提示信息的输出。这个任务在引导程序中无法完成。

二、我有一个计划，就是以后会使内核工作在保护模式下。这样一来内核加载程序也就不可或缺了。到时候，我只需要修改一下内核加载程序就可以了。而没有必要连引导程序也修改。

从图 4.1 可以看出，内核加载程序的运行流程和引导程序几乎是一样的。其实现也是差不多的。实现的功能也和引导程序差不多。其运行流程图也和引导程序几乎一样，可以参考图 3.3。

首先，设置环境，比如堆栈等等。输出提示信息“Loading kernel　　”。

接着，从软盘的根目录中寻找内核，并把内核从磁盘读取到内存中。具体的寻找过程可以看 3.4 节里面的介绍。

然后，给出一些提示信息。告诉用户内核加载成功或失败。

最后，跳转到内核执行。

4.4 代码细节

与引导程序不同的是我是使用 MASM 写的内核加载程序。而引导程序我使用的是 NASM。其中的原因是，MASM 可以更好的跟 VC1.5 配合起来。我在扩展内核加载程序的时候肯定会使用到 C 语言代码。这就决定了我最好是使用 MASM，这样不会在以后扩展内核加载程序的时候再将 NASM 的代码转换成 MASM 的代码了。

当引导程序把内核加载程序从软盘加载到内存并执行以后，内核加载程序首先要做的是初始化运行环境。和引导程序一样，它也会把自己的堆栈放到程序的开始处。不同的是这一次，堆栈是紧靠着代码的了。

接下来的工作就是加载内核了。流程图和图 3.2 几乎一样，这儿就不再给出了。

第一步，显示一些提示信息。告诉用户正在加载内核。

第二步，从软盘的根目录里面寻找内核程序。这个寻找过程跟引导程序找内核加载程序是一样的。这儿就不重复介绍了。

第三步，显示提示信息。告诉用户内核加载的结果。

完成内核加载以后，内核加载程序的工作也就结束了。下面要做的就是跳转到内核运行了。

这儿我采用的跳转方法和引导程序中是不一样的。在引导程序中采用的是长跳转，而在这儿的方法是构造一个函数返回环境然后调用 retf 指令。这样做是迫不得已的。因为 MASM 不允许 TINY 格式的程序有长跳转。不是可以长跳转，而是 MASM 不允许。有时候微软的编译器就是这样，总是给你制造麻烦。在使用 C 语言的时候我遇到了更多的麻烦。在相关的章节会提到。在这儿，我们就模拟函数调用的返回过程来跳转到内核去。

跟引导程序中一样。会不会再返回到内核加载程序继续运行呢？答案和引导程序里面一

样，不会。首先，我们跳转到内核的方法是函数返回，内核根本就不是内核加载程序调用的，相反倒是很想内核调用的内核加载程序。其次，内核会修改堆栈位置。虽然通过函数返回的方式内核是可以知道从哪儿返回的。但是，内核不会使用这些信息。在内核里面首先要做的就是重新设置运行环境，其中就包括堆栈。还有就是，内核运行的终点只有关机和重启。所以，不会在返回到内核加载程序来执行。

因此，在内核加载程序执行完以后，它也就没有什么意义了。完全可以覆盖掉了。

五、build 工具

5.1 简介

前面提到过，内核可能是一些特殊的格式，而不一定是纯粹的二进制格式，比如 PE 格式、ELF 格式等等。这就需要在内核运行之前对其进行解析。那么到底是什么时候解析呢？是在编译完成以后就解析还是在加载到内存以后再解析呢？这个没有固定的做法。怎么样都可以。一般来说，在编译完成以后就对其进行解析是个不错的做法。因为这样做可以最大限度的降低内核加载程序的复杂度。要知道内核加载程序是在内核运行前的裸机里面运行的。也就是说它的运行环境有诸多的限制。所以，在编译完以后立即对内核解析，然后在内核加载程序里面加载的其实是一个内核在内存是运行的镜像，这样就可以直接将其加载到内存中然后运行了。

实际上，在 linux 里面也是这样做的。它从最早期的版本开始就有一个 build 工具用来生成内核镜像。其中就有解析 A.OUT 格式的功能。因为早期的 linux 是在 minix 下开发的，使用的是 A.OUT 格式。

我的这个 build 工具的用途和它差不多，也是用来对内核的格式进行一下解析以使内核加载程序在加载内核的时候不需要再对其进行解析。

5.2 相关原理

在前面介绍到，我的内核使用的是 dos 下的 exe 格式。而且我是使用 build 工具对其进行解析的。那么 dos 写的 exe 格式到底是个什么样子的呢？EXE 文件主要是由三部分部分组成：EXE 头、EXE 程序体，请看下图：

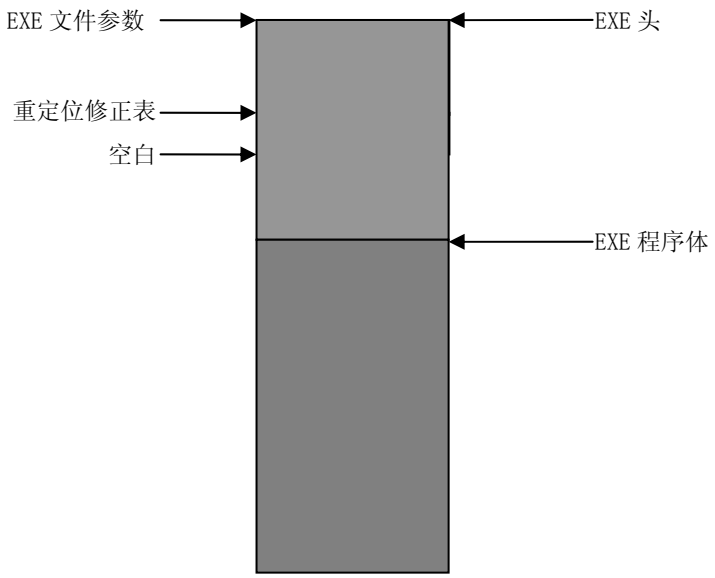

图 5.1

在EXE头的开始处保存了程序的一些参数,这些参数就是在解析exe文件格式的时候用到的。其定义如下：

```
struct exeHead
{
    WORD wMagic;            // 幻数
    WORD wSizeInLastBlock;  // 最后一个块使用的字节数
    WORD wBlockCount;       // 512 字节块的个数
    WORD wReallocCount; // 重定位项个数
    WORD wHeadSize;     // 文件头部大小（以 16 字节为单位）
    WORD wMinAlloc;     // 需额外分配的最小内存量
    WORD wMaxAlloc;     // 需额外分配的最大内存量
    WORD wInitSS;           // 初始 SS
    WORD wInitSP;           // 初始 SP
    WORD wChecksum;     // 文件校验和(保留不用，为 0)
    WORD wInitIP;           // 初始 IP
    WORD wInitCS;           // 初始 CS
    WORD wFirstRealloc; // 重定位修正表位置(第一个重定位项相对于文件的偏移值)
    WORD wNoverlay;     // 重叠的个数，程序为 0
};
```

其中，各个参数的具体含义是：

1、wMagic：幻数，必需为 0x5A4D，用于识别该文件是否为 dos 的 exe 格式。

2、wSizeInLastBlock：该 exe 程序最后一个块使用的字节数。其中一个块是 512 个字节，也就是一个扇区大小。

3、wBlockCount：该 exe 程序总共占用的块数，每个块 512 个字节，也就占用多少个扇区。结合 sizeInLastBlock 就可以算出该 EXE 程序的大小为：(wBlockCount - 1) * 0x200 + wSizeInLastBlock

4、wReallocCount：重定位修正表中项的个数

5、wHeadSize：EXE 头的大小，以 16 个字节为单位。通过它就可以计算出 EXE 程序体先对程序开始的偏移量为 wHeadSize * 0x10。

6、wMinAlloc：需额外分配的最小内存量

7、wMaxAlloc：需额外分配的最大内存量，编译器一般都会将其写成 0xffff。

8、wInitSS：初始 SS 的值

9、wInitSP：初始 SP 的值

10、wChecksum：保留不用（但是也有些资料说的校验和，但是它的值在实际应用中始终为 0）

11、wInitIP：初始 IP 的值

12、wInitCS：初始 CS 的值

13、wFirstRealloc：重定位修正表位置(第一个重定位项相对于 EXE 头开始处的偏移值)

14、wNoverlay：重叠的个数，程序为 0。

紧接着 EXE 文件参数的就是重定位修正表。重定位修正表中每项都是一个双字，记录的就是需要重定位处理的数据相对于 EXE 程序体的偏移值。用 C 语言表示重定位过程就是：

```
for (i=0; i<pHead->wReallocCount; i++)
{
    p = (WORD *)((DWORD)pBody + pReallocTable[2*i] + pReallocTable[2*i+1] * 0x10);
    *(WORD *)p += wBaseSegment;
}
```

其中，wBaseSegment 就是该 EXE 程序将要被加载到内存中的段地址。

5.3 具体实现

EXE 程序加载执行的具体过程为：

1、读入 EXE 文件参数，验证幻数是否有效。

2、找一块大小合适的内存区域。wMinAlloc 和 wMaxAlloc 域说明了在被加载程序末尾后需额外分配的内存块的最大和最小尺寸(链接器总是缺省的将最小尺寸设置为程序中类似 BSS 的未初始化数据的大小，将最大尺寸设置为 0xFFFF）。

3、创建一个程序段前缀（Program Segment Prefix），即位于程序开头的控制区域。

4、在 PSP 之后读入程序的代码，也就是 EXE 程序体。

5、从 wFirstRealloc 处开始读取 wReallocCount 个修正地址项。对每一个修正地址，将其中的基地址与程序代码加载的基地址相加，然后将这个重定位后的修正地址作为指针，将程序代码的实际基地址与这个指针指向的程序代码中的地址相加。用 C 语言代码表示就是：

```
for (i=0; i<pHead->wReallocCount; i++)
{
    p = (WORD *)((DWORD)pBody + pReallocTable[2*i] + pReallocTable[2*i+1] * 0x10);
    *(WORD *)p += wBaseSegment;
}
```

其中，wBaseSegment 就是程序代码加载的基地址。

6、将栈指针设置为重定位后的 sp，然后跳转到重定位后的 ip 处开始执行程序。

解析 EXE 文件其实就是按照 EXE 文件参数的说明依次的修改重定位项。因为我的内核不会申请内存，所以这第 2 步不用考虑。我的内核也不使用 PSP，第 3 步和第 4 步也不用考虑。而 sp 指针我们会在内核里面设置，并且我们仅仅是解析而不执行它，所以第 6 步也不用考虑。这样一来我要做就只有第 1 步和第 5 步。当然第一步我们读入的不仅仅是 EXE 文件参数，而

是整个文件。处理完重定位后，在把 EXE 程序体保存到一个文件里面就可以了。这个就是所需要的目标数据，也是可以直接加载到内存执行的二进制程序。程序的运行流程是：

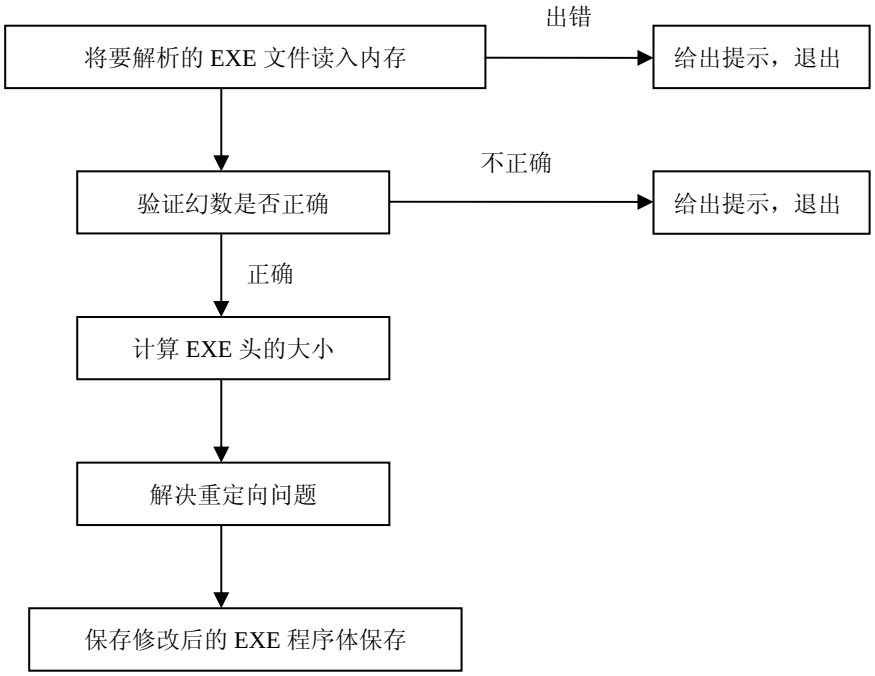

图 5.2

由于程序代码比较简单，而且其中有足够的注释，就不再对其中的细节进行详细的介绍了。

六、内核

6.1 简介

对于操作系统来讲最主要部该就是它的内核。在内核里面包含了内存管理、文件系统、进程管理和调度等等操作系统中最核心的东西。这也是每个操作系统的差别最明显的地方。每个操作系统的内核肯定是相差很大的。拿 inux 和 freeBSD 来说，虽然它们同属于类 UNIX 操作系统，但是它们之间有着很大的差别。一个内核的好坏直接决定了一个操作系统的好坏。这也是为什么 Windows 9X 系列操作系统与 Windows NT 系列操作系统在稳定性上相差那么多的原因——Windows NT 有一个稳定的内核。

对于现代的操作系统来讲，内核有两种，一种是微内核，一种是宏内核。关于微内核和宏内核相关概念的介绍可以参考相关的理论书籍，我在这儿不具体的讲了。

6.2 具体实现

由于我实现的是实模式下的操作系统，与现代的操作系统有着很大的区别，所以我就仅仅介绍以下我自己的内核是如何实现的、包含哪些功能。用图表表示的话，我的内核是这样的：

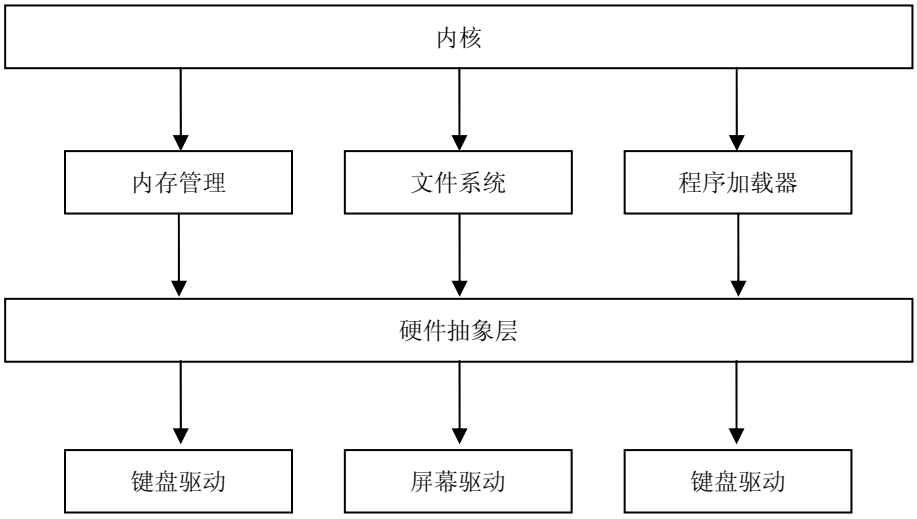

图 6.1

这儿仅仅是列出内核的各个部分，不对其进行介绍。在后面的章节会一一的对其进行介绍。下面我们来看一下内核的引导过程，如图：

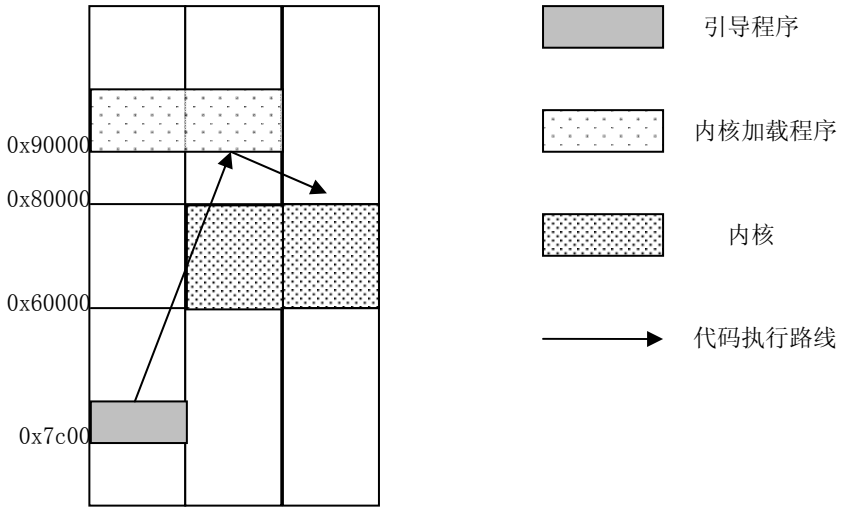

图 6.2

对于引导过程的详细介绍前面已经有了，请参考引导程序和内核加载程序这两章的内容。

七、硬件抽象层

7.1 简介

对于操作系统来说，它会经常的访问硬件，比如显示需要访问硬件。而且一个操作系统有可能运行与多种类型的硬件上面。像 linux 就可以运行与很多平台上面，比如 X86、arm、powerPC 等等。这些硬件可能千差万别。那么，是不是对每种类型的硬件都需要重新开发一份呢？不是这样的。对于操作系统来讲，真正与硬件紧密相连的代码并不是很多，绝大多数代码是与硬件无关的，而且在各个平台上面都是通用的。这样以来我就只需要把与硬件关系比较紧密的代码统一的放到一个地方，然后每个类型的硬件都有一份。并且对外提供统一的界面。这个就是硬件抽象层。

硬件抽象层的作用就是让内核的其他部分尽量的远离硬件。硬件抽象层为内核的其他部分提供了访问硬件的统一的方法。如果系统需要迁移到新的硬件平台时仅仅需要在硬件抽象层添加代码就可以了。内核的其他部分不需要或者很少需要改动。

在我的系统中，也存在一个薄薄的硬件抽象层。我设计这个硬件抽象层的目地有点特殊，因为我并没有将这个系统迁移到其他硬件平台上去的需求。但是，硬件抽象层还会给我带来另一个好处，那就是统一的硬件访问。这是我想获得的效果。那么，我为什么要统一的访问硬件呢？而不是什么时候需要访问硬件的时候就直接访问，还要通过硬件抽象层，这么麻烦？其中，一个原因就是我想实现数据流这个功能。

7.2 相关理论

在操作系统理论里面有一个流的概念。我这儿要讲的是数据流。数据流在 C++语言中就有体现，比如在 C++里面有输入输出流。我们只需要把数据放到这个流上就可以了，而不用管数据如何转换、输出格式等等。还有一个地方可以体现，那就是重定向和管道。这两个概念都是 UNIX 下的。重定向就是把程序数据流（包括输入流和输出流）重定向到其他的位置，比如一个文件里面。管道的概念也是这样，不过它是个动态的过程。它的做法是将一个程序的输出当成另一个程序的输入。这样几个程序排成一排就如同一个管道一样，数据流在这个管道里面流动。

实现数据流有很多好处。首先，就是可以实现重定向和管道。这两个功能是很好用的。UNIX 下有很多小工具，本身的功能都不是很多。但是，将他们使用管道组合起来就可以实现很强大的功能。而且，由于每个部分功能很单一，一般不会有什么问题，组合以后的稳定性也是相当好的。

更重要的是实现这两个功能并不需要多进程的支持。需要的是对数据的统一处理。而硬件抽象层恰好给我提供了这个功能。

7.3 具体实现

这个硬件抽象层的名头很大。在实际实现的时候并不是多么麻烦。我们需要的仅仅是提供一系列的方法来提供访问硬件能力。并且在内核的其他地方访问硬件的时候都通过这些方法即可。而这些方法其实就是 C 语言的标准库函数。在 C 语言的标准库函数里面，其实暗含了访问硬件。比如 putchar，就需要访问屏幕，getchar 就需要访问键盘。

我们只需要把这些标准库函数实现一遍，其实就相当于实现了一个薄薄的硬件抽象层。比如端口访问等等。都是通过这个函数来进行。需要注意的是，我并没有考虑往不同的硬件平台迁移的问题，所以这种实现是可以的。当考虑适应多个硬件平台的时候，所要做的工作

就复杂多了。不是仅仅几个函数可以解决的。

事实上，在写操作系统的时候绝大多数标准库函数都是不能使用的。前面已经提到过原因了，那就是在标准库函数里面隐含着访问硬件，而不同的操作系统访问硬件的方法也是不一样的。所以，在写操作系统的时候不能调用这些很少，一旦调用就使我们的内核无法在裸机上运行。所以，绝大多数的标准库函数都是需要我们重新来写的。最简单 getchar 也需要我们自己来写。像 memset 之类的也需要我们自己写。其实，需要我们重写的常用标准库函数也不是很多。因为很多库函数我们根本就不会用，比如数学函数等等。还有一部分函数需要我们在其他模块完成以后才能完成，比如文件操作函数就需要在完成文件系统以后才可以重新写。去掉这些，剩下的就是我们通常使用的函数了，不是很多，才 20 多个。它们分别是：

字符处理函数：

```
getch()     putchar()       getchar()
```

字符串处理函数：

```
puts()      gets()      strcpy()
strcmp()    strlen()    strcat()
```

字符串格式化输入输出函数：

```
printf()        scanf()
```

转换函数：

```
itoa()      ltoa()      atoi()      atol()
```

内存函数：

```
memcpy()        memcmp()        memset()
```

端口读写函数：

```
outb()      outw()      inb()       inw()
```

中断调用函数：

```
int86()
```

按照以前的经验，我又添加了几个函数：

```
void far * lineToFar(DWORD dwIn);//线性地址转化成长指针地址
DWORD farToLine(void far * pIn);//长指针地址转化成线性地址
BOOL isASCII(char CharIn);// 判断是否是可显示的 ASCII 码
```

上面的函数在实现的时候参考了很多我以前的代码，有些甚至是直接拷贝过来的。不过这些函数基本上都是我自己写的。其中 printf 参考的是 linux0.1.0 中的相关代码。但是，由于编译器的问题，被我改造的地方很多，几乎是重写了一遍。在写的时候，主要是以 VC6.0 中的实现为标准的来实现的。调试的时候也是直接根 VC6.0 做对比，知道两者在各种情况下的输出全部一样为止。

其中，唯一的不同点在 printf 之类的可变参数里面必须显式的使用长指针。比如有以下代码：

```
char string[20];
strcpy(string, "hello word");
printf("%s", string);
```

就必须写成：

```
char string[20];
strcpy(string, "hello word");
printf("%s", (char far *)string);
```

没有办法，这是由于编译器的原因造成的。我已经把编译参数改成默认为长指针了，可

以对于可变参数，它依然认为是短指针。在 VC6.0 里面默认就是长指针类型的，实在是太方便了。

在实现这些库函数的时候，需要注意一点，那就是在访问硬件时候调用相关的驱动。比如，访问键盘就调用键盘驱动提供的方法，访问磁盘就调用磁盘驱动提供的方法。这样一来才能真正的实现前面所说的硬件抽象的作用。

7.4 代码细节

在具体实现的时候，有一个地方是需要特别注意。那就是 printf 和 scanf。这两个函数有同样的一个问题，那就是它们都使用可变参数。

在所有 C 语言的标准库函数里面，最难实现的就是 printf 和 scanf，因为这两个函数涉及到可变参数。C 语言在处理变长参数时有很多规则。下面是我从 vc1.5 里面找出来的几个宏，就是用于可变参数处理的：

```
#define _INTSIZEOF(n)     ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )
#define va_start(ap,v) ap = (va_list)&v + _INTSIZEOF(v)
#define va_arg(ap,t) ( *(t far *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
#define va_end(ap) ap = (va_list)0
```

其中在处理前调用 va_start 宏，然后在处理过程中调用 va_arg 来获得相应的参数，最后调用 va_end。详细的做法我就不说，在参考资料【12】中有很详细的介绍。我这儿要说是的我在写这个函数时遇到的一些问题。

首先，遇到的就是一个很奇怪的问题。问题是这样的：当输出 printf("%d%s", i, str);的时候回出错，而输出 printf("%s%d", str, i);的时候却没有错。我折腾到凌晨两点多页没有找到出现问题的地方，我最后把问题归咎与 va_arg(ap,t)这个宏，我认为这个宏处理的时候对指针的处理有问题。后来才发现不是这个样子的。是因为在可变长参数中，应用的是"加宽"原则。char, short 被扩展成 int【13】。而我是按照 linux0.10 中的做法弄的，直接使用了 unsigned long 当然就出错了。正确的做法是使用 unsigned int 类型。

后来又遇到了负数问题。负数是做了特殊编码的，所以应该需要做编码处理。我没有处理，输出当然就会出错。这个问题没有修正，因为这个功能应该是很少使用的。

在 printf 中，我做了一点扩展。我添加了一个%b 类型。这个类型用来输出二进制数。很多时候我需要输出二进制数，一般都是自己写个函数专门来处理，现在不需要了。我可以直接使用 printf 来完成了。

由于各种原因，在使用 printf 有以下几点需要注意：

1、对于指针必须显示的使用长指针，不支持短指针，使用短指针会出错而且没有提示。

2、对于类型不能自动适应，比如想输出 int 类型的数据就必须对应%d，输出 long 类型数据必须对应%ld，%x%b%o 于此类似。

3、对于负数的处理很多时候会出错，对于负数的输出除了%d 和%ld 以外的其他标识符都是不可靠的，几乎可以肯定输出的是错误的数据

注：以上所有错误都不会崩溃，仅仅是得不到正确的输出。

sanf 没有写，原因是时间不够了。赶时间，只好把它去掉了。没有它也并没有太大的影响。只是不够方便了。它完全可以用 gets()和数据转换函数来代替。

八、内存管理

8.1 简介

内存管理在操作系统中起着非常重要的作用。我一直认为，开发操作系统的时候在完成基本的引导以后最先要做的就是内存管理模块。因为，每次是程序运行的基础。其他的模块无论是文件系统还是进程调度都是以它为基础的。

内存管理模块主要是对外提供两个功能一个是内存的申请，另一个是内存的释放。内存的申请和释放是有很多技巧的。因为，内存的管理需要满足以下要求：

1、尽量的快速

如果速度过慢会直接造成系统变得很慢。因为在程序运行时申请和释放内存是十分频繁的动作。

2、尽量的紧凑

程序有一个局部性原则。在计算机里面，利用这个原则做了很多工作，比如高速缓存、虚拟内存等等。为了最大限度的发挥这些机制的作用，需要内存管理模块分配的内存尽量的紧凑。

当然，在我的操作系统里面由于某些原因，并没有考虑上面的这些要求。我的目标仅仅是简单的实现，而并追求实现的十分完美。

8.2 相关原理

在操作系统理论里面，关于内存管理讲解最多的就是分页式管理和分段式管理，还有就是两者的结合段页式管理。可惜，这三种内存管理方式我们都用不上。因为，我的操作系统在实模式下，没有办法使用这些管理技术。这些内存管理技术都是依赖硬件在保护模式下实现的。

在实模式下，内存的访问是以“段地址+偏移量”的形式来描述线性内存地址，也可以说是一种段式管理。但是这儿的段不是固定的，可以从任意的 16 字节对齐的地方开始，而且段的大小是固定的 64KB。

因此，我不可能一个段一段的分配。在 dos 下的做法是按照 16 字节对齐，然后仅仅返回段地址。其实，它并没有把整个段分配给用户，而是把地址转化成了一个偏移量为 0 的段。也就是说用户只能使用这个段的开头部分。这样做有一个小小的坏处那就是，因为段地址需要两个字节对齐，所以可能会浪费掉一个字节。但是，由于返回的数据是一个字节（段地址），而不是两个字节（段地址+偏移量），所以这个小小的缺点被大大的弥补的。因为返回一个字节意味着在用户程序里面只要使用短指针就可以了。对于段地址由于是统一的，所以可以统一处理。这也是为什么在以前的处理器里面默认使用的是短指针的原因之一。但是，我并没有这样做。原因是：我并不使用短指针。短指针很多时候会出现莫名其妙的问题，想找到这些问题十分困难，所以，我不使用短指针，全部使用长指针。

我对 dos 的内存管理的研究不是很深，仅仅是通过一些书籍看到过一点理论，并没有详细的去看源代码。在 dos 里面分配给用户使用的每个内存块都对应这一个内存控制快 MCB（Memory Contrl Block）。它的定义如下：

```
 struct MCB
{
    char magic;          // 幻数，4DH 或 5AH
    unsigned owner; // 所有者的 PSP 段地址
```

```
    unsigned size;  // 该块内存的大小
    char reserved[3];   // 保留
    char filename[8];   // 从 dos4.0 以后才具有
}
```

注：

1、如果是最后一个内存块其幻数是 5AH,其余内存控制块使用的幻数是 4DH。

2、在 dos 里面可以通过 21H 中断的 33H 号子例程来获取到第一个内存控制快的地址。

显然，在 dos 下使用的是在某个地方集中存放内存控制段的方法来管理 MCB 的。这样做的好处是快速和安全。由于是线性表，所以处理起来很方便。又是集中存放，与实际的内存隔离开,所以安全,不会因为用户程序对申请到的内存使用不当而造成内存管理损坏的问题。

在我的操作系统里面没有使用这种策略，而是采用的其他类似的方法（链表）来实现的内存管理。

8.3 具体实现

先来看看在实模式下内存的布局情况，如下图：

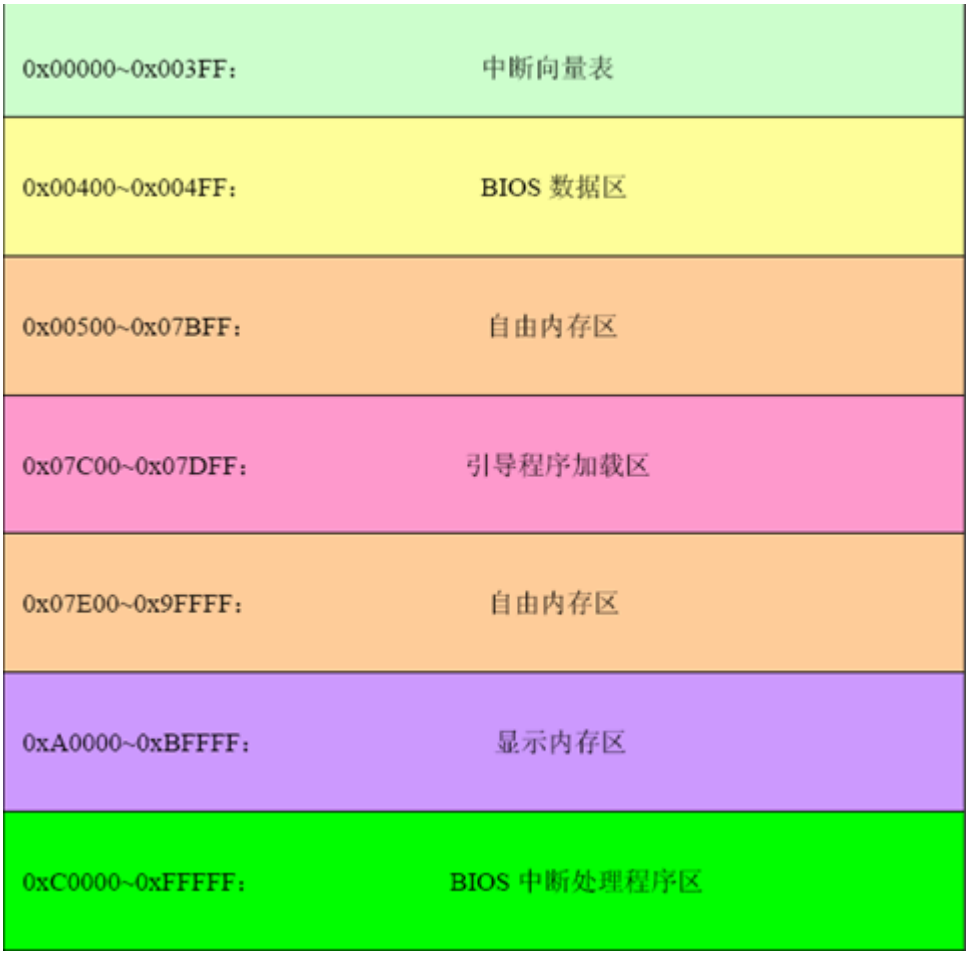

图 8.1

从上图可以看出，其中可以供操作系统使用的内存并不足 1MB，才 600 多 KB。在除去内核和驻留程序占用的内存，用户程序可以使用的内存就更少了。回到正题，下面就来看看我是如何实现内存管理的。

我也有一个和 dos 下类似的 MCB 结构，定义如下：

```
typedef struct memControlBlock
{
    struct memControlBlock far * front;
    struct memControlBlock far * next;
    BOOL bSate;      // 内存使用状态，FREE：空闲；USED：使用
    BYTE byProcessID;   // 内存所属进程号
    DWORD dwSize;   // 实际可以内存的大小，不包括 MCB 的大小
}MCB;
```

显然，我使用的是一个双链表结构。在实际使用时内存就会是下面这种形式：

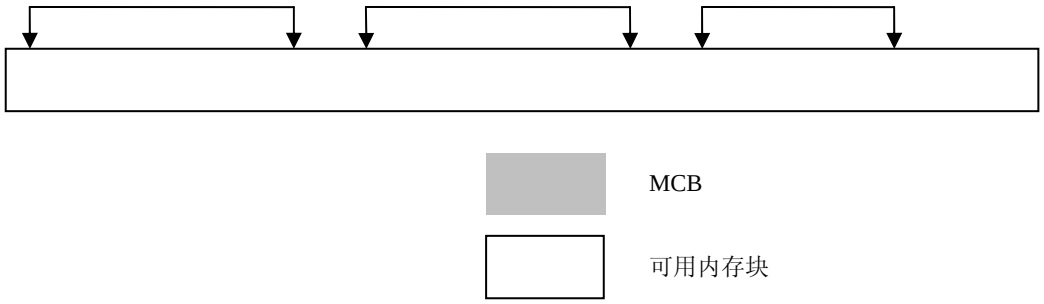

图 8.2

但是，我的内存管理并不管理全部内存，而仅仅管理 0x500-0x70000 这一段内存。只有这一段内存是分配用户可以使用的内存。0x500 以下和 0x70000 以上都是系统保留的内存。我并没有考虑内存过小的情况，也就是说当内存小于 0x70000 的情况。实际上当内存小于 0x95000 的时候内核就不允许加载了。会提示说内存不足。这是考虑到以下两个原因而做出的选择：

1、现在的内存肯定都超过 1MB 了。所以在实模式下一般都是可以使用全部的内存。只有在少数情况下无法使用全部的内存，那就是在启动这个操作系统以前有程序已经运行了。这种情况很少，但是是存在的。我以前就写过一个虚拟软驱的软件，就会在启动操作系统以前启动，并且会将可用内存的最高端的一部分划归为自己使用。但是，这种程序不会太大，一般不会超过 64KB。所以，我允许的最小内存是 0x95000（其中，系统堆栈使用 0x5000）。

2、第二个原因是使用链表来管理内存更加的方便一些，因为链表大小是随着内存的分配情况而变化的，所以不会出现浪费内存的问题。但是，也存在不安全因素。由于我做的不是商业产品，所以不考虑这个问题。

也就是说，内核运行以后内存的使用情况是这样的：

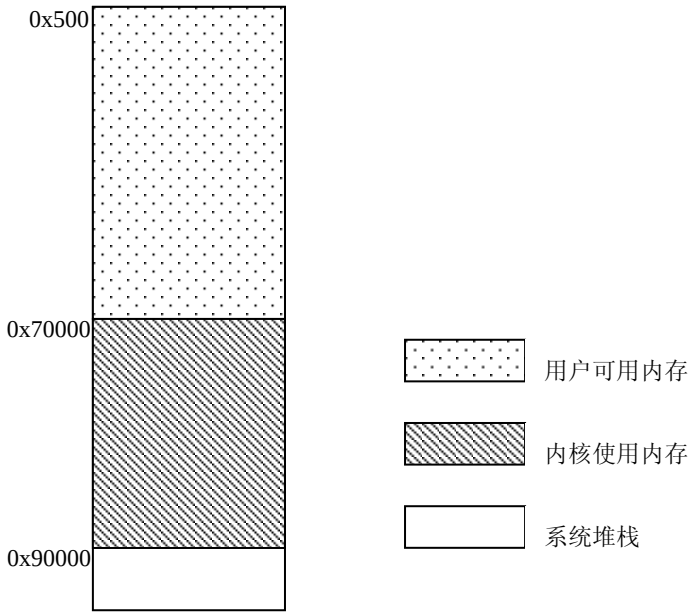

图 8.3

8.4 代码细节

实现的时候就比较简单了。由于都是 C 语言代码，而且是双链表操作，所以还是比较容易理解的。程序的大致流程如下：

一、初始化

会在内核初始化的时候被调用。

内存管理的初始化主要的任务就是手工构建第一个 MCB。要注意的是第一个 MCB 肯定在 FRIST_MCB（0x500）处。

二、申请内存

申请内存的流程如下：

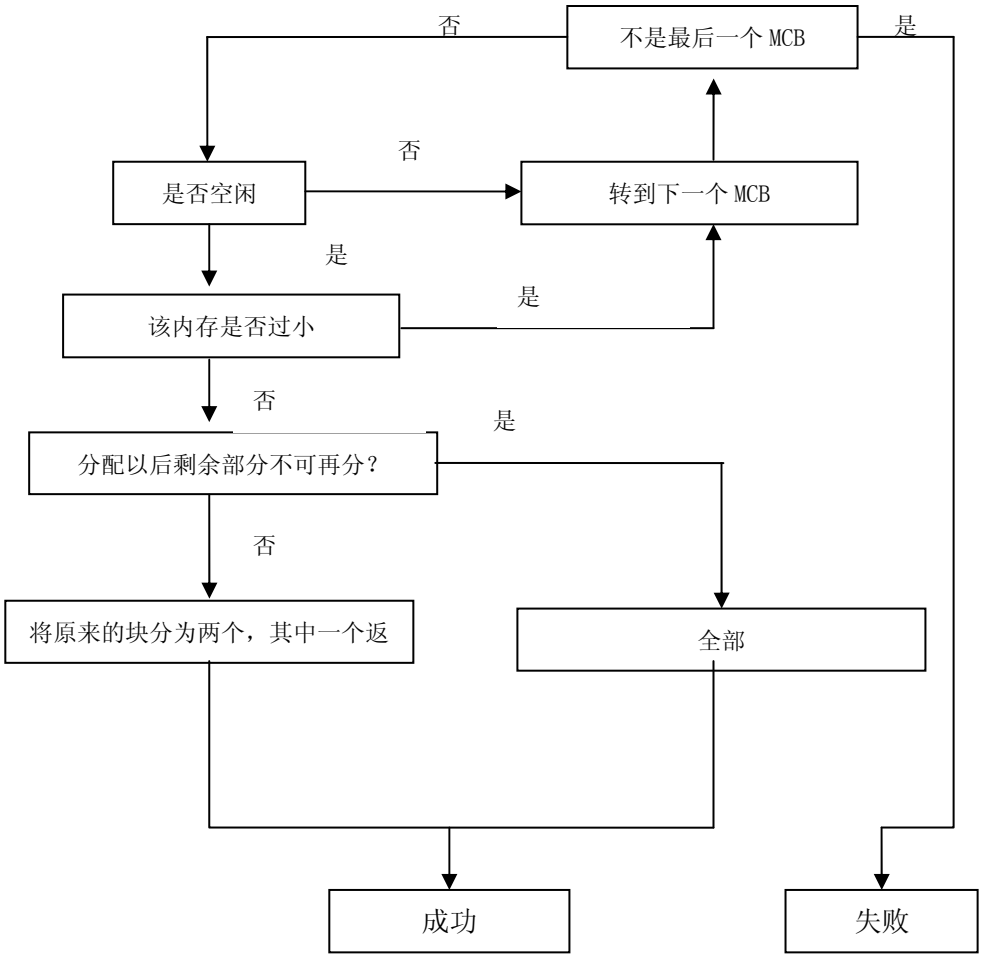

图 8.4

分配内存的时候有两种情况需要考虑，一种是找到的可以内存块在被分配以后剩余的部分已经无法在分了。在这种情况下，应该把该快内存全部分配给用户使用。另一种情况是，剩余的还可以在分，那么就把这个内存块一分为二，一部分给用户使用，一部分留着下次分配。

这儿要说的是，每次分配都会从链表头开始找，直到找到可用内存为止，如果找不到就会返回一个空指针。在 dos 下会返回最大的可用内存。我不是这样做的。我是尽量的分配，如果分不了就直接告诉不能得到你想要的内存，这是 C 语言标准库函数中的做法。我这样做的原因是，我不会限制用户申请内存。比如在 dos 下对 COM 程序申请内存就有限制。我这儿没有，也就没有必要处理的那么复杂。

三、释放内存

释放内存的流程如下：

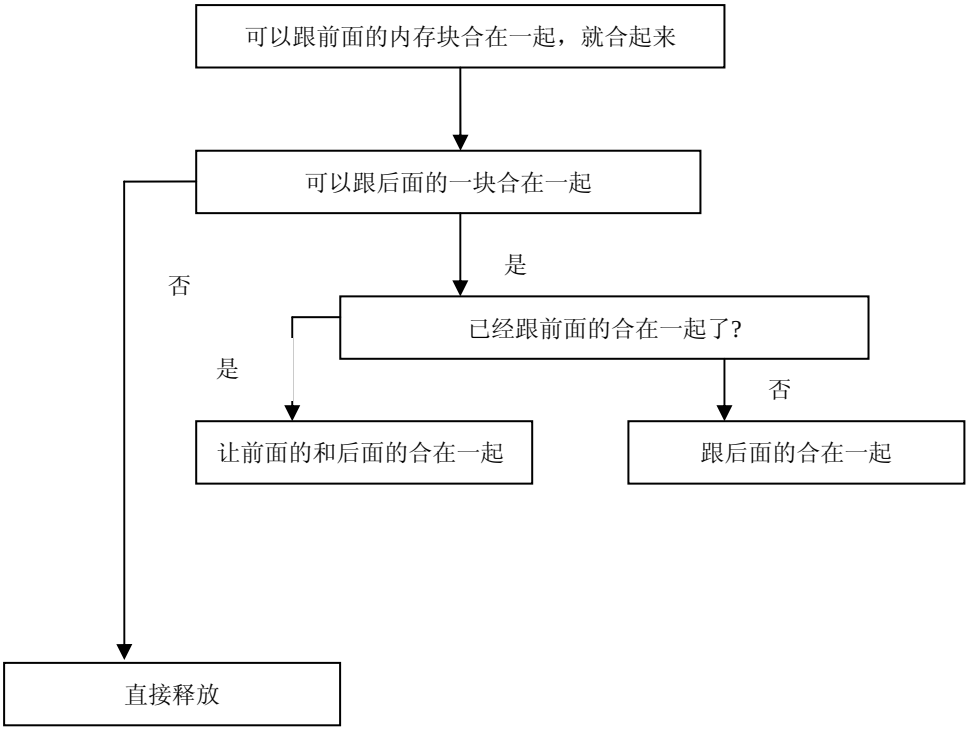

图 8.5

从上图可以看出释放比分配的时候要简单一些。因为在释放内存的时候不用考虑的问题比较少，仅仅考虑前面一个内存块和后面一个内存块就可以了。而且我使用的是双向链表，处理起来也很简单。但是分配起来就不是那么容易了。分配的时候要考虑很多问题。我这儿考虑的还不是很细，比如还要考虑是否紧凑等问题。

其实，我可以使用两套分配内存的方案的。一套用于自己，一套为用户提供服务。提供给自己的，采用上面的方法来分配和释放内存。提供给用户的将 MCB 分离出来放在系统申请的空间里面，这样就可以在保持现有优点的情况下有照顾了安全性。但是，这样做比较麻烦。以后如果有时间可以回过来再弄这一块，就是内存分配的安全性问题。

九、文件系统

9.1 简介

操作系统在我们的计算机里面扮演的一个重要的角色就是资源的管理者。它管理着我们的硬件，让我们的屏幕显示图像，让我们的耳机发出声音……但是，它管理的最重要的东西不是硬件，而是数据，用户的数据。我们使用电脑不是因为这些硬件，而是因为数据。为了处理这些数据，为了存放这些数据。所以，操作系统的一个很重要的功能就是数据的管理。那么，操作系统是怎么管理数据的呢？如何有效的，安全的保存数据？如何方便的快速的存取数据呢？这就是文件系统的功能。文件系统的作用就是管理硬盘上的数据，为用户提供数据的读写支持，保证数据的正确性和安全性。

其实，文件系统有不仅仅是用来管理数据的。它还可以用来管理硬件。这一点在 windows 下我们是看不到的。但是，在 UNIX（Linux）下，就很明显了。这就像一个饭店里面的厨师可以到街头小巷卖早点是一样的。文件系统确实可以用来管理硬件。为什么呢？我们可以这样来考虑。硬件的基本功能不外乎两种，一个是数据的输出，一个是产生数据。有可能有些硬件仅有数据的输出，比如显示器；有些仅有数据的产生，比如键盘；而某些硬件更是两者都有，比如网卡、声卡等等。但是，不会有其他的了。不会存在既不输出数据也不产生数据的硬件。也不是绝对的，比如鼠标垫就既不输出也不输入，但它并不能成为计算机的一个组成部分。既然硬件都是跟数据交道的，就很好办了。数据的输出不就是往文件中写数据吗？而数据的输入很显然就是从文件中读数据。也就是说，我们可以把一个设备看成是一个文件。数据的输入和输出就是对文件的读写。这种处理方法很好，它很大程度上屏蔽了硬件的复杂性。fopen 相比学过 C 语言的都会用。但是，从设备读取数据却不一定。比如，在 windows 读取摄像头的数据，读取声卡的数据，都是比较复杂的事情。而有了设备文件的概念以后事情就变的简单多了。我们可以使用 fopen 打开一个设备，再使用 fread 读取其中的数据，最后使用 fclose 关闭掉这个硬件。是不是方便很多？

现在，由于各种原因，数据的安全性变的异常重要。相比于其他问题，比如效率，安全性更加难以解决。这儿讲的安全性不是说某个人删掉自己的某个文件的时候操作系统不让删，并且告诉你说这个文件对你来说很重要删除以后你会后悔的。计算机，确切说是计算机软件，还没有那么聪明。他不能分辨那些是重要的那些是不重要的。它只能按照你定义的一系列规则来行事。比如你告诉它，这个文件除了我别人谁都不能查看也不能修改或删除。那么，文件系统，其实更重要的是操作系统，会自动的拒绝别的用户对这个文件的一些操作。这样我们就说操作系统保证了这个文件的安全性。那么，这一点是不是很容易实现呢？事实上实现起来很复杂。说实话安全性太复杂了，我在这方面连个皮毛都不懂。更不要说实现其中的一些原理了。所以，在我的实现中没有任何地方涉及到安全性问题。

安全性还有另外的一种理解。这种理解更加的类似与稳定性。比如，我们正运行着计算机的时候突然停电了，我不是说像我这样有电池可以续航，我说的是停电后便无法工作的情况。在这种情况下如何保证数据不丢失或损坏？这也是一种安全。这种安全，现在解决的已经比较完善了，比如日志文件系统就是专门处理这个问题的。当你的计算机被突然停电后可以通过日志来找到损坏的数据并试图恢复它。

前面提到了效率可以通过其他方式缓解。比如，读取数据比较慢，我们可以换速度更快的硬盘，换大内存，在内存中建立更大的数据缓冲区等等。但是，有另一种效率问题越来越引起我们的注意，那就是数据管理的效率。什么叫数据管理的效率呢？那就是怎么样才能让用户尽快的得到他们想要的数据，而这些数据他们可能根本就不知道在硬盘的什么地方。这就涉及到现在很热门的一个话题——搜索。数据的搜索技术，解决的就是如何使用户提高用户的效率，节省用户在数据的查找上的时间。实际上，这已经不是文件系统管理数据效率的问题了。他更加的应该叫做数据的查找，比如现在 google、百度等公司就有一种产品叫做硬盘搜索工具，它可以帮助你在你的硬盘上找到你想要的数据。但是，这些工具都是一些附加的外围软件，很多时候并不能很好的工作。比如你把它关掉一阵子再打开，比如你删掉一些东西，这些工具就往往会给用户提供错误的搜索结果。如果把这个功能涵盖到文件系统里面，事情就不一样了。我们可以把普通的数据文件单独放在一个地方，把程序等文件放在另一个地方。在你写入数据的同时文件系统就已经做好了你查找他们时的工作，在你修改他们的同时，文件系统已经为你准备好了新的查找工作。这不是简单的，做个索引就可以就解决的问题。比如你在看《美丽心灵》这部电影的时候，文件系统问自动给你找到关于约翰·纳什的一些数据，比如他的事迹、他的论文等等。这些数据甚至可以不再你的硬盘里面，因为我们前面经过设备可以作为一个文件来处理，而网卡就是网上的一些数据的抽象，这些数据

都可以从网卡（确切说是网络）中读取。Windows 曾经承诺在它的新的文件系统里面提供这样的功能，可惜最后很令人失望。

文件系统可以说是操作系统所有模块里面最复杂的一个了。它涉及到的问题很多，上面仅仅是提到很少的几点而已。比如网络文件系统就没有提及。

在我的实现中，没有涉及到这些高深的事情。我解决的是最原始的问题，数据的存放和读写。

9.2 相关原理

文件系统是一个很大的论题。按照我的能力，不要说发展它，即便是认识它弄懂它，也不是一件容易的事情。既然如此我不如把我的精力放在一些很基本的问题上来。是什么基本的问题呢？那就是数据的存放和读取。

我们知道一旦断电，内存中的数据就会丢失。要想数据不丢失，就必须把数据写到磁盘上，无论是硬盘、光驱还是软驱，甚至是网络，都可以存放数据。但是，数据如何保存呢？怎样保存才能更有效的使用磁盘而不浪费空间？如何保存才能更快的读取和写入数据？这些是文件系统设计者的问题。我并不考虑这些，我要考虑的是一个最简单的问题，在实模式下采用那个文件系统才更有利。这儿有利的概念是更容易实现，而不是数据的快速存取或者更加的安全等等。

在 TEOS 里面，我设计了一个全新的文件系统。并且实现了其中的一部分。但是，这样做很不好。最直接的坏处就是我没有办法在 windows 下使用成熟的软件读写这个文件系统里面的数据。我需要自己写一些工具来实现对它的读写。这在很大程度上增加了我的工作量。而且，我难以深刻的理解别人是如何实现的，他们在实现的时候都是考虑到了那些问题，他们设计出来的文件系统都有哪些优点和缺点。所以，在 QUOS 里面我选择实现 FAT12 文件系统，而不是去创建一个新的文件系统。

在一个磁盘（我这儿以硬盘来举例）中数据的管理分为好多的级别。

首先，分区，一个硬盘可以被分成多个分区，分区有分为主分区和逻辑分区。在 linux 下还有一个交换分区的概念。分区的信息会写到硬盘的 MBR 里面。这个我在引导程序那一章中提到过。通常每个分区都会有它自己的文件系统格式。这也是为什么我们在 windows 下，从一个分区剪切一个文件到另一个分区要远远比分区内的剪切慢的原因。

然后，文件。文件系统里面的数据会被分为很多的文件。各个文件存放着不同的数据。我在这儿把目录当成一个文件。因为在硬盘上存放的时候与其他文件没有多大的差别。不同的仅仅是它的用途。

接着是簇。簇是一个逻辑上的单位。它不像前面提到的分区和文件一样可以被用户看到，也不像下面要说的扇区一样真真实实的在硬盘上存在。它仅仅是为了文件系统处理上的方便和灵活而存在的概念。比如在有些文件系统里面簇的大小是可以改变的。簇是文件的最小单位。一个文件必须包含整数个簇。

最后才是扇区。现在来讲，扇区的大小都是固定的了。都是 512 个字节。据一些比较古老的资料讲，扇区的大小是可以变化的。对于现在来讲没有这个必要了。因为有簇的概念，一个簇可以包含一个或多个扇区，从软件层解决问题要比从硬件层上解决问题容易的多。而且现在的磁盘都很大，512 个字节已经是很小的单位了，没有再小的必要了。

那么 FAT12 是怎么做的呢？拿一个软盘来讲，被格式化成 FAT12 以后就会变成下图所示的格式：

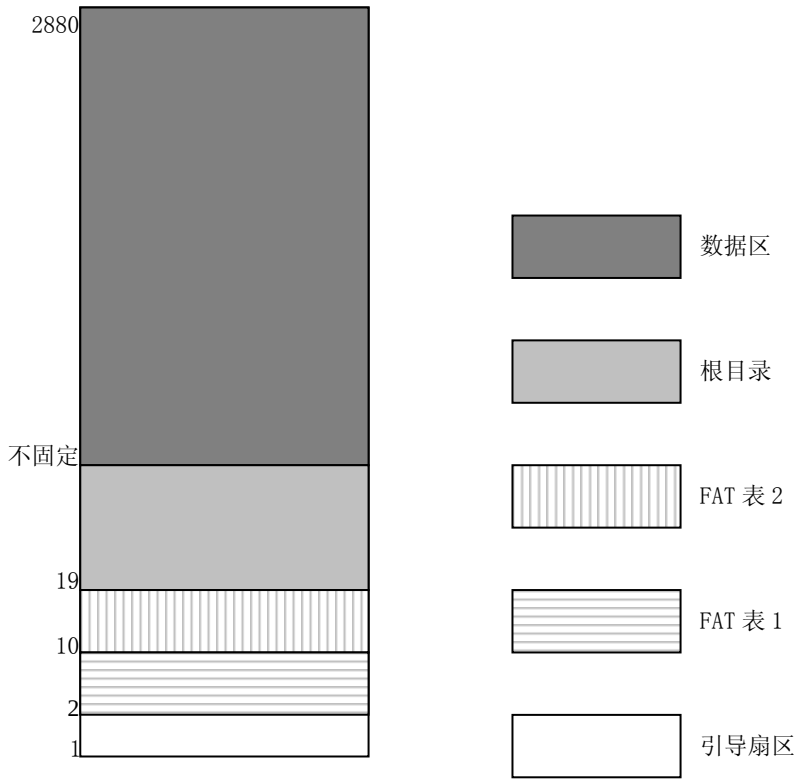

图 9.1

从上图可以看出，引导扇区和 FAT 表的大小是固定的，而根目录是不固定的。那么各个部分都是什么功能呢？

引导扇区，这个在引导程序那一章讲的已经十分详细了，这儿就不再重复了。

FAT 表，FAT 表是用来找到文件具体在什么位置的表格。也就是说，FAT 表相当于一个索引，它记录了某文件的数据具体被存放在磁盘的哪个扇区里面。如上图所示，FAT 表有两个，而且他们的大小是相同的。我们可以把 FAT2 看成是 FAT1 的备份。在很多软件中就是采用这个特点来实现分区的还原功能的。在建立还原点的时候只是把 FAT1 保存到 FAT2 里面。然后在开机的时候用 FAT2 恢复 FAT1，开机以后对该分区的所有的修改都仅仅是反应在 FAT1 里面。这样，每次开机以后用户所保存的数据有都消失了。而且还是可以保护还原点之前写入的数据不被覆盖或删除。FAT 具体的格式我到最后再讲。

根目录，根目录是一个比较特殊的目录。它必须存在，而且起点是固定的。但是，大小不固定。大小被放在了文件系统参数里面。在引导程序一章，我讲过文件系统的参数，其中有一个叫 BPB_RootEntCnt 的常量，就是指定在根目录下最多可以有多少个文件的。那么，根目录是如何组织的呢？根目录实际上是一个线性表。每个表项的格式如下：

```
struct dirEntry
{
    char strFileName[8];    // 文件名
    char strFileExt[3]; // 文件扩展名
    BYTE byFileAttrib;      // 文件属性
    BYTE byReserved[10];    // 保留
    WORD wTime;          // 最后一次修改时间
```

```
    WORD wDate;          // 最后一次修改日期
    WORD wStartClus;         // 开始簇号
    DWORD dwFileSize;        // 文件大小，以字节为单位
};
```

其中的文件名、扩展名和文件大小的含义都很容易理解就不讲了。

最后一次修改时间的格式是——时：分:秒（5:6:5）。也就是说最高的 5 为代表小时，中间的 6 位代表分钟，最后 5 位代表秒。其中的秒是的精度是 2，也就是说要乘以 2 才是真正的秒数，所以精度为 2。

最后一次修改日期的格式是——年月日，7:4:5，其中的年份是从 1980 年开始算起的。也就是说年份需要加上 1980，最后得到的结果才是真正的年数。

开始簇号，代表的是该文件的第一个簇。需要说明的是在 FAT12 里面一个簇仅包含一个扇区，也就是说簇和扇区是等价的。还需要注意的是簇号是从 2 开始的，而不是 0 或 1.那么怎么通过簇号找到文件呢？过程是这样的：

首先，簇与 FAT 表的项序号是对应的。我们可以通过簇号找到对应的 FAT 项。FAT 项记录的是该簇的一些属性，比如是否为最后一个簇。FAT 项值的含义如下：

000 - 此簇未用；

FF8 - FFF 该簇为文件的最后一簇；

FF0 - FF7，此簇为坏，不可用；

其它值表示文件下一簇的簇号。

FAT 表中并没有记录文件数据的位置。那么，如何根据簇找到数据呢？是这样着的

数据所在扇区号 = 簇号-2 + 根目录开始扇区号 + 进 1 法取整（(根目录最大文件数 * 32) / 每个扇区的字节数）

也可以写成：

数据所在扇区号 =簇号-2 +根目录开始扇区号 + (根目录最大文件数 * 32 + 每个扇区的字节数 - 1) / 每个扇区的字节数

我们前面提到根目录开始扇区号是固定的，其实也可以通过 FAT12 文件系统参数来计算，就是：

根目录开始扇区号 = FAT 表个数 * FAT 表占用扇区数 + 引导程序占用扇区数

这三个值都是在 FAT12 的文件系统参数里面包含的。但是，他们一般不变，所以这个值是固定的 19。

还有一个就是在前面的计算数据所在扇区号的公式里面为什么减二呢？原因在与簇号是从 2 开始计数的。

通过这样就找到文件数据所在的扇区，然后通过 FAT 表可以找到文件的数据都是在那几个簇里面。两者结合就可以找到文件所在的位置了。最后我们来说一下这个 FAT 表。

FAT 表和根目录一样也是一个线性表。可惜不是字节对其的，所以处理起来有点麻烦。FAT 表中每一项占 12 位，也就是 1.5 个字节。所以，在分析一个 FAT 项的时候要先解析一下，计算出 FAT 表项的值然后再做决定。之所以这样做，就是为了节省这半个字节。一个 FAT 项节省半个字节，全部的 FAT 表就可以节省 1/4,也就是说如果一个 FAT 项是 2 个字节的话，那么 FAT 表就不是现在的 9 个扇区而是 12 个扇区。对于整个软盘来说就会浪费掉 6 个扇区，也就是 3KB 的大小。为了这 3KB 的大小，我们就需要这样来计算一个 FAT 项的值：

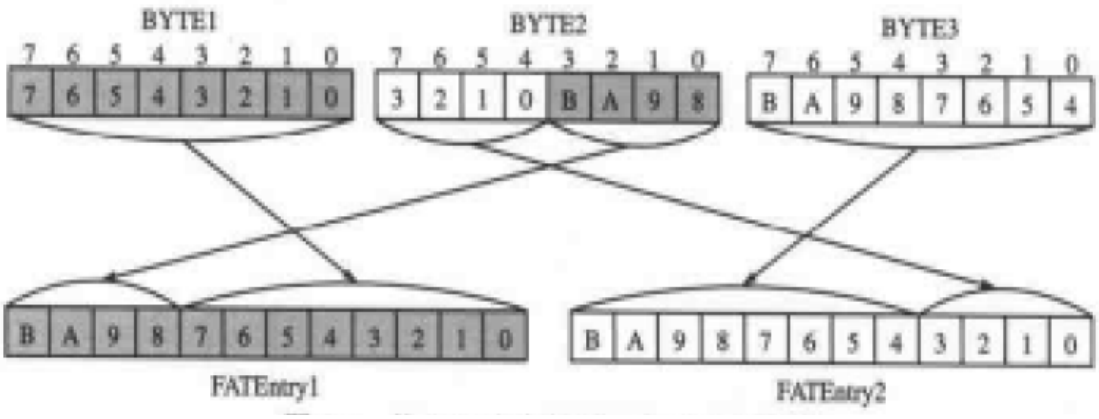

图 9.2

这个图来自于参考资料【1】。它讲的就是如何计算 FAT 表项的值。我就不具体的解释了。

9.3 具体实现

我在实现的时候考虑了很多具体情况。比如我需不需要支持虚拟文件系统？我是不是把文件系统的功能全部实现？还是是仅仅实现一部分？最后的决定是，现在虽然不支持虚拟文件系统，但是留出接口来。本来计划把 FAT12 文件系统全部实现，但是由于时间原因只实现的读部分。而且读部分还不支持目录。

尽管我仅仅实现了其中的一部分功能，但是其代码量依然是最大的。实际上很多的代码都浪费在了处理 FAT 表的半个字节问题上。由于我没有读过其他人的实现，所以并不清楚别人是如何实现的，或许我的实现不是很简练。废话少说，我们还是看看我是如何具体实现的。

首先，文件系统会在内核初始化的时候被初始化。初始化过程很简单，不过就是读取和分析文件系统的“超级块”（super block）。我在前面提到过 FAT12 文件系统在其起始处有一些参数，这就是它的超级块。里面记录了关于它的几乎所有信息。比如一个扇区多少字节，FAT 表占多少个扇区等等。在分析 FAT 文件系统的时候这些参数是很有用的。比如一个扇区多少个字节。虽然现在基本上已经固定下来是 512 个字节，但是，为了最大的兼容性，采用文件系统里面定义的参数还是比使用常量要兼容的多。因为文件系统参数是可以修改的，它就在磁盘的起始处。这其实是一种后期绑定的思想。就是把会变化的部分尽量的放在最后决定，甚至是不决定，而是在使用的时候再输入。使用文件系统的参数实际上是采用配置文件，这个配置文件就是文件系统的“超级块”。

然后，就是文件的打开了。打开文件要做的工作就是把该文件的一些信息读入到内存中来，其中最主要的就是该文件的开始簇是什么。通过开始簇就可以找到该文件的数据所在的扇区。当然，还有文件的大小，最后修改日期等等信息也是保存在目录里面的，这个在前面讲 FAT12 文件系统原理的时候已经提到过了。

在打开文件的时候有一个路径问题。比如要读取“A:\TEST\W.TXT”这个文件，就需要先找到“A:\TEST\”这个目录，然后再从这个目录里面找到 W.TXT 这个文件的基本信息。然后才能通过这些基本信息在 FAT 表中找到其数据所在扇区，最后才能将数据读到内存中来。

我在实现的时候，没有实现对目录的支持，也就是说我没有提供解析目录的功能。所有文件我全部默认在根目录下。不是说，这个功能很难或是无法实现，而是我没有时间也没有必要去实现它。如果以后需要的话可以很容易的将这部分功能加进来。现在的 fat12_Path 函数原计划就是实现这个功能的。现在，仅仅是在这个函数里面分析了根目录而已。

接下来，就是读取文件的数据了。在上一步文件打开的时候已经，从目录里面获取到了该文件的开始簇号，通过开始簇号就可以计算出该文件的数据所在的扇区。在前面的 FAT12 文件系统原理里面，我讲到这个开始簇号既是 FAT 表的索引号，也是计算数据所在扇区的一个变量。FAT 表其实就是一个单链表。知道了第一个项也就知道了所有的项，也就相当于知道了该文件所在的所有扇区。getClusterNO 这个函数就是用来解决这个问题的。

不是说，找到数据所在扇区就万事大吉了。还要分析，用户到底需要那些数据，可能用户仅仅需要其中的一两个字节而已。而 BIOS 仅仅可以整个扇区整个扇区的读取，所以就必须先将数据读取到一个缓存里面来，然后从缓存中将用户所需要的那部分数据拷贝过去。

最后，当文件读取完成以后，还要修改文件结构的偏移量。否则下次再读的时候就会有问题了。fseek 相信都用过，它其实就是做这个事情的。fseek 的实现比较简单。这儿就不再详细介绍了。

我在前面提到过我为虚拟文件系统预留了接口，这个接口在哪儿呢？就在 File 结构里面。这个结构是这样定义的：

```
// FILE 结构
typedef struct File
{
    char strName[FILE_NAME_SIZE];   // 文件名
    char strExt[FILE_EXT_SIZE];     // 扩展名
    BYTE byMode;        // 文件读写方式
    BYTE byAttribute;   // 文件属性
    DWORD dwSize;       // 文件大小
    DWORD dwPos;        // 读写位置
    BYTE byDevNO;       // 所属设备号
    WORD wTime;             // 最后一次修改时间
    WORD wDate;             // 最后一次修改日期
    WORD wFirstCus; // 磁盘中的开始簇号
    struct FileOpts far * pOpts;    // 文件系统操作函数
}File;
```

其中的 pOpts 这个指针其实就是用于虚拟文件系统的。它指向一个结构，这个结构记录了对应文件系统的操作函数。这样就可以通过这个指针关联到不同的文件系统上面去。这儿要处理的就是在文件打开的时候做一些处理，需要在打开文件的时候确定这些操作函数到底是什么。也就是文件具体在那个磁盘或者分区里面，这个磁盘或分区采用的是那种文件系统。然后，将指针指向该文件系统的操作结构就可以了。

当然，这样一来在初始化文件系统的时候也要稍微麻烦一点，需要把各个分区和磁盘的文件系统类型及其超级块全部解读一遍。我并没有提供这些支持，而是预留了这些接口。未来，如果有未来的话，这儿肯定会得到扩展，实现完整的虚拟文件系统的。

十、程序加载器

10.1 简介

一个操作系统不仅仅自己可以运行就可以了。因为用户使用计算机有两个目的，一个是数据存储，另一个就是数据的处理。而操作系统可以完成绝大多数的数据存储工作，但是它只能完成很少的数据处理工作。这样就需要其他的一些软件来完成操作系统完成不了的那一部分工作。

那么操作系统系统是如何运行其他程序的呢？一个应用程序是通过哪些步骤一点一点的运行起来的呢？这些都是程序加载器的工作。

在操作系统理论里面一般都不会讲这个模块。在编译原理等课程里面也不会讲这个问题。同样没有相关课程讲解的还有链接器。在实际应用中，这两个模块的作用是很大的。可惜，在理论上得不到应有的重视。我在这儿不打算讨论链接器，仅仅讨论加载器。因为链接器不是操作系统的一部分，虽然很多时候加载器和链接器完成的是相同的工作。但是，它们所处的位置是不一样的。我不会涉及到链接器的实现上。如果想知道关于链接器的详细原理和实现细节可以读一下参考资料【8】。里面详细的介绍来链接器和加载器的原理和实现的细节。

10.2 相关原理

我在内核加载程序里面已经提到过了，应用程序一般来讲不是纯粹的二进制格式。也就是说一个程序不是被从硬盘上直接读到内存中就可以直接运行了。它还需要很多的步骤，需要解析，需要动态的链接等等。

在早期的操作系统里面，应用程序确实就是纯粹的二进制格式的。比如 dos 下的 COM 文件就是二进制纯粹格式的。我们可以直接加载然后运行。不需要做任何麻烦的工作。

但是，随着计算机的发展，程序越来越大，越来越复杂。纯粹的二进制格式也就是无法满足需要了。我们知道 COM 格式的程序不能超过 64KB，也就是不能超过 8086 处理器的一个段大小。要想超过一个段就必须有一定的格式，因为跨段访问和跳转都不能仅仅使用偏移地址来完成。而需要段地址加偏移地址。但是，这个段地址是未知的。一个应用程序每次被运行加载到内存中的位置是不固定的。所以，就必须在加载运行程序前对应用程序做一些处理。这些处理就叫做程序的重定位。这就是 dos 下的 exe 格式解决的问题。

在后来的操作系统中，问题变的更加复杂了。操作系统进入了保护模式，每个应用程序都可以使用 4GB 的地址空间。应用程序也变的庞大起来，不再可以一次性全部加载到内存里面来了。而且，很多的程序都使用相同的部分，这些相同的部分没有必要在每个程序里面都有一份拷贝。不仅如此，一个操作系统可能需要在多个硬件平台上运行。但是，他们想让他们底下运行的应用程序可以有相同的格式，这样就可以在实现操作系统的时候不再专门对每个平台做处理。于是，PE 格式、ELF 格式和动态链接器技术开始出现。它们就是为了解决这些问题而出现的。它们考虑到了保护模式，考虑到了代码复用，考虑到了动态链接，考虑到了平台性等等各种问题。由于我的这个操作系统并没有涉及这些问题。所以，不对其进行深入的分析。如果想了解其中的秘密可以阅读相关的资料，比如参考资料【8】。

在我的操作系统里面实现的主要是 DOS-EXE 格式。我的内核采用的就是这种格式。我在 build 那一章已经详细的讲解了 DOS-EXE 格式的细节和具体的解析方法。我在这儿就不再重复了。我下面要讲的是在前面没有讲到过的一些东西。比如，PSP。

我们都有一个经验。那就是如果我们把同样的一个程序放在不同的目录里面运行会得到

不同的效果。比如我在程序里面打开了一个名为“1.txt”的文件。在我打开它的时候并没有指定这个文件的位置。这个时候操作系统就会认为这个文件在程序的运行目录里面。这儿的程序运行目录就是程序运行环境的一部分。它在 dos 下被成为 PSP。

在 PSP 里面记录了所有的与程序运行时相关的一些参数。这些参数，仅仅是在程序运行的时候才会有。比如它的根目录是什么，它的父进程是什么，它的打开了哪些文件，它在被运行的时候应用户给他了哪些参数等等。在 DOS 下面 PSP 记录了很多东西。我没有深入的研究，这儿我仅仅给出在 freeDOS 下的 PSP 定义，但并不对其进行详细的分析。在我的实现里面是有一些差别的。详细的介绍会在具体实现一节里面

```
typedef struct
{
    UWORD   ps_exit;    /* CP/M-like exit poimt     */
    UWORD   ps_size;    /* memory size in paragraphs    */
    BYTE    ps_fill1;   /* single char fill     */
    /* CP/M-like entry point                */
    BYTE    ps_farcall; /* far call opcode      */
    VOID    (FAR *ps_reentry)(); /* re-entry point      */
    VOID    (interrupt FAR *ps_isv22)(), /* terminate address */
        (interrupt FAR *ps_isv23)(), /* break address   */
        (interrupt FAR *ps_isv24)(); /* critical error address */
    UWORD   ps_parent;  /* parent psp segment       */
    UBYTE   ps_files[20];   /* file table - 0xff is unused  */
    UWORD   ps_environ; /* environment paragraph    */
    BYTE FAR *ps_stack; /* user stack pointer - int 21  */
    WORD    ps_maxfiles;    /* maximum open files       */
    UBYTE FAR *ps_filetab;  /* open file table pointer  */
    VOID FAR *ps_prevpsp;   /* previous psp pointer     */
    BYTE FAR *ps_dta;   /* process dta address      */
    BYTE    ps_fill2[16];
    BYTE    ps_unix[3]; /* unix style call - 0xcd 0x21 0xcb */
    BYTE    ps_fill3[9];
    union
    {
        struct
        {
            fcb _ps_fcb1; /* first command line argument */
        } _u1;
        struct
        {
            BYTE    fill4[16];
            fcb _ps_fcb2; /* second command line argument */
        } _u2;
        struct
        {
            BYTE    fill5[36];
```

```
            struct
            {
                BYTE _ps_cmd_count;
                BYTE _ps_cmd[127]; /* command tail  */
            } _u4;
        } _u3;
    } _u;
} psp;
```

10.3 具体实现

我并没有照搬 DOS 下的实现方式，而是采用了其他的实现方法。这种实现方法十分相似与现代操作系统中的进程管理。是的，我也把一个运行的程序当成一个进程来处理。每个进程也有它自己的进程控制块。很是像模像样，好像真的支持多进程似的。其实现在并不支持。我并没有时钟中断函数，也没有调度函数。所以，虽然有进程的概念却没有进程的管理。只有用户进程和系统进程的切换。而且这种切换是谦让式的，就像 DOS 一样，如果用户进程陷入了死循环其结构就是死机。系统就会崩溃。

我这儿采用进程管理的思想来管理运行中的程序，而不是采用 dos 里面的那种思想，其原因就在于我有一个计划——实现保护模式下的多进程。这不是不可以实现，只是在实模式下实现意义不是很大，因为内存太少了。但是，从技术上讲是可行的。既然如此，我就希望实现一下。至于内存的问题，我可以通过其他方法来解决。

由于，我现在仅仅进行程序的加载和运行而不涉及到多进程的调度和管理，所以还是比较简单的。先来看看我的 PCB 结构：

```
struct PCB //进程控制块
{
    BYTE byProcessNO;       // 进程号
    //File far * file[MAX_FILE];    // 该进程打开的文件号
    struct fileNO file[MAX_FILE];
    char strRootPath[PATH_SIZE];    // 该进程的根目录
    char strArg[PATH_SIZE];         // 命令行参数
    union REGS r;                   // 该进程的运行环境寄存器的值，用于任务切换，但是现在还不支持任务切换
    struct SREGS sr;
    WORD IP;
    WORD SP;
    WORD BP;
};
```

其中每个成员变量的含义如下：

byProcessNO：它实际上是一个下标，这个下标就是全局数组 g_Process[MAX_PROCESS] 的下标。这个全局数组就是 PCB 结构体数组。对于进程号，还有两个函数，他们是 allocProcessNO 和 freeProcessNO。从名字就可以看出，他们是用来分配进程号和释放进程号的。

file： 这个数组记录的是文件号的使用情况。struct fileNO 的定义为：

```
struct fileNO
{
```

```
        BYTE NO;
        File far * fileStruct;
    };
```

strRootPath： 为该进程的根目录，现在并没有使用，因为我的文件系统并不支持目录。

strArg： 该进程的运行参数。

r、sr、IP、SP、BP： 全部是用于进程切换的时候记录其运行环境的。现在来讲，只有系统进程才会需要，因为用户进程一旦运行就不会停止，知道它运行完为止，也就不存在切换问题。但是，以后实现实模式下的多进程的时候这儿会十分有用的。

程序的加载和运行是在 exec 这个函数里面实现的。其实现的原理我已经在 build 里面讲到过了。而且这儿所做的处理实际上和 build 工具里面差不多。不同的时候这儿涉及到进程切换问题。需要先将运行环境和返回地址保存在系统进程的 PCB 里面来。然后，再跳转到用户进程。在用户进程运行完成以后会调用系统中断的 0x4C 这个子例程。它会恢复系统进程的运行环境然后在跳转的系统进程来运行。这样就完成了用户进程和系统进程之间的切换。

以后支持了多进程以后，这些功能都会放到调度函数里面来完成。构造好一个进程以后就会调用调度函数，然后在用户进程运行完毕以后也会调用调度函数，而不是像现在这样直接跳转到目标进程去。而且每个时钟中断来临的时候也都会调用调度函数。这些都是以后要实现的，现在还不支持这些功能。但是，可以看出，要想实现多进程的支持，对现有代码的改动是很小的。仅仅是把任务切换部分做一下调整就可以了。

十一、shell

11.1 简介

Shell 这个单词的本意就是外壳的意思。在操作系统里面，把这个单词作为用户界面的名字。用户界面其实就是操作系统的一个壳。它给用户提供的是一组易于使用和理解的命令，而不是操作系统内部哪些晦涩难懂的中断或系统调用。

一般来说，shell 可以分为两类，一类是字符模式的，也就是传统意义上的 shell。这种 shell 用起来不是很友好，因为你面对的是一个黑黑的屏幕，没有任何的暗示。如果你不知道其中的命令的话将不知所措。我记的看过一个文章写到一个初学者，在装上 dos 以后不知道该干什么才好。于是，费了半天的劲输入了一句话：who are you?结果可想而知。最好笑的是他有敲入了下一句话：what can you to do?

但是，对于另一种类型的 shell 就不同了。它就是 GUI。是图形用户界面。它里面给用户了很多暗示。这些暗示来自与现实生活。比如一个按钮跟小时候整天抱着听的单放机的按钮非常像，我们很容易的就会想到他可以被点击。在比如鼠标的移动等等。

但是，GUI 在降低操作难度的同时给软件的编写带来很大的复杂性。而且用户往往被程序华丽的外观所迷惑而不再重视其真正的内涵，比如那能不能更加快速有效的解决问题。用户宁愿使用那些功能不是很好但看起来很漂亮的程序，而不会去使用那些功能强大但没有漂亮界面的程序。

GUI 的另一个坏处是他降低了批处理能力。在字符模式下，我们很容易的将多个程序通过一个脚本变成一个批处理任务。在批处理过程中完全可以自动完成。但是，GUI 就没有办法实现这些。我们很难写一个脚本完成“安装程序 1-安装程序 2-处理文件 1-删除文件 2……”等一系列的工作。我们不得不守在计算机前随时听从计算机的调遣。它弹出一个对话框，你

就必须做出选择，不然他会拒绝往下继续运行知道你做出选择为止。在运行完一个程序以后想运行其他程序，你就必须先找到那个程序或其快捷方式然后双击它。并且等待在电脑前面，随时满足他的一切要求。

我在这儿并无意讨论字符界面和图形界面孰优孰劣。因为对于不同的用户有着不同的答案。从我实现操作系统的角度来讲，实现基本的字符界面就可以了。没有必要弄得那么复杂，去实现 GUI。而且对 VGA 的编程不是那么容易完成的。这同样是一个在所有课程中都不学而又十分重要的知识。

11.2 具体实现

我没有计划把我的 shell 实现的很复杂。Dos 的 shell 就是 command.com 这个程序。它仅仅是提供了简单一些命令和批处理文件支持。我实现的更少。甚至连命令都只有很少的几个。并不支持批处理文件。

因为时间实在是不够（我每天都在赶时间，依然不够），所以没有办法，我仅仅是把 TEOS 里面的一些实现搬了过来装点一下门面。他其实不能称为真正意义上的 shell。所以，我也就不再详细的讲解它是如何实现的，以及其中的代码细节。希望以后能有机会继续完成这个模块。

十二、驱动程序

12.1 概述

我在第一章概述里面就提到过，我的操作系统里面并不支持驱动模型。这是什么意思呢？也就是说，我不支持外部的动态的驱动程序。就像早期的 UNIX 一样，要想添加新的驱动，需要重新编译整个操作系统才行。这样做的目的也只有一个，保存我的操作系统的简单性。

不是说我不想实现，而是没有时间和精力来实现这个功能。所以，我把驱动程序放在了内核里面。而不支持外部的动态加载的驱动程序。

驱动程序的作用就是为软件提供一个访问硬件的接口。每个硬件都有一些他们特有的东西，不可能在每个软件里面考虑这些问题。而且很多硬件的秘密都是商业上的机密，写软件可能根本就不知道。所以，驱动程序一般由硬件厂商来提供。驱动程序给用户提供的是一个统一的接口，至少在同类软件中有一个统一的接口供应用程序来使用。在大多数操作系统里面，软件也不会直接跟驱动程序打交道，而是访问操作系统的提供的接口。操作系统提供的接口更加的统一和一致。

在我的操作系统中，没有支持那些特殊的硬件。我支持的都是最最基础的硬件。所以，驱动也都比较简单。对于驱动的访问也没有特殊的接口，因为对他们的访问都被隐藏到了我实现的 C 语言标准库函数里面了。

12.2 键盘驱动

键盘驱动很简单，他仅仅是对 int 16h 做了一些封装。使其更好用一些。需要注意的是我在这儿没有调用我实现的 C 语言标准库函数里面的 int86。因为 C 语言标准库函数是基于它的，而不是反过来。何况底层的驱动需要在性能上多做些考虑，如果调用 int86 可能会更方便一些，但是性能上会带来一些损失，这是我不愿意看到的。但是，也有例外。

因为驱动程序被调用的次数可能会很频繁，所以要尽量的简短而且快速。实际上，我更

愿意通过端口读写的方式实现。可惜我没有找到相关的资料，所以只好借助 BIOS 的中断了。

12.3 屏幕驱动

屏幕驱动是这三个驱动中依赖 BIOS 中断最少的一个。因为 BIOS 的中断实在是太慢了。如果输出字符比较多的时候会变得比较慢，所以，我不得不采用直接写显存的方式输出字符。虽然输出字符可以这样做，但是由于缺乏资料，我没有办法直接设置光标在屏幕上的位置，也没有办法直接卷屏，所以只好继续依赖 BIOS 的中断。

屏幕驱动也是这几个驱动中逻辑上最复杂的一个，因为我需要自己解决换行和卷屏问题。什么时候改换行，什么时候该卷屏都需要我自己来判断。如果处理的不是很恰当，就会出现莫名其妙的问题。比如在前面的时候输出到最后一行的时候光标会丢失，但是并不影响输出，而且当按一下退格键光标就会重新出现。实在是一个奇怪的问题。原因就是我在处理光标的时候有一个逻辑上的小错误。

12.4 磁盘驱动

磁盘驱动分为两个，一个是硬盘驱动，一个是软盘驱动。

由于现在的硬盘都是支持扩展读写的，也就说不再有 8G 限制。所以，我在硬盘驱动里面仅仅支持扩展读。当硬盘不支持扩展读写的时候就直接返回错误，而并不会尝试使用旧的读写硬盘的方式。其实这儿的检查已经属于多余的了。现在很少有小于 10GB 的硬盘了。所以，都是支持扩展读写的。

对于软盘的读写肯定需要使用古老的方式。因为软驱是绝对不会支持扩展读写的，没有那个必要。因此，我在软驱驱动中仅仅是封装了 BIOS 的磁盘读写中断。

还要说的就是，这两个驱动都使用的我实现的 C 语言标准库函数。我的考虑是磁盘读写很少是个耗时间的工作，没有必要省那一点点时间了。而且磁盘的读写应该不会非常频繁。实模式下的操作系统不像保护模式下的操作系统那样有虚拟内存，需要经常的把数据读入内存和写到硬盘上。所以，这样做不会带来多少影响。

参考文献

1、《自己动手写操作系统》于渊 编著，北京：电子工业出版社 2005.8

2、《FAT: General Overview of On-Disk Format》 Microsoft Corporation Version 1.02, May 5, 1999

3、《Intel Architecture Software Developers Manual Volume 3 System Programming》Intel 1999

4、《linux 内核源码情景分析》毛德操 胡希明 著，浙江大学出版社 2001.5

5、《linux 内核完全注释》赵炯 编著 北京：机械工业出版社，2004.9

6、《操作系统引导探究》谢煜波 哈尔滨工业大学

7、《硬盘结构及其分区简介》网上资料

8、《链接器与加载器》网上资料

9、《DOS EXE 格式文件的加载》网上资料

10、《freedos 1.0 源代码》网上资料

11、《linux 0.1.0 源代码》网上资料

12、《C 语言中可变参数的用法》网上资料

13、《C 语言中的可变长参数》网上资料

14、《dos 程序员参考手册》网上资料

15、《汇编程序设计教程》网上资料

16、《C 语言程序设计》网上资料