1. 绪论
2. 操作系统概述

什么是操作系统

按照笔者的理解，操作系统是：

\*为应用程序提供环境

\*管理硬件驱动程序

\*以文件的形式管理磁盘数据

\*具有输入输出功能

的一种软件。它是一种软件，和一般的应用软件相比没有本质上的区别，只不过比应用软件特殊一些。我们平时用的应用软件都有很明确的用途，比如word，你可以用来写论文，比如QQ，你可以用来聊天斗图，比如各种游戏，你可以用来愉悦身心获得快乐，

但操作系统好像没有什么明显的用途，Windows操作系统我们天天都在用，可是我们用它来干嘛呢？想象一下，假如你的电脑没装操作系统，刚才提到的word、QQ还有游戏你能安装吗？显然不能，所以为应用程序提供环境就是操作系统的用途之一，我们在使用那些应用软件的同时，也是在使用操作系统。

驱动程序是一种控制硬件设备，使它能正常工作的程序。驱动程序的英文是Driver，这更有助于理解：硬件设备就好比一辆车，要让它跑起来就必须有一位司机，这位“司机”就是驱动程序。和其他程序一样，驱动程序要运行就必然需要分配内存，当它不再使用时就需要回收内存。由于驱动程序要对硬件进行各种操作，因此需要获得较高的权限，另外大多数驱动程序都会用到中断。分配与回收内存、授予权限以及中断向量表的维护等涉及到驱动管理的工作都要由操作系统来完成。

一般电脑中的数据都是保存在硬盘里的，硬盘读写的单位是扇区（1扇区=512字节），CPU读写硬盘只能一扇区一扇区的读写。然而应用程序读取或修改数据是以文件形式操作的，比如在C语言程序中先用fopen函数打开一个文件，再用read或write函数读写文件中的数据，在这个过程中只需指定路径和文件名即可，完全不必考虑文件的各个部分保存在哪些扇区里。这种以文件形式读写数据的功能正是操作系统提供的，操作系统有一个非常重要的子系统，叫“文件系统”，它在硬盘的各个分区上建立庞大而复杂的数据结构，从而在茫茫多的扇区之上抽象出一个个文件和目录，最后把数据以文件的形式呈现给用户。

操作系统，顾名思义，用来操作电脑的系统。既然让用户操作，就必须有输入输出系统。应用程序在开发过程中使用的各种输入输出函数并不是本身自带输入输出功能，而是通过操作系统的输入输出系统来获取用户的输入或者往屏幕上输出。所以归根结底，输入输出过程中关键的操作还是由操作系统完成的。

操作系统的启动过程

在计算机的主板上有一块CMOS（互补金属氧化物半导体）芯片，这块芯片里储存着BIOS程序，在用户按下开机键后，CPU开始执行BIOS程序的第一条指令。BIOS程序主要负责检测内存大小，扫描连在主板上的各个硬件设备并检查它们是否有故障。然后按照用户指定的顺序依次检查各存储设备上的第一个扇区的后2个字节，若这2个字节是16进制的55和aa（MBR标志）则将该设备的第一个扇区读到内存的7c00h处，然后跳转到7c00h开始执行该扇区中一段被称为主引导记录（MBR）的代码。主引导记录遍历所在存储设备上的所有分区，找到被标记为“活动”的分区后便启动存储在这个分区里的操作系统。

毕业设计的各个组成部分

内核： 内核包含了系统的初始化代码，系统调用以及许多重要的系统变量。其中初始化代码是整个操作系统中最先运行但只运行一次的部分，它主要负责打开A20地址线，进入保护模式，创建页表和内存位图，初始化中断系统，加载基本模块以及创建外壳进程。系统调用是一种特殊的中断程序，在我的毕业设计中只有一个，它主要用于内存和页表的管理以及进程的创建和销毁。系统调用随内核被装入后便常驻内存。

硬盘驱动模块： 硬盘驱动主要负责提供底层的读写硬盘功能，该功能最重要的用途是确保文件系统能正常工作，所以硬盘驱动模块在系统初始化时就要被加载。

文件系统模块： 该模块提供所有和文件系统以及分区相关的功能，包括创建文件，删除文件，读文件，写文件，创建分区，格式化分区和删除分区。文件系统模块在系统初始化的过程中被加载。

终端模块： 该模块向应用程序提供除文件操作外的基本功能，在系统初始化的过程中被加载。

外壳： 外壳应该算是一个应用程序，它在我毕业设计中的作用就如同bash在Linux系统中的作用一样，是操作系统和用户交互的“前台”，它主要负责接收用户输入的命令并响应。外壳是系统中第一个进程，在系统初始化的过程中被创建，随后在系统中一直存在直到关机。

1. 搭建开发环境

NASM汇编器

我的毕业设计使用汇编语言编写，所以就需要一个汇编器。NASM是一个为可移植性与模块化而设计的一个80x86的汇编器。它支持相当多的目标文件格式，包括Linux和'NetBSD/FreeBSD','a.out','ELF','COFF',微软16位的'OBJ'和'Win32'。它还可以输出纯二进制文件。它的语法设计得相当的简洁易懂，和Intel语法相似但更简单。它支持'Pentium','P6','MMX','3DNow!','SSE' and 'SSE2'指令集。

NASM的安装过程非常简单，访问NASM的官方网站<https://www.nasm.us>并点击页面上方的DOWNLOAD，然后选择一个版本，笔者安装的版本是2.13.03。在接下来的页面中带有doc字样的是NASM的文档，nasm-2.13.03.XXX是软件源码，而那几个文件夹里的是各操作系统下的NASM安装程序或免安装压缩包。在Linux下编译安装的方法如下：

wget https://www.nasm.us/pub/nasm/releasebuilds/2.13.03/nasm-2.13.03.tar.xz #下载

tar -xJvf nasm-2.13.03.tar.xz #解压缩

cd nasm-2.13.03/ #进入源码目录

./configure #配置安装选项

笔者配置的是默认安装选项，关于详细的安装选项可以使用./configure –-help来查看。在这一步中可能出现的问题是no acceptable C compiler found in $PATH，这个错误是没有安装C编译器导致的，NASM的源码是C语言，所以编译它就需要一个C编译器。

安装gcc：

apt-get install gcc #debian及其衍生版本的命令

yum install gcc #Red Hat及其衍生版本的命令

安装完gcc之后再次./configure，在输出信息的后几行只要没有error字样就意味着配置成功。

然后编译源码：

make

如果make命令未找到，则需要先安装make软件：

apt-get install make #debian及其衍生版本的命令

yum install make #Red Hat及其衍生版本的命令

在输出信息中只要没有error字样就意味着编译成功。

然后安装：

make install

执行完命令后NASM的可执行文件就被安装到了prefix/bin中，prefix在./configure中设置，默认为/usr/local。如果prefix/bin不在环境变量PATH中则需要把它添加进去。安装好之后输入nasm –v能显示出版本信息则安装无误。

NASM的安装方法不止这一种，但其他安装方式都是傻瓜式的，只要注意修改环境变量PATH即可，故在此不做赘述。如果读者是很新很新的新手，实在不会安装，可以给笔者发邮件求助：1017201217@qq.com。另外，下文中出现的16进制数都用NASM的语法书写。

bochs模拟器

bochs是一个高度可移植的开源x86 PC模拟器，用C++编写，运行在大多数流行的平台上。它包括了英特尔x86 CPU、通用I/O设备和自定义BIOS的模拟。bochs可以用来模拟多种不同的x86 CPU，从早期的386到近期英特尔和AMD的x86\_64处理器。它就像一个虚拟机，不过对我们来说最重要的还要数它的调试功能，你可以让这台虚拟机每执行一条机器指令就暂停一次，然后查看各寄存器、内存的值。对于我们开发出的操作系统来说是一个绝佳的调试环境。

笔者不建议Linux用户使用yum或者apt-get安装bochs，那样安装的版本可能比较旧而且功能不全。编译安装的方法如下：

访问bochs的官方主页<http://bochs.sourceforge.net>（这可能需要科学上网，手段请自备），点击页面左边的See All Releases打开下载页面，点击bochs文件夹然后选一个版本，笔者安装的版本是2.6.9，在接下来的页面中点击bochs-2.6.9.tar.gz下载。

tar -xzvf bochs-2.6.9.tar.gz #下载完之后解压缩

cd bochs-2.6.9/ #进入源码目录

接下来就该配置安装选项了，这一步非常关键。全部的选项及其说明可以在官方文档中找到：点击官网主页左侧Documentation下边的User Guide，然后打开3.4.2. Configure Options。这里只介绍必要的选项：

--enable-debugger 启用bochs的调试功能

--enable-x86-64 添加x86\_64指令集支持

其他功能方面的选项要么不需要启用要么默认启用。在配置过程中如果出现了错误和警告不必气馁，那都是bochs编译所需的软件未安装导致的，只要安装了缺少的软件再重新配置即可。

缺少C++编译器：

apt-get install g++ #debian及其衍生版本

yum install gcc-c++ #Red Hat及其衍生版本

缺少readline库：

apt-get install libreadline-dev #debian及其衍生版本

yum install readline-devel #Red Hat及其衍生版本

在输出信息中只要没有error字样就意味着配置成功，接下来就该编译并安装了。

make

make install

如果在配置时修改了安装路径prefix的话安装之后需要将prefix/bin添加到环境变量PATH中。安装好之后输入bochs --help能显示出帮助信息则安装无误。

编译安装bochs略为复杂，可能遇到的问题也比安装NASM多，如果遇到了无法解决的问题可以下载安装程序或免安装压缩包安装。只需要在安装完之后需要修改环境变量PATH即可，在此不做赘述。

jdk

为了方便在硬盘镜像文件上创建分区以及写入文件，笔者用Java做了一个磁盘镜像编辑器。虽然它不属于操作系统的一部分，但是没有它就无法把操作系统写入虚拟机硬盘。因此需要安装jdk。安装jdk非常简单，在Linux中用yum或apt-get工具可以“一键安装”，在Windows中下载并运行安装程序，然后配置好环境变量PATH和CLASSPATH就行了，在此不做赘述。

文本编辑器

一个好用的文本编辑器通常都有显示行号、语法高亮、自动缩进、自动补全等功能，它让我们在写代码时事半功倍。不过文本编辑器的选择却因人而异，读者们安装一个自己用着顺手的就行。

编译项目

搭建好开发环境后就可以编译项目了，进入os目录然后执行以下命令：

nasm src/boot.asm -o boot

nasm src/kernel.asm -I src/ -o kernel

nasm src/drv/HD\_drv.asm -I src/ -o 硬盘驱动

nasm src/filesystem.asm -I src/ -o 文件系统

nasm src/Terminal.asm -I src/ -o 终端

nasm src/sh.asm -I src/ -o 外壳

nasm src/a1.asm -I src/ -o a1

然后需要创建一个硬盘镜像文件，笔者使用的是bximage，一个bochs附带的小工具。输入bximage回车后按照屏幕提示一步步操作就行，不过注意镜像类型要选flat。以下是笔者的屏幕输出：

========================================================================

bximage

Disk Image Creation / Conversion / Resize and Commit Tool for Bochs

$Id: bximage.cc 13069 2017-02-12 16:51:52Z vruppert $

========================================================================

1. Create new floppy or hard disk image

2. Convert hard disk image to other format (mode)

3. Resize hard disk image

4. Commit 'undoable' redolog to base image

5. Disk image info

0. Quit

Please choose one [0] 1↵

Create image

Do you want to create a floppy disk image or a hard disk image?

Please type hd or fd. [hd] ↵

What kind of image should I create?

Please type flat, sparse, growing, vpc or vmware4. [flat] ↵

Enter the hard disk size in megabytes, between 10 and 8257535

[10] ↵

What should be the name of the image?

[c.img] disk.img↵

Creating hard disk image 'disk.img' with CHS=20/16/63

The following line should appear in your bochsrc:

ata0-master: type=disk, path="disk.img", mode=flat

注意倒数第3行的“CHS=20/16/63”，待会创建bochs虚拟机的时候会用到这3个值。

有了“硬盘”之后就要把操作系统“安装”上去。首先写入boot和kernel，这2个东西是直接写入扇区的，笔者使用的是Linux中的dd工具，Windows用户可以去<http://www.chrysocome.net/dd>下载dd for windows。

dd if=boot of=disk.img conv=notrunc

dd if=kernel of=disk.img seek=1 conv=notrunc bs=512

剩下的就需要保存到硬盘镜像的文件系统中了，镜像所有和分区及文件系统相关的操作都由磁盘镜像编辑器完成。其用法如下：

* java Nfs 要编辑的镜像文件名

运行之后是一个命令行界面，“>”号前边的是当前目录，目前只支持6条命令：

* mkfs id 起始扇区号 分区容量 卷标 创建一个分区

其中id的范围是1-4，起始扇区号必须是8的倍数，分区容量的单位是簇（8扇区）。

* ls 查看当前目录下的文件
* cd 路径 改变当前路径
* mkdir 目录名 创建一个目录
* emerge 文件名 导入文件到当前目录
* exit 退出程序

先编译好磁盘镜像编辑器，然后创建一个分区：

javac tools/Nfs.java

java –classpath tools/ Nfs disk.img

/>mkfs 1 72 200 p1

进入该分区，导入2个文件：

/>cd p1

/p1/>emerge 外壳

/p1/>emerge a1

创建名为“模块”的目录然后进入，再导入剩下的3个文件：

/p1/>mkdir 模块

/p1/>cd 模块

/p1/模块/>emerge 硬盘驱动

/p1/模块/>emerge 文件系统

/p1/模块/>emerge 终端

/p1/模块/>exit

至此，我们的操作系统就安装到了disk.img上。不过以后每修改一次代码就得重新生成disk.img，因此为了方便起见，笔者建议把这些命令保存成一个脚本，这样一来改完代码后只要运行一次脚本即可：

#!/bin/bash

nasm src/boot.asm -o boot

nasm src/kernel.asm -I src/ -o kernel

nasm src/drv/HD\_drv.asm -I src/ -o 硬盘驱动

nasm src/filesystem.asm -I src/ -o 文件系统

nasm src/Terminal.asm -I src/ -o 终端

nasm src/sh.asm -I src/ -o 外壳

nasm src/a1.asm -I src/ -o a1

echo 1 > ipt

echo hd >> ipt

echo flat >> ipt

echo 10 >> ipt

echo disk.img >> ipt

echo yes >> ipt

bximage < ipt

dd if=boot of=disk.img conv=notrunc

dd if=kernel of=disk.img seek=1 conv=notrunc bs=512

echo mkfs 1 72 200 p1 > ipt

echo cd p1 >> ipt

echo emerge 外壳 >> ipt

echo emerge a1 >> ipt

echo mkdir 模块 >> ipt

echo cd 模块 >> ipt

echo emerge 硬盘驱动 >> ipt

echo emerge 文件系统 >> ipt

echo emerge 终端 >> ipt

echo exit >> ipt

java -classpath tools/ Nfs disk.img < ipt

rm -f boot kernel ipt 硬盘驱动 文件系统 终端 外壳 a1

注意Nfs.java如果不改的话就不用重新编译，所以在脚本里并没有javac tools/Nfs.java这条命令。

创建bochs虚拟机

有了“启动盘”后，只要创建一台虚拟机就能让操作系统运行起来。创建bochs虚拟机可以用图形界面也可以用命令行，二者学会了一个也就掌握了另一个。打开bochs然后修改虚拟机的软硬件配置，笔者能力有限，不能把bochs虚拟机的所有配置项都讲的面面俱到，这里只简单介绍几个有必要修改的配置项：

Logfile：

日志配置。它有3个子项，其中Log filename是虚拟机运行日志的文件名，该文件记录了虚拟机运行过程中产生的信息，如果该项留空那么这些信息就会输出到调试命令行中，妨碍我们观察调试输出。Debugger Log filename是调试日志的文件名，如果指定此项那么每次调试的屏幕输出都会记录在指定文件中，这在有的时候还是很有用的。不过笔者发现用Bochs-2.6.9.exe安装的Windows版本中这一项不可用（Linux下编译安装就可用）。

Memory：

内存、BIOS和VGA BIOS配置。Standard Options子项中的内存大小设置为32M或更高，其他保持默认。

Disk & Boot：

磁盘与启动配置。这一项很关键，它有2个软盘子项、4个ATA通道共12个子项，外加一个启动子项，2个软盘子项不用管，ATA通道0和ATA通道1默认是打开的，修改First HD/CD on channel 0这一项来指定虚拟机ATA通道0的主硬盘。类型选择disk，文件名就填上一步生成的镜像文件名disk.img，镜像模式要选择flat，在bximage那一步的CHS三个值分别对应接下来要填的cylinders、heads和sectors per track，剩下的都保持默认。然后修改Boot Options子项，它的前3个子项是启动顺序，把第一项改成disk，从硬盘启动，不过当虚拟机中有多个硬盘时好像并不能指定从哪个硬盘启动。设置了第一项后边两项就不用管了。

修改完成后记得保存配置，否则下一次运行前还得再改一遍。建议文件的扩展名写成.bxrc，因为这是bochs能识别的扩展名。保存下来的其实是一个文本文件，因此当你配置熟练之后可以更加高效快捷地用文本编辑器修改配置。

启动与调试

使用如下命令来启动一台虚拟机：

bochs –q –f 虚拟机配置文件名 #Linux

bochsdbg –q –f虚拟机配置文件名 #Windows

其中-q表示快速启动，省去了载入配置文件后修改的步骤。-f表示指定配置文件名。

启动之后有按钮的那个窗口是虚拟机的显示器，在调试控制台中可以看到类似这样的输出：

(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b ; ea5be000f0

<bochs:1>

其中[0x0000fffffff0]代表下一条指令的物理地址，f000:fff0代表虚拟地址，jmpf 0xf000:e05b是反汇编后的机器指令，它和NASM的语法稍有不同，分号后边的ea5be000f0就是机器指令，<bochs:1>中的1代表当前这是本次运行的第一条命令。我们在它后边输入各种调试命令来进行调试：

r 查看16个通用寄存器、IP寄存器和标志寄存器的值

trace-reg on/off 打开/关闭每执行一次指令就输出上述各寄存器的值

s 单步步入，遇到call,int,loop,rep和外中断时进入

n 单步步过，遇到call,int,loop,rep和外中断时不进入

b 地址 在物理地址上下断点

lb 地址 在线性地址上下断点

vb 段:偏移 在虚拟地址上下断点

c 一直运行，直到遇到断点或严重到使机器重启的异常

blist 查看下过的断点

d 断点号 删除一个断点

x /nuf 地址 显示线性地址的内容

xp /nuf 地址 显示物理地址的内容，这2条命令中nuf的含义如下：

n 显示的单元数。

u 每个显示单元的大小，u可以是下列之一：

b 字节

h 字

w 双字

f 显示格式，f可以是下列之一：

x 按照十六进制显示

d 十进制显示

u 按照无符号十进制显示

o 按照八进制显示

t 按照二进制显示

c 按照字符显示

n、f、u是可选参数，如果不指定，则u默认是w，f默认是x。如果前面使用过x或者xp命令，会按照上一次的x或者xp命令所使用的值。n默认为1。addr 也是一个可选参数，如果不指定，addr是0，如过前面使用过x或者xp命令，指定了n=i，则再次执行时n默认为i+u。

setpmem 地址 长度 值 设置物理内存某地址的内容，长度<5且不能等于3

set 寄存器=值 设置寄存器的值，只能设置16个通用寄存器的值。

不过不能设置当前模式没有的寄存器，比如在实模式下就不能设置r15的值。

print-stack num 显示堆栈前num\*8个字节，num可以省略，默认是16

sreg 显示各个段寄存器的内容

creg 显示各个控制寄存器的内容

info gdt/ldt/idt [n1] [n2] 显示gdt/ldt/idt中从第n1到第n2个描述符，若只写n1则n2=n1，若n1,n2都不写则显示全部。

info tss 显示TSS

q 结束调试，退出bochs

在这些命令中出现的数字，值也好地址也罢，都可以使用16进制数，使用16进制数需要在前边加上“0x”。如果在c的时候程序陷入了死循环就按Ctrl+C强制暂停。什么都不输入直接按回车表示重复执行上一条命令。

1. 内核

内核启动

一般来讲，内核以一个文件的形式保存在操作系统所在的那个分区里。文件系统的数据结构是非常复杂的，从一个分区中读取某个文件的内容需要反复的读取扇区、分析数据，操作如此复杂的代码肯定不会太短，不可能放入长度只有400多字节的MBR中。MBR找到活动分区后，将位于分区最前端的一段被称为分区引导记录（PBR）的代码读到内存中并执行，由PBR来读取内核文件，将内核文件的内容全部读到内存后再跳转到内核的第一条指令开始执行。

然而为了简单起见，笔者没有编写PBR，也没有把内核以文件的形式保存，而是把内核保存在了第2到第66个扇区上（内核大小为8200h字节），这样一来MBR只需简简单单的把这连续的65个扇区读到内存中，就一步到位地加载了内核。代码如下：

%define SEC\_NUM 65;读取扇区的数量

mov dx,1f2h

mov al,SEC\_NUM

out dx,al

inc dx

mov al,1

out dx,al

inc dx

xor al,al

out dx,al

inc dx

out dx,al

inc dx

mov al,0e0h

out dx,al

inc dx

mov al,20h

out dx,al

mov cx,SEC\_NUM\*512/4

xor di,di

mov ax,7e0h

mov es,ax

sg:

in al,dx

test al,80h

jnz sg

test al,8

jz sg

mov dx,1f0h

cld

rep insd

;-------------------------

jmp 7e00h

代码解释：读写硬盘需要向硬盘控制器发送以下信息：要操作哪个硬盘、要读写几个连续的扇区、起始扇区的编号（长28位，从0开始）是多少以及要进行什么操作（读还是写）。向硬盘控制器发送命令是通过端口进行的，这段代码先向1f2h号端口发送了扇区数量65，然后从低到高依次向1f3h～1f5h号端口发送了扇区号的低24位，然后又向1f6h号端口发送了一个字节，在这个字节中低4位表示扇区号的最高4位，第5位若是0表示操作的是主硬盘，若是1则表示操作的是从硬盘，高3位都是固定值1，最后向1f7h号端口发送命令，20h表示读取，若要写入则发送30h。发送完命令之后通过读取1f7h号端口来获取当前硬盘的状态，在读出的字节中最高位是BSY位，如果此位是1表示当前硬盘正忙，且其他7位无效，第4位是DRQ位，如果此位是1表示硬盘已准备好传输数据，可以读取了，第7位是DRDY位，如果此位是1表示硬盘已就绪，可以向硬盘写入数据了。反复读取1f7h号端口并检查这几个位，当它们的值合适的时候就可以传输数据了。不管是读取还是写入，数据都是通过1f0h号端口传输的，若是读取就从1f0h号端口中读出扇区数量×512个字节，若是写入则向1f0h号端口写入扇区数量×512个字节。这段代码把内核所在的那65个扇区读到内存的7e00h处之后便跳转到了7e00h，让内核开始运行。

进入保护模式

系统初始化代码首先要做的就是进入保护模式。在进入保护模式之前需要加载全局描述符表（GDT），中断描述符表（IDT）可以等到进入保护模式之后再加载，不过出于稳定性考虑在进入之前就把它加载了。GDT和IDT位于内核的末尾，

+0

|  |
| --- |
| 系统初始化代码  ≈+228fh |
| ≈+5ceeh |
| 中断预引导/引导代码  +68a3h |
| 系统调用  +7067h |
| 时钟中断  +7120h |
| 系统变量区  +7154h |
| TSS  +71c2h |
| IDT  +81c8h |
| GDT  +8200h |
| 内核分布图 |

其中GDT里有5项，从上到下依次是空白项、代码段、数据段、任务状态段（TSS）和局部描述符段（LDT）。其中代码段的属性为32位非一致可读，段基址和段界限分别是0和0fffffh，界限粒度是字节。数据段的属性为可读可写，D位为1，段基址和段界限也是0和0fffffh，界限粒度是4K。GDT中所有段的特权级（DPL）都是0。在进入保护模式之前还得把A20地址线打开，在实模式下如果访问不小于100000h的地址那么该地址就会“回滚”到100000h以内。例如ds=0ff00h，现访问ds:1100h，访问到的地址本该是ds<<4+1100h=100100h，然而超出20位的那个1会被强制转成0，真正访问到的地址就变成了100h。只有打开了A20地址线这一位才不会被强制清零。据说打开A20地址线的方法不止一种，不过笔者只知道一种。

做完上述这些操作后就可以进入保护模式了，将cr0寄存器的最低位（PE位）置1，然后进行一次远跳转。这个跳转非常重要，在PE位置1后CPU已经进入保护模式了，可是这个时候cs寄存器里还是原来在实模式下的值，这条跳转指令的作用就是把代码段的选择符装入cs。这条指令还有一个小细节：它是一条16位指令，而目标地址却是32位的，所以要在目标地址前加一个dword指定宽度，否则如果便宜地址超过16位的话就会在编译时被截断，只保留低16位。进入保护模式的代码如下：

org 7e00h

[BITS 16]

lgdt [GDTPtr]

cli

lidt [IDTPtr]

in al,92h

or al,2

out 92h,al

mov eax,cr0

or eax,1

mov cr0,eax

jmp dword Code-GDT:SYS\_INIT

启用分页并进入64位模式

分页机制不仅是CPU进入64位模式的必要条件，而且对于操作系统中的进程来说非常重要。先看一个小实验：

在Windows下用C语言编写如下程序：

#include <stdio.h>

#include <stdlib.h>

int var;

int main()

{

int\* a = &var;

printf("Address=0x%x, input a value:", a);

scanf("%d", a);

system("pause");

printf("Address=0x%x, value=%d\n", a, var);

return 0;

}

编译，然后同时打开2个命令行窗口运行它。可以看到2个var变量的地址都是0x407070。在第一个窗口中输入一个值，比如10，然后它就会暂停。这时在第二个窗口中输入一个不同的值，比如25，然后分别按任意键继续这2个窗口中的程序，第一个窗口中的输出是Address=0x407070, value=10，第二个窗口中的输出是Address=0x407070, value=25。可以看到虽然2个var变量的地址一样，但是它们的值却互不影响。这可不是因为内存中有2块区域的地址都是0x407070，而是Windows系统通过修改或切换页表改变了线性地址到物理地址的映射关系。这样既管理了进程的内存，又相互隔离了各个进程。

内核在进入保护模式后将栈底设置为7c00h，然后构建一套页表。这套页表把线性地址映射到相等的物理地址中，页面大小为2M，属性为可写，系统级权限，4级页表的基址（cr3）是PageDirBase，它是一个宏，在sys.inc.asm中被定义为100000h。不过在这里只创建前2M的映射，等到64位模式中检测内存大小的时候再把它补完。代码如下：

SYS\_INIT:

[BITS 32]

mov ax,Data-GDT

mov ds,ax

mov ss,ax

mov es,ax

mov esp,7c00h

mov esi,PageDirBase

mov edi,PageDirBase+1000h

mov ebx,PageDirBase+2000h

mov dword[esi],PageDirBase+1000h+11b

mov dword[esi+4],0

mov dword[edi],PageDirBase+2000h+11b

mov dword[edi+4],0

mov dword[ebx],10000011b

mov dword[ebx+4],0

mov cr3,esi

紧接着开启PAE（物理地址扩展）并把EFER寄存器的LME位置1，这些都是进入64位模式的必要条件，不过做完这些操作后CPU还没有进入64位模式，因为分页机制还没开启。在开启分页机制前需要先修改GDT中代码段的属性，将其D位清零并把L位置1。最后把cr0寄存器的PG位置1，开启分页机制。和之前一样，用一条远跳转指令重新加载cs寄存器，否则CPU将工作在兼容模式下。代码如下：

mov eax,cr4

or eax,100000b

mov cr4,eax

mov ecx,IA32\_EFER

rdmsr

or eax,100h

wrmsr

mov byte[Code+6],2fh

mov eax,cr0

or eax,80000000h

mov cr0,eax

jmp Code-GDT:x64

构建页表

进入64位模式后首先要做的就是补全页表。这套页表映射的原则是线性地址映射到相等的物理地址，有多大内存就映射多大的地址空间，因此在构建时必须知道内存的大小。在x86架构中如果读取的物理地址大于内存容量的话CPU并不会产生异常，而是会读取到0ffh，所以要判断一个物理地址是否大于内存容量就可以先向它写入一个非0ffh的值再读取，若读到的值和写入的值相等则不大于。因此笔者采用的方式是一边构建页表扩大映射区间，一边通过写入再读取的方式检测内存大小。每当一个新页面建立了映射，就向该页面的最后一个字节写入0cch，紧接着再读取这个字节并检测它的值，如果是0cch则继续构建，不是则意味着内存已映射完。64位模式下的页表你可以视为一棵最多4层深的512叉树，笔者将这棵512叉树以先根顺序存储的方式来创建，因为只有这样才能最大限度的减少内存的浪费。构建页表的代码如下：

bits 64

x64:

xor rax,rax

nextPL4:

mov rdx,rdi

or dl,11b

mov qword[rsi],rdx

add rsi,8

nextPDPTE:

mov rdx,rbx

or dl,11b

mov qword[rdi],rdx

add rdi,8

mov rcx,200h

nextPDE:

mov rdx,rax

or dl,10000011b

mov qword[rbx],rdx

mov byte[rax+200000h-1],0cch

cmp byte[rax+200000h-1],0cch

jne mmap

add rbx,8

add rax,200000h

loop nextPDE

test rdi,0fffh

jnz nextPDPTE

add rbx,1000h

add rdi,200000h

cmp rsi,PageDirBase+1000h

jb nextPL4

;-------------------------

mmap:

mov qword[rbx],0

mov [memsize],rax

为了便于理解，笔者用一段逻辑相同的伪代码来解释它：

rax = 0;

do

{

[rsi] = rdi | 11b;

rsi += 8;

do

{

[rdi] = rbx | 11b;

rdi += 8;

for(rcx = 200h; rcx > 0; rcx--)

{

[rbx] = rax | 10000011b;

byte[rax+200000h-1] = 0cch;

if(byte[rax+200000h-1] != 0cch)

goto mmap;

rbx += 8;

rax += 200000h;

}

}while(rdi%1000h != 0);

rbx += 1000h;

rdi += 200000h;

}while(rsi < PageDirBase+1000h);

mmap:

qword[rbx] = 0;

memsize = rax;

在这段代码中rax保存已映射的最高地址，rsi,rdi和rbx分别保存4级页表项（PML4E）、3级页表项（PDPTE）和2级页表项（PDE）的地址，memsize是一个系统变量，用来记录内存容量。这段程序中嵌套了3层循环，最外层循环遍历4级页表，每次遍历先填写页表项，然后构建对应的3级页表。注意构建一套完整的3级页表将会占用1个4K页（用来存储3级页表）+200h个4K页（用来存储2级页表），所以需要把rdi+200000h，rbx+1000h，以此让rdi等于下一个3级页表的地址，rbx等于下一个2级页表的地址。中间的那一层循环遍历3级页表，每次遍历先填写3级页表项，然后构建对应的2级页表。最内层循环将rax以2M为递增值依次填入表中，每填写一个也就建立了此地址的映射，然后用前面提到的方法检测该地址是否超出内存容量，若超出则跳转到mmap，否则继续构建。在映射完所有内存后，把被检测到超出容量的最后那个2级页表项清空，然后填写memsize。由于清空2级页表项之后没有检查3级页表项，因此万一主板上插的内存容量达到了字节的整数倍，在页表上就会多浪费一个4K页面，不过既然内存都这么大了，谁还会去计较那区区4KB呢？

创建内存位图

操作系统将空闲的内存视为一个内存池，笔者考虑了几方面因素后决定以4K页面为池中内存的单位，当进程需要内存时系统就从内存池中取出一个页面分配给它，当系统中有页面不再使用时系统就会将它放回内存池中。这种“取”和“放”的操作都是借助内存位图来完成的。在内存位图中一个二进制位表示一个页面，第一个位表示第一个页面，第二个位表示第二个页面……以此类推，每个位若是0表示对应的页面空闲，若是1则表示对应的页面已被占用。创建内存位图的代码如下：

（紧接着上一段代码）

test rbx,0fffh

jz j0

add rbx,1000h

and rbx,~0fffh

j0: mov [memmapaddr],rbx

Build\_mmap

其中Build\_mmap是一个宏，它在sys.inc.asm中定义：

%macro Build\_mmap 0

mov r8,rax

shr r8,12

shr rax,15 ;rax=内存位图字节数

mov rdx,rax

add rdx,rbx

test rdx,0fffh

jz Bd\_LB0

add rdx,1000h

Bd\_LB0: shr rdx,12 ;rdx=已占用页数

sub r8,rdx ;r8=空闲页数

mov [freemem],r8

mov [froffset],rdx

mov rcx,rdx

and rcx,111b

shr rdx,3

Bd\_LP0: mov byte[rbx+rax],0

dec rax

cmp rax,rdx

ja Bd\_LP0

mov ch,0ffh

shl ch,cl

not ch

mov byte[rax+rbx],ch

Bd\_LP1: dec rax

mov byte[rax+rbx],0ffh

cmp rax,0

jne Bd\_LP1

%endmacro

代码解释：一开始rax=内存大小，rbx=最后一个页表项的地址+8，memmapaddr、freemem和froffset是3个系统变量，分别用来记录内存位图的起始地址、空闲页面数和已占用的页面数。内存位图被笔者安排在了页表的后边，所以将rbx对1000h向上取整，rdx就是内存位图的起始地址。接下来计算内存位图的字节数rax、已被占用的页面数rbx和空闲页面数r8，然后填写变量freemem和froffset，最后根据rdx置1或清零位图中的各个位。

在创建完内存位图后，系统初始化代码加载了TR寄存器和LDTR寄存器。在TSS中只有rsp0字段会被用到，它的值是100000h，至于为什么是这个值在后面的章节中会提到。在加载LDTR时系统中还没有LDT，但这并不会产生异常，只有在访问LDT中的段时才会。关于LDT的使用在后面的章节中会提到。

紧接着下一步就是初始化8259A可编程中断控制器，在这一步中主片被设置为采用特殊全嵌套，非自动结束方式，从片被设置为采用一般全嵌套，非自动结束方式。在发送完主片从片一共8个初始化控制字（ICW）之后又分别向主片和从片发送了操作控制字1（OCW1）来屏蔽除了时钟中断外的所有外中断。

接下来的操作是清零中断向量表和进程表。中断向量表位于0～3bffh，它和CPU在实模式下的那个中断向量表没有任何关系，只是名字一样罢了。进程表位于10000h～90000h，关于这两张表的使用会在后面的章节中提到。

系统初始化过程中剩下的步骤分别是注册硬盘驱动、文件系统、终端三个模块，创建外壳进程并引用这3个模块，把中断标志位（IF）位置1，同时开始执行外壳进程。这些步骤中具体的操作会在以后的章节中讲解。

中断系统

x86架构中的可屏蔽外中断的中断号可以用8259A分配，在我的毕业设计中这些外中断被分配到了20h～30h。在全部的256个中断中0f1h～0ffh号中断系统保留未使用，0f0h号中断用作系统调用，它的中断服务程序在内核里，20h号中断（时钟中断）的中断服务程序也在内核里。这样一来，还有239个中断的服务程序不在内核中，它们是按需被装载到内存中的，而实现这种按需装载的灵活机制就使得它们的运行过程略显复杂：在内存的0～3bffh中保存着系统的中断向量表，表中有240个组，依次对应前240个中断（对应时钟中断的那个组只是凑个数），每组64字节分为8项。现规定每个中断服务程序最多32KB（8占个页面），当中断服务程序被装载到内存是按顺序把装入页面的物理地址填入对应的项中。IDT中各中断门的偏移地址指向各自中断的预引导代码，预引导代码只有两条指令：把中断向量表中对应组的首项压栈，然后跳转到统一的中断引导代码。中断引导代码首先将栈顶元素出栈并保存到一个临时变量中，然后切换堆栈和页表，再调用中断服务程序。如果中断服务程序要跨页执行则自行查中断向量表跳转，中断服务程序执行完后用ret指令返回到被调用的地方，然后切换回原来的页表和堆栈，最后用iretq指令返回。中断引导代码如下：

\_ep: dq 0 ;存放中断的首页物理地址

\_rax: dq 0 ;用于临时保存rax

\_boot:

pop qword[\_ep]

mov [int\_stk\_btm],rsp

mov rsp,int\_stk\_btm

mov [\_rax],rax

mov rax,PageDirBase

mov cr3,rax

mov rax,[\_rax]

call qword[\_ep]

mov [\_rax],rax

mov eax,[cPid]

mov rax,[protab + eax \* 8]

mov cr3,rax

mov rax,[\_rax]

mov rsp,[int\_stk\_btm]

iretq

系统调用

系统调用本质上是一个中断，不过它既不是靠CPU产生异常触发也不是靠硬件设备触发，而是靠int指令触发。操作系统需要为进程提供一些底层的功能，而设计系统调用就是为了给进程提供一个方便的调用这些功能的接口。在毕业设计的唯一一个系统调用中有8个功能，调用的时候给rax寄存器赋一个0～7之间的值来指示要调用的是哪一个功能，调用所有功能都用寄存器传递参数，如果有返回值的话用rax保存返回值。代码如下：

mov [int\_stk\_btm],rsp

mov rsp,int\_stk\_btm

cmp rax,fvct\_len / 8

jb etf

mov rsp,[int\_stk\_btm]

iretq

etf:

shl rax,3

jmp [fvct + rax]

fvct: dq malloc\_4K

dq free

dq ysh

dq unysh

dq line\_phy

dq ctPro

dq chkPro

dq dsPro

fvct\_len equ $-fvct

【各个功能的代码】

代码解释：进入系统调用后首先切换堆栈，然后检查rax（功能码）是否超范围，fvct是一张记录着各个功能代码首地址的列表，若rax不超范围则根据rax从这张表中查找要调用功能的首地址，然后跳转到功能代码。

【待续】系统调用的8个功能

时钟中断

在我的毕业设计中时钟中断是用来在进程的时间片用完之后切换进程的，这部分代码放到后面的章节中讲解。

1. 进程与可执行文件

进程概述

所谓进程，就是已经开始运行的程序。在硬盘中保存着很多应用程序，这些程序在没有运行的时候只是一些可执行文件，操作系统把用户运行应用程序的操作视为一种任务，系统每接到一项任务就会把程序的可执行文件装入内存，程序被装入内存中就变成了进程。现代操作系统几乎无一例外都是多任务操作系统，多任务的含义是可以有条不紊地“同时”执行多个进程，要实现这种特性就必须做到至少两点：第一，进程之间必须相互隔离，互不影响，这在上一章的那个小实验中有所体现。第二，各个进程必须分时轮流执行。我们知道，CPU在某一时刻只能执行一段代码（不考虑多核CPU和超线程技术），那么当系统中有多个进程的时候，先让CPU执行第一个进程，执行一段时间后暂停，再让CPU执行第二个进程，执行一段时间后暂停……最后一个进程暂停执行后再让CPU从第一个进程开始轮流执行，如此循环往复，每次执行的时间大约是【待查阅】，这样在用户看来各个进程就是同时执行。下面将为大家讲解进程是如何隔离以及如何切换的。

进程的线性地址空间

操作系统在创建进程的过程中首先创建一套页表，以此来建立一部分线性地址到物理地址的映射，然后把要执行程序的可执行文件装入已映射的线性地址中，就这样一个萝卜一个坑，每个程序都处在各自页表所映射的线性地址空间中，不会和其他的进程有交集，并且为了防止应用程序私自切换页表，再给它设置一个最低的特权级（ring3），这样就相互隔离了各个进程。

在我的毕业设计中，进程的页表都是以4K页面为单位进行映射的，笔者选择4K页面主要是出于以下考虑：1.内存池中内存的单位就是4K页，因此扫描位图、从池中取出或放回4K页面的操作要比2M或1G页面简便的多。2.页表中的页面是4KB，如果映射的页面是2M或1G的话会使二者的单位不统一，当这些大页面随着系统的使用被4KB小页面碎片化以后很容易出现有内存却用不了的窘境。3.很多小程序所需要的内存也就几KB，把一个2M或1G页面分给它实在是有点浪费。4.实验环境中bochs虚拟机的内存一般都不大，比如笔者的只有32M，以2M页面为单位的话页面数太少，很可能不够用，以1G页面为单位更是不切实际。

系统创建进程页表时首先映射前0ef000h的线性地址，不过这部分地址并不是从内存池中取出页面来映射，而是直接映射到物理地址的前0ef000h，以此来确保中断系统的正常工作。然后映射0ef000h～0fffffh的线性地址，这部分地址映射到从内存池中取出的页面上，其中0ef000h～0effffh用来存储进程在执行或切换过程中各种必要的参数，这部分数据有个名字，叫进程描述符/进程控制块/PCB。0f0000h～0f7fffh用来存储进程的LDT，0f8000h～0fffffh是进程的系统栈，它的作用稍后会提到。进程前100000h 线性地址空间里的数据都不允许应用程序私自修改，因此这些页面的特权级都是ring0。从线性地址的100000h开始才是存储应用程序代码和数据的空间，因此这些页面的特权级都是ring3。在创建进程页表时只映射刚好能装入应用程序的空间，当程序运行过程中需要更多内存时再从内存池中取出页面扩大进程页表的映射区间。

进程调度

进程每次执行的最小时间单位被称为时间片，前边提到的CPU分时轮流执行各个进程的策略被称为时间片轮转算法，它是进程调度的基本算法。在我的毕业设计中，时间片轮转算法是借助时钟中断来实现的，时钟中断每隔【待查阅】就会触发一次，这【待查阅】就是时间片的大小。时钟中断触发后由中断服务程序负责切换进程，为了叙述方便我们把中断前正在执行的那个进程称为进程A，即将执行的那个进程称为进程B，中断服务程序首先把进程A的“执行现场”（各个通用寄存器的值）保存在系统栈里，然后切换到进程B的页表。注意在页表切换前后 前0ef000h的线性地址都映射到了相等的物理地址，因此中断服务程序的运行不受影响，然而0ef000h以后的线性地址空间（进程描述符、LDT、系统栈以及代码和数据）全部都换成了进程B的，这就好比CPU从进程A的房间走出来然后进入了进程B的房间。在切换完页表之后从系统栈中恢复进程B的执行现场，然后返回。代码如下：

push rax

push rcx

push rdx

push rbx

push rbp

push rsi

push rdi

push r8

push r9

push r10

push r11

push r12

push r13

push r14

push r15

mov [pro\_spc],rsp

;;切换到下一个进程的cr3

xor rax,rax

mov eax,[cPid]

shl rax,3

add rax,protab

Timer\_lb0:

add rax,8

cmp rax,0a0000h

jne Timer\_lb1

mov rax,protab

Timer\_lb1:

cmp qword[rax],0

je Timer\_lb0

mov rbx,[rax]

mov cr3,rbx

sub rax,protab

shr rax,3

mov [cPid],eax

mov rsp,[pro\_spc]

pop r15

pop r14

pop r13

pop r12

pop r11

pop r10

pop r9

pop r8

pop rdi

pop rsi

pop rbp

pop rbx

pop rdx

pop rcx

Timer\_rtn:

mov al,20h

out 20h,al

pop rax

iretq

可执行文件的格式

可执行文件，Windows中有PE格式，Linux中有ELF格式，在我的毕业设计中也有一套格式，不过这套格式还没有被正式命名。可执行文件格式如下：

+0

|  |
| --- |
| 映像体所占页面数N  +8 |
| 初始rsp |
| 初始rip  +16  +24 |
| 保留  +1000h |
| 映像体  +1000h\*(N+1) |

如果你了解PE格式或者ELF格式，一定会觉得这套可执行文件格式非常简陋，不过这倒很符合笔者简洁够用的设计思想。可执行文件的前200h个字节称为文件头，文件中的映像体就是程序的代码和数据，创建进程时从线性地址100000h开始映射N个页面，然后把映像体整个装入其中。

进程描述符与LDT

进程描述符中的成员变量如下：

+0

|  |
| --- |
| 栈顶指针  +8 |
| 用户页面数 |
| 系统页面泡沫数  +24  +16 |
| LDT中调用们个数 |
| 保留  +30  +26 |
| 输入缓冲区的线性地址  +38 |
| 输出缓冲区的线性地址 |
| 输入缓冲区的物理地址  +54  +46 |
| 输出缓冲区的物理地址 |
| 保留  +1000h  +62 |

这里介绍一下栈顶指针和用户页面数的含义：在进程切换过程中通用寄存器入栈后，rsp的值会被保存在进程描述符的栈顶指针字段中，现在看来这似乎是多此一举，不过在下一章中你就会明白这是十分必要的。用户页面数记录了进程从线性地址100000h开始映射了多少个页面。剩下的字段以后再说。

进程的LDT中有4个段描述符，从上到下依次用作用户代码段、用户数据段、系统代码段和系统数据段。2个代码段都是64位非一致代码段，用户代码段的DPL是3，系统代码段的DPL是0。2个数据段都是可写段，用户数据段的DPL是3，系统数据段的DPL是0。在64位模式下CPU在访问一个段时会忽略段基址和段界限，因此这4个段的段基址和段界限都是0。

进程表和相关变量

物理地址的10000h～9ffffh是进程表，表中每8个字节为一项，每项的值是一个进程的4级页表基址，0表示无效的空白项。一个进程的4级页表基址在进程表中位于第几项，那么这个进程的进程号就是几（从0开始计）。系统变量Procnt记录了当前系统中的进程个数，cPid记录了当前正在执行进程的进程号。

进程的创建

在系统初始化代码中创建进程的操作由mkshpro函数来完成，该函数唯一的参数是一个文件结构体的地址。该函数首先把可执行文件的前1000h个字节读到缓冲区pgbuf中，然后调用系统调用的ctPro功能创建出进程的页表。mkshpro函数是在系统初始化代码中被调用的，因此调用前cr3指向的是系统页表。用系统调用创建出进程页表之后更改cr3，切换到进程的页表。然后填写进程描述符并在LDT中创建4个段描述符，然后在系统栈底依次填入ss、rsp、标志寄存器、cs和rip的值，以此制造出一个“中断现场”，为以后启动进程做准备。其中ss是用户数据段的选择符，rsp是文件头中的初始rsp，标志寄存器填202h，cs是用户代码段的选择符，rip是文件头中的初始rip。接下来从线性地址的100000h开始往后映射N个页面，再把可执行文件的映像体读入其中。代码如下：

mkshpro:

push 1000h

push pgbuf

push qword[rsp + 24]

call read

mov rax,5

int 0f0h

mov rax,[protab+rax\*8]

mov cr3,rax

mov rdi,[pgbuf]

mov qword[pro\_spc],pro\_coda-20\*8

mov eax,[CKnum]

mov [pro\_spc + 26],eax

mov word[pro\_ldt+5],20f8h

mov byte[pro\_ldt+13],0f2h

mov word[pro\_ldt+5+16],2098h

mov byte[pro\_ldt+13+16],92h

mov qword[pro\_coda-8],8+TI+RPL3

mov rax,[pgbuf+8]

mov [pro\_coda-16],rax

mov qword[pro\_coda-24],202h

mov qword[pro\_coda-32],TI+RPL3

mov rax,[pgbuf+16]

mov [pro\_coda-40],rax

mkshpro\_LP0:

mov rbx,[pro\_spc + 8]

shl rbx,12

add rbx,pro\_coda

mov rsi,111b

mov edx,0ffffffffh

mov rax,2

int 0f0h

inc qword[pro\_spc + 8]

cmp qword[pro\_spc + 8],rdi

jne mkshpro\_LP0

shl rdi,12

push rdi

push pro\_coda

push qword[rsp + 24]

call read

ret 8

外壳进程的启动就是系统初始化工作的终点，启动的代码和时钟中断的后半段相比除了没有发送中断结束信号（EOI）之外一模一样：

mov rsp,[pro\_spc]

pop r15

pop r14

pop r13

pop r12

pop r11

pop r10

pop r9

pop r8

pop rdi

pop rsi

pop rbp

pop rbx

pop rdx

pop rcx

pop rax

iretq

进程的销毁

进程的销毁和进程的创建几乎是“对称”的。进程的销毁有2种情况，一种是进程执行完毕后自己销毁自己，另一种情况是被其他进程销毁。2种情况都由销毁操作的发起者调用终端模块的进程销毁函数，进程销毁函数完成销毁工作然后返回。注意在返回的时候这2种情况就有区别了，如果是进程A销毁进程B，那么返回的时候还返回到进程A调用销毁函数的地方。可是如果是进程A自己销毁自己，那销毁之后进程A就不存在了，因此必须返回到下一个进程。进程销毁函数的代码放到后面的章节中讲解。

1. 模块机制

x86架构的CPU在进入保护模式之后有4个特权级，这些特权级英特尔给出的名字分别是ring0、ring1、ring2和ring3，数字越小权限越高。在我的毕业设计中只用到了ring0和ring3，笔者把ring0称为系统特权级，ring3称为用户特权级，所有的应用程序代码一律设置为用户特权级，其他代码设置为系统特权级。由于包括特权级在内的几方面原因，应用程序不可能直接操作硬件设备，当应用程序需要使用硬件时，它会委托操作系统帮它完成硬件操作。然而操作系统本身并没有对硬件特别全面的支持，几乎所有对硬件的操作都是通过驱动程序完成的，因此操作系统对驱动程序的管理以及接口的定义就显得十分重要。在我的毕业设计中这些东西是通过模块机制来实现的，本章就为大家详细讲解这一机制。

模块文件的格式

模块文件也属于可执行文件的一种，因此它的格式沿用可执行文件，只不过和应用程序相比某些字段的含义有了一些变化：

+0

|  |
| --- |
| 映像体所占页面数N  +8 |
| 初始化函数的映像体偏移  +16 |
| 卸载函数的映像体偏移  +24 |
| 模块引用数  +28 |
| 保留  +200h |
| 后继（固定值0）  +208h |
| 模块名（长度不定，后边的空间用0填充）  +1000h |
| 映像体  +1000h\*(N+1) |

操作系统对模块有4种操作：注册、引用、解除引用和注销。简单来讲，注册就是把模块文件全部装入内存，然后记下映像体所在的各个页面的物理地址的一系列操作。当进程需要使用某个模块时，根据模块注册时记录的物理地址映射模块各个页面到进程的线性地址空间，然后在进程的LDT中建立一个调用门。这些映射页面和创建调用门的操作称为引用。解除引用和注销差不多就是引用和注册的逆操作。

模块表与注册表

在注册模块的时候先取出一个页面，将模块文件的前1000h个字节读入其中，然后让一个指针指向该页面3c0h的位置，一边取出页面读取映像体，一边把页面的物理地址填入指针所指的位置中，每填写一个地址指针就后移一个qword，如果指针指向了页面中最后一个qword而映像体却还没读完的话，就取一个页面并将其物理地址填入这最后一个qword中，然后把刚填的地址赋给指针，继续读取映像体，直到把整个映像体都读入内存。从第一个页面3c0h处开始，保存映像体各个页面地址的这片空间称为注册表，在引用这个模块的时候就根据注册表来引用。那怎么找到某个模块的注册表呢？系统将各个模块的模块名所在的页面视为一个个节点，把它们“串”成一张单向链表，这张链表就是模块表。每个节点中的后继字段用来保存后继节点的物理地址，0表示无后继，表头节点的物理地址保存在内核系统变量区的modlist变量中。

1. 文件系统
2. 终端模块