西南林業大學
SOUTHWEST FORESTRY UNIVERSITY

本科毕业论文(设计)
(二〇二三届)

题　　目　基于 Linux 的操作系统研究

学　　院　大数据与智能工程学院　　专　业　计算机科学与技术专业

学生姓名　杨鑫　　学　号　20211159013

指导教师　王晓林（讲师）

评 阅 人　评阅人姓名（职称）

2023 年 5 月 20 日

原创性声明

本人郑重声明，所呈交的学位论文是本人在指导教师指导下进行的研究工作及取得的研究成果，论文成果归西南林业大学所有。尽我所知，除了论文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 西南林业大学或其他教育机构的学位或证书而使用过的材料。与我共同工作的同志对本研究所作的贡献均已在论文中作了明确的说明。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

作者签名：  日期：2023 年 5 月 20 日

基于 Linux 的操作系统研究

杨鑫

（西南林业大学 大数据与智能工程学院 昆明 650224）

摘　要：本文研究的是基于 Linux 的操作系统。首先介绍了 Linux 的发展历程及其特点，包括开源、自由、可定制等。接着详细介绍了 Linux 操作系统的内核结构和系统调用，解释了 Linux 的开机过程、中断、进程管理、文件系统。

在学习了 Linux 操作系统的基础后，本文在此基础上实现了一个简单的操作系统。该操作系统采用汇编语言与 C 语言编写，实现了简单的进程管理、文件系统和一个简单的 Shell. 在实现过程中，本文详细阐述了操作系统的设计思路、算法实现和关键技术，包括进程调度算法、中断处理算法、文件系统实现和 shell 的一些基本功能应用。

关键词：Boot; Loader; GDT; Kernel; 中断; 进程调度; Shell;

Research on Operating System based on Linux

YangXin

College of Big Data and Intelligence Engineering
Southwest Forestry University
Kunming 650224, Yunnan, China

Abstract: This paper studies the operating system based on Linux. Firstly, it introduces the development history and characteristics of Linux, including open source, freedom, System etc. Next, the internal structure and system adjustment of the Linux operating system are introduced in detail, and the boot process, interruption, and process of Linux are explained. Management, file system.

After learning the basis of the Linux operating system, this article implements a simple operating system on this basis. The operating system is based on the Linux kernel, written in assembly language and C language, and realizes simple program management, file system and a simple single shell. In the actual process, this paper describes in detail the design ideas, algorithm reality and key technologies of the operating system, including Some basic functional applications of calculation method, interrupt processing method, file system implementation and shell.

Key Words: Boot; Loader; GDT; Kernel; Interrupt; Process Scheduling; Shell;

目录

1 绪论

1.1 背景及意义

1.1.1 背景

如果要谈 Linux 是从哪里来，是怎么来的，那就还得从 Unix 操作系统说起。1968 年贝尔实验室、麻省理工、通用电器公司的研究人员开发了名叫“多路复用信息和计算机系统”简称 Multics 的特殊操作系统，Mutics 是在二战结束后的冷战时期诞生的，1957 年苏联发射了第一颗人造卫星，进而准备发射载人宇宙飞船；美国航天局的研究这时却连连受挫，美国总统埃森豪威尔便下决心发展科技，巨款支持科学界。科学家们开始设想将大型计算机作为一种公共设施，通过许许多多的终端为用户提供计算时间的“计算机公用事业”，但是最终以失败告终。

1969 年，贝尔实验室的 Ken Thompson 和 Dennis Ritchie 为了把名叫太空旅行的游戏移植到一台没人用的 PDP-7 小型机上，给程序中加入了文件管理、进程管理的功能，和一组实用工具，一个只能给 2 个用户使用的系统诞生了。受到 MULTICS 的影响，Brian Kernighan 玩笑地给系统取名为“UNICS”（没路信息与计算系统），取谐音便是“UNIX”。1969 — 1970 年，AT&T 的贝尔实验室开发了 UINX 系统，引起了众人的关注，很多人找 Thompson 和 Ritchie 要 Unix 的源代码。一份份的 Unix 源码被流传到各个实验室、学校、公司。

后来 AT&T 回收了 Unix 版权，特别是要求禁止对学生群体提供 Unix 系统源代码，这引起了人们的恐慌。于是在 1984 年，Richand Stallman 发起了开发自由软件的运动。一名大学教授为了教学使用开发了 Minux，因为 Minux 操作系统主要用于教学使用，所以不适合商用。后来赫尔辛基大学的一名学生名叫 Linus Torvalds，接触了 Unix，发现 Unix 操作等待时间长等一些问题，因而学习了 Minux 的核心技术，开发了 Linux。1991 年底，Linus Torvalds 公开了 Linux 内核源码 0.02 版，并吸引了世界各地的顶级黑客不断的完善 Linux，一直发展的到今天。Linux 是一种自由和开放源代码的类 UNIX 操作系统，严格来讲，Linux 只是操作系统内核本身，但通常采用“Linux 内核”来表达该意

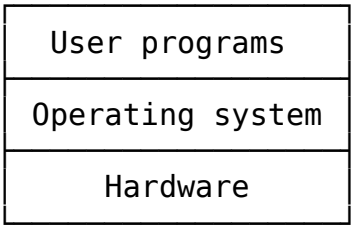

图 1.1　操作系统是介于用户进程和硬件之间的软件

思，而 Linux 则常用来指基于 Linux 内核的完整操作系统。

1.1.2　意义

随着 IT 行业的高速发展，现在基本上每家每户都会使用计算机，而想正常的使用计算机来处理一些事情，比如办公、通信、游戏娱乐等，都得有支持应用软件运行的一个“载具”，而这个“载具”就是操作系统。操作系统，如图 1.1 所示，是硬件基础上的第一层软件，是用户进程与计算机硬件之间的桥梁，它能有效管理软硬件资源，合理组织工作流程，向用户进程提供服务，使用户能够更方便地使用计算机，使整个计算机系统能高效运行，可以说是计算机软件系统的“心脏”。

现在市面上最流行的操作系统莫过于 Windows 和 Linux。大多数人都是从 Windows 系统开始了解计算机和网络的，Windows 易用性高，很适合新接触计算机的人上手，但也因此使多数人对操作系统的选择上产生了很大的误区，认为 Windows 系统比 Linux 系统好用且简单。对于那些使用计算机只是为了上个网看个视频、打打游戏、查查资料的用户确实使用 Windows 系统是最简单的。但抛开这些不谈，Linux 是使用命令行字符模式为主要操作方式，而 Windows 是使用窗口、图标、鼠标点击形象化的方式为主要操作方式，从这点来看 Windows 系统就得比 Linux 系统要多耗费更多的计算机内存，并且“鼠标+键盘”的操作效率比起使用键盘敲指令就可以完成操作的效率是要慢很多的。甚至熟练后 Linux 系统的查资料、看视频等体验都是要优于 Windows 系统的。这些只是用户基本体验上的区别，而 Linux 与 Windows 最大的不同在于 Linux 系统是自由的。自由，不止是说使用这个系统不出钱，重点是 Linux 内核是完全开源的，如果本身技术水平比较高的话，甚至可以用它自己 DIY 一个只属于自己的任何风格的 OS，仔细一想这不是一件非常酷且很有意思的事吗？而 Windows 系统的内核是闭源的，所以到现在 Linux 的版本有很多很多，而 Windows 仅仅只有微软公司的 Windows XP、Win7、

Win11 等，用户的选择很有限。除版本区别外，Linux 与 Windows 还有个很大的区别在于安全性，因为 Linux 的软件基本上也都是开源的，在这个大家庭里的很多都是相互扶持的关系，互帮互助，你需要什么都可以通过一个指令直接获取到软件包直接安装到你的电脑上。而 Windows 系统想要个软件，如果不是官方的可能还要担心有没有病毒等，若是官方的也有可能需要收费购买，破解版又缺乏了安全性并且也是不合法的[4]。

综上所述，所以我觉得研究并且能够做出一个属于自己的基于 Linux 技术的 OS 是一件很有意义的事，它不但能够令我对 Linux 的运行逻辑更加清晰，也能够使我对操作系统开发的了解更加深入透彻。

1.2 开发工具及环境

1.2.1 开发工具

编辑器: VIM v0.7.2; Emacs v28.2;

编译器: GCC v12.2.0;

汇编器: NASM v2.16.01;

虚拟机: Bochs v2.6.9[1];

Makefile: GNU Make v4.3.

1.2.2 环境配置

宿主 OS: Linux v6.1.0-7-amd64 #1 SMP Debian v6.1.20-2

硬件: x86_64

1.3 该研究的主要内容

该研究的主要内容为从操作系统最底层的原理探索一个操作系统从 0 到 1 的实现。首先是计算机的开机过程：引领我们走向计算机这一整个系统的神秘代码，`0x7C00`，一切都要从它开始。计算机通电开机之后，BIOS 便会开始自检，在找到可用的磁盘后，

[1]宿主机如果是 Debian GNU/linux 可以直接敲命令：“sudo apt install vgabios bochs bochs-x bximage” 来安装 bochs。

BIOS 就会把它的第一个扇区加载到 0x7C00，之后由一个 512 字节的主引导记录 MBR[2] 从 BIOS 中接过系统的控制权，也就是 CPU 的使用权。MBR 便是从 0x7C00 处接管 CPU 的，刚好是 512 字节。之后，MBR 寻找操作系统所在的分区，我们规定用 0x80 来表示分区上有引导程序，方便 MBR 从众多分区中方找到操作系统所在的分区，MBR 如果找到了这个分区，就会将 CPU 使用权交给这个分区上的引导程序，该引导程序通常就是内核加载器，所以，为了让 MBR 能够更方便的在那么大的分区里找到内核加载器，通常会把内核加载器的入口地址固定在分区最开始的扇区，该扇区就是操作系统引导扇区（MBR 扇区）。而在 MBR 扇区的前 3 个字节处存放了跳转指令，目的是为了让 MBR 找到分区交接工作后，将处理器带入操作系统引导程序中，至此 MBR 就完成了所有工作，CPU 的控制权就交到了内核手里。到此为止，计算机也只算是开机了而已，此时计算机的状态就像是在执行代码：

```
while(1)
{
  操作系统代码();
}
```

因此，想让计算机知道我们要让它做什么事就还得需要进程，当然，一个进程也只能完成一项工作，在很多时候工作往往并不简单，因此，我们想让进程尽可能的同时多做一些子工作，而这些“子工作”就是线程。现在有了进程以及线程，但是计算机仍然在做一个 while(1) 的循环，所以还需要给计算机一个中断，让它能够判断出事情的重要程度，也就是优先级，来选择先把哪件事情给做好，之后再进行之前没完成的工作，到这才算是实现了一个操作系统[3]。

[2]实际上只有 446 字节用于引导程序和参数，剩下的 64 字节用于分区表和 2 字节用于结束标记的 0x55 和 0xAA。

2 系统设计

2.1 总体设计

本文主要参照 Linux 操作系统的内核框架，尝试实现一个带有文件操作功能和命令行接口的简单操作系统。本系统主要包含以下四个模块：1）启动模块；2）进程模块；3）数据存储模块；4）外围功能模块。

这四个模块分别有自己负责的范围。其中，启动模块负责从上电开机到操作系统启动的过程；进程模块负责进程的产生、调度、和管理；数据存储模块实现硬盘、内存等存储数据的功能，它是后续的文件系统、键盘输入等功能的基础；外围功能模块建立在前三个模块的基础之上，主要是一些针对文件的操作，包括文件的打开、关闭、读写、删除等等。这些功能的实现都离不开前三个模块，只有先完成了前三个模块，才能实现这些基本的功能。

2.2 模块设计

2.2.1 启动模块

启动模块主要涉及 BIOS、MBR、Loader、Kernel 等部分。在操作系统启动过程中，先由 BIOS 来找到 MBR，然后由 MBR 来引导并加载 Loader，再由 Loader 来加载 Kernel。如图 2.1 所示，启动模块的目的可以总结为一个，那就是加载 Kernel。它的详细过程在第 3 章中将会介绍。

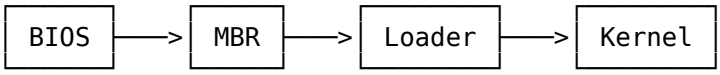

图 2.1 与系统启动相关的各模块

2.2.2 进程模块

进程模块主要包括特权级的转变、用户进程的创建、还有进程调度等方面。有了该模块操作系统才能使用户程序在一个相对安全的环境中执行，并且能够实现多任务同时进行与切换。在本操作系统中，特权级转变采用的是中断返回的方法：当中断发生时，

在中断入口函数 intr%1entry() 中通过 push 操作来保存当前任务的上下文数据，因此，之后也需要有相应的 pop 操作来恢复数据，这属于 intr%1entry() 函数的逆过程，在使用 pop 操作恢复数据后，CPU 就会认为用户进程从中断返回了。在此操作之后，用户进程将在最低的特权级 3 下运行，操作系统处于最高特权级 0, 这样就达到了该模块的目的。当用户进程处在特权级 3 之后，进程的创建就是通过一个 process_execute() 函数来创建的，创建成功后再由时钟中断用 schedule() 从就绪队列中进行调度。详细的步骤在第 4.4 节中将会介绍。

2.2.3 内存模块

此模块就是为了进程而存在的，主要为新产生的进程分配对应的内存空间，在进程结束后再对内存进行回收。内存的创建并不是提前准备好的，它是在需要的时候由程序动态创建的，例如进程需要内存的时候，会调用相关的函数，然后动态分配并且维护内存块资源，再使用完内存后，还需要把内存回收回去，而内存的使用情况一直是由位图 [1] 来进行管理的，所以无论内存的分配或者释放，本质上其实就是在设置相关位图的相应位，也就是在读写位图。

2.2.4 外围功能模块

在有了启动、进程、内存模块之后，就可以实现一个简单的文件系统，一个简单的 shell，shell 的一些简单功能，本文中主要实现的功能有：文件的创建、文件的删除、文件的打开、文件的关闭、文件的写入、文件的读取、文件的删除这些文件操作。

而在 shell 中实现了 Ctrl+L 和 Ctrl+U 快捷键，分别是清屏和清除输入，还有一些简单的 ls、cd、mkdir、rmdir 操作。

[1] 位图是操作系统中常用的一种数据结构，是一个二进制数组，其中的每个 Bit 表示相应的存储快的状态，通常用 0 表示未使用，1 表示已使用。

3 操作系统的启动过程

3.1 基本输入输出系统（BIOS）

BIOS 的全称是 Basic Input/Output System，意思就是基本输入输出系统。它是计算机上第一个运行的软件。当然，它不可能自己加载自己，那么它就只能是由硬件来加载了，而这个硬件就是 CPU[9]。在计算机通电瞬间，如图 3.1 所示，寄存器 CS:IP 会被赋予一个初值：F000:FFF0。之后，CPU 将会执行下面这条指令：

```
1 jmp f000:e05b
```

JMP，顾名思义就是要跳转到内存中的 F000:E05B 这个地址，这里就是 BIOS 待机的地方。JMP 到这之后，BIOS 被唤醒，接管了 CPU 的使用权。之后 BIOS 要做的事情就是：

- 准备好和基本输入输出相关的函数
- 检查硬件
- 将主引导扇区（MBR）的代码加载到内存中的 0x7C00 位置
- 跳转到 0x7C00

图 3.1 Bochs 屏幕截图

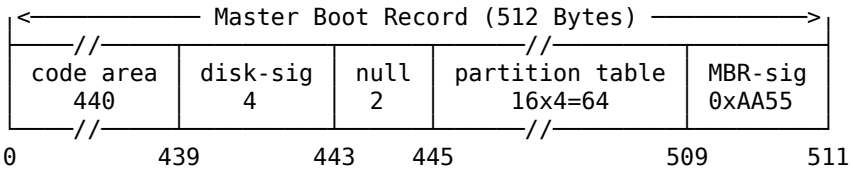

图 3.2 MBR 的结构

3.2 MBR 的工作

MBR，全称是 Master Boot Record，主引导扇区，共有 512 个字节，其结构如图 3.2 所示。主要由三个部分组成，分别是:

- 主引导程序 (Bootloader): 占扇区前 446 个字节，在计算机开机，BIOS 完成自检后，会将 MBR 加载到内存中，然后执行前 446 字节的引导程序。
- 硬盘分区表 (DPT): 占扇区中间 64 个字节，主要用来定位各个分区，访问用户数据。
- MBR 结束标志 0xAA55: 占扇区最后 2 字节，也被称为魔数。每次执行引导程序时都会检查程序的结尾是否是 0xAA55, 若不是的话则会认定这是一个无效的 MBR 引导扇区，将会中止引导。

BIOS 在 0x7C00 处把 CPU 的控制权交由 MBR 之后，将会再次进入待机状态，至此，BIOS 的任务就完成了。MBR.S 中的代码部分是由自己编写的，它可以简单到只要如下三条汇编程序语句:

- 告诉汇编器把 MBR 的起始地址编译为 0x7C00;

```
1 SECTION MBR vstart=0x7c00
```

- $表示本行所在的地址，$-$$意思是本行到本 section 的偏移量，MBR 必须得填满 512 个字节，而最后两个是固定的 0x55 和 0xAA, 所以剩下空着的部分用 0 来填充。

```
1 times 510-($-$$) db 0
```

- MBR 结束;

```
1 db 0x55,0xaa
```

这样就拥有了一个很小的 MBR, 当然只是这样做的话, 屏幕上可能会有很多乱七八糟的信息，因为 Bochs 虚拟机的设计师默认是会显示一些 Bochs 的版本号等等信息的，

所以还是让 Bochs 的界面能够显示的清爽一些，调用汇编寄存器 0x60 号功能号来实现清屏，再通过对字符串的操作来实现在 Bochs 界面上打印出来一个 Hello, OS world! 来表示 MBR 已经被成功加载了。代码如下:

```
    org 07c00h          ; 告诉编译器程序加载到 7c00 处
    mov ax, cs
    mov ds, ax
    mov es, ax
    mov ah, 0x6         ; 利用 0x60 号功能清屏
    mov al, 0x0         ; AL = 上卷的行数 (0 表示全部)
    mov bx, 0x700
    mov cx, 0x0
    mov dx, 0x184f
    int 10h
    call DispStr        ; 调用显示字符串例程
    jmp $               ; 无限循环
DispStr:
    mov ax, BootMessage
    mov bp, ax          ; ES:BP = 串地址
    mov cx, 16          ; CX = 串长度
    mov ax, 01301h      ; AH = 13,  AL = 01h
    mov bx, 000ch       ; 页号为 0(BH = 0) 黑底红字 (BL = 0Ch, 高亮)
    mov dl, 0
    int 10h             ; 10h 号中断
    ret
BootMessage:        db  "Hello, OS world!"

times   510-($-$$)  db  0   ; 填充剩下的空间，使生成的二进制代码恰好为 512 字节
dw  0xaa55                  ; 结束标志
```

如图 3.3 所示，一个最小的操作系统就完成了。

图 3.3 Hello, OS World!

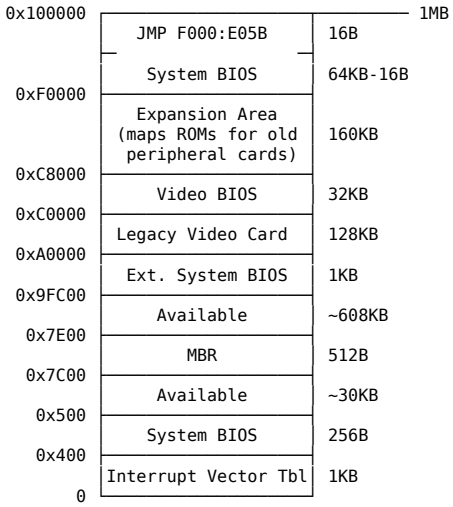

图 3.4　实模式下的内存布局

至此，MBR 已经从 BIOS 上接管了 CPU 的使用权。

3.3　Loader 的工作

MBR 在完成跟 BIOS 的交接后，需要解决三个问题：1）从硬盘上找到 Loader；2）把 Loader 搬运到内存中的什么地方；3）怎样搬运。其中第 1、2 条其实是由自己定义的，可以自己选择把 Loader 放在硬盘里的任何一个地址，该操作系统中放在了第二扇区，所以就需要 MBR 去硬盘的第二扇区寻找 Loader。那么找到 Loader 后要搬运到哪呢?

如图 3.4 所示，有两个可用的区域 `0x7E00 ~ 0x9FBFF` 和 `0x500 ~ 0x7BFF`，本操作系统中将它放在了 `0x900` 这个地方，所以最终 MBR 就会把 Loader 搬运到 `0x900` 这个地方。

3.4　内核的启动

3.4.1　从实模式到保护模式

在 Loader 被搬运到 `0x900` 之后，需要将操作系统由实模式转变到保护模式[7]，原因有以下几点：

- 实模式下操作系统和用户程序会属于同一个特权级。
- 逻辑地址 = 物理地址，顾名思义就是用户程序引用的地址最终都指向真实的物理地址。
- 用户程序能够自由地修改段基址。

- 当需要访问超过 64KB 内存的区域时，要切换段基址。
- 浪费计算机资源，因为一次只能够运行一个程序。
- 一共只有 20 条地址线，加起来最大的可用内存也仅有 1MB。

所以需要将操作系统引导到保护模式下才可以既安全，又能充分地利用计算机的资源，还不用过于担心内存问题。

实模式的寻址方式是“(段基址: 段内偏移地址)”的形式；而保护模式的寻址方式是“(选择子: 段: 偏移地址)”，所以保护模式还需要段描述符来描述一个段的信息，一个段描述符是 8 个字节，多个段描述符就构成了段描述符表也就是“GDT”[1]。因此第一步就是打开 A20 地址线 (详见第 4.1.2 节)，让操作系统能够访问更大的空间，方法也很简单，就是把 0x92 的第一个比特置 1 就可以了。代码如下：

```
in al, 0x92
or al, 0000_0010B
out 0x92, al
```

仅需要 3 行代码，就可以打开 A20 地址线。之后加载之前写好的 GDT。最后一步就是将保护模式的开关 — CR0 寄存器 — 的 PE 比特置 1, 也很简单，就三行代码：

```
mov eax, cr0
or  eax, 0x00000001
mov cr0, eax
```

至此，操作系统已经成功进入了保护模式，接下来就是把 CPU 的使用权交给 Kernel (内核)。

3.4.2 内核初始化

Loader 会把内核从硬盘上读出来，然后加载到内存中去，至此，CPU 的使用权就会传到内核手上，这就是操作系统从电脑开机到接管 CPU 使用权的一个大致过程。内核的代码如下：

```
int main(void)
{
    while(1);
}
```

没错，就可以是这么一个简单的程序，操作系统实质上从头到尾也就是在做这么一件简单的事情。

[1]GDT(Global Descriptor Table)[5], 全局描述符表，一个 CPU 对应一个 GDT，它可以被放在内存的任何地方，也就是说它是全局的，存放在内存中的某个位置，而这个位置将由我们来指定

4 SheepOS 的实现

“SheepOS”，是我为该操作系统取的一个名字，这名字的由来也很简单，毕竟是自己做的一个 OS，那也总得自己给它取个名字，Sheep 其实跟我的名字也有一点关系，在以前初中的时候刚学没几个单词，同学们就开始取英文名玩儿，他们简单粗暴的给我取了个“Sheep Star”的名字，现在想想还挺好笑的。但我也并不排斥 Sheep 这个词，而且我也很喜欢绵羊，所以索性就把我的 OS 叫做 SheepOS 了。

4.1 系统启动过程

当按下计算机的开机键后，计算机开始自检，之后运行第一个软件 BIOS。BIOS 的工作在第 3.1 节有过介绍，在 BIOS 把 CPU 使用权交给了 MBR 之后，MBR 会在硬盘上的第二个扇区找到 `Loader.bin` 并把它加载到内存中去，之后操作系统将在 Loader 中跳入保护模式，并且由 Loader 来加载内核。

4.1.1 读取硬盘扇区

为了让 MBR 能够定位 Loader，需要一个 `rd_disk()` 函数来读取 Loader 所在硬盘的扇区。该函数的完整代码在附录 A.2 中。该函数的工作步骤大致如下：1）备份 `eax` 和 `cx` 寄存器；2）设置要读取的扇区数；3）将 LBA[1] 地址存入 `0x1F3 ~ 0x1F6`；4）向 `0x1F7` 端口写入读命令，`0x20`；5）检测硬盘状态；6）从 `0x1F0` 端口读数据；

之前在第 3.3 节中提到过，SheepOS 的 Loader 放在硬盘的第二个扇区，所以，在 MBR 中可以：

```
1 mov al,2
```

来让 MBR 来读取硬盘的第二个扇区，在这儿就可以找到 `Loader.bin`，这些就是 MBR 对硬盘的操作。在 MBR 找到 Loader 之后会把它搬运到之前写好的位置：`0x900` 处，然后就会把 CPU 的使用权交给 Loader, MBR 的工作至此也就结束了。

[1]Logical Block Address, 逻辑块地址，可以理解为硬盘的物理地址

4.1.2 进入保护模式

保护模式的必要性在之前第 3.4.1 节中已经提及，并且大致介绍了进入保护模式的方法。保护模式是在 Loader.bin 中进入的。在上一节中，MBR 已经完成了它的工作，将 Loader 引导到了合适的位置，并把 CPU 的使用权交给了 Loader。这里的 Loader.bin 其实已经超过了 512 个字节，所以在 MBR 中加载 Loader 的读入扇区数需要扩大，将它改成读入第 4 扇区。

```
mov cx,4
call rd_disk
```

- 描述 Loader 的起始地址:

```
LOADER_BASE_ADDR equ 0x900    ;equ 是 nasm 的提供的伪指令，意为 equal，
                              ;用于给表达式起个意义更明确的符号名
```

- 配置全局描述符表 GDT(GDT 的概念详见第 3.4.1 节);

```
DESC_G_4K   equ   1_00000000000000000000000b
DESC_D_32   equ    1_00000000000000000000000b
DESC_L      equ     0_00000000000000000000b    ;64 位代码标记，此处标记为 0
↪  便可。
DESC_AVL    equ      0_00000000000000000000b    ;cpu 不用此位，暂置为 0
DESC_LIMIT_CODE2  equ 1111_0000000000000000b
DESC_LIMIT_DATA2  equ DESC_LIMIT_CODE2
DESC_LIMIT_VIDEO2 equ  0000_000000000000000b
DESC_P            equ     1_000000000000000b
DESC_DPL_0        equ      00_0000000000000b
DESC_DPL_1        equ      01_0000000000000b
DESC_DPL_2        equ      10_0000000000000b
DESC_DPL_3        equ      11_0000000000000b
DESC_S_CODE       equ        1_000000000000b
DESC_S_DATA       equ DESC_S_CODE
DESC_S_sys        equ        0_000000000000b

DESC_TYPE_CODE    equ  1000_00000000b   ;x=1,c=0,r=0,a=0
                                        ;代码段是可执行的，非依从的，不可读
                                        ↪  的，已访问位 a 清 0.
DESC_TYPE_DATA    equ  0010_00000000b   ;x=0,e=0,w=1,a=0
                                        ;数据段是不可执行的，向上扩展的，可写
                                        ↪  的，已访问位 a 清 0.
DESC_CODE_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + DESC_L +
↪  DESC_AVL + DESC_LIMIT_CODE2 + DESC_P + DESC_DPL_0 + DESC_S_CODE +
↪  DESC_TYPE_CODE + 0x00
DESC_DATA_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + DESC_L +
↪  DESC_AVL + DESC_LIMIT_DATA2 + DESC_P + DESC_DPL_0 + DESC_S_DATA +
↪  DESC_TYPE_DATA + 0x00
DESC_VIDEO_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + DESC_L +
↪  DESC_AVL + DESC_LIMIT_VIDEO2 + DESC_P + DESC_DPL_0 + DESC_S_DATA +
↪  DESC_TYPE_DATA + 0x0b
```

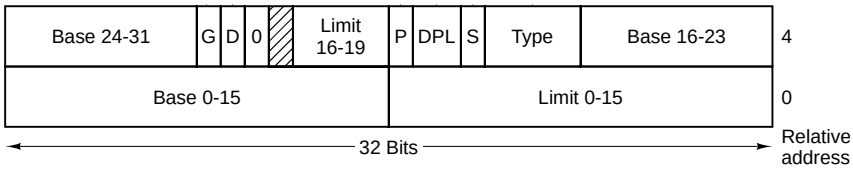

图 4.1　段描述符格式

图 4.1 所示为 GDT 的标准格式，GDT 表就是根据该格式来进行配置的。值得注意的是从 `DESC_DPL_0` 到 `DESC_DPL_3` 这四行代码，它们代表的是保护模式特有的特权级概念。如图 4.2 所示，特权级号越小权限越大，因此在进入保护模式后，操作系统的特权级应当是在 0 处，而用户程序位于最低的特权级 3 处；若是实模式的话所有的一切都存在于同一个特权级，这样实在是太危险了，所以这也是要费尽千辛万苦也要使操作系统进入保护模式的原因之一。

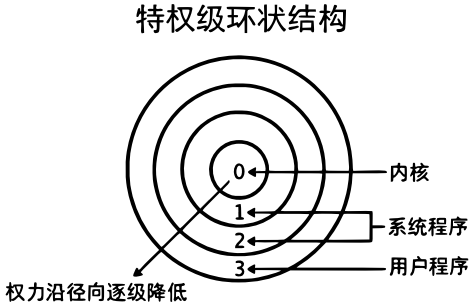

图 4.2　特权级结构

关于 A20 地址线打开与否的区别:

- 如果 A20 地址线被打开，当访问到 `0x100000 ~ 0x10FFEF` 之间的地址时，CPU 将真正访问到这块物理地址。
- 如果 A20 地址线未被打开，当访问 `0x100000 ~ 0x10FFEF` 之间的地址时，CPU 将采用 8086/8088 的地址回绕。实模式下的地址线是 20 位，最大寻址空间是 1MB，即 `0x00000 ~ 0xFFFFF`。超出 1MB 内存的部分在逻辑上虽然也是正常的，但物理内存中没有与之对应的部分，所以为了让 “(段: 偏移地址)” 的策略可用，CPU 会采取将超过 1MB 的部分自动绕回 0 地址的做法，继续从 0 地址开始映射，这就叫做地址回绕。如：`0x100000`，由于没有第 21 位地址线，相当于丢掉了进位 1, 将会变成 `0x00000`。

所以，在进入保护模式后就需要抛弃地址回绕这种做法，相应的也就需要超过 20

```
<bochs:1> c
^CNext at t=1356216071
(0) [0x000000000ce6] 0008:00000ce6 (unk. ctxt): jmp .-2 (0x00000ce6)      ; ebfe
<bochs:2> xp 0xb00
[bochs]:
0x00000b00 <bogus+       0>:    0x02000000
<bochs:3>
```

图 4.3　用 xp 命令查看物理内存

条地址去访问更大的空间，而打开 A20 地址线就是以此为目的。最后，

```
1 jmp dword SelectorCode32:0
```

将 CR0 寄存器的 PE 比特置 1 后，操作系统就成功的跳入了保护模式。

4.1.3　加载内核

在加载 Kernel 之前，得先获取物理内存容量，只有掌握了物理内存的大小，才能更好地操作虚拟内存。在此，可以利用 BIOS 中断号 0x15 的子功能来获取物理内存的大小。代码详见附录 A.3。其工作过程大致如下：

- 定义好调用之前输出的寄存器 ECX 和 EDX;
- 调用 int 0x15 号中断;
- 在返回后输出的 CF 位为 0 时，寄存器 EAX、ES:DI、ECX、EBX 便会出现对应的结果。

如图 4.3 所示，在执行 xp 0xB00 后，结果是 0x02000000，换算成十进制正好是 32MB, 故检测结果是正确的。到这里 SheepOS 已经有了物理内存检测的功能，这下就能够对物理内存做到心中有数了。虽然对物理内存的多少已经能够掌握了，但这些内存对于以后整个计算机而言还是太小了，在保护模式中，地址空间达到了 4GB, 但目前是所有的进程包括操作系统共享这一个 4GB, 这样听起来就觉得 4GB 其实也很小，所以还需要对内存进行分页。

图 4.4 所示是分页机制的工作原理，在打开分页机制后，原本的保护模式的寻址方式又将改变，将从“(段选择子：段：偏移地址)”变成“(GDT: 虚拟地址: 页: 物理地址)”的寻址方式。因为有了分页机制后，就可以将线性地址转换成物理地址，且可以用大小相等的页来代替大小不等的段，具体如图 4.5 所示。

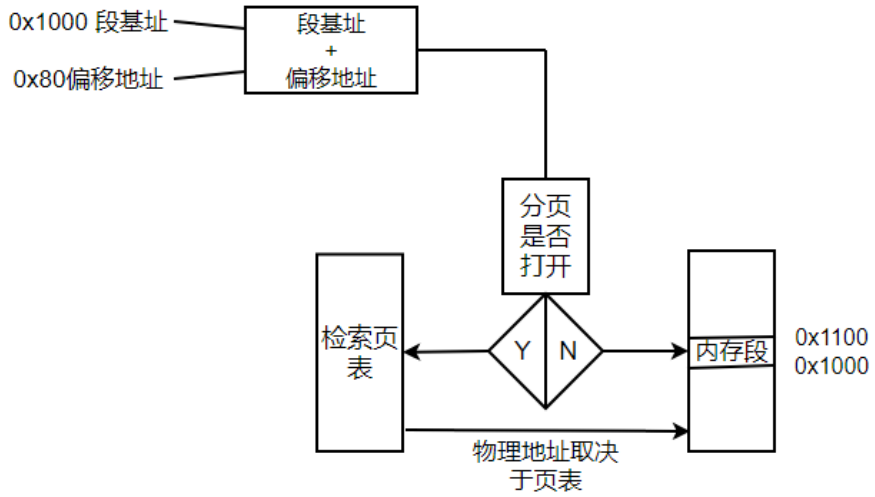

图 4.4　分页机制

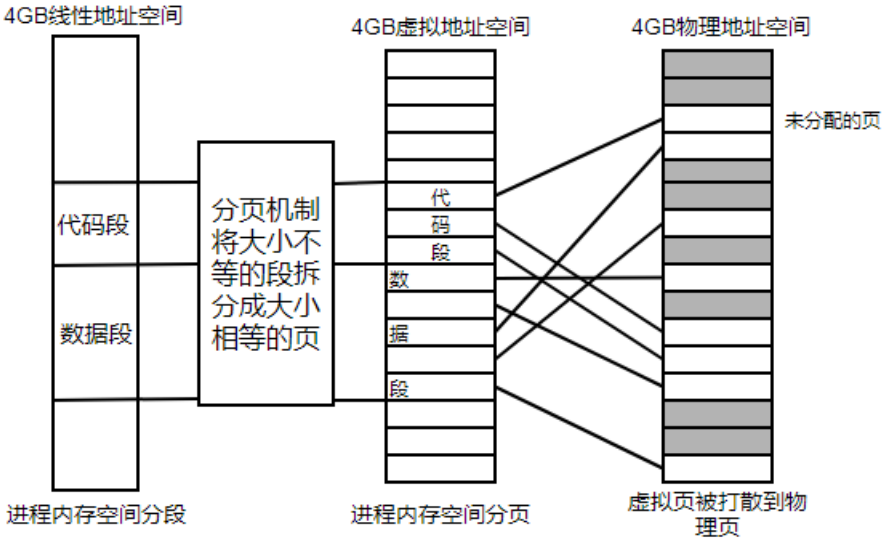

图 4.5　分页机制的作用

虚拟地址与物理地址之间有一个映射关系，它们之间的转换是以页 (4KB) 为单位的。一个页表有 1024 个页表项，一个页表项指向一个页的首地址，那 4GB 就需要 1M 个页，而一个页是 4KB, 所以就需要 1024 个页表，也就是一个页表目录 (二级页表)。而启用分页机制需要按顺序做 3 件事：1）准备好页表目录和页表；2）将页表地址写入控制寄存器 CR3；3）寄存器 CR0 的 PG 比特置 1。页表目录和页表的创建过程大致如下:

- 需要先把页目录占用的空间逐字节清 0;
- 之后创建页目录项 (PDE);

- 再将页目录项 0 和 0xc00 都存为第一个页表的地址;
- 创建页表项 (PTE);
- 创建内核其它页表的 PDE;
- 把页表地址写入 CR3;
- 将 CR0 的 PG 位置 1;
- 最后在开启分页后，用 GDT 新的地址重新加载;

代码详见附录 A.4。

如图 4.6,GDT 的段基址已经变成了 0xc0000900,段描述符的段基址也变成 0xC00B8000,而不是 0xB8000 了，说明 SheepOS 已经成功进入了分页机制的运行模式。

```
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file log.bochsrc.disk
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b          ; ea5be000f0
<bochs:1> c
^CNext at t=538266908
(0) [0x000000000ce6] 0008:00000ce6 (unk. ctxt): jmp .-2 (0x00000ce6)      ; ebfe
<bochs:2> info gdt
Global Descriptor Table (base=0x00000900, limit=31):
GDT[0x00]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x01]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Conforming, Accessed, 32-bit
GDT[0x02]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
GDT[0x03]=Data segment, base=0x000b8000, limit=0x00007fff, Read/Write, Accessed
You can list individual entries with 'info gdt [NUM]' or groups with 'info gdt [NUM] [NUM]'
<bochs:3>
```

图 4.6　分页后 GDT 的变化

接下来就该加载 SheepOS 的“心脏” — 内核[2]了,加载内核到内存这一步跟 MBR 的工作基本上都是同样的,首先第一步就是像把 Loader.bin 放到第 4 扇区那样把 Kernel.bin 放到自己想放的位置上去，因为 Loader 在第 4 扇区上，第 0 扇区放的是 MBR, 最好给 Loader 多预留些位置，所以在该操作系统中选择把 Kernel.bin 放到第 7 扇区，确定好要放的扇区的位置后，可以直接用 dd 命令往磁盘上写:

```
dd if=kernel.bin of=/SheepOS/hd60M.img bs=512 count=200 seek=7 conv=notrunc
```

- seek 为 7，目的是越过前 7 个扇区 (第 0 ~ 6 个扇区)，在第 7 个扇区写入内核。
- count 为 200, 目的是一次往参数 of 指定的文件中写入 200 个扇区。

这样，Kernel.bin 就被成功的写入磁盘了，接下来就该让 Loader 去找到 Kernel 然后把它从硬盘上搬运到内存中去，跟之前 MBR 搬运 Loader 一样，还需要一个缓冲区，如图 4.7 所示，跟之前 Loader 一样，有两个可用区域，准确的说是三个，因为现在 MBR 也结束任务了，所以之前 MBR 所占用的区域现在也可以使用。

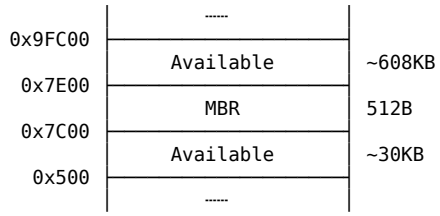

图 4.7　可用缓冲区

之后内核文件会越来越大，所以，为了能够预留出足够的空间，需要将 Kernel.bin 加载到地址较高的空间，但内核映像要放置在较低的地址中，因为 Kernel.bin 在被 Loader 加载之后就没用了，之后，内核映像在向高地址处扩展的时候也可以覆盖之前加载到高地址的 Kernel.bin 所占用的空间。所以该操作系统中把 Kernel 加载到了 0x70000 这个地址[1] 。Loader 加载 Kernel 到内存的代码如下:

```
KERNEL_START_SECTOR equ 0x200
KERNEL_BIN_BASE_ADDR equ 0x70000
mov eax, KERNEL_START_SECTOR        ; kernel.bin 所在的扇区号
mov ebx, KERNEL_BIN_BASE_ADDR       ; 从磁盘读出后，写入到 ebx 指定的地址
mov ecx, 200                ; 读入的扇区数

call rd_disk_32
```

其中，0x200 就是为 Kernel.bin 设置的地址，0x70000 是 Loader 在 0x200 找到 Kernel.bin 后加载到的地址。在 Loader 把 Kernel 加载到 0x70000 之后，还需要对内核进行一下初始化（程序代码参见附录 A.5），其过程大致如下:

- 将 kernel.bin 中的段拷贝到各个段自己被编译的虚拟地址处;
- 把这些段单独提取到内存中;
- 判断段类型，如果不是 PT_NULL，就把这些段拷贝到编译的地址中。

至此，SheepOS 已经成功拥有内核了，且之后 CPU 的控制权也将交给内核。

4.2　简单的屏幕输出

在之前没有 Print 函数，想在屏幕上输出文本时，不是利用 BIOS 中断就是利用系统调用，很不方便，所以就需要为 SheepOS 增加属于它自己的 Print 函数。要实现一个 Print 函数大致可以分为以下几个步骤（具体代码参见 A.6）:

- 备份寄存器;

- 获取光标当前所处的位置坐标，该位置就是下一个可打印字符的位置;
- 获取需要打印的字符;
- 判断字符类型; 如回车、换行、退格则会被特殊处理，除此之外将会直接输出显示;
- 判断需不需要滚屏;
- 更新光标的位置坐标，让它指向下一个打印字符的位置;
- 恢复寄存器。

这样就可以实现对单个字符的打印，效果如图 4.8 所示，这里只实现了对单个字符的 print，目前该效果是由多个 put_char 来打印字符拼到一起来实现的

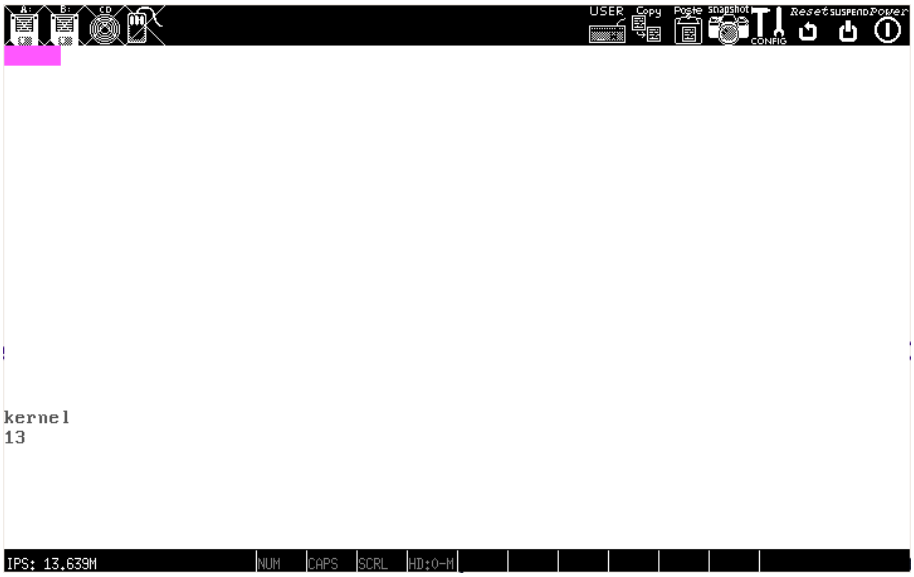

图 4.8　字符打印

```
int main(void)
{
    put_char('k');
    put_char('e');
    put_char('r');
    put_char('n');
    put_char('e');
    put_char('l');
    put_char('\n');
    put_char('1');
    put_char('2');
    put_char('\b');
    put_char('3');
    while(1);
}
```

这样就形成了图 4.8 中的效果，在字符 “kernel” 最后换行符"\n"也起到了效果，成功换了行，而最后的 “1” 和 “3” 便是 2 和 3 之间的退格键将字符 “2” 删除了，只留下

了“1”和“3”。

4.2.1 字符串的打印

对于字符串 Str 的打印其实就是对单个字符打印函数进行一个封装。方法如下：

- 声明全局函数；

```
global put_str
```

- 备份 ebx 和 ecx 寄存器

```
   push ebx
   push ecx
   xor ecx, ecx                    ; 准备用 ecx 存储参数，清空
   mov ebx, [esp + 12]             ; 从栈中得到待打印的字符串地址
.goon:
   mov cl, [ebx]
   cmp cl, 0                       ; 如果处理到了字符串尾，跳到结束处返回
   jz .str_over
   push ecx                        ; 为 put_char 函数传递参数
   call put_char
   add esp, 4                      ; 回收参数所占的栈空间
   inc ebx                         ; 使 ebx 指向下一个字符
   jmp .goon
.str_over:
   pop ecx
   pop ebx
   ret
```

在有了 put_char 之后 put_str 无非就是多调用几个 put_char 来进行打印最终达到字符串的效果，只不过封装以后打印字符串就不用像之前那么繁琐，使用很多行代码来实现。效果如图 4.9：

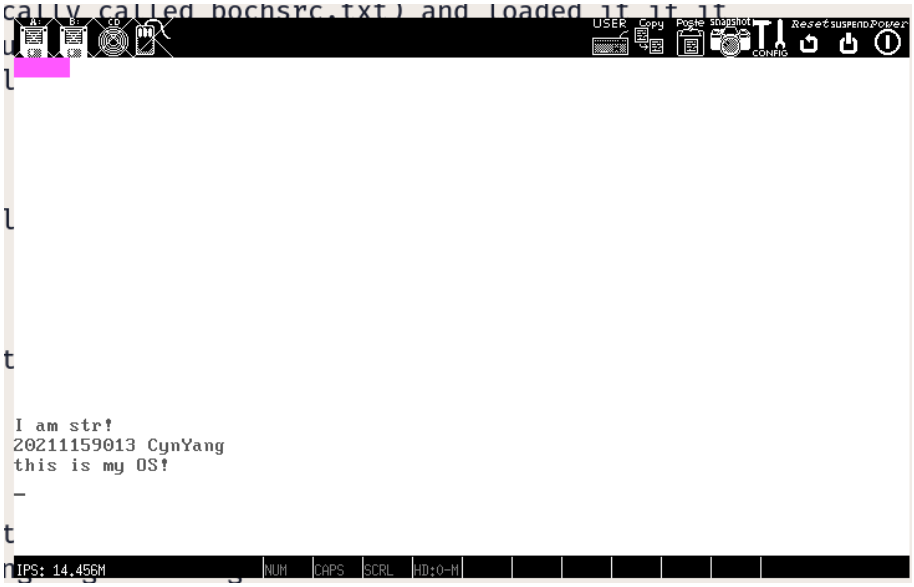

图 4.9　打印字符串

至此，SheepOS 就可以在打印出来想要打印的东西。在有了 put_str 之后打印这些字符就变得容易了许多:

```
void main(void)
{
   put_str("I am str!\n");
   put_str("20211159013 CynYang\n");
   put_str("this is my OS!\n");
   while(1);
}
```

像这么多个字符只需要三行代码，不换行的话其实一行也就够了。

4.2.2　整数的打印

既然 SheepOS 有了能够打印 Str 的功能，那怎么能没有数字打印呢，在此次设计中只实现了对数字的打印，原理就是将数字 9 转变为字符的'9' 来输出，提到这个转换就不难想到 ASCII 码，在 ASCII 码表中，字符‘0’ ~ ‘9’的范围是 48 ~ 57, 大写字母‘A’ ~ ‘Z’的范围是 65 ~ 90，所以

- 对数字的处理就是:0 ~ 9 之间的数字，用该数字加上字符'0' 的 ASCII 码 48;
- 对字母的处理就是:A ~ F 之间的数字，用该数字减去 10 后再加上字符'A' 的 ASCII 码 65;

代码详见附录 A.7 现在 SheepOS 就有一个简单的打印整形 Int 的功能了。

```
void main(void)
{
   put_str("I am Str\n");
   put_str("20211159013 CynYang\n");
   put_int(0);
   put_char('\n');
   put_int(9);
   put_char('\n');
   put_int(0x00021a3f);
   put_char('\n');
   put_int(0x12345678);
   put_char('\n');
   put_int(2023-04-03);
   while(1);
}
```

图 4.10　打印整形字符 int

如图 4.10 所示。它将输入的数字经过 ASCII 码表转换成对应的字符成功输出了。

4.3　中断

现在操作系统一直在做一件事情，那就是

```
while(1)
{
  操作系统代码();
}
```

不论在其中完成了多少功能，它都是在进行一个死循环，要是没有中断的话，都不能使操作系统一边处理图片，一边敲键盘聊天。就好似我正在玩一个游戏，但现在得跟朋友

出去吃饭或是门铃响了要去开门，若是没有暂停功能的话应该很难马上放开这个游戏去做别的事吧。操作系统的中断亦是如此，正是因为有了中断，我们才能够同时享受计算机的多种服务。中断又分为两种：

- 外部中断: 外部中断顾名思义就是 CPU 外部的中断，所以又叫做硬件中断。这个中断信号都是来自外部硬件的，如网卡、磁盘控制器、键盘、鼠标、音频等等，与内部中断不同，它是异步的，就算有正在运行的指令，它也能够发生。
- 内部中断: 又称为软中断，它通常作为 CPU 的异常处理，中断存在的意义主要是为了让计算机能够实现多任务，包括用户程序之间的切换、软件与软件之间的并存都需要有中断才能实现。

4.3.1 中断描述符表

中断描述符表就是保护模式下用于存储中断处理程序入口的表，当 CPU 收到一个中断后，需要用中断向量在此表中检索对应的描述符，然后在描述符中找到中断处理程序的起始地址，之后执行中断处理程序。如表 4.1 所示 [2]，第一列 `INT_NUM` 表示中断向量号，共有 255 个，中断的机制就是在收到一个中断信号后，调用相对应的中断处理程序，所以 CPU 在收到中断信号后，会根据中断向量号去到中断描述符表中找到相应的中断程序的地址。

4.3.2 中断控制器 8259A

在认识中断的概念后，才知道操作系统真是太忙了，计算机上若是插入一个鼠标，操作系统要加载一下，插入一个键盘，操作系统也要加载一下，本来操作系统就有很多自己的事情需要做了，所以就需要一个部件来对这些中断进行统一的管理，而这个部件就是可编程中断控制器 `8259A`。`8259A` 的作用是负责所有来自外设的中断，且它是可编程的，应该可以把它称作是打开中断的入口。计算机内有两片 `8259A` 芯片, 分别是主片和从片，如图 4.11。

[2] 来源：`https://en.wikipedia.org/wiki/Interrupt_descriptor_table`

表 4.1　中断描述符表

INT_NUM	Short Description
0x00	Division by zero
0x01	Single-step interrupt
0x02	NMI
0x03	Breakpoint
0x04	Overflow
0x05	Bound Range Exceeded
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double Fault
0x09	Coprocessor Segment Overrun
0x0A	Invalid Task State Segment
0x0B	Segment not present
0x0C	Stack Segment Fault
0x0D	General Protection Fault
0x0E	Page Fault
0x0F	reserved
0x10	x87 Floating Point Exception
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception
0x14	Virtualization Exception
0x15	Control Protection Exception

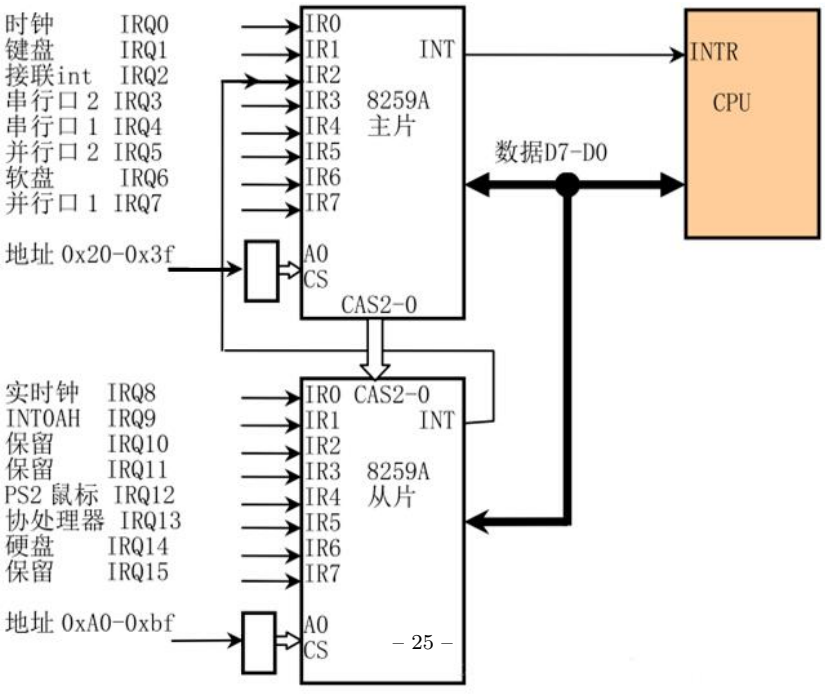

图 4.11　8259A 芯片

8259A 的芯片是直接安装在主板上的，所以可以直接对其进行操作，例如移动了一下鼠标，如图 4.11 在 8259A 的从片 IRQ12 引脚处就会收到一个高电频，之后从片就会把该信息通过 8259A 主片的 IRQ2 引脚传递到主片上，再从主片直接反映到 CPU 上说鼠标将要移动了请求 CPU 来处理。那么处理其它中断也是一样的，在 8259A 收到一个优先级为多少的中断后会告诉 CPU: 这是优先级靠前或是靠后的中断，需要进行处理。如图 4.12 在 CPU 收到后会根据中断向量号在中断描述符表中进行索引然后找出相应的选择子和偏移地址，在得到选择子与偏移地址后，再到 GDT 中去寻找对应的描述段，之后就再在内存中找到该描述段的代码然后再去执行。进入中断的方法就是 Kernel 上调用中断处理函数，代码详见 A.8 和 A.9，其过程如下:

- 如果是从片上进入的中断, 除了往从片上发送 EOI 外, 还要往主片上发送 EOI;
- 调用 idt_table 中的 C 版本中断处理函数;
- 存储各个中断入口程序的地址，形成 intr_entry_table 数组

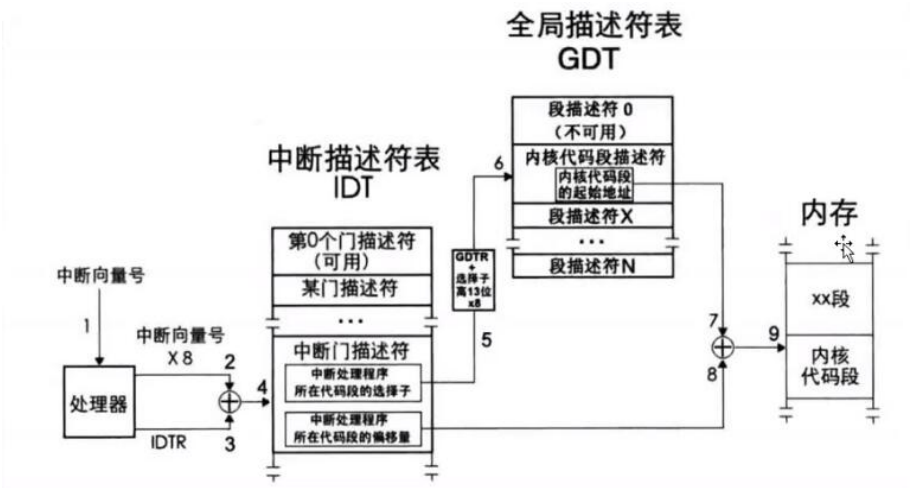

图 4.12　中断过程

如上图 4.11, 当向主片 0xA0 地址和从片 0x20 地址发送 EOI 后，就代表告诉 8259A 中断程序处理结束了。

在进入中断前，还需要先保存中断发生前上下文的环境，以免发生数据丢失的情况，这里使用 PUSHAD 指令压栈压入 32 位寄存器:

```
1 push ds
2 push es
3 push fs
4 push gs
5 pushad
```

在中断调用完后也需要用 POPAD 来恢复之前的环境:

```
add esp, 4     ; 跳过中断号
   popad
   pop gs
   pop fs
   pop es
   pop ds
   add esp, 4 ; 跳过 error_code
   iretd
```

至此就是 8259A 控制器的相关中断内容。

4.3.3 键盘输入

SheepOS 现在已经有了中断，而在之前的图 4.11 上不难发现 8259A 主片 IRQ1 引脚对应的是键盘，所以可以大胆试想一下，是不是 SheepOS 即将可以用键盘直接在屏幕上进行输入了？答案是肯定的。

在此之前，首先简单介绍下键盘敲击的过程。在键盘中有着一个名为“键盘编译器”的芯片，它通常是 8048 以及兼容芯片，作用就是监控键盘的输入，然后把数据发送给计算机，当然计算机主板上也有一个键盘控制器，专门接受和解码来自键盘的数据，然后跟 8259A 和各个软件之间进行通信。敲击键盘的步骤可以分为三种情况:

- 按下
- 保持按下的状态
- 松开

键盘按下松开后产生的编码叫做“扫描码”，当键盘按下和保持按下的时候产生 Make Code，而当松开后会产生 Break Code。在知道有扫描码的概念之后,就可以对照着键盘扫描码的编码来与相应的字符进行处理。如附录中代码 A.10 就是对键盘各个按键进行解析，之后就能将字符正确的打印在屏幕上。如图 4.13,SheepOS 现在就可以自由的在屏幕上输入任何想输入的字符了。

图 4.13　键盘输入

4.4　进程的实现

特权级的概念在前面图 4.2 稍有提及，到现在 SheepOS 的程序一直是在最高特权级 0 级下运行的，这非常危险，意味着用户程序有着跟操作系统同等的权限，想干什么干什么，若是有什么不听话的程序的话，后果往往是灾难性的。操作系统存在的目的之一就是要管理资源的，所以操作系统可以一直处于最高特权级，拥有最高的权利，但被管理的程序当然不能跟操作系统拥有同等的权利，所谓“一山不容二虎”，一般情况下用户程序得处于特权级 3 级才行，所以在实现进程之前得先把权限的高低分清楚。

4.4.1　从特权级 0 到特权级 3

一般情况下 CPU 是不允许从高特权级跳向低特权级的，所以如果想从特权级 0 跳到特权级 3 的话，得采取一些手段，可以用从调用门和中断返回的手段来实现特权级 0->特权级 3 的转变，在 SheepOS 里使用的是中断返回的方式。既然要用中断返回，那就少不了 `iretd` 指令[8]，`iretd` 指令会将栈中的数据当作返回的地址，还会加载栈里的 `eflags` 的值到 EFLAGS 寄存器中，若栈里的 `cs.rpl` 特权级更低的话，CPU 的特权级 Check 通过后，还会把 `cs` 载入到 CS 寄存器中，`ss` 载入到 SS 寄存器中，之后 CPU 进

入低特权级。因此就必须在栈中的时候提前把数据准备妥当为 iretd 指令使用。简单的说就是把进程前后的内容都存到栈里，然后通过 pop 把用户进程的数据加载到寄存器，最后使用 iretd 退出中断。

- 退出中断的出口:intr_exit 函数;

```
intr_exit:
; 以下是恢复上下文环境
   add esp, 4 ; 跳过中断号
   popad
   pop gs
   pop fs
   pop es
   pop ds
   add esp, 4 ; 跳过 error_code
   iretd
```

该函数的作用是用来恢复发生中断时，被中断了的前后任务状态，并且退出中断。在有了这些基础之后，只需将栈中存储的 CS 选择子里的 RPL 的值设为 3, 然后栈中的段寄存器的选择子要指向 DPL 为 3 的内存段，使栈中 eflags 的 IF 位为 1、IOPL 位为 0, 这样用户程序的特权级就成为最低的第 3 位了。代码如下:

```
void start_process(void* filename_)
{
   void* function = filename_;
   struct task_struct* cur = running_thread();
   cur->self_kstack += sizeof(struct thread_stack);
   struct intr_stack* proc_stack = (struct intr_stack*)cur->self_kstack;
   proc_stack->edi = proc_stack->esi = proc_stack->ebp = proc_stack->esp_dummy =
   ↪  0;
   proc_stack->ebx = proc_stack->edx = proc_stack->ecx = proc_stack->eax = 0;
   proc_stack->gs = 0;                  // 用户态用不上,直接初始为 0
   proc_stack->ds = proc_stack->es = proc_stack->fs = SELECTOR_U_DATA;
   proc_stack->eip = function;          // 待执行的用户程序地址
   proc_stack->cs = SELECTOR_U_CODE;
   proc_stack->eflags = (EFLAGS_IOPL_0 | EFLAGS_MBS | EFLAGS_IF_1);
   proc_stack->esp = (void*)((uint32_t)get_a_page(PF_USER, USER_STACK3_VADDR) +
   ↪  PG_SIZE) ;
   proc_stack->ss = SELECTOR_U_DATA;
   asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (proc_stack) :
   ↪  "memory");
 }
```

用户进程的前后数据会保存在 struct intr_stack 栈中,那么相同的 struck thread_stack 就是用来保存在中断处理程序的时候，切换任务时的前后文数据，其余是为 8 个通用寄存器进行初始化，在程序开始运行之前，都没什么实际意义的值，因此直接初始化为 0 就行。

4.4.2 创建用户进程

创建用户进场的大概流程是:

- 先在内核中分配一页内存，为线程 `thread` 分配一页 PCB[3];
- 初始化线程，为进程创建虚拟地址位图;
- 创造一个为了调用初始化函数 `start_process` 的线程;
- 分配进程的页表空间;
- 将带有初始化函数 `start_process` 的线程放到就绪队列中，等待调用;

用户进程的详细代码在附录 A.11, 大部分是页表项和页目录项的创建，因为为了方便操作系统为用户进程提供各种系统功能的调用，就必须确保用户程序要在自己的地址空间中访问到内核才行，也就是说内核空间得是用户空间的一部分，要做到这点的话，虚拟地址空间就得由页表控制，页表由操作系统来管理，因此，用户进程的虚拟空间是由操作系统来进行规划和分配的。既然用户空间得由页表来表示的话，那么就得用设置页表来解决了，所以创建用户进程的意义实质上就是为其创建页表。

4.4.3 进程调度

至此，用户进程已经创建完毕，并且已经准备就绪随时等待调用，(进程调度代码详见附录 A.12) 调用的过程大致如下:

- 调用时钟中断处理函数;
- 调用调度器函数 `schedule`;
- 调用任务切换函数 `switch_to`.

对于进程的优先级高低，是通过调度器函数 `schedule` 中的 `ticks` 来衡量的，操作系统会先给进程初始化一个优先级，再通过调用系统调用函数 `get_pid` 调用查看进程执行的 ticks, 再和初始化的优先级进行比对，最后根据最终的优先级对进程进行调度。

[3]PCB（Process Control Block，进程控制块）是一种数据结构，用于存储和管理进程的状态信息。每个进程对应一个 PCB, 它包含着操作系统所需要的所有信息，以便操作系统对进程进行调度和管理。

4.5 文件系统

现在在 SheepOS 上虽然可以随意的敲击键盘输入字符，但这些字符始终都只是存在于屏幕上的东西，没有任何的实际意义，而文件系统就是为了使这些东西拥有“生命”而存在的，让它们真真实实的存在于操作系统中。对于 LinuxOS 而言，文件是最熟悉的东西了，在 LinuxOS 中的所有操作，可以说都是对文件进行操作，包括使用的一些命令、脚本、甚至视频和图片，都是文件。

在创建文件系统之前，需要先定义三个数据结构。

- 超级块:

```
struct super_block {
   uint32_t magic;                     // 用来标识文件系统类型，支持多文件系统
   ↪  的操作系统通过此标志来识别文件系统类型
   uint32_t sec_cnt;                     // 本分区总共的扇区数
   uint32_t inode_cnt;                     // 本分区中 inode 数量
   uint32_t part_lba_base;             // 本分区的起始 lba 地址

   uint32_t block_bitmap_lba;            // 块位图本身起始扇区地址
   uint32_t block_bitmap_sects;      // 扇区位图本身占用的扇区数量

   uint32_t inode_bitmap_lba;            // i 结点位图起始扇区 lba 地址
   uint32_t inode_bitmap_sects;            // i 结点位图占用的扇区数量

   uint32_t inode_table_lba;             // i 结点表起始扇区 lba 地址
   uint32_t inode_table_sects;             // i 结点表占用的扇区数量

   uint32_t data_start_lba;            // 数据区开始的第一个扇区号
   uint32_t root_inode_no;             // 根目录所在的 I 结点号
   uint32_t dir_entry_size;            // 目录项大小

   uint8_t  pad[460];                    // 加上 460 字节，凑够 512 字节 1 扇区
   ↪  大小
} __attribute__ ((packed));
```

这里的数据块大小用的与扇区大小一致，也就是说 1 扇区 =1 块，因为后续磁盘操作要以扇区为单位，这个数据块其实要不了那么多，所以最后加上 460 字节凑够 512 字节正好为 1 扇区。

- inode 结点:

```
void inode_init(uint32_t inode_no, struct inode* new_inode) {
   new_inode->i_no = inode_no;
   new_inode->i_size = 0;
   new_inode->i_open_cnts = 0;
   new_inode->write_deny = false;

```

```
    /* 初始化块索引数组 i_sector */
    uint8_t sec_idx = 0;
    while (sec_idx < 13) {
    /* i_sectors[12] 为一级间接块地址 */
       new_inode->i_sectors[sec_idx] = 0;
       sec_idx++;
    }
}
```

- 目录项:

```
#define MAX_FILE_NAME_LEN  16
/* 目录结构 */
struct dir {
   struct inode* inode;
   uint32_t dir_pos;          // 记录在目录内的偏移
   uint8_t dir_buf[512];  // 目录的数据缓存
};

/* 目录项结构 */
struct dir_entry {
   char filename[MAX_FILE_NAME_LEN];  // 普通文件或目录名称
   uint32_t i_no;                      // 普通文件或目录对应的 inode 编号
   enum file_types f_type;              // 文件类型
};
```

因为文件名要存储目录项里，所以目录项的大小是固定的，因此文件的名字长度也得有个限定，代码中使用

```
#define MAX_FILE_NAME_LEN  16
```

将其最大长度定义到了 16 个字符。

完成这些工作后，就可以对文件系统[6]进行创建了，代码参见附录 A.13，创建的方法如下：

- 根据分区的容量，计算各分区文件系统元信息所需要的位置和扇区数;
- 往内存里创建超级块，把上一步计算的元信息数据写入超级块;
- 把超级块写入到磁盘里;
- 把元信息写入磁盘上对应的地方;

至此，SheepOS 已经有了基础的一个文件系统，之后对文件的操作都将建立在这个系统之上。

4.5.1 创建文件

在有了文件系统之后，就可以对文件进行相应的处理了，如创建、打开、关闭、删除文件等操作。创建文件的操作只需要再增加一个 file_create 的函数即可，代码参见附录 A.14，其工作过程如下：

- 创建文件, 若成功则返回文件描述符, 否则返回-1;
- 从堆中为 inode 申请内存;
- 同步内存数据到硬盘;

以上就是 file_creat 函数的创建过程，在 main.c 中直接调用该函数即可实现对文件的创建。

4.5.2 文件的打开和关闭

有了文件之后当然就需要对文件进行打开和关闭的操作了，这两个步骤实现的方法也很简单，就是一个 file_open 和 file_close 的函数。代码参见附录 A.15，其工作过程如下：

- 打开编号为 inode_no 的 inode 对应的文件, 若成功则返回文件描述符, 否则返回-1;
- 打开或创建文件成功后, 返回文件描述符, 否则返回-1;
- 关闭文件;
- 将文件描述符转化为文件表的下标;
- 关闭文件描述符 fd 指向的文件, 成功返回 0, 否则返回-1.

之后也是直接在 main.c 中调用这两个函数就可以实现对文件的打开和关闭操作。

4.5.3 文件的写入和读取

写入和读取的代码都很类似，也是两个函数,file_write 和 file_read, 写入和读取都有 3 个参数，分别是:1）文件 File;2）数据缓冲区 Buf;3）字节数 Count; 这三个参数。

函数 file_write 和 file_read 的功能分别是：

- file_write: 把 Buf 里的 Count 个字节写入 File;

```
uint32_t write(int32_t fd, const void* buf, uint32_t count)
{
   return _syscall3(SYS_WRITE, fd, buf, count);
}
```

- `file_read`: 读取 Count 个字节写入 Buf;

```
int32_t read(int32_t fd, void* buf, uint32_t count)
{
   return _syscall3(SYS_READ, fd, buf, count);
}
```

4.5.4 删除文件

删除文件用的是 `unlink` 函数，而删除文件可以理解为与创建文件相反的一个过程，怎么创建的文件就怎么删除，所以会涉及到 inode、位图、目录项、目录 inode 中的 `i_size`、数据块等数据的回收。代码详见附录 A.16, 工作过程如下：

- 回收 inode;
- 接下来是删除目录项;
- 创建 `sys_unlink` 函数。

至此，就完成了文件的删除操作。

4.6 一个简单 shell 的实现

现在 SheepOS 有了一套属于自己的文件系统，但是对这些文件进行操作的话还得在程序里通过调用之后编译才能实现，操作系统本就是为用户服务的，所以用户和操作系统之间的交互尤为重要，Shell 的存在就是为了使用户与操作系统直接能够更直接的进行互动。Shell 的功能大概就是能够收到用户输入的指令，解析输入的字符是内部指令还是外部指令，然后执行与用户输入的指令相关的操作。

- 存储输入的命令:

```
static char cmd_line[MAX_PATH_LEN] = {0};
char final_path[MAX_PATH_LEN] = {0};
```

- 用来记录当前目录, 是当前目录的缓存, 每次执行 `cd` 命令时会更新此内容:

```
char cwd_cache[MAX_PATH_LEN] = {0};
```

- 输出提示符:

```
void print_prompt(void)
{
   printf("[yx@SheepOS %s]$ ", cwd_cache);
}
```

Shell 就像是为用户和操作系统之间搭了一把梯子，有了这个梯子之后用户有什么需求可以直接向操作系统伸出手去索要，而操作系统也能直接把我们想要的东西递交到我们的手中。接下来就向操作系统发出第一个需求: 清屏和清除输入内容。

```
void my_shell(void) {
   cwd_cache[0] = '/';
   while (1) {
      print_prompt();
      memset(final_path, 0, MAX_PATH_LEN);
      memset(cmd_line, 0, MAX_PATH_LEN);
      readline(cmd_line, MAX_PATH_LEN);
      if (cmd_line[0] == 0) {         // 若只键入了一个回车
         continue;
      }

      argc = -1;
      argc = cmd_parse(cmd_line, argv, ' ');
      if (argc == -1) {
         printf("num of arguments exceed %d\n", MAX_ARG_NR);
         continue;
      }
```

4.6.1 Ctrl+L 和 Ctrl+U

这两个组合键需要用到之前提到过的键盘扫描码，代码如下:

- 清屏 Ctrl+L:

```
case 'l' - 'a':
   /* 1 先将当前的字符'l'-'a'置为 0 */
   *pos = 0;
   /* 2 再将屏幕清空 */
   clear();
   /* 3 打印提示符 */
   print_prompt();
   /* 4 将之前键入的内容再次打印 */
   printf("%s", buf);
   break;
```

- 清除输入 Ctrl+U:

```
case 'u' - 'a':
   while (buf != pos) {
      putchar('\b');
      *(pos--) = 0;
   }
   break;
```

繁杂的工作在之前就已经差不多做完了，所以在 Shell 中基本上只需要直接调用之前写好的函数就可以。

4.6.2 **打印目录清单** (ls)

- 利用 Shell 的内部命令建立 ls 函数:

```
void buildin_ls(uint32_t argc, char** argv)
```

- 检测符号"-"，如果是则 ls 后视为参数 (目前只支持-h 和-l):

```
       if (argv[arg_idx][0] == '-')
       {
          if (!strcmp("-l", argv[arg_idx])) {          // 如果是参数-l
             long_info = true;
          } else if (!strcmp("-h", argv[arg_idx])) {   // 参数-h
             printf("usage: -l list all infomation about the file.\n-h for
             ↪  help\nlist all files in the current dirctory if no
             ↪  option\n");
             return;
          } else {
             printf("ls: invalid option %s\nTry `ls -h' for more
             ↪  information.\n", argv[arg_idx]);
             return;
          }
       }
```

- 若 ls 后参数是-l 的话则把 long_info 置为 true, 可以理解为显示文件详细信息的开关:

```
       if (long_info)
       {
          printf("-  %d  %d  %s\n", file_stat.st_ino, file_stat.st_size,
          ↪  pathname);
       } else {
          printf("%s\n", pathname);
       }
```

- 若 ls 后不是参数，则视为路径参数且只能为 1 个，超过做出提示:

```
if (arg_path_nr == 0) {
   pathname = argv[arg_idx];
   arg_path_nr = 1;
} else {
   printf("ls: only support one path\n");
   return;
}
```

- 如果只输入了 ls 或 ls -l 则默认以当前路径处理:

```
    if (pathname == NULL)
    {
       if (NULL != getcwd(final_path, MAX_PATH_LEN)) {
          pathname = final_path;
       } else {
          printf("ls: getcwd for default path failed\n");
          return;
       }
```

```
    } else {
       make_clear_abs_path(pathname, final_path);
       pathname = final_path;
    }
```

- 如果在目录中没有找到指定路径，则提示:

```
if (stat(pathname, &file_stat) == -1)
{
   printf("ls: cannot access %s: No such file or directory\n", pathname);
   return;
}
```

4.6.3 跳转到指定目录 (cd)

- 利用 shell 的内部命令建立 cd 函数:

```
char* buildin_cd(uint32_t argc, char** argv)
```

- 设定参数限制，若参数大于 1 个，则做出提示:

```
if (argc > 2)
{
   printf("cd: only support 1 argument!\n");
   return NULL;
}
```

- 若只键入了 cd 而无参数的话，返回根目录:

```
if (argc == 1)
{
   final_path[0] = '/';
   final_path[1] = 0;
} else {
   make_clear_abs_path(argv[1], final_path);
}
```

- 如果没有找到指定的目录，则做出提示:

```
if (chdir(final_path) == -1)
{
   printf("cd: no such directory %s\n", final_path);
   return NULL;
}
```

4.6.4 创建文件 (mkdir)

- 利用 shell 的内部命令建立 mkdir 函数:

```
int32_t buildin_mkdir(uint32_t argc, char** argv)
```

- 设定参数限制，只能有一个参数就是文件名字:

```
if (argc != 2)
{
   printf("mkdir: only support 1 argument!\n");
}
```

- 创建文件

```
   make_clear_abs_path(argv[1], final_path);
     if (strcmp("/", final_path))
     {
       if (mkdir(final_path) == 0)
         {
            ret = 0;
         } else {
            printf("mkdir: create directory %s failed.\n", argv[1]);
         }
     }
   }
```

argv[1] 表示要创建的文件名,final_path 是文件的路径,strcmp("/",final_path) 用来判断创建的文件是否是在根目录下。

4.6.5 **删除文件** (rmdir)

删除文件与创建文件代码基本相似，毕竟就是一个创建文件的逆过程。

- 利用 shell 的内部命令建立 rm 函数:

```
int32_t buildin_rm(uint32_t argc, char** argv)
```

删除和判断是否在根目录的方法与创建文件相同。

- 调用 unlink 函数释放相应路径里的数据:

```
if (unlink(final_path) == 0)
{
   ret = 0;
} else {
  printf("rm: delete %s failed.\n", argv[1]);
}
```

5 不足与展望

通过对 Linux 操作系统的学习和本次毕业设计实践，让我对计算机的工作原理、操作系统的概念、和系统编程有了更加深刻的了解。不足之处有以下几点：

1. 实现的功能较少，仅有一些简单的对文件的操作。
2. 我觉得接触到的东西虽然已经说是底层的了，但并不是最底层的，像 `nasm`、`gcc`、`ld` 这些指令都是前人写好的，直接拿来用了，以后若是有时间定会研究一下由自己写的编译器来编译自己写的程序。
3. Tab 自动补全的功能，我自己在日常使用中最常用的也就是这个键，这次没有实现还是挺遗憾的。
4. 网络相关的知识，虽然现在感觉离这个层面还很远，但感觉以后若是涉及到这方面的问题一定也会很有趣。

基本上就是以上这些地方感觉要是有时间的话还是能够实现的，所以就带有一丝丝遗憾，由于要准备考试以及一些其他原因，对这次毕业设计的时间其实很紧张，然后操作系统里的很多东西都是很底层的，像汇编、和一些基础的 C 语言[10]学习起来就需要更多的时间去了解相关的知识，最后感觉也只是了解到了一点皮毛，说实话也就是在别人铺好的路上走了一遍，用着别人提前写好的一些框架。

其实在最初定下要研究这个题目的时候，目标是想将显示 PDF 这些功能也加进去，最后能在我自己制作的 SheepOS 进行论文答辩，那肯定是件非常有意义的事，但现实总比理想要残酷的多，感觉还差的很远，以后还需要继续努力，加油。

参考文献

[1] DANIEL P.BOVET M C. 深入理解 LINUX 内核. 东南大学出版社, 2006.

[2] LOVE R. Linux 内核设计与实现. 机械工业出版社, 2011.

[3] 于渊. OrangeS: 一个操作系统的实现. 电子工业出版社, 2009.

[4] 刘遄. Linux 就该这么学. 人民邮电出版社, 2021.

[5] 川合秀实. 30 天自制操作系统. 人民邮电出版社, 2012.

[6] 张书宁. 文件系统技术内幕：大数据时代海量数据存储之道. 电子工业出版社, 2022.

[7] 李忠王晓波余洁. x86 汇编语言: 从实模式到保护模式. 电子工业出版社, 2013.

[8] 王爽. 汇编语言. 清华大学出版社, 2013.

[9] 郑钢. 操作系统真象还原. 人民邮电出版社, 2016.

[10] 霍顿. C 语言入门经典. 清华大学出版社, 2008.

指导教师简介

王晓林，男，50 岁，硕士，讲师，毕业于英国格林尼治大学，分布式计算系统专业。现任西南林业大学计信学院教师。执教 Linux、操作系统、网络技术等方面的课程，有丰富的 Linux 教学和系统管理经验。

致　谢

首先我要感谢对我悉心指导和引导我走进 LinuxOS 世界的指导老师，王晓林老师，如果没有遇到他的话，我甚至可能不会接触 Linux 这个操作系统，感谢他在这方面给予了我很多帮助，在遇到很多 Linux 方面的问题是都是他帮助我解决了，因为有他所以我才能从一位 Windows 用户变到 Linux 用户然后再爱上 Linux。其次，还要感谢两本书的作者，他们分别是《OrangeS: 一个操作系统的实现》的作者于渊和《操作系统真相还原》的作者郑钢，如果没有他们写的这两本书的话，我的这次毕业设计将会困难重重，他们写的书给了我很多帮助，很多地方都解释的非常到位，基本上从头到尾跟着做一遍下来的话都是能够理解的。最后还要感谢期间帮助过我的朋友、同学们，如果没有他们的鼓励和支持的话也许我也早就放弃了，感谢大家对我的帮助。

A 相关代码

A.1 MBR

```
%include "boot.inc"
SECTION MBR vstart=0x7c00
   mov ax,cs
   mov ds,ax
   mov es,ax
   mov ss,ax
   mov fs,ax
   mov sp,0x7c00
   mov ax,0xb800
   mov gs,ax

   mov     ax, 0600h
   mov     bx, 0700h
   mov     cx, 0
   mov     dx, 184fh

   int     10h                         ; int 10h

   mov byte [gs:0x00],'1'
   mov byte [gs:0x01],0xA4

   mov byte [gs:0x02],' '
   mov byte [gs:0x03],0xA4

   mov byte [gs:0x04],'M'
   mov byte [gs:0x05],0xA4

   mov byte [gs:0x06],'B'
   mov byte [gs:0x07],0xA4

   mov byte [gs:0x08],'R'
   mov byte [gs:0x09],0xA4

   mov eax,LOADER_START_SECTOR
   mov bx,LOADER_BASE_ADDR
   mov cx,4
   call rd_disk_m_16

   jmp LOADER_BASE_ADDR + 0x300

rd_disk_m_16:

      mov esi,eax
      mov di,cx

      mov dx,0x1f2
      mov al,cl
      out dx,al

      mov eax,esi
```

```
51
52          mov dx,0x1f3
53          out dx,al
54
55          mov cl,8
56          shr eax,cl
57          mov dx,0x1f4
58          out dx,al
59
60          shr eax,cl
61          mov dx,0x1f5
62          out dx,al
63
64          shr eax,cl
65          and al,0x0f
66          or al,0xe0
67          mov dx,0x1f6
68          out dx,al
69
70          mov dx,0x1f7
71          mov al,0x20
72          out dx,al
73
74      .not_ready:
75
76          nop
77          in al,dx
78          and al,0x88
79          cmp al,0x08
80          jnz .not_ready
81
82          mov ax, di
83          mov dx, 256
84          mul dx
85          mov cx, ax
86
87          mov dx, 0x1f0
88      .go_on_read:
89          in ax,dx
90          mov [bx],ax
91          add bx,2
92          loop .go_on_read
93          ret
94
95       times 510-($-$$) db 0
96       db 0x55,0xaa
```

A.2 Rd Disk

```
1  ;-------------------------------------------------------------------------------
2                                ;功能:读取硬盘n个扇区
3  rd_disk_m_32:
4  ;-------------------------------------------------------------------------------
5                                                        ; eax=LBA扇区号
```

```
                                                    ; ebx=将数据写入的内存
                                                    ↪ 地址
                                                    ; ecx=读入的扇区数
      mov esi,eax           ; 备份 eax
      mov di,cx                   ; 备份扇区数到 di
;读写硬盘:
;第 1 步: 设置要读取的扇区数
      mov dx,0x1f2
      mov al,cl
      out dx,al             ;读取的扇区数

      mov eax,esi           ;恢复 ax

;第 2 步: 将 LBA 地址存入 0x1f3 ~ 0x1f6

      ;LBA 地址 7~0 位写入端口 0x1f3
      mov dx,0x1f3
      out dx,al

      ;LBA 地址 15~8 位写入端口 0x1f4
      mov cl,8
      shr eax,cl
      mov dx,0x1f4
      out dx,al

      ;LBA 地址 23~16 位写入端口 0x1f5
      shr eax,cl
      mov dx,0x1f5
      out dx,al

      shr eax,cl
      and al,0x0f           ;lba 第 24~27 位
      or al,0xe0            ; 设置 7~4 位为 1110,表示 lba 模式
      mov dx,0x1f6
      out dx,al

;第 3 步: 向 0x1f7 端口写入读命令, 0x20
      mov dx,0x1f7
      mov al,0x20
      out dx,al

;;;;;;; 至此,硬盘控制器便从指定的 lba 地址 (eax) 处,读出连续的 cx 个扇区,下面检查硬盘
↪ 状态,不忙就能把这 cx 个扇区的数据读出来

;第 4 步: 检测硬盘状态
  .not_ready:                   ;测试 0x1f7 端口 (status 寄存器) 的的 BSY 位
      ;同一端口,写时表示写入命令字,读时表示读入硬盘状态
      nop
      in al,dx
      and al,0x88           ;第 4 位为 1 表示硬盘控制器已准备好数据传输,第 7 位为 1 表
      ↪ 示硬盘忙
      cmp al,0x08
      jnz .not_ready            ;若未准备好,继续等。

```

```
57 ;第 5 步：从 0x1f0 端口读数据
58       mov ax, di            ;以下从硬盘端口读数据用 insw 指令更快捷，不过尽可能多的
         ↪  演示命令使用，
59                             ;在此先用这种方法，在后面内容会用到 insw 和 outsw 等
60
61       mov dx, 256           ;di 为要读取的扇区数，一个扇区有 512 字节，每次读入一个
         ↪  字，共需 di*512/2 次，所以 di*256
62       mul dx
63       mov cx, ax
64       mov dx, 0x1f0
65   .go_on_read:
66       in ax,dx
67       mov [ebx], ax
68       add ebx, 2
69                             ; 由于在实模式下偏移地址为 16 位，所以用 bx 只会访问到
                               ↪  0~FFFFh 的偏移。
70                             ; loader 的栈指针为 0x900,bx 为指向的数据输出缓冲区，且为
                               ↪  16 位，
71                             ; 超过 0xffff 后，bx 部分会从 0 开始，所以当要读取的扇区数过
                               ↪  大，待写入的地址超过 bx 的范围时，
72                             ; 从硬盘上读出的数据会把 0x0000~0xffff 的覆盖，
73                             ; 造成栈被破坏，所以 ret 返回时，返回地址被破坏了，已经不是
                               ↪  之前正确的地址，
74                             ; 故程序出会错，不知道会跑到哪里去。
75                             ; 所以改为 ebx 代替 bx 指向缓冲区，这样生成的机器码前面会有
                               ↪  0x66 和 0x67 来反转。
76                             ; 0X66 用于反转默认的操作数大小！ 0X67 用于反转默认的寻址
                               ↪  方式。
77                             ; cpu 处于 16 位模式时，会理所当然的认为操作数和寻址都是 16
                               ↪  位，处于 32 位模式时，
78                             ; 也会认为要执行的指令是 32 位。
79                             ; 当我们在其中任意模式下用了另外模式的寻址方式或操作数大
                               ↪  小（姑且认为 16 位模式用 16 位字节操作数，
80                             ; 32 位模式下用 32 字节的操作数）时，编译器会在指令前帮我们
                               ↪  加上 0x66 或 0x67，
81                             ; 临时改变当前 cpu 模式到另外的模式下。
82                             ; 假设当前运行在 16 位模式，遇到 0X66 时，操作数大小变为
                               ↪  32 位。
83                             ; 假设当前运行在 32 位模式，遇到 0X66 时，操作数大小变为
                               ↪  16 位。
84                             ; 假设当前运行在 16 位模式，遇到 0X67 时，寻址方式变为 32 位
                               ↪  寻址
85                             ; 假设当前运行在 32 位模式，遇到 0X67 时，寻址方式变为 16 位
                               ↪  寻址。
86
87       loop .go_on_read
88       ret
```

A.3 BIOS 0x15 interrupt

```
    xor ebx, ebx                        ; 第一次调用时，ebx 值要为 0
    mov edx, 0x534d4150                 ;edx 只赋值一次，循环体中不会改变
    mov di, ards_buf                    ;ards 结构缓冲区
.e820_mem_get_loop:                     ; 循环获取每个 ARDS 内存范围描述结构
    mov eax, 0x0000e820                 ; 执行 int 0x15 后,eax 值变为 0x534d4150, 所以每次执行 int 前都要更新为子功能号。
    mov ecx, 20                         ;ARDS 地址范围描述符结构大小是 20 字节
    int 0x15
    jc .e820_failed_so_try_e801         ; 若 cf 位为 1 则有错误发生，尝试 0xe801 子功能
    add di, cx                          ; 使 di 增加 20 字节指向缓冲区中新的 ARDS 结构位置
    inc word [ards_nr]                  ; 记录 ARDS 数量
    cmp ebx, 0                          ; 若 ebx 为 0 且 cf 不为 1, 这说明 ards 全部返回，当前已是最后一个
    jnz .e820_mem_get_loop

; 在所有 ards 结构中，找出 (base_add_low + length_low) 的最大值，即内存的容量。
    mov cx, [ards_nr]                   ; 遍历每一个 ARDS 结构体, 循环次数是 ARDS 的数量
    mov ebx, ards_buf
    xor edx, edx                        ;edx 为最大的内存容量, 在此先清 0
.find_max_mem_area:                     ; 无须判断 type 是否为 1, 最大的内存块一定是可被使用
    mov eax, [ebx]                      ;base_add_low
    add eax, [ebx+8]                    ;length_low
    add ebx, 20                         ; 指向缓冲区中下一个 ARDS 结构
    cmp edx, eax                        ; 冒泡排序，找出最大,edx 寄存器始终是最大的内存容量
    jge .next_ards
    mov edx, eax                        ;edx 为总内存大小
.next_ards:
    loop .find_max_mem_area
    jmp .mem_get_ok
```

A.4 Page

```
    ; 创建页目录及页表并初始化页内存位图
    call setup_page

    ; 要将描述符表地址及偏移量写入内存 gdt_ptr, 一会用新地址重新加载
    sgdt [gdt_ptr]                  ; 存储到原来 gdt 所有的位置

    ; 将 gdt 描述符中视频段描述符中的段基址+0xc0000000
    mov ebx, [gdt_ptr + 2]
    or dword [ebx + 0x18 + 4], 0xc0000000       ; 视频段是第 3 个段描述符, 每个描述符是 8 字节, 故 0x18。
                                                ; 段描述符的高 4 字节的最高位是段基址的 31~24 位

    ; 将 gdt 的基址加上 0xc0000000 使其成为内核所在的高地址
    add dword [gdt_ptr + 2], 0xc0000000

    add esp, 0xc0000000         ; 将栈指针同样映射到内核地址
```

```
16
17      ; 把页目录地址赋给 cr3
18      mov eax, PAGE_DIR_TABLE_POS
19      mov cr3, eax
20
21      ; 打开 cr0 的 pg 位 (第 31 位)
22      mov eax, cr0
23      or eax, 0x80000000
24      mov cr0, eax
25
26      ;在开启分页后,用 gdt 新的地址重新加载
27      lgdt [gdt_ptr]              ; 重新加载
```

A.5 Kernel init

```
1   kernel_init:
2       xor eax, eax
3       xor ebx, ebx                ;ebx 记录程序头表地址
4       xor ecx, ecx                ;cx 记录程序头表中的 program header 数量
5       xor edx, edx                ;dx 记录 program header 尺寸,即 e_phentsize
6
7       mov dx, [KERNEL_BIN_BASE_ADDR + 42]          ; 偏移文件 42 字节处的属性是
        ↪ e_phentsize,表示 program header 大小
8       mov ebx, [KERNEL_BIN_BASE_ADDR + 28]   ; 偏移文件开始部分 28 字节的地方是
        ↪ e_phoff,表示第 1 个 program header 在文件中的偏移量
9                                           ; 其实该值是 0x34,不过还是谨慎一点,这里
                                            ↪ 来读取实际值
10      add ebx, KERNEL_BIN_BASE_ADDR
11      mov cx, [KERNEL_BIN_BASE_ADDR + 44]    ; 偏移文件开始部分 44 字节的地方是
        ↪ e_phnum,表示有几个 program header
12  .each_segment:
13      cmp byte [ebx + 0], PT_NULL                  ; 若 p_type 等于 PT_NULL,说明此
        ↪ program header 未使用。
14      je .PTNULL
15
16      ;为函数 memcpy 压入参数,参数是从右往左依然压入.函数原型类似于
        ↪ memcpy(dst,src,size)
17      push dword [ebx + 16]                   ; program header 中偏移 16 字节的地方是
        ↪ p_filesz,压入函数 memcpy 的第三个参数:size
18      mov eax, [ebx + 4]                         ; 距程序头偏移量为 4 字节的位置是
        ↪ p_offset
19      add eax, KERNEL_BIN_BASE_ADDR          ; 加上 kernel.bin 被加载到的物理地址,eax
        ↪ 为该段的物理地址
20      push eax                                  ; 压入函数 memcpy 的第二个参数:源地址
21      push dword [ebx + 8]                        ; 压入函数 memcpy 的第一个参数:目
        ↪ 的地址,偏移程序头 8 字节的位置是 p_vaddr, 这就是目的地址
22      call mem_cpy                                ; 调用 mem_cpy 完成段复制
23      add esp,12                                 ; 清理栈中压入的三个参数
24  .PTNULL:
25      add ebx, edx                                ; edx 为 program header 大小,即
        ↪ e_phentsize,在此 ebx 指向下一个 program header
26      loop .each_segment
```

```
27      ret
28
29  ;----------  逐字节拷贝 mem_cpy(dst,src,size) ------------
30  ;输入:栈中三个参数 (dst,src,size)
31  ;输出:无
32  ;---------------------------------------------------------
33  mem_cpy:
34      cld
35      push ebp
36      mov ebp, esp
37      push ecx                    ; rep 指令用到了 ecx, 但 ecx 对于外层段的循环还有用,
        ↪  故先入栈备份
38      mov edi, [ebp + 8]            ; dst
39      mov esi, [ebp + 12]            ; src
40      mov ecx, [ebp + 16]            ; size
41      rep movsb                    ; 逐字节拷贝
42
43      ;恢复环境
44      pop ecx
45      pop ebp
46      ret
```

A.6 Print Put Char

```
1   global put_char
2   put_char:
3       pushad            ;备份 32 位寄存器环境
4       ;需要保证 gs 中为正确的视频段选择子,为保险起见,每次打印时都为 gs 赋值
5       mov ax, SELECTOR_VIDEO                ; 不能直接把立即数送入段寄存器
6       mov gs, ax
7
8   ;;;;;;;;;  获取当前光标位置 ;;;;;;;;;
9       ;先获得高 8 位
10      mov dx, 0x03d4  ;索引寄存器
11      mov al, 0x0e            ;用于提供光标位置的高 8 位
12      out dx, al
13      mov dx, 0x03d5  ;通过读写数据端口 0x3d5 来获得或设置光标位置
14      in al, dx            ;得到了光标位置的高 8 位
15      mov ah, al
16
17      ;再获取低 8 位
18      mov dx, 0x03d4
19      mov al, 0x0f
20      out dx, al
21      mov dx, 0x03d5
22      in al, dx
23
24      ;将光标存入 bx
25      mov bx, ax
26      ;下面这行是在栈中获取待打印的字符
27      mov ecx, [esp + 36]                ;pushad 压入 4*8 = 32 字节,加上主调函数的返回地
        ↪  址 4 字节,故 esp+36 字节
28      cmp cl, 0xd                                    ;CR 是 0x0d,LF 是 0x0a
```

```
29      jz .is_carriage_return
30      cmp cl, 0xa
31      jz .is_line_feed
32
33      cmp cl, 0x8                                     ;BS(backspace) 的 asc 码是 8
34      jz .is_backspace
35      jmp .put_other
36  ;;;;;;;;;;;;;;;;;;
37
38   .is_backspace:
39  ;;;;;;;;;;;;        backspace 的一点说明              ;;;;;;;;;;
40  ; 当为 backspace 时，本质上只要将光标移向前一个显存位置即可. 后面再输入的字符自然会覆
    ↪  盖此处的字符
41  ; 但有可能在键入 backspace 后并不再键入新的字符，这时在光标已经向前移动到待删除的字符
    ↪  位置，但字符还在原处，
42  ; 这就显得好怪异，所以此处添加了空格或空字符 0
43      dec bx
44      shl bx,1
45      mov byte [gs:bx], 0x20                    ; 将待删除的字节补为 0 或空格皆可
46      inc bx
47      mov byte [gs:bx], 0x07
48      shr bx,1
49      jmp .set_cursor
50  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
51
52   .put_other:
53      shl bx, 1                                    ; 光标位置是用 2 字节表示，将光标值乘
        ↪  2，表示对应显存中的偏移字节
54      mov [gs:bx], cl                             ; ascii 字符本身
55      inc bx
56      mov byte [gs:bx],0x07                      ; 字符属性
57      shr bx, 1                                     ; 恢复老的光标值
58      inc bx                                     ; 下一个光标值
59      cmp bx, 2000
60      jl .set_cursor                             ; 若光标值小于 2000，表示未写到显存的最
        ↪  后，则去设置新的光标值
61                                                 ; 若超出屏幕字符数大小 (2000) 则换行处理
62   .is_line_feed:                                      ; 是换行符 LF(\n)
63   .is_carriage_return:                              ; 是回车符 CR(\r)
64                                                 ; 如果是 CR(\r)，只要把光标移到行首就
                                                   ↪  行了。
65      xor dx, dx                                      ; dx 是被除数的高 16 位，清 0.
66      mov ax, bx                                      ; ax 是被除数的低 16 位.
67      mov si, 80                                      ; 由于是效仿 linux，linux 中 \n 便表
        ↪  示下一行的行首，所以本系统中，
68      div si                                     ; 把 \n 和 \r 都处理为 linux 中 \n 的意思，
        ↪  也就是下一行的行首。
69      sub bx, dx                                      ; 光标值减去除 80 的余数便是取整
70                                                 ; 以上 4 行处理 \r 的代码
71
72   .is_carriage_return_end:                      ; 回车符 CR 处理结束
73      add bx, 80
74      cmp bx, 2000
```

```
75   .is_line_feed_end:                                ; 若是 LF(\n),将光标移+80 便可。
76     jl .set_cursor
77
78   ;屏幕行范围是 0~24,滚屏的原理是将屏幕的 1~24 行搬运到 0~23 行,再将第 24 行用空格填充
79   .roll_screen:                                     ; 若超出屏幕大小,开始滚屏
80     cld
81     mov ecx, 960                                    ; 一共有 2000-80=1920 个字符要搬
       ↪  运,共 1920*2=3840 字节.一次搬 4 字节,共 3840/4=960 次
82     mov esi, 0xc00b80a0                             ; 第 1 行行首
83     mov edi, 0xc00b8000                             ; 第 0 行行首
84     rep movsd
85
86   ;;;;;;;将最后一行填充为空白
87     mov ebx, 3840                                 ; 最后一行首字符的第一个字节偏移= 1920
       ↪  * 2
88     mov ecx, 80                                     ;一行是 80 字符 (160 字节),每次清
       ↪  理 1 字符 (2 字节),一行需要移动 80 次
89   .cls:
90     mov word [gs:ebx], 0x0720                     ;0x0720 是黑底白字的空格键
91     add ebx, 2
92     loop .cls
93     mov bx,1920                                     ;将光标值重置为 1920,最后一行的首
       ↪  字符.
94
95   .set_cursor:
96                                                   ;将光标设为 bx 值
97   ;;;;;;; 1 先设置高 8 位 ;;;;;;;;
98     mov dx, 0x03d4                                 ;索引寄存器
99     mov al, 0x0e                                        ;用于提供光标位置的高 8 位
100    out dx, al
101    mov dx, 0x03d5                                 ;通过读写数据端口 0x3d5 来获得或设置光
       ↪  标位置
102    mov al, bh
103    out dx, al
104
105  ;;;;;;; 2 再设置低 8 位 ;;;;;;;;;
106    mov dx, 0x03d4
107    mov al, 0x0f
108    out dx, al
109    mov dx, 0x03d5
110    mov al, bl
111    out dx, al
112  .put_char_done:
113    popad
114    ret
115
116  global cls_screen
117  cls_screen:
118    pushad
119    ;;;;;;;;;;;;;;;;
120         ; 由于用户程序的 cpl 为 3,显存段的 dpl 为 0,故用于显存段的选择子 gs 在低于自己
           ↪  特权的环境中为 0,
121         ; 导致用户程序再次进入中断后,gs 为 0,故直接在 put_str 中每次都为 gs 赋值.
```

```
122     mov ax, SELECTOR_VIDEO                  ; 不能直接把立即数送入 gs，须由 ax 中转
123     mov gs, ax
124
125     mov ebx, 0
126     mov ecx, 80*25
127   .cls:
128     mov word [gs:ebx], 0x0720                ;0x0720 是黑底白字的空格键
129     add ebx, 2
130     loop .cls
131     mov ebx, 0
132
133   .set_cursor:                               ;直接把 set_cursor 搬过来用，省事
134  ;;;;;;; 1 先设置高 8 位 ;;;;;;;;
135     mov dx, 0x03d4                          ;索引寄存器
136     mov al, 0x0e                              ;用于提供光标位置的高 8 位
137     out dx, al
138     mov dx, 0x03d5                          ;通过读写数据端口 0x3d5 来获得或设置光
        ↪ 标位置
139     mov al, bh
140     out dx, al
141
142  ;;;;;;; 2 再设置低 8 位 ;;;;;;;;;
143     mov dx, 0x03d4
144     mov al, 0x0f
145     out dx, al
146     mov dx, 0x03d5
147     mov al, bl
148     out dx, al
149     popad
150     ret
```

A.7 Print Put Int

```
1   global put_int
2   put_int:
3     pushad
4     mov ebp, esp
5     mov eax, [ebp+4*9]                        ; call 的返回地址占 4 字节+pushad 的 8
      ↪ 个 4 字节
6     mov edx, eax
7     mov edi, 7                              ; 指定在 put_int_buffer 中初始的偏移量
8     mov ecx, 8                                ; 32 位数字中，16 进制数字的位数是 8 个
9     mov ebx, put_int_buffer
10
11  ;将 32 位数字按照 16 进制的形式从低位到高位逐个处理，共处理 8 个 16 进制数字
12  .16based_4bits:                             ; 每 4 位二进制是 16 进制数字的 1 位，
    ↪ 遍历每一位 16 进制数字
13    and edx, 0x0000000F                       ; 解析 16 进制数字的每一位。and 与操作
      ↪ 后，edx 只有低 4 位有效
14    cmp edx, 9                              ; 数字 0~9 和 a~f 需要分别处理成对应的
      ↪ 字符
15    jg .is_A2F
```

```
16      add edx, '0'                                    ; ascii 码是 8 位大小。add 求和操作
        ↪ 后,edx 低 8 位有效。
17      jmp .store
18  .is_A2F:
19      sub edx, 10                                     ; A~F 减去 10 所得到的差,再加上字符 A
        ↪ 的 ascii 码,便是 A~F 对应的 ascii 码
20      add edx, 'A'
21
22  ;将每一位数字转换成对应的字符后,按照类似“大端”的顺序存储到缓冲区 put_int_buffer
23  ;高位字符放在低地址,低位字符要放在高地址,这样和大端字节序类似,只不过咱们这里是字
    ↪ 符序.
24  .store:
25  ; 此时 dl 中应该是数字对应的字符的 ascii 码
26      mov [ebx+edi], dl
27      dec edi
28      shr eax, 4
29      mov edx, eax
30      loop .16based_4bits
31
32  ;现在 put_int_buffer 中已全是字符,打印之前,
33  ;把高位连续的字符去掉,比如把字符 000123 变成 123
34  .ready_to_print:
35      inc edi                                     ; 此时 edi 退减为-1(0xffffffff),加 1 使
        ↪ 其为 0
36  .skip_prefix_0:
37      cmp edi,8                                     ; 若已经比较第 9 个字符了,表示待打印的
        ↪ 字符串为全 0
38      je .full0
39  ;找出连续的 0 字符, edi 做为非 0 的最高位字符的偏移
40  .go_on_skip:
41      mov cl, [put_int_buffer+edi]
42      inc edi
43      cmp cl, '0'
44      je .skip_prefix_0                             ; 继续判断下一位字符是否为字符 0(不是
        ↪ 数字 0)
45      dec edi                                     ;edi 在上面的 inc 操作中指向了下一个字符,
        ↪ 若当前字符不为'0',要恢复 edi 指向当前字符
46      jmp .put_each_num
47
48  .full0:
49      mov cl,'0'                                     ; 输入的数字为全 0 时,则只打印 0
50  .put_each_num:
51      push ecx                                    ; 此时 cl 中为可打印的字符
52      call put_char
53      add esp, 4
54      inc edi                                     ; 使 edi 指向下一个字符
55      mov cl, [put_int_buffer+edi]                    ; 获取下一个字符到 cl 寄存器
56      cmp edi,8
57      jl .put_each_num
58      popad
59      ret
```

A.8 Interrupt

```
    mov al,0x20                     ; 中断结束命令 EOI
    out 0xa0,al                     ; 向从片发送
    out 0x20,al                     ; 向主片发送

    push %1                           ; 不管 idt_table 中的目标程序是否需要参数，都一律压入中断向量号，调试时很方便
    call [idt_table + %1*4]         ; 调用 idt_table 中的 C 版本中断处理函数
    jmp intr_exit

section .data
    dd    intr%1entry           ; 存储各个中断入口程序的地址，形成 intr_entry_table 数组
%endmacro

section .text
global intr_exit
intr_exit:
; 以下是恢复上下文环境
    add esp, 4                              ; 跳过中断号
    popad
    pop gs
    pop fs
    pop es
    pop ds
    add esp, 4                              ; 跳过 error_code
    iretd
```

A.9 Idt Table

```
/* idt_table 数组中的函数是在进入中断后根据中断向量号调用的，
 * 见 kernel/kernel.S 的 call [idt_table + %1*4] */
      idt_table[i] = general_intr_handler;                    // 默认为 general_intr_handler。

                                                             // 以后会由 register_handler 来注册具体处理函数。

      intr_name[i] = "unknown";                                  // 先统一赋值为 unknown
   }
   intr_name[0] = "#DE Divide Error";
   intr_name[1] = "#DB Debug Exception";
   intr_name[2] = "NMI Interrupt";
   intr_name[3] = "#BP Breakpoint Exception";
   intr_name[4] = "#OF Overflow Exception";
   intr_name[5] = "#BR BOUND Range Exceeded Exception";
   intr_name[6] = "#UD Invalid Opcode Exception";
   intr_name[7] = "#NM Device Not Available Exception";
   intr_name[8] = "#DF Double Fault Exception";
   intr_name[9] = "Coprocessor Segment Overrun";
   intr_name[10] = "#TS Invalid TSS Exception";
```

```
18      intr_name[11] = "#NP Segment Not Present";
19      intr_name[12] = "#SS Stack Fault Exception";
20      intr_name[13] = "#GP General Protection Exception";
21      intr_name[14] = "#PF Page-Fault Exception";
22      // intr_name[15] 第 15 项是 intel 保留项，未使用
23      intr_name[16] = "#MF x87 FPU Floating-Point Error";
24      intr_name[17] = "#AC Alignment Check Exception";
25      intr_name[18] = "#MC Machine-Check Exception";
26      intr_name[19] = "#XF SIMD Floating-Point Exception";
27
28  }
```

A.10 keyboard

```
1   #define        KB_IN_BYTES        32        /* size of keyboard input buffer */
2   #define MAP_COLS        3        /* Number of columns in keymap */
3   #define NR_SCAN_CODES        0x80        /* Number of scan codes (rows in
    ↪  keymap) */
4
5   #define FLAG_BREAK        0x0080              /* Break
    ↪  Code                        */
6   #define FLAG_EXT        0x0100              /* Normal function
    ↪  keys                */
7   #define FLAG_SHIFT_L        0x0200              /* Shift
    ↪  key                        */
8   #define FLAG_SHIFT_R        0x0400              /* Shift
    ↪  key                        */
9   #define FLAG_CTRL_L        0x0800              /* Control
    ↪  key                        */
10  #define FLAG_CTRL_R        0x1000              /* Control
    ↪  key                        */
11  #define FLAG_ALT_L        0x2000              /* Alternate
    ↪  key                */
12  #define FLAG_ALT_R        0x4000              /* Alternate
    ↪  key                */
13  #define FLAG_PAD        0x8000              /* keys in num
    ↪  pad                */
14
15  #define MASK_RAW        0x01FF              /* raw key value = code passed to
    ↪  tty & MASK_RAW
16                                              the value can be found either in the
                                                ↪  keymap column 0
17                                              or in the list below */
18
19  /* Special keys */
20  #define ESC                (0x01 + FLAG_EXT)        /* Esc                */
21  #define TAB                (0x02 + FLAG_EXT)        /* Tab                */
22  #define ENTER                (0x03 + FLAG_EXT)        /* Enter        */
23  #define BACKSPACE        (0x04 + FLAG_EXT)        /* BackSpace        */
24
25  #define GUI_L                (0x05 + FLAG_EXT)        /* L GUI        */
26  #define GUI_R                (0x06 + FLAG_EXT)        /* R GUI        */
27  #define APPS                (0x07 + FLAG_EXT)        /* APPS        */
28
29  /* Shift, Ctrl, Alt */
```

```
#define SHIFT_L                 (0x08 + FLAG_EXT)       /* L Shift        */
#define SHIFT_R                 (0x09 + FLAG_EXT)       /* R Shift        */
#define CTRL_L                  (0x0A + FLAG_EXT)       /* L Ctrl         */
#define CTRL_R                  (0x0B + FLAG_EXT)       /* R Ctrl         */
#define ALT_L                   (0x0C + FLAG_EXT)       /* L Alt          */
#define ALT_R                   (0x0D + FLAG_EXT)       /* R Alt          */

/* Lock keys */
#define CAPS_LOCK       (0x0E + FLAG_EXT)       /* Caps Lock      */
#define       NUM_LOCK        (0x0F + FLAG_EXT)       /* Number Lock        */
#define SCROLL_LOCK       (0x10 + FLAG_EXT)       /* Scroll Lock        */

/* Function keys */
#define F1                  (0x11 + FLAG_EXT)       /* F1                 */
#define F2                  (0x12 + FLAG_EXT)       /* F2                 */
#define F3                  (0x13 + FLAG_EXT)       /* F3                 */
#define F4                  (0x14 + FLAG_EXT)       /* F4                 */
#define F5                  (0x15 + FLAG_EXT)       /* F5                 */
#define F6                  (0x16 + FLAG_EXT)       /* F6                 */
#define F7                  (0x17 + FLAG_EXT)       /* F7                 */
#define F8                  (0x18 + FLAG_EXT)       /* F8                 */
#define F9                  (0x19 + FLAG_EXT)       /* F9                 */
#define F10                 (0x1A + FLAG_EXT)       /* F10                */
#define F11                 (0x1B + FLAG_EXT)       /* F11                */
#define F12                 (0x1C + FLAG_EXT)       /* F12                */

/* Control Pad */
#define PRINTSCREEN       (0x1D + FLAG_EXT)       /* Print Screen       */
#define PAUSEBREAK        (0x1E + FLAG_EXT)       /* Pause/Break        */
#define INSERT              (0x1F + FLAG_EXT)       /* Insert         */
#define DELETE              (0x20 + FLAG_EXT)       /* Delete         */
#define HOME                (0x21 + FLAG_EXT)       /* Home               */
#define END                 (0x22 + FLAG_EXT)       /* End                */
#define PAGEUP              (0x23 + FLAG_EXT)       /* Page Up        */
#define PAGEDOWN        (0x24 + FLAG_EXT)       /* Page Down      */
#define UP                  (0x25 + FLAG_EXT)       /* Up                 */
#define DOWN                (0x26 + FLAG_EXT)       /* Down               */
#define LEFT                (0x27 + FLAG_EXT)       /* Left               */
#define RIGHT               (0x28 + FLAG_EXT)       /* Right          */

/* ACPI keys */
#define POWER               (0x29 + FLAG_EXT)       /* Power          */
#define SLEEP               (0x2A + FLAG_EXT)       /* Sleep          */
#define WAKE                (0x2B + FLAG_EXT)       /* Wake Up        */

/* Num Pad */
#define PAD_SLASH       (0x2C + FLAG_EXT)       /* /              */
#define PAD_STAR        (0x2D + FLAG_EXT)       /* *              */
#define PAD_MINUS       (0x2E + FLAG_EXT)       /* -              */
#define PAD_PLUS        (0x2F + FLAG_EXT)       /* +              */
#define PAD_ENTER       (0x30 + FLAG_EXT)       /* Enter      */
#define PAD_DOT             (0x31 + FLAG_EXT)       /* .              */
#define PAD_0               (0x32 + FLAG_EXT)       /* 0              */
#define PAD_1               (0x33 + FLAG_EXT)       /* 1              */
#define PAD_2               (0x34 + FLAG_EXT)       /* 2              */
#define PAD_3               (0x35 + FLAG_EXT)       /* 3              */
```

```
86  #define PAD_4                (0x36 + FLAG_EXT)       /* 4                */
87  #define PAD_5                (0x37 + FLAG_EXT)       /* 5                */
88  #define PAD_6                (0x38 + FLAG_EXT)       /* 6                */
89  #define PAD_7                (0x39 + FLAG_EXT)       /* 7                */
90  #define PAD_8                (0x3A + FLAG_EXT)       /* 8                */
91  #define PAD_9                (0x3B + FLAG_EXT)       /* 9                */
92  #define PAD_UP                PAD_8                      /*
    ↪  Up                */
93  #define PAD_DOWN        PAD_2                    /* Down                */
94  #define PAD_LEFT        PAD_4                    /* Left                */
95  #define PAD_RIGHT        PAD_6                    /* Right        */
96  #define PAD_HOME        PAD_7                    /* Home                */
97  #define PAD_END                PAD_1                      /*
    ↪  End                */
98  #define PAD_PAGEUP        PAD_9                    /* Page Up        */
99  #define PAD_PAGEDOWN        PAD_3                    /* Page Down        */
100 #define PAD_INS                PAD_0                      /*
    ↪  Ins                */
101 #define PAD_MID                PAD_5                      /* Middle
    ↪  key        */
102 #define PAD_DEL                PAD_DOT                    /*
    ↪  Del                */
```

A.11 process

```
1   svoid page_dir_activate(struct task_struct* p_thread) {
2   /************************************************************
3    * 执行此函数时，当前任务可能是线程。
4    * 之所以对线程也要重新安装页表，原因是上一次被调度的可能是进程，
5    * 否则不恢复页表的话，线程就会使用进程的页表了。
6    ************************************************************/
7
8   /* 若为内核线程，需要重新填充页表为 0x100000 */
9      uint32_t pagedir_phy_addr = 0x100000;  // 默认为内核的页目录物理地址，也就是内
       ↪  核线程所用的页目录表
10     if (p_thread->pgdir != NULL)        {    // 用户态进程有自己的页目录表
11        pagedir_phy_addr = addr_v2p((uint32_t)p_thread->pgdir);
12     }
13
14     /* 更新页目录寄存器 cr3，使新页表生效 */
15     asm volatile ("movl %0, %%cr3" : : "r" (pagedir_phy_addr) : "memory");
16  }
17
18  /* 击活线程或进程的页表，更新 tss 中的 esp0 为进程的特权级 0 的栈 */
19  void process_activate(struct task_struct* p_thread) {
20     ASSERT(p_thread != NULL);
21     /* 击活该进程或线程的页表 */
22     page_dir_activate(p_thread);
23
24     /* 内核线程特权级本身就是 0，处理器进入中断时并不会从 tss 中获取 0 特权级栈地址，故
       ↪  不需要更新 esp0 */
25     if (p_thread->pgdir) {
26        /* 更新该进程的 esp0，用于此进程被中断时保留上下文 */
27        update_tss_esp(p_thread);
```

```
28      }
29  }
30
31  /* 创建页目录表,将当前页表的表示内核空间的 pde 复制,
32   * 成功则返回页目录的虚拟地址,否则返回-1 */
33  uint32_t* create_page_dir(void) {
34
35      /* 用户进程的页表不能让用户直接访问到,所以在内核空间来申请 */
36      uint32_t* page_dir_vaddr = get_kernel_pages(1);
37      if (page_dir_vaddr == NULL) {
38          console_put_str("create_page_dir: get_kernel_page failed!");
39          return NULL;
40      }
41
42  /************************** 1  先复制页表
    ↪  ***********************************/
43      /*  page_dir_vaddr + 0x300*4 是内核页目录的第 768 项 */
44      memcpy((uint32_t*)((uint32_t)page_dir_vaddr + 0x300*4),
        ↪  (uint32_t*)(0xfffff000+0x300*4), 1024);
45  /*****************************************************************************/
46
47  /************************** 2  更新页目录地址
    ↪  **********************************/
48      uint32_t new_page_dir_phy_addr = addr_v2p((uint32_t)page_dir_vaddr);
49      /* 页目录地址是存入在页目录的最后一项,更新页目录地址为新页目录的物理地址 */
50      page_dir_vaddr[1023] = new_page_dir_phy_addr | PG_US_U | PG_RW_W | PG_P_1;
51  /*****************************************************************************/
52      return page_dir_vaddr;
53  }
54
55  /* 创建用户进程虚拟地址位图 */
56  void create_user_vaddr_bitmap(struct task_struct* user_prog) {
57      user_prog->userprog_vaddr.vaddr_start = USER_VADDR_START;
58      uint32_t bitmap_pg_cnt = DIV_ROUND_UP((0xc0000000 - USER_VADDR_START) /
        ↪  PG_SIZE / 8 , PG_SIZE);
59      user_prog->userprog_vaddr.vaddr_bitmap.bits =
        ↪  get_kernel_pages(bitmap_pg_cnt);
60      user_prog->userprog_vaddr.vaddr_bitmap.btmp_bytes_len = (0xc0000000 -
        ↪  USER_VADDR_START) / PG_SIZE / 8;
61      bitmap_init(&user_prog->userprog_vaddr.vaddr_bitmap);
62  }
63
64  /* 创建用户进程 */
65  void process_execute(void* filename, char* name) {
66      /* pcb 内核的数据结构,由内核来维护进程信息,因此要在内核内存池中申请 */
67      struct task_struct* thread = get_kernel_pages(1);
68      init_thread(thread, name, default_prio);
69      create_user_vaddr_bitmap(thread);
70      thread_create(thread, start_process, filename);
71      thread->pgdir = create_page_dir();
72      block_desc_init(thread->u_block_desc);
73
74      enum intr_status old_status = intr_disable();
75      ASSERT(!elem_find(&thread_ready_list, &thread->general_tag));
76      list_append(&thread_ready_list, &thread->general_tag);
```

```
77
78      ASSERT(!elem_find(&thread_all_list, &thread->all_list_tag));
79      list_append(&thread_all_list, &thread->all_list_tag);
80      intr_set_status(old_status);
81  }
```

A.12 Schedule

```
1   /* 时钟的中断处理函数 */
2   static void intr_timer_handler(void) {
3      struct task_struct* cur_thread = running_thread();
4
5      ASSERT(cur_thread->stack_magic == 0x19870916);         // 检查栈是否溢出
6
7      cur_thread->elapsed_ticks++;          // 记录此线程占用的 cpu 时间嘀
8      ticks++;          //从内核第一次处理时间中断后开始至今的滴哒数,内核态和用户态总
       ↪  共的嘀哒数
9
10     if (cur_thread->ticks == 0) {          // 若进程时间片用完就开始调度新的进程上
       ↪  cpu
11        schedule();
12     } else {                                  // 将当前进程的时间片-1
13        cur_thread->ticks--;
14     }
15  }
16
17  /* 实现任务调度 */
18  void schedule() {
19     ASSERT(intr_get_status() == INTR_OFF);
20
21     struct task_struct* cur = running_thread();
22     if (cur->status == TASK_RUNNING) { // 若此线程只是 cpu 时间片到了,将其加入到就
       ↪  绪队列尾
23        ASSERT(!elem_find(&thread_ready_list, &cur->general_tag));
24        list_append(&thread_ready_list, &cur->general_tag);
25        cur->ticks = cur->priority;     // 重新将当前线程的 ticks 再重置为其
          ↪  priority;
26        cur->status = TASK_READY;
27     } else {
28        /* 若此线程需要某事件发生后才能继续上 cpu 运行,
29        不需要将其加入队列,因为当前线程不在就绪队列中。*/
30     }
31
32     /* 如果就绪队列中没有可运行的任务,就唤醒 idle */
33     if (list_empty(&thread_ready_list)) {
34        thread_unblock(idle_thread);
35     }
36
37     ASSERT(!list_empty(&thread_ready_list));
38     thread_tag = NULL;          // thread_tag 清空
39  /* 将 thread_ready_list 队列中的第一个就绪线程弹出,准备将其调度上 cpu. */
40     thread_tag = list_pop(&thread_ready_list);
```

```
    struct task_struct* next = elem2entry(struct task_struct, general_tag,
    ↪  thread_tag);
    next->status = TASK_RUNNING;

    /* 击活任务页表等 */
    process_activate(next);

    switch_to(cur, next);
}

```

A.13 File System

```
/* 格式化分区,也就是初始化分区的元信息,创建文件系统 */
static void partition_format(struct partition* part) {
/* 为方便实现,一个块大小是一扇区 */
    uint32_t boot_sector_sects = 1;
    uint32_t super_block_sects = 1;
    uint32_t inode_bitmap_sects = DIV_ROUND_UP(MAX_FILES_PER_PART,
    ↪  BITS_PER_SECTOR);           // I结点位图占用的扇区数.最多支持4096个文件
    uint32_t inode_table_sects = DIV_ROUND_UP(((sizeof(struct inode) *
    ↪  MAX_FILES_PER_PART)), SECTOR_SIZE);
    uint32_t used_sects = boot_sector_sects + super_block_sects +
    ↪  inode_bitmap_sects + inode_table_sects;
    uint32_t free_sects = part->sec_cnt - used_sects;

/************** 简单处理块位图占据的扇区数 ***************/
    uint32_t block_bitmap_sects;
    block_bitmap_sects = DIV_ROUND_UP(free_sects, BITS_PER_SECTOR);
    /* block_bitmap_bit_len是位图中位的长度,也是可用块的数量 */
    uint32_t block_bitmap_bit_len = free_sects - block_bitmap_sects;
    block_bitmap_sects = DIV_ROUND_UP(block_bitmap_bit_len, BITS_PER_SECTOR);
/*****************************************************/

    /* 超级块初始化 */
    struct super_block sb;
    sb.magic = 0x19590318;
    sb.sec_cnt = part->sec_cnt;
    sb.inode_cnt = MAX_FILES_PER_PART;
    sb.part_lba_base = part->start_lba;

    sb.block_bitmap_lba = sb.part_lba_base + 2;         // 第0块是引导块,第1块是
    ↪  超级块
    sb.block_bitmap_sects = block_bitmap_sects;

    sb.inode_bitmap_lba = sb.block_bitmap_lba + sb.block_bitmap_sects;
    sb.inode_bitmap_sects = inode_bitmap_sects;

    sb.inode_table_lba = sb.inode_bitmap_lba + sb.inode_bitmap_sects;
    sb.inode_table_sects = inode_table_sects;

    sb.data_start_lba = sb.inode_table_lba + sb.inode_table_sects;
    sb.root_inode_no = 0;
    sb.dir_entry_size = sizeof(struct dir_entry);
```

```
38
39      printk("%s info:\n", part->name);
40      printk("   magic:0x%x\n   part_lba_base:0x%x\n   all_sectors:0x%x\n
        ↪  inode_cnt:0x%x\n   block_bitmap_lba:0x%x\n   block_bitmap_sectors:0x%x\n
        ↪  inode_bitmap_lba:0x%x\n   inode_bitmap_sectors:0x%x\n
        ↪  inode_table_lba:0x%x\n   inode_table_sectors:0x%x\n
        ↪  data_start_lba:0x%x\n", sb.magic, sb.part_lba_base, sb.sec_cnt,
        ↪  sb.inode_cnt, sb.block_bitmap_lba, sb.block_bitmap_sects,
        ↪  sb.inode_bitmap_lba, sb.inode_bitmap_sects, sb.inode_table_lba,
        ↪  sb.inode_table_sects, sb.data_start_lba);
41
42      struct disk* hd = part->my_disk;
43    /*******************************
44     * 1 将超级块写入本分区的 1 扇区 *
45     *******************************/
46      ide_write(hd, part->start_lba + 1, &sb, 1);
47      printk("   super_block_lba:0x%x\n", part->start_lba + 1);
48
49    /* 找出数据量最大的元信息，用其尺寸做存储缓冲区*/
50      uint32_t buf_size = (sb.block_bitmap_sects >= sb.inode_bitmap_sects ?
        ↪  sb.block_bitmap_sects : sb.inode_bitmap_sects);
51      buf_size = (buf_size >= sb.inode_table_sects ? buf_size :
        ↪  sb.inode_table_sects) * SECTOR_SIZE;
52      uint8_t* buf = (uint8_t*)sys_malloc(buf_size);        // 申请的内存由内存管理
        ↪  系统清 0 后返回
53
54    /*************************************
55     * 2 将块位图初始化并写入 sb.block_bitmap_lba *
56     *************************************/
57      /* 初始化块位图 block_bitmap */
58      buf[0] |= 0x01;       // 第 0 个块预留给根目录，位图中先占位
59      uint32_t block_bitmap_last_byte = block_bitmap_bit_len / 8;
60      uint8_t  block_bitmap_last_bit  = block_bitmap_bit_len % 8;
61      uint32_t last_size = SECTOR_SIZE - (block_bitmap_last_byte %
        ↪  SECTOR_SIZE);             // last_size 是位图所在最后一个扇区中，不足一扇区
        ↪  的其余部分
62
63      /* 1 先将位图最后一字节到其所在的扇区的结束全置为 1，即超出实际块数的部分直接置为
        ↪  已占用*/
64      memset(&buf[block_bitmap_last_byte], 0xff, last_size);
65
66      /* 2 再将上一步中覆盖的最后一字节内的有效位重新置 0 */
67      uint8_t bit_idx = 0;
68      while (bit_idx <= block_bitmap_last_bit) {
69         buf[block_bitmap_last_byte] &= ~(1 << bit_idx++);
70      }
71      ide_write(hd, sb.block_bitmap_lba, buf, sb.block_bitmap_sects);
72
73    /*************************************
74     * 3 将 inode 位图初始化并写入 sb.inode_bitmap_lba *
75     *************************************/
76      /* 先清空缓冲区*/
77      memset(buf, 0, buf_size);
78      buf[0] |= 0x1;       // 第 0 个 inode 分给了根目录
```

```
79      /* 由于 inode_table 中共 4096 个 inode, 位图 inode_bitmap 正好占用 1 扇区,
80       * 即 inode_bitmap_sects 等于 1, 所以位图中的位全都代表 inode_table 中的 inode,
81       * 无须再像 block_bitmap 那样单独处理最后一扇区的剩余部分,
82       * inode_bitmap 所在的扇区中没有多余的无效位 */
83      ide_write(hd, sb.inode_bitmap_lba, buf, sb.inode_bitmap_sects);
84
85  /**************************************
86   * 4 将 inode 数组初始化并写入 sb.inode_table_lba *
87   **************************************/
88   /* 准备写 inode_table 中的第 0 项, 即根目录所在的 inode */
89      memset(buf, 0, buf_size);  // 先清空缓冲区 buf
90      struct inode* i = (struct inode*)buf;
91      i->i_size = sb.dir_entry_size * 2;         // . 和 ..
92      i->i_no = 0;   // 根目录占 inode 数组中第 0 个 inode
93      i->i_sectors[0] = sb.data_start_lba;             // 由于上面的
        ↪  memset,i_sectors 数组的其它元素都初始化为 0
94      ide_write(hd, sb.inode_table_lba, buf, sb.inode_table_sects);
95
96  /**************************************
97   * 5 将根目录初始化并写入 sb.data_start_lba
98   **************************************/
99      /* 写入根目录的两个目录项 . 和 .. */
100     memset(buf, 0, buf_size);
101     struct dir_entry* p_de = (struct dir_entry*)buf;
102
103     /* 初始化当前目录"." */
104     memcpy(p_de->filename, ".", 1);
105     p_de->i_no = 0;
106     p_de->f_type = FT_DIRECTORY;
107     p_de++;
108
109     /* 初始化当前目录父目录".." */
110     memcpy(p_de->filename, "..", 2);
111     p_de->i_no = 0;   // 根目录的父目录依然是根目录自己
112     p_de->f_type = FT_DIRECTORY;
113
114     /* sb.data_start_lba 已经分配给了根目录, 里面是根目录的目录项 */
115     ide_write(hd, sb.data_start_lba, buf, 1);
116
117     printk("   root_dir_lba:0x%x\n", sb.data_start_lba);
118     printk("%s format done\n", part->name);
119     sys_free(buf);
120 }
```

A.14 File Create

```
1   /* 创建文件, 若成功则返回文件描述符, 否则返回-1 */
2   int32_t file_create(struct dir* parent_dir, char* filename, uint8_t flag) {
3      /* 后续操作的公共缓冲区 */
4      void* io_buf = sys_malloc(1024);
5      if (io_buf == NULL) {
6         printk("in file_creat: sys_malloc for io_buf failed\n");
7         return -1;
```

```
8     }
9
10    uint8_t rollback_step = 0;                // 用于操作失败时回滚各资源状态
11
12    /* 为新文件分配 inode */
13    int32_t inode_no = inode_bitmap_alloc(cur_part);
14    if (inode_no == -1) {
15       printk("in file_creat: allocate inode failed\n");
16       return -1;
17    }
18
19 /* 此 inode 要从堆中申请内存,不可生成局部变量 (函数退出时会释放)
20  * 因为 file_table 数组中的文件描述符的 inode 指针要指向它.*/
21    struct inode* new_file_inode = (struct inode*)sys_malloc(sizeof(struct
      ↪  inode));
22    if (new_file_inode == NULL) {
23       printk("file_create: sys_malloc for inode failded\n");
24       rollback_step = 1;
25       goto rollback;
26    }
27    inode_init(inode_no, new_file_inode);            // 初始化 i 结点
28
29    /* 返回的是 file_table 数组的下标 */
30    int fd_idx = get_free_slot_in_global();
31    if (fd_idx == -1) {
32       printk("exceed max open files\n");
33       rollback_step = 2;
34       goto rollback;
35    }
36
37    file_table[fd_idx].fd_inode = new_file_inode;
38    file_table[fd_idx].fd_pos = 0;
39    file_table[fd_idx].fd_flag = flag;
40    file_table[fd_idx].fd_inode->write_deny = false;
41
42    struct dir_entry new_dir_entry;
43    memset(&new_dir_entry, 0, sizeof(struct dir_entry));
44
45    create_dir_entry(filename, inode_no, FT_REGULAR, &new_dir_entry);        //
      ↪  create_dir_entry 只是内存操作不出意外,不会返回失败
46
47 /* 同步内存数据到硬盘 */
48    /* a 在目录 parent_dir 下安装目录项 new_dir_entry, 写入硬盘后返回 true,否则 false
      ↪  */
49    if (!sync_dir_entry(parent_dir, &new_dir_entry, io_buf)) {
50       printk("sync dir_entry to disk failed\n");
51       rollback_step = 3;
52       goto rollback;
53    }
54
55    memset(io_buf, 0, 1024);
56    /* b 将父目录 i 结点的内容同步到硬盘 */
57    inode_sync(cur_part, parent_dir->inode, io_buf);
58
59    memset(io_buf, 0, 1024);
```

```
60      /* c 将新创建文件的 i 结点内容同步到硬盘 */
61      inode_sync(cur_part, new_file_inode, io_buf);
62
63      /* d 将 inode_bitmap 位图同步到硬盘 */
64      bitmap_sync(cur_part, inode_no, INODE_BITMAP);
65
66      /* e 将创建的文件 i 结点添加到 open_inodes 链表 */
67      list_push(&cur_part->open_inodes, &new_file_inode->inode_tag);
68      new_file_inode->i_open_cnts = 1;
69
70      sys_free(io_buf);
71      return pcb_fd_install(fd_idx);
```

A.15 File Open/Close

```
1   int32_t file_open(uint32_t inode_no, uint8_t flag) {
2      int fd_idx = get_free_slot_in_global();
3      if (fd_idx == -1) {
4         printk("exceed max open files\n");
5         return -1;
6      }
7      file_table[fd_idx].fd_inode = inode_open(cur_part, inode_no);
8      file_table[fd_idx].fd_pos = 0;              // 每次打开文件,要将 fd_pos 还原为
       ↪  0,即让文件内的指针指向开头
9      file_table[fd_idx].fd_flag = flag;
10     bool* write_deny = &file_table[fd_idx].fd_inode->write_deny;
11
12     if (flag == O_WRONLY || flag == O_RDWR) {        // 只要是关于写文件,判断是否
       ↪  有其它进程正写此文件
13                                                     // 若是读文件,不考虑 write_deny
14     /* 以下进入临界区前先关中断 */
15        enum intr_status old_status = intr_disable();
16        if (!(*write_deny)) {    // 若当前没有其它进程写该文件,将其占用.
17           *write_deny = true;   // 置为 true,避免多个进程同时写此文件
18           intr_set_status(old_status);          // 恢复中断
19        } else {                 // 直接失败返回
20           intr_set_status(old_status);
21           printk("file can`t be write now, try again later\n");
22           return -1;
23        }
24     }  // 若是读文件或创建文件,不用理会 write_deny,保持默认
25     return pcb_fd_install(fd_idx);
26  }
27
28  /* 关闭文件 */
29  int32_t file_close(struct file* file) {
30     if (file == NULL) {
31        return -1;
32     }
33     file->fd_inode->write_deny = false;
34     inode_close(file->fd_inode);
35     file->fd_inode = NULL;   // 使文件结构可用
36     return 0;
```

```
}

/* 将文件描述符转化为文件表的下标 */
static uint32_t fd_local2global(uint32_t local_fd) {
   struct task_struct* cur = running_thread();
   int32_t global_fd = cur->fd_table[local_fd];
   ASSERT(global_fd >= 0 && global_fd < MAX_FILE_OPEN);
   return (uint32_t)global_fd;
}

/* 关闭文件描述符 fd 指向的文件，成功返回 0，否则返回-1 */
int32_t sys_close(int32_t fd) {
   int32_t ret = -1;   // 返回值默认为-1，即失败
   if (fd > 2) {
      uint32_t _fd = fd_local2global(fd);
      ret = file_close(&file_table[_fd]);
      running_thread()->fd_table[fd] = -1; // 使该文件描述符位可用
   }
   return ret;
}
```

A.16 File Delete

```
/* 将硬盘分区 part 上的 inode 清空 */
void inode_delete(struct partition* part, uint32_t inode_no, void* io_buf) {
   ASSERT(inode_no < 4096);
   struct inode_position inode_pos;
   inode_locate(part, inode_no, &inode_pos);     // inode 位置信息会存入 inode_pos
   ASSERT(inode_pos.sec_lba <= (part->start_lba + part->sec_cnt));

   char* inode_buf = (char*)io_buf;
   if (inode_pos.two_sec) {   // inode 跨扇区，读入 2 个扇区
      /* 将原硬盘上的内容先读出来 */
      ide_read(part->my_disk, inode_pos.sec_lba, inode_buf, 2);
      /* 将 inode_buf 清 0 */
      memset((inode_buf + inode_pos.off_size), 0, sizeof(struct inode));
      /* 用清 0 的内存数据覆盖磁盘 */
      ide_write(part->my_disk, inode_pos.sec_lba, inode_buf, 2);
   } else {    // 未跨扇区，只读入 1 个扇区就好
      /* 将原硬盘上的内容先读出来 */
      ide_read(part->my_disk, inode_pos.sec_lba, inode_buf, 1);
      /* 将 inode_buf 清 0 */
      memset((inode_buf + inode_pos.off_size), 0, sizeof(struct inode));
      /* 用清 0 的内存数据覆盖磁盘 */
      ide_write(part->my_disk, inode_pos.sec_lba, inode_buf, 1);
   }
}

/* 回收 inode 的数据块和 inode 本身 */
void inode_release(struct partition* part, uint32_t inode_no) {
   struct inode* inode_to_del = inode_open(part, inode_no);
   ASSERT(inode_to_del->i_no == inode_no);

```

```
31  /* 1 回收 inode 占用的所有块 */
32     uint8_t block_idx = 0, block_cnt = 12;
33     uint32_t block_bitmap_idx;
34     uint32_t all_blocks[140] = {0};          //12 个直接块+128 个间接块
35
36     /* a 先将前 12 个直接块存入 all_blocks */
37     while (block_idx < 12) {
38        all_blocks[block_idx] = inode_to_del->i_sectors[block_idx];
39        block_idx++;
40     }
41
42     /* b 如果一级间接块表存在,将其 128 个间接块读到 all_blocks[12~], 并释放一级间接块
       ↪  表所占的扇区 */
43     if (inode_to_del->i_sectors[12] != 0) {
44        ide_read(part->my_disk, inode_to_del->i_sectors[12], all_blocks + 12, 1);
45        block_cnt = 140;
46
47        /* 回收一级间接块表占用的扇区 */
48        block_bitmap_idx = inode_to_del->i_sectors[12] - part->sb->data_start_lba;
49        ASSERT(block_bitmap_idx > 0);
50        bitmap_set(&part->block_bitmap, block_bitmap_idx, 0);
51        bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
52     }
53
54     /* c inode 所有的块地址已经收集到 all_blocks 中,下面逐个回收 */
55     block_idx = 0;
56     while (block_idx < block_cnt) {
57        if (all_blocks[block_idx] != 0) {
58           block_bitmap_idx = 0;
59           block_bitmap_idx = all_blocks[block_idx] - part->sb->data_start_lba;
60           ASSERT(block_bitmap_idx > 0);
61           bitmap_set(&part->block_bitmap, block_bitmap_idx, 0);
62           bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
63        }
64        block_idx++;
65     }
66
67  /*2 回收该 inode 所占用的 inode */
68     bitmap_set(&part->inode_bitmap, inode_no, 0);
69     bitmap_sync(cur_part, inode_no, INODE_BITMAP);
70
71     /******      以下 inode_delete 是调试用的      ******
72     * 此函数会在 inode_table 中将此 inode 清 0,
73     * 但实际上是不需要的,inode 分配是由 inode 位图控制的,
74     * 硬盘上的数据不需要清 0,可以直接覆盖*/
75     void* io_buf = sys_malloc(1024);
76     inode_delete(part, inode_no, io_buf);
77     sys_free(io_buf);
78     /***********************************************/
79
80     inode_close(inode_to_del);
81  }
82
83  bool delete_dir_entry(struct partition* part, struct dir* pdir, uint32_t
    ↪  inode_no, void* io_buf) {
```

```
84      struct inode* dir_inode = pdir->inode;
85      uint32_t block_idx = 0, all_blocks[140] = {0};
86      /* 收集目录全部块地址 */
87      while (block_idx < 12) {
88         all_blocks[block_idx] = dir_inode->i_sectors[block_idx];
89         block_idx++;
90      }
91      if (dir_inode->i_sectors[12]) {
92         ide_read(part->my_disk, dir_inode->i_sectors[12], all_blocks + 12, 1);
93      }
94
95      /* 目录项在存储时保证不会跨扇区 */
96      uint32_t dir_entry_size = part->sb->dir_entry_size;
97      uint32_t dir_entrys_per_sec = (SECTOR_SIZE / dir_entry_size);       // 每扇区
        ↪  最大的目录项数目
98      struct dir_entry* dir_e = (struct dir_entry*)io_buf;
99      struct dir_entry* dir_entry_found = NULL;
100     uint8_t dir_entry_idx, dir_entry_cnt;
101     bool is_dir_first_block = false;     // 目录的第 1 个块
102
103     /* 遍历所有块,寻找目录项 */
104     block_idx = 0;
105     while (block_idx < 140) {
106        is_dir_first_block = false;
107        if (all_blocks[block_idx] == 0) {
108           block_idx++;
109           continue;
110        }
111        dir_entry_idx = dir_entry_cnt = 0;
112        memset(io_buf, 0, SECTOR_SIZE);
113        /* 读取扇区,获得目录项 */
114        ide_read(part->my_disk, all_blocks[block_idx], io_buf, 1);
115
116        /* 遍历所有的目录项,统计该扇区的目录项数量及是否有待删除的目录项 */
117        while (dir_entry_idx < dir_entrys_per_sec) {
118           if ((dir_e + dir_entry_idx)->f_type != FT_UNKNOWN) {
119              if (!strcmp((dir_e + dir_entry_idx)->filename, ".")) {
120                 is_dir_first_block = true;
121              } else if (strcmp((dir_e + dir_entry_idx)->filename, ".") &&
122                 strcmp((dir_e + dir_entry_idx)->filename, "..")) {
123                 dir_entry_cnt++;     // 统计此扇区内的目录项个数,用来判断删除目录
                    ↪  项后是否回收该扇区
124                 if ((dir_e + dir_entry_idx)->i_no == inode_no) {          // 如果
                    ↪  找到此 i 结点,就将其记录在 dir_entry_found
125                    ASSERT(dir_entry_found == NULL);  // 确保目录中只有一个编号为
                       ↪  inode_no 的 inode,找到一次后 dir_entry_found 就不再是 NULL
126                    dir_entry_found = dir_e + dir_entry_idx;
127                    /* 找到后也继续遍历,统计总共的目录项数 */
128                 }
129              }
130           }
131           dir_entry_idx++;
132        }
133
```

```
134         /* 若此扇区未找到该目录项，继续在下个扇区中找 */
135         if (dir_entry_found == NULL) {
136            block_idx++;
137            continue;
138         }
139
140      /* 在此扇区中找到目录项后，清除该目录项并判断是否回收扇区，随后退出循环直接返回
         ↪  */
141         ASSERT(dir_entry_cnt >= 1);
142       /* 除目录第 1 个扇区外，若该扇区上只有该目录项自己，则将整个扇区回收 */
143         if (dir_entry_cnt == 1 && !is_dir_first_block) {
144            /* a 在块位图中回收该块 */
145            uint32_t block_bitmap_idx = all_blocks[block_idx] -
               ↪  part->sb->data_start_lba;
146            bitmap_set(&part->block_bitmap, block_bitmap_idx, 0);
147            bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
148
149            /* b 将块地址从数组 i_sectors 或索引表中去掉 */
150            if (block_idx < 12) {
151               dir_inode->i_sectors[block_idx] = 0;
152            } else {    // 在一级间接索引表中擦除该间接块地址
153               /*先判断一级间接索引表中间接块的数量，如果仅有这 1 个间接块，连同间接索引
                  ↪  表所在的块一同回收 */
154               uint32_t indirect_blocks = 0;
155               uint32_t indirect_block_idx = 12;
156               while (indirect_block_idx < 140) {
157                  if (all_blocks[indirect_block_idx] != 0) {
158                     indirect_blocks++;
159                  }
160               }
161               ASSERT(indirect_blocks >= 1);  // 包括当前间接块
162
163               if (indirect_blocks > 1) {          // 间接索引表中还包括其它间接块，
                  ↪  仅在索引表中擦除当前这个间接块地址
164                  all_blocks[block_idx] = 0;
165                  ide_write(part->my_disk, dir_inode->i_sectors[12], all_blocks +
                     ↪  12, 1);
166               } else {        // 间接索引表中就当前这 1 个间接块，直接把间接索引表所在
                  ↪  的块回收，然后擦除间接索引表块地址
167                  /* 回收间接索引表所在的块 */
168                  block_bitmap_idx = dir_inode->i_sectors[12] -
                     ↪  part->sb->data_start_lba;
169                  bitmap_set(&part->block_bitmap, block_bitmap_idx, 0);
170                  bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
171
172                  /* 将间接索引表地址清 0 */
173                  dir_inode->i_sectors[12] = 0;
174               }
175            }
176         } else { // 仅将该目录项清空
177            memset(dir_entry_found, 0, dir_entry_size);
178            ide_write(part->my_disk, all_blocks[block_idx], io_buf, 1);
179         }
180
```

```
181     /* 更新 i 结点信息并同步到硬盘 */
182        ASSERT(dir_inode->i_size >= dir_entry_size);
183        dir_inode->i_size -= dir_entry_size;
184        memset(io_buf, 0, SECTOR_SIZE * 2);
185        inode_sync(part, dir_inode, io_buf);
186
187        return true;
188     }
189     /* 所有块中未找到则返回 false, 若出现这种情况应该是 serarch_file 出错了 */
190     return false;
191  }
192
193  /* 删除文件 (非目录), 成功返回 0, 失败返回-1 */
194  int32_t sys_unlink(const char* pathname) {
195     ASSERT(strlen(pathname) < MAX_PATH_LEN);
196
197     /* 先检查待删除的文件是否存在 */
198     struct path_search_record searched_record;
199     memset(&searched_record, 0, sizeof(struct path_search_record));
200     int inode_no = search_file(pathname, &searched_record);
201     ASSERT(inode_no != 0);
202     if (inode_no == -1) {
203        printk("file %s not found!\n", pathname);
204        dir_close(searched_record.parent_dir);
205        return -1;
206     }
207     if (searched_record.file_type == FT_DIRECTORY) {
208        printk("can`t delete a direcotry with unlink(), use rmdir() to
           ↪  instead\n");
209        dir_close(searched_record.parent_dir);
210        return -1;
211     }
212
213     /* 检查是否在已打开文件列表 (文件表) 中 */
214     uint32_t file_idx = 0;
215     while (file_idx < MAX_FILE_OPEN) {
216        if (file_table[file_idx].fd_inode != NULL && (uint32_t)inode_no ==
           ↪  file_table[file_idx].fd_inode->i_no) {
217           break;
218        }
219        file_idx++;
220     }
221     if (file_idx < MAX_FILE_OPEN) {
222        dir_close(searched_record.parent_dir);
223        printk("file %s is in use, not allow to delete!\n", pathname);
224        return -1;
225     }
226     ASSERT(file_idx == MAX_FILE_OPEN);
227
228     /* 为 delete_dir_entry 申请缓冲区 */
229     void* io_buf = sys_malloc(SECTOR_SIZE + SECTOR_SIZE);
230     if (io_buf == NULL) {
231        dir_close(searched_record.parent_dir);
232        printk("sys_unlink: malloc for io_buf failed\n");
233        return -1;
```

```
234     }
235
236     struct dir* parent_dir = searched_record.parent_dir;
237     delete_dir_entry(cur_part, parent_dir, inode_no, io_buf);
238     inode_release(cur_part, inode_no);
239     sys_free(io_buf);
240     dir_close(searched_record.parent_dir);
241     return 0;   // 成功删除文件
242 }
```