# 摘要

本文首先对操作系统的基础知识以及主要的功能进行研究，接着对操作系统开发工具、运行环境以及所用到的语言等进行简述，并对X86架构下的IA32的保护模式进行详细的分析。最后重点进行了操作系统内核的设计和实现。

本次毕设从零开始实现一个简单的操作系统内核，旨在对本科所学的计算机知识进行巩固并综合应用，从而能为以后阅读甚至开发一个大型商用操作系统打下坚实的基础。该操作系统内核的结构精简，主要包含Loader的实现、内核整体结构的搭建、进程管理、文件管理、内存管理、多控制台输入输出系统管理。并采用命令行的用户交互方式，通过系统调用，完成了echo、fwd简单命令的编写以及键盘的输入输出交互。

# 关键词

内核；操作系统；IA32保护模式；简单的shell

# ABSTRACT

In this paper, the basic knowledge and main functions of the operating system are studied, and then the operating system development tools, the running environment and the language used are briefly described, and the protection mode of IA32 under the X86 architecture is analyzed in detail. Finally, the design and implementation of the operating system kernel are emphasized.

The design and implementation of a simple operating system kernel from scratch is aimed at consolidating and synthetically applying the computer knowledge learned by undergraduate students. It can lay a solid foundation for reading and even developing a large commercial operating system. The kernel structure of the operating system is simple, including the realization of Loader, the construction of the whole kernel structure, the process management, the file management, the memory management, the multi-console input and output system management. By using the command line user interaction mode, and by calling the designed system call, the simple command of echo Pwd and the input and output of the keyboard are completed. Mutual .

# Keywords

Kernel ;Operating system ;IA32bit Protect Mode;Simple shell

目录

[摘要 1](#_Toc17298)

[关键词 1](#_Toc26633)

[ABSTRACT 2](#_Toc4639)

[Keywords 2](#_Toc27097)

[1绪论 5](#_Toc19306)

[1.1 论文背景和目的 5](#_Toc27755)

[1.2 国内外研究现状 6](#_Toc30848)

[1.3 主要研究内容 7](#_Toc1638)

[1.4 全文结构 7](#_Toc2039)

[2操作系统基本原理研究与分析 9](#_Toc31853)

[2.1计算机系统硬件概述 9](#_Toc700)

[2.1.1中央处理器（CPU） 9](#_Toc5929)

[2.1.2存储器 10](#_Toc11396)

[2.1.3输入输出设备（Input/Output） 10](#_Toc23167)

[2.1.4总线 10](#_Toc5270)

[2.2核心功能的研究与分析 10](#_Toc12206)

[2.2.1进程运行原理分析 11](#_Toc511)

[2.2.2文件管理策略对比分析 12](#_Toc24874)

[2.2.3内存管理策略分析 13](#_Toc16001)

[2.2.4输入输出管理研究 13](#_Toc32358)

[2.3基本特性分析 14](#_Toc7595)

[2.3.1并发性 14](#_Toc12020)

[2.3.2共享性 14](#_Toc18941)

[2.3.3异步性 15](#_Toc30628)

[2.3.4虚拟性 15](#_Toc28693)

[3工作环境部署与相关技术研究 16](#_Toc18820)

[3.1工作环境的部署 16](#_Toc24612)

[3.2WOS的运行原理分析 17](#_Toc12780)

[3.2.1使用引导扇区从BIOS获取计算机权限 17](#_Toc16457)

[3.2.2使用Loader加载内核的策略设计与实现 17](#_Toc31750)

[3.2.3运行内核 19](#_Toc32384)

[3.3保护模式研究 19](#_Toc22381)

[3.3.1保护模式概述 19](#_Toc26437)

[3.3.2全局描述符研究 19](#_Toc29905)

[3.3.3分页机制策略分析 21](#_Toc11808)

[3.3.4特权级原理分析 22](#_Toc20684)

[3.3.5中断策略研究 23](#_Toc24494)

[3.3.6保护模式的小结 25](#_Toc26183)

[4操作系统内核的设计与实现 26](#_Toc12523)

[4.1内核整体设计 26](#_Toc3363)

[4.2进程管理设计与实现 27](#_Toc32194)

[4.2.1进程运行算法实现 27](#_Toc12029)

[4.2.2时钟中断算法实现 30](#_Toc6452)

[4.2.3进程优先级调度算法实现 32](#_Toc2890)

[4.2.4进程间通信策略对比分析及实现 33](#_Toc25486)

[4.3文件系统设计与实现 34](#_Toc32654)

[4.3.1创建文件的系统调用算法实现 34](#_Toc2133)

[4.3.2文件的打开与关闭算法实现 36](#_Toc1377)

[4.3.3文件的读写算法实现 36](#_Toc8456)

[4.3.4删除文件算法实现 38](#_Toc19169)

[4.4系统交互与I/O管理设计与实现 39](#_Toc30557)

[4.4.1fork的算法设计与实现 39](#_Toc21120)

[4.4.2进程消亡的系统调用算法设计与分析 40](#_Toc17215)

[4.4.3实现exec算法 41](#_Toc16061)

[4.4.4键盘、显示器驱动设计 43](#_Toc9860)

[4.4.5 多控制台算法实现 45](#_Toc22043)

[5总结与展望 48](#_Toc29059)

[5.1工作总结 48](#_Toc17391)

[5.2展望 48](#_Toc13619)

[参考文献（待补充） 49](#_Toc25728)

[致谢 50](#_Toc22331)

# 1绪论

## 1.1 论文背景和目的

在计算机高速发展的今天，挑选自己喜欢的配件来组装一台个性化的PC对我们来说已经不困难了。不仅如此，只要有合适的编译环境，我们可以自己制作游戏、APP、网页等。但是作为软件与硬件桥梁的操作系统，却很少有人会去做，因为一个可用的商用的操作系统所需要的不仅仅是一个人的聪明才智，更需要的是一个操作系统的生态系统[1]，像Windows、Linux和Mac等操作系统，要是凭借一个人的兴趣开发，不知道要经过多久才能完成，甚至可以说单凭一个人，一辈子也做不出来。当然，目前网上也有很多前辈留下的开源的操作系统内核项目（例如类Unix、Linux源码），种类繁多，如果没有很强的代码阅读能力以及专业知识能力，是很难真正看懂的，因为大量代码都是直接针对CPU的以及输入输出端口进行的。所以从零开始，一步一步的开发一个简单的操作系统内核，不仅能将本科期间的知识进行综合应用，并且能为今后从事相关行业、自己将来能为国家操作系统生态网贡献一份力量做铺垫。

作为一个未来立志从事计算机行业的本科生来说，在本科期间所学的操作系统相关知识都停留在一个很表层的阶段，没有从底层一步步实验的经验，无法做到对计算机从下至上的整体把握，这对以后的深入学习都留下隐患。例如本次毕设所用到的语言是汇编和C语言，其中汇编语言的本科学习还是停留在16位实模式下，而目前市面上因为操作系统配合使用的内存都很大了，所以都使用的是32位的保护模式。虽然两者的语法类似，但是32位的相关知识在汇编这门课中基本没有学习；同理作为底层开发的C语言，也仅仅作为一门基础语言进行学习，没有更加深入的使用，更加无法掌握其精髓。而操作系统作为需要对概念和原理深入理解的学科，如果没有大量阅读学习经典开源源码以及进行大量底层实验，是无法做到了然于心的，也更加无法做到创新了，与之相关的生态系统只能依靠国外技术，将永远无法站在技术的制高点。

本次毕设通过研究X86架构下的操作系统内核，达到实现一个简单的操作系统内核的目的，同时对本科的课程进行加深理解与巩固，由原来的理论落实带具体的代码，这样才能将原先的基础课程的理解加强和综合运用，并能有全新的理解，最终具备能够对大型操作系统源码进行阅读和学习的能力，为以后的继续深造打下更好的坚实的基础。

## 1.2 国内外研究现状

作为计算机软件与硬件之间的桥梁与核心，操作系统在整个计算机领域的地位是不言而喻的。作为整个计算机中可以说是最重要的软件，它不仅管理这整个计算机中运行的程序，也对硬件资源的分配起到的决定性的作用，可以说，操作系统的发展，影响着整个计算机的发展。

在计算机刚刚诞生时，是没有操作系统的，那个时候的计算机是采用手工的方式来进行操作的，所以资源利用率以及效率都十分低下，这个时候就应运而生了一中监视程序，用于提高计算机的资源利用率以及强化了硬件设备的工作效率，它是计算机系统和用户之间的界面，而后，由于中断等技术的引入，监视系统进一步发展，形成了操作系统，而后由于超大规模集成电路的出现，使得微处理器（CPU、GPU)的发展得到了显著的提升，同时硬件系统也相较于以前有了很大的提高，这就使得网络操作系统出现了。

总的来说操作系统的发展，可以分为单道批处理系统阶段，多道批处理系统的阶段，然后又经过了分时处理系统到实时操作系统，到现在由于硬件的提升，从而云技术的发展出，网络操作系统成为了这个时代的产物，这很明显是一个漫长而又具有突破性的过程。

在无操作系统的计算机时代，因为人工操作的原因，CPU和内存等大部分处于空闲的状态，所以计算机工作效率比较低下，那个时候的研究主要是在解决CPU执行速度与I/O设备处理速度不匹配的情况。

而后，为了将CPU的资源利用率提高，面对单道批处理系统的时候，程序员就将所有作业都放在磁带上，然后让计算机自动地处理，这也就是单道批处理的雏形，这个时候研究的方向主要在如何对单道批处理进行优化，让CPU等资源得到更好的利用

接着就是多道批处理系统的出现，它将左右排列成队列，然后由相应的调度算法调度作业到内存中，这样的好处就是能充分利用内存的空闲资源，达到匹配CPU与外存速度的目的。

然后，为了满足用户的不同需求，分时系统诞生了，它让一台计算机有多个终端，还允许每个终端通过交互共享主机的资源。

现在，为了满足快速响应外部事件的各种请求，并在规定实践完成相关操作，协调各个任务的工作，实时系统应运而生。

而国内的操作系统的发展一直都是生机勃勃，比如优麒麟、深度Linux等等，虽然有借鉴的成分，但都是在开源的项目的基础上，进行了创新式的开发，并随之带来了一些列的操作系统生态程序。

伴随着计算机的不断飞速发展以及大数据、人工只能时代的到来，在目前主流操作系统仍然占据大部分市场份额的基础上，未来的操作系统，会朝着专用化发展，并会有如下特性：

1.专用化：当今计算机发展到各个领域，移动计算机的高速发展，使得很多领域都需要专门的操作系统，比如企业专用操作系统、国家机构安全操作系统、个性工业化操作系统等。用来满足不同领域的个性化需要。

2.开源化：开源是的操作系统的发展能够汇集大家之力，从而可以有与现有的主流的操作系统进行抗衡的可能，避免了操作系统的垄断，也让个性化的操作系统的发展日益强盛。同时，也为诞生高质量、更安全的操作系统提供机会。

3.网络化：随着各种网络云空间的诞生和普及，未来的操作系统可能就不会在实体的个人主机中存在，而是可能存在某台服务器中，供大家使用，从而大大节约了个人成本，也是的资源的利用率更高。

4.安全化：因为当今的操作系统涉及到军事、金融、支付、社交等各个领域，所以需要更加安全、更加私密的操作系统，而网络的开源又会和其相矛盾，所以安全问题也是很重要的发展研究方向。

## 1.3 主要研究内容

1. 技术上需要重点研究以下几点：

操作系统内核运行之前据的技术研究

理解 x86 架构启动过程

1. 对Bootloader的 原理与实现
2. 操作系统内核原理研究

进程管理原理分析以及设计

文件管理原理分析以及设计

内存管理原理分析以及设计

操作系统与外部设备的通信技术研究

输入输出系统原理分析以及设计

1. 将操作系统的引导程序做成grub，并成功引导入内存加载。
2. 将最终的操作系统内核制作成虚拟软盘，并在虚拟机上运行。

## 1.4 全文结构

本论文供分为六个部分，具体的文组织结构如下：

第一部分是绪论，主要介绍了论文的背景和目的，国内外的操作系统系统研究现状，论文主要研究内容。

第二部分是操作系统基本原理研究与分析，主要以操作系统底层硬件为切入点，对操作系统的四大功能进行管理策略的研究，并对其特性进行分析。

第三部分是操作系统相关技术研究分析与环境部署，首先对本次毕设的工作环境的部署进行说明，并将整体的开发进行流程整理，然后着重对保护模式进行着重分析。

第四部分是操作系统内核的设计与实现，围绕4大核心的模块进行具体设计与算法的分析，详细说明了各个模块的设计思路、解决方案，并对模块的实现进行相关说明。

第五部分是总结与展望，对整个内核进行收获总结，指出不足之处，并对今后可以改进和增加的部分进行展望。

最后是参考文献和致谢。

# 2操作系统基本原理研究与分析

## 2.1计算机系统硬件概述

在日常购买计算机的时候，一般人都关注的是计算机硬件的价格与配置，会觉得硬件对计算机来说是最重要最有价值的，但其实不然，硬件是必要的，但在硬盘上还有大量的软件，其中，操作系统就是计算机最为重要的软件了，它不仅让其他程序有条不紊的运行在电脑上，还合理的调配计算机的硬件资源，让软件和硬件都做大最大化的利用。所以在分析操作系统之前，需要对计算机系统的整体硬件有所了解。

由图2-1计算机系统结构图可以很直观的对整个计算机系统有个整体的了解。

图2-1 计算机系统结构图

CPU

ALU

寄存器文件

PC

总线接口

I/O桥

主存储器

图形适配器

磁盘控制器

USB控制器

磁盘

鼠标

键盘

显示器

系统总线

存储器总线

I/O总线

扩展插槽

### 2.1.1中央处理器（CPU）

中央处理器又可以简称为CPU，它主要由算术逻辑运算单元（ALU）、控制器以及一些寄存器等部分组成。

从计算机通电到结束，处理器就在不断的贡献它的运算能力，不停的执行指令，它按照已定的模型进行执行指令，这就是指令集的来源。

运算器。其主要功能是对二进制数进行包裹加、减、乘、除等在内的算数运算和与、或、非等基本逻辑运算，实现计算机及逻辑判断。运算器在控制器的控制下实现其功能，运算结果由控制器只会送到内存储器中。

控制器。其基本功能就是从内存储器中取出指令和执行指令，即控制器按程序计算器指出的指令地址从内存储器中取出该指令进行译码，然后更具该指令功能向有关部件发出控制命令，执行该指令。另外，控制器在工作过程中，还要接受各部件反馈回来的信息。

寄存器。顾名思义，它是用来“寄存”的容器，而这些“东西”可以是执行的指令临时存放的操作数或者一些中间的或者最终的结果。它的分类也是多种多样，既有控制寄存器、也用通用控制器，它的容量不大，但是速度很快。

通过具体来说明CPU的功能的话，主要可以分为从主存将数据复制到寄存器里，而其中的值将被覆盖、从寄存器中将数据覆盖掉内存中的值用来存储、将若干寄存器中的值取出进行逻辑或者算数运算，并返还到另一个寄存器中。

### 2.1.2存储器

计算机的存储功能可以说是除了CPU以后最为核心的部分了，它主要分为了外部存储（如硬盘、U盘）也被称作辅存，它的速度不如内存但是胜在可存储空间大，由以前以MB为单位，到现在的GB、TB为单位的个人电脑也随处可见；而主存也叫做内存，其读写速度很快。从物理角度来说，是由若干动态随机存储芯片组成，正因为如此，它的价格较为高昂，所以在操作系统的演变过程中，从实模式到保护模式的发展，有很大一部分是因为内存的变化导致的。

### 2.1.3输入输出设备（Input/Output）

为了从过去的手动输入到更为方便快捷的输入方式，现在有很多的输入设备，包括键盘、写字板、眼球捕捉器等等，将人与计算机的交互过程变得更为的简单，同时层出不穷的输出设备，比如显示屏、投影仪、可穿戴设备等等，让今天的世界都变的更具有科技感，人们的生活也因为各种各样的输入输出设备而变的更加丰富多彩了，相信不久的将来，各种语音设备的普及，以后真的可以做到和机器进行说话交流了。

### 2.1.4总线

因为计算机系统的总线主要承担了数据信息、控制信息和地址信息的传输，总线就如穿梭在城市里固定线路的公交车线路，传递的信息不同；它也如同中枢神经，联系电脑的各个硬件，进行各类型的数据信息的传递功能。

## 2.2核心功能的研究与分析

操作系统提供给用户进程一些系统调用，所以，在用户看来需要哪些功能，只要使用这些系统调用就可以了，而不用在考虑硬件的兼容或者优化资源分配等问题了，所以，用户就能够全身心的做自己的事情。所以操作系统的主要功能，就是对多程序进程良好的管理，并对有限的资源进行合理并高效的分配使用。抽出操作系统的共性，总的来说就可以有以下几个部分（因为操作系统可以根据需要进行特殊的制定，但是基本都会包含如下功能）。

### 2.2.1进程运行原理分析

进程是操作系统中最重要的概念之一，同时它也是最复杂的概念，一个最简单的进程雏形，也要考虑很多因素。对进程的具体概念，我们不妨把系统中运行的若干进行想象成一个人在一天内要做的若干样工作：总体来看，每样工作相对独立，并可产生某种结果；从细节上看，每样工作都有自己的方法、工具和需要的资源；从时间上看，每一个时刻只能有一项工作正在处理中，各项工作可以轮换来做，这对与最终结果没有影响。

进程和这个是有相同之处的，从大的层面上去理解，它有自己的功能或者想要达到的目的，同时又必须受控于进程调度模块；然后再从小的层面上观察，它可以利用系统的资源，并且拥有独自的代码和数据和自己的堆栈；进程需要被调度，就好比一个人需要扮演不同角色轮换着做着不同的工作。进程的示意图如下图2-2所示。

图2-1 进程示意图

进程调度

代码

数据

代码

数据

堆栈

堆栈

进程A

进程B

对进程的管理，主要分为以下三个部分：

进程控制。进程控制通过进程原语对进程进行控制，基本的进程控制原语有：

进程建立（在系统启动的时候就创建一个始祖进程，由它来进程后续的进程的创建工作）、进行的撤销（主要是释放进程占用的资源以及告知父进程修改PCB的相关内容）、进程等待（因为在进程调度的过程中，时不时的会发生类似资源不够的情况，这时就需要进程等待原语，将进行重新放到队列中排队）和进程唤醒（当某个系统条件被满足了以后，需要接受新的进程来运行，这时候进程唤醒原语就发挥作用了）。

进程同步。因为进程是并发程序的执行，所以对于进程来说必然有各种各样的关系形式，对资源的争夺、对共同任务的协作完成，这些就是进程的同步。它包括互斥关系和同步关系。

进程通信。为了让进程之间相互制约或者共同协作完成某个任务，就需要在进行之间进行一定的数据传递。传递的方式分为管道和消息通信（在本次毕设中主要使用的是消息通信的方式）。

### 2.2.2文件管理策略对比分析

对于计算机来说，最重要的作用之一就是能快速处理大量信息。由于计算机的内存容量有限，且不能长期保存信息（系统断电就擦除内存信息了），所以需要用文件作为数据的载体，长期保存或者输入到应用程序中。所以无论是图片、文本文档、视频等等，在不用时都是存在辅存中的，只有当需要使用时才会调入到内存中。在文件系统中，块是文件的系统的读写单位，一个文件只有达到一个块时，才能被读写操作，而那么大的文件，肯定不止会产生一个块，这时候，文件的组织管理就是最大问题。

在FAT文件系统中，块是通过链式结构来组织的，而在Unix操作系统中，文件的组织方式为索引结构——inode。在本次毕设中，选择以inode的方式，以为这种方式更加简洁，已经被广泛使用了。

文件系统为实现文件管理方案，所以创造了一些辅助管理的数据结构，只要用于管理、控制文件相关信息的数据结构都被称为FCB（File Control Block），即文件控制块，inode也是这种结构，因此inode也是一种FCB。想要通过文件系统获得文件的实体，必须先要找到文件的inode，所以从这个角度来说，inode等同于文件。为了方便管理，分区中的所有文件的inode都用一个大表格来维护，这个表格被称为inode\_table它的本质就是inode的数据。同时，为了满足用户通过文件名来查找问文件的方式，所以就有了目录的概念，在计算机看来，目录也是一个文件，所以是同一种inode,也就是说，如果该inode是目录文件，在该inode中的数据项就是目录项，否则就是普通文件的数据。

有了目录项以后，通过文件名找文件实体数据块的流程就是：

1. 在目录中找到文件名所在的目录项
2. 从目录项中获取inode编号
3. 从inode编号作为inode数组的索引下标，找到inode
4. 从该inode中获取数据块的地址，读取数据块

为了说明inode数组在哪里，大小为多少等数据，我们需要在固定的地方读取文件系统的信息配置，这个地方就是超级块。

对于用户来说，文件的组织方式和存储方式都无所谓，自需要在查找文件时简便快捷就好了，所以这就需要文件系统对文件进行目录管理，以及基本的文件的建立、文件的打开、关闭、文件的读写和撤销进行设计。

当然随着社会的进步、科技的发展，共享也变得很重要了，在文件系统中，共享文件和文件的安全是会一直被联系在一起的。在实现文件共享的同时保障文件的安全，在保证文件的安全性的时候，也要兼顾文件共享的性能。其中文件的共享方法主要有绕道法、链接法；文件的安全通过系统权级的不同来进行设立，同时兼顾有目录级安全管理（读写权限设置）和文件级的安全（对文件的读写权限设置）等等。同时密码学的发展，也让文件的安全控制手段有了质的飞跃。

### 2.2.3内存管理策略分析

在计算机的世界里，任何资源都是宝贵的，而内存是仅次于CPU的宝贵资源（现在的硬盘也有如SSD一样比较宝贵的资源）。而在多道批处理程序系统中，时刻都可能发生众多进行共用一个内存的情况，这个时候，内存的管理显得极其重要，内存如何分配才能是内存的利用率最高？使用什么方法不同进程的安全才能得到保障？如何处理不同进程对同一内存空间的共享？在内存不足时，有什么办法能调用外存进行辅助？这些都是内存管理需要解决的事情。

所以总的来说，内存管理需要有的功能包括：用户进程实体的状态与存储空间的分配问题、存储释放的问题、存储保护与共享、地址的映射、虚拟存储器的设计、存储区的整理。

随着文件越来越多，需要编译的文件也多了起来，如果每次都是手动的逐个编译，然后再手动链接到一起，那就会很麻烦。所以自动的针对有过改动的文件进行编译，Linux为我们提供了make命令，并将规则、依赖关系定义在一个名叫makefile的文件中，它和make搭配在一起，就可以实现发现某个文件更新以后，只编译该文件和受该文件影响的相关文件，其他就不编译，从而提高编译效率。

### 2.2.4输入输出管理研究

计算机是一个结构精密的器件，从最严格的意义来说，用户接触到的各种输入输出设备都不是计算机的组成部分，但是，如果没有键盘、鼠标、显示器这些设备，普通用户很难操作计算机。所以计算机系统中，一般都会将除了处理机和主存储器之外的部分统一称作外围设备。而这样一来，又有新的的问题出现了，市面上的外围设备不下百种，那么不可能对每一种设备制作一个对应的接口吧，所以，操作系统必须解决这些设备的管理问题。

所以，在操作系统的输入输出管理，应该有为每一类设备提供对应的设备驱动程序的功能、要为设备分配逻辑名从而让设备的工作具有独立性、同时要兼顾CPU与外围设备速度不匹配的问题，所以还有具备缓存区管理的功能。而最重要的是对输入输出控制方式的设计，总的趋势是CPU和输入输出设备的并行程度越来越高，从而让输入输出的效率达到和CPU、内存的一个平衡点。

## 2.3基本特性分析

### 2.3.1并发性

在现今的计算机系统中，很少见到单个程序的计算机，所以，解决系统中在同一个时间间隔内多程序运行的问题，是操作系统的一大特性。并发与并行性不同的地方在于，并发性是指在同一个时刻运行的多个程序，就如同跑步比赛，信号声响起，所有参赛选手都一起出发；而并发性就如同等红绿灯，在一定的时间内，汽车都可以通过。

更形象点说，并行的方式是无时无刻，都是一同执行的，就如同有多个水池，同一时间段内，每个水池都会有一池子的水。但是并发则不同，它在微观上来说是在不同的时间内执行的，仅仅是将时间看作是分成了一段一段的，从而达到若干个进程快速执行的目的，从宏观外来看，好像是这些进程都在执行罢了。

在不同类型的操作系统中并发性的含义会有一些区别。在单处理器操作系统中每个特定时刻只有一个程序在CPU中运行。但是一个较长的时间可以被分为很多小的时间段，来运行不同的程序，使得这个较长的时间段内所有的程序都得到了运行。这些程序就具有并发性不具有并行性。所以从真正的本质来看，并发的操作对于多个进程来说，在使用的资源有所冲突的时候，从根本上来说，不会提高效率。

长期以来，进程是操作系统中的一个基本单元，它可以拥有资源并独立运行。当进程因任何原因无法继续运行时，操作系统将调度另一个程序运行。然而，由于进程具有自己的资源，所以调度的成本非常高，所以人们比进程线程引入更小的单元。通常，进程包含多个线程，这些线程可以利用进程拥有的资源。在引入了线程的操作系统中，通常将进程视为资源分配的基本单位，并将线程视为独立操作和独立调度的基本单元。由于线程小于进程，并且基本上不具有系统旋转，所以在陆地调度上花费的系统资源非常小，因此可以更有效地提高系统中多个程序的并发执行的程度。

### 2.3.2共享性

计算机如同社会，本质都是资源的不断分配，而这其中，有个重要的概念就是共享，它能让有限的，可以被重复利用资源被多个对象同时或者不同时的使用的性质。在计算机中是指系统中的资源（比如内存资源、CPU资源、输入输出设备资源）可以被多个并发执行的程序共同使用，而不是被其中一个独占。资源共享有两种方式：互斥访问和同时访问。互斥共享方式：一段时间内只允许一个进程访问该资源，如磁带机、打印机等，虽然可供多个进程使用，但为了打印或记录的结果不造成混淆，应规定一段时间内只允许一个进程访问该资源。

同时共享方式：某些资源，一段时间内允许多个进程“同时”对他们进行访问，这个“同时”是宏观上，在微观上可能是分时共享，典型的例子：磁盘设备。并发和共享是操作系统的最基本特征，互为依存。并发执行的要求引出了资源的共享；而资源共享的管理又直接影响到程序的并发执行。

### 2.3.3异步性

进程的异步性是指一个进程在计算机中，因为一台计算机只有一个处理器，而处理器处理起来又只能一个一个的进行，所以当有一个进程在处理器上的时候，其他的程序只能处于等待的状态（waiting）,并且当运行的程序需要某些资源的时候，如果该资源无法得到满足，那么这个进程只能中断处理，所以一个进程的执行是无法一直顺顺利利的进行到结束的，而且，这种停止以后可能有继续的运行方式，是不可预测的，是一种随机的事件，不仅因为计算机本身的资源分配问题，同时还与不同的机器性能，任务之间的关联等等，都有很大的关联。这种不能预测，但是最终的执行结果又必须得是一定的方式，就是操作系统的另一种特性，异步性。为了防止或者说应对这种异步性，操作系统需要解决好同步的问题，使得不同的进行间对资源的利用以及分配达到协调统一的底部，从而得到一个符合预期的结果。

### 2.3.4虚拟性

虚，可以理解为是用户感觉上的存在，比如在计算机上显示某个接口是存在的，但是可能很多个显示出来的结构都是共同使用的一个硬件结构，这就使得一系列的物理实体实际上呈现出来的就只有一个真正的逻辑实体；而也可以将一个真正的物理硬件变成多个逻辑上的硬件（比如网络云盘，可能只有一个真正的硬盘，但是用户会觉得我和其他人用的不是一个）。

# 3工作环境部署与相关技术研究

## 3.1工作环境的部署

在开始开发之前，很有必要把准备工作做好，这其中最为关键，也是以后的开发的依仗的就是工作环境的部署。在没有接触内核开发之前，会将操作系统内核这种近乎底层的开发视为登天之难，以为没有任何上层的高级软件进行构建，但是事实是有的！

首先操作系统是一个软件，而一个软件必定是由编程语言实现的，作为和CPU等硬件直接的对话的机器语言当然是可以编写的，但是过于繁琐，所以通过稍微高级的语言进行编写，然后通过编译器翻译成机器语言觉得是开发的首选。

对于编译器，使用GCC和NASM分别作为C语言和汇编语言的编译器是最好的选择之一。在本次毕设只会用到GCC的C编译器部分，所以GCC实际上就是GNU C Compiler；而汇编语言作为能直接针对寄存器的开发，所以是无法避免使用的，所以也要使用汇编的编译器——NASM，作为一个为了可移植性和模块化而诞生的汇编器，它支持Linux平台，并且语法简单和免费。而本次的开发语言也显而易见，是C和汇编语言了，它们时而分开开发，时而“混合开发”但是这个“混合开发”指的是单独的汇编代码文件好单独的C语言分别编译成目标文件以后，一起链接成可执行文件。

在选择完编译器之后，就开始Bochs的说明。

操作系统不同于其他软件的地方在于，其他软件都是在操作系统上运行的，开发者只用关注自己的业务编写相关的逻辑就可以了，而操作系统的支持就是开发者自身了，但是在现今，有很多计算机前辈，开源了很多调试操作系统的工具，这样就让我们避免了出现错误的时候不断关机重启了，这一切都是因为虚拟机的诞生。

虚拟机的出现，可以极大的利用计算机，让以往一台只能被一个用户使用的实体计算机被软件逻辑分为好多个虚拟计算机，各个计算机之间是互不干涉的，在各个计算机上自由的调试不用担心损伤硬件。

在本次毕设中，因为条件有限，所以只能在Window7下操作，但为了更好的兼容性，所以使用Vmware虚拟机在上面安装一个Ubuntu然后在Ubuntu里再安装Bochs虚拟机，有了Bochs，在开发的时候就可以监控CPU、内存、硬盘等的情况，并进行调试，如下图3-1是Bochs的安装完后的样子：

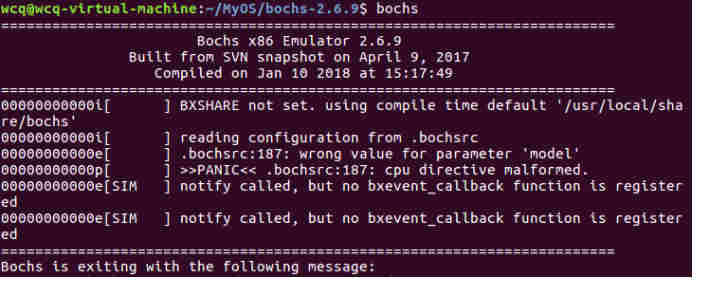


图3-1 Bochs安装结果图

## 3.2WOS的运行原理分析

### 3.2.1使用引导扇区从BIOS获取计算机权限

计算机在接电以后，运行的是基本的输入输出系统BIOS，而BIOS是内置在主办中的一个小小的程序，而因为它小，所以没法承担所有程序的运行管理工作，更无法作为操作系统，所以，它必须在完成了它的任务之后，将它的控制权利交给其他的程序，BIOS只需完成简单的初始化和检测工作即可，而它交接的下一个“选手”就是主引导记录MBR（Main Boot Record）。

MBR固定在特定的位置，也就是整个硬盘最开始的地方——0盘0道1扇区，这个扇区就是引导扇区，这里需要注意的一个地方就是，计算机界加电自检过程中，只有BIOS找到0XAA、0X55，才会将这个扇区当作是一个引导扇区，而0XAA、0X55作为为结束符的标志占有2字节，并且MBR引导扇区只能有446字节的引导程序和相关的参数，以及64字节的分区表，所以整个MBR最大不能超过512字节。

在MBR引导扇区中存储引导程序的目的，就是为了从BIOS手上结果对系统的控制权利，而MBR扮演了一个挑选下一个接班人的角色，也就是从主引导程序向次引导程序的一个过程，它帮助了原本就事情很多的BIOS完成了真正的系统权利交接任务。

### 3.2.2使用Loader加载内核的策略设计与实现

在引导扇区将BIOS对系统的控制权得到后，它要将权利给予真正可以使用该权利的模块Loader，因为Loader模块没有512字节的限制，可以很好的包括准备保护模式在内的一系列操作。

和引导扇区处理Loader进内存一样，加载内核也是先寻找文件，然后再定位文件，最后读入内存。内核文件是ELF（Executable and Link able Format）格式的，做完上述的寻址、定位以后，就是通过该格式中的Program header table值，把对应的段放到正确的地方。

这里所说的加载内核不是运行内核代码，而是将内核从硬盘上拷贝到内存中，而这需要用到loader模块了。在BIOS将权利交给MBR以后，实际上还要做准备保护模式等等一系列的操作，而这些操作如果还在512字节内完成，显然会有些紧凑，所以交给Loader。

在内核加载到内存中，首先要有一个加载地址，也就是缓冲区，这个区域只能放在内存的可用区域内，然后loader还要通过分析其ELF结构将其展开到新的位置。其中ELF结构是内核进内存的关键，所以先要对ELF结构有了解，PE是Window下的可执行文件格式而不是EXE（EXE只是后缀名，是文件名的一部分）而Linux下的可执行文件格式是ELF。表3-5是ELF文件格式布局示意图，从图中，可以很清楚的看到，ELF结构分文文件头和文件体，先用ELF hearder从全局给出程序文件的组织结构，其内容包括其他表头的大小、位置等信息。

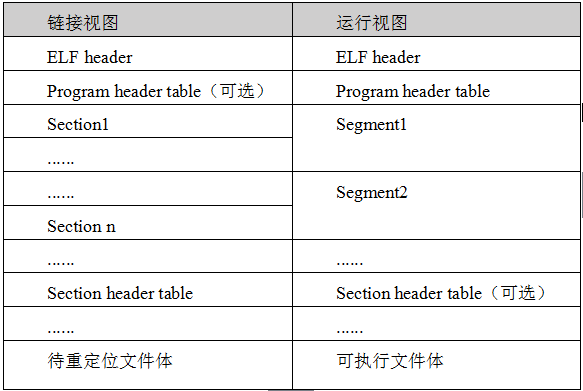


图3-5 ELF文件格式布局示意图

有了ELF结构以后，就是根据它进程内核的初始化，将内核文件中的段复制展开到内存的相应位置。因为在分页机制之下，对处理器来说，无论是数据或者是指令的地址都没区别，因为都是虚拟地址，这些虚拟地址在初始化内核阶段就规划好了，所以要将内核安排到那个区域就要将对应的内核地址编译成对应的虚拟地址，而内存中的物理地址则要看那块是空闲的就可以存放进去即可。然后就是进入到保护模式，打开分页机制，而后就是交接棒的时候，需要将系统权利由Loader交给内核，在后面将对内核进行更为详细的讲解与系统的扩充。

### 3.2.3运行内核

在内核被加载到了内核之后，第一步要做的就是进入保护模式，关于保护模式的具体内容会在下个小节进行详细介绍，这里只说明为什么要进入保护模式，因为在没有进入保护模式之前，所处的模式叫做实模式，在这个模式里，操作系统和用户程序都在同一个特权级别里，并且用户程序引用的地址都是真实地址，没有逻辑地址这一概念；用户可以自由修改段基地址，可以随时访问所以内存；一次只能运行一个程序，计算机资源极度浪费；一共只有20条地址线最大可用的内存大小为1MB，这个早就无法满足现有的用户需求了。

所以为了改善使用安全、计算机资源的充分利用以及适应大内存的需要，CPU的制造商们就开发出来了保护模式。在进入完保护模式之后，就可以开始计算机各种资源管理以及进程的调度的操作了，内核也就完运行起来了。

## 3.3保护模式研究

在上面的叙述中，多次提到了保护模式、实模式这些词语，为了说明这些概念，同时也因为它们在操作系统的是从始至终都会存在的，所以在此进行系统的说明。

### 3.3.1保护模式概述

无论是实模式还是保护模式都是针对8086CPU来讲的，它们不是指的某个具体的东西，而是CPU的寻址方式、指令的用法、工作方式、工作环境等等，是一种抽象的概念。

在保护模式之前，并没有什么实模式和保护模式之分，因为在以8086为首的16位时代，没有人会对早已经习惯的模式（工作状态、工作环境等）进行命名。但到了32位的CPU出现以后，就有了两种模式，只有在32位时，才会有实模式（运行在16位时），保护模式（运行在32位模式）。

在保护模式下，寄存器、寻址、运行模式都有了很大的改变。因为32位的寻址空间为4GB，所以寄存器由原来的16位变为了32位；在寻址上，基址寄存器不再知识bx、bp而是所有32位的通用寄存器、偏移量由16位变成了32位；运行模式由以前的单一模式，编程了实模式和保护模式互相切换转移的复杂过程。

### 3.3.2全局描述符研究

在16位的CPU时，它有16位的寄存器、16位的数据总线，以及1MB的寻址能力（需要用段：偏移的方式才能达到），而在保护模式下，一个寄存器就有4GB的寻址能力，但是仍然要保证“段基址：段内偏移”的形式，只是这个时候的段有了根本性的变化，这个段变成了一个索引，指向一个数据结构的某一项，在这个数据结构表中，详细的将段基址、段界限等内容进行了定义，这个数据结构就是全局描述符表（Global Descriptor Table）也可以是局部描述符表LDT，用来应对段信息的变多，对保护模式下的内存段进行登记，GDT中的表项有个专门的名称叫做描述符，所以更简单来说，GDT就是描述符的数组。

下图3-2就是代码段和数据段描述符的示意图，他们分别定义了段基址和段界限，为了方便展示，将8字节的段描述分为了高低两个部分：

**高32位**

**低32位**

31~16

16~0

段基址 15~0

段界限 15~0

段基址31~24

段界限19~16

段基址23~16

G

L

D/B

AVL

P

S

TYPE

DPL

31~24

23

22

21

20

15

12

19~16

14~13

11~8

7~0

图3-2 段描述符格式示意图

GDT中的每一个描述定义一个段，然后通过选择子将cs、ds等寄存器将其对应起来。所以整个段寻址方式可以用图3-3来表示。

接下来就是剩下打开A20地址线了，因为16位时只能寻址1MB，当超过该值时，就要地址回绕了，而到了32位，确实可以寻址1MB以上了，那么就需要控制什么时候地址回绕什么时候不回绕，所以就有了A20地址线来控制。开机的时候默认为关闭的，在需要到保护模式的时候就打开。

H

线性地址空间

②

①

SEG:OFFSET的线性地址

**........**

**........**

Descriptor

GDT/LDT

逻辑地址 SEG **:** OFFSET

1. 由Descriptor定义的段
2. 段内偏移OFFSET

L

图3-3 段寻址方式示意图

### 3.3.3分页机制策略分析

前面我们对于内存的分段机制有了些了解，当只有一个Loader运行时，分段的方式足够应付了，但是当有多个应用程序的时候，或者应为内存分配不当导致内存碎片过多的而没法接受新的进程，再或者硬盘中的内存段需要重新到内存中，但是找不到大小合适的内存区域怎么办？这个时候分页存储机制就解决了上面的问题。

分页机制的作用有两个方面：一是将线性地址转换成物理地址，从线性空间到虚拟空间再到物理地址空间，每个空间的大小都是4GB，在当前页的内存块被分配给了其他进程的时候，当前进程只能使用未分配的页。因为分页机制是建立在分段机制上的，所以即时在分页机制下的进程也必须首先由逻辑上的分段才可以，每一次加载进程，操作系统都要按照进程中各段的开始的范围，在进程自己的4GB虚拟地址空间中找可以使用的空间分配内存段。通过一级页表将线性地址转换成物理地址的例子就是，对基址为0x0，偏移地址为0x1234，线性地址高20位作为页表项索引：0x1234高20位是1，所以第一个页表项取到的物理地址为0x9000，线性地址低12位+物理页地址（低12位是0x234）最后和为0x9234这就是线性地址0x1234最终转换成的物理地址。

二是用大小相等的页代替不同大小的段。

启动分页机制只要按照如下顺序即可：

1. 准备好页目录表及页表
2. 将页表地址写入控制寄存器cr3
3. 寄存器cr0的PG位置1

正式因为分页机制的存在，让原本会访问物理地址的进程，现在都会通过线性地址间接的访问，这样做，就让应用程序不必再关心计算机实际有多少内存、要使用哪一段内存，从而更好的专注其他事务，这就是操作系统中分页机制所带来的变化。

### 3.3.4特权级原理分析

既然谈到了保护二字，那么最好的体现之处就是特权级了。保护模式的安全性体现在：对于具有潜在“危险”的程序，要剥夺它们的能力与权限，让它们没有用处，只能按部就班的运行。特权级按照权力从大到小分为0、1、2、3，权力越大数字越小，如下图3-4所示。其中0级是操作系统内核所在的特权级，它要直接控制硬件，并且要掌控核心数据，所以它的权利必须最大。而系统程序分别位于1级特权和2级特权，一般都是驱动程序之类的系统服务。而在最外层的3级特权就是用户程序了。总之，必须让操作系统处在最高级的地位，这样才能控制所有事物。

011

0

1

2

3

内核

系统程序

用户程序

权力随着数字

越大逐渐降低

图3-4 特权级示意图

为了让CPU知晓谁的特权级高，所以有了DPL、CPL和RPL这三个计算机特权级标签，就如同为了知道谁的学历高，用本科生、研究生、博士生三种证书来区分的道理一样。

处理器的当前特权级（CPL）是指：在CPU中运行的指令，而只要是指令就必定会属于某个代码段，该代码段描符中的DPL，就是当前CPU所处的特权级。除了某些特殊情况，一般转移后的目标代码段的DPL就是将来处理器的当前特权级CPL，在任何时刻，CPL都保存在CS选择子的RPL部分。在当前正在运行的代码段的特权级DPL就是当点CPU的特权级，当代码段在CPU上运行，由一个变为另外一个时，当前特权级就发生了改变，所以，目标代码段的DPL就保存在了代码段寄存器CS的RPL位。

而DPL作为描述符特权级，它表明了内存的特权等级。计算机的世界有很多和人类社会相似之处，在学校里，老师有权利查看学生成绩、分配各个班委的职责，而班长也可以分配部分班委的职责，但是班长没法查看成绩；同样的道理，DPL是段描述符所代表的内存区域的“门槛”权限，作为“访问者”的指令和数据，只有当它们的权限大于等于该DPL表示的最低的特权级才能继续访问。

但是，凡事都有例外，当处理器从中断处理程序中返回到用户态的时候就是这个例外。因为中断的发生多半都是在外部硬件发生了某种状况或发生了某种不可抗力时间而必须通知CPU导致的，所以在中断的处理过程中，需要具备访问硬件的能力，而访问硬件的特权级必须是0级，所以，在用户进程被中断的时候，CPU暂停用户进程，并且保留用户进程的上下文，进入中断程序，待问题被解决之后，CPU再回复上下文，这其中的特权级变化就是由用户进程的3级特权级变为0级的中断处理，然后再恢复为0级特权级。

但有时必须执行高特权级的指令，但是又不能编程高特权级，这个时候，一致性代码段的诞生就解决了这个需求，它能用来实现从低特权级的代码向高特权级代码的转移。

对处理器来说，操作系统只是它的应用而已，而处理器只有通过“门结构”才嫩实现有低特权级向高特权级的转移，而“门结构”仅仅就是记录一段程序起始地址的描述符。

下面通过一个列子将特权级进行形象的说明：有个小学生M（用户进程）特别喜欢飞机，所以他想靠个飞行员驾驶证，但是训练基地的保安（调用门）以年龄不够拒绝让他进行学习，于是M找了他的叔叔K（内核）帮他报名，保安就K放行，然后K到了招生办公室，办公室的工作人员对K说帮别人报名也行，但是要有对方的身份证（RPL），于是K就把M的身份证拿出来了，别人一看，不够年龄，也拒绝了，并抛出异常。

同时在I/O特权级上，保护模式也有相关操作。I/O的读写特权是由标志寄存器eflags中的IOPL位和TSS中的I/O位图决定的，它们用来指定执行I/O操作的最小特权级。

### 3.3.5中断策略研究

中断作为计算机时间里的常见现象，具体来说就是当处理器在执行某一程序时，出现了一些外部硬件，或者不可抗力的突发状况时，暂停当前的操作，转而进行事件的处理，处理完了以后，再回复之前的状态，继续执行，这整个过程，就叫做中断处理，也叫做中断。中断多为硬件的问题，而异常多为处理器检测到指令错误。虽然中断打断了当前的事情，但是在整体上来说，它的出现，整体提高了计算机的效率，因为有了中断的出现，系统才能并发的运行，我们才能“同时”享受计算机的多种服务。

为了让处理器周而复始的运作，同时又要保证处理器不会成为脱缰的野马，所以，操作系统是一个死循环，而中断的出现，就可以很好的驱动操作系统了，操作系统就是被动的工作，在需要的时候才会工作，所以它是被事件驱动的。

中断可以分为外部中断和内部中断两大类。

外部中断其实就是硬件中断，CPU提供了INTR（INTeRrupt）和NMI（Non Maskable Interrupt）两根信号线来通知CPU外部中断。

而内部中断就是软中断和异常了。软中断是又软件发起的中断，是软件本身主动发起的，所以不能算作是错误，我们可以用指令“int 8位立即数”、“int3”（机器码为0xCC）、into、bound等来发起中断指令。异常是内部中断，是指令在执行的过程中，CPU内部产生错误引起的。安装异常的轻重等级可以分成：fault（成为故障，是可以被修复的，是最轻的一种异常）、trap（顾名思义，这个陷阱是软件掉进了CPU的陷阱之中，主要是int3指令引发的）、abort（称为终止，由于错误无法被修复，也无法进行操作，所以操作系统为了自保，只能将程序从进程表中去除，是程度最重的一种异常类型）。

在CPU内的中断过程中，首先处理器根据中断向量号定位到中断门描述符，中断向量号是中断描述符的索引，当处理器收到一个外部中断向量号以后，它用此向量号在中断描述符表中查询对应的中断描述符，然后再去执行该中断描述符中的中断处理程序。然后再进行特权级的检查，主要有一下2步：

1. 若由软中断intn、int3引发的中断，则是由用户进程主动发起的中断，由用户代码控制，处理器要检测当前特权级CPL和门描述符DPL，这事检查进门的特权下限，若CPL权限大于等于DPL，则进入下一步检查，否则处理器抛出异常。
2. 然后处理器要检查当前特权级CPL和门描述符中的选择子对应的目标代码段DPL，若CPL权限小于目标代码段DPL，检测通过，否则处理器抛出异常。

特权级检查完后，最后再执行中断处理程序，将门描述符目标代码段选择子加载到代码段寄存器CS中，把门描述符中中断处理程序的偏移地址加载到EIP，开始执行中断处理程序。图3-6就是中断处理过程。

而对与外部中断来说，外部设备的中断则是由中断代理芯片接收，处理后将该中吨的中断向量发送到CPU。而这部分要通过可编程中断控制器8259A来完成，当某个外部设备发出一个中断信号的时候，由于主板上已经将信号通路指向8259A芯片的某个IRQ接口，所以该中断信号最终被送到8259A中了。

中断向量号x8

IDTR

中断向量号

内存

GDT

IDT

4

9

8

7

⊕

⊕

处理器

1

第0个门描述符（可用）某门描述符

**...**

中断门描述符

中断处理程序所在代码段的选择子

中断处理程序所在代码段的偏移量

段描述符0（不可用）

内核代码段描述符

内核代码段

的起始地址

段描述符X

**...**

段描述符N

XX段

内核代码段

GDTR

+选择子高13位x8

2

3

6

5

图3-6 中断处理过程图

### 3.3.6保护模式的小结

经过上述的说明，可以将进入保护模式的主要步骤就可以归纳为如下：

1. 准备GDT
2. 打开A20
3. 设置cr0的PE位
4. 跳转，进入保护模式

保护二字的含义总结：

首先，对每一个对象，都由对应的描述符进行描述，描述的内容包括界限、大小、位置等等，这使得对象本身有了一个约束，同样得到了保护。

其次，在分页机制的作用下，每当使用内存时，不再直接控制物理内存地址，而是通过虚拟地址的映射方式，使得内存的使用上得到了很好的保护，也提升了计算机的使用效率。

接着，在特权级的概念出来以后，从用户进程到系统进程，都本监控起来，不再是完全的平等，而是进行了权级的标签进行权力的分层，很好的保护了计算机不被随意使用，进行了一个监管。从以上部分可以说明保护模式的含义，但是肯定不是全部，同时也让我们对处理器为操作系统提供的强力的硬件平台感到佩服。

# 4操作系统内核的设计与实现

## 4.1内核整体设计

在前面的准备工作做好了之后，接下来的操作系统内核的开发设计都会水到渠成。从计算机通电之后BIOS开机自检，而后找到MBR引导扇区，然后引导扇区突破512字节的界限加载Loader，同时进入到保护模式，然户Loader模块加载一个简单的内核之后，系统就成功运行了，这些就是我们在准备工作期间所完成的工作，而接下来要做的就是内核的完善了，主要就是下面的5个模块进行。为了更好的做下面的开发工作，首先将整个开发的文件结构理清，这对全局的把握大有好处，图4-1就是本次毕设内核开发的文件目录结构图：

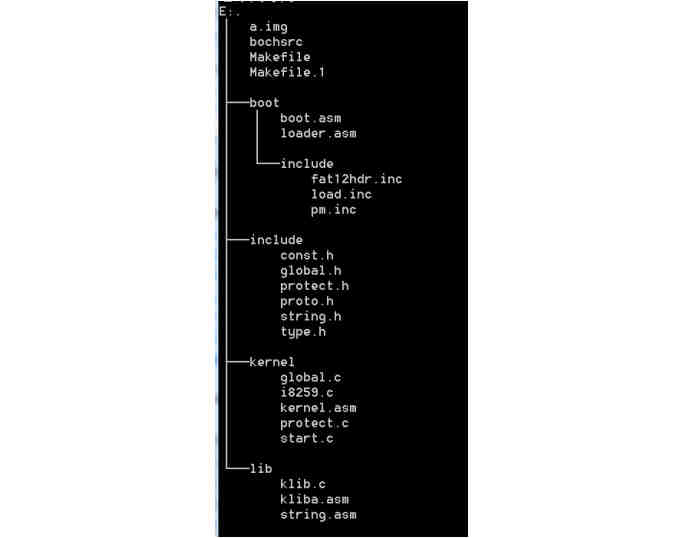


图4-1 内核目录文件结构图

有了上面的结构图，就更加清晰的知道接下来的实现过程了。操作系统的开发流程主要分为如下六个部分：

1. BIOS将引导扇区读入内存
2. 开始执行引导代码
3. 引导代码找到Loader
4. Loader再将内核找到，并读入内存
5. 执行内核代码
6. 系统运行

其中最为重要的部分、也是本次毕业的核心设计与实现就是内核部分，而前面的引导工作都是操作系统与计算机衔接的工作。在后面的内核开发中会重点说明内核各个部分开发的具体情况，这里不再赘述。

而在实际开发过程当中，具体到每个部分时，会有如下的一个开发过程：

1. 编写C与汇编源码，以及Makefile文件
2. 再由各自对应的编译器将源码编译成对应的机器语言
3. 将机器语言链接并生成相应的为虚拟软盘映像文件
4. 将映射写入磁盘文件（硬盘或者虚拟软盘），做成一个操作系统启动文件

## 4.2进程管理设计与实现

因为进程数是多余CPU数的，于是在同一时刻，总是有“正在运行的”和“正在休息的”进程。所以，对于“正在休息的”进程，我们需要让它在重新醒来的时候记住自己挂起之前的状态，以便让原来的任务继续执行下去。  
 所以，我们要一个数据结构记录一个进程的状态，在进程要被挂起的时候，进程信息就被写入这个数据结构，等到进程重新启动的时候，这个信息重新被读出来。如图4-2所示：

CPU

进行A（运行中） 进行B（休息中） 进行C（休息中）

进程体 进程体 进程体

进行A的状态 进行B的状态 进行C的状态

图4-2 进程示意图

### 4.2.1进程运行算法实现

首先准备一个简单的进程：

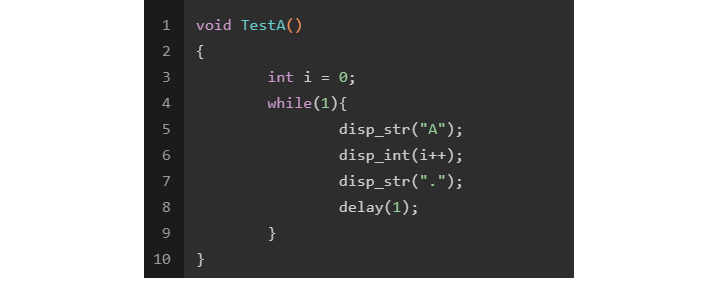


图4-3 进程A代码

这个进程体（函数）的功能就是打印一个字符并显示数字，并稍微停顿。

接着我们要注视掉hlt，并让程序跳转到kernel\_main()中。delay()函数也就是一个循环嵌套。

接着进行初始化进程表，关键代码如下：

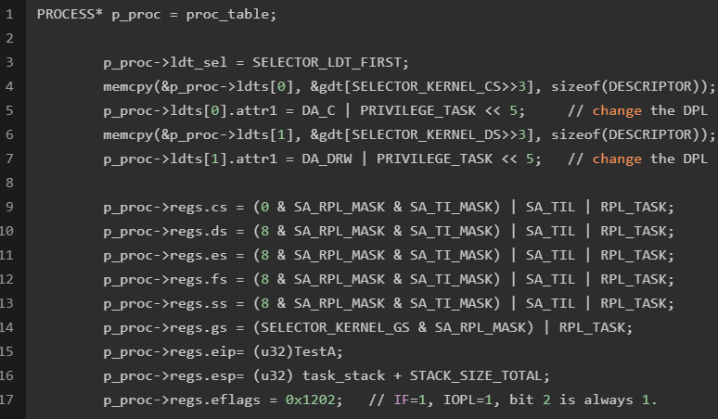


图4-4 进程A的进程表初始化

然后启动后，运行结果如下：

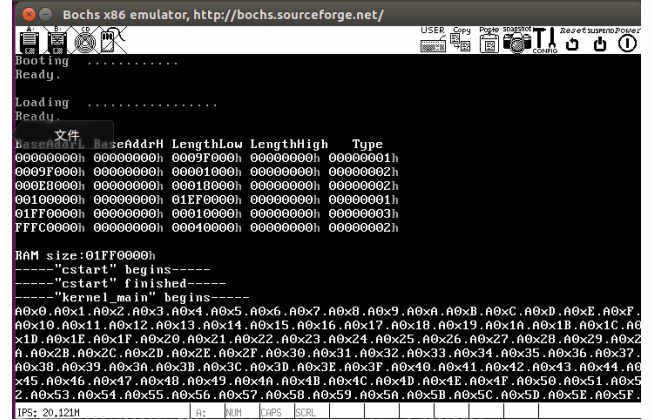


图4-5 单进程运行结果图

上面进行单进程，但是显然现今都是多进程的时代了，所以为了体现出多进程的特性，我们再添加一个进行：

图4-6 进程B代码



接着进行时钟中断处理，关键代码如下：

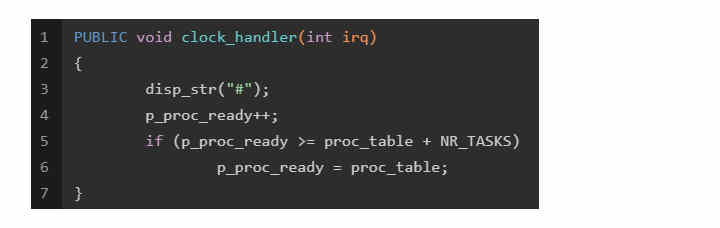


图4-7 时钟中断代码

最后就能两个进程进行切换了，如图是AB交替出现的运行结果：

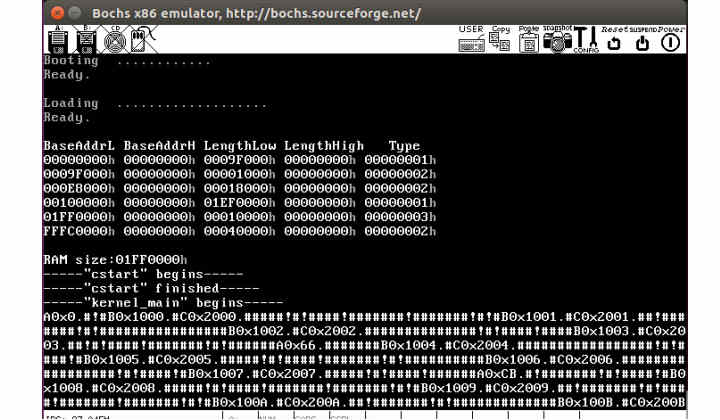


图4-8 多进程运行结果图

### 4.2.2时钟中断算法实现

系统调用和API类似，当应用程序很多事做不了的时候，只能交给操作系统来做。所以一个事情，可能应用程序做了一部分，操作系统做一部分，这就涉及到特权级的问题了（前面已经说明特权级问题）。下面图4-9是本操作系统的运行过程，其中包含了时钟中断最简单的算法，首先要有init\_all函数初始化所有的设备及数据结构，而init\_all将先调用idt\_init，初始化中断相关的内容，，而idt\_init中会调用pic\_init函数初始化8259A后就可以加载IDT，也就说明打开中断的条件就准备好了。

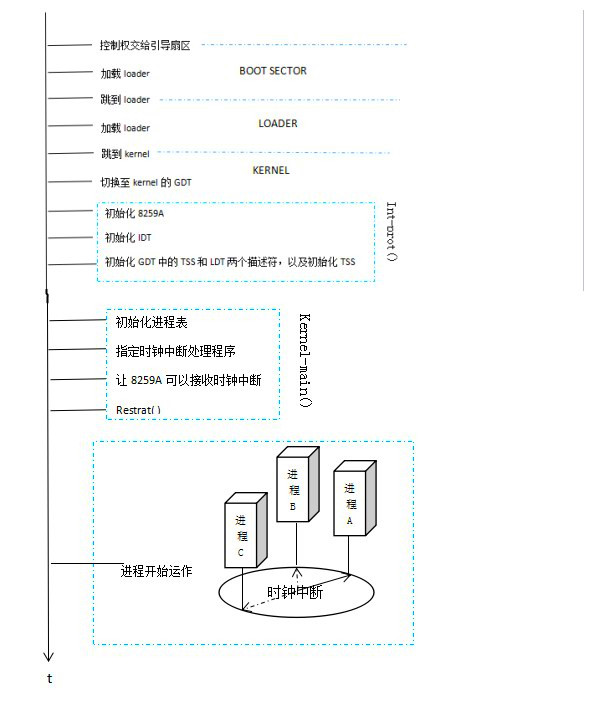
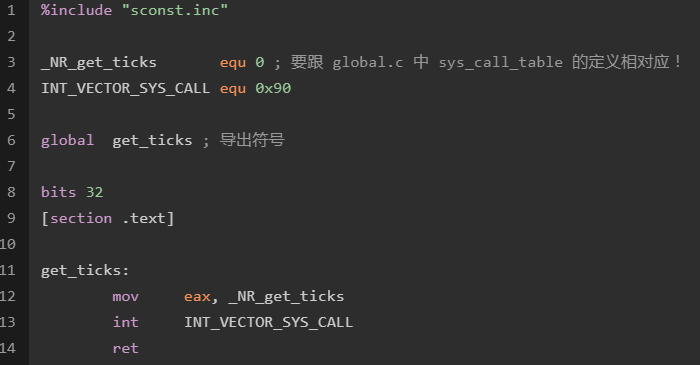


图4-9 多进程运行结果图

我们通过实现get\_tick()得到当前总共发生多少次时钟中断。设置一个全局变量ticks，每次发生一次时钟中断，它就加1,进程可以随时通过get\_tick()这个系统调用来得到这个值。关键代码如下：



### 4.2.3进程优先级调度算法实现

进程调度的方式有很多，而咋本次毕设中，是用的基于优先级的简单调度。

前面的进程延迟相同，现在将其改变下A、B、C三个的延迟分别为300、900、1500ms，运行结果如下

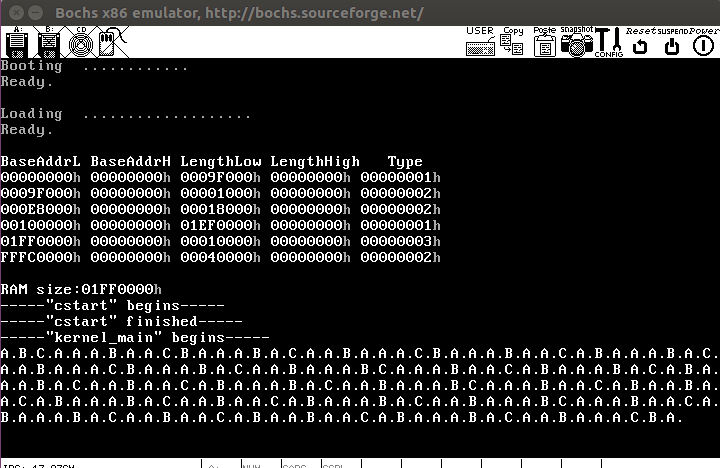


图4-10 普通的程调度运行结果图

从这个我们可以想到，通过延迟的不同设置不同的优先级。  
通过“轻重缓急”反应在时间上，来表达优先级调度，最重要的事情应该被赋予更高的优先级，应该给予更多的时间。

我们给每一个进程都添加一个变量，在一段时间 的开头，这个变量的值又大又下，进程获得一个运行周期，这个变量就减1，当减到0，此进程就不再获得执行的机会，指导所有进程都为0。  
 由于每一次进程调度的时候只有某一个进程的ticks会减少1，所以总共调度的次数应该是3个进程的ticks之和（150+50+30）=230。所以:

1. 进程A执行循环的次数为：（100+20x2+30x3）/20=230/20=11.5次
2. 进程B执行循环的次数为：（0+20x2+30x3）/20=230/20=6.5次
3. 进程C执行循环的次数为：（0+0x2+30x3）/20=230/20=4.5次

将各个进程的延迟时间改为10m后，make一下，运行如下：

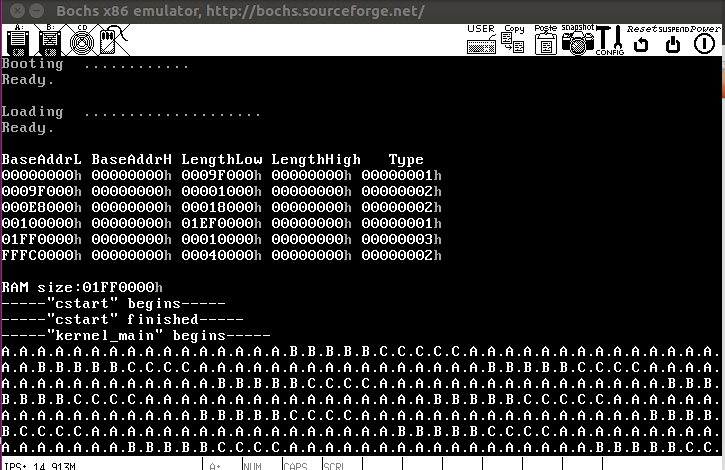


图4-11 基于优先级进的程调度运行结果图

总的来说，优先级调度方式来源于现实社会的轻重缓急，最重要的事给予最多的资源与时间。

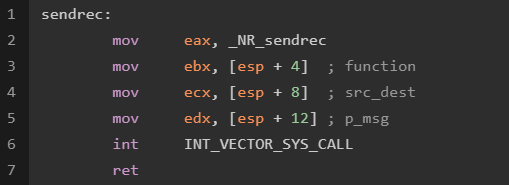
### 4.2.4进程间通信策略对比分析及实现

在宏内核的时候，用户进程直接调用内核进行工作，方式十分直接，但是到了微内核以后，工作方式就复杂的多，这时候的内核就如同一个中介，任何的系统调用都会通过它作为转移信息，这时候，进程间的通信就显得十分重要了。

IPC是Inter—Process Communication的缩写，是进程间通信的意思，就是进程间发消息。IPC有同步和异步之分，同步是消息发送发和消息接收方都会一直等待消息，它的好处是：

* 1. 操作系统不需要另外维护缓冲区来存放正在传递的消息
  2. 操作系统不需要保留一份消息副本
  3. 操作系统不需要维护接受队列
  4. 发送者和接受者都可以在任何时刻清晰且容易的知道消息是否送达
  5. 从实现系统调用的角度看，同步IPC更加合理，当使用系统调用时，我们的确需要等待内核返回结果之后再继续

IPC的机制已经清楚了，它的核心在int SYSVEC这个软中断以及与之对应的sys\_call()这个函数。下面是新的系统调用sendrec的核心代码:

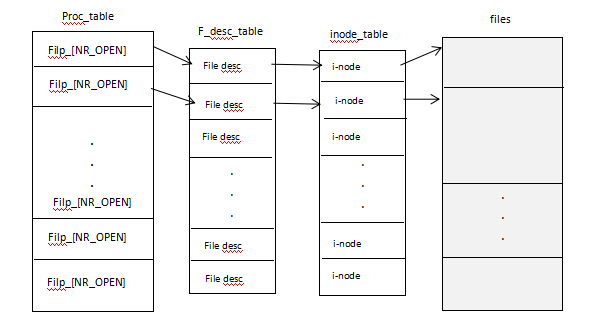


4.3文件系统设计与实现

### 4.3.1创建文件的系统调用算法实现

对文件进行创建以及读写等操作，需要用到open()、write()、read()、close()等系统调用，而这些熊调用都用到了一个变量——fd，文件描述符（file descriptor）。如下是它的结构图：

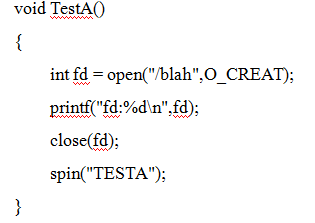
图4-12 文件描述符结构图



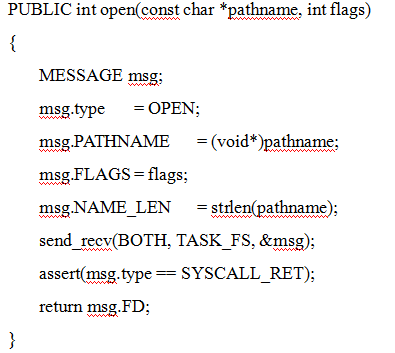
其中fd\_mode用来记录这个fd是用来做什么操作的。fd\_pos用来记录读写到了文件的什么位置。fd\_inode便是指向inode的指针。

每当一个进程打开一个文件——无论是打开一个已经存在的还是创建一个新的，该进程的进程变filp数组就会分配一个文职——假设是k，用于存放打开文件的fd指针，而这个k就是返回给用户进程open()函数的返回值了。

open()现在用户进程中创建一个文件：



下面是open()的系统调用：



创建文件所设计的其他函数如下：

strip\_path()该函数用于把路径分为文件名和文件夹两个部分，定位直接包含文件的文件夹，并得到给定文件在此文件夹中的名称。

search\_file()用来得到文件所在目录的i-node，通过这个i-node来得到目录所在的扇区，然后读取这些扇区，查看里面是否有我们要找的文件，如果找到就返回文件的i-node，如果没有就返回0.

do\_clode()关闭文件显得十分简单，CLOSE消息是由do\_close()来处理。  
完成了open()和close()两个系统调用，我们接着运行一下结果如下：

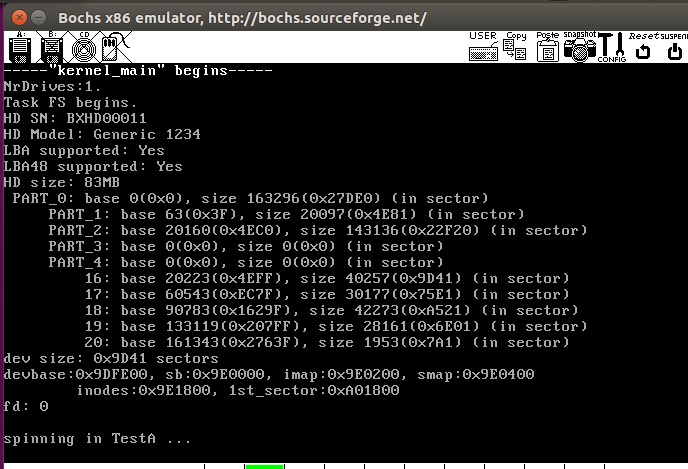


图4-13 文件的创建图

可以看到进程TestA打印出了新创建的文件的fd:0。

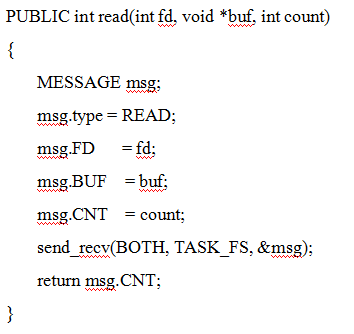
### 4.3.2文件的打开与关闭算法实现

关闭在上面已经说了就不再赘述，打开文件其实就是根据文件名找到i-node，并且建立进程表、f\_desc\_table[]和inode\_table[]之间的关联。对于普通文件而言，打开操作有以下情况：

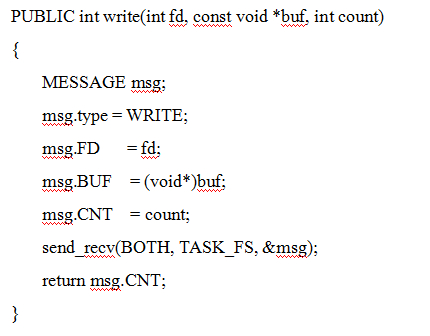
1. 文件存在。这时我们获得文件的i-node号，读出i-node，建立前面所述三表的关联，并返回fd。
2. 文件不存在。直接返回-1。
3. 文件不存在。创建文件，建立前面所述的关联，并返回fd。

### 4.3.3文件的读写算法实现

因为采取一次分配的原则，所以读写变得比较简单，但是这个方式并不好。在读写的过程中，我们任然是把它扔给相应的驱动程序——虽然驱动程序并未准备还处理，但发送一个消息只是举手之劳。下面是read()函数：



下面是write():



如图4-14是文件的写入结果，从中可以很明显的看到成功写入，并且最终读取出来了abc字符，证明文件的读写成功。：

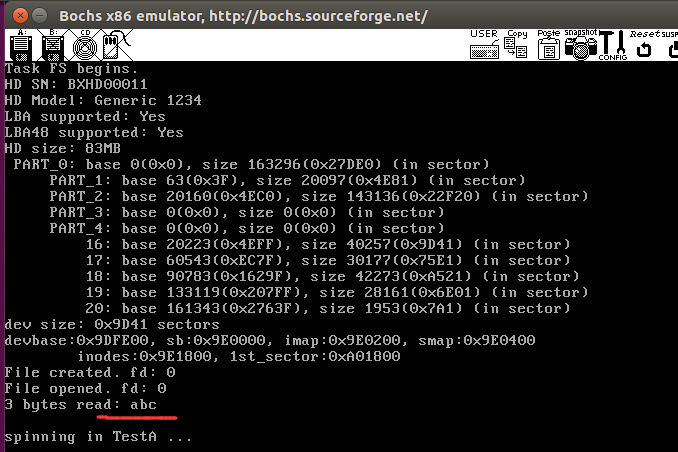


图4-14 文件的写入图

### 4.3.4删除文件算法实现

删除是添加的反过程，以下是要做的工作，最后的结果如下图4-15所示：

1. 释放inode-map中的相应位
2. 释放sector-map中的相应位。
3. 将inode\_array中的i-node清零
4. 删除根目录中的目录项

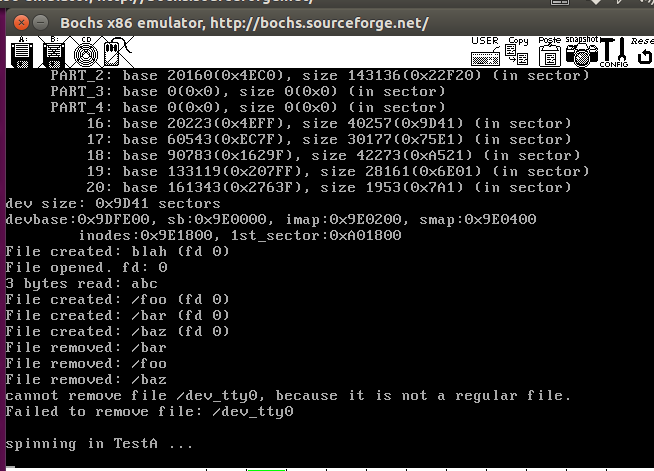


图4-15 文件的删除图

## 4.4系统交互与I/O管理设计与实现

### 4.4.1fork的算法设计与实现

fork前要做好的工作：

1. 在proc\_table[]中预留出一些空白项，供新进程使用
2. 在proc\_table[]中的每一个进程表项中的idt\_sel项都设定好
3. 将进程所需的GDT表项都初始化好

生成一个子进程的系统调用被成为fork()，操作系统接到一个fork请求后，会将调用者复制一份，这时就会有两个一模一样的进程同时进行。其中子进程是从父进程得到数据、堆栈以及代码而来的。

最先要解决的问题是，谁作为最开始的父进程？参考Linux以及Minix可以知道是init进程，所以我们先写出Init进程：

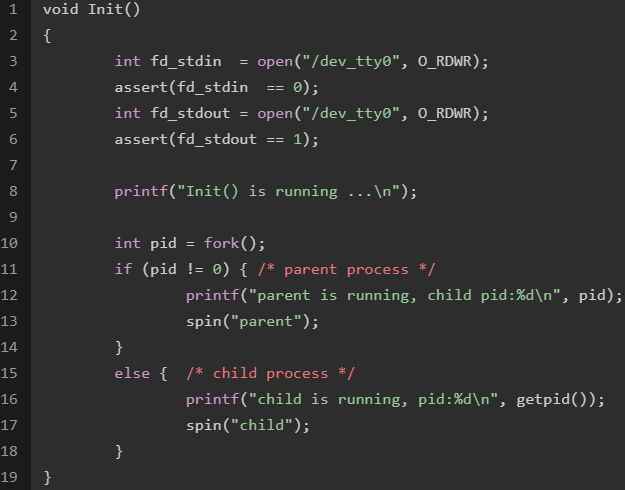


图4-16 初始化进程核心代码

可以看到其中调用了即将实现的fork()，而且判定了返回值。如果返回0则表明自己是子进程，否则返回的是子进程的pid并表明自己是父进程。

然后我们要增加MM进程，它将负责从用户进程接受消息。MM和文件管理进程一样，MM要有一个主消息循环。

当MM接到FORK消息后，调用do\_fork()来处理。在内存的分配上，我们采用固定内存的方式，每个内存块大小为1MB。fork()系统调用的结果如图：

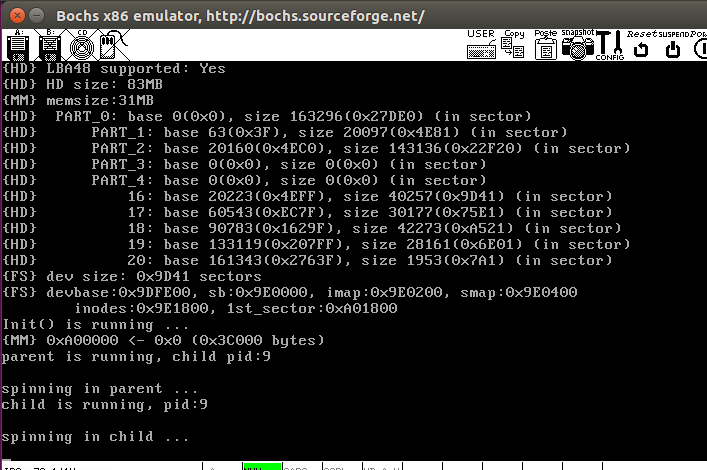
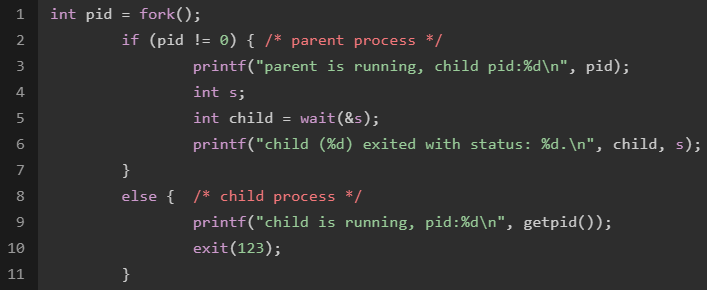


图4-17 fork()系统调用的结果

### 4.4.2进程消亡的系统调用算法设计与分析

生成子进程最重要的是fork()，而进程的消亡则是用到系统调用exit()。而系统调用wait()是父进程得到返回值的方法，用该系统调用挂起，等子进程退出时，wait()调用方结束，并且父进程因此得带返回值。具体代码如下：



和fork()类似，上述两个系统调用同样是发送消息给MM，它们发送的消息分别是EXIT和WAIT。并由MMM中对应的消息函数进行处理。do\_exit/do\_wait和msg\_send/msg\_receive这两对函数是类似的例如，假设进程P有子进程。而A调用exit()，那么MM会：

1. 告诉FS，A退出，请做出相应处理
2. 释放A占用的内存
3. 判断P是否正在WAITING
4. 遍历proc\_table[]
5. 如果P的子进程没有一个在HANGING，则设P的WITING位
6. 如果P没有子进程，则向P发送消息，消息携带一个表示出错的返回值。

完成了这以后，子进程的产生和消亡都有了，运行一下，结果如图4-18，很明显child子进程显示退出也就是消亡：

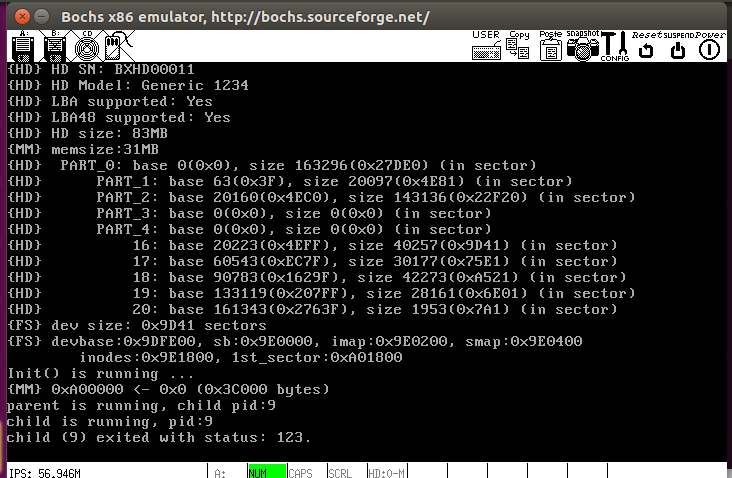


图4-18进程消亡图

### 4.4.3实现exec算法

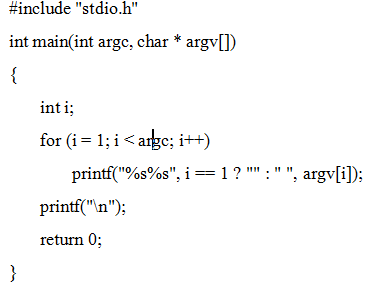
exec的语义很简单，它将当前的进程映像替换成另一个。也就是说我们可以从硬盘上读取另一个可执行的文件，用它替换掉刚刚被fork出来的子进程，于是被替换的子进程就成为了新进程。

有了上述铺垫，下面开始为自己的操作系统编写应用程序

例如Linux中的echo，它和操作系统的接口是系统调用。本质上，一个应用程序只能调用两种东西：属于自己的函数，以及中断。写一个echo最笨的方法就是将send\_recv()、printf()、write()等所有用到的系统调用的代码都复制到源文件中，然后编译下。而更好的做法是制作一个类似C运行时库的东西，我们把之前已经写好的应用程序可以使用的库函数单独链接成一个文件，每次写应用程序时的时候直接链接起来就好了。

到目前为止，可以被用来链接成库的文件及其包含的主要函数有这些：

1. 真正的系统调用：sendrec和printx:lib/syscall.asm
2. 字符串操作：memcpy、memset、strcpy、strlen：lib/string.asm
3. FS的接口：lib/open.c lib/read.c lib/write.c lib/close.c lib/unlink.c
4. MM的接口：lib/fork.c lib/exit.c lib/wait.c
5. SYS的接口：lib/getpid.c
6. 其他：lib/misc.c lib/vsprintf.c lib/printf.c  
   把这些函数单独链接成一个库，起名为orangescrt.a表明这是我们的C运行时库.  
   现在先写一个最简单的echo：



然户就是安装应用程序，而安装程序到我们的文件系统中，需要做以下工作：

1. 编程应用程序，并编译链接
2. 将链接好的应用程序打成一个tar包：inst.tar
3. 将inst.tar用工具dd写入磁盘的某段特定扇区
4. 启动系统，这时mkfs()会在文件系统中建立一个新文件cmd.tar，它的inode中的i\_start\_sect成员会被设为X
5. 在某个进程中将cmd.tar解包，将其中包含的文件存入文件系统。  
   运行结果如下，成功显示出hello world：

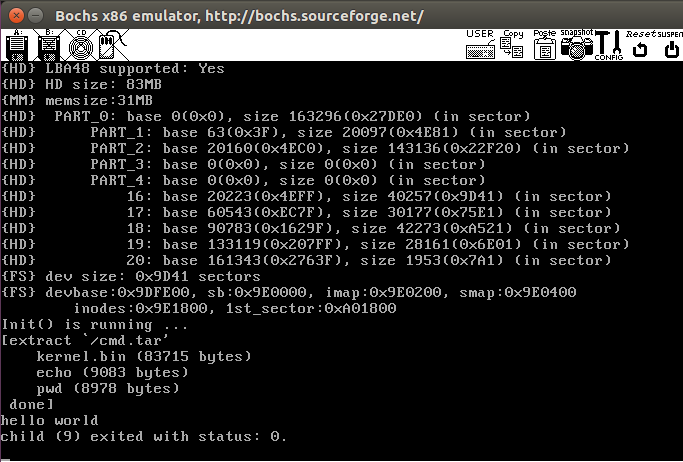


图4-19echo安装图

### 4.4.4键盘、显示器驱动设计

键盘操作系统需要交互，首先就是键盘，下面从中断开始：键盘的初体验现在我们写一个键盘的处理程序：

PUBLIC void keyboard\_handler(int irq)  
{  
 disp\_str("\*");  
}

这个结果就是每按一次键盘，就会打印一个星号。然后添加中断处理程序，并打开键盘中断：

PUBLIC void init\_keyboard()  
{  
 put\_irq\_handler(KEYBOARD\_IRD,keyboard\_handler);/\*设定键盘中断程序\*/  
 enable\_irq(KEYBOARD\_IRQ);/\*开键盘中断\*/  
}

然后在proto.h中声明init\_keyboard()并调用它。

处理shift、alt、ctrl先对这三个按键的状态进行判断，因为有左右之分，所以有6个键，当按下shift\_l时，相应的变量就变为TRUE,如果立即释放则变为FALSE。如果if(shift\_l||shift\_r)成立，则表示左shift被按下且未被释放，此时colum值变为1。运行一下：

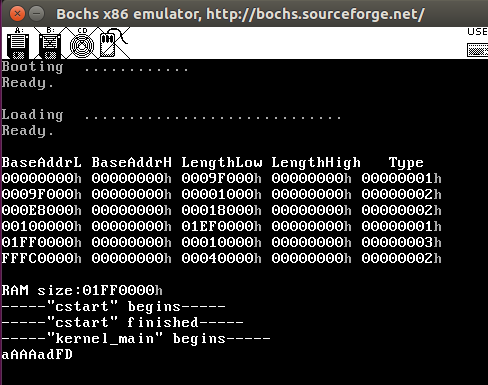


图4-20键盘区分大小写输入结果图

对于其他功能键，我们统一放在in\_process()中，并在不可打印的字符的定义中，都加一个FLAG\_EXT。运行结果如下：

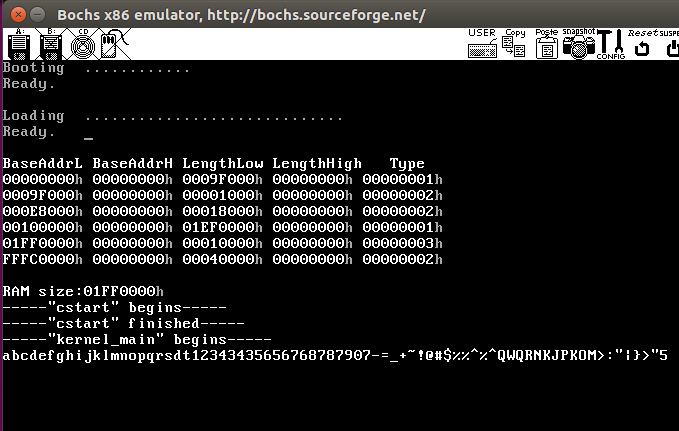


图4-21全键盘结果图

接下来就是对输入输出设备中的显示器说明：

初识TTY。TTY在Linux中就是终端。当按下ALT+F1、ALT+F2、ALT+F3等组合键时，会切换到不同的屏幕。对于不同的TTY可以理解成下面：

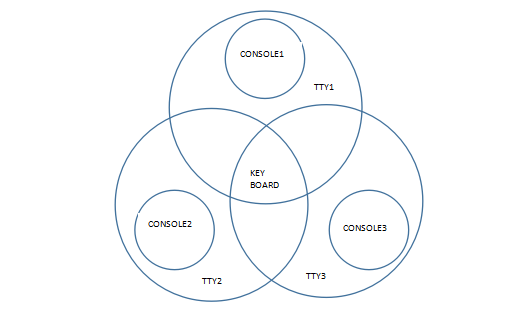


图4-22 TTY图

虽然不同的TTY对应的输入设备是同一个键盘，但是输出却好比是在不同的显示器上，因为不同的TTY对应的屏幕画面可能不同。实际上，画面的不同，仅仅是显示了显存的不同位置罢了。

既然3个CONSOLE公用一块显存，那就有一种方式在切换CONSOLE的瞬间，让屏幕显示显存中某个位置的内容。

### 4.4.5 多控制台算法实现

在TTY任务中执行一个循环，这个循环将轮询每一个TTY，处理它的事件，包括从键盘缓冲区读取数据、显示字符等。它的运行方式如下：在TTY任务中执行一个循环，这个循环将轮询每一个TTY，处理它的事件，包括从键盘缓冲区读取数据、显示字符等。

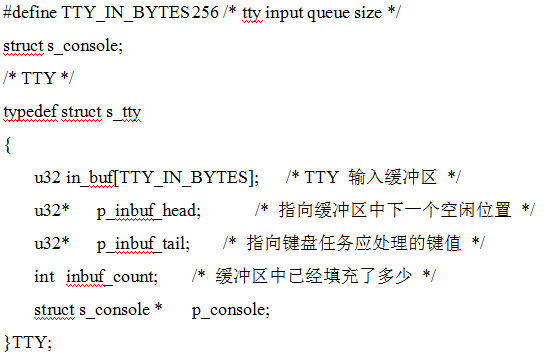
其实轮询到每一个TTY时，不外乎做两件事：

处理输入：查看是不是当前TTY，如果是则从键盘缓冲区读取数据。

处理输出：如果有要显示的内容则显示它。  
下面要做的TTY任务不再简单，主要表现为：

1. 每一个TTY都应该有自己的读和写的动作。所以在keyboard\_read（）内部，函数需要了解自己是被哪一个TTY调用。我们通过为函数传入一个参数来做到这一点，这个参数是指向当前TTY的指针。
2. 为了让输入输出分离，被keyboard\_read（）调用的in\_process()不应该再直接回显示符，而应该将回显的任务交给TTY来完成，这样我们就需要为每个TTY建立一块缓冲区，用以放置将被回显的字符。
3. 每个TTY回显字符时操作的console是不同的，所以每个TTY都应该有个成员来记载其对应的console信息。

下面进行简单的TTY任务框架的搭建TTY结构如下：



整个程序的流程如下：

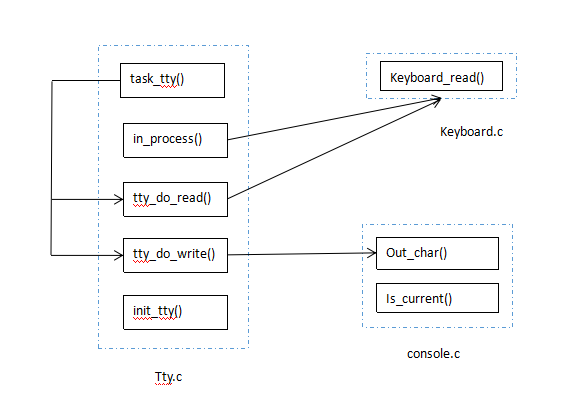


图4-23多控制台运行示意图

在task\_tty()中，通过循环来处理每一个TTY的读和写操作，读写操作都放在了tty\_do\_read()和tty\_do\_write()两个函数中，这样就让taske\_tty()很简洁，而且逻辑清晰。读操作会调用keyboard\_read()，当然此时已经多了一个参数；写操作会调用out\_char()，它会将字符写入指定的CONSOLE。，当TTY任务开始运行时，所有TTY都将被初始化，并且全局变量nr\_current\_console会被赋值为0.然后循环开始并一直进行下去。对于每一个TTY，首先执行tty\_do\_read（），它将调用kerboard\_read()并将读入的字符交给函数in\_process（）来处理，如果是需要输出的字符，会被in\_process（）放入当前接受处理的TTY的缓冲区中。然后tty\_do\_write()会接着执行，如果缓冲区中有数据，就被送入out\_char显示出来。

# 5总结与展望

## 5.1工作总结

本次毕设总的来说主要以研究操作系统的内核为目的，并在此基础上完成一个内核的实现。操作系统内核最终虽然较为简陋，但是该有的东西都有，包括多进程的管理、多任务台的控制、文件的管理、内存与系统交互的管理等等。

在本次毕设的整个过程中，因为难度很大，需要学习的东西很多很庞杂，多次想要放弃，但是最终都坚持了下来，而坚持下来的以后，确实收获了很多，不仅对整个计算机从下至上有了更加全面的认识，更对本科期间所学有了更加深刻的认识，具体来说，有下面一些收获：

1. 进过资料的查阅对操作系统的发展有了更为清晰的认识，以及未来的操作系统发展有了自己的见解。
2. 对操作系统的基本工作原理进行研究，并对底层硬件与上层软件之间的协调工作有了更加具体的认识，以及如何对CPU进行部分操作有了更进一步的体会。
3. 对操作系统的四大管理和一些功能有了更为深刻的认识，并切实的落在代码层面上而不是理论分析。
4. 对毕设所用的C语言以及汇编语言有了更熟练的掌握与理解，同时对先前汇编课程中的保护模式有了实际操作，所以理解更为透彻。
5. 对系统调用有了进一步的认识，并能自己编写简单的系统调用，同时对键盘、显示器等输入输出设备与寄存器、CPU的交互有了全新的体会。

## 5.2展望

虽然本次毕设将一个内核实现了，并能成功在虚拟机上运行，但是还有些地方可以在今后的开发中进行完善和增加，主要包括一下方面：

1. 本毕设的下一步就是要增加图形界面，虽然这不是操作系统最核心的地方，但是对普通用户来说，这事最必不可少的需求。而本次毕设没有图形界面，只有命令行，这是因为要写图像界面所花的时间太长，而我将精力放在了操作系统内核部分，所以没来的及做这方面的工作，今后会在这块内容进行增加。
2. 还要针对各个模块的内容进行优化，并且丰富各个模块的功能，包括对系统程序的补充扩展、文件管理模式的优化、内存碎片更加合理的处理等等，可以对Linux等大型操作系统的优化设计进行学习参考。