

单位代码: 10293 密 级:           

南京邮电大学

# 专 业 学 位 硕 士 论 文



论文题目: 轻量级操作系统的设计与开发

学 号 1216012118

姓 名 冯小建

导 师 马明栋

专 业 学 位 类 别 工程硕士

类 型 全日制

专 业 ( 领 域 ) 电子与通信工程

论 文 提 交 日 期 2019.02

# **Lightweight Operating System Design and Development**

Thesis Submitted to Nanjing University of Posts and  
Telecommunications for the Degree of  
Master of Engineering



By

Xiaojian Feng

Supervisor: Prof. Mingdong Ma

April 2019

## 南京邮电大学学位论文原创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了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，也不包含为获得南京邮电大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

本人学位论文及涉及相关资料若有不实，愿意承担一切相关的法律责任。

研究生学号：\_\_\_\_\_ 研究生签名：\_\_\_\_\_ 日期：\_\_\_\_\_

## 南京邮电大学学位论文使用授权声明

本人授权南京邮电大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档；允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索；可以采用影印、缩印或扫描等复制手段保存、汇编本学位论文。本文电子文档的内容和纸质论文的内容相一致。论文的公布（包括刊登）授权南京邮电大学研究生院办理。

涉密学位论文在解密后适用本授权书。

研究生签名：\_\_\_\_\_ 导师签名：\_\_\_\_\_ 日期：\_\_\_\_\_

# 摘要

操作系统是管理和控制计算机硬件和软件资源的计算机程序，也是计算机硬件和其他软件之间的接口。虽然操作系统技术日益成熟，但仍然是一个不断发展，不断更新的领域。然而，国产操作系统的发展相对落后，目前难以与国际知名的操作系统相竞争。**Linux** 是国际上常用且强大的操作系统，它拥有着自由、公开且免费的特性，为人们提供了学习优秀国外操作系统的设计理念和实现方法的机会。因此，研究和改进 **Linux** 操作系统对于国内操作系统的发展具有重要意义。

此外，当前市场上大多数的操作系统内核都是由 C 语言和汇编语言混合编程，主要基于 C 语言。而由 C 语言构成的大型项目往往会面临功能与模块关系不清晰、代码的复用性和维护性较差等问题。在当下 **Linux** 代码急速膨胀的趋势下，解决代码的维护问题尤为重要。因此，由面向对象语言实现的操作系统内核将会体现出明显的优势，具有重要的现实意义和应用价值。

本文将从引导模块出发，从实模式转换到保护模式，研究并论述 **Linux** 操作系统中分页机制与虚拟内存、进程与线程和文件系统等概念的涵义。然后设计并开发内核中的时钟管理、内存管理、任务管理、文件管理和终端等功能模块，完成一个较为完善的 32 位操作系统。并且本文采用 C++ 语言和汇编语言混合编程的方式来设计和开发操作系统内核的各个模块，为改善大型操作系统的封装性与维护性打下基础。

本文对内核的各功能模块进行了测试与分析。考虑到内核模块较多，开发与测试工作较为繁杂，故将程序文件放在不同的目录下管理和维护，并采用 **makefile** 工具简化编译过程。将编译成功的二进制可执行文件写到虚拟磁盘映像文件中，再用 **Bochs** 虚拟机直接运行虚拟磁盘，测试结果基本符合预期。最后对相关研究工作和现有成果进行了总结与展望，总结不足之处，为后续的优化工作确立了方向。

**关键词：**操作系统，**Linux**，内核，模块

## Abstract

An operating system is a computer program that manages and controls computer hardware and software resources, as well as an interface between computer hardware and other software. Although operating system technology is becoming more mature, it is still an area that is constantly evolving and constantly updated. However, the development of domestic operating systems is relatively backward, and it is currently difficult to compete with internationally renowned operating systems. Linux is a commonly used and powerful operating system in the world. It has the characteristics of freedom, openness and free of charge, providing people with the opportunity to learn the design concepts and implementation methods of excellent foreign operating systems. Therefore, research and improvement of the Linux operating system is of great significance to the development of the domestic operating system.

In addition, most operating system kernels on the market today are mixed programming in C and assembly languages, mainly based on C language. Large projects consisting of C language often face problems such as unclear relationship between functions and modules, reusability of code and poor maintainability. In the current trend of rapid expansion of Linux code, it is especially important to solve the problem of code maintenance. Therefore, the operating system kernel implemented by object-oriented language will show obvious advantages, and has important practical significance and application value.

This article will start from the boot module, from real mode to protected mode, research and discuss the meaning of the paging mechanism and virtual memory, process and thread and file system concepts in the Linux operating system. Then design and develop functional modules such as clock management, memory management, task management, file management and terminal in the kernel to complete a relatively complete 32-bit operating system. And this article uses C++ language and assembly language mixed programming to design and develop the various modules of the operating system kernel, to lay the foundation for improving the packaging and maintenance of large operating systems.

This paper tests and analyzes the functional modules of the kernel. Considering that there are many kernel modules, the development and testing work is so complicated. Therefore, the program files are managed and maintained in different directories, and the makefile tool is used to simplify the compilation process. Write the compiled binary executable file to the virtual disk image file, and

then run the virtual disk directly with the Bochs virtual machine. The test results are basically in line with expectations. Finally, the related research work and existing achievements are summarized and forecasted, and the inadequacies are analyzed, which establishes the direction for the subsequent optimization work.

**Key words: operating system, Linux, kernel, module**

# 目录

|                         |    |
|-------------------------|----|
| 第一章 绪论 .....            | 1  |
| 1.1 课题背景与研究意义 .....     | 1  |
| 1.2 国内外研究现状 .....       | 1  |
| 1.3 论文重点研究内容 .....      | 3  |
| 1.4 论文组织结构 .....        | 3  |
| 第二章 操作系统相关技术分析 .....    | 5  |
| 2.1 内核分类 .....          | 5  |
| 2.2 保护模式概述 .....        | 6  |
| 2.3 分页机制与虚拟内存 .....     | 7  |
| 2.3.1 分页机制 .....        | 7  |
| 2.3.2 虚拟内存 .....        | 7  |
| 2.4 进程与线程 .....         | 8  |
| 2.4.1 进程概述 .....        | 8  |
| 2.4.2 进程与线程的关系 .....    | 9  |
| 2.4.3 任务的调度 .....       | 9  |
| 2.5 文件系统 .....          | 10 |
| 2.5.1 文件系统简述 .....      | 10 |
| 2.5.2 文件系统分类 .....      | 10 |
| 2.6 本章小结 .....          | 11 |
| 第三章 内核设计与实现 .....       | 12 |
| 3.1 内核架构设计 .....        | 12 |
| 3.2 时钟与中断 .....         | 13 |
| 3.2.1 中断的处理 .....       | 13 |
| 3.2.2 特权级检验 .....       | 14 |
| 3.2.3 中断发生时的压栈 .....    | 15 |
| 3.2.4 时钟信号的产生 .....     | 16 |
| 3.3 内存管理 .....          | 16 |
| 3.3.1 内存池规划 .....       | 16 |
| 3.3.2 页内存分配 .....       | 18 |
| 3.3.3 堆内存管理 .....       | 19 |
| 3.4 任务的管理与调度 .....      | 20 |
| 3.4.1 程序控制块 .....       | 20 |
| 3.4.2 进程的创建 .....       | 21 |
| 3.4.3 任务的切换 .....       | 22 |
| 3.4.4 任务的阻塞与唤醒 .....    | 24 |
| 3.5 本章小结 .....          | 24 |
| 第四章 文件系统与终端 .....       | 25 |
| 4.1 文件系统分析 .....        | 25 |
| 4.1.1 INODE 与块索引表 ..... | 25 |
| 4.1.2 文件系统布局 .....      | 26 |
| 4.2 创建文件系统 .....        | 27 |
| 4.2.1 基础结构的实现 .....     | 27 |
| 4.2.2 文件系统的创建 .....     | 28 |
| 4.3 文件描述符与原理实现 .....    | 29 |

|                              |    |
|------------------------------|----|
| 4.4 文件操作的实现 .....            | 30 |
| 4.4.1 文件的创建与删除 .....         | 30 |
| 4.4.2 文件的打开与关闭 .....         | 30 |
| 4.4.3 文件的写入与读取 .....         | 31 |
| 4.4.4 目录的创建与删除 .....         | 32 |
| 4.5 终端的实现 .....              | 33 |
| 4.5.1FORK 克隆进程.....          | 33 |
| 4.5.2 系统调用 WAIT 和 EXIT ..... | 33 |
| 4.5.3SHELL 命令与环形缓冲区 .....    | 35 |
| 4.5.4 命令行的实现 .....           | 36 |
| 4.6 管道设计与实现 .....            | 38 |
| 4.6.1 管道设计 .....             | 38 |
| 4.6.2 在 SHELL 中支持管道.....     | 38 |
| 4.7 本章小结 .....               | 39 |
| 第五章 系统测试与结果分析 .....          | 40 |
| 5.1 工作与测试环境的搭建 .....         | 40 |
| 5.2 测试结果分析 .....             | 40 |
| 5.2.1 内存测试 .....             | 40 |
| 5.2.2 多任务测试 .....            | 41 |
| 5.2.3 文件处理命令测试 .....         | 43 |
| 5.3 本章小结 .....               | 44 |
| 第六章 总结与展望 .....              | 45 |
| 6.1 论文工作总结 .....             | 45 |
| 6.2 论文贡献与创新 .....            | 45 |
| 6.3 研究展望 .....               | 45 |
| 参考文献 .....                   | 47 |
| 附录 1 攻读硕士学位期间撰写的论文 .....     | 49 |
| 致谢 .....                     | 50 |



# 第一章 绪论

## 1.1 课题背景与研究意义

自从第一台计算机于 1946 年推出以来,在 70 多年的发展过程中,计算机和人类的生产和生活变得越来越不可分割。在计算机的发展过程中,运算速度不断提高,体积不断缩小,硬件工艺不断进步,同硬件需求相对应的操作系统也一直推陈出新<sup>[1]</sup>。

在计算机发展之初,没有操作系统,操作员可以通过各种按钮控制计算机。但随着功能的丰富和应用范围的扩展,有限的按钮逐渐不能满足操作的需要。于是,针对计算机控制的汇编语言应运而生,操作系统也开始兴起<sup>[2]</sup>。1964 年 4 月 7 日,IBM 推出了划时代的大型计算机 System/360,这是世界上第一台指令集可兼容计算机。它的出现使单一操作系统适用于各种计算机<sup>[3]</sup>。自此之后,经过 UNIX,MS-DOS,Windows,MacOS 一代代的更迭;经过批处理,多批处理,分时,实时系统一代代的发展,操作系统功能日益丰富,处理效率日益提高,安全性能也越来越好<sup>[4]</sup>。

近年来,虽然操作系统技术日益成熟,但仍然是一个不断发展,不断更新的领域。在当前众多操作系统百花齐放的时代,于 1991 年诞生的 Linux 脱颖而出,广受企业青睐<sup>[5]</sup>。特别是在高端服务器领域,随着开源软件在全球的影响越来越大,Linux 服务器在整个服务器市场中占据越来越大的市场份额,形成了大规模应用的局面<sup>[6]</sup>。凭借强大的发展势头,Linux 已成为服务器操作系统领域的支柱,引起了全球 IT 行业的关注,具有很大的研究价值。本文将深入研究 Linux 的早期内核架构,并对其加以改进,实现一个全新的 32 位操作系统。

随着 Linux 开源系统的快速推广,它的诸多商业版本早已如雨后春笋般蓬勃发展。但为了系统的高效运行,以及历史原因的束缚,几乎所有 Linux 的衍生版本都会以 C 语言为主、汇编语言为辅的方式实现内核。C 语言虽然具有执行高效等优势,但它是面向过程的语言,代码的复用性和维护性都比较差,结构与功能的关联不够清晰。

基于以上考虑,本文选择以 C++代替 C 语言实现内核,在保证系统运行效率的同时,让底层的操作系统技术更加面向未来。

## 1.2 国内外研究现状

人们在对计算机硬件系统不断增强性能的同时,也在相应地升级软件系统,其中最重要

的就是对操作系统的研究<sup>[7]</sup>。我们最熟悉的、最具代表性的操作系统便是 Windows 和 Linux，它们为操作系统的发展做出了巨大的贡献，拥有难以超越的性能优势和市场优势。

### （1）Windows

Windows 是一款窗口化操作系统，于 1985 年由美国微软推出。它采用 GUI 图形化操作模式，是目前世界上应用最为广泛的操作系统，占据了全球桌面操作系统市场中 90% 的份额<sup>[8]</sup>。与此同时，Windows 在中低端服务器市场也有广泛的应用，例如 Web 服务器和数据库服务器。近年来微软花费了大量的研发资金来提高 Windows 运行大型程序的能力，以扩展应用领域。作为一种现代操作系统，Windows 在技术和市场方面都是最成功的。2017 年的操作系统市场份额如图 1.1 所示。

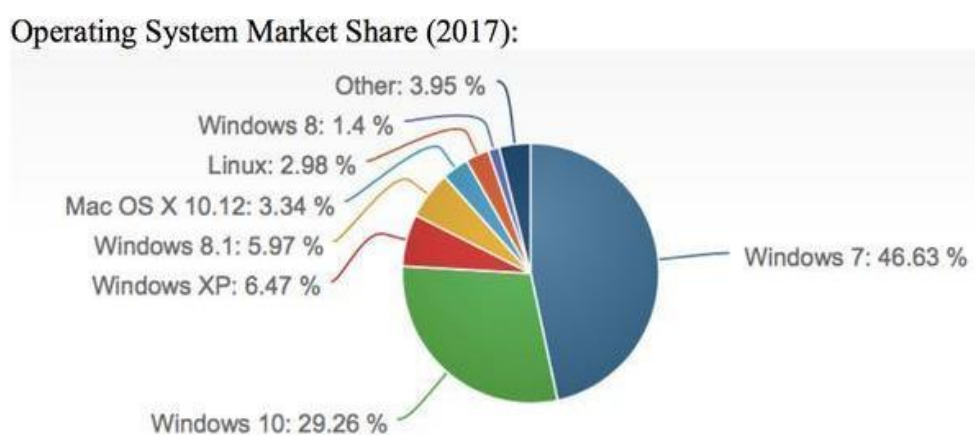


图 1.1 2017 年各操作系统的市场占有率

### （2）Linux

Linux 操作系统是 Unix 操作系统的一个衍生系统，于 1991 年 10 月 5 日首次发布。之后借助计算机网络和全世界计算机爱好者的共同努力，现已成为世界上使用最广泛的类 Unix 操作系统，且每年的用户数量仍在快速增长。这个系统由全球成千上万的程序员共同设计和实现，目的是创建与 Unix 兼容的产品，这些产品不受任何商业软件的版权限制，并且可以免费向公众开放<sup>[9]</sup>。

无论从硬件市场还是从软件性能角度来观察，Linux 都已经是一个非常成熟的操作系统。免费与开源的特性也使得 Linux 对 Windows 更具威胁性。在服务器和嵌入式系统市场上，Linux 已经是主流的操作系统之一。Linux 正在稳步拓展桌面操作系统市场，越来越多的厂商开始在它们销售的计算机上预装 Linux<sup>[10]</sup>。

由于起步时间晚，国内操作系统的研发水平较国外还有很大的差距，且大多数是基于 Linux 的二次开发。但近年来各企业与机构都在致力于国产操作系统的研究，并取得了明显的进步。麒麟操作系统的成功推出更是引起了广泛的关注。

### （1）红旗 Linux

Red Flag Linux 是一系列性能优良的国产 Linux 发行版，由北京中科红旗软件技术有限公司开发并推出<sup>[11]</sup>。这一系列包括诸多版本的 Linux 产品，如工作站版、数据中心服务器版、HA 集群版等等<sup>[12]</sup>。目前，在中国各种软件专卖店可以购买到红旗 Linux 的光盘，同时官方网站也可以免费下载光盘镜像。红旗 Linux 是一款较为成熟的发行版国产 Linux。

### （2）银河麒麟

Galaxy Kirin 是由国防科技大学、中软、联想、浪潮集团和民族恒星联合研发的闭源服务器操作系统，是 863 计划重大攻关科研项目。它的研制目标是打破国外操作系统的长期垄断，研制一款中国自主知识产权的服务器操作系统。它有以下几个特点：高安全性、高可靠性、高实用性、跨平台和汉语化（具有强大的中文处理能力）<sup>[13]</sup>。银河麒麟的推出给国内计算机科学界极大鼓舞，证明自主研发操作系统是可行的。

事实上，国家对国产操作系统的扶持，从上世纪末就已经开始。虽然国内操作系统已经取得长足的进步，然而对比目前国内外的发展现状，国内的整体形式仍然不容乐观。因此，为了提高国内操作系统的研发技术，缩小与国外的差距，仍需加大创新力度，加强对操作系统技术的研究。

## 1.3 论文重点研究内容

本文重点研究内容有如下几点：

（1）Linux 内核的研究与改进：借鉴 Linux 内核中的相关技术与技巧，建立一个类似 Linux 内核的操作系统内核，使其支持中断响应机制、分页机制、多任务调度管理和用户态与内核态的内存管理等技术。

（2）文件系统的搭建：在虚拟磁盘中建立一个较为完善的文件系统，实现文件和目录在 shell 中的创建、删除、查询和打印等基本操作。

（3）面向对象的设计：使用面向对象的思想来设计内核，封装内核中的各功能模块，使代码具有较好的封装性和一定的可复用性。

## 1.4 论文组织结构

本文内容分为六章进行论述，内容安排如下：

第一章：绪论部分。首先介绍了本文的研究背景和研究意义以及国内外的研究现状，并对全文的脉络进行了梳理，然后总结了各章的主要内容。

第二章：介绍操作系统内核的相关理论和技术基础。对 Linux 的内核设计做详细阐述，分析内核中各个模块的相关概念和历史发展。

第三章：详细介绍本文操作系统的内核架构设计，研究页表的建立、虚拟内存原理、进程和线程的管理等技术。在功能设计过程中采用子模块设计的方法设计各个模块的功能。不仅完成基本的操作需求，也为后续的二次开发工作提供便利。

第四章：对比 NFS、FAT32 等已有文件系统的优劣，为本文的操作系统建立较为完善的文件系统，满足基本的文件操作。

第五章：内核的测试与分析，对操作系统的功能与性能进行全面评估。

第六章：全面总结论文的研究工作，阐述论文的创新点，分析系统的不足之处，并对后续的研究工作进行探索。

## 第二章 操作系统相关技术分析

### 2.1 内核分类

内核是一个操作系统的核心，是计算机中基于硬件的第一层扩展，提供操作系统最基本的功能。不同的内核架构设计决定着系统的性能和稳定性的优劣。按照系统架构的不同，内核分为单内核，微内核及混合内核等<sup>[14]</sup>。

单内核架构，也称为整体式内核体系结构，是操作系统发展早期最广泛使用的架构组织形式。后来随着模块不断扩充，其结构也变得愈加复杂。最初的 Linux 内核便采用了单内核模式。随着 Linux 不断发展，内核逐渐具有诸多功能模块，如进程调度、内存管理、文件系统、进程间通信和网络接口等。在单内核模式的系统中，操作系统所提供的服务流程如下：用户程序按照指定的信息参数值执行对应的系统调用，使处理器从用户态切换到内核态，在内核态完成任务，再切回到用户态中继续执行<sup>[15]</sup>。单内核模式的主要优缺点：1) 内核代码紧凑，执行速度快；2) 内核和硬件之间直接通信；3) 针对不同的硬件体系结构，内核性能不同，造成内核的移植性不是很高；4) 当内核增加某个功能模块或函数时，必须重新编译整个内核。

微内核是一种扩展性较强的内核体系结构。单内核结构很难对其它模块进行有效的管理，而微内核的出现恰好弥补了这个缺点<sup>[16]</sup>。它的内核较小，实现了较低的平台依赖性。微内核结构将系统服务的实现和系统的基本操作规则分离开来，受到很多操作系统设计者的青睐。微内核体系结构的主要特点有：1) 内核很小；2) 有一个硬件抽象层，内核能移植到其它的硬件体系结构中去；3) 兼容性强，不同的 API、文件系统，甚至不同的操作系统的特性能够在同一个系统中共存。

混合内核很像微内核结构，只是它的组件在内核态中运行得更多，以获得更快的执行速度。这种体系结构设计避免出现微内核结构系统运行速度不佳的情况发生，现在大多数操作系统都遵循这种设计方式。采用这种体系结构的操作系统有 BeOS 内核，Dragon Fly BSD，React OS 内核等<sup>[17]</sup>。混合内核从单内核和微内核系统中都吸取了一定的设计理念，所以它兼有两种模式的主要优点：1) 移植简单，只需要替换一部分模块而无需额外编写任何代码；2) 当系统增加、修改和删除某些功能时，不必重新编译整个系统；3) 稳定性好，各个层次间的实现比较独立，当某一部分出现问题，系统一般不会崩溃。

在以上三种内核结构中，单内核结构体系虽然不乏弊端，但也拥有最好的执行效率和系

统稳定性，且避免了过于复杂的架构设计。基于以上原因，本文的 32 位操作系统采用了单内核结构。

2.2 保护模式概述

操作系统的保护模式，是相对于实模式提出的，实模式的 CPU 运行环境为 16 位，保护模式的运行环境是 32 位<sup>[18]</sup>。如前文所述，本文研究目标是实现一个 32 位操作系统，故保护模式的研究是一项重要工作。

16 位操作系统之下，CPU 只能运行于实模式。此时，地址总线有 20 位，最大可用内存为 1MB，这样的内存即使在 20 年前也显得捉襟见肘。当 CPU 发展到 32 位后，拥有 32 位地址总线 and 数据总线，寻址空间便达到了 4GB，系统性能相比实模式有着巨大改善。但为了保证 CPU 的向下兼容，32 位系统下的 CPU 也须先进入实模式，然后通过切换机制进入保护模式。由实模式进入保护模式只需三个步骤：

- (1) 打开 A20 总线；
- (2) 加载 gdt（全局描述符表）；
- (3) 将 cr0 寄存器的 pe 位值 1。

这三个步骤不需要顺序执行或连续执行，便可以实现工作模式的切换。

保护模式仍旧采用“段基址：段内偏移”的形式来描述地址，但寻址方式与实模式截然不同。在实模式下，地址值等于 16 位的段基址左移 4 位，再加上段内偏移。而保护模式下的地址转换机制则相当复杂。

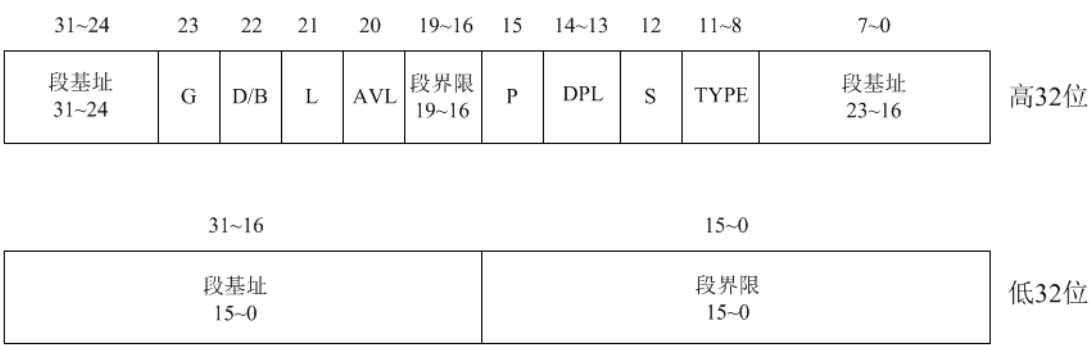


图 2.1 段描述符结构

与实模式不同，保护模式下的段基址并不存储在段寄存器中，而是存于数据结构全局描述符表中。全局描述符表的每一个表项称为段描述符，其大小为 64 字节，用来记录各个内存段的起始地址、大小、权限等信息<sup>[19]</sup>，具体格式如图 2.1 所示。该描述符表占用空间比较大，所以存在内存之中以减轻内核的负载，由 GDTR 寄存器指向它。而段寄存器中存储的数据结

构称为选择子（Selector），用于索引全局描述符表中的段描述符。

由此可见，保护模式下的地址计算虽然复杂，但增大了寻址空间，并在段描述符中加入了内存段的权限信息，建立了内存段的安全访问机制。

## 2.3 分页机制与虚拟内存

### 2.3.1 分页机制

分页存储是区别于分段存储的一种内存控制机制，使得内存的分配与释放更加灵活。分页机制的思想是：通过映射，可以使连续的线性地址与任意物理内存地址相关联，逻辑上连续的线性地址对应的物理地址可以不连续<sup>[20]</sup>。

在分页机制中，系统所有内存被划为若干个容量相等的物理页，每个物理页内存均为 4KB。每个物理页的地址存储在页表（Page Table）中<sup>[21]</sup>。页表其实就是一个 N 行 1 列的表格，其中每一行存储的信息称为页表项（Page Table Entry, PTE），其大小是 4 字节，页表的作用是存储内存物理地址，是若干个地址的索引。当访问一个线性地址时，实质上便是在访问页表项中所记录的物理内存地址。

由于 4GB 的寻址空间需要 1M 个物理页，每个进程的页表将会占用很大的空间，所以采用二级页表来灵活管理物理页的开辟与回收。二级页表是将这 1M 个标准页平均放置于 1K 个页表中，每个页表中含有 1K 个页表项。再将这 1K 个页表的物理地址放在一个页目录表中，每个页表的物理地址在页目录表中都以页目录项（Page Directory Entry, PDE）的形式存储，这个页目录表便是二级页表。所以页表与页目录表都是 4KB 大小，与标准页的大小相一致。

### 2.3.2 虚拟内存

CPU 在不打开分页机制的情况下，是按照默认的分段方式运行的，段基址和段内偏移地址经过 CPU 的段部件处理后输出的线性地址即是物理地址<sup>[22]</sup>。如果打开了分页机制，段部件输出的线性地址便是虚拟地址，而不等于实际地址。此虚拟地址必须在两级页表中查找才能得到物理地址，这项查找工作是由 CPU 的页部件自动完成的。

由于每个进程都维护了一个私有的页目录表，所以即使实际物理内存比较小，每个进程也都有 4GB 的虚拟寻址空间。

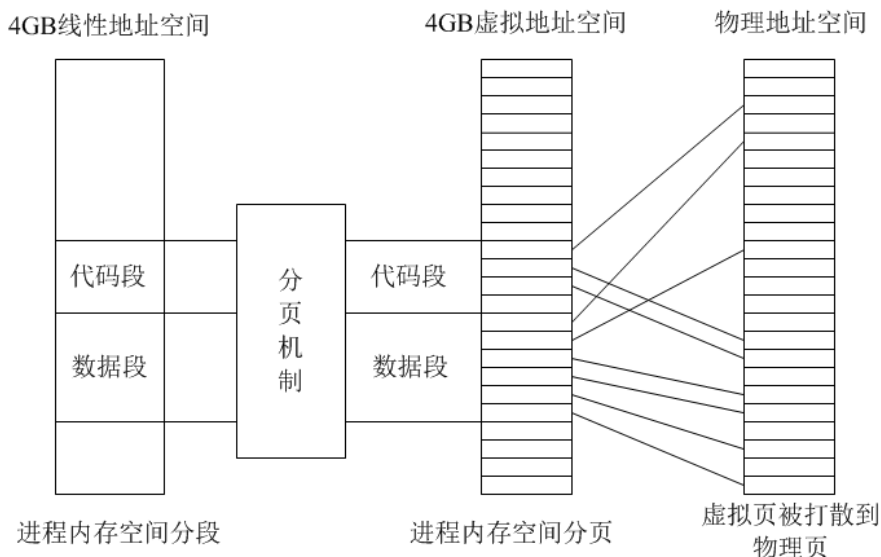


图 2.2 分页机制的作用

如图 2.2 所示，进程的地址转换过程中，线性空间通过虚拟空间映射到物理地址空间，线性空间和虚拟空间大小都是 4GB。虽然分段的内存中大多是连续内存，但分配到的物理内存可能是不连续的物理页。

分页机制下，代码段和数据段在逻辑上被拆分成以页为单位的小内存块，它们便是虚拟页。操作系统为这些虚拟页分配真实的物理页内存，将可用的物理页地址关联到页表项，便完成了虚拟页到物理页的映射<sup>[23]</sup>。

分页和虚拟内存技术使内存的管理更为高效和灵活，分页存储有效减少了内存碎片造成的浪费，虚拟内存让操作系统在有限的实际内存之上同时执行多个较大的任务。

## 2.4 进程与线程

### 2.4.1 进程概述

随着多任务操作系统的出现，进程的概念出现在了计算机领域。进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元<sup>[24]</sup>。

进程执行时的间断性，决定了进程可能具有多种状态，以下列出三种基本状态。

(1) 就绪状态（Ready）：进程已经具备所有执行条件，等待处理器的执行，只要受到了调度器的调度就可以立即运行<sup>[25]</sup>。进程进入就绪态有多种情况，例如刚刚创建的进程直接进入就绪态、运行的进程时间片用完而进入就绪态以及由于某些条件不满足而阻塞的进程满足了



相应条件而进入就绪态等等。

(2) 运行状态 (**Running**)：进程被处理器执行，正在占用处理器资源，单内核体系中同一时刻最多只有一个进程处于此状态<sup>[26]</sup>。只有就绪态的进程才能转入运行态，此时必须得到调度器的调度。运行态的进程可能转为就绪态或阻塞态。

(3) 阻塞状态 (**Blocked**)：程序从开辟到结束的过程中并不是一直都可以执行的，比如读写日志的程序需要访问磁盘，这种需要经过外部设备的操作便必须等待处理器之外的其他系统资源，一旦获取到资源立即转入就绪态。这种需要等待外界条件的状态便称为阻塞态。

### 2.4.2 进程与线程的关系

线程是一个相对较新的名词，是表示进程中一个指令组的执行过程的抽象概念，是操作系统任务调配的基本单位<sup>[27]</sup>。进程包含线程，且多个线程可以并存于同一个进程中，它们各自执行、协同完成整个程序。对于处理器来说，进程是一种控制流集合，集合中至少包含一条执行流，这些执行流便是线程。

线程的引入是为了提升程序的执行速度，将单一的进程合理地分成多个分支并各自运行。当进程需要等待某些系统资源（如 IO 设备）时，传统的进程便会阻塞自己，直到资源空闲时被唤醒<sup>[28]</sup>。若将进程分为多个线程分别执行，一项资源的缺乏只会阻塞其中一个线程，避免了整个进程停滞不前。

对于 CPU 的处理而言，进程与线程有着很多共性。但是从操作系统角度来说，进程与线程是截然不同的实体。进程拥有整个地址空间，从而拥有程序运行所需的全部资源。线程不具备私有空间，需要借助进程的资源来执行<sup>[29]</sup>。只有线程才具有能动性，是处理器的执行单元，而进程只是一个资源容器，容纳了所有线程共享的资源。

### 2.4.3 任务的调度

多任务操作系统必须要解决任务的调度问题，协调多个任务的并发执行<sup>[30]</sup>。操作系统的任务调度本质上是对所有进程中的线程进行调度，使已经具备执行条件的线程获得处理器资源并开始运行。执行进程调度工作的部分称为任务调度器，它的基本功能有如下几点：

(1) 记录系统中所有任务的执行情况。调度器在调度任务时会将它们的信息（如运行状态和上下文等）记录到线程各自的 PCB 中，以准备下一次执行该线程。

(2) 为就绪任务分配 CPU。调度器的主要任务便是依照特定的规则为 CPU 选择一个线程来执行<sup>[31]</sup>。有众多不同的调度规则可以选择，如先来先服务，短作业优先等。这些策略的选择

决定了系统的性能<sup>[32]</sup>。

(3) 进行进程上下文切换。一个进程的上下文(context)包括它的指令寄存器、堆栈指针、通用寄存器中的内容以及进程打开的文件描述符等。一个任务的执行必须依赖它的上下文,当发生任务切换时,调度器必须先切换上下文,再转移 CPU 的控制权<sup>[33]</sup>。

## 2.5 文件系统

### 2.5.1 文件系统简述

文件系统是一个操作系统必须具备的功能模块,是为用户提供的大规模数据处理的接口。在存储信息数据时,通常都会以文件的形式进行保存,以供特定用户使用。在 Linux 系统中,计算机中所有事物都可以看成一个文件,程序完全可以像对待文件那样对待串口,打印机等设备<sup>[34]</sup>。所以,操作系统需要拥有文件管理的功能。

文件系统,简单来说就是磁盘设备上的管理文件的策略和存储文件的数据结构。所有的操作系统都拥有与其自身相适应的文件系统,且操作系统通常需要对文件系统进行管理<sup>[35]</sup>。文件系统通常需要具备一些基本功能,如对文件存储空间进行管理规划、文件检索、文件的读/写管理、文件的共享及文件保护等。

文件存储空间管理必须高效且合理。文件系统是建立在硬盘上的,硬盘是低速设备,其读写单位是扇区(Sector),大小为 512B。为了避免频繁访问硬盘,操作系统以块(block)为单位读写磁盘,块的大小是扇区的整数倍<sup>[36]</sup>。

对于文件检索,每个文件都有自己的父目录,一个文件系统中最高的目录称为根目录。每个目录中可能含有多个目录项,目录项可能是文件,也可能是另一个目录。用户可以通过文件名来找到相应的文件,实现按名检索。文件读写也使用目录结构,来获取文件在磁盘的存储位置,实现读写操作。

文件共享与保护,即权限管理。通常会在文件的存储结构中增加一个权限位,而每个用户都有自己的权限,这样便可以防止用户存取不属于它的文件,达到文件共享与保护的目的。

### 2.5.2 文件系统分类

Windows 操作系统中常见的文件系统有 FAT32、NFS 等;Linux 中有几十种文件系统,最具代表性的是 Ext2/3/4。这些不同的文件系统分别代表了存储设备上组织文件的不同方法。

FAT(File Allocation Table)是微软在 DOS/Windows 系列操作系统中共同使用的一种文

件系统的总称，FAT12、FAT16、FAT32 均是 FAT 文件系统。由于早期的计算机性能限制，FAT 采用了较为简单的设计，对普通 PC 都非常友好<sup>[37]</sup>。但是当文件有增删时，散落的空间不能得到整合，长此以往，磁盘便越来越碎片化，不但降低了磁盘利用率，也严重影响了读写效率。虽然可以通过碎片整理来应对这种不足，但这种途径必须经常重组磁盘来保持 FAT 文件系统的整合性。因此，FAT 文件系统面临磁盘空间利用率低、文件管理效率低、文件存储受限以及安全性较差等问题。

NTFS (New Technology File System) 是由微软开发的专有文件系统。从 Windows NT 3.1 开始，它是 Windows NT 系列的默认文件系统。针对 FAT 等传统文件系统的诸多弊端，NTFS 进行了全面的改善<sup>[38]</sup>。例如改进了对元数据的支持，以便于提高访问效率、系统的安全性和设备存储的利用率。

Ext (Extended file system) 是扩展文件系统，于 1992 年推出，是首个为 Linux 内核所制作的文件系统。为了避免 Minix 文件系统的性能缺陷，Ext 使用了 Unix 文件系统 (UFS) 的相关技术。之后出现了 Ext2, Ext3, Ext4 等更新版本，逐渐完善了功能<sup>[39]</sup>。

文件系统是计算机中一个不可或缺模块，也是实现的难点。本文利用 Linux 文件系统的架构设计出一个更为轻便实用的文件系统，并通过研究与改进文件操作的底层原理，逐步找出较为简便的文件管理策略。

## 2.6 本章小结

本章是操作系统的相关理论与技术研究，首先介绍了常见操作系统内核的分类，选取了本文采用的内核类型。然后介绍了保护模式的概念，简要对比了保护模式与实模式下寻址的异同，以及保护模式的优势。之后分析了分页机制及其作用，说明了虚拟内存的含义。再介绍了操作系统中进程与线程的概念，对二者进行了详细比较，并研究了任务的调度工作。最后研究文件系统具备的基本功能，分析了几种常见文件系统的发展和各自的优劣

### 第三章 内核设计与实现

#### 3.1 内核架构设计

本文的设计是基于 Linux 架构的，内核空间的功能模块包括时钟管理、内存管理、任务管理、文件系统管理与终端控制。在内核空间的底层是硬件驱动，它们是操作系统与硬件之间通信的桥梁<sup>[40]</sup>。

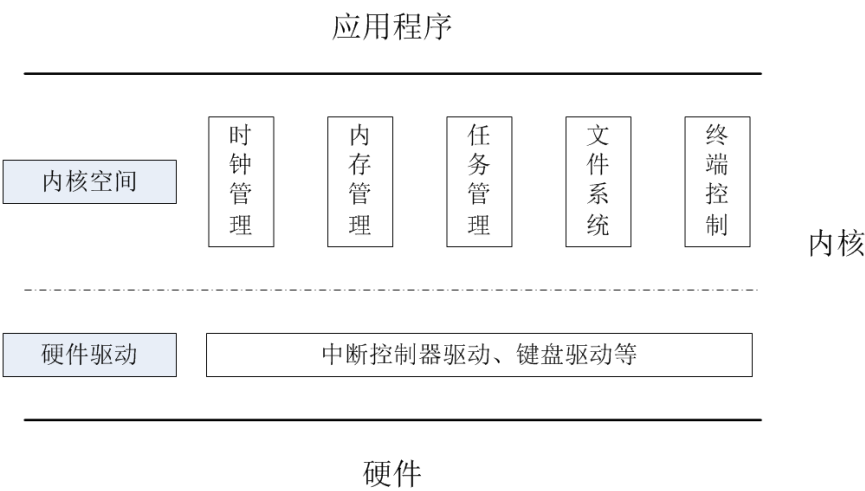


图 3.1 系统架构

如图 3.1 所示，内核模型分为两层：硬件设备驱动层和内核空间层。硬件驱动层是键盘、终端控制器等硬件与内核之间数据传输的规则，系统通过它们来管理硬件设备。内核空间层是各个核心功能模块，其中时钟管理模块和内存管理模块本质上是服务于任务管理和文件系统的，终端控制模块是用户管理计算机的工具。

#### 3.2 时钟与中断

##### 3.2.1 中断的处理

按照发起原因和处理方式的不同，中断可以分成硬件中断和软中断这两类<sup>[41]</sup>。硬件中断也称为外部中断，是由键盘、鼠标等硬件设备产生的中断；软中断就是在程序用汇编指令发起的中断，如指令 `int n` 的意义便是引发号码为 `n` 的中断，用于实现系统调用功能。

在处理器由实模式转换为保护模式时，中断处理机制也会产生变化。在实模式下使用中断向量表来连接需要进行的操作，而在保护模式不再使用中断向量表。保护模式下用中断描

述符表 (Interrupt Descript Table, IDT) 来记录每个中断要执行的程序段信息。在保护模式下，每一种中断都有一个描述符存储于中断描述符表中，这些中断描述符将中断向量和全局描述符表中的段描述符连接起来。

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，然后去执行该中断描述符中的中断处理程序。由于中断描述符是 8 个字节，所以处理器用中断向量号乘以 8 后，再与 IDTR 中的中断描述符表地址相加，所得的地址之和便是该中断向量号对应的中断描述符的存储位置<sup>[42]</sup>。

由于中断是通过中断向量号通知到处理器的，中断向量号只是个整数，其中并没有特权级的检查。但如果是由软件主动发起的软中断，便必须通过检查特权级后再执行目标代码段。保护模式下的中断处理流程如图 3.2 所示。

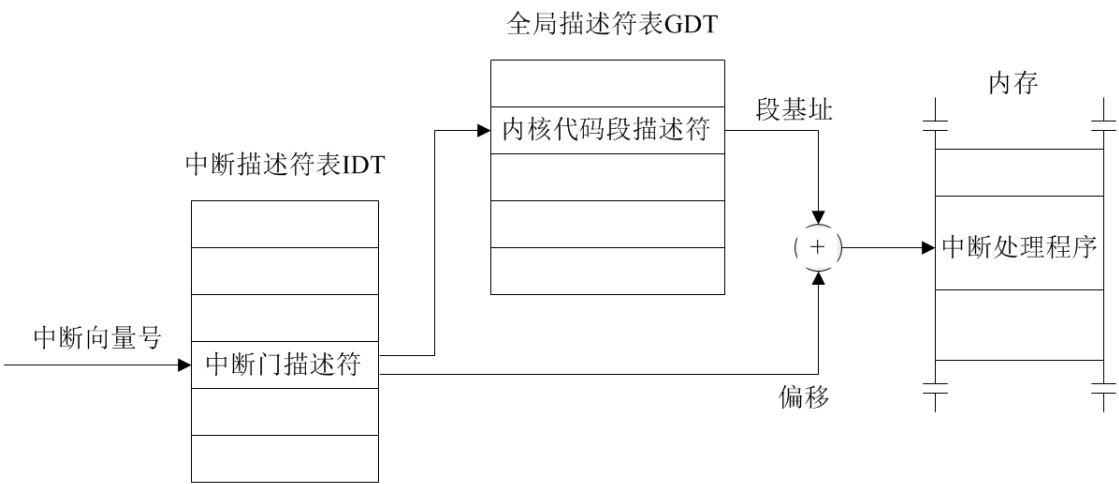


图 3.2 保护模式下的中断处理流程

3.2.2 特权级检验

中断的发生往往伴随着程序控制转移，保护模式中引入了特权级机制，使得每个程序的权力有限，以此来保护数据和阻止程序的恶意行为。在 IA32 架构中，特权级按权限从大到小分为 0、1、2、3 级，数字越小，权限越大<sup>[43]</sup>。在本文的操作系统中，特权级 0 表示内核态，特权级 3 表示用户态，特权级 1 和 2 保留暂不使用。0 级特权是操作系统内核所在的特权级，3 级特权是用户程序运行的特权级。一个任务按照特权级来划分的话，实质上是被分成了 3 特权级的用户程序和 0 特权级的内核程序，这两部分合在一起才是完整的程序。所以完整的任务必须要经历这两种特权级的变换。

这样将不同位置的程序分为不同的等级，有效保证了计算机内核心程序的安全性，防止

外部程序对操作系统的随意更改和恶意侵犯。在保护模式下，主要使用 RPL、CPL、DPL 这三种级别标记程序或数据的特权级，以供特权级检查时进行比较。

RPL (Requested Privilege Level) 是请求特权级，存储在代码段选择子的低 2 位。在保护模式中，每个代码段都有自己的 RPL。它表示了代码的请求、访问其它资源的能力。代码段的 RPL 并不是一直不变的，它会随着程序执行的层级变化而变化<sup>[44]</sup>。

CPL (Current Privilege Level) 是当前特权级，表示处理器正在执行的代码的特权级别，即当前 CPU 所处的特权级。位于 CS 寄存器中选择子低 2 位，称为请求特权级。因此，处理器的当前特权级实质上就是正在执行的代码段的 RPL。

DPL (Descriptor Privilege Level) 是描述符特权级，表示受访者的特权标签，存储在段描述符的第 13 位和第 14 中。DPL 是段描述符所对应的内存区域的访问门槛，访问者自身的特权级必须高于某段的 DPL 才能通过处理器的检查而访问该段。所以当程序访问内存中的段时会将 RPL 和 CPL 与此段的 DPL 相比较<sup>[45]</sup>。

不同段的 DPL 含义也不相同。数据段的 DPL 表示访问该数据段的程序必须达到的特权等级；而代码段的 DPL 表示访问该段的进程必须所处的特权级。假如当前特权级为 2，则可以访问 DPL 为 2、3 的数据段，但只能转移到特权级为 2 的代码段上运行。综上所述，对于受访者是代码段的情况，只能是平级访问。

在每一次程序控制转移或访问申请中，处理器都会对上述三个数值进行比较。当程序请求某数据段时，CPL 和 RPL 要同时大于等于受访者的 DPL，即在数值上  $CPL \leq DPL$  并且  $RPL \leq DPL$ 。当访问代码段时，必须满足  $RPL = DPL = CPL$ 。如果不满足特权级允许条件，处理器便会拒绝加载选择子，并产生异常。

### 3.2.3 中断发生时的压栈

当中断发生时，CPU 根据中断向量号在中断向量表 IDT 中找到相应的段描述符，从段描述符中将目标代码段的相关信息加载到相应寄存器中。但是寄存器的原始值都会丢失，当中断结束处理器要回到原来的代码段时，便不能回到之前的执行进度<sup>[46]</sup>。为了结束中断后能继续运行该程序，处理器在进入中断之前会先将相关寄存器中的值压入栈中，待从中断返回时再将栈中数据弹入寄存器即可。栈的变化顺序如下：

(1) 处理器找到中断描述符后，将 CPL 和中断门描述符中选择子对应的目标代码段的 DPL 对比，如果  $CPL > DPL$ ，这表示程序要从低特权级向高特权级转移，需要切换到高特权级的栈。此时，处理器先保存当前旧栈 SS 和 ESP 的值（记作 `ss_old` 和 `esp_old`），再将新栈加载到寄

存器 SS 和 ESP 中（记作 ss\_new 和 esp\_new）。由于 ss\_old 是 16 位数，32 位模式下的栈操作数是 32 位，所以将 ss\_old 用 0 扩展其高 16 位，变为 32 位数后入栈。

（2）在新栈中压入 EFLAGS 寄存器。

（3）将 CS 和 EIP 保存到当前栈中备份（记作 cs\_old 和 eip\_old），以便中断程序执行后能回到被中断的进程中。同样，cs\_old 也用 0 扩充其高 16 位再压入栈中。

（4）某些异常会产生错误码，用于指明异常发生在哪个段上。错误码会紧跟在 EIP 之后入栈，记作 error\_code。

此时新栈的内容如图 3.3 所示。

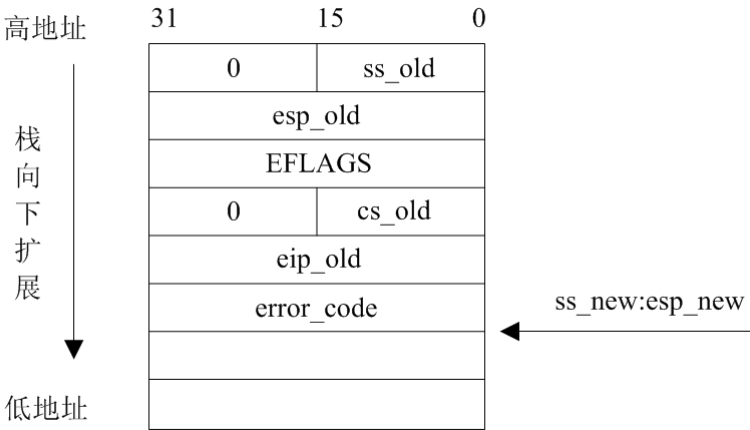


图 3.3 新栈中的内容

3.2.4 时钟信号的产生

操作系统的运行需要一个特定的周期性中断，这个中断可以协调系统各模块间的工作，给系统提供一个可参照的时间标准，此中断称为时钟中断，也称为时钟信号。

时钟信号是处理器工作的节拍，它的产生通常依赖于可编程定时器。常用的可编程定时器有 Intel 8253、8254 及 82C54A，本文只研究 8253 的时钟管理。8253 采用倒计时的方式，因此对它的编程就是围绕如何为其计数器赋初始的计数值。

8253 的内部有 3 个独立的计数器，分别是计数器 0、1、2，对应的端口号分别为 0x41，0x42，0x43，大小都是 16 位。在个人计算机中，计数器 0 专用于产生实时时钟信号。8253 内部还有一个控制字寄存器，操作端口为 0x43，大小为 8 位。控制字用来设置所指定的寄存器的工作方式、读写格式及数制。

综上所述，只需先设定好控制字寄存器的值，再将计数器 0 设定成适当值，便可以让 8253

产生特定频率的时钟信号<sup>[47]</sup>。8253 默认的时钟频率是 18.206Hz，即每秒大约发出 18 次中断信号。本节将其设置为 100Hz，让系统的时钟中断更为频繁。根据已有公式：

$$1193180/\text{中断信号的频率}=\text{计数器 0 的初始计数值}$$

将计数器 0 的初始计数值设为 11932 即可。

## 3.3 内存管理

### 3.3.1 内存池规划

程序是指令的集合，而内存是程序的舞台。处理器的设计是从内存中取指令，这导致了程序必须要加载到内存中才能运行。用户程序所占用的内存是由操作系统分配的，内存管理是操作系统的核心模块。

用户进程和内核程序分别运行在自己的地址空间，称为用户空间和内核空间。在实模式下，程序中的地址等于物理地址；而在保护模式下，程序地址是虚拟地址，虚拟地址与物理地址之间的映射需要由分页机制建立。因此，在分页机制下操作系统必须分别管理虚拟地址空间和物理地址空间，并通过页表将二者相关联。本节将规划这两类内存池。

为了操作系统的正常运行，必须为内核预留足够的物理空间，防止用户程序占用内存过多导致内核无法运行。所以将物理内存分成两个相同大小的内存池，一部分作为内核物理内存池，供操作系统使用；另一部分作为用户物理内存池，分配给用户进程。当用户内存池中的内存被用户进程耗尽时，也不会向内核内存申请，而是返回信息“内存不足”，拒绝请求。

在保护模式下，内核和所有用户进程都拥有各自的 4GB 虚拟地址空间，因此需要为所有任务（包括用户进程、内核线程）都维护一个自己的虚拟地址池。当用户进程向操作系统申请内存时，操作系统先从用户进程自己的虚拟地址池中分配空闲虚拟地址，再从用户物理内存池中分配物理内存，然后在用户进程自己的页表中将这两块内存建立好映射关系。内核程序申请内存时，有权限直接获取物理内存并使用，但为了规范管理，采取和用户进程申请内存相同的方法。

内存池基本结构如代码 3.1 所示。

代码 3.1

```
class CVirtual_addr
{
public:
    void* palloc(uint32_t n = 1);
    void release(uint32_t m, uint32_t n);
```



```

    void init(uint32_t s);
private:
    uint32_t m_vaddr_start;
    CBitmap m_vaddr_bitmap;
};
class CPool
{
public:
    void* palloc(uint32_t n = 1);
    void release(uint32_t m, uint32_t n);
    void init(uint32_t s, uint32_t z);
private:
    CLock m_lock;
    CBitmap m_pool_bitmap;
    uint32_t m_phy_addr_start;
    uint32_t m_pool_size;
};
CPool kernel_pool, user_pool;

```

从代码中可以看到，虚拟内存和物理内存的管理都借助了位图结构。与 CPool 类相比，CVirtual\_addr 类中没有 m\_pool\_size 成员，这是因为虚拟地址空间最大是 4GB，相对 32 位操作系统来说是无限的，不需要指定地址空间大小，而物理地址则是有限的。CPool 类的一个数据成员是一个互斥锁，避免多个用户同时访问内存。成员函数 palloc 的作用是在该物理内存池中分配 n 个物理页，成功则返回页框的物理地址，失败则返回 NULL，默认分配 1 个物理页；release 的作用是从第 m 页开始连续释放 n 个页。

### 3.3.2 页内存分配

内存池中的内存资源是以页为单位分配的，所以，从内存池中直接获取的内存大小只能是 4KB 的倍数。单个页内存的分配分为三个步骤：

- (1) 在虚拟内存池中申请一页虚拟内存。
- (2) 在物理内存池中申请一页物理内存。
- (3) 将以上两步得到的虚拟地址和物理地址在页表中建立映射。

分配页内存的核心函数是 malloc\_page()，其实现如代码 3.2 所示。

代码 3.2

```

void* malloc_page(enum pool_flags pf, uint32_t pg_cnt)
{
    ASSERT(pg_cnt > 0 && pg_cnt < 3840);
    void* vaddr_start = vaddr_get(pf, pg_cnt);

```

```
if (vaddr_start == NULL)
{
    return NULL;
}
uint32_t vaddr = (uint32_t)vaddr_start, cnt = pg_cnt;
CPool mem_pool = pf & PF_KERNEL ? &kernel_pool : &user_pool;
while (cnt-- > 0)
{
    void* page_phyaddr = mem_pool.palloc();
    if (page_phyaddr == NULL)
    {
        return NULL;
    }
    page_table_add((void*)vaddr, page_phyaddr);
    vaddr += PG_SIZE;
}
return vaddr_start;
}
```

枚举参数 `pf` 表示要申请内核内存还是用户内存，参数 `pg_cnt` 表示要申请的页数。此函数与上述的三个步骤略有不同，函数中先申请了一块连续的大小为 `pg_cnt` 页的虚拟内存，然后逐页申请物理内存并关联，直到关联了 `pg_cnt` 个物理页。这是因为物理内存与虚拟内存不同，一次申请的物理内存不一定是连续的物理页，散乱物理页和连续的虚拟页单独映射<sup>[48]</sup>。

### 3.3.3 堆内存管理

除了操作系统为任务分配的内存之外，程序在运行过程中也有申请内存的需求，这种动态申请一般是在堆中申请内存。操作系统接受申请后，为进程或内核在堆中分配空闲的虚拟空间。进程在创建时已获得系统按页分配的堆内存，所以堆内存的管理无需再向操作系统申请内存页，只需要将这些页内存划分成更小粒度的内存块，分配给程序使用。

堆是一个很大的内存块，在分配给程序使用之前，需要从中划出小块内存，它们被称为 `arena`，每个 `arena` 之间互不干涉，分别管理若干相同大小的内存块。`arena` 是一个内存分配的数据结构，它分为两部分，一部分是元信息，用来描述自己内存池中空闲的内存数量，还有内存块描述符指针，通过它可以间接获知此 `arena` 所包含内存块的规格大小，此部分是固定的，约为 12 字节。另一部分是内存池区域，其中大量规格相同的内存块，供程序申请。`arena` 结构如图 3.4 所示。

内存块规格为64字节的arena

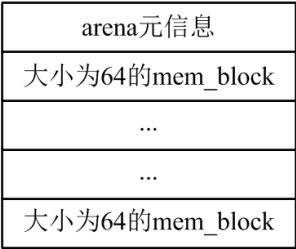


图 3.4 arena 结构简图

在内存管理系统中，arena 为任意大小的内存分配提供了统一的接口。它既支持 1024 字节以下的小块内存的分配，也支持 1024 字节以上的大块内存分配，本系统中 operator\_new 所封装的 sys\_malloc 函数便是通过 arena 来申请这些内存块。但 arena 并不直接对外提供内存分配，只有内存块描述符才对外提供分配入口。因此内存块描述符与 arena 是一对多的关系，每个 arena 都要与唯一的内存块描述符关联起来，一个内存块描述符管理多个同一规格的 arena。基础结构如代码 3.3 所示。

代码 3.3

```
struct CMem_block
{
    CArena* block2arena();
    CList_elem m_free_elem;
};
class CMem_block_desc
{
public:
    void block_desc_init();
private:
    uint32_t m_block_size;
    uint32_t m_blocks_per_arena;
    CList_elem m_free_elem;
};
struct CArena
{
    CMem_block* arena2block(uint32_t idx);
    CMem_block_desc* desc;
    uint32_t cnt;
    bool large;
};
```

本系统通过 arena 的管理实现了 sys\_malloc 和 sys\_free 内核函数。但为了让用户进程能申请内存，必须为用户提供系统调用接口，所以添加了 operator\_new 和 operator\_delete 两个系

统调用如代码 3.4 所示。

代码 3.4

```
void* operator_new(uint32_t size)
{
    return (void*)_syscall_1(SYS_MALLOC, size);
}
void operator_delete(void* ptr)
{
    _syscall_1(SYS_FREE, ptr);
}
```

## 3.4 任务的管理与调度

### 3.4.1 程序控制块

现代操作系统都是多任务操作系统，每个任务被调度到处理器上分时运行，运行一段时间之后再被换下来，由调度算法再选择下一个任务<sup>[49]</sup>。为了控制这些进程的状态，操作系统为每个进程提供了一个 PCB（程序控制块），并用它来记录进程相关的信息。PCB 没有固定的结构，其实际格式取决于操作系统的功能复杂度。本文的 PCB 结构如代码 3.5 所示。

代码 3.5

```
struct task_struct
{
    uint32_t* self_kstack;
    pid_t pid;
    enum task_status status;
    char name[TASK_NAME_LEN];
    uint8_t priority;
    uint8_t ticks;
    uint32_t elapsed_ticks;
    struct list_elem general_tag;
    struct list_elem all_list_tag;
    uint32_t* pgdir;
    struct virtual_addr userprog_vaddr;
    struct mem_block_desc u_block_desc[DESC_CNT];
    int32_t fd_table[MAX_FILES_OPEN_PER_PROC];
    uint32_t cwd_inode_nr;
    pid_t parent_pid;
    int8_t exit_status;
    uint32_t stack_magic;

    void thread_create(thread_func function, void* func_arg);
}
```

```
void init_thread(char* name, int32_t prio);  
void thread_unblock();  
void thread_exit(bool need_schedule);  
};
```

内核线程和用户进程都使用这个 PCB 类。在 CTask\_struct 类中，self\_kstack 指向进程自己的内核栈；status 记录了进程当前的状态；priority 是进程的优先级；ticks 是进程所拥有的时间片；pgdir 是进程页表的地址。

### 3.4.2 进程的创建

进程的创建由函数 process\_execute 来完成，由于本文中进程的实现基于内核线程，故进程创建过程中也会创建线程并调用线程的成员函数。

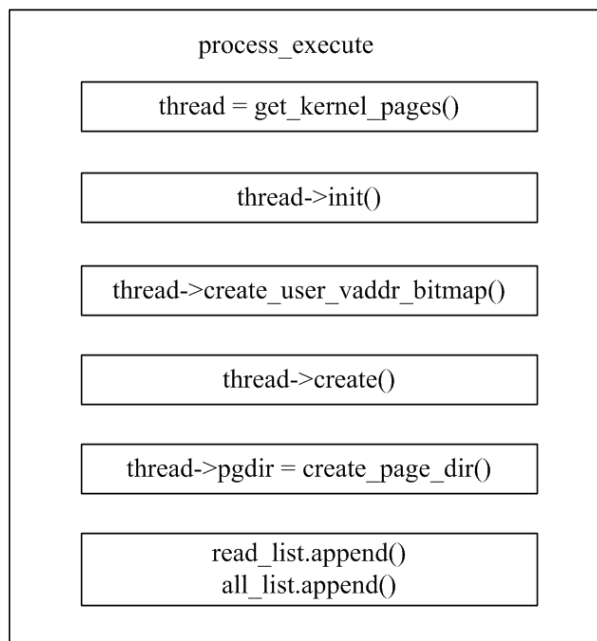


图 3.5 进程创建的步骤

如图 3.5 所示，函数 `process_execute` 主要分为六个步骤，依次执行。

(1) 调用 `get_kernel_pages` 申请 1 页内存作为进程的 PCB。由于进程和线程都使用相同的 PCB，所以此 PCB 也可看作一个内核线程（thread）。将此 PCB 地址赋给一个线程指针 `thread`，以便后面用已有的线程函数来处理进程的 PCB。

(2) 调用成员函数 `init_thread` 对（1）中获取的 PCB 页进行初始化。此部分初始化了进程的基本信息，填充了 PCB 的基本内容，如内核栈地址、时间片大小、父进程 PID 以及已打开文件列表等。

(3) 调用函数 `create_user_vaddr_bitmap` 为用户进程创建一个位图，用于管理此进程的虚拟地

址空间。

(4) 调用 `thread` 的成员函数 `create` 创建线程，此函数的作用是初始化线程栈 `thread_stack`，将待执行的函数和参数放到 `thread_stack` 中相应的位置。

(5) 调用函数 `create_page_dir` 为进程创建页表。

(6) 将此进程加入就绪队列 `ready_list` 和全部队列 `all_list`。

### 3.4.3 任务的切换

线程被创建之后，并不会马上执行，而是等待操作系统的调度。本文在内核中维护两个全局的队列：就绪队列 `ready_list` 和全部队列 `all_list`。就绪队列只存储正在等待运行的线程，全部队列存储所有进程，包括就绪的、阻塞的和正在执行的。

调度器的主要任务就是读写就绪队列，增删其中的结点。本文采用时间片轮转调度算法，线程每次在处理器上执行的时间由时间片 `ticks` 决定。在初始化线程的时候，已经将 `ticks` 的值赋为 `prio`，优先级越大，时间片越大。时间片是由时钟中断驱动的，系统每发生一次时钟中断，时钟中断的处理程序就将当前运行线程的 `ticks` 减 1。当 `ticks` 为 0 时，时钟的中断处理程序就会调用调度器 `schedule`，将该线程换下处理器，选择另一个线程并执行。

调度器 `schedule` 的实现如代码 3.6 所示：

代码 3.6

```
void schedule()
{
    ASSERT(intr_get_status() == INTR_OFF);
    task_struct* cur = running_thread();
    if (cur->status == TASK_RUNNING)
    {
        ASSERT(!elem_find(&thread_ready_list, &cur->general_tag));
        list_append(&thread_ready_list, &cur->general_tag);
        cur->ticks = cur->priority;
        cur->status = TASK_READY;
    }
    else {}
    if (list_empty(&thread_ready_list))
    {
        thread_unblock(idle_thread);
    }
    ASSERT(!list_empty(&thread_ready_list));
    thread_tag = NULL;
    thread_tag = list_pop(&thread_ready_list);
    task_struct* next = elem2entry(task_struct, general_tag, thread_tag);
```

```
next->status = TASK_RUNNING;
process_activate(next);
switch_to(cur, next);
}
```

函数开始时先关中断，避免同步异常。再获取当前正在执行的线程，若此线程当前为运行态，说明此次调度是由于时间片用完而引起的，否则说明当前线程因为被阻塞而需要换下处理器（如获取资源失败等）。如果就绪队列已空，则启动 `idle` 线程（空转线程）无限执行，直到有新的线程添加到就绪队列。在维护了就绪队列之后，将就绪队列中第一个线程切换为运行态并执行，至此调度器便完成了任务的切换。

### 3.4.4 任务的阻塞与唤醒

在多任务操作系统中，除了实现进程之间的切换，还有一个重要部分就是进程的阻塞与唤醒。当某一个任务在运行中，由于某些条件不能满足而暂停执行时，就要将其阻塞；而条件变得满足、这个任务需要再次运行时，就要将其唤醒。

任务的阻塞与唤醒本质是为了解决多任务的同步问题，本文引用了信号量作为同步机制。信号量是由一个资源总数和一个等待者队列组成的，结构如图 3.6 所示。

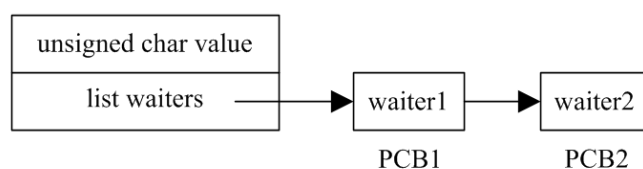


图 3.6 信号量结构

在信号量结构中，`value` 表示当前该资源的空闲数量，它的初始值是系统所拥有该资源总数。当有用户成功获取资源时，`value` 值减 1；当某用户释放该资源时，`value` 值加 1。当 `value` 值为 0 时，所有用户不能再获取该资源。

每一个可能发生同步异常的资源（如屏幕、键盘等）都需要一把锁来保证在同一时间只被一个用户使用。本文的锁便是用信号量来实现的，换言之，每种资源都维护了一个信号量。当一个运行中的线程无法获取到某种资源时，它便会阻塞自己，将自己添加到该资源信号量的等待队列 `waiters` 中，并由调度器切换到下一个任务运行。当某用户释放资源时，此用户会检查该资源信号量的等待队列，若队列不为空，则唤醒队列中的第一个用户，被唤醒的用户会被调度器添加到就绪队列中继续排队，等待处理器执行。

阻塞是线程主动发起的动作，而不是由调度器发起。但一旦线程阻塞自己之后，便只能等待别的进程释放资源并唤醒它，而不能主动唤醒自己。

### 3.5 本章小结

本章设计了内核架构，并实现了一些重要的功能模块。先研究了 IA32 架构上中断处理的流程和特权级检验机制，之后说明了中断发生时栈的变化，并完成了适当频率时钟信号的产生。然后规划了操作系统的内存池，并先后实现了页内存的分配和堆内存的管理。最后实现了任务管理系统，介绍了任务控制块的作用与结构，完成了进程的创建与任务的切换，并且实现了多任务同步机制，解决了系统并发问题。



## 第四章 文件系统与终端

### 4.1 文件系统分析

#### 4.1.1 inode 与块索引表

文件以块为单位存储在磁盘中，块的大小是扇区的整数倍。在 FAT 文件系统中，文件所有的块用链式结构来组织，在每个块的末尾存储下一个块的地址，将块与块连接在一起。因此文件是不连续的，文件中的块可以分布在各个零散的空间中，有效提升了磁盘的利用率<sup>[50]</sup>。但每访问一个结点，就要涉及一次磁盘寻道，使原本低速的设备在文件操作中效率更加低下。

UNIX 系统较为先进，将文件以索引结构来组织，避免了访问某一数据块时需要把前面所有数据块再遍历一次。在采用索引结构的文件系统中，文件的块依然可以分散到不连续的零散空间中，保留了磁盘高利用率的优点，更重要的是文件系统为每个文件的所有块建了一个索引表。

块索引表就是块地址数组，数组的每个元素记录数据块在磁盘中的地址（扇区偏移），数组元素下标是文件块的索引，第 n 个元素指向文件中的第 n 个块，这样在访问任意一个块的时候，速度大大提升。文本采用与此类似的索引结构来构建文件系统。

包含块索引表的索引结构称为 inode（index node，即索引结点），用来索引、跟踪一个文件的所有块。inode 是文件索引结构组织形式的具体体现，必须为每个文件都单独配备一个这样的元信息数据结构，因此每个文件对应一个 inode。本文的索引结构如图 4.1 所示。

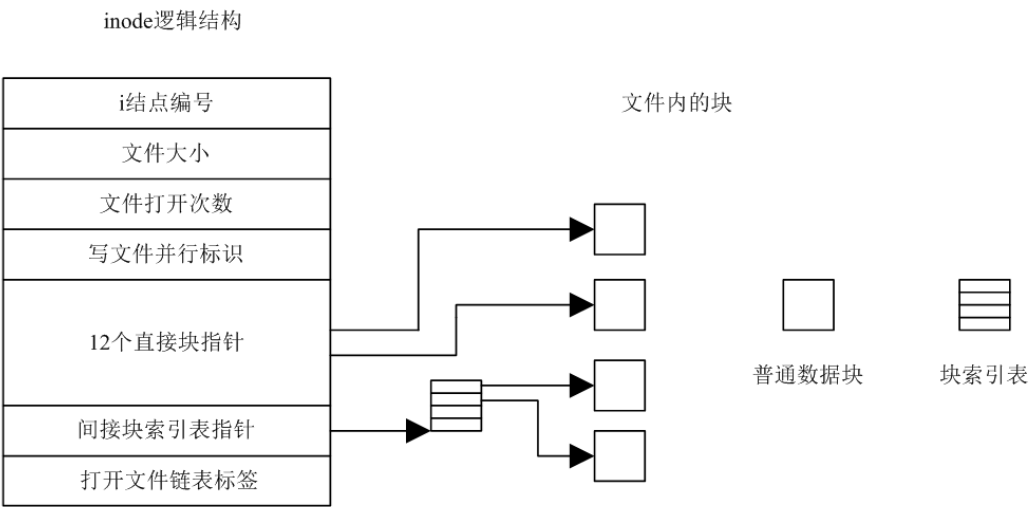


图 4.1 inode 结构

inode 结构几乎囊括了一个文件的所有信息，i 结点编号是指此 inode 的序号，通常等于它在 inode 数组中的下标。文件大小是指文件的字节尺寸（非所占块数）。文件打开次数记录了文件此时被多少个用户打开，每次有用户关闭此文件时，这个数值减 1，直到为 0 时关闭文件。写文件并行标识是一个布尔变量，当此布尔值为真时，表示已有用户正在对此文件进行写操作，其他用户不可进行写访问，用户写文件之前会先检查此标识。

块索引表也存储在 inode 结构中，本文的 inode 包含了 12 个直接块指针和 1 个间接块索引表指针。直接块指针是文件的 12 个块的直接地址，当文件尺寸大于 12 个块时，便启用间接块索引表。间接块索引表指针也指向了磁盘中的某一个块，这个块中又存储了 128 个块地址，所以文件的最大尺寸为 128 块。

4.1.2 文件系统布局

本文文件系统布局如图 4.2 所示。

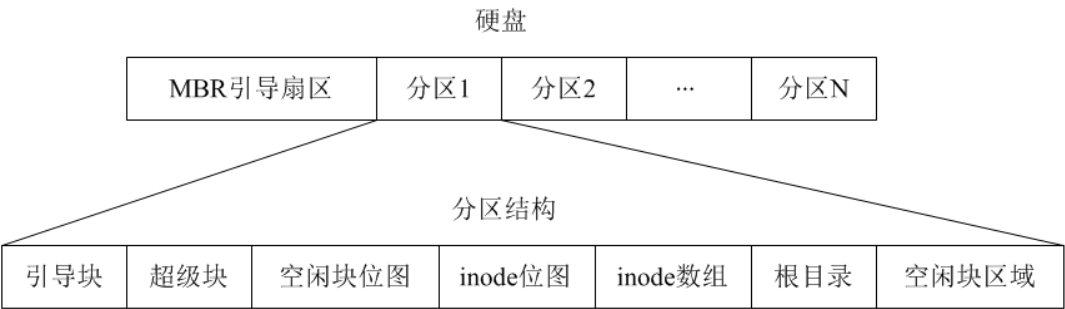


图 4.2 文件系统布局

在分区结构布局中，起始部分有空闲块位图、inode 位图、inode 数组等文件系统元信息，用户通过这些元信息来访问磁盘中的文件。超级块是保存文件系统元信息的目录，用户在超级块中获取文件系统配置信息，如本分区的总扇区数、inode 数量、分区起始地址等等<sup>[51]</sup>。为了管理本分区的空闲块，分区的元信息部分维护了一个空闲块位图来标记分区中所有空闲块的使用情况。在空闲块之后，放置了分区所包含的 inode 位图和数组，以此来管理分区所有文件。元信息后面是文件系统的根目录，其余所有文件和目录都放置在空闲块区域中。

4.2 创建文件系统

4.2.1 基础结构的实现

在创建文件系统之前，必须实现超级块、inode 和目录项等基础数据结构。为了降低系统

实现的复杂度，本文数据块大小与扇区大小一致，等于 512 字节。超级块大小刚好等于一个数据块的尺寸。超级块结构如代码 4.1 所示。

代码 4.1 超级块结构

```
struct super_block
{
    uint32_t magic;           // 用来标识文件系统类型
    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
};
```

内存中必须有目录与目录项等辅助结构，以支持目录相关的操作，目录与目录项结构如代码 4.2 所示。

代码 4.2 目录与目录项结构

```
class CDir
{
    friend CDir* dir_open(const CPartition& part, uint32_t inode_no);
    friend int32_t dir_remove(CDir* parent_dir, CDir* child_dir);
    friend CDir_entry* dir_read(const CDir& dir);
private:
    CInode* m_inode;
    uint32_t m_dir_pos;
    uint8_t m_dir_buf[512];
public:
    void dir_close();
    bool search_dir_entry(const CPartition& part, const string& name, const CDir_entry& dir_e);
    bool sync_dir_entry(const CDir_entry& p_de, void* io_buf);
    bool delete_dir_entry(CPartition* part, uint32_t inode_no, void* io_buf);
    bool dir_is_empty();
};

class CDirEntry
{
    friend class CDir;
```

```
friend CDir_entry* dir_read(const CDir& dir);  
public:  
    void create_dir_entry(string filename, uint32_t inode_no, uint8_t file_type, struct dir_entry* p_de);  
private:  
    string m_filename;  
    uint32_t m_i_no;  
    enum file_types m_f_type;  
};
```

Cdir 类是目录类，它的对象不存储在磁盘上。操作目录时，在内存中创建目录对象作为待操作目录的临时信息映像，用完即释放。其成员 m\_inode 是指针，指向已打开 inode 链表中的 inode 结构。成员 m\_dir\_pos 用于遍历目录时记录“游标”在目录中的偏移，所以其大小是目录项大小的整数倍。成员 m\_dir\_buf 用于目录的数据缓存，读取目录时用来存储返回的目录项。

目录项类 CDir\_entry 是连接文件名与 inode 的纽带。inode 结构中不包含文件名，所以需要此类型来完成二者的关联。成员 m\_i\_no 是文件对应的 inode 编号，也是此文件在 inode 数组中的索引。m\_f\_type 是此文件的类型，如目录和普通文件等。

## 4.2.2 文件系统的创建

创建文件系统就是在磁盘上填充分区所需要的元信息，包括超级块的位置及大小、空闲块位图的位置及大小、inode 位图的位置及大小、inode 数组的位置及大小、空闲块起始地址以及根目录起始地址<sup>[52]</sup>。创建步骤分为 5 个部分：

- (1) 根据分区 part 大小，计算分区文件系统各元信息需要的扇区数及位置；
- (2) 在内存中创建超级块，将（1）中计算的元信息写入超级块中；
- (3) 将（2）的超级块从内存写入磁盘；
- (4) 将元信息写到磁盘上各自的位置；
- (5) 将根目录写入磁盘。

## 4.3 文件描述符原理与实现

Linux 中所有的文件操作都基于文件描述符。读写文件的本质是通过文件的 inode 找到文件的扇区地址，随后读写该扇区，从而实现了文件内容的读写。几乎所有操作系统都可以在一个进程的运行过程中，多次打开同一个文件，同样该文件也可以被多个不同的进程同时打开。为实现文件任意位置的读写，执行读写操作时可以指定偏移量为操作的起始地址，此偏

移量相当于文件内的指针。因此文件每被打开一次，都可以指定操作偏移量，并随时记录每个用户操作该文件的当前偏移量。除了偏移量之外，文件操作还有读写属性等信息需要记录，这些信息只与每次的操作相关，而与文件本身无关。因此，为了记录文件操作的状态信息，引入文件结构。

每次打开一个文件便会产生一个文件结构，各文件结构都存储着操作的偏移量等信息，即使一个文件被多个用户同时打开，各自操作的偏移量也互不影响<sup>[53]</sup>。在全局区维护一个数组统一管理系统内所有的文件结构，该数组称为文件表。为了让进程能找到自己已经正在操作的文件结构，将属于各进程的文件结构在文件表中的下标存储在 PCB 中，以整数数组的形式统一管理，此数组的下标便称为文件描述符，是一个非负整数。文件结构与文件描述符的关系如图 4.3 所示。

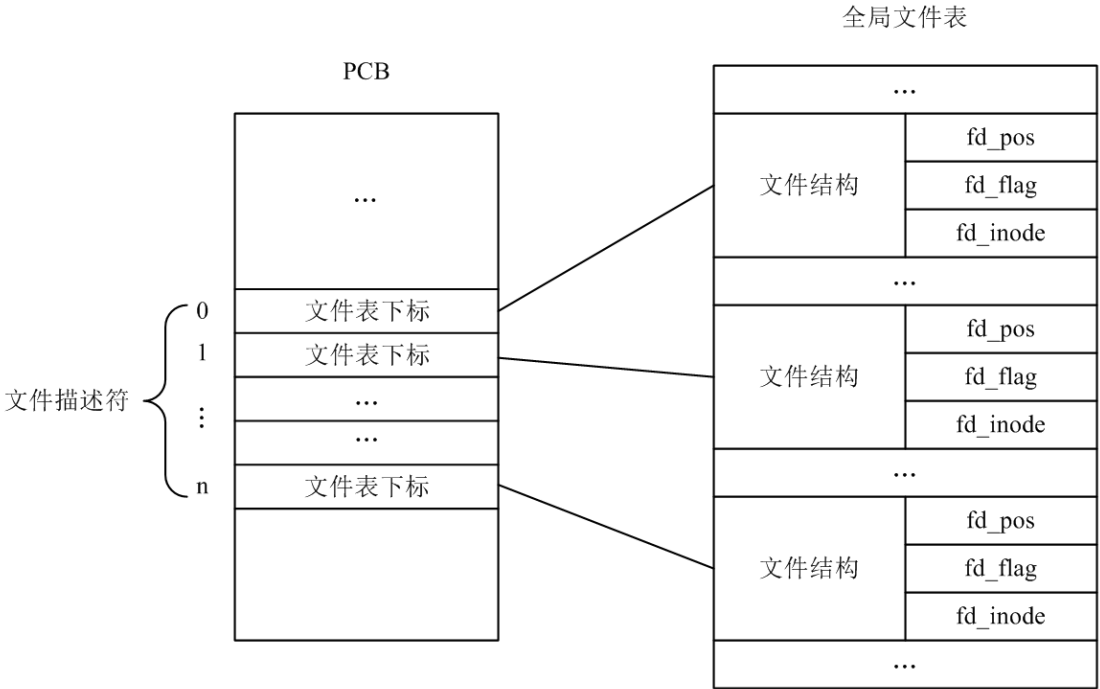


图 4.3 文件结构与文件描述符

图 4.3 中文件结构包含三个数据，fd\_pos 表示文件内的偏移量，fd\_flag 表示读写标识，fd\_inode 标识被操作文件的 inode 编号。当进程将文件描述符作为参数提交给文件系统时，文件系统用此文件描述符在该进程的 PCB 中找到对应的文件表下标，用该下标在全局文件表中索引相应的文件结构，利用文件结构中的 inode 编号，从而找到文件的数据块<sup>[54]</sup>。

在进程创建时便构建文件描述符数组，文件表是操作系统初始化的时候便已构建的全局数据结构，因此当进程打开文件时，文件描述符的创建只需在已有的本地描述符数组和全局文件表中填写信息即可。文件描述符的创建分为以下三个步骤：

- (1) 在全局的 inode 队列中新建一个 inode，然后返回该 inode 地址。

(2) 在全局文件表中找到一个空位，在该位置填充文件结构，使其 `fd_inode` 指向 (1) 中返回的 `inode` 地址，然后返回本文件结构在文件表中的下标值。

(3) 在 `PCB` 中的文件描述符数组中找到一个空位，使该位置的值指向上一步中返回的文件结构下标，并返回此空位的下标值，返回结果便是文件描述符。

## 4.4 文件操作的实现

### 4.4.1 文件的创建与删除

文件的创建是一项复杂的工作，必须在内存和磁盘中同步任何变更的数据，有以下几个关键点：

(1) 文件需要 `inode` 来描述大小、位置等属性，所以创建文件就要创建其 `inode`。先在 `inode` 位图中申请 `inode` 号，并更新 `inode` 位图，再在 `inode` 数组中填充此空闲 `inode`。

(2) `inode` 结构中包含的块索引地址是文件具体存储的扇区地址，所以需要向块位图申请可用块，并更新块位图。

(3) 新增加的文件必然存在于某个父目录，所以该父目录的大小增加一个目录项的尺寸。新增加的文件对应的目录项需要写入父目录的某个块地址索引中。若原有扇区已满，则写入申请新扇区来存储目录项。若直接块索引地址刚好用完，则启用间接索引。

(4) 若其中某步操作失败，需要回滚之前已成功操作。

(5) `inode_bitmap`、`block_bitmap`、新文件的 `inode` 及所在目录的 `inode` 等信息位于内存中已被改变的数据一律同步到硬盘。

文件删除是文件创建的逆操作，只须注意各项资源的回收和磁盘与内存信息的同步即可。

### 4.4.2 文件的打开与关闭

打开文件的核心函数是 `file_open` 函数，实现如代码 4.3 所示。

代码 4.3 文件的打开

```
int32_t file_open(uint32_t inode_no, uint8_t flag)
{
    int32_t fd_idx = get_free_slot_in_global();
    if (fd_idx == -1)
    {
        printk("exceed max open files\n");
        return -1;
    }
}
```

```
}
file_table[fd_idx].fd_inode = inode_open(cur_part, inode_no);
file_table[fd_idx].fd_pos = 0;
file_table[fd_idx].fd_flag = flag;
bool* write_deny = &file_table[fd_idx].fd_inode->write_deny;
if (flag == O_WRONLY || flag == O_RDWR)
{
    enum intr_status old_status = intr_disable();
    if (!(*write_deny))
    {
        *write_deny = true;
        intr_set_status(old_status);
    }
    else
    {
        intr_set_status(old_status);
        printk("The file can't be write now, try again later\n");
        return -1;
    }
}
return pcb_fd_install(fd_idx);
}
```

file\_open 接受 2 个参数，inode 编号 inode\_no 和打开标识 flag，函数功能是打开编号为 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。函数开头从文件表 file\_table 中获取空位的下标，然后进行初始化。在将 write\_deny 指针指向此 inode 的写文件同步标识 write\_deny 元素。如果此次以写文件的方式打开，则需要判断 write\_deny 标识，若此标识为真，则失败返回；否则将标识改为真，再将全局描述符下标安装到进程或线程自己的文件描述符数组 fd\_table 中，文件成功打开。文件的关闭只需释放相应资源。

#### 4.4.3 文件的写入与读取

文件的内容存储在硬盘里属于它的块中，这些块的地址存储在文件 inode 结构中的块索引表中，直接块索引表存储了前 12 个块的地址(扇区偏移)，如果 12 个块的尺寸无法容纳文件，就会使用间接块。

基于以上文件存储机制，需要考虑文件的写入给这个文件的块带来的变化。先分为两种大情况：不需要增加新的块和需要增加新的块，再向下细分：

(1) 不需要增加新的块：

a)写在直接块中；

b)写在间接块中。

(2) 需要增加新的块:

a)原本只用了直接块,写的过程也只用直接块;

b)原本只用了直接块,写的过程需要用间接块;

c)原本就用了间接块。

函数中先申请能容纳 140 个扇区地址的内存作为块地址数组,命名为 `all_blocks`。结合文件当前的大小和需要写入的字节数来判断具体属于上述哪一种情况,然后在每一个分支之下分别按照不同的需要来填写该数组,同时若需要增加 block,则在 `inode` 的索引表或硬盘间接块表中相应位置增加一个 block。若需要添加间接块索引表,则在硬盘中申请一个扇区作为间接块表,将其地址填写在 `inode` 中,再在这个间接块表中添加新的块地址。同时将磁盘中需要写的块地址都填在 `all_blocks` 中。

准备完 `all_blocks` 数组后,便收集了将要写的所有块的块地址。接下来只用遍历 `all_blocks` 中每一个块地址,将待写的内容拷贝在一个临时缓冲(512B)中,将临时缓冲写入到对应的块地址。

文件的读取和写入的思想类似,先将需要读的块地址准备在 `all_blocks` 中,由于读取不会造成块数量的变化,所以只分为 3 种情况:

(1) 起始块和终止块都属于直接块;

(2) 起始块是直接块,终止块是间接块;

(3) 起始块和终止块都是间接块。

分为这三种情况将需要读的块地址填写在 `all_blocks` 缓冲区中,然后遍历缓冲区,逐块从硬盘中将数据读出来即可。

#### 4.4.4 目录的创建与删除

实现 `sys_mkdir` 函数创建目录,操作与创建文件略有不同。创建目录时创建的新目录虽然是空的,但是要在其中加入“.”和“..”两个目录项,再在这两个目录项中填入目录本身和父目录的 `inode` 地址。而 `inode` 和 `block` 等资源的分配和文件基本一致, Linux 中的目录本就是一种文件,也称作目录文件。

完成删除目录的函数是 `sys_rmdir`。先要判断目录是否为空,本文只支持删除空目录。如果目录中只有两个目录项(“.”和“..”),则目录为空,回收其资源,并删除其在父目录中的目录项,不允许删除根目录。删除目录时的资源回收和删除文件基本相同。



## 4.5 终端的实现

### 4.5.1 fork 克隆进程

fork 是 Linux 系统中的分叉函数，作用是克隆进程。fork 的返回值有三种可能：子进程的 pid，0 或-1。由于 Linux 中父进程没有获取子进程 pid 的方法，所以 fork 会给父进程返回子进程的 pid；子进程可以通过 getpid 获得父进程的 pid，且没有 pid 为 0 的进程，所以 fork 给子进程返回 0，这样从 fork 的返回值是否为 0 便可以判断当前执行的程序体是在父进程中还是在子进程中；若执行失败，则 fork 返回-1。

fork 利用一个已有进程克隆出一个新进程并使新进程执行，新进程之所以能够执行，本质上是因为它具备程序体，这其中包括代码和数据等资源<sup>[55]</sup>。因此 fork 就是把某个进程的全部资源复制了一份，然后让处理器的 cs:eip 寄存器指向新进程的指令部分。所以实现 fork 要分为两步：

(1) 将父进程资源拷贝至子进程，拷贝内容有：a) 进程的 PCB；b) 程序体，即代码段和数据段等；c) 用户栈，编译器会在栈中创建局部变量；d) 内核栈，进入内核态时，用它来保存上下文环境和存储局部变量；e) 虚拟地址池，每个进程拥有独立的内存空间，用虚拟地址池来管理虚拟空间；f) 页表，建立子进程的虚拟空间。

(2) 让处理器执行克隆出来的子进程，只需将子进程加入到就绪队列 ready\_list 中即可。

### 4.5.2 系统调用 wait 和 exit

wait 和 exit 是经常成对使用的系统调用，父进程用它们来管理子进程的运行，本节实现了这两个系统调用。

wait 通常是由父进程调用的，它的作用是阻塞父进程自己，直到任意一个子进程结束运行。当有子进程结束运行时，会将退出时的返回值传递给父进程并唤醒父进程<sup>[56]</sup>。通过阻塞父进程，可以解决父子进程同步的问题；通过获取子进程状态，父进程可以获知子进程的运行结果。wait 的实现如代码 4.4 所示。

代码 4.4 系统调用 wait

```
pid_t sys_wait(int32_t* status)
{
    task_struct* parent_thread = running_thread();
    while (1)
    {
```

```
list_elem* child_elem = thread_all_list.list_traversal(find_hanging_child, parent_thread->pid);
if (child_elem != NULL)
{
    task_struct* child_thread = elem2entry(struct task_struct, all_list_tag, child_elem);
    *status = child_thread->exit_status;
    uint16_t child_pid = child_thread->pid;
    thread_exit(child_thread, false);
    return child_pid;
}
child_elem = thread_all_list.list_traversal(find_child, parent_thread->pid);
if (child_elem == NULL)
{
    return -1;
}
else
{
    thread_block(TASK_WAITING);
}
}
```

当父进程调用 `wait` 时，优先处理挂起状态的任务，若有挂起的子进程，则将子进程的退出状态保存至 `status` 中，并从就绪队列和全部队列中删除子进程表项，彻底回收子进程资源，返回子进程 `pid`。若没有挂起的子进程，则判断原进程是否有子进程，若没有，则 `wait` 返回-1，否则父进程阻塞自己，等待唤醒。父进程被唤醒后，再跳至第一步检测是否有挂起的子进程，此时必然会有已结束运行的子进程。

`exit` 的作用是使进程主动退出，结束运行。任何时候进程结束时都会调用 `exit`，即使程序中未写入显式调用 `exit` 的代码，在 C 运行库的最后也会发起 `exit` 的调用。`exit` 是由子进程调用的，调用时内核会将进程的除了 `PCB` 以外的资源全部回收。`exit` 的执行可以分为以下几步：

- (1) 将运行状态 `status` 存入 `PCB` 中。
- (2) 将所有的子进程过继给 `init` 进程（`pid` 为 1 的进程，系统初始化时创建）。
- (3) 回收进程资源，包括所用内存和进程页表，但不回收 `PCB`，也不会从就绪队列和全部队列中删除该进程。
- (4) 如果父进程正在等待子进程退出，则将父进程唤醒。
- (5) 将自己挂起等待父进程获取其 `status` 并释放其 `PCB`。

当子进程调用 `exit` 退出时，若父进程没有调用 `wait` 等待，则子进程的 `PCB` 没有得到释放，子进程会变成僵尸进程。若进程结束时，它的子进程还在运行，则这些子进程会变成孤儿进程，自动过继给 `init` 进程，变为 `init` 的子进程。

4.5.3 shell 命令与环形缓冲区

操作系统是为用户服务的，要实现和用户的交互，操作系统必须感知用户的输入并给予反馈，也就是为用户提供一个交互接口。在 Windows 中，图形界面的资源管理器和命令行窗口都是交互接口。Linux 中采用命令行窗口的方式为用户提供服务，它被称为 shell 程序<sup>[57]</sup>。

shell 的功能是获取用户的键入，然后分析输入的字符串，判断是内部命令还是外部命令，然后执行不同的策略。shell 命令是由多个字符组成的，并且要以回车键结束，因此在键入命令的过程中，必须要有一个缓冲区将已键入的信息保存起来，当凑成完整的命令行时再一并解析处理。

shell 缓冲区的管理是一个“生产者与消费者”问题，必须要解决多个线程协同工作，并防止缓冲区被破坏。“生产者与消费者”模型如图 4.4 所示。



图 4.4 生产者与消费者

在“生产者与消费者”模型中，有一个或多个生产者、一个或多个消费者和一个固定大小的缓冲区，所有生产者和消费者共享这个缓冲区，生产者每次生产一个数据放到缓冲区中，消费者每次从缓冲区消费一个数据<sup>[58]</sup>。在同一时刻，缓冲区只能被一个生产者或消费者使用。当缓冲区已满时，生产者不能继续向缓冲区添加数据，当缓冲区为空时，消费者不能从缓冲区消费数据，过度生产或过度消费都会造成缓冲区的破坏。

缓冲区是多个线程共同使用的共享内存，内存的缓冲区也是按地址来访问的，因此缓冲区实际上是线性存储。但是可以设计出逻辑上环形的内存缓冲区，使得缓冲区的操作更加合理，如图 4.5 所示。

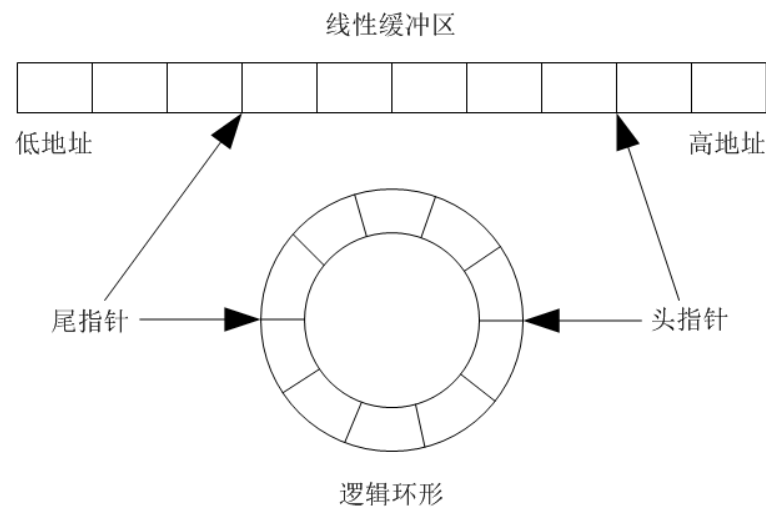


图 4.5 环形缓冲区

为了管理缓冲区的访问，本文提供两个指针：一个头指针和一个尾指针。头指针用于生产者向缓冲区写数据，每次写入一个数据时头指针加 1，指向下一个可写入数据的地址；尾指针用于消费者消费数据，每消费一个数据时尾指针加 1，指向下一个可消费的数据。缓冲区相当于一个队列，数据在队列头被写入，在队尾处被读出。缓冲区类如代码 4.5 所示。

代码 4.5 环形缓冲区

```
class ioqueue
{
public:
    void ioqueue_init();
    bool ioq_full();
    char ioq_getchar();
    void ioq_putchar(char byte);
    uint32_t ioq_length();
private:
    CLock lock;
    task_struct* producer;
    task_struct* consumer;
    int8_t buf[bufsize];
    int32_t head;
    int32_t tail;
};
```

4.5.4 命令行的实现

在 Linux 的 shell 或微软的 DOS 中，命令都可以分为两大类：外部命令和内部命令。外部命令是指该命令是一个存储在文件系统上的外部程序，执行该命令时实际上是从文件系统

上加载该程序到内存后运行的过程，外部命令是以进程的方式执行；内部命令也称为内建命令，是系统本身提供的功能，它们并不以单独的程序文件存在，只是一些独立的功能函数，系统执行命令时实际上是在调用这些函数<sup>[59]</sup>。本文以内部命令的方式实现了 `ls`、`cd`、`mkdir`、`rmdir`、`rm`、`pwd`、`ps` 和 `clear` 等内部命令。

外部命令在执行时，`shell` 进程会先 `fork` 一个子进程，然后调用 `exec` 去执行命令，`exec` 从磁盘上加载外部命令对应的程序，然后执行该程序。关键步骤如代码 4.6 所示。

代码 4.6 外部命令的处理

```
int32_t pid = fork();
if (pid)
{
    int32_t status;
    int32_t child_pid = wait(&status);
    if (child_pid == -1)
    {
        panic("my_shell: no child\n");
    }
    printf("child_pid %d, it's status: %d\n", child_pid, status);
}
else
{
    make_clear_abs_path(argv[0], final_path);
    argv[0] = final_path;
    stat file_stat;
    memset(&file_stat, 0, sizeof(struct stat));
    if (stat(argv[0], &file_stat) == -1)
    {
        printf("my_shell: cannot access %s: No such file or directory\n", argv[0]);
        exit(-1);
    }
    else
    {
        execv(argv[0], argv);
    }
}
```

当处理外部命令时，当前的进程 `shell` 先 `fork` 出子进程，接着父进程通过 `wait` 阻塞自己，等待子进程执行完毕。在子进程中，先调用 `make_clear_abs_path` 函数（代码见附录）获取可执行文件 `argv[0]` 的绝对路径到 `final_path` 中，然后将 `argv[0]` 重新指向 `final_path`。接着调用 `stat` 判断可执行文件是否存在，若存在则执行该文件。本文以外部命令的方式实现了 `cat` 命令，以输出文件内容。

## 4.6 管道设计与实现

### 4.6.1 管道设计

进程间通信的方式有很多种，如消息队列、共享内存、socket 网络通信以及管道等，本节实现管道通信，以支持父子进程间的通信。管道可看作一种文件，只是该文件并不存在于文件系统上，而是存在于内存中，因此也要使用 `open`、`close`、`read` 和 `write` 等方法来操作管道。管道实质上是内核空间中的内存缓冲区，常被多个进程共享。

管道是用于存储数据的中转站，当某个进程向管道写入数据后，该数据很快就会被另一个进程读取，之后便可以用新的数据覆盖老数据。因此使用内核空间的环形缓冲区来实现管道。管道有两端，一端用于从管道中读出数据，另一端用于向管道写入数据。这两端使用文件描述符来读写。用户进程为内核提供一个长度为 2 的文件描述符数组 `fd`，内核在数组中填入管道操作的两个描述符，`fd[0]`用于读取管道，`fd[1]`用于写入管道，进程与管道的读写关系如图 4.6 所示。

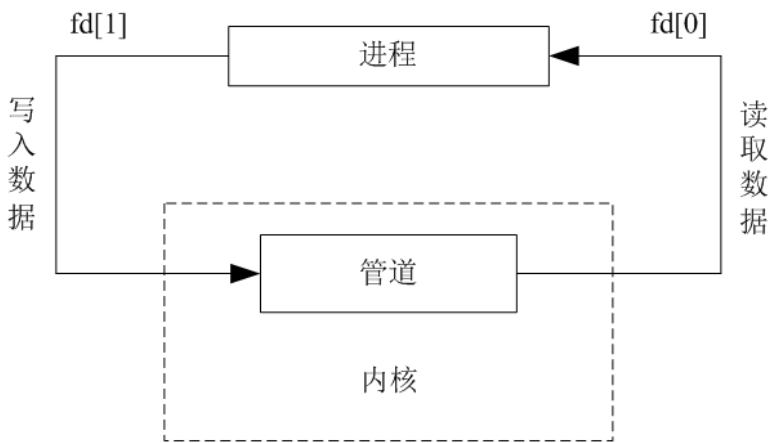


图 4.6 管道

在进程创建管道之后，调用 `fork` 克隆一个子进程，子进程继承了父进程的管道描述符，它们的管道描述符指向了同一个管道，这样便实现了父子进程间的通信。

### 4.6.2 在 shell 中支持管道

在 Linux 的 shell 中，管道符是“`|`”，命令行中可以有多多个管道符，管道符的左右两端各有一条命令，因此若命令行中包含管道符，则至少有两条命令。在命令行中支持管道符通常是为了数据的二次加工、过滤出感兴趣的部分。

管道的核心技术是输入输出重定向。通常情况下键盘是程序的标准输入，屏幕是程序的标准输出。如果命令的输入不来自键盘，而来自于文件，则称为输入重定向；若命令的输出终点不是屏幕，而是文件，则称为输出重定向。重定向实现如代码 4.7 所示。

代码 4.7 文件描述符重定向

```
void sys_fd_redirect(uint32_t old_local_fd, uint32_t new_local_fd)
{
    task_struct* cur = running_thread();
    if (new_local_fd < 3)
    {
        cur->fd_table[old_local_fd] = new_local_fd;
    }
    else
    {
        uint32_t new_fd = cur->fd_table[new_local_fd];
        cur->fd_table[old_local_fd] = new_fd;
    }
}
```

函数 `sys_fd_redirect` 的功能是将旧文件描述符 `old_local_fd` 重定向为新文件描述符 `new_local_fd`。文件描述符是 PCB 中数组 `fd_table` 的下标，数组元素的值是全局文件表 `file_table` 的下标，因此将数组 `fd_table` 中下标为 `new_local_fd` 的元素值赋给下标是 `old_local_fd` 的元素，便完成了文件描述符的重定向。

在 `shell` 的命令行处理模块解析中添加对管道符“|”的解析步骤，将管道符左侧的命令的输出重定向为临时管道，并将管道符右侧命令的输出重定向为此管道，就实现了 `shell` 对管道的支持。

## 4.7 本章小结

文件系统是操作系统的核心模块，本章搭建了一个较为完善的文件系统，并为用户提供了一个命令终端。本章先介绍了文件系统的基本数据结构，并详述了磁盘中文件系统的布局。之后分别在内存和磁盘中创建了文件系统，并实现了文件描述符作为文件操作的基础。然后实现了文件的创建与删除、读取与写入等多种文件操作。最后实现了 `shell` 终端，让操作系统能执行用户输入的命令，并加入输入输出重定向技术，使 `shell` 支持命令行中的管道操作。

## 第五章 系统测试与结果分析

### 5.1 工作与测试环境的搭建

本文的全部工作是在微软公司的 Windows10 下进行的,在 Windows10 工作环境中安装了 Source Insight、Vmware、Bochs 等软件。Source Insight 是一个面向项目开发的程序编辑器,它能分析源代码并自动显示有用的上下文信息,在 Windows 环境下用它来编写操作系统代码。安装 Vmware,并在 Vmware 上添加一个 CentOS 虚拟机,在此虚拟机中安装 NASM 和 GCC 编译器,用这个 CentOS 虚拟机来编译所有代码。在 Windows 环境下安装 Bochs 虚拟机,并在 bochs 上运行和调试操作系统。调试之前的准备工作如下:

- (1) 使用 dd 命令创建一个虚拟磁盘作为启动盘,命名为 myos.img。
- (2) 将编译生成的二进制可执行文件写入 myos.img。
- (3) 将 myos.img 拷贝至 Bochs 的目录下。
- (4) 更改 Bochs 配置文件,将启动盘设置为 myos.img,将内存设置为 128M,启动 Bochs 虚拟机。

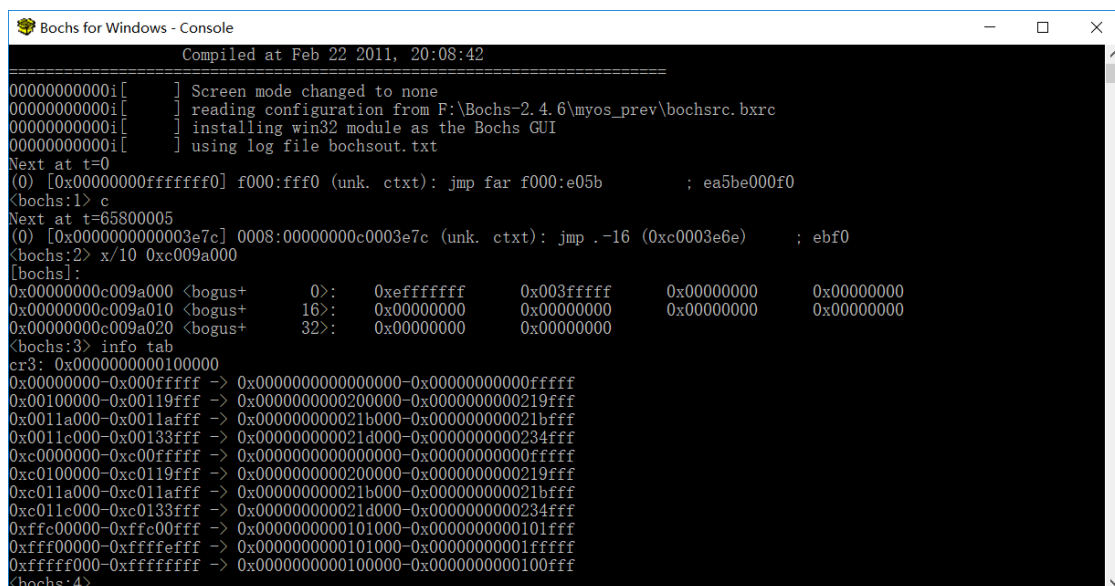
### 5.2 测试结果分析

#### 5.2.1 内存测试

在调试版 bochs 中启动磁盘映像文件 myos.img,在调试窗口输入命令来查看内存使用情况,结果如图 5.1 所示。先用 x/10 0xc009a000 查看了内核内存池的位图所在地址,在位图中低若干位为 1,其余高位值皆为 0,这是由于内核程序和已开辟的进程占用了前面的若干页内存,因此低若干位为 1,其余内存皆为可用内存,所以为 0。

先用 info tab 命令查看了页表中虚拟地址与物理地址的映射关系,左边是虚拟地址的范围,右边是所映射的物理地址。在 Bochs 的输出中,如果虚拟地址和物理地址都是连续的,就会合并到一起输出。





```

Bochs for Windows - Console
Compiled at Feb 22 2011, 20:08:42
=====
00000000000i[ ] Screen mode changed to none
00000000000i[ ] reading configuration from F:\Bochs-2.4.6\myos_prev\bochsrc.bxrc
00000000000i[ ] installing win32 module as the Bochs GUI
00000000000i[ ] using log file bochsout.txt
Next at t=0
(0) [0x00000000fffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b ; ea5be00f0
<bochs:1> c
Next at t=65800005
(0) [0x0000000000003e7c] 0008:00000000c0003e7c (unk. ctxt): jmp .-16 (0xc0003e6e) ; ebf0
<bochs:2> x/10 0xc009a000
[bochs]:
0x00000000c009a000 <bogus+ 0>: 0xffffffff 0x003fffff 0x00000000 0x00000000
0x00000000c009a010 <bogus+ 16>: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000c009a020 <bogus+ 32>: 0x00000000 0x00000000 0x00000000 0x00000000
<bochs:3> info tab
cr3: 0x0000000000100000
0x00000000-0x000fffff -> 0x0000000000000000-0x000000000000fffff
0x00100000-0x00119fff -> 0x0000000000200000-0x0000000000219fff
0x0011a000-0x0011afff -> 0x000000000021b000-0x000000000021bfff
0x0011c000-0x00113fff -> 0x000000000021d000-0x000000000021dfff
0xc0000000-0xc00fffff -> 0x0000000000000000-0x000000000000fffff
0xc0100000-0xc0119fff -> 0x0000000000200000-0x0000000000219fff
0xc011a000-0xc011afff -> 0x000000000021b000-0x000000000021bfff
0xc011c000-0xc0113fff -> 0x000000000021d000-0x000000000021dfff
0xffc00000-0xffc0ffff -> 0x0000000000101000-0x0000000000101fff
0xffff0000-0xffffefff -> 0x0000000000101000-0x0000000000101fff
0xfffff000-0xfffffff -> 0x0000000000100000-0x0000000000100fff
<bochs:4>

```

图 5.1 内存使用情况

第一个虚拟地址映射是：0x00000000~0x000fffff -> 0x00000fffff~0x00000fffff。左侧的虚拟地址是低端 1MB 内存，对应的地址是物理地址的低端 1MB 地址。这是因为本文在建立分页机制后为内核创建了一个页表，并为此页表前 256 个页表项分配了物理页，用这 1MB 内存来存放内核。测试结果符合预期。在第五行中，虚拟地址是 0xc0000000~0xc00fffff，物理地址也是低端 1MB 内存，这是由于内核功能是由操作系统与所有用户进程所共享的，因此让内核虚拟内存的低端 1MB 和用户虚拟内存的低端 1MB 都与物理内存的低端 1MB 内存映射，这 1MB 内存中所存储的便是内核。测试结果符合预期

### 5.2.2 多任务测试

在 main 函数中创建两个进程：prog\_a 与 prog\_b，将 prog\_a 的优先级设置为 8，将 prog\_b 的优先级设置为 4，并在两个进程中分别无限打印字符串“userA”和“userB”，运行结果如图 5.2 所示。

进程 prog\_a 打印的字符串比进程 prog\_b 打印的多，并且没有发生同步问题，显示屏有序地打印了整个字符串。

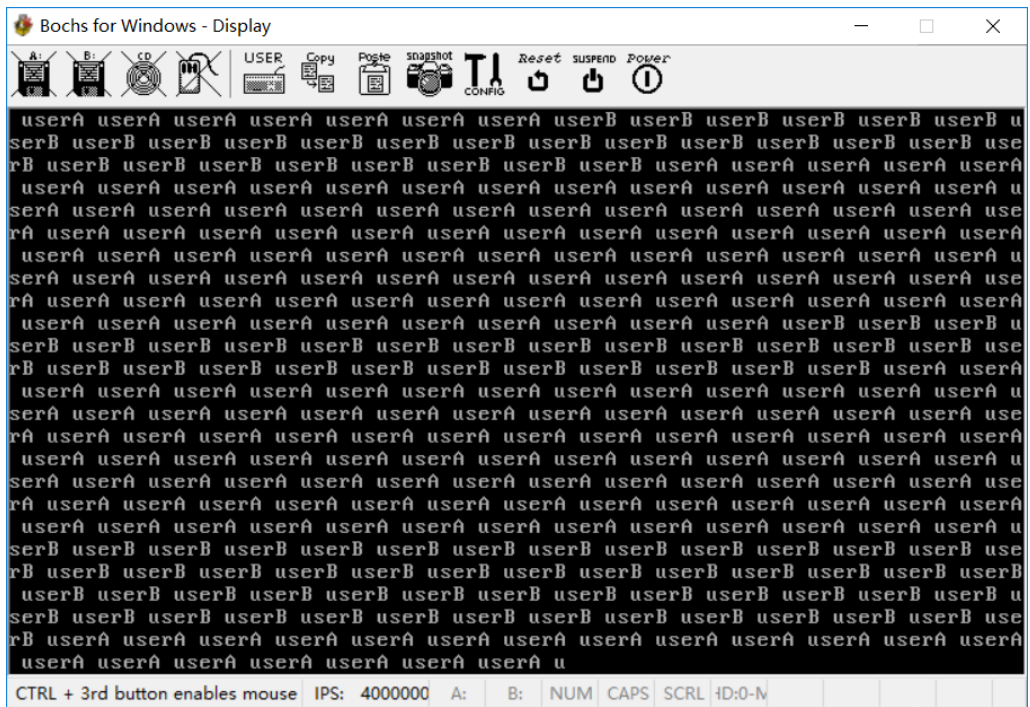


图 5.2 多进程

在用调试版运行虚拟磁盘映像文件，进程 prog\_a 的地址是 0xc00015df，所以在此处设置断点，然后查看 cs 寄存器的值，调试结果如图 5.3 所示。

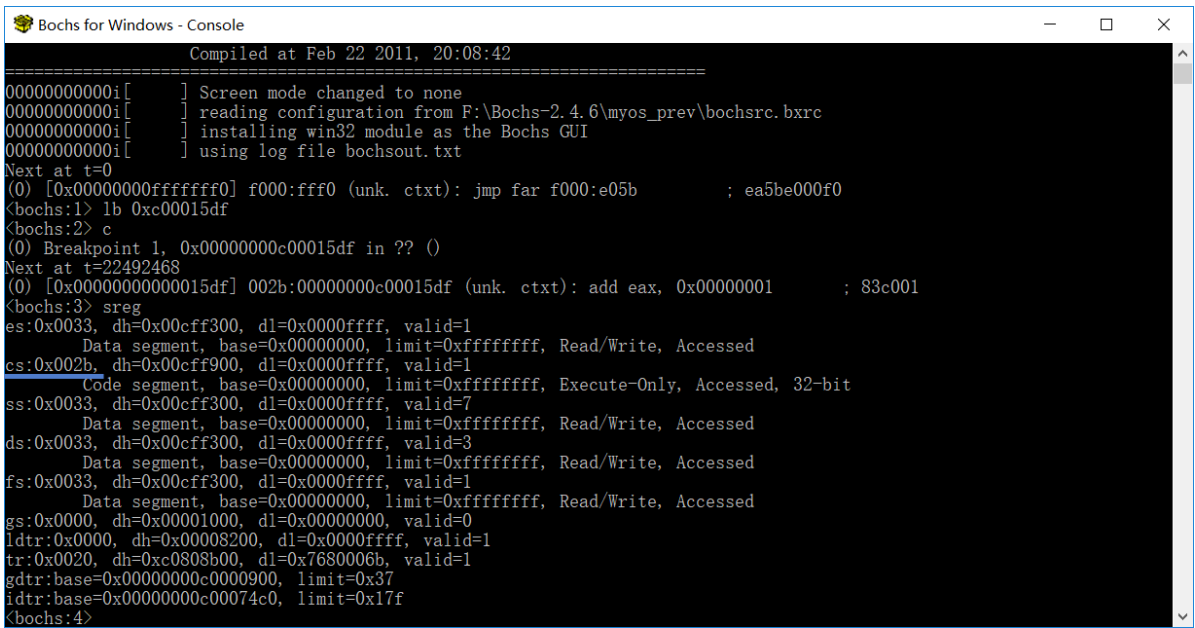


图 5.3 进程调试

先用 lb 设置断点，使 bochs 在虚拟地址 0xc00015df 处停下，用命令 c 持续执行，当 bochs 停住后，执行命令 sreg 查看段寄存器，此时 cs 的值为 0x002b，低 4 位的二进制值是 1011，低两位为 RPL，值为 3，所以此时用户进程是在 3 特权级下，与预期符合。

5.2.3 文件处理命令测试

运行磁盘文件，输入命令 help 结果如图 5.4 所示。

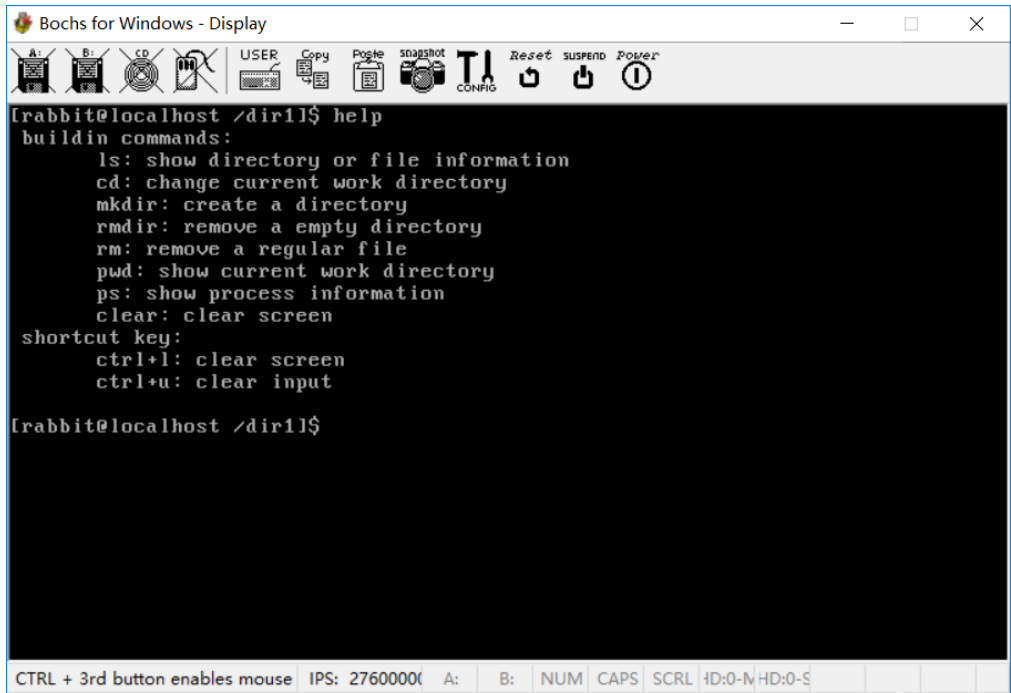


图 5.4 help 命令测试

本文支持的内部命令都显式在下方，并且提示了快捷键 ctrl+l 和 ctrl+u 分别是清屏功能和清空输入功能。

输入其他命令，测试结果如图 5.5 所示。

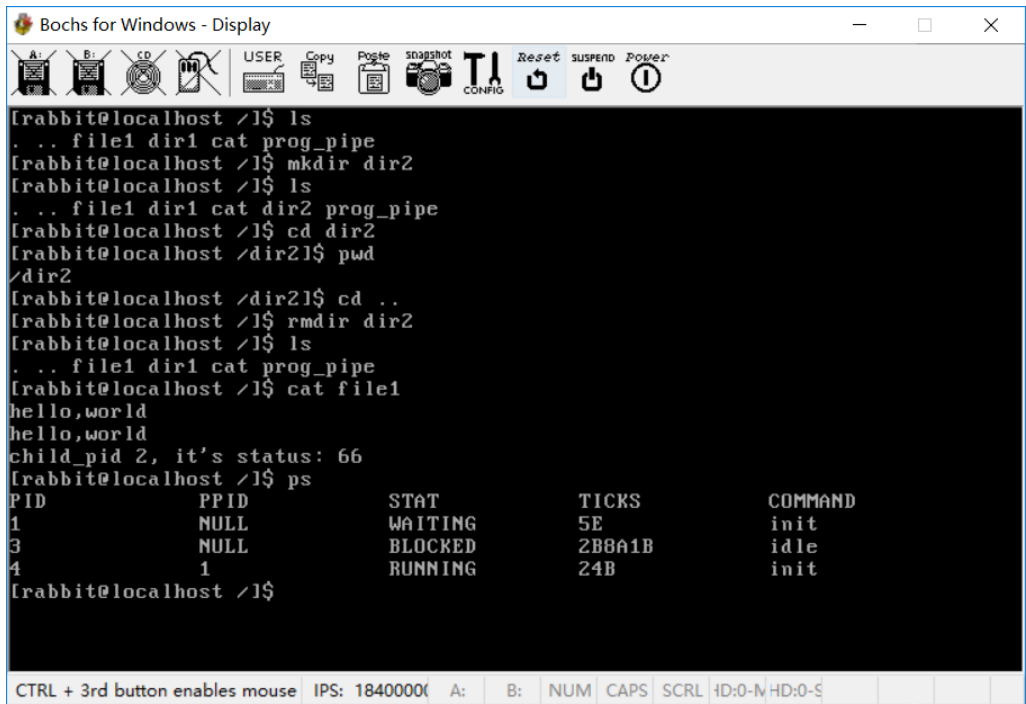


图 5.5 命令测试

输入各命令测试结果，如 `ls` 显式了当前目录下的所有子目录和文件，输入 `cat` 外部命令查看文件 `file1` 的内容，结果完全符合预期。

### 5.3 本章小结

本章对操作系统的各核心模块进行了相应的测试工作。先描述了本文工作环境的搭建，然后从操作系统的内存管理、多任务管理和文件管理这三个角度分别测试了系统的功能。经测试，本文预期功能基本实现。

## 第六章 总结与展望

### 6.1 论文工作总结

本文从计算机底层原理出发,结合现有的操作系统架构与操作系统应该具备的某些功能,从引导项开始,最后到终端的输出显示,完成了一个较为完善的操作系统。本文主要研究内容详述如下:

(1) 设计主引导记录(MBR)和内核加载器(LOADER)。利用主机硬件的 BIOS 启动功能,编写启动盘的主引导记录,在主引导记录中跳转到内核加载器,再在加载器中从实模式跳转至保护模式并加载和运行内核。

(2) 研究与设计中断处理程序。研究保护模式下的中断处理机制,编写中断处理程序,设置定时器 8253 以获取特定频率的时钟信号。

(3) 实现内存管理系统。为内核程序与用户程序规划虚拟与物理内存池,实现页内存的分配以供内核程序调用,进而实现更小粒度的堆内存分配以供用户程序调用。

(4) 设计并实现任务管理系统。研究并实现进程与线程实体,编写任务调度器,建立内存中的全部任务列表和就绪任务列表。为处理多任务的同步问题,实现锁以防止多个程序抢占资源产生异常。

(5) 设计并实现与该操作系统相适应的文件系统。在磁盘中写入元信息、超级块等数据结构管理磁盘分区,在内存中建立已挂载分区的数据映像。以内建命令和外部命令的方式各实现一些基本的文件操作,并在用户进程 shell 中提供接口,构成一个较为完善的文件管理系统。

本文的研究工作以操作系统技术为核心,涉及到计算机组成原理、微机原理、数据结构以及编程语言等众多领域的计算机软硬件知识。在论文的整个研究与实践过程中,我不仅对各个领域的基础知识初窥门径,更是将相应技术相互结合加以应用;不仅提高了自主学习的能力,更对自己的工程实践能力有了全新的认识。

### 6.2 论文贡献与创新

本论文的研究工作主要体现在以下几个方面:

(1) 研究了早期的 Linux 版本,自主完成了全新的操作系统的设计与开发,最终生成镜像文件,只需要虚拟机便可以运行自制的操作系统。

(2) 在模块设计方面,使用层次化设计方法,来防止模块间的多次调用与重复调用,并且将

系统分为系统内核级与用户应用程序级，以防止用户级程序任意访问或修改内核地址信息。

（3）区别于市面上其他的操作系统，使用 C++ 语言编写内核，获得了更好的封装性和可移植性。

（4）搭建了较为完善的全新文件系统，让操作系统支持基本的文件操作。

## 6.3 研究展望

由于时间和技术等诸多因素，本文实现的操作系统还存在一些缺陷，无法满足日常使用，更不能与市场上已经成熟的操作系统相比较。该系统还需要在以下几个方面进行改善与改进：

（1）虽然对模块进行了分层，但仍然存在维护性比较差的问题。

（2）只完成了最简单的功能。成熟的现代操作系统功能十分复杂，比如说网络功能，图形化界面等功能都需要继续研究。

（3）在算法运用方面，本文的操作系统与市面先进的操作系统也有较大差距。Linux 系统的进程拷贝采用了写时拷贝技术，进程的加载采用了页面置换算法，大大提高了运行效率和内存利用率。因此本文还需要改进相关算法以改善系统性能。

## 参考文献

- [1] 孙浩林. 支持中文的操作系统内核的研究与实现[D]. 北京交通大学, 2017.
- [2] 李文. 在历史与未来的结点上——纪念IBM S/360大型主机应用40周年[J]. 中国金融电脑, 2004(6):68-77.
- [3] 百度统计\_流量研究院.操作系统市场份额[OL]. 北京:百度公司, 2017.<http://tongji.baidu.com/data/os>.
- [4] 杜佩佩.Linux内核开发者群体的结构演化及行为特征研究[D].大连理工大学,2015.
- [5] 解卫静.基于众科理论的国产操作系统演化和推广研究[D].石家庄铁道大学,2017.
- [6] 任怡,吴庆波,戴华东,等.通用操作系统对比评测标准研究[J].计算机科学,2011,38(11):286-290.
- [7] Ortega G.Linux for the International SpaceStation Programe[J].Linux Journal,1999,1999(59es):8.
- [8] 王国伟.多内核操作系统文件系统的设计开发与性能研究[D].西北师范大学,2016.
- [9] 续继俊. 3种FAT格式中簇链地址的计算方法研究[J].科技情报开发与经济,2010,20(13):100-102.
- [10] 杨虹.80386保护模式下的编程[J].电脑开发与应用,2001,14(3):12-14.
- [11] 李忠,王晓波,余洁.x86汇编语言:从实模式到保护模式[M].电子工业出版社,2013.
- [12] Accetta,M,R.Baron, W.Bolosky, D.Golub, R.Rashid, A.Tevanian, M.Young. Mach: a new kernel foundation for Unix development [A]. 1986 summer usenix conference, July 1986.
- [13] 毛德操,胡希明.Linux内核情景分析[M].杭州:浙江大学出版社,2000.
- [14] 陈榕.因特网时代操作系统的演变[J].计算机世界,No.42, 2001,10.
- [15] 熊安萍,蒋溢.浅析操作系统内核体系结构[J].重庆工商大学学报(自然科学版),2004,22-26.
- [16] 孙艳, 叶梅, 赵京伟. LINUX操作系统内核关键技术剖析与设备驱动程序的设计实现[C]. 第10届全国核电子学与核探测技术学术年会论文集.2000, 425-432.
- [17] IA-32 Intel Architecture®Software Developer's Manual Volume 1:Basic architecture[M], Oder number 253665.
- [18] Courtney M P. Operating system services for wide area appliacations[EB/OL]. <http://www.cs.duke.edu/ari/issg/webos>, 2001-04-12/2004-06-12.
- [19] 郑钢. 操作系统真象还原[M]. 北京:人民邮电出版社, 2016, 3.
- [20] Dibora C, S.Ockman and M.Stone. Open Sources voices from the Open Source Revolution[M]. O'REILLY, 1999.
- [21] Maurice J.Bach著, 陈葆钰, 王旭, 柳存录, 冯雪山. UNIX操作系统设计[M]. 北京:机械工业出版社, 2000.
- [22] 于渊. 自己动手写操作系统[M]. 北京:电子工业出版社, 2005, 8.
- [23] Engler, D.R., M.F.Kaashoek, J.O'Toole Jr.Exokernel:An Operating System Architecture for Applicationg-specific Resource management[A]. In Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles, December 1995.
- [24] David B.Stewart. Desinging Software Componets for Real-Time Applications, 2001 Embedded Systems Conference San Francisco[C], CA.April 2001.
- [25] Edward A.Lee, Embedded Software-An Agenda for Research[C], USB ERL Memorandum M99/63. December 15, 1999.
- [26] 赵红武, 金瑜, 刘云生. 一种减少中断延迟的中断调度模型[J]. 现代计算机. 2005(09), 104-107.
- [27] 武军彪. 嵌入式操作系统核间任务同步与通信的设计与实现[D]. 西安电子科技大学, 2012.
- [28] 闫茂德, 贺昱曜, 陈金平, 许化龙. 嵌入式实时操作系统内核的设计与实现[J]. 长安大学学报(自然科学版), 2004, 38-39.
- [29] Michael Elizabeth Chastain,Linux Kernel Config Language Specification[M], 18 October 1999 .
- [30] Dejan Milojicic. Embeded systems.IEEE Concurrency[C]. October-December 2000.

- [31] 杨季文等编著. 80x86汇编语言程序设计教程[M]. 青海大学出版社.
- [32] 面向计算系统的虚拟化技术[J]. 金海,廖小飞. 中国基础科学. 2008(06),78-79.
- [33] 程学先. 林珊. 汇编语言程序设计[M]. 北京:机械工业出版社, 2009.
- [34] 沈美明, 温冬婵编著. IBM-PC 汇编语言程序设计[M]. 青海大学出版社, 1996.
- [35] 陈莉君. Linux内核的分析及应用[J]. 西安邮电学院学报, 2001, 6(1):17-20.
- [36] 吕腾飞. 基于Linux内核页表构建内核隔离空间的研究及实现[D]. 南京大学, 2017.
- [37] 潘清. 操作系统研究回顾与展望[J]. 指挥技术学院学报, 2001, 12(1):1-4.
- [38] 顾宝刚. 基于VxWorks的异构多核处理器软件系统的研究与设计[D]. 国防科学技术大学, 2008.
- [39] 赵炯. Linux内核完全注释[M]. 机械工业出版社, 2004.
- [40] 赵炯. Linux内核完全剖析[M]. 机械工业出版社, 2006.
- [41] 池志雄. 多内核操作系统资源管理设计与实现[D]. 中国科学院大学(中国科学院工程管理与信息技术学院), 2017.
- [42] 汤元斌. 多线程模拟进程时间片轮转调度算法研究[J]. 四川文理学院院报, 2014,28-29.
- [43] 刘洪, 刘欣. 基于多线程的进程调度演进过程的仿真设计[J]. 信息通信, 2014, 182-185.
- [44] Andrew S.Tanenbaum著. 陈渝等译. 操作系统设计与实现[M]. 北京:电子工业出版社, 2007.
- [45] Andrew S.Tanenbaum著. 陈向群, 马洪兵译.现代操作系统[M]. 北京:机械工业出版社, 2005, 6.
- [46] IA-32 Intel Architecture®Software Developer's Manual Volume 3: System Programming Guide. 2004.
- [47] Richard B, 马朝辉. 汇编语言程序设计[M]. 北京:机械工业出版社, 2005.
- [48] 陈向群, 杨芙清编著. 操作系统教程[M]. 北京大学出版社. 2006.
- [49] 唐宁九编著. 8086/8088汇编语言程序设计[M]. 四川大学出版社, 2008.
- [50] Danial P.B. 深入理解Linux内核[M]. 电力出版社, 2001.
- [51] 许斌. 基于DM3730异构多核处理器的嵌入式操作系统设计与实现[D]. 电子科技大学, 2013.
- [52] 刘俊. 基于Windows7的维文多语种操作系统本地化的研究与实现[D]. 新疆大学, 2012.
- [53] 王明. 面向小文件的分布式文件存储管理系统的设计与实现[D]. 北京邮电大学, 2018.
- [54] Liu J, Department N. Discussing Several Problems about 80×86 Assembly Language teaching based on Windows System[J]. Computer Knowledge&Technology, 2016:257-262.
- [55] Nollat , Mignolet J Y, Bartic T A, et al. Hierarchical Run-Time Reconfiguration Managed by an Operating System for International Conference on Engineering Reconfigurable Systems&Algorithms[C]. Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, January 2003.
- [56] 卢军, 曾茂城. Linux体系与编程:基于Linux0.01版本[M]. 水利水电出版社, 2010.
- [57] 卢军. Linux0.01内核分析与操作系统设计[M]. 清华大学出版社, 2004.
- [58] 张浩. 嵌入式操作系统中设备管理和驱动程序的开发[J]. 计算机光盘软件与应用, 2013(1):251-252.
- [59] 张明, 张正兰.MS—DOS操作系统的设备管理策略[J]. 计算机应用, 1992(5):33-34.



## 附录 1 攻读硕士学位期间撰写的论文

(1) 冯小建, 马明栋, 王得玉. 基于改进的 Adaboost 算法的人脸检测系统[J]. 计算机技术与发展, 2019,29(03):89-92。

## 致谢

一年以来，从论文开题到最终定稿，经历过潜心学习与实践的不断交织，也经历过后期对文章反反复复的修改。如今论文即将完成，内心颇有感慨。

本文的研究工作是在导师马明栋教授的精心指导和悉心关怀下完成的。自三年前进入学校以来，马老师一直耐心引导着我的学习与工作。导师孜孜不倦的育人精神、渊博的学识、丰富的人生经验和无私的奉献精神深深地影响着我、启迪着我，令我终生受益。三年来，导师给我们上课的情景历历在目，导师的敦敦教诲犹在耳旁。马老师不仅为我打下了扎实的专业基础，也培养了我较强的实践能力，更是传授了我宝贵的人生经验和做人的道理，让我在人生道路上变得更加自信和勇敢。在即将毕业之际，谨向马老师致以最诚挚的感谢，感谢三年来给予我耐心的指导，使我在学术研究和实践工作上取得迅速的进步。

在日常学习和生活中，韩贝、戴伟、高云云、刘培军和任毅等同学给予了我很大帮助。三年来，我们互相鼓励，共同前进，从他们身上我获得了许多宝贵的经验，使我受益匪浅。同时也感谢所有关心我的亲人朋友，感谢他们多年来的帮助和在我迷茫时的鼓舞，没有他们就没有我今天的成绩。

最后，由衷地感谢在百忙之中评阅论文和参加答辩的老师教授们，感谢老师们的指导意见！