

2025 届本科生学士学位论文

学校代码: 10269



華東師範大學
East China Normal University

本科生毕业论文

从零开始的小型操作系统内核编写 与实验设计

Development of an OS Kernel from Scratch and Design of Experiments

姓 名: 夏浩东

学 号: 10215102452

学 院: 计算机科学与技术

专 业: 计算机科学与技术

指导教师: 石亮

职 称: 教授

2025 年 4 月

华东师范大学学位论文诚信承诺

本毕业论文是本人在导师指导下独立完成的，内容真实、可靠。本人在撰写毕业论文过程中不存在请人代写、抄袭或者剽窃他人作品、伪造或者篡改数据以及其他学位论文作假行为。

本人清楚知道学位论文作假行为将会导致行为人受到不授予/撤销学位、开除学籍等处理（处分）决定。本人如果被查证在撰写本毕业论文过程中存在学位论文作假行为，愿意接受学校依法作出的处理（处分）决定。

承诺人签名：

日期： 年 月 日

华东师范大学学位论文使用授权说明

本论文的研究成果归华东师范大学所有，本论文的研究内容不得以其它单位的名义发表。本学位论文作者和指导教师完全了解华东师范大学有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅；本人授权华东师范大学可以将论文的全部或部分内容编入有关数据库进行检索、交流，可以采用影印、缩印或其他复制手段保存论文和汇编本学位论文。

保密的毕业论文（设计）在解密后应遵守此规定。

作者签名：

导师签名：

日期： 年 月 日

目录

摘要:	I
ABSTRACT:	II
1、绪论	1
1.1 背景	1
1.2 相关工作	4
1.3 论文组织	6
2、内核的整体架构与模块设计	8
2.1 整体架构	8
2.2 机器启动模块	9
2.3 内存管理模块	9
2.4 中断异常模块	11
2.5 进程管理模块	12
2.6 文件系统模块	14
2.7 小结	17
3、内核的实现过程与实验指导	18
3.1 内核实现思路	18
3.2 第一阶段：无进程的内核	19
3.3 第二阶段：无持久化能力的内核	21
3.4 第三阶段：相对完善的内核	24
3.5 实验设计与使用情况	25
3.6 小结	26
4、内核中高级机制的设计实现	27
4.1 概述	27
4.2 伙伴系统	27
4.3 文件映射机制	29
4.4 多级反馈队列调度算法	31
4.5 小结	33
5、系统测试	34
5.1 过程性测试	34
5.2 系统调用测试	35
5.3 高级机制测试	37
5.4 小结	40
6、总结与展望	41
6.1 工作总结	41
6.2 未来展望	41
参考文献	42
附录	44
致谢	45

从零开始的小型操作系统内核编写与实验设计

摘要:

在计算机科学领域，操作系统作为连接用户软件与底层硬件的重要桥梁，其设计与实现始终是学术界和产业界的重点研究方向。然而，大多数研究聚焦于优化现有的操作系统，而较少关注操作系统最初是如何建立起来的。

本文以麻省理工学院的教学操作系统 xv6 和产业界的开源操作系统 Linux 为参考，基于 QEMU 硬件模拟平台和 RISC-V 体系结构，从零开始设计实现了一个小型操作系统内核，涵盖内存管理、文件系统、进程管理、中断异常处理等操作系统核心模块。内核开发过程遵循“从零开始，逐步演进”的原则，形成了 9 个中间版本；在此基础上，本文提出了一套包含 9 个实验的完整实验方案，引导初学者按照学习路径从零开始逐步构建简单内核，助力操作系统课程改革。最后，在所实现的基础版本内核上，本文增加了若干 Linux 中的高级机制与实用特性，包括伙伴系统、文件映射、多级反馈队列调度算法，形成进阶版本内核。

总而言之，在工程实践方面，本文提供了从零开始编写小型操作系统内核的清晰思路；在课程教学方面，本文为操作系统课程实验提供了创新性的实施方案；在前沿探索方面，本文在小型内核上实现了 Linux 中的先进技术，深入分析了背后的设计原理与运行机制。

关键词：操作系统内核，xv6，Linux，从零开始，实验设计

Development of an OS Kernel from Scratch and Design of Experiments

Abstract:

In the field of computer science, operating systems serve as a critical bridge connecting user-level software with underlying hardware. Their design and implementation have consistently been a focal point of research in both academia and industry. However, the majority of existing studies emphasize the optimization of contemporary operating systems, with comparatively less attention devoted to the foundational process of constructing an operating system from scratch.

This study references the teaching operating system xv6 developed at MIT and the open-source operating system Linux. Utilizing the QEMU hardware simulation platform and the RISC-V architecture, a compact operating system kernel was designed and implemented starting from a minimal foundation. The kernel encompasses essential modules such as memory management, file systems, process management, and interrupt/exception handling. The development process adhered to the principle of incremental evolution, yielding nine intermediate iterations. Building upon this foundation, a comprehensive framework of nine experiments was devised to guide beginners through the construction of a rudimentary kernel step by step, thereby facilitating pedagogical reforms in operating system courses. Furthermore, advanced mechanisms and practical features from Linux, including the buddy system, file mapping, and the multi-level feedback queue scheduling algorithm, were incorporated into the foundational kernel version, resulting in an enhanced, advanced kernel variant.

In summary, this work provides a systematic methodology for constructing a compact operating system kernel from the ground up, contributing to the domain of engineering practice. In terms of educational innovation, it proposes a novel experimental framework tailored for operating system curricula. Finally, in the realm of cutting-edge exploration, it implements and analyzes advanced Linux technologies within a compact kernel, elucidating their underlying design principles and operational mechanisms.

Keywords: Operating System Kernel, xv6, Linux, From Scratch, Experiment Design

1、绪论

1.1 背景

1.1.1 操作系统的定位和重要性

操作系统是连接用户软件和底层硬件的关键桥梁。如图 1-1 所示，系统软件栈的最上层是用户程序，它通过系统调用或库函数（封装后的系统调用）向操作系统发出请求。广义上的操作系统可以大致分为三层：系统调用接口层为用户程序提供标准化的服务接口，所有用户程序的底层逻辑都是若干系统调用请求的叠加；操作系统内核层是操作系统的灵魂所在，包含复杂的模块实现和模块间通信，狭义上的操作系统通常指操作系统内核；设备驱动层主要与硬件设备打交道，为操作系统内核提供硬件服务。操作系统之下是底层硬件设备，包括 CPU、内存、磁盘、网卡、GPU 等，它们负责将上层软件的指令落实到每一块芯片、每一个寄存器、每一个比特。

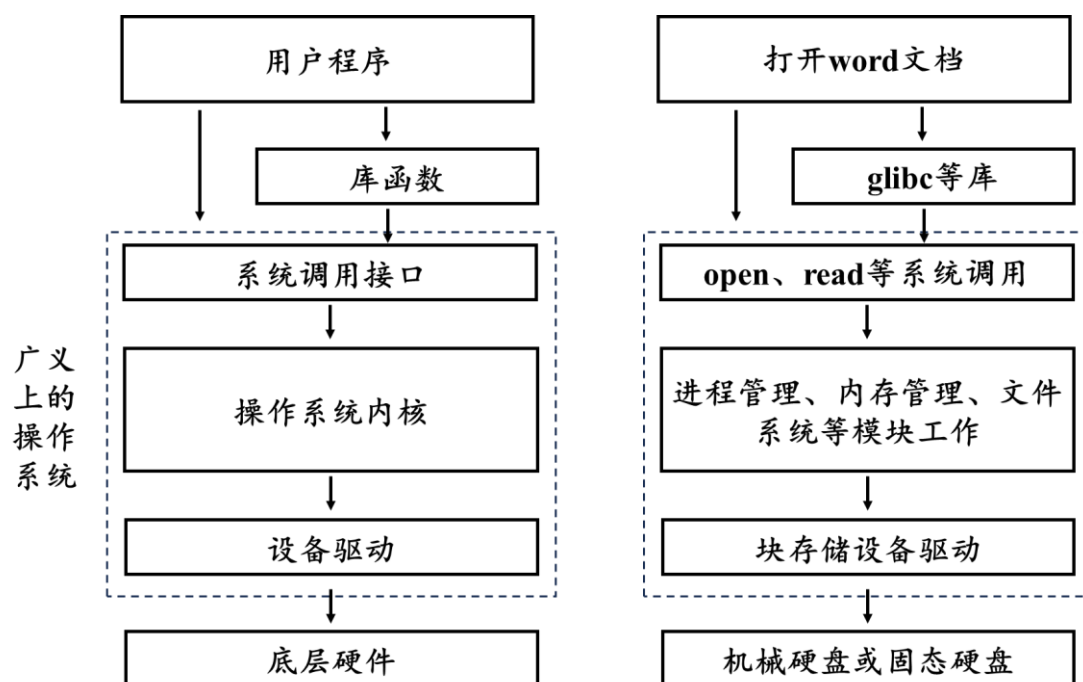


图 1-1 从用户程序到底层硬件的软件栈

Figure 1-1 The Software Stack from User Application to Underlying Hardware

图 1-1 以打开 word 文档为例，对系统软件栈的工作流程做了说明。word 文档程序向系统调用层或 glibc 等函数库发出请求；请求包括 open 系统调用用于打开文件，read 系统调用用于读取文件内容等；系统调用层调用内核提供的服务，服务由进程管理、内存管理、文件系统等多个模块协同提供；内核将上层请求下发到块设备驱动层，并最终作用于机械硬盘或固态硬盘。操作系统的桥梁定位使得上层软件可以忽略底层硬件的实现细节，通过简单的系统调用达成目的。

由上文的例子可以发现，操作系统的重要性体现在两个方面：一方面，操作系统为上层应用程序提供可靠便捷的服务。由于操作系统在软件栈中的桥梁定位，上层的应用程序和库函数往往依赖特定操作系统提供系统调用服务。如果操作系统在性能和稳定性方面存在缺陷，建立在操作系统上的软件生态就不能长远发展。另一方面，操作系统管理和分配底层的硬件资源。现代计算机通常是多核心多进程并行工作，同一时刻有成百上千的用户程序在同一台计算机中工作。然而，计算机无法让所有用户程序同时获取足量的 CPU 资源、内存资源、I/O 带宽等，于是需要操作系统通过资源调度的方式保证用户程序按需、按序获取和释放硬件资源，保证系统整体的高效和公平。

1.1.2 产业界操作系统概况

从操作系统的应用场景来看，产业界操作系统大致可以分为三类：第一类是服务器操作系统，大多数市场份额都被 Linux 各种发行版占据，如 Ubuntu、CentOS、Red Hat 等；第二类是 PC 端操作系统，大多数市场份额都被 Windows、MacOS 占据；第三类是移动端操作系统，大多数市场份额都被 Android、iOS、HarmonyOS 占据。

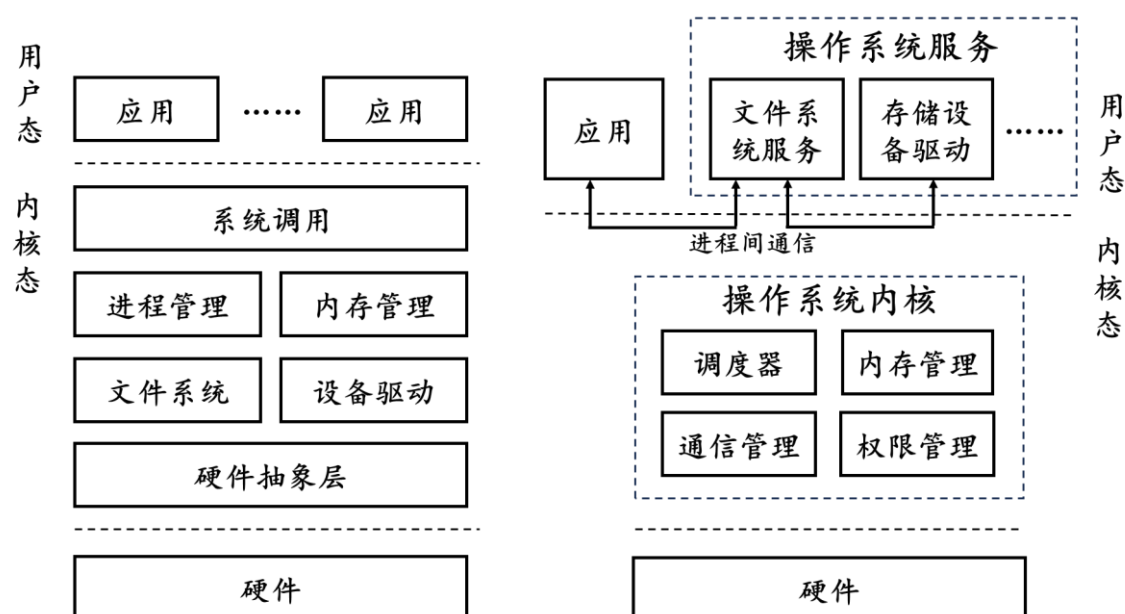


图 1-2 宏内核架构（左）与微内核架构（右）对比

Figure 1-2 Comparison of Monolithic Kernel Architecture (Left) and Microkernel Architecture (Right)

图 1-2 显示了典型的宏内核和微内核操作系统架构。目前主流的操作系统通常采用宏内核架构，主要特征是操作系统内核的所有功能模块都运行在内核态，具备直接操作硬件的能力。宏内核操作系统虽然理论性能很高，但由于大多数内核模块都工作在内核态，在编码复杂性、可靠性、可扩展性等方面存在不足。近些年来，随着分布式技术的发展，微内核操作系统^{[1][2]}逐渐崛起。微内核操作系统将操作系统的某些功

能或模块（如文件系统、设备驱动等）从内核中拆分出来，作为“服务”部署到用户态环境中，内核中只保留必要模块和通信能力。微内核在模块化、可靠性、可扩展性方面具备明显优势，但在性能方面可能弱于宏内核。

1.1.3 教学性操作系统概况

“操作系统”作为计算机专业课程体系中必不可少的核心课程，强调编写和调试内核级代码的能力。为帮助学生理解操作系统基本原理，国内外高校通常不推荐学生直接阅读和修改产业界的操作系统，而是基于小型的教学性操作系统开展实验^[3]。

国内外高校设计实现了许多成熟的教学性操作系统。国内著名的教学性操作系统包括：（1）清华大学开发的 uCore 和 rCore：前者基于 x86 体系结构，后者是基于 RISC-V 体系结构。（2）上海交通大学开发的 ChCore：这是一个基于 ARM 体系结构的微内核操作系统。（3）西北工业大学开发的 NPUCore：这是一个基于国产指令集体系结构 LoongArch 的操作系统。国外著名的教学性操作系统包括：（1）美国麻省理工学院开发的 xv6：基于 x86 和 RISC-V 体系机构，简洁清晰，容易理解。（2）荷兰阿姆斯特丹自由大学开发的 Minix：它是一个简化版的 Unix 操作系统，对 Linux 的诞生产生了重要影响。（3）Linus Torvalds 开发的 Linux（0.11 版本）：奠定了如今 Linux 的基础，实现了操作系统的核心功能。

这些教学性操作系统背后的课程实验从思路大致分为两类：（1）给学生一个可以运行的完整内核，要求学生增加新功能或改进现有功能（以 xv6 的实验为代表）。

（2）从一个可以运行的完整内核中删去一部分关键代码，要求学生按照实验指导补全并通过测试（以 ChCore 的实验为代表）。

本文提出了第三种操作系统课程实验思路：引导学生从零开始，按照操作系统的构建逻辑，逐步搭建自己的操作系统内核。前两种实验思路里，学生大约只需要编写 5%到 10%的代码；而本文提出的实验思路里，超过 80%的代码由学生自主编写。本文提出的实验思路更适合培养和选拔对操作系统内核有浓厚兴趣，且愿意动手实践的综合型创新型人才。

1.1.4 RISC-V 体系结构简介

本文开发的操作系统内核基于 RISC-V 体系结构。RISC-V 体系结构由美国加州伯克利分校的研究人员开发，是一种开源精简指令集计算机（Reduced Instruction Set Computer, RISC）体系结构。操作系统的指令经过编译器转换成基于 RISC-V 的汇编指令，再翻译成底层硬件可以识别的机器码。

RISC-V 体系结构的优点包括：（1）模块化设计：可以根据需求灵活选择指令集大小，基础整数指令集（RV64I）提供最基本功能，其他的可选扩展模块（如 M/A/F 等）用于支持乘除法、原子操作、浮点运算等。（2）简洁灵活：相比复杂指令集计算机（Complex Instruction Set Computer, CISC），RISC-V 体系结构的指令复杂性更低，通过多个简单指令的组合来实现复杂功能。（3）发展迅速、应用广泛：RISC-V 体系结构凭借开放性和灵活性，越来越多应用于嵌入式、物联网、人工智能等领域。

1.1.5 QEMU 软件模拟器简介

本文开发的操作系统内核基于 QEMU（Quick EMUlator）平台运行。QEMU 是一个著名的开源的硬件虚拟化软件，能用软件模拟的方法支持不同体系结构的硬件环境。QEMU 能对硬件环境进行全系统的模拟，包括处理器、内存、硬盘、网卡等，非常适合作为教学性操作系统的运行平台。本文采用的典型配置是：使用 QEMU 模拟一台基于 RISC-V 的小型计算机（开发板），配备 3 个 CPU 核心、128MB 内存、以及必要的外设（如磁盘、串口、时钟等）。通过简单的参数配置，QEMU 能达到与 RISC-V 开发板相近的使用体验，并且更加利于系统测试和 Debug。

1.2 相关工作

1.2.1 xv6 教学操作系统

xv6 是一个使用 C 语言编写的、简单的、类 Unix 的、宏内核架构的教学性操作系统^[4]，支持 x86 和 RISC-V 两种体系结构^[5]，其中 xv6-riscv 版本是本文设计实现的内核的主要参考。

表 1-1 列出了 xv6 核心模块对应的源文件，包括 kernel 目录下的内核态代码、user 目录下的用户态代码、mkfs 目录下的磁盘初始化代码、Makefile 文件。xv6 的内核模块包括机器启动模块、内存管理模块、进程管理模块、文件系统模块、中断异常模块、设备驱动模块、系统调用模块等，代码量大约 4500 行（包括 kernel 目录下的 C 语言源代码文件、头文件、汇编文件）。除了内核模块，xv6 还有较为完善的用户态支持，主要包括两部分：第一部分是 shell 命令行，用户可以输入 Linux 常用命令（如 ls、cat、echo、mkdir 等），xv6 的 shell 会像 Linux 的 shell 一样做出响应；第二部分是内核测试工具，xv6 实现了完善的用户态测试工具，既有针对单个系统调用的正确性测试，也有针对内核整体的可靠性测试和压力测试。除了内核模块和用户支持这两个核心部分，xv6 的 mkfs/mkfs.c 和 Makfile.c 也值得注意，前者用于格式化磁盘映像文件 fs.img 作为文件系统的初始状态，后者用于便捷地编译和运行 xv6。

表 1-1 xv6 中各模块对应的源文件分析

Table 1-1 Analysis of Source Files Corresponding to Each Modules in xv6

源文件	主要作用
kernel/ entry.S start.c main.c	机器启动、设置寄存器环境、各个模块初始化
kernel/ uart.c printf.c console.c	与输入输出相关，逐级包装的关系
kernel/ kalloc.c vm.c	物理内存管理与虚拟内存管理
kernel/ proc.c exec.c	进程生命周期管理、进程调度、程序加载和执行
kernel/ spinlock.c sleeplock.c	并发控制：自旋锁和睡眠锁
kernel/ plic.c trap.c	中断和异常处理
kernel/ virtio_disk.c bio.c log.c fs.c file.c pipe.c	自底向上的文件系统层次
kernel/ sysfile.c sysproc.c	操作系统向用户提供的系统调用接口
user/ cat.c echo.c grep.c kill.c ln.c ls.c mkdir.c rm.c wc.c zombie.c	用户可以通过 shell 执行的命令
user/ forktest.c mmaptest.c grind.c stressfs.c usertest.c	内核的功能和性能测试程序
user/ initcode.S init.c sh.c ulib.c umalloc.c printf.c	用户态启动、shell、库函数
mkfs/ mkfs.c	构造初始状态的磁盘
Makefile	快捷编译和运行内核

1.2.2 Linux 开源操作系统

Linux（通常被称为 GNU/Linux）是世界范围内最著名的开源操作系统^{[6][7]}，与 xv6 一样，它也是类 Unix 的宏内核操作系统。Linux 由芬兰程序员 Linus Torvalds 于 1991 年首次发布，目前由 Linux 开发者社区共同维护。Linux 内核最新版本的代码量已经突破 4000 万行，其中约 70%的代码是驱动程序，约 15%的代码与体系结构相关，剩余约 15%的代码构成操作系统核心模块。基于开源的 Linux 内核，Ubuntu、Debian、CentOS、Arch Linux 等 Linux 发行版已在不同领域形成各自生态。

凭借开源、成本低、性能强、可靠性强的优势，Linux 内核广泛应用于超级计算

机、高性能服务器、移动设备、嵌入式系统等^[8]。然而，Linux 内核代码横跨数十年，由上万名程序员协同开发，机制复杂且体量巨大，不适合直接阅读和学习^{[9][10]}。更好的方式是，通过阅读文档和部分关键代码理解 Linux 中的某些高级机制，尝试移植到一个简单的小型内核上，在代码实现的过程中加深理解，这也是本文的工作之一。

1.3 论文组织

1.3.1 论文的主要工作

本文的主要工作分为三部分：（1）从零开始，设计并实现了一个与 xv6 规模相当的小型操作系统内核，能提供核心的系统调用服务。（2）将小型内核实现历经的 9 个阶段整理形成 9 次实验，在线上仓库中提供基础代码、测试用例和实验指导书，引导学生动手实现一个小型操作系统内核。（3）基于实现的小型操作系统内核，设计并实现一些 Linux 中的高级机制和特性，学习产业界的前沿设计思想。

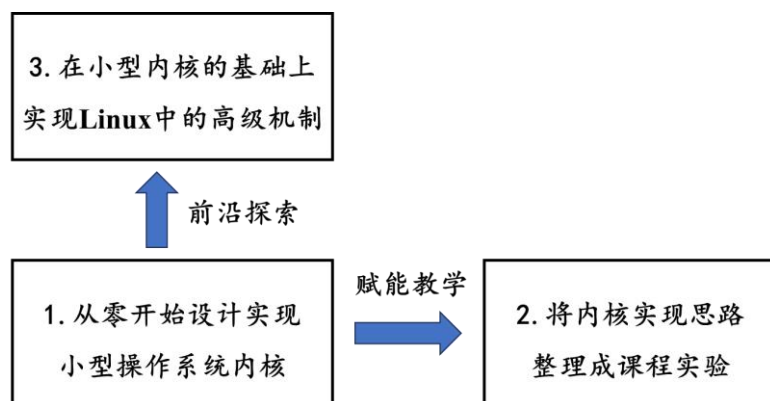


图 1-3 三个工作之间的逻辑关系

Figure 1-3 Logical Relationships Among the Three Tasks

如图 1-3 所示，本文的三个工作具备以下逻辑关系：工作 1（小型内核实现）是工作 2 和工作 3 的基石；在工作 1 的基础上，出于赋能教学的目的产生了工作 2（内核实验设计）；出于前沿探索的目的产生了工作 3（高级机制实现）。

1.3.2 论文的组织结构

如图 1-4 所示，本文包括 6 个章节，每个章节的主题如下：

- 第一章 绪论：介绍问题背景和主要工作。
- 第二章 内核的整体架构与模块设计：介绍工作 1（小型内核实现）。
- 第三章 内核的实现过程与实验指导：介绍工作 2（内核实验设计）。
- 第四章 内核中高级机制的设计实现：介绍工作 3（高级机制实现）。
- 第五章 系统测试：对三个工作进行全面系统的测试评估。

- 第六章 总结和展望：总结工作成果，展望改进方向。

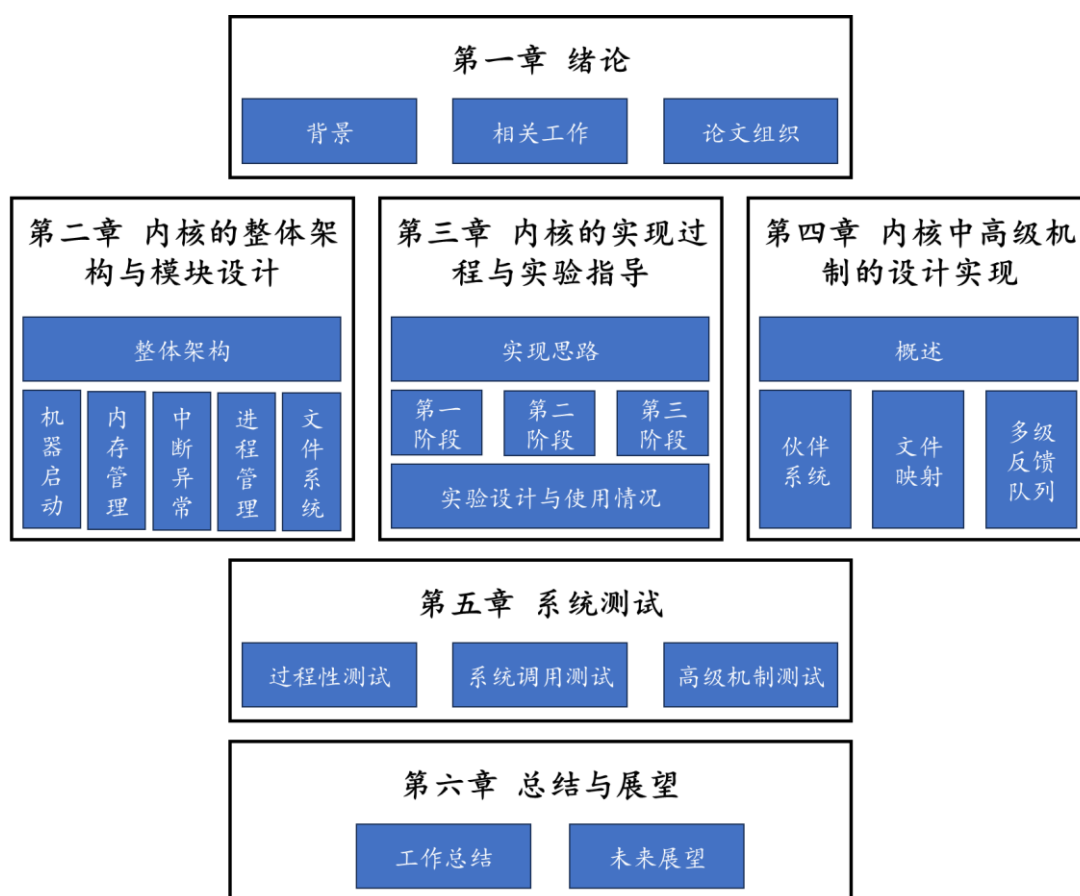


图 1-4 本文各章节组织结构

Figure 1-4 Organizational Structure of Each Chapter in this Paper

2、内核的整体架构与模块设计

2.1 整体架构

如图 2-1 所示，本文实现的内核在架构上分为三部分：用户态测试程序、内核态功能模块、QEMU 模拟的硬件环境。RISC-V 体系结构规定了三种特权模式：用户模式 (User mode, U-mode)、监管者模式 (Supervisor mode, S-mode)、机器模式 (Machine mode, M-mode)。图中黄色部分 (用户态测试程序 `initcode`、`test`、`ulib`) 工作在 U-mode，只有执行少部分指令的权限；蓝色部分 (除去机器启动的其他内核功能模块) 工作在 S-mode，拥有执行大部分指令的权限；绿色部分 (机器启动模块和 QEMU) 工作在 M-mode，拥有执行全部指令的权限。

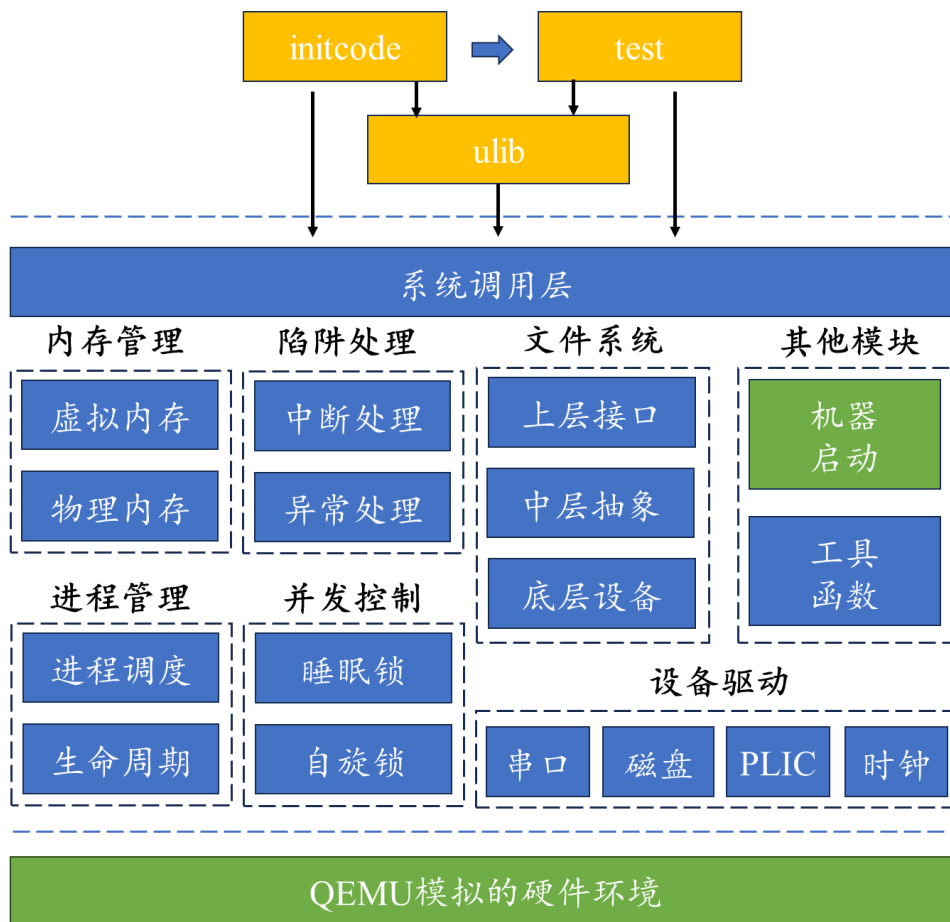


图 2-1 内核整体架构图

Figure 2-1 Overall Architecture of the Kernel

后面的小节将首先介绍机器启动模块 (2.2)，分析操作系统从上电到完成初始化的过程；随后分别介绍四个核心模块，包括内存管理 (2.3)、中断异常 (2.4)、进程管理 (2.5)、文件系统 (2.6)，它们分别管理计算机中三个核心组件：内存 (Memory)、CPU (Central Processing Unit)、磁盘 (Disk)。并发控制、设备驱动、工具函数等模

块，由于它们设计简单且远离操作系统核心功能，本文不做详细介绍。

2.2 机器启动模块

机器启动过程包括三个阶段：QEMU 阶段、准备阶段、初始化阶段。

在 QEMU 阶段，QEMU 为操作系统模拟出如下硬件环境：3 个 CPU 核心、128MB 内存、2GB 磁盘空间（包括一个初始的文件系统磁盘映像）、其他必要外设。此外，QEMU 还准备了如下软件环境：特权状态设为 M-mode，CPU 中的 PC 寄存器设为 0x80000000，内存中 0x80000000 处存储着第一个操作系统函数_entry。随着时钟周期的律动，内核的第一行汇编代码即将执行。

准备阶段分为两个部分：第一部分是 C 语言程序环境设置（主要在 entry.S 中），内核读取 CPU 序号并为各个 CPU 设置栈寄存器，保证 C 代码能正常执行。第二部分是 M-mode 初始化（主要在 start.c 中），包括禁用分页机制、设置 trap 相关寄存器、开启时钟中断、最后进入 S-mode 的 main 函数。

初始化阶段（主要在 main.c 中）已经从 M-mode 进入 S-mode，操作系统大多数模块都运行在 S-mode。初始化阶段的工作分为两部分：第一部分是调用各个模块的初始化函数，包括输入输出模块、物理内存模块、虚拟内存模块、进程模块、陷阱模块等；第二部分是启动进程调度器，进入第一个用户进程的控制流。

经过上述三个过程，内核完成了上电和自身初始化，从绝对的主导者转变为用户进程的服务者和管理者，开始履行管理硬件资源和服务上层应用的职责。

2.3 内存管理模块

内存管理分为物理内存管理和虚拟内存管理两个部分，虚拟内存管理建立在物理内存管理的基础上。物理内存管理的核心是物理页的链式组织，虚拟内存管理的核心是页表映射。

2.3.1 物理内存管理

如图 2-2 所示，物理内存的原始状态是 128MB 的连续地址空间（0x80000000 ~ 0x88000000）。为了便于管理和使用，先将内存区域切分成若干分立的 4KB 物理页；随后将物理页分为两部分，一部分供内核空间使用，一部分供用户空间使用，可以采用位示图（bitmap）管理；但是为了优化物理页分配和回收的效率，本文选择用两个独立的链表管理。对于物理页分配操作（pmem_alloc），只需从对应链表的头部（list_head->next）删除一个结点；对于物理页回收操作（pmem_free），只需传入物理页地址，找到物理地址对应的节点，并把它插入目标链表头部即可。

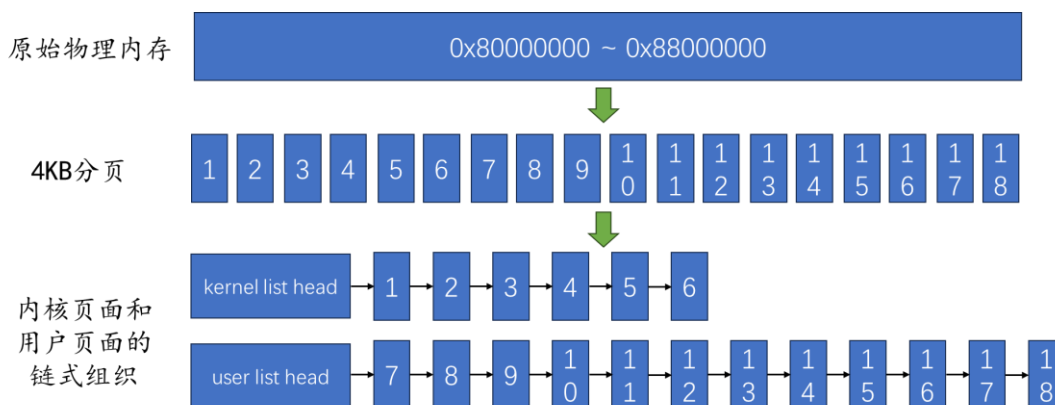


图 2-2 物理内存管理方式

Figure 2-2 Physical Memory Management Methods

值得注意的是，这种物理内存管理方式存在一个明显的缺点：不适合分配多个连续物理页。这种需求通常来自用户空间，比如用户申请 128 个连续的 4KB 物理页用于存储一个长数组。这个问题将在“4.2 伙伴系统”章节中得到解决。

2.3.2 虚拟内存管理

在详细介绍虚拟内存管理策略前，需要对“虚拟内存”这个概念做一个区分：虚拟内存有时侧重于指代“利用外存空间(如磁盘)作为逻辑上的内存空间以扩展内存”的做法；有时侧重于指代“使用页表隔离进程所见的逻辑地址空间与物理地址空间”的做法；本文所说的虚拟内存是后者。

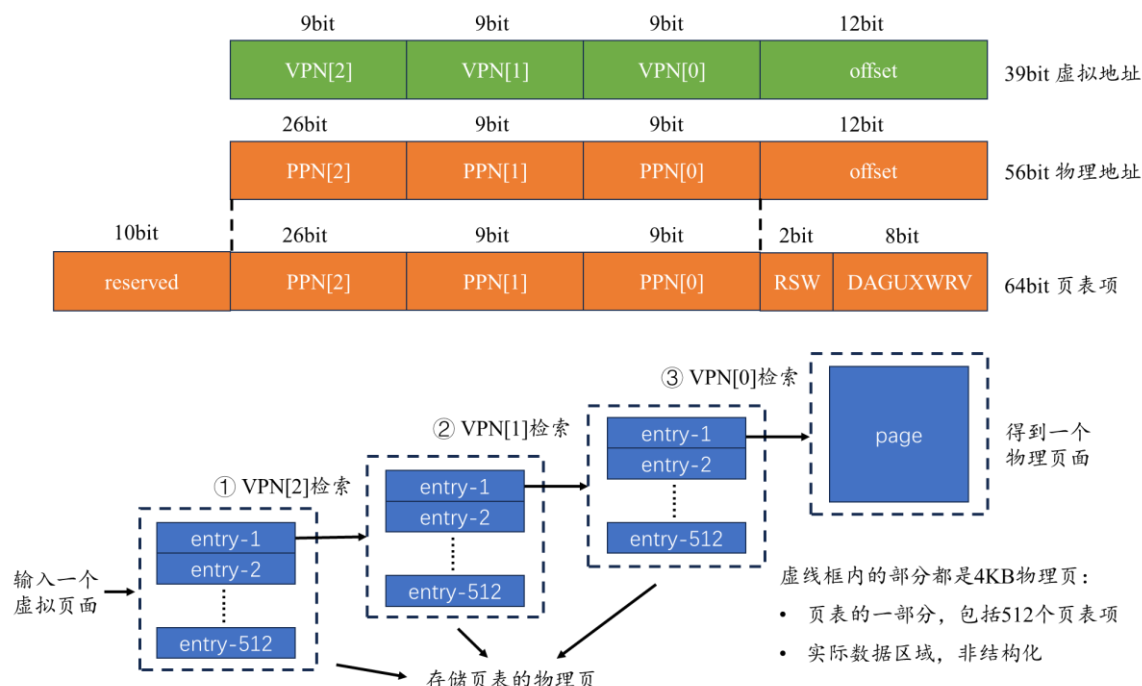


图 2-3 虚拟内存的页表映射机制

Figure 2-3 Paging Mechanism of Virtual Memory

之所以要引入虚拟内存，是为了解决多进程共享地址空间面临的问题：假设有两个用户态进程并行执行，且用户态进程都希望使用 $0x0000 \sim 0x1000$ 作为地址空间，由于请求冲突，不可能同时满足它们的需求。为了解决这个问题，可以把 $0x80000000$ 开始的 4KB 内存区域分配给进程 1， $0x80001000$ 开始的 4KB 内存区域分配给进程 2。当用户访问逻辑地址时自动翻译为物理地址，使得物理内存分配过程对用户进程透明。为了记录这种逻辑地址到物理地址的映射关系，需要一种专门的数据结构——页表。

本文采用 RISC-V 的 SV39 标准作为页表组织格式，顾名思义虚拟地址是 39bit 的，其中前 27bit 用于标识虚拟页序号，后 12bit 用于标识页内偏移量。如图 2-3 所示，页表是三级结构，由多个节点构成，每个节点都是一个 4KB 物理页，存储了 512 个页表项。虚拟地址翻译过程如下：在三级页表中找到 $VPN[2]$ 对应的页表项（ $VPN[2]$ 占 9bit，可以标识 512 个页表项中的 1 个）；根据页表项里存储的物理地址信息找到对应的二级页表（一个三级页表最多管理 512 个二级页表）；在这个二级页表里找到 $VPN[1]$ 对应的页表项；以此类推最终找到真正存储数据的物理页。地址翻译过程由硬件自动执行，内核只需将三级页表的地址填写到特定寄存器并填写页表项即可。

2.4 中断异常模块

RISC-V 使用陷阱（trap）的概念统领中断（interrupt）和异常（exception），中断和异常都属于陷阱。从编程角度出发，中断和异常的共同点是会打断当前的执行流，陷入一个新的执行流，处理完成后返回原来的执行流；区别在于中断结束后会执行触发中断的命令的下一条命令（ $PC+4$ ），而异常结束后会重新执行触发异常的命令。

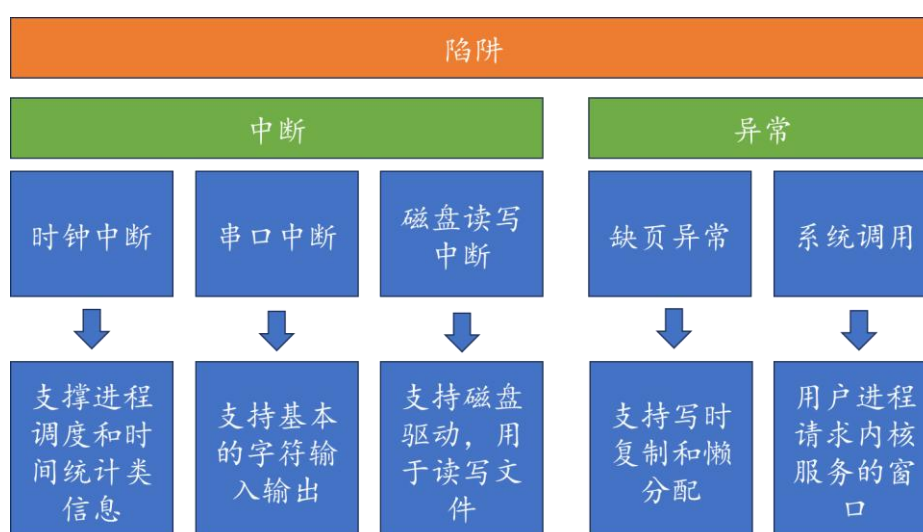


图 2-4 内核中陷阱种类概览

Figure 2-4 Overview of Trap Types within the Kernel

如图 2-4 所示，中断和异常的一个关键作用是将控制流从用户进程手里收回，目的通常有两个：（1）控制资源分配，防止某些进程长期占用 CPU 资源。（2）响应用户请求，如串口读写、文件读写、页表填写、系统调用等。总而言之，中断异常模块是用户程序和内核程序间的“润滑剂”，保障两者各尽其能、有序运行。

2.5 进程管理模块

进程管理模块可以分为两个部分：首先是进程生命周期管理，核心思想是实现一个完善的进程状态机；其次是多进程调度，核心思想是以内核里的调度器为媒介，依据调度算法，让 CPU 控制权在多个进程间有序流转。

2.5.1 进程生命周期

进程即进行中的程序，它由静态的程序指令和动态的程序状态组成。程序指令指的是程序编译后生成的可执行指令；程序状态包括程序使用的寄存器状态、程序管理的内存空间状态、程序申请的操作系统资源等。进程管理的一个重要工作就是让进程从诞生到死亡的全过程有序进行，始终处于操作系统内核的掌控之中。

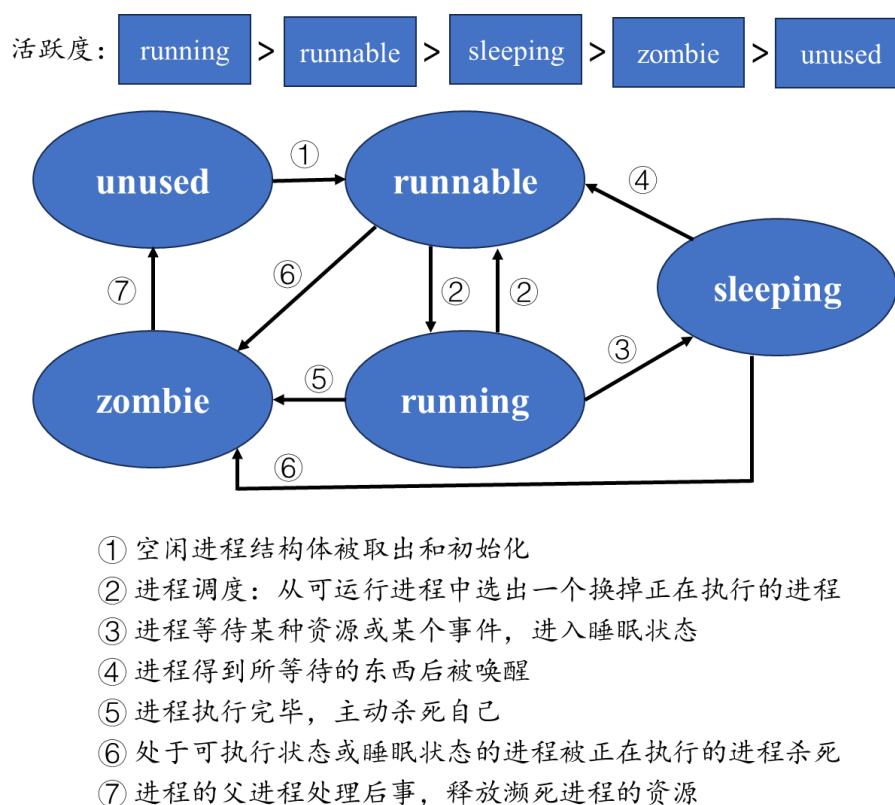


图 2-5 进程的状态机模型

Figure 2-5 State Machine Model of Processes

图 2-5 显示了具备五个状态的进程状态机模型。以在看书为例解释各个状态的含义：图书在一开始都是白纸（unused），经过印刷（初始化）后有了内容；读者取出一

些书放在书桌上，这些书触手可及（runnable），但读者一次只能看一本书（running），有一些不常看的书可以放回书架（sleeping），在需要的时候拿回书桌；有的书看了一遍就没有兴趣了（zombie），过一段时间就会有人把这些书回收变回白纸（unused），以供未来的印刷。进程的生命周期管理与读书的例子类似，构成了包括 5 个状态和 7 个事件的状态机，具体的状态转化过程在图 2-5 中做了说明。

2.5.2 多进程调度

进程状态机中最常发生的事件就是进程调度，即处于 running 状态的进程 A 主动或被迫放弃 CPU 使用权，操作系统从处于 runnable 状态的进程集合中选择一个进程 B 运行。图 2-6 显示了一个简单的进程调度模型：内核初始化完成后开启调度器，进程 A 和进程 B 在调度器的控制下，按序共享 CPU 资源，达到并发作业的目的。其中虚线内的空白区域代表进程此时处于 runnable 状态，有色区域代表处于 running 状态，箭头代表上下文切换过程。

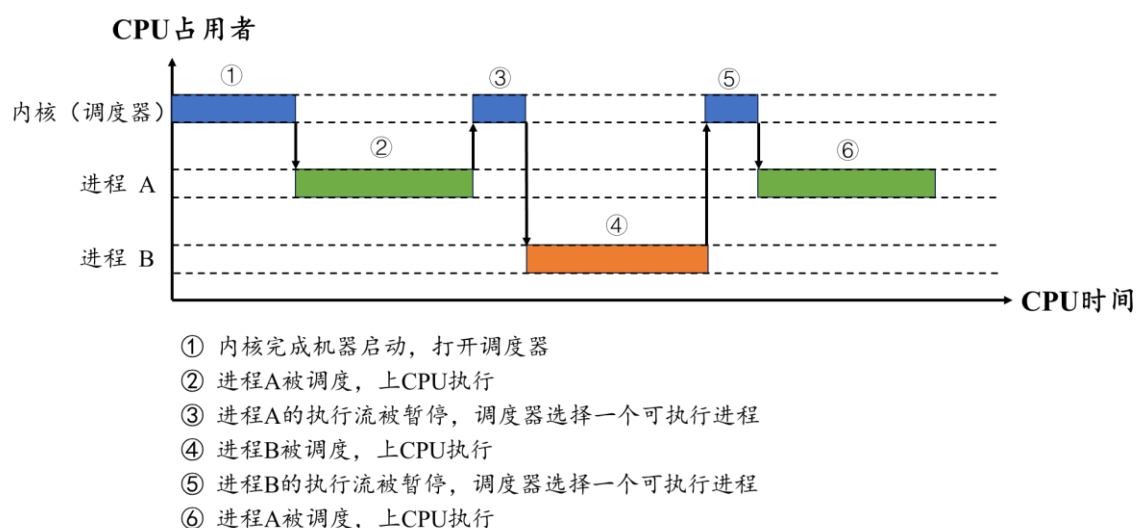


图 2-6 进程调度过程中的控制流切换

Figure 2-6: Control Flow Switching in Process Scheduling

基础版本内核采用的进程调度算法是时间片轮转（Round Robin, RR）。算法思路如下：在初始化时为每个进程配置一个时间片；当某个进程处于 running 状态且发生时钟中断时，这个进程的时间片减 1；当时间片减到 0 时，触发调度，将 CPU 使用权切换到调度器，同时重置该进程的时间片。RR 算法的优点是：（1）简单高效，易于实现；（2）公平性好，不做优先级区分；（3）可以通过时间片的大小控制调度发生的频率，防止调度过于频繁或过于稀少。对于以功能测试为目的的教学性操作系统来说，这种简单且高效的调度算法是比较合适的。

2.6 文件系统模块

文件系统模块^[1]脱离了计算机核心区域的 CPU 调度和内存管理，转而关注磁盘这种非常常见且重要的外设。与前面介绍的模块相比，文件系统模块最大的特点就是清晰的层次化结构，如图 2-7 所示，文本实现的文件系统分为底层设备、中层抽象、上层接口三个层次。

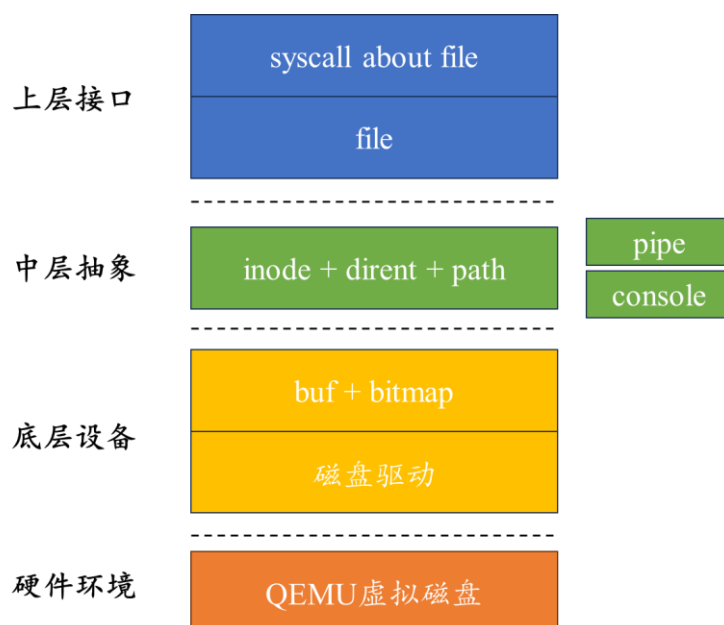


图 2-7 文件系统层次结构

Figure 2-7 File System Hierarchy

2.6.1 底层设备

QEMU 为操作系统内核提供了虚拟磁盘，通过读写特定寄存器，磁盘驱动程序可以为上层提供两个重要的函数接口：（1）以 **block** 为单位将磁盘里的一个区域读取到内存或将内存的一个区域写入磁盘，这个接口提供给文件系统。（2）以磁盘中断的方式将读写操作传递至虚拟磁盘，这个接口提供给中断异常模块。

磁盘驱动程序需要 **buffer** 结构体作为读写的依托，这个结构体必须包括以下信息：（1）被操作的 **block** 序号。（2）**block** 数据在内存中的缓存空间。为了管理这样的 **buffer** 资源，本文将其组织成双向环形链表的格式，将 **buffer** 的申请和释放转换成链表的插入和删除操作。此外，本文采用 LRU 策略和懒惰写回策略优化 **buffer** 与磁盘的协作：将最近经常访问的数据留在内存，而不是立刻更新至磁盘；在 **buffer** 不足时将长时间不访问的数据写回磁盘。

上面提到，内存中使用 **buffer** 管理 **block** 里的数据，对应的，磁盘里使用位示图（**bitmap**）管理 **block**。如图 2-8 所示，磁盘分为五个区域：最前面的 **block** 被称为

superblock，它存储着磁盘和磁盘上文件系统的全局信息；后面的 inode bitmap 用于指示 inode region 里的 inode 是否被分配出去；再后面的 data bitmap 用于指示 data region 里的 block 是否被分配出去。bitmap 的操作主要是查询空闲比特和设置比特值，比特的操作通过位运算实现。

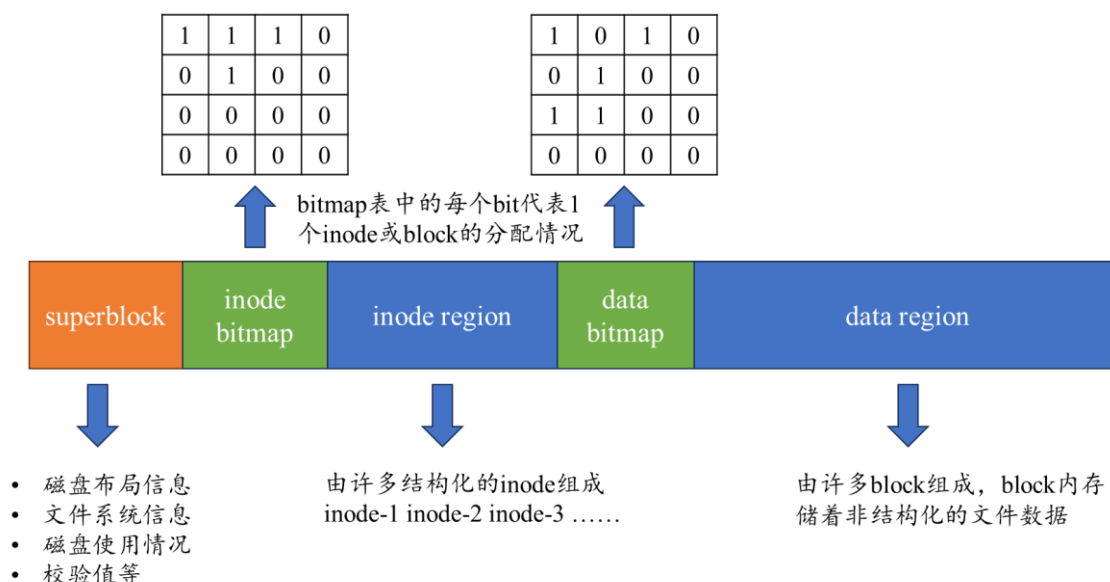


图 2-8 磁盘布局和各区域的作用

Figure 2-8 Disk Layout and the Functions of Each Region

2.6.2 中层抽象

中层抽象是连接上层和底层的关键，是文件系统最核心的部分。中层抽象主要依赖索引节点（index node, inode）、目录项（directory entry, dentry）、文件路径（path）三个重要元素构建。

如图 2-9 所示，inode 主要存储了两类信息：一类是文件信息（如文件名、时间戳、读写权限等），一类是索引信息（用于记录文件内容存储在哪些 block 中）。文件索引的组织结构与前文提到的多级页表类似，思路是用少部分 block 存储索引信息，大部分 block 存储数据。区别在于文件索引采用 $10+2*N+N*N$ 的组织格式，可以很好适应小型文件（10 个 block 足够存储）、中型文件（ $10+2*N$ 个 block 足够存储）和大型文件（ $10+2*N+N*N$ 个 block 足够存储）。目录是一种特殊的结构化文件，一个目录包含多个目录项，目录项用于建立文件名到 inode 序号的映射。

多个目录和文件构成一颗目录树，文件路径指的是从树根到枝叶的路径，如 `/home/xhd/workdir/oslab/hello.c`。路径解析的过程包括两件事：路径的切割（字符串处理和信息提取）和目录访问，目标是拿到路径对应的 inode。读到 inode 后可以通过其

中存储的索引信息查询文件放在哪些 block，最终完成数据读取。

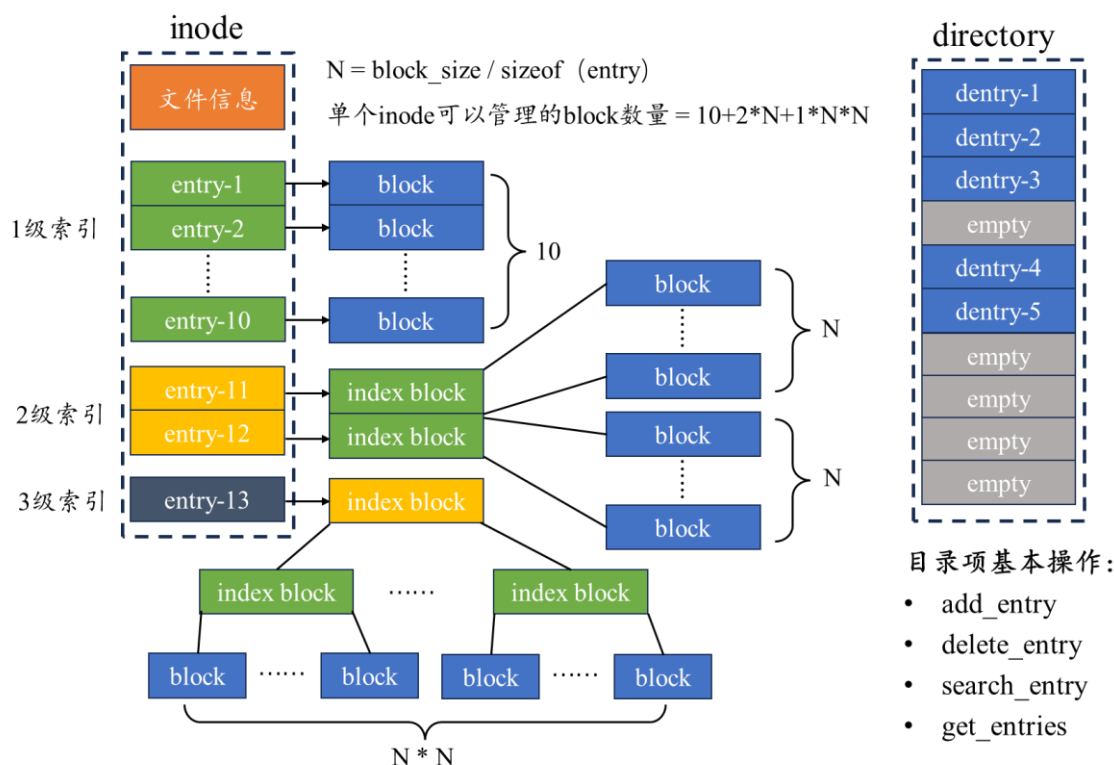


图 2-9 索引节点结构图（左）和目录结构图（右）

Figure 2-9 Index Node Structure (left) and Directory Structure (right)

2.6.3 上层接口

如图 2-10 所示，文件系统的上层秉持“一切皆文件”（Everything is a file）的 UNIX 设计思想，将普通文件（包括可执行文件）、目录文件、设备文件、管道文件这四类常见文件以统一的框架组织起来。这样的好处是，上层的系统调用可以用统一的语义来操作各种各样的文件。

对于文件的共性操作（如 open、read 等），可以用同一个系统调用接口，在接口内部依据文件类型，执行类似 switch-case 的语句，分别调用文件注册的操作函数；对于文件的特性操作（如 mknod、mkdir 等），可以设立不同的系统调用接口，满足文件的私有需求。通过共性接口和特性接口相结合的方式，文件系统在最大化复用现有路径的同时，兼顾各类文件的特殊需求。

以设备文件为例，C 语言编程中常见的函数 printf 和 fprintf 分别用于向 shell 终端和某个打开的文件中输出字符。它们的底层都是 write 系统调用，区别在于前者输出到设备文件并显示在屏幕上，后者输出到普通文件并存储到外存中。通过巧妙复用文件系统的系统调用，避免了为输出设备单独设置接口的麻烦。

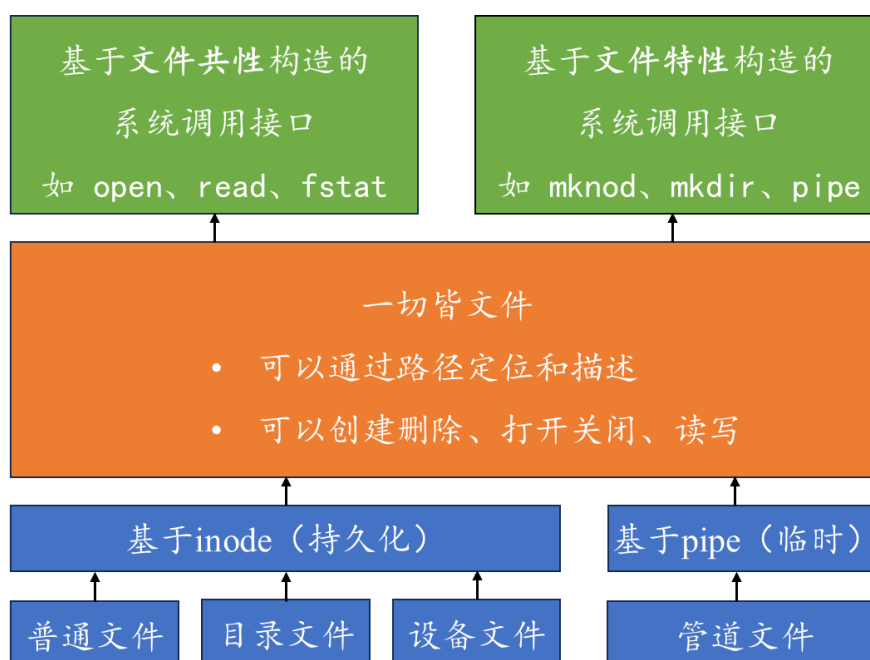


图 2-10 文件和文件相关系统调用

Figure 2-10 Files and File-Related System Calls

2.7 小结

本章按照模块顺序，系统性地介绍了本文设计实现的基础版本内核。首先介绍了内核的整体架构，分为用户态测试程序、内核态功能模块、QEMU 模拟环境三部分；之后介绍五个具体模块：机器启动、内存管理、中断异常、进程管理、文件系统，在每一小节都展开介绍了该模块的核心设计思想和关键技术实现。

然而，编写操作系统内核并不是逐个编写各个模块，而是将模块拆碎并梳理出其中的依赖关系，沿着依赖关系构成的拓扑序列一步步前进。笔者以为，拆分重组并逐步搭建的过程，是理解操作系统内核与培养系统性思维的关键，这部分将在下一章“内核的实现过程与实验指导”中做详细阐述。

3、内核的实现过程与实验指导

本章具体讨论两个问题：本文实现的基础版本内核是如何一步一步搭建起来的（3.1 - 3.4）；以及笔者如何将内核编写经验设计为课程实验并投入使用（3.5）。

3.1 内核实现思路

3.1.1 内核编写的困难

对于希望动手编写内核的操作系统初学者，一种常见的学习方式是：选择一个足够简单的教学性操作系统作为参考，尝试从结果出发逆向解构，找出它从零开始逐步发展到完整内核的路径。本文所实现基础版本内核也是这种思路，而进阶版本内核实现了更多 Linux 中的高级功能（将在第 4 章详细论述）。

逆向解构的过程并不轻松，它面临以下困难：（1）需要对现有内核的所有模块足够熟悉，理解各个模块的核心逻辑。（2）需要厘清模块间的依赖关系，有时需要把模块的基础功能和进阶功能分离，才能找出隐藏的依赖关系。（3）以操作系统内核为代表的系统级编程涉及很多操作系统之外的知识，如编译原理、计算机体系结构、软件工程等。第二点通常是最困难的，也是本章论述的重点。

3.1.2 实现思路概览

本文所实现的内核遵循以下基本原则：（1）模块化的源文件组织：与 xv6 将所有源文件直接放在 `kernel` 目录下的做法不同，本文在 `kernel` 目录下设置多个子目录，每个模块的源文件放在对应目录内。（2）从开发者角度而非阅读者角度出发，思考内核构建逻辑，将复杂功能拆分成简单功能的迭代，遵循用最简单的方法实现目的再逐步优化的思路。（3）保持代码的独立性，以 xv6 为参考和样本，但不套用 xv6 的逻辑。

具体来说，内核实现思路可以细分为下面九个步骤，每三个步骤构成一个阶段。图 3-1 显示了各个步骤的代码量变化趋势，代码量快速增长的步骤往往更困难。

- （1）机器启动（进入 `main` 函数 + 第一次输出字符串）
- （2）内存管理初步（物理内存管理 + 内核态虚拟内存管理）
- （3）中断异常处理初步（RISC-V 的 `trap` 机制，关注 `kernel trap`）
- （4）第一个用户进程的诞生（`context` + `trapframe` + `user trap`）
- （5）用户态虚拟内存管理 + 系统调用流程建立
- （6）从单进程到多进程（进程生命周期 + 多进程调度）
- （7）文件系统之底层设备（磁盘布局 + 磁盘驱动 + `buffer` 管理）
- （8）文件系统之中层抽象（`inode` + `directory` + `path`）

(9) 文件系统之上层接口 (file + fs_syscall + 文件系统与进程的融合)



图 3-1 各步骤的内核代码量变化趋势

Figure 3-1 Trend of Kernel Code Volume Changes at Each Steps

3.2 第一阶段：无进程的内核

首先需要考虑的问题是：内核中哪些部分是最基础的？考虑这个问题要先跳脱出操作系统的大框架，就会发现需要编写的只是一个相对底层的程序，这个程序不承担操作系统需要承担的责任，只需要向底层硬件发号施令，运行起来即可。具体来说，本阶段可以分成三个步骤：机器启动、内存管理初步、中断异常初步。

3.2.1 第一步：机器启动

机器启动阶段首先需要考虑的问题是如何让执行流转移到操作系统的 `main` 函数。这个过程涉及 RISC-V 特权状态的转换：M-mode (QEMU、`entry.S`、`start.c`) 到 S-mode (`main.c`)。其中 QEMU 负责提供硬件环境抽象，保证 PC 初始时指向 `0x80000000`；`entry.S` 负责为每个 CPU 核心准备栈空间；`start.c` 负责关闭分页和保存 `cpu_id`；`main.c` 只需要简单输出一些字符，显示已经进入 `main` 函数即可。

`main.c` 的输出需求引出了第二个需要考虑的问题：如何向控制台终端输出一些字符。这一步可以拆分为两个子步骤：第一步是实现底层的串口 (Universal Asynchronous Receiver/Transmitter, UART) 驱动，本质是按照 QEMU 对串口的定义做寄存器读写；第二步是将串口驱动包装成更方便的 `printf` 函数，主要关注函数栈内参数读取。

`printf` 函数还存在一个隐患：多核并行带来的资源争用。比如 CPU-A 和 CPU-B 都在执行 `printf` 函数，会争用 UART 的输出能力，导致两个 CPU 核心的输出发生交织。

为了解决这个问题，需要引入一个最简单的并行控制工具——自旋锁。自旋锁的基本原理是原子操作加循环检查过程：设置一个让 CPU-A 和 CPU-B 都可见的共享变量 `UART_BUSY`，上锁操作是检查 `UART_BUSY`，如果它是 0 就设置为 1 宣布占有资源，如果它是 1 说明上锁失败，再次尝试；解锁操作是将 `UART_BUSY` 设置为 0 宣布释放资源。自旋锁最重要的就是上锁过程“检查和设置”一气呵成，由 RISC-V 提供的原子操作来保证。自旋锁为短时间独占资源提供了可靠保障，是并行性的基础。

3.2.2 第二步：内存管理初步

第一步涉及的硬件设备是 CPU 和 UART，第二步主要关注内存。UART 这种资源通常是独占的，任何时刻只能被一个 CPU 占有；而内存的共享性更强，多个 CPU 可以共享一块内存空间，这就提出了一个新的需求：如何管理共享的内存空间？

首先只考虑物理内存，将内存空间切分成多个子空间（通常是 4KB），称为页面（page），不同的页面可以分配给不同的 CPU 独占使用。为了记录这种分配关系，本文使用空闲链表将所有可分配的页面串联起来，把页面的分配和回收转换成链表节点的删除和插入。为了保障内存资源分配的有序进行，使用自旋锁保障每一时刻只有一个 CPU 能访问这个全局的页面链表。

只实现物理内存不足以支撑一些进阶功能：（1）通过将外设寄存器映射到内存区域，复用内存访问方式操作寄存器。（2）程序之间的地址空间隔离，由于不同 CPU 执行的内核态程序共用一套代码，这个问题在内核中并不明显，但是用户态进程非常需要这种隔离性。于是，还需要建立内核页表，它是一一映射的（物理地址等于虚拟地址），包括外设寄存器地址映射、内核数据和代码映射、剩余空闲内存空间映射。

3.2.3 第三步：中断异常管理初步

中断和异常在 RISC-V 体系结构中都被定义为陷阱，陷阱是一种主线控制流外的支线控制流，好像走在路上（主线）突然掉进坑里（支线），后面又爬上来。在当前阶段，只需处理两种中断——时钟中断和 UART 中断；对于其他中断和异常，只需要输出提示信息即可。在后面的阶段，会逐步增加磁盘中断、缺页异常、系统调用等新的陷阱类型。根据从简单到复杂的原则，这些问题目前不需要考虑。

值得一提的是，时钟中断默认发生在 M-mode，M-mode 负责完成一些寄存器操作（如重置计数器），随后主动制造一个 S-mode 软件中断，让运行在 S-mode 的操作系统完成剩下的工作。目前可以做的事情只有输出时钟滴答，时钟中断后面会服务于进程的各种调度算法和事件响应。

3.2.4 一阶段小结

走完这三个步骤，内核具备了以下能力：（1）可以多核启动并在自旋锁的帮助下正确并行。（2）管理了两类共享资源：串口和物理页。（3）构建了页表系统，书写了内核页表。（4）构建了陷阱机制，可以打断执行流，目前能处理两类中断：时钟中断和 UART 外设中断。总而言之，第一阶段（代码量为 1292 行）搭建了一个无进程的基础内核，创建进程的基本条件（物理页、页表、陷阱处理）已经成熟，第二阶段主要关注用户进程的创建和管理。

3.3 第二阶段：无持久化能力的内核

需要注意的是，内核程序本身也可以看作一种进程，它也具备静态的代码和动态的运行状态，本文称其为 CPU 进程。每个 CPU 在执行内核代码时都会产生对应的 CPU 进程，一阶段实现了多个 CPU 进程并行执行，二阶段将引入用户进程，CPU 进程的职责是服务并管理用户进程。二阶段同样分为三个步骤：第一个用户进程的诞生、用户态虚拟内存管理与系统调用流程建立、从单进程到多进程。

3.3.1 第一步：第一个用户进程的诞生

前面提到，在第一个用户进程诞生之前，执行 start 函数、main 函数、printf 函数的是 CPU 进程，它使用全局的提前分配的函数栈作为函数执行环境。本步骤的核心目标是构建第一个用户进程 `proczero`，让 `proczero` 从用户态发出一个内核可以接收和响应的请求即可。

引入用户进程带来了两个问题：上下文切换和用户态内核态切换。不同于 CPU 进程只运行在内核态，用户进程诞生于内核态，但大多数时间在用户态执行。一个典型的过程如下：（1）CPU 进程创建 `proczero`；（2）CPU 进程将 CPU 使用权交给 `proczero`；

（3）`proczero` 从内核态进入用户态。第二步中，CPU 进程需要将运行环境上下文保存到特定内存区域中；第三步中，`proczero` 需要将内核态运行环境上下文保存到特定内存区域中。前者在进程调度过程中发生，使用的内存区域称为 `context`；后者在特权状态切换过程中发生，使用的内存区域称为 `trapframe`。两个过程的共同点在于都是将寄存器信息保存到内存中，以便腾出寄存器空间服务于新的进程；不同点在于特权状态切换需要保存的寄存器更多。

如图 3-2 所示，第一个进程的地址空间由以下几个部分构成：`trampoline` 是特权状态切换的代码区域，`trapframe` 是特权状态切换时寄存器数据暂存区域，`ustack` 是用户使用的函数栈，`heap` 是用户使用的堆区域，中间部分是空白区域用于承载堆和栈的

生长，code + data 是用户程序的代码和静态数据区域，最下面的空白区域不使用。此时，进程结构体的定义非常简单，主要用于描述进程的地址空间。值得注意的是，进程在用户态使用用户页表 and 用户栈，在内核态使用内核页表 and 内核栈。

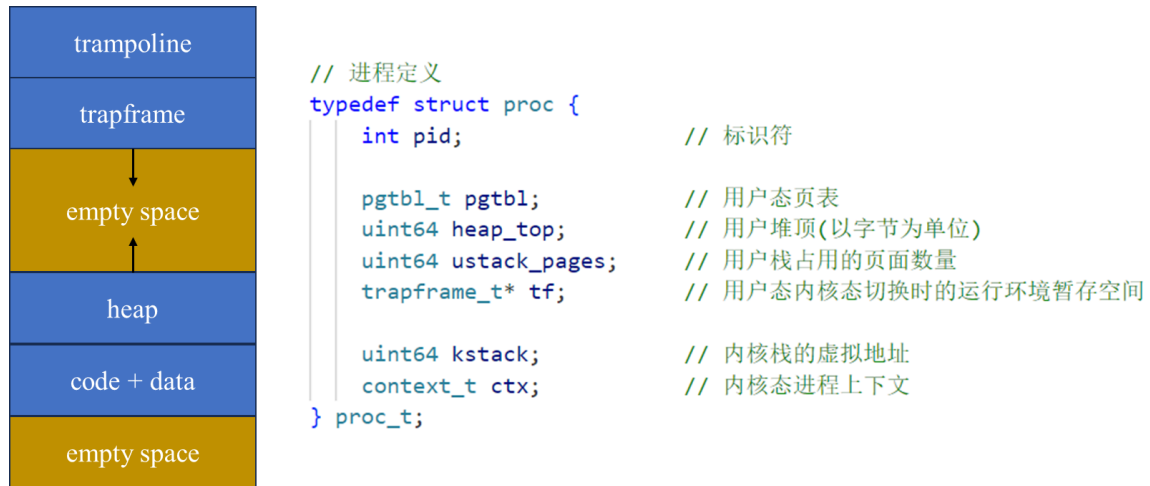


图 3-2 第一个进程的地址空间（左）与结构体定义（右）

Figure 3-2 The address space of the first process (left) and the structure definition (right)

3.3.2 第二步：用户态虚拟内存管理与系统调用流程建立

第一个进程 `proczero` 诞生时只具备最简单的能力（即进入用户态执行），本步骤将为它增加两个重要功能：用户态虚拟内存管理与系统调用发起。

首先讨论系统调用流程的建立过程。系统调用在 RISC-V 中被定义为一种异常，它是用户程序请求内核服务的接口，用户程序依赖系统调用完成绝大多数功能。系统调用流程建立分为三步：（1）在用户态和内核态约定一致的系统调用编号和系统调用的传参规范，如将 `read` 系统调用编号设置为 1，参数包括 32bit 文件描述符、32bit 文件内偏移量、32bit 读取长度、64bit 目标地址等。（2）用户进程在对应寄存器填写函数参数并主动触发一个异常，这个异常序号在 RISC-V 里定义为 8。（3）内核中的陷阱处理函数识别系统调用编号，根据前面的约定调用对应的系统调用处理函数。

系统调用的建立过程留下了一个问题：用户和内核间的数据传递。数据传递方法分为两种：（1）在寄存器里存放数据本身（如系统调用编号）。（2）在寄存器里存放数据地址（如长数组的地址）。前者可以直接访问，后者则涉及页表映射的翻译，因为用户和内核使用的是不同的两张页表。页表映射翻译分为两种情况：第一种是相关硬件根据 `satp` 寄存器中的页表地址自动完成翻译；第二种是根据页表地址手动逐级查询页表获得物理地址。为了读入或传出用户地址空间的数据，内核需要查询用户页表，通过手动翻译的方法获取逻辑地址对应的物理地址。

除了数据传递，内核态虚拟内存管理还负责控制栈和堆的生长，以及页表的复制和销毁。栈的生长通过缺页异常实现：内核收到缺页异常信息后检查产生异常的地址，如果发现是栈顶的下一个页面，就判定为缺页异常，通过页表映射的方式申请新的地址空间消除这个异常。堆的生长通过系统调用实现：用户通过系统调用声明需要堆生长到什么地址空间，内核通过页表映射的方式申请新的地址空间。总而言之，栈的生长是被动的，对用户透明的；堆的生长是主动的，对用户不透明的。而页表的复制和销毁是为下一阶段多进程管理做准备。

3.3.3 第三步：从单进程到多进程

目前为止，内核里只有两个进程（假设只有 1 个 CPU）：CPU 进程和 `proczero`。前两步已经将 `proczero` 的基本功能实现了，这一步将通过批量复制引入更多用户进程，并通过进程生命周期和进程调度器来调节进程间的资源共享。

进程生命周期管理的本质是调节进程获取资源能力的强弱：处于 `running` 状态的进程可以使用一切可用的资源；处于 `runnable` 状态的进程有较大可能性在一段时间后获得所需资源；处于 `sleeping` 状态的进程除非被唤醒否则不可能获得资源；处于 `zombie` 状态的进程只能释放已有资源，不能新获取任何资源；处于 `unused` 状态的进程是空白的，不持有任何资源且不能获取资源。进程调度的本质是动态的资源分配：由于 CPU 资源的稀缺性（1 个 CPU 上并发执行多个进程），不能让单个进程长时间占用 CPU 资源，于是需要进行调度，确保 CPU 资源在一段时间内被各个进程共享。

从单进程到多进程，体现了内核设计的重要原则：从最简单的方案开始，随着需求提高逐步优化和扩充，即从简单到复杂。除了 `proczero` 是手动创建的，其他进程都是复制父进程产生的，最困难的步骤往往是第一步。

3.3.4 二阶段小结

走完这三个步骤，内核新具备了以下三个能力：（1）用户进程从无到有，从单个走向多个；（2）引入进程生命周期管理和进程调度器；（3）从内核态走向用户态，将内核态陷阱扩展到用户态陷阱，建立了内核和用户的沟通窗口——系统调用。到目前为止，机器启动、进程管理、内存管理、陷阱处理、并发控制、设备驱动等模块已经基本完成（共 2696 行代码）。然而，所有操作主要发生在 CPU 和内存中，断电重启后所有数据都会清零和丢失。内核还欠缺一项必要能力——数据持久化，这需要磁盘（硬件）和文件系统（软件）的帮助。

3.4 第三阶段：相对完善的内核

文件系统是操作系统中层次结构最丰富的模块，本文将其拆分为三个大的层次：底层设备（提供基本的磁盘读写能力）、中层抽象（提供关键的磁盘逻辑抽象）、上层接口（提供统一的文件访问方式）。另外，由于本节内容在 2.6 节已有详细论述，故主要讨论构建思路而不展开细节。

3.4.1 第一步：文件系统之底层设备

设计文件系统首先要考虑磁盘布局，本文设计的文件系统磁盘布局包括三个部分：第一部分是超级块，它记录了文件系统和磁盘的全局信息；第二部分是元数据区域，它包括 `inode bitmap` 和 `inode` 数组；第三部分是数据区域，它包括 `data bitmap` 和 `block` 数组。确定磁盘布局后，下一步是创建磁盘映像文件 `fs.img`，创建者是一个用户态程序，通过 Linux 提供的库函数新建一个文件并初始化成目标磁盘布局。`fs.img` 会作为虚拟磁盘的参数供 QEMU 使用。有了符合需求的虚拟磁盘后，下一步就是使用磁盘驱动程序对虚拟磁盘进行读写操作，由 `virtio_disk.c` 中的函数实现。磁盘读写以 `block` 为单位（通常为 512KB），内存中与之对应的缓冲区结构称为 `buffer`，磁盘的读写被具体化为数据从 `block` 到 `buffer` 或从 `buffer` 到 `block` 的双向流动过程。除了基本的 `block` 读写，对于 `bitmap` 中每一个 `bit` 的管理，也在这一层通过位运算实现。

3.4.2 第二步：文件系统之中层抽象

第一步主要向上层提供读写物理磁盘的能力，本步主要构建对磁盘中文件元数据和数据的逻辑访问能力。磁盘中元数据和数据的组织离不开三个重要组件：`inode`、目录项、文件路径。`inode` 用于维护单个文件的数据布局、读写权限、时间戳等信息；目录项用于维护目录树中上层到下层的逻辑关系；文件路径用于打通各个目录层次，汇集成一个完整的树上路径。具体来说，本步需要做的事情就是维护这三个组件的数据结构并实现相应操作函数，如 `inode` 申请、释放、同步，目录项创建、删除、查找，文件路径解析等。

3.4.3 第三步：文件系统之上层接口

上层用户程序可以通过文件的绝对路径、相对路径（配合进程里的当前目录）、文件描述符（配合进程里的打开文件表）定位一个文件。文件系统为用户进程提供持久化数据的能力，本步的重点就是进程与文件系统的配合。进程发出的文件请求经过系统调用接口层和文件抽象层的翻译下发给中层抽象层，中层抽象层将文件的逻辑操作转换成磁盘读写操作，进一步下发给底层设备层，底层设备层驱动虚拟磁盘完成相

应读写任务。一个典型的理解是 `exec` 函数的执行，这个函数用于从磁盘中读取某个 ELF 文件，以这个文件的内容为血肉填充子进程的骨架，形成逻辑完全不同于父进程的新进程。`exec` 函数涉及内存管理、进程管理、文件系统三大模块，对于理解模块间连续很有帮助。

3.4.4 三阶段小结

走完这三个步骤，内核新具备了以下能力：（1）引入磁盘这种新的外设，能以 `block` 为单位读写，使得内核具备持久保存数据的能力。（2）引入文件系统，以文件为逻辑单位管理磁盘并向上提供系统调用接口。（3）补上了进程模块最后一块拼图——`exec`，提高多进程的实用性。至此，历经三个阶段九个步骤，一个具备基本功能的小型操作系统内核已经成型（共计 5603 行代码）。

3.5 实验设计与使用情况

前文之所以对内核的实现步骤做细致拆分与分析，主要是服务于“小型内核实现”实验设计。如图 3-3 所示，每个实验都提供了两份信息：第一份是整体框架，包括文件夹创建、源文件创建、目录组织结构、必要的非核心的代码（如汇编代码、设备驱动等）；第二份是实验指导书（以 markdown 格式给出），组织结构通常是：代码组织结构图、实验目的和原理、具体步骤指导、测试用例说明。



图 3-3 实验指导书示意图

Figure 3-3 Schematic Diagram of the Experimental Manual

本套实验已在计算机科学与技术拔尖班中投入使用，取得了良好反馈。在代码框架和实验指导书的帮助下，大多数学生可以独立完成各个实验，且普遍认为这种实验方式相比 xv6 的实验更具灵活性和挑战性，增强了系统实践能力，深化了内核机制理解。此外，本套实验兼容操作系统竞赛需求，在全国大学生计算机系统能力大赛操作系统设计赛（华东区域赛）中，6 名本科生基于实验成果，在“小型内核实现”赛道取得了 1 个一等奖，3 个二等奖，2 个三等奖的优异成绩。获奖选手在实验过程中提出了许多改进建议，推动实验质量持续提升，实验内容持续丰富，测试用例持续完善。

3.6 小结

本章首先介绍了小型内核实现的困难性和重要性，提出了清晰的内核实现思路；随后分三个阶段九个步骤详细介绍了如何从 0 到 1、从易到难实现一个具备基本功能的小型内核；最后介绍了基于九个实现步骤设计的实验方案，以及实验方案投入使用后获得的学生反馈与竞赛成果。然而，笔者认为，哪怕对于教学性的小型内核来说，仅仅具备基本功能也是不够的，进阶版本内核将在下一章中论述。

4、内核中高级机制的设计实现

4.1 概述

第二章和第三章分别从模块设计和实现思路的角度，论述了本文所实现的内核的基础版本（对应分支 lab-1 到 lab-9），基础版本的能力与 xv6 相当，适合作为操作系统课程的实验材料。本章重点讨论内核的进阶版本（对应分支 version-1.0 到 version-1.3），进阶版本在基础版本的基础上增加了若干 Linux 的特性与机制，包括伙伴系统、文件映射、多级反馈队列调度算法三个部分。伙伴系统是一种 Linux 使用的物理内存分配器，为应用程序提供高效的连续物理内存分配服务；文件映射是有一种 Linux 使用的文件访问模式，将频繁读写的文件映射至指定内存区域，像访问内存一样访问文件；多级反馈队列调度是一种 Linux 常用的经典调度思想，通过优先级动态升降的方法兼顾任务的响应时间和吞吐量。从整体结构上看，伙伴系统是内存管理模块的一部分，文件映射是文件系统模块的一部分，多级反馈队列调度算法是进程管理模块的一部分，三个升级点分别覆盖了操作系统核心的三个模块。

4.2 伙伴系统

4.2.1 伙伴系统的原理

在 2.3.1 中讨论过，基础版本内核使用的物理内存分配器是一种链式分配器，它将每个 4KB 物理页视为一个节点，分配的基本单元是物理页。以物理页为基本单元的分配策略在内核中是合适的，因为内核程序通常不会动态获取和释放较大的连续物理内存空间。然而，用户程序对连续物理内存的需求往往更大。以视频播放程序为例，它经常需要申请一个较大的连续空间，用于缓存从网络侧收到的视频，再将视频传送到 GPU 进行渲染。这个过程涉及大块连续内存空间的申请和释放，如果使用内核的链式物理内存分配器，一方面不能保证物理页连续性，另一方面管理效率较低。

为了满足用户程序对连续物理内存高效分配回收的需求，使用伙伴系统^[12]作为用户空间的内存分配器。伙伴系统将内存划分成大小为 2 的幂次方的块^{[13][14]}（如 4KB、8KB、16KB、32KB 等），以块为单位进行物理内存的分配和回收。具体来说，伙伴系统维护一个多阶链表，同阶链表内各个块大小相同，例如：阶为 0 的链表所管理的块都是 4KB，阶为 1 的链表所管理的块都是 8KB，以此类推。如图 4-1 所示，假设用户进程申请 8KB 物理内存，伙伴系统发现最小的可以满足需求的块是 32KB，于是对这个 32KB 块进行两次分裂：第一次分裂的碎片 1 进入第二次分裂，碎片 2 进入 list[2]；第二次分裂的碎片 1 作为被申请的内存块离开空闲链表，碎片 2 进入 list[1]。相对应

的，内存释放过程如图 4-2 所示，被释放的 4KB 块与 list[0]上的 4KB 块构成伙伴关系（它们由同一个 8KB 块分裂形成，在物理地址上存在联系），通过第一次合并形成 8KB 块；这个 8KB 块与 list[1]上的 8KB 块也构成伙伴关系，于是发生第二次合并，形成 16KB 块，进入 list[2]。伙伴块的分裂与合并是伙伴系统的核心逻辑，保证了大块连续空间的管理效率。

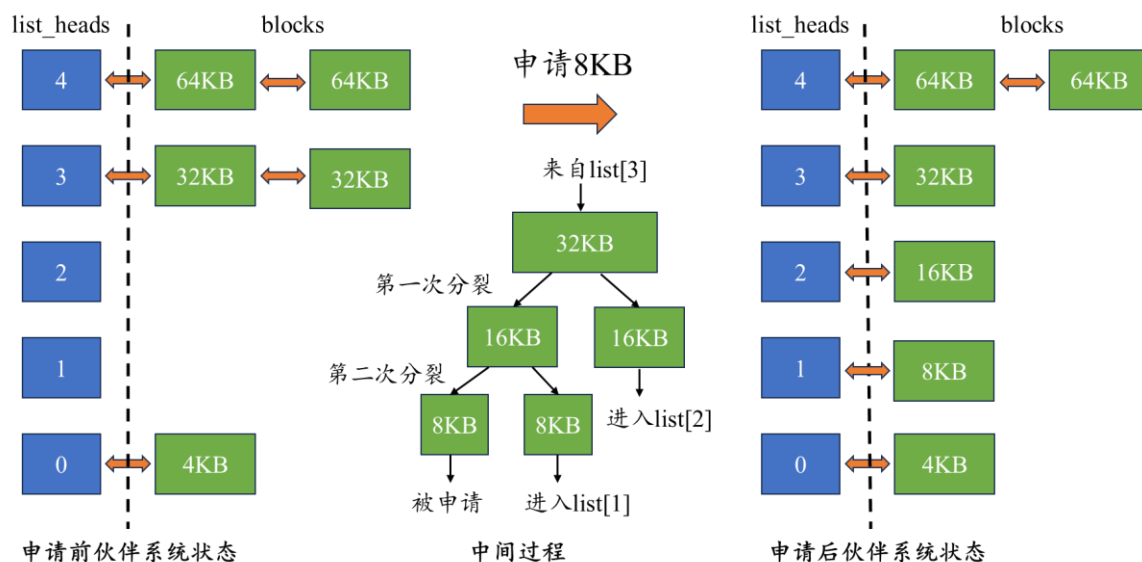


图 4-1 通过伙伴系统申请 8KB 内存块的过程

Figure 4-1 The process of allocating an 8KB memory block through the buddy system

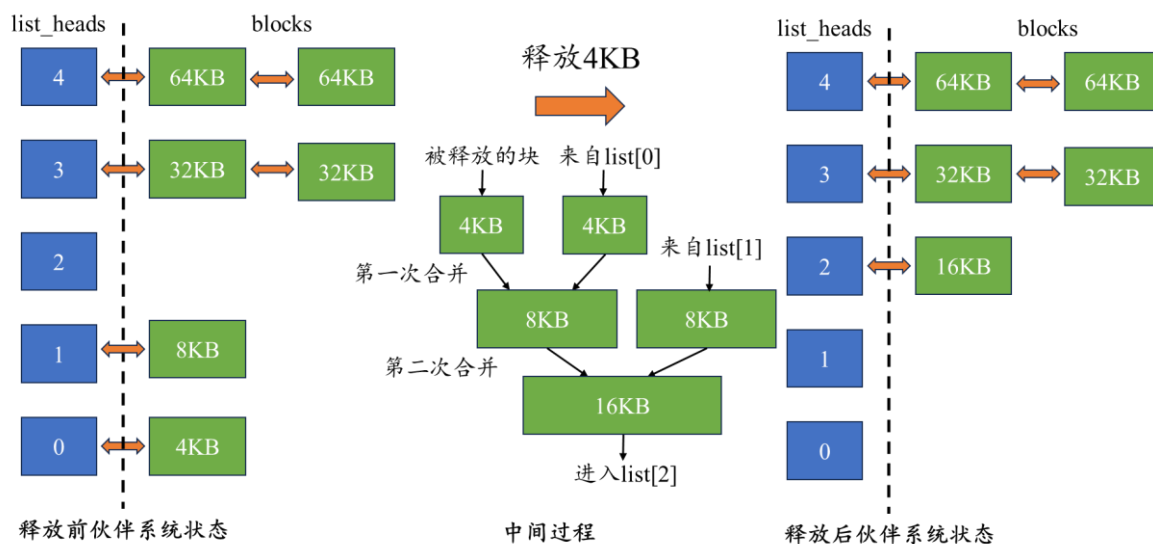


图 4-2 通过伙伴系统释放 4KB 内存块的过程

Figure 4-2 The process of freeing a 4KB memory block through the buddy system

4.2.2 伙伴系统的实现

伙伴系统的实现集中在 mem 目录下的 pmem.c 文件，主要分为四个部分：第一部

分是链表操作，buddy_alloc_block 函数负责从某个阶的链表中获取一个 block，buddy_free_block 函数负责将节点放回对应阶的链表中；第二部分负责低阶块合并和高阶块分裂，buddy_merge_block 函数负责尽可能合并小的 block 形成一个大的 block，buddy_split_block 负责把一个大的 block 拆分成一个目标大小的 block；第三部分负责对外提供伙伴系统的访问接口，buddy_alloc 函数用于申请 N 阶连续块，buddy_free 函数负责释放之前申请的连续块；第四部分负责提供 debug 信息，buddy_show_list 函数负责输出伙伴系统各阶空闲链表的状态。

除了核心逻辑部分，为了便于用户态程序访问，使用伙伴系统的函数接口替换掉了 sys_mmap 和 sys_munmap 系统调用中的 pmem_alloc 函数和 pmem_free 函数。于是，用户程序可以通过 mmap 的方式向伙伴系统申请连续的物理内存空间，这是 5.3.1 中功能与性能测试的基础。

4.3 文件映射机制

4.3.1 文件映射的原理

基础版本内核的文件系统实现了基本的文件访问接口，一种典型的文件访问方式是：用户程序先使用 sys_open 打开文件，随后使用 sys_read 读取某些区域，修改后使用 sys_write 写回，最后使用 sys_close 关闭文件。这种访问模式的问题在于用户程序过于依赖系统调用接口来操作文件。如图 4-3 所示，对于 N 次修改一个大文件的场景（如修改 Linux 源代码），需要 $2+2*N$ 次文件相关系统调用（不考虑指针移动）。

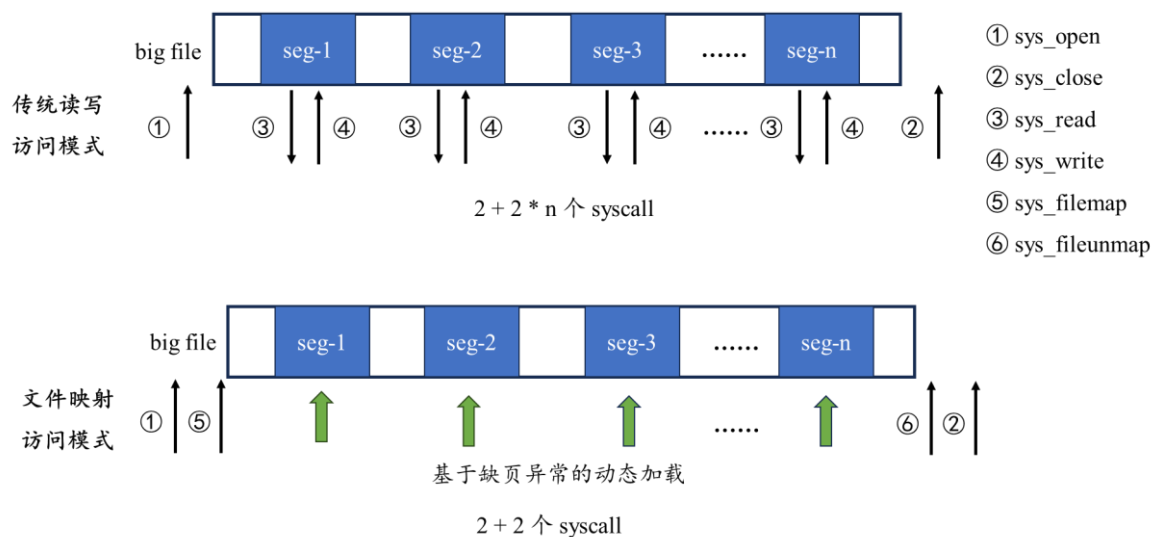


图 4-3 传统读写访问模式与文件映射访问模式的系统调用流分析

Figure 4-3 Analysis of System Call Flow for Traditional Read/Write Access Patterns and File Mapping

为了解决这个问题，本文引入 Linux 中的文件映射机制^[15]。如图 4-4 所示，传统文件访问模式首先将磁盘里的文件内容拷贝至内核空间的一块区域（`kern_buf`），随后将这块区域的内容拷贝至用户空间以供用户访问。文件映射机制只做一次拷贝，即必要的磁盘到内存的拷贝，随后通过用户页表将内核空间的物理内存映射到用户的虚拟地址空间^[16]（并非第一次这么做，中断异常模块的 `trapframe` 和 `trampoline` 也是这样映射）。通过地址映射的方式，用户可以自由访问内核资源而不需要走系统调用的流程。相比传统访问模式，文件映射访问模式处理相同任务只需发出 2+2 个系统调用。

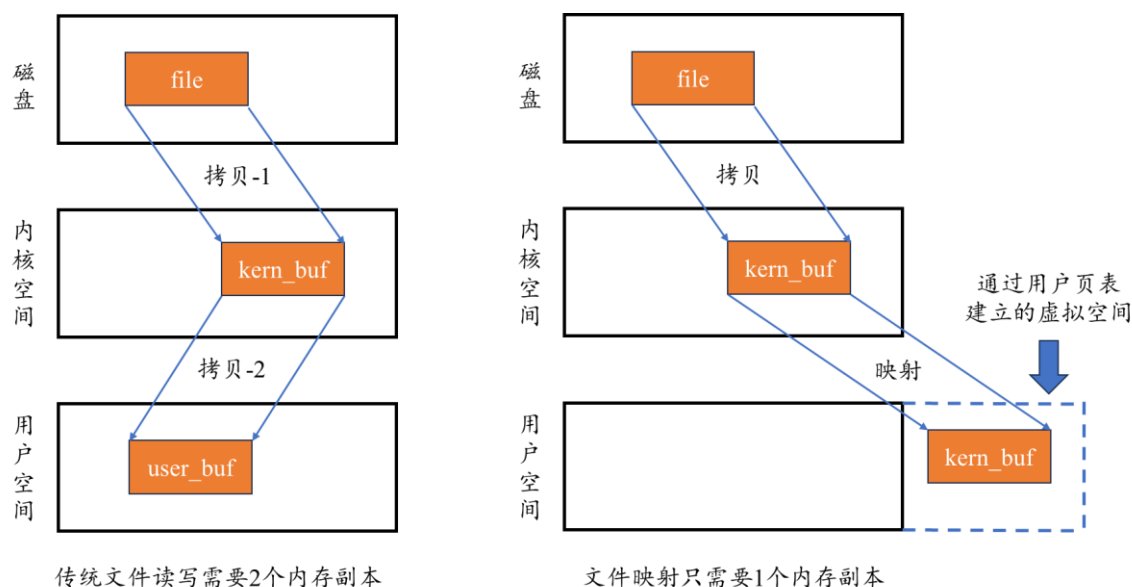


图 4-4 传统读写访问模式与文件映射访问模式的内存副本

Figure 4-4 Memory Copy in Traditional Read/Write Access Pattern and File Mapping Access Pattern

这种想法还存在一个问题：不同于用户空间，内核空间的资源是共享的和稀有的，如果一个用户程序打开了很大的文件但只操作文件中一小部分数据，会导致大量内核空间的物理内存被浪费。为解决这个问题，本文使用延迟分配策略（`lazy allocation`）：用户使用 `sys_filemap` 映射文件时，只在用户页表中建立虚拟空间（没有实际开销），如果后面访问了文件虚拟空间的某个部分，会触发缺页异常，并为这个虚拟区域分配实际的物理页。通过延迟分配，可以确保宝贵的内核物理页面按需分配，同时保证应用程序感知不到额外的时延或其他问题。

4.3.2 文件映射的实现

文件映射的实现可以分为 4 个部分：（1）进程相关：在进程定义时增加记录文件映射信息的数据结构 `vma`，维护一个数组 `vmalist` 记录当前进程映射了哪些文件，在 `proc_make_first`、`proc_fork`、`proc_exit` 函数中做相应维护。（2）`sys_filemap` 系统调用：

这个系统调用负责接收外部参数，并在 `proc` 的 `vma_t` 数组中找到一个空闲空间记录这些参数，同时扩展进程的虚拟地址空间。（3）缺页异常处理：检查引发缺页异常的地址属于进程的哪个映射空间，申请物理页并读取这个映射空间在磁盘里对应的数据。

（4）`sys_fileunmap` 系统调用：将内存中的数据写回磁盘，解除虚拟地址空间的映射，释放进程的 `vmlist` 资源等。

4.4 多级反馈队列调度算法

4.4.1 多级反馈队列调度算法的原理

在基础版本内核中，进程调度算法采用简单的时间片轮转调度算法，当发生时钟中断时，当前进程让出 CPU 使用权，下一个 `runnable` 进程获取 CPU 使用权。这种算法存在一些问题：（1）不区分进程优先级，紧急的前台任务和长期的后台任务一视同仁。（2）时间片设置后不可变化，欠缺灵活性。（3）对于新进程的响应时间较长。

为了解决以上问题，本文参考 Linux 使用的完全公平调度器^{[17][18]}，使用多级反馈队列算法^{[19][20]}（Multi-Level Feedback Queue, MLFQ）作为进程调度算法。如图 4-5 所示，MLFQ 算法维护一组全局的优先级队列，每个优先级队列对应一个时间片（优先级越高的队列时间片越少），各个优先级队列上链接着零个到多个处于 `runnable` 状态的进程，同一个优先级队列上的进程拥有的时间片相同。

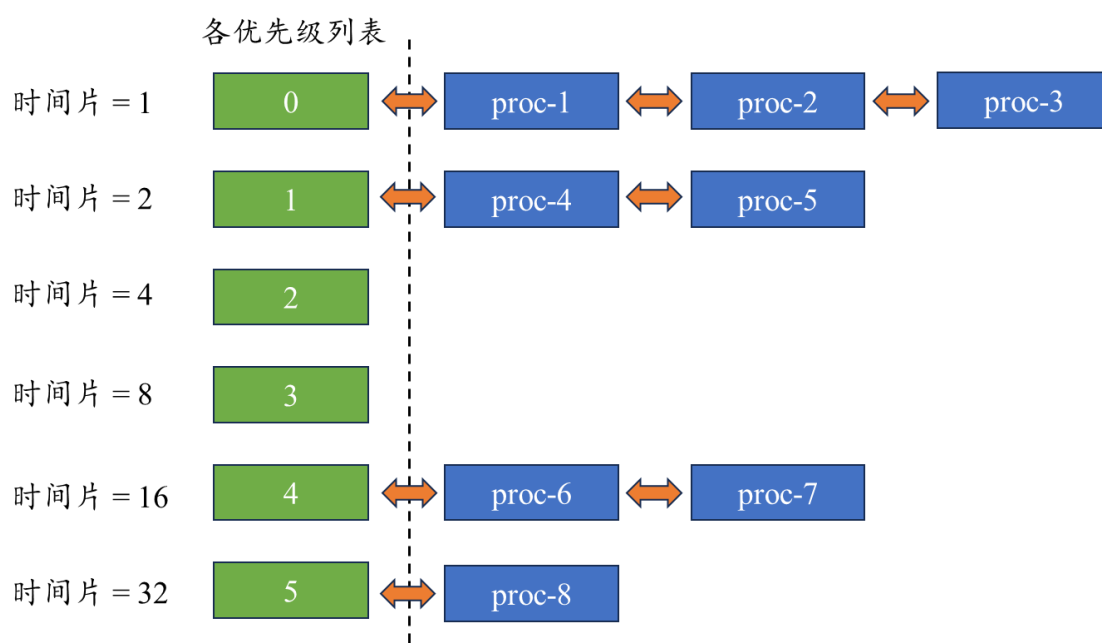


图 4-5 多级反馈队列示意图

Figure 4-5 Schematic Diagram of Multi-Level Feedback Queue

MLFQ 的工作过程遵循以下几条规则：（1）如果进程 A 的优先级高于进程 B，则

运行 A 而不是 B。(2) 如果进程 A 的优先级等于进程 B，则轮转运行进程 A 和进程 B。(3) 新的进程进入队列时，默认放在最高优先级的队列。(4) 进程耗尽时间片后进入低一级队列，最低优先级的队列遵循时间片轮转原则。(5) 一段时间后，调度器洗牌，将所有进程放入最高优先级队列。基于这些规则，MLFQ 算法可以在不提前获知优先级的情况下，智能调整用户优先级，兼顾响应时间、周转时间、公平性、吞吐量等指标。

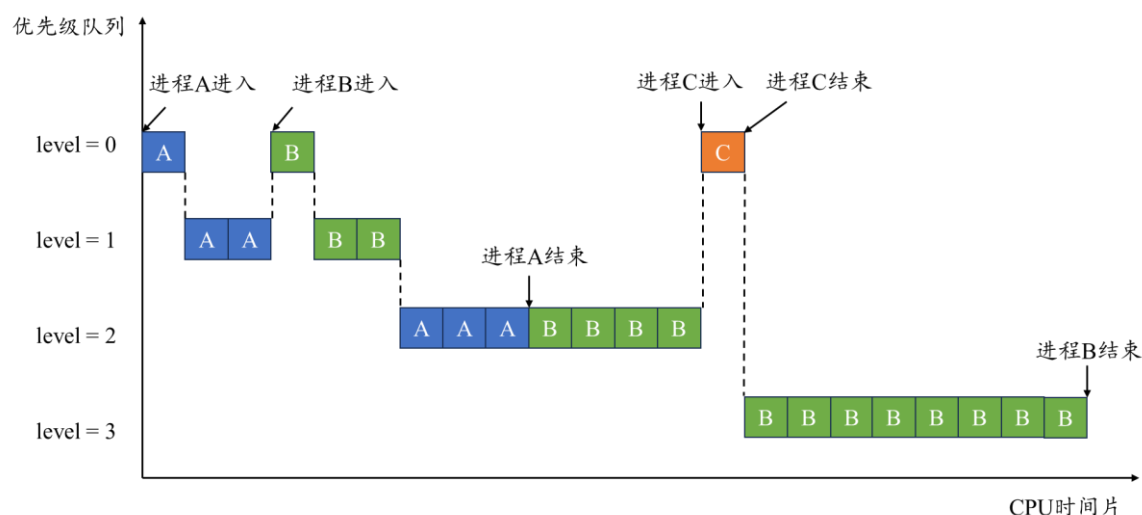


图 4-6 MLFQ 算法多进程调度过程

Figure 4-6 Scheduling Process of Multiple Processes in the MLFQ Algorithm

图 4-6 显示了典型的 MLFQ 调度算法工作过程：(1) 0 时刻，进程 A 进入 0 级队列，执行 1 个时间片；(2) 进程 A 调入 1 级队列，执行 2 个时间片，调入 2 级队列；(3) 进程 B 进入 0 级队列，执行 1 个时间片；(4) 进程 B 调入 1 级队列，执行 2 个时间片，调入 2 级队列；(5) 进程 A 执行 3 个时间片，结束；(6) 进程 B 执行 4 个时间片，调入 3 级队列；(7) 进程 C 进入抢占 CPU，执行 1 个时间片，结束；(9) 进程 B 执行剩下的 8 个时间片，结束。

4.4.2 多级反馈队列调度算法的实现

MLFQ 调度算法的实现分为三个部分：(1) 修改进程调度器 `proc_scheduler` 函数，查询和修改多级反馈队列。(2) 通过 `proc_make_first` 函数或 `proc_fork` 函数创建新进程后，插入最高优先级调度队列。(3) 修改时钟中断处理函数，负责修改进程剩余时间片，若时间片清零则触发调度操作；此外，当时钟滴答达到定时器设置的洗牌时间时，触发洗牌操作，将所有处于 `runnable` 状态的进程依次放入最高优先级调度队列。总而言之，MLFQ 算法依赖进程模块与中断异常模块的紧密配合。

4.5 小结

在基础版本内核的基础上，本章参考 Linux 中的高级特性与机制，对关键模块做了补充与改进，形成了进阶版本内核。具体来说：对于内存管理模块，使用伙伴系统物理内存分配器改进了之前的链式物理内存分配器；对于文件系统模块，使用文件映射访问模式补充了之前的读写访问模式；对于进程管理模块，使用多级反馈队列调度算法改进了之前的时间片轮转调度算法。更复杂的算法设计带来了更强大的功能与性能，相关测试将在下一章详细介绍。


```

inode information:
num = 1, ref = 1, valid = 1
type = INODE_FILE, major = 0, minor = 0, nlink = 1
size = 0, addr = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

bitmap:
bit 0 is allocated
over

inode information:
num = 1, ref = 1, valid = 1
type = INODE_FILE, major = 0, minor = 0, nlink = 1
size = 1058816, addr = 132 133 134 135 136 137 138 139 140 141 142 399 656
check-1 success
check-2 success
check-3 success
free success

inode information:
num = 1, ref = 1, valid = 1
type = INODE_FILE, major = 0, minor = 0, nlink = 1
size = 0, addr = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

bitmap:
bit 0 is allocated
over

inode_num = 0 dirents:
inum = 0 dirent = .
inum = 0 dirent = ..
inum = 0 dirent = .
inum = 0 dirent = ..
inum = 1 dirent = a.txt
inum = 2 dirent = b.txt
inum = 3 dirent = c.txt

inode_num = 0 dirents:
inum = 0 dirent = .
inum = 0 dirent = ..
inum = 2 dirent = b.txt
inum = 3 dirent = c.txt

inode_num = 0 dirents:
inum = 0 dirent = .
inum = 0 dirent = ..
inum = 1 dirent = d.txt
inum = 2 dirent = b.txt
inum = 3 dirent = c.txt
over

```

图 5-4 文件读写（左）与目录创建删除（右）

Figure 5-4 File Read/Write (Left) and Directory Creation/Deletion (Right)

图 5-1、图 5-2、图 5-3、图 5-4 分别显示了机器启动、内存管理、进程管理、文件系统等模块的典型测试，包括模块功能测试、临界状态测试、随机请求测试等。过程性测试对于系统开发的重要性不言而喻，如果等内核完全开发完成再进行测试，那么各个模块都存在大量故障和错误，调试和 Debug 会变得异常困难。

5.2 系统调用测试

过程性测试是局部模块的、系统视角的测试，而系统调用测试是全链路的、用户视角的测试。用户通过系统调用请求内核服务，系统调用往往穿过多层，需要多个模块合作完成。因此，相比过程性测试，系统调用测试更具综合性和实用性。表 5-1 显示了系统调用列表和对应的测试方法。

表 5-1 系统调用测试列表

Table 5-1 System Call Test List

系统调用名称	系统测试方法
sys_exec	进程 1 调用 sys_exec 执行进程 2，检查进程 2 执行情况
sys_brk	测试用户进程堆空间的伸缩，检查物理内存管理和虚拟内存管理
sys_mmap	测试匿名映射和文件映射的正确性
sys_munmap	测试是否能消除 sys_mmap 的影响
sys_fork	测试是否能正确复制进程并产生新的执行流分支
sys_wait	测试是否能正确等待并处理退出的子进程
sys_exit	测试进程是否能正常退出

sys_sleep	测试进程是否能正常睡眠指定时间
sys_open	测试文件打开
sys_close	测试文件关闭
sys_read	测试文件读取
sys_write	测试文件写入
sys_lseek	测试文件读写指针偏移
sys_dup	测试文件描述符复制
sys_fstat	测试文件状态获取
sys_getdir	测试目录项获取
sys_mkdir	测试目录创建
sys_chdir	测试当前目录修改
sys_link	测试文件链接
sys_unlink	测试文件解链接
sys_getticks	获取时钟滴答数目，辅助性能测试

```

-----test start-----
sys_dup success
world hello
root dirents information:
inum = 0, dirname = .
inum = 0, dirname = ..
inum = 1, dirname = test
inum = 2, dirname = console
inum = 3, dirname = hello.txt
inum = 3, dirname = world.txt

file state:
inum = 3 nlink = 2 size = 12 type = FILE
file state:
inum = 3 nlink = 1 size = 12 type = FILE
root dirents information:
inum = 0, dirname = .
inum = 0, dirname = ..
inum = 1, dirname = test
inum = 2, dirname = console

-----test over-----

-----test start-----
root dirents information:
inum = 0, dirname = .
inum = 0, dirname = ..
inum = 1, dirname = test
inum = 2, dirname = console

root dirents information:
inum = 0, dirname = .
inum = 0, dirname = ..
inum = 1, dirname = test
inum = 2, dirname = console
inum = 3, dirname = workdir

workdir dirents information:
inum = 3, dirname = .
inum = 0, dirname = ..
inum = 4, dirname = student
inum = 5, dirname = teacher
inum = 6, dirname = hello.txt

workdir dirents information:
inum = 3, dirname = .
inum = 0, dirname = ..
inum = 4, dirname = student
inum = 5, dirname = teacher
inum = 6, dirname = hello.txt

root dirents information:
inum = 0, dirname = .
inum = 0, dirname = ..
inum = 1, dirname = test
inum = 2, dirname = console
inum = 3, dirname = workdir

-----test over-----

```

图 5-5 测试组 4（左）和测试组 5（右）的测试结果

Figure 5-5 Test Results of Test Group 4 (Left) and Test Group 5 (Right)

测试分组情况：由于 sys_brk、sys_mmap、sys_munmap 都与地址空间管理相关，

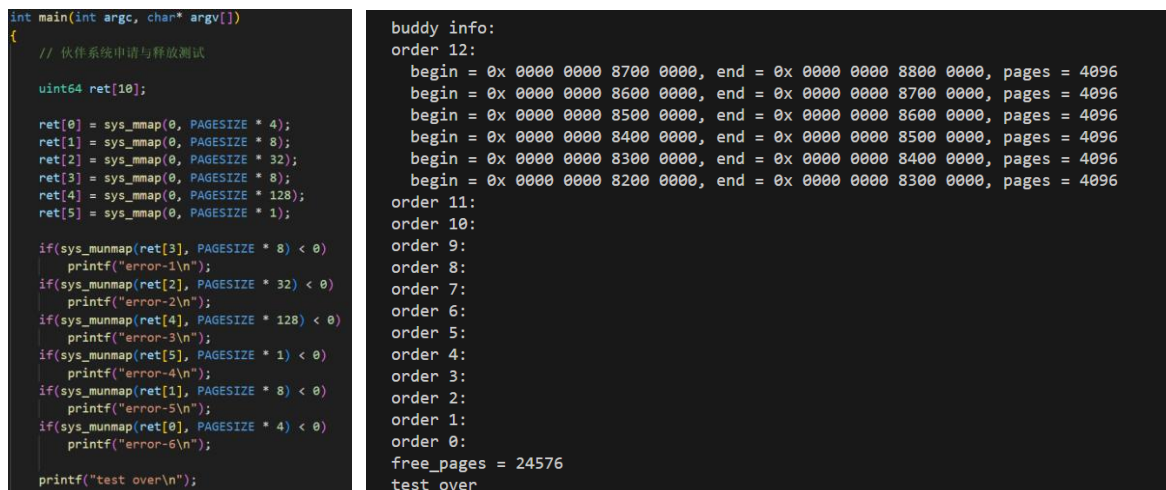
作为测试组 1；由于 `sys_fork`、`sys_wait`、`sys_exit`、`sys_sleep` 都与进程状态转换相关，作为测试组 2；由于 `sys_open`、`sys_close`、`sys_read`、`sys_write`、`sys_lseek` 都属于文件系统基本能力，作为测试组 3；由于 `sys_dup`、`sys_fstat`、`sys_link`、`sys_unlink` 都属于文件系统进阶能力，作为测试组 4；由于 `sys_getdir`、`sys_mkdir`、`sys_chdir` 都与目录管理有关，作为测试组 5；剩下的 `sys_exec` 和 `sys_getticks` 相对独立，作为测试组 6。

每个测试小组对应一个用户态测试程序，测试程序向内核发出一组系统调用请求以测试组内系统调用的正确性和协作能力。图 5-5 显示了测试组 4 和测试组 5 的测试结果，受限于篇幅，其他小组的测试结果不再赘述。

5.3 高级机制测试

5.3.1 伙伴系统测试

如图 5-6 所示，用户测试程序分别申请 4、8、32、8、128、1 个连续物理页，随后乱序释放，在 `sys_mmap` 和 `sys_munmap` 结束时调用 `buddy_show_list` 函数输出伙伴系统状态。检查输出的中间状态，确认符合预期。当所有申请的空间都释放后，伙伴系统恢复了原来的状态（即 6 个 16MB 块空间）。通过这个实验，验证了伙伴系统功能的正确性，可以正确完成各阶物理块的申请和释放。



```

int main(int argc, char* argv[])
{
    // 伙伴系统申请与释放测试

    uint64 ret[10];

    ret[0] = sys_mmap(0, PAGE_SIZE * 4);
    ret[1] = sys_mmap(0, PAGE_SIZE * 8);
    ret[2] = sys_mmap(0, PAGE_SIZE * 32);
    ret[3] = sys_mmap(0, PAGE_SIZE * 8);
    ret[4] = sys_mmap(0, PAGE_SIZE * 128);
    ret[5] = sys_mmap(0, PAGE_SIZE * 1);

    if(sys_munmap(ret[3], PAGE_SIZE * 8) < 0)
        printf("error-1\n");
    if(sys_munmap(ret[2], PAGE_SIZE * 32) < 0)
        printf("error-2\n");
    if(sys_munmap(ret[4], PAGE_SIZE * 128) < 0)
        printf("error-3\n");
    if(sys_munmap(ret[5], PAGE_SIZE * 1) < 0)
        printf("error-4\n");
    if(sys_munmap(ret[1], PAGE_SIZE * 8) < 0)
        printf("error-5\n");
    if(sys_munmap(ret[0], PAGE_SIZE * 4) < 0)
        printf("error-6\n");

    printf("test over\n");
}

```

```

buddy info:
order 12:
    begin = 0x 0000 0000 8700 0000, end = 0x 0000 0000 8800 0000, pages = 4096
    begin = 0x 0000 0000 8600 0000, end = 0x 0000 0000 8700 0000, pages = 4096
    begin = 0x 0000 0000 8500 0000, end = 0x 0000 0000 8600 0000, pages = 4096
    begin = 0x 0000 0000 8400 0000, end = 0x 0000 0000 8500 0000, pages = 4096
    begin = 0x 0000 0000 8300 0000, end = 0x 0000 0000 8400 0000, pages = 4096
    begin = 0x 0000 0000 8200 0000, end = 0x 0000 0000 8300 0000, pages = 4096
order 11:
order 10:
order 9:
order 8:
order 7:
order 6:
order 5:
order 4:
order 3:
order 2:
order 1:
order 0:
free_pages = 24576
test over

```

图 5-6 伙伴系统正确性的测试方法（左）与测试结果（右）

Figure 5-6 Test Methods (left) and Test Results (right) for the Correctness of Buddy System

如图 5-7 所示，对于 16MB 物理内存的申请和释放，修改每次请求的粒度（4096 次 4KB 请求、2048 次 8KB 请求.....32 次 512KB 请求），重复 200 次申请和释放过程，测试所需时间（以时钟滴答为单位）。对于传统链式分配器和伙伴系统分配器，在相同工作负载下分别测试，实验结果表明：（1）在 4KB 粒度下，传统链式分配器所需时间略短；（2）在 8KB 粒度下，两者耗时接近；（3）在 16KB 到 512KB 粒度下，

伙伴系统分配器耗时明显更短（512KB 粒度下，耗时是 3 ticks 和 22 ticks）；（4）伙伴系统分配器耗时随着请求粒度的翻倍呈现明显的减半趋势，体现了伙伴系统对于大量连续内存空间管理的高效性。

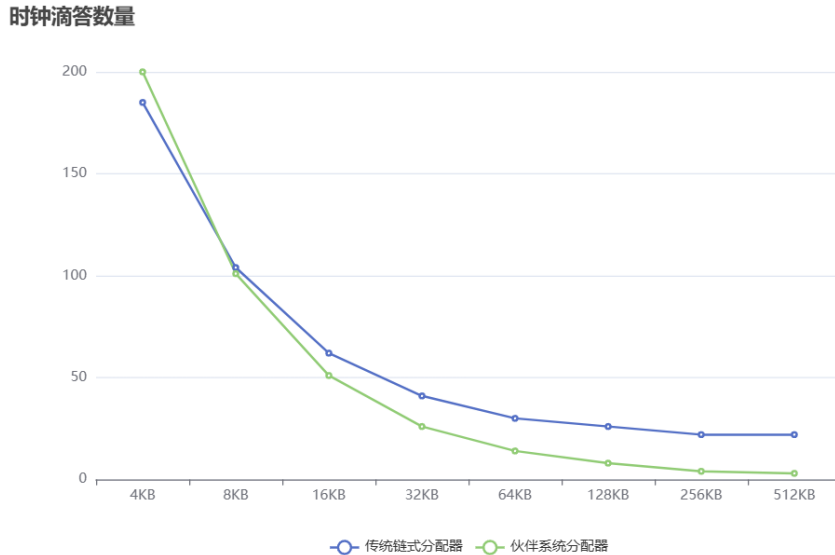


图 5-7 两种内存分配器以不同的页面粒度分配 16MB 内存所需的时钟滴答数（循环 200 次）

Figure 5-7 The Number of Clock Ticks Required by Two Memory Allocators to Allocate 16MB of Memory with Different Page Granularities (200 Iterations)

5.3.2 文件映射测试

如图 5-8 所示，文件映射的正确性测试方法如下：（1）创建一个测试文件，通过文件映射方法写入 4096 个字符 ‘A’。（2）通过传统方法读出测试文件的内容，检查内容是否符合预期。（3）通过传统方法追加写入 4096 个 ‘B’。（4）通过文件映射方法读取文件内容，检查文件是否由 4096 个 ‘A’ 和 4096 个 ‘B’ 组成。测试结果符合预期，触发了三次缺页异常并正确处理。

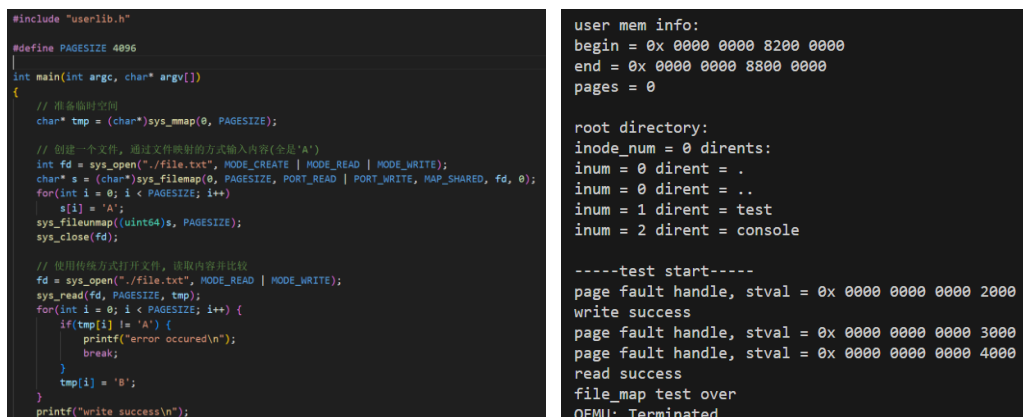


图 5-8 文件映射正确性的测试方法（左）与测试结果（右）

Figure 5-8 Testing Methods (left) and Testing Results (right) for the Correctness of File Mapping

时钟滴答数量

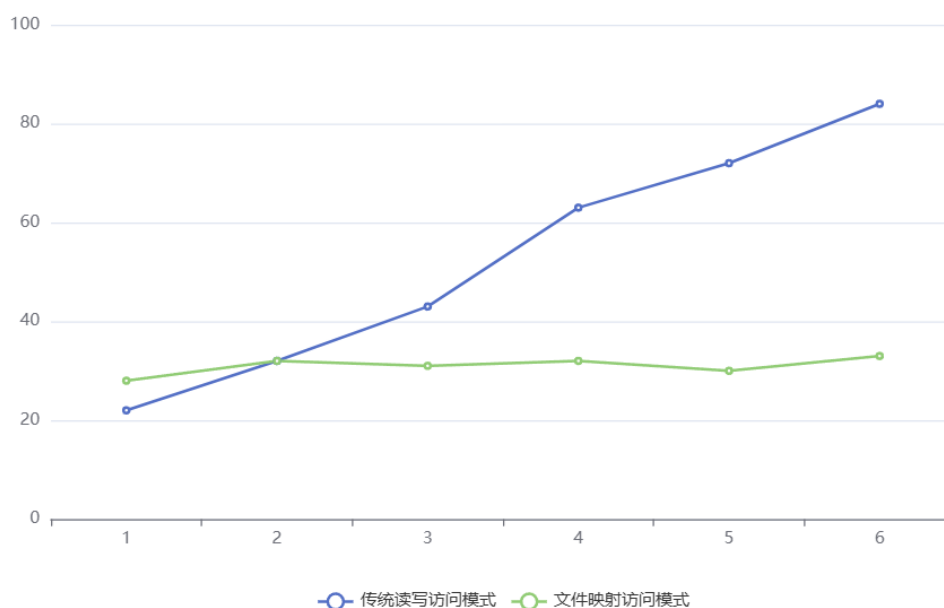


图 5-9 两种文件访问模式循环读写 N 次文件所需的时钟滴答数量

Figure 5-9 The Number of Clock Ticks Required for Two File Access Modes to Read and Write a File N Times in a Loop

文件访问效率取决于收益和开销谁占主导：（1）当文件映射带来的缺页异常处理开销大于内存拷贝减少和系统调用减少的收益时，传统读写访问模式时延更低。（2）反之，当收益大于开销时，文件映射访问模式时延更低。图 5-9 的测试显示了两种访问模式适合的工作场景：对于只访问 1 次的文件，使用传统读写访问模式更合适；对于一段时间内经常访问的文件，使用文件映射访问模式更合适。两种访问模式形成互补关系，为用户程序提供更丰富、更高效的文件访问方法。

5.3.3 多级反馈队列调度算法测试

如图 5-10 所示，用户态测试程序创建了 3 个死循环进程，用于测试长期工作情况下 MLFQ 调度算法的正确性和稳定性。当处于 `running` 状态的进程消耗完分配的时间片后，时钟中断处理程序输出时间片耗尽的提示信息和 MLFQ 队列信息，随后启动调度器。测试输出表明：（1）3 个进程诞生时进入 `level-0` 队列；轮流消耗掉 1 个时间片后进入 `level-1` 队列；轮流消耗掉 2 个时间片后进入 `level-2` 队列。（2）由于 `level-2` 是最低级的队列，同时没有新的进程进入，MLFQ 算法退化为时间片设为 4 的 RR 算法。（3）一段时间后，重启计时器触发，时钟中断程序调用 `MLFQ_reset` 函数，将所有 `runnable` 进程放入 `level-0` 队列，新一轮循环从此开始。测试结果与预期符合，调度器可以长时间稳定工作。

```
int main(int argc, char* argv[])
{
    int pid = sys_fork();
    if(pid == 0) {
        while(1);
    } else {
        int npid = sys_fork();
        if(npid == 0) {
            while(1);
        } else {
            while(1);
        }
    }
    return 0;
}
```

```
proc-2 run out time 4!
MLFQ info:
level 0:
level 1:
level 2: pid = 3 pid = 4

proc-3 run out time 4!
MLFQ info:
level 0:
level 1:
level 2: pid = 4 pid = 2

MLFQ reset!

proc-4 run out time 4!
MLFQ info:
level 0: pid = 2 pid = 3
level 1:
level 2:

proc-2 run out time 1!
MLFQ info:
level 0: pid = 3
level 1:
level 2: pid = 4
```

图 5-10 MLFQ 算法正确性的测试方法（左）与测试结果（右）

Figure 5-10 Testing Methods (left) and Testing Results (right) for the Correctness of MLFQ Algorithm

5.4 小结

本章的主要工作是测试与评估本文实现的小型操作系统内核，测试分为三类：

（1）过程性测试：在内核编写过程中测试函数和模块；（2）系统调用测试：内核编写完成后由用户程序发出系统调用请求；（3）高级机制测试：对伙伴系统、文件映射、多级反馈队列调度算法进行功能和性能测试。大量测试结果表明，本文实现的内核在功能、性能、高级特性等方面均达到预期。

6、总结与展望

6.1 工作总结

本文工作包括三部分：（1）参考 xv6 和 Linux，基于 QEMU 模拟环境和 RISC-V 体系结构，从零开始编写一个小型操作系统内核（代码量大约 6000 行）。（2）将内核编写过程拆分和记录，形成一套操作系统课程实验指导，包括代码、文档、测试等。（3）在基础版本内核上做进一步扩展和优化，针对三个核心模块增加高级特性和实用机制，在功能和性能方面取得显著突破，形成进阶版本内核。

工作 1 的动机出于对操作系统技术的热爱，希望能全面系统性了解和实现一个自己的内核；工作 2 的动机出于助教工作的需要，希望能将编写内核的经验推广开来，吸引更多有热情的初学者投入操作系统内核研究；工作 3 的动机出于研究 Linux 的需要，通过在小型内核中实现 Linux 的高级技术，深化对工业级内核的理解。

从零开始编写内核的各阶段代码都有线上仓库存档：<https://gitee.com/HaoDong-Xia/ecnu-oslab-answer>，其中 lab-1 到 lab-9 对应基础版本，version-1.0 到 version-1.3 对应进阶版本，未来会持续更新。九次实验的代码、文档、测试也有线上仓库存档供操作系统初学者查阅：<https://gitee.com/HaoDong-Xia/ecnu-oslab>。

6.2 未来展望

作为一个初级的小型内核，本文的工作存在一些改进空间：（1）内核基于 QEMU 模拟器运行，计划移植到真实的 RISC-V 开发板上，实现虚拟硬件和物理硬件的全面支持。（2）内核目前支持 20 多个系统调用，计划扩展到 70 个左右，以支持真实的 Linux 应用程序（如 libc、git、gdb、busybox 等）。（3）本文已经实现了 3 个 Linux 高级机制，计划将内核作为一个试验场，验证和学习更多 Linux 的机制。（4）计划增加更丰富更全面的测试用例，保障内核的可靠性和健壮性。

总而言之，未来工作展望聚焦四个方向：增加硬件支持（真实性）、增加系统调用数量（可用性）、增加 Linux 机制（前沿性）、增加测试用例（可靠性）。

参考文献

- [1] 熊谱翔,全召. RT-Thread Smart 微内核操作系统概述[J]. 单片机与嵌入式系统应用, 2021,21(3):9-12,17.
- [2] 易佳佳. 面向微内核的操作系统构建技术研究[D]. 四川:电子科技大学,2022.
- [3] 崔依婷. 基于 LoongArch 体系架构的操作系统设计与实现[D]. 北京:北京交通大学,2023.
- [4] MIT PDOS Group. xv6: a simple, Unix-like teaching operating system [E].
<https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf>. 2020.
- [5] Li Q, Xu W, Chen S, et al. Porting and Enhancing the XV6 Kernel for LoongArch Architecture Elevating Functionality: A Unified Approach to Kernel Porting and Enhancement[C]//Proceedings of the 2023 International Conference on Electronics, Computers and Communication Technology. 2023: 1-5.
- [6] Wen E, Ma S, Denny P, et al. KernelVM: Teaching Linux Kernel Programming through a Browser-Based Virtual Machine[C]//Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1. 2025: 1204-1210.
- [7] Nadi S, Holt R. The Linux kernel: A case study of build system variability[J]. Journal of Software: Evolution and Process, 2014, 26(8): 730-746.
- [8] Passos L, Queiroz R, Mukelabai M, et al. A study of feature scattering in the linux kernel[J]. IEEE Transactions on Software Engineering, 2018, 47(1): 146-164.
- [9] Panter S, Eisty N. Rusty linux: Advances in rust for linux kernel development[C]//Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 2024: 496-502.
- [10] Wang Z, Guang Y, Chen Y, et al. {SeaK}: Rethinking the Design of a Secure Allocator for {OS} Kernel[C]//33rd USENIX Security Symposium (USENIX Security 24). 2024: 1171-1188.
- [11] Nieh E, Zhang Z, Nieh J. ezFS: A Pedagogical Linux File System[C]//Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1. 2025: 861-867.
- [12] 薛峰. Linux 内核伙伴系统分析[J]. 计算机系统应用,2018,27(1):174-179.
DOI:10.15888/j.cnki.csa.006177.
- [13] Yu Z, Yang C, Zhang R, et al. Wear-leveling-aware buddy-like memory allocator for persistent memory file systems[J]. Future Generation Computer Systems, 2024, 150: 37-48.

- [14] Marotta R, Ianni M, Pellegrini A, et al. NBBS: A non-blocking buddy system for multi-core machines[J]. IEEE Transactions on Computers, 2021, 71(3): 599-612.
- [15] Crotty A, Leis V, Pavlo A. Are you sure you want to use mmap in your database management system?[C]//CIDR. 2022.
- [16] Korb I, Kotthaus H, Marwedel P. mmapcopy: efficient memory footprint reduction using application knowledge[C]//Proceedings of the 31st Annual ACM Symposium on Applied Computing. 2016: 1832-1837.
- [17] Miller S, Kumar A, Vakharia T, et al. Enoki: High velocity linux kernel scheduler development[C]//Proceedings of the Nineteenth European Conference on Computer Systems. 2024: 962-980.
- [18] Lawall J, Nishimura K, Lozi J P. Should we balance? towards formal verification of the linux kernel scheduler[C]//International Static Analysis Symposium. Cham: Springer Nature Switzerland, 2024: 194-215.
- [19] Iqbal M S, Chen C. P4-MLFQ: A P4 implementation of Multi-level Feedback Queue Scheduling Using A Coarse-Grained Timer for Data Center Networks[C]//2023 IEEE 12th International Conference on Cloud Networking (CloudNet). IEEE, 2023: 120-125.
- [20] Chen T, Li W, Sun Y K, et al. M-DRL: Deep reinforcement learning based coflow traffic scheduler with MLFQ threshold adaption[C]//IFIP International Conference on Network and Parallel Computing. Cham: Springer International Publishing, 2020: 80-91.

附录

由于内核实现的代码量较大，不适合粘贴到附录，具体代码请参考线上开源仓库：

- 内核实现: <https://gitee.com/HaoDong-Xia/ecnu-oslab-answer>
- 实验设计: <https://gitee.com/HaoDong-Xia/ecnu-oslab>

致谢

在华东师范大学计算机科学与技术学院学习的四年时间，给我留下了深刻的印象。在内核实现和论文撰写过程中，我得到了许多老师和同学们的帮助与支持，值此论文完成之际，特向他们表示由衷的感谢。

首先感谢我的父母，他们不辞辛苦将我抚养成人，在本科期间给了我许多帮助和鼓励，是我最坚实的后盾，也是我前行的动力。其次要感谢我的指导老师石亮教授，他是领我进入操作系统领域学习的引路人，也是我未来科研工作的指导者，内核实现和论文撰写过程都离不开石老师的严格要求和悉心指导，每次与石老师的讨论都令我受益匪浅。最后要感谢实验室的师兄师姐们，当我遇到困难和挫折时，为我提供了科学有效的指导，帮助我解决问题和优化设计，期待实验室永远保持温馨互助的氛围。

愿我在未来的研究生生涯中，不忘初心，砥砺前行，为国产操作系统领域蓬勃发展贡献自己的力量。