**编号**

JNDX_M

**本科生毕业设计（论文）**

**题目：** 自制操作系统的设计与实现

理 学院 信息与计算科学 专业

学 号 1131210333

学生姓名 胡涛涛

指导教师 陈蕾

讲师

二〇 二五 年 二 月

设计总说明

SpiderOS 是基于 x86 架构平台独立开发设计的轻量级教学操作系统，主要面向学习研究与实验验证操作系统基本原理与模块实现。该系统设计任务来源于高校操作系统课程的实践要求，旨在通过亲手实现一套完整的小型操作系统，深入掌握内核结构、进程调度、内存管理、文件系统、设备驱动等核心技术，并提升开发者对操作系统体系结构的理解与动手能力。

本系统遵循模块化、层次化的设计标准，设计结构划分清晰、功能模块边界明确，整个操作系统从启动加载到内核运行，最终完成用户交互，均按标准的软硬件交互规范展开。SpiderOS 的整体架构包括：引导加载模块、内存管理模块、线程与进程调度模块、文件系统模块、系统调用接口、I/O设备驱动模块、用户 Shell 命令解析模块。各模块相互独立，接口规范，且便于维护与扩展，能够良好支撑后续功能完善与特性增强。

在设计原则方面，SpiderOS 注重简洁性、实用性、可扩展性三者的结合。首先，系统实现的所有功能均遵循“简洁有效、功能闭环”的原则，避免无意义的代码膨胀，关注关键特性落地。例如，内存管理采用页表结合位图机制，既方便物理内存的高效分配，又能保障虚拟内存地址空间的安全性；线程调度使用时间片轮转结合阻塞唤醒机制，满足基本的多任务调度需求；文件系统采用扇区读取、分区扫描、路径解析等底层实现，实现用户对文件与目录的基本操作。其次，SpiderOS 强调实用性，系统实现过程中充分考虑实验环境的兼容性，支持在 Bochs、QEMU 等虚拟机平台稳定运行，具备基础进程创建与切换、文件存储、用户命令输入等实际操作能力。最后，SpiderOS 保持良好的可扩展性，所有模块的接口设计均留有拓展空间，便于后续添加用户态程序加载、多级目录支持、复杂的文件访问权限控制等功能。

整个开发流程从硬件引导逻辑开始，手动编写 MBR 和 Loader，完成从 BIOS 实模式到内核保护模式的转换，接着进行分页机制初始化、内存池划分、虚拟地址位图构建等内存管理操作。在线程与进程调度上，实现了通过双向链表管理的就绪队列和阻塞队列，结合主动让出和中断驱动的时间片轮转，实现多任务公平调度。文件系统模块完成了硬盘分区扫描、文件读写、目录解析、文件描述符管理等功能，用户通过 Shell 命令行可以完成目录切换、新建文件、查看目录列表等基本操作。

**关键词：** 操作系统；x86架构；内存管理；线程调度；文件系统

General description of the design

SpiderOS is a lightweight teaching operating system developed and designed independently based on x86 architecture platform, which is mainly oriented to learning research and experimental verification of operating system basic principles and module implementation. The design task of this system comes from the practical requirements of operating system courses in colleges and universities, aiming at realizing a complete set of small-scale operating system, in-depth mastery of core technologies such as kernel structure, process scheduling, memory management, file system, device driver, etc., as well as improving the developers' understanding of the operating system architecture and hands-on ability.

The system follows the modularization and hierarchization design standard, with clear design structure division and clear boundary of functional modules, and the whole operating system from boot loading to kernel running, and finally completing the user interaction, all of them are carried out in accordance with the standard hardware and software interaction specification.The overall architecture of SpiderOS includes the following modules: boot loading module, memory management module, thread and process scheduling module, file system module, system call interface, I/O device driver module, and device driver module. The overall architecture of SpiderOS includes: boot load module, memory management module, thread and process scheduling module, file system module, system call interface, I/O device driver module, and user shell command parsing module. The modules are independent of each other, with standardized interfaces and easy to maintain and extend, which can well support the subsequent function improvement and feature enhancement.

In terms of design principles, SpiderOS focuses on the combination of simplicity, practicality and scalability. First of all, all the functions realized by the system follow the principle of “simplicity and effectiveness, functionality and closed-loop”, avoiding meaningless code expansion and focusing on the implementation of key features. For example, memory management adopts page table combined with bitmap mechanism, which not only facilitates efficient allocation of physical memory, but also guarantees the security of virtual memory address space; thread scheduling uses time slice rotation combined with blocking wake-up mechanism to meet the basic multi-task scheduling needs; file system adopts sector reading, partition scanning, path parsing and other underlying implementations, which realizes the user's basic operations on files and directories. Secondly, SpiderOS emphasizes the practicality, and in the process of system implementation, the compatibility of the experimental environment is fully taken into account, supporting the stable operation in Bochs, QEMU and other virtual machine platforms, and possessing the ability of basic process creation and switching, file storage, and user command input and other practical operations. Finally, SpiderOS maintains good scalability, and the interface design of all modules leaves room for expansion, so that it is easy to add user-state program loading, multi-level directory support, complex file access rights control and other functions.

The whole development process starts from the hardware boot logic, manually writing MBR and Loader, completing the conversion from BIOS real mode to kernel protected mode, and then initializing the paging mechanism, dividing the memory pool, constructing the virtual address bitmap and other memory management operations. In terms of thread and process scheduling, it realizes the ready queue and blocking queue managed by bi-directional chain list, and combines the active yield and interrupt-driven time-slice rotation to realize the fair scheduling of multi-tasks. The file system module accomplishes hard disk partition scanning, file reading and writing, directory parsing, file descriptor management, etc. Users can complete basic operations such as directory switching, new file creation, and viewing directory listings through the Shell command line.

Keywords: operating system; x86 architecture; memory management; thread scheduling; file system

**Keywords**: operating system; x86 architecture; memory management; thread scheduling; file system

目录

[设计总说明 2](#_Toc786238536)

[General description of the design 3](#_Toc394075144)

[第1章 绪论 8](#_Toc261325408)

[1.1 研究背景与意义 8](#_Toc788476736)

[1.2 国内外研究现状分析 8](#_Toc1589813888)

[1.2.1 微内核技术的发展 8](#_Toc56539121)

[1.2.2 国内外微内核操作系统 9](#_Toc1565914770)

[1.3 本文研究的主要内容 10](#_Toc2120707506)

[第2章 操作系统运行环境概述 11](#_Toc108719525)

[2.1 x86架构计算机硬件简述 11](#_Toc814818639)

[2.1.1 CPU与总线 11](#_Toc2101176943)

[2.1.2 内存结构   11](#_Toc1676743072)

[2.1.3 I/O设备模型   12](#_Toc876787130)

[2.2 操作系统的基本特征   12](#_Toc334863736)

[2.2.1 并发性与异步性   12](#_Toc498022419)

[2.2.2 共享性与虚拟性   12](#_Toc1704095738)

[2.3 操作系统的核心功能   12](#_Toc201365908)

[第3章 系统结构设计与实现 12](#_Toc141830536)

[3.1 运行环境简述 12](#_Toc681216908)

[3.1.1试验平台概述 12](#_Toc1960683282)

[3.1.2 编译工具链与开发环境 13](#_Toc477488761)

[3.2 引导程序实现 13](#_Toc402467282)

[3.2.1 引导程序简述 13](#_Toc50080212)

[3.2.1 启动加载器设计 13](#_Toc206063711)

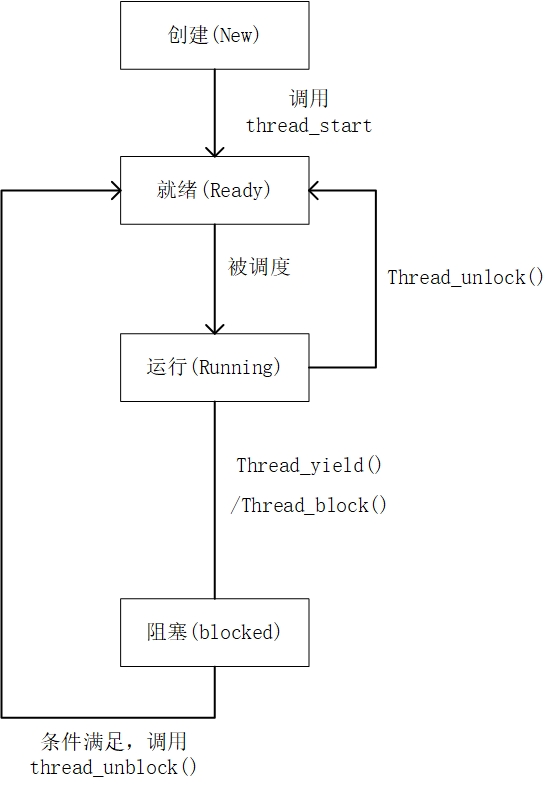
[3.1.2 从实模式到保护模式 14](#_Toc1207931568)

[3.3 内核核心模块设计与实现 14](#_Toc63302810)

[3.3.1 中断管理机制 14](#_Toc128465452)

[3.3.2 内存管理系统 15](#_Toc995201964)

[3.3.3 线程与进程调度机制设计 17](#_Toc855184311)

[ 18](#_Toc6987737)

[3.3.4 输入输出系统 18](#_Toc1651782868)

[3.3.5 用户进程 19](#_Toc1641422847)

[3.3.6 文件系统 19](#_Toc401062882)

[3.3.7 系统交互 20](#_Toc1913108276)

[第4章 SpiderOS功能验证与测试 22](#_Toc282415936)

[4.1 启动加载与保护模式切换验证 22](#_Toc1990876770)

[4.1.1 MBR与Loader的引导加载测试 22](#_Toc1969647397)

[4.1.2 实模式到保护模式切换验证 22](#_Toc1848330706)

[4.1.3 内核elf加载与内存映射测试 22](#_Toc1964100628)

[4.2 内存管理模块功能测试 22](#_Toc2078366923)

[4.2.1 页表建立与地址转换验证 22](#_Toc515665697)

[4.2.2 内存位图分配机制正确性测试 22](#_Toc1917793923)

[4.2.3 用户空间动态分配（malloc/free）测试 22](#_Toc1607626347)

[4.3 线程与进程调度机制测试 22](#_Toc1392452827)

[4.3.1 主线程与子线程创建测试 22](#_Toc105174011)

[4.3.2 阻塞与唤醒机制测试 22](#_Toc2105648766)

[4.3.3 进程创建fork与程序替换exec测试 22](#_Toc949064918)

[4.4 文件系统功能测试 22](#_Toc306539920)

[4.4.1 分区扫描与挂载测试 22](#_Toc99995654)

[4.4.2 文件读写与创建删除测试 22](#_Toc1630281826)

[4.4.3 路径解析与目录管理测试 22](#_Toc119739554)

[4.5 系统调用接口测试 22](#_Toc577484415)

[4.5.1 文件操作相关系统调用测试 22](#_Toc2032749108)

[4.5.2 内存分配与释放系统调用测试 22](#_Toc169819767)

[4.5.3 进程调度相关系统调用测试 23](#_Toc783548126)

[4.6 Shell命令解释器功能测试 23](#_Toc1093197029)

[4.6.1 内建命令（如ls、cd、mkdir、rmdir）执行测试 23](#_Toc233122577)

[4.6.2 用户输入解析与指令调度测试 23](#_Toc912013578)

[4.6.3 文件系统相关命令的正确性验证 23](#_Toc2088398993)

[4.7 I/O设备驱动测试 23](#_Toc1088306888)

[4.7.1 硬盘驱动读写功能验证 23](#_Toc919001316)

[4.7.2 键盘输入中断响应测试 23](#_Toc1592698213)

[4.7.3 控制台输出打印完整性测试 23](#_Toc582246088)

[第5章 总结与展望   23](#_Toc1320064198)

[5.1 研究总结   23](#_Toc1358322841)

[5.2 系统不足与后续改进  23](#_Toc864662024)

[23](#_Toc1163457320)

[参考文献（不少于15篇，其中至少5篇英文） 24](#_Toc1180486590)

[致 谢 25](#_Toc565509082)

[附录B： XX 28](#_Toc980074300)

第1章 绪论

1.1 研究背景与意义

操作系统（Operating System，简称OS）作为计算机系统架构中最核心的软件之一，承担着硬件资源的统一管理、任务调度、设备驱动、文件系统支持以及用户与硬件的桥梁作用。在现代计算机体系结构中，操作系统的稳定性与可靠性直接决定了系统的整体性能和用户体验。

随着计算机科学与技术的飞速发展，主流的操作系统如Windows、Linux和macOS已变得日益复杂，功能愈发完善，但相应的源码体量和系统结构也愈发庞大。在1991年第一个发布的相对完整的0.11版本的Linux内核大小不到200K，而如今最新的Linux内核4.20.13压缩包大小约100M。[1]足足增加了500倍，也侧面提现了设备的负责度和种类增长速度。而如今我们在看较出名的几款微内核大小，sel4微内核下载宝约有4M左右；hurd微内核大小约有3M左右；mach微内核大小约有4M左右，以及谷歌最新启动的全新操作系统fuchsia开发项目所使用的微内核zircon的下载压缩包大小为9M左右。对比之下我们可以看出无论是哪个内核，在学习上的成本都是巨大的。

因此，通过自主设计和实现一个简易的操作系统，能够让学习者真正从零起步，掌握操作系统的核心架构和运行机制。

本课题《自制操作系统的设计与实现》正是基于这一需求展开，目的是通过构建一个可独立运行的简易操作系统，在实践中深入理解操作系统的引导加载机制、内核设计思想、进程调度策略、内存管理方法、中断响应机制以及文件系统的设计原则。通过亲自实现操作系统各功能模块，能有效巩固理论课程中抽象的概念，加深对计算机体系结构和操作系统原理的认识。

同时，操作系统的开发不仅要求扎实的底层编程能力，还需要具备模块化设计思维和良好的代码规范。在构建过程中，开发者将面对模块划分、接口设计、数据结构选择、资源同步与互斥等关键问题，提升对工程项目开发流程的整体把控能力。且通过实际上手开发，更深入的理解软硬件写作的关系，掌握从Bootloader到用户态的完整执行流程

自制操作系统的实现过程不仅具有重要的学术研究价值，而且对未来从事嵌入式开发、系统级软件开发以及安全方向研究都具有非常积极的促进作用。因此，本课题的研究对培养打算从事计算机专业的开发者的综合开发能力、实践创新能力和系统架构思维，具有重要的理论意义和现实应用价值。

1.2 国内外研究现状分析

1.2.1 微内核技术的发展

在操作系统的发展历程中，微内核因其模块化、高安全性和可维护性的优势，逐渐成为研究和应用的热点。

20世纪80年代中期，卡内基梅隆大学开发了Mach操作系统，标志着第一代微内核的诞生。Mach旨在将传统操作系统中的服务模块移至用户态，以实现更高的模块化和灵活性。然而，由于当时的硬件性能限制，Mach在进程间通信（IPC）方面存在较高的开销，导致系统整体性能不佳。尽管如此，Mach在多线程支持和多处理器架构方面的探索为后续微内核的发展奠定了基础。这也是第一代微内核的创新性尝试。

第二代微内核针对第一代微内核在性能上的不足，德国计算机科学家Jochen Liedtke在1990年代初提出了L4微内核。L4通过极简化内核功能，优化了IPC机制，显著提高了系统性能。具体而言，L4采用寄存器传递消息的方式，减少了内存访问的开销，使得IPC速度比Mach快了20倍以上。此外，L4的内核体积仅为12KB，远小于Mach的300KB，体现了微内核在性能和资源占用上的优势。

随着信息安全需求的增加，第三代微内核在保持高性能的同时，更加注重系统的安全性。seL4是这一阶段的代表，它在L4的基础上引入了能力空间（capability space）机制，确保进程只能访问其被授权的资源，从而增强了系统的安全性。此外，seL4是第一个经过形式化验证的微内核，其8700行的C代码通过数学方法验证了内核的正确性和安全性，广泛应用于航空航天、汽车电子等对安全性要求极高的领域[2]。

1.2.2 国内外微内核操作系统

在国际上，经典的操作系统研究起步较早。早在1960年代，IBM公司推出了世界上第一个多任务分时系统——CTSS，为现代操作系统的进程管理与时间片调度奠定了理论基础。随后，UNIX操作系统的诞生与开源，推动了Linux、FreeBSD等开源内核的发展，形成了如今开放性强、稳定性好、广泛应用于服务器、嵌入式和云计算领域的Linux生态体系。

近十年来，随着嵌入式设备与移动计算需求的崛起，轻量化操作系统的研究日益活跃。开源社区中出现了如FreeRTOS、Zephyr等适配物联网设备的微内核系统。这些项目注重裁剪功能、提升稳定性，成为嵌入式开发和实时控制系统的良好范例。同时，Google推出的Fuchsia OS基于全新的Zircon微内核架构，打破了传统UNIX的框架思路，展示了未来操作系统设计的新趋势。

在国内，操作系统的研究与实践同样蓬勃发展。早期代表性的工作有中科院计算技术研究所的“银河操作系统”与“普华Linux”；近年来，随着国产芯片与信创产业政策的推动，涌现出了如麒麟（Kylin）、UOS（统一操作系统）、鸿蒙OS等面向桌面、移动和物联网场景的操作系统。这些项目致力于提升国产软硬件的自主可控能力，强化对安全性、兼容性和生态建设的支持。

从教学与科研角度，许多高校也逐渐采用类似的实践教学方案，例如南京大学的NEMU模拟器，鼓励学生通过亲自编写和调试操作系统模块，培养软硬件协同设计能力。操作系统的学习已从单纯的理论课程逐渐拓展为项目驱动式学习。MIT开源的“xv6”操作系统、Harvard的“JOS”教学操作系统，都是国外高校广泛使用的简易操作系统教学范例。通过实现简化版的内核，帮助学生掌握虚拟内存管理、系统调用、文件系统、进程调度等核心知识。

1.3 本文研究的主要内容

本文围绕基于x86架构平台的简易操作系统设计与实现展开研究，结合实际开发实践，深入分析操作系统从引导加载、内核初始化、内存管理到文件系统的完整运行流程，最终完成了一个具备基本功能的自制操作系统。本文的主要研究内容包括以下几个方面：

（1）**操作系统引导与启动机制**  
本系统从裸机环境出发，首先编写了主引导记录（MBR）与加载器（Loader）。通过MBR完成硬盘引导区的读取与内存中的加载流程，Loader则负责进入保护模式并初始化GDT表，最终将内核正确加载到指定内存位置并跳转执行，完成从硬件引导到内核启动的关键环节。

（2）**内核初始化与中断机制设计**  
本文实现了包括全局描述符表（GDT）、中断描述符表（IDT）以及可编程中断控制器（PIC）的初始化配置，实现了软硬件中断的注册与管理，支持键盘输入、时钟中断等外设驱动的基本交互。此外，内核还设计了统一的中断服务框架，能够打印异常错误信息并保证系统稳定运行。

（3）**内存管理模块的实现**  
内存管理是操作系统资源管理的核心组成部分。本文设计了基于位图的物理内存分配算法，实现了页级内存分配与释放，支持内核线程与用户进程的虚拟地址空间管理，完成了动态内存申请（malloc）与释放机制，为进程和文件系统提供了可靠的内存支持。

（4）**线程与进程管理机制**  
系统实现了线程控制块（TCB）和进程控制块（PCB）的结构设计，完成了内核线程的创建、调度与切换功能。调度采用简单的优先级轮转调度算法，支持线程阻塞与唤醒机制，能够实现基本的线程同步。此外，用户进程通过系统调用接口完成从内核态向用户态的任务切换，并支持进程的fork复制与exec加载。

（5）**文件系统的设计与实现**  
文件系统模块支持磁盘分区的扫描与挂载，采用自制的简易文件系统格式，完成了文件的创建、读取、写入与删除操作，支持目录的创建与遍历，具备基本的层级文件目录结构。文件系统通过系统调用对上层提供接口，满足用户进程的文件操作需求。

（6）**系统调用接口设计**  
本文实现了从用户空间到内核空间的系统调用机制，支持常用系统调用如：进程控制、文件读写、内存分配、屏幕输出等，用户程序通过软中断调用内核功能，完成用户态与内核态的安全切换。

（7）**Shell命令行交互功能**  
为方便用户进行基本操作，本文设计了简易Shell命令行解释器，支持如：ls、pwd、mkdir、cd、rmdir、touch等基础命令，能够实现文件的创建、查看与路径切换，提升了系统的人机交互性。

通过上述各模块的有机组合，本文完成了一个从裸机引导到用户进程运行的完整操作系统框架，实现了多线程调度、磁盘读写、文件系统、系统调用、用户进程管理等基本功能，验证了操作系统核心理论的工程实现效果，达到了预期的研究目标。

1. 操作系统运行环境概述

2.1 x86架构计算机硬件简述

2.1.1 CPU与总线

中央处理器（Central Processing Unit, CPU）作为x86架构计算机的核心组件，承担着指令执行、数据处理及硬件协调的核心任务。在计算机系统中，CPU通过总线与其他硬件设备（如内存、外设控制器等）进行通信。总线作为各部件间传输数据、地址和控制信号的公共通道，其设计与性能直接影响系统的整体效率。

在x86架构中，总线按功能可分为数据总线、地址总线和控制总线。数据总线负责在CPU与内存或I/O设备之间传输数据，其宽度（如32位或64位）决定了单次传输的数据量。例如，32位数据总线可一次性传输4字节数据，显著提升批量数据处理的效率。地址总线用于指定内存或I/O端口的物理地址，其宽度直接决定了CPU的寻址能力。以32位地址总线为例，其最大寻址空间为 ,而64位地址总线则将寻址范围扩展至理论上的16 EB（艾字节）。控制总线则负责传输读写信号、中断请求、时钟同步等控制信息，协调各硬件模块的工作时序。

CPU与总线的交互流程主要包括指令获取、数据读写和中断处理三个阶段。当CPU需要执行指令时，首先通过地址总线发送指令所在的内存地址，控制总线发出“读”信号，内存接收到信号后通过数据总线将指令返回至CPU。在数据读写过程中，CPU根据指令要求，通过地址总线指定目标地址，并利用控制总线发出读写信号，数据总线则完成具体的数据传输。此外，外设（如键盘、硬盘）可通过控制总线向CPU发送中断请求，CPU响应后暂停当前任务，转而执行对应的中断服务程序，从而实现实时交互。

2.1.2 内存结构

内存是x86架构计算机中用于临时存储程序指令与数据的关键硬件组件，其结构与访问效率直接影响系统的整体性能。x86架构的内存系统采用分层设计，涵盖物理内存、虚拟内存以及多级缓存机制，以满足不同场景下的速度与容量需求。

物理内存通常由动态随机存取存储器（DRAM）构成，作为CPU直接访问的主要存储介质。其容量由地址总线宽度决定，例如32位系统支持最大4 GB物理内存，物理内存通过内存控制器与CPU连接，现代CPU通常将内存控制器集成于芯片内部（如Intel的集成内存控制器技术），通过专用总线（如DDR4/DDR5通道）实现高速数据传输，显著降低延迟并提升带宽。

2.1.3 I/O设备模型

输入输出（I/O）设备是计算机系统与外部环境交互的桥梁，其模型定义了设备与CPU、内存及其他组件间的通信机制。在x86架构中，I/O设备模型通过硬件接口、总线协议及操作系统驱动协同工作，确保高效的数据传输与设备控制。

2.2 操作系统的基本特征

2.2.1 并发性与异步性

2.2.2 共享性与虚拟性

2.3 操作系统的核心功能

第3章 系统结构设计与实现

3.1 运行环境简述

3.1.1试验平台概述

试验用主机配置：

|  |  |
| --- | --- |
| 架构: | x86\_64 |
| CPU: | 13th Gen Intel(R) Core(TM) i7-13700 |
| CPU核心数： | 24核 |
| 内存: | 24GB |
| 操作系统 | Ubuntu 24.04.2 LTS |

模拟运行环境：

Bochs是一款开源的x86架构仿真器，能够完整模拟硬件平台，包括CPU指令集、内存管理单元、I/O设备和硬盘控制器。因此，SpiderOS采用了Bochs虚拟机作为主要的开发调试平台。

3.1.2 编译工具链与开发环境

SpiderOS的开发目标平台为x86架构，宿主平台为Linux操作系统，主要工具链如下：

|  |  |
| --- | --- |
| 工具名称 | 作用 |
| nasm | 用于汇编 mbr.S、loader.S、kernel.S 等汇编代码 |
| GNU LD | 用于将编译完成的目标文件链接，设置内核入口地址 0xC0001500 |
| dd | 用于将MBR、Loader和内核映像写入虚拟硬盘镜像文件 |
| Make | 用于自动化构建流程，管理编译与打包过程 |

3.2 引导程序实现

3.2.1 引导程序简述

在传统的x86架构计算机中，操作系统的启动流程一般遵循“固件 → 引导程序 → 操作系统内核”的顺序。当计算机加电启动后，首先由主板BIOS完成硬件自检（POST）与初始化，然后根据启动设备顺序，读取启动介质（如硬盘、U盘）上的主引导扇区（MBR），将其加载到内存 0x7C00 处，并将控制权交由MBR中的引导程序负责后续的引导工作。

SpiderOS的引导程序遵循这一经典架构，由两部分组成：主引导记录（MBR）和加载器（Loader）。MBR的任务相对简单，主要负责调用BIOS中断功能，将位于磁盘后续扇区的Loader加载至内存 0x600 地址，并将控制权传递给Loader。随后，Loader完成更复杂的系统初始化工作，包括内存检测、保护模式切换、分页机制配置以及内核的加载与跳转。

对于操作系统开发而言，引导程序不仅仅是启动的入口，它还需要根据硬件环境进行适配，合理规划内存布局，并完成CPU从实模式到保护模式的平滑切换。不同操作系统会根据内核的需求，设计出各自独立的引导方案。例如，Windows操作系统通过 Bootmgr 完成内核引导，而Linux内核则通常依赖GRUB等通用引导器完成加载；而SpiderOS则采用了自研的MBR+Loader方案，完全由汇编语言实现，具备高度的可控性和透明性，能够针对操作系统内核结构和硬件特性灵活调整。

引导程序的设计不仅直接决定了内核加载是否可靠，还影响到操作系统的可移植性和启动安全性。SpiderOS采用的手写引导方案，明确遵循x86架构硬件规范，避开了复杂的多平台兼容性问题，确保内核能够稳定在BIOS传统引导链的环境中正确加载运行。

3.2.1 启动加载器设计

SpiderOS的启动加载器采用经典的两阶段设计，遵循x86架构的引导流程规范。该设计充分考虑了硬件环境约束和操作系统内核加载需求，实现了从BIOS控制权交接到内核初始化的完整过程，确保内核能够在保护模式和分页机制下稳定启动运行。

对于一般的计算机设备，加电后，CPU首先执行主板固化的BIOS代码，完成硬件自检与初始化，并根据设定的启动顺序。本设计采用Bochs环境模拟，首先创建空白镜像文件hd40M.img,将主引导扇区（MBR）加载至hd40M.img第一个扇区，随后为避免相互干扰或覆盖，确保系统的稳定性和数据安全，将 Loader 放在 LBA 2(即第三个扇区）。随后将kernel内核加载到第10个扇区 。(hd40M.img结构如图3-1)

hd40M.img

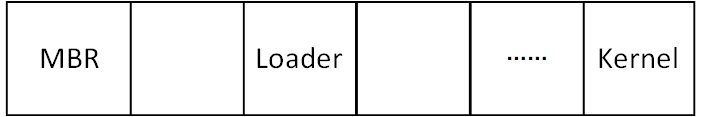


图 3-1

3.1.2 从实模式到保护模式

在x86架构的计算机中，CPU上电复位后默认处于实模式（Real Mode）。此模式的寻址方式基于段寄存器和偏移地址，最多只能访问1MB的物理地址空间，且无法使用现代操作系统所需的内存保护、多任务与分页机制。因此，操作系统必须在启动阶段将CPU从实模式切换至保护模式（Protected Mode），为内核的运行提供稳定的硬件基础。

SpiderOS的启动加载器（Loader）在完成内存检测后，按照标准的三步流程实现了实模式到保护模式的安全切换。其具体切换遵循MBR->Loader->Kernel流程。

首先BIOS会将**主引导扇区MBR**加载到内存的 0x7C00处，然后CPU跳转执行0x7C00处MBR的代码。经过一系列处理之后，会调用读硬盘函数 rd\_disk\_m\_16，将硬盘上的Loader程序加载到内存，并使用jmp跳转到Loader的内存地址，正式讲控制权交给Loader。

Loader内存地址通过宏LOADER\_BASE\_ADDR来控制，本SpiderOS中为0x900。它的任务比MBR复杂，首先会进行内存检测，检查机器支持的内存大小；然后第二步是设置全局描述附表（GDT），准备好段描述符表后，就可以为即将切换的保护模式配置好段基址与属性，最后切换到保护模式，完成Loader最重要的任务。在做完以上所有的工作之后，通过jmp指令跳转到内核main函数所在的地址，将控制权交给内核。从此进入C语言世界。

3.3 内核核心模块设计与实现

引导程序完成了将内核从磁盘加载到内存并完成实模式向保护模式的切换。引导程序执行结束后，CPU的控制权正式交由内核代码接管，SpiderOS内核的初始化阶段随之开始。内核作为操作系统的中枢，将持续负责处理用户进程的系统调用请求，协调硬件资源，保障系统的稳定运行。

3.3.1 中断管理机制

中断是操作系统内核与硬件通信的基础机制，SpiderOS设计的中断子系统，实现了从硬件层到内核层的完整中断响应框架，确保CPU在接收外设事件或异常情况时，能够及时、安全地切换至内核处理逻辑，保障系统的稳定运行。

x86架构为中断管理提供了硬件支持——**中断描述符表（IDT）**。IDT是一张表(结构如图3-2)，存储了129个中断向量，每个向量的内容是一个**中断门描述符**，用于告诉CPU：如果触发这个中断，应该跳转到哪个内存地址执行对应的服务程序。SpiderOS中使用结构体gate\_desc对这个表项进行了抽象，定义了中断服务函数的地址（低16位和高16位），段选择子，属性等信息。操作系统启动时，内核会首先准备好中断描述符表，填充每个表项的中断处理函数地址。这一过程通常由 idt\_desc\_init() 函数完成。填充完成后，通过 lidt 指令将IDT表的首地址和长度告诉CPU。此时CPU就具备了处理中断的能力。当硬件设备发出中断信号（例如键盘按键、硬盘读写完成等）时，CPU会根据中断向量自动查表，从IDT中取出对应的中断门描述符，跳转到指定的中断服务程序，通常由 intr\_entry\_table 汇编入口作为第一跳，保存上下文、调用C语言层的 general\_intr\_handler 统一处理。在这个过程中，CPU会自动完成堆栈切换，保护现场，防止中断程序破坏用户程序的状态。

gate\_desc数据结构图 3-2

SpiderOS的中断框架提供了中断注册接口 register\_handler()，允许内核和驱动程序将具体的处理逻辑与中断向量绑定。例如，键盘驱动会将 keyboard\_handler() 注册到键盘中断的向量上，硬盘驱动会将 intr\_hd\_handler() 注册到硬盘中断向量。当硬件事件发生，用户代码不需要干预，内核就会自动进入已注册的服务程序中，完成硬件与内核的数据交互。为了保证并发安全，SpiderOS还设计了 intr\_disable() 和 intr\_enable() 两个函数用于临界区的中断屏蔽。关闭中断后，内核可以在修改关键数据结构时防止被中断打断，避免竞态条件，处理完毕再恢复中断。

3.3.2 内存管理系统

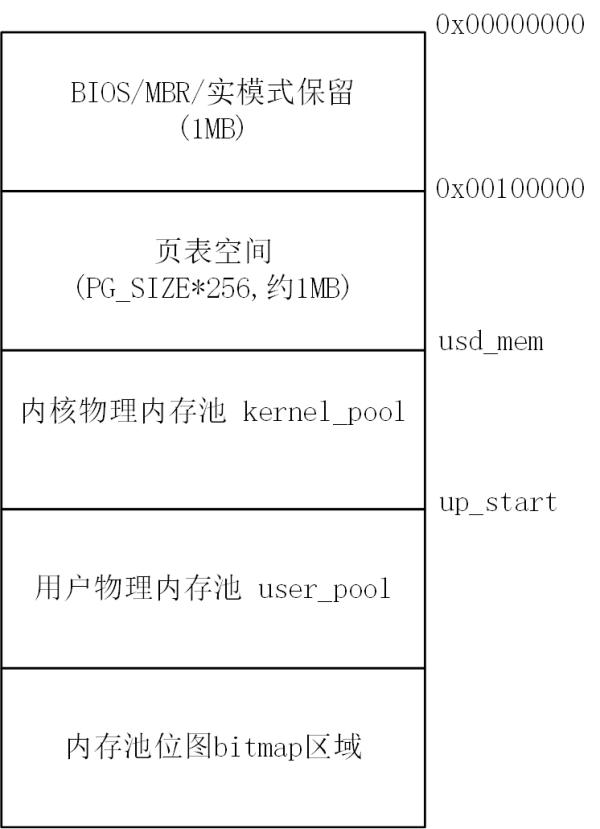
在 SpiderOS 的设计中，内存管理模块承担着至关重要的角色。它不仅关系到内核与用户程序的稳定运行，同时也是操作系统资源分配与隔离机制的核心基础。内存的初始化与管理，直接决定了操作系统的可靠性与扩展性。

在系统上电完成引导加载，进入内核阶段并初始化中断之后，SpiderOS 会首先调用 mem\_init() 函数启动内存管理系统的初始化。该函数通过读取 BIOS 预存放在 0x800 地址的内存总量，确定系统实际可用的物理内存容量，并将其传递给 mem\_pool\_init() 函数。

mem\_pool\_init() 是内存池的核心初始化逻辑。它首先根据页表的结构，计算出用于分页机制的页表空间大小，默认将 0x100000（低端1MB）以及页表所需空间 256 \* 4KB 保留给内核管理。这部分空间由页表所占据，操作系统无法直接调度。

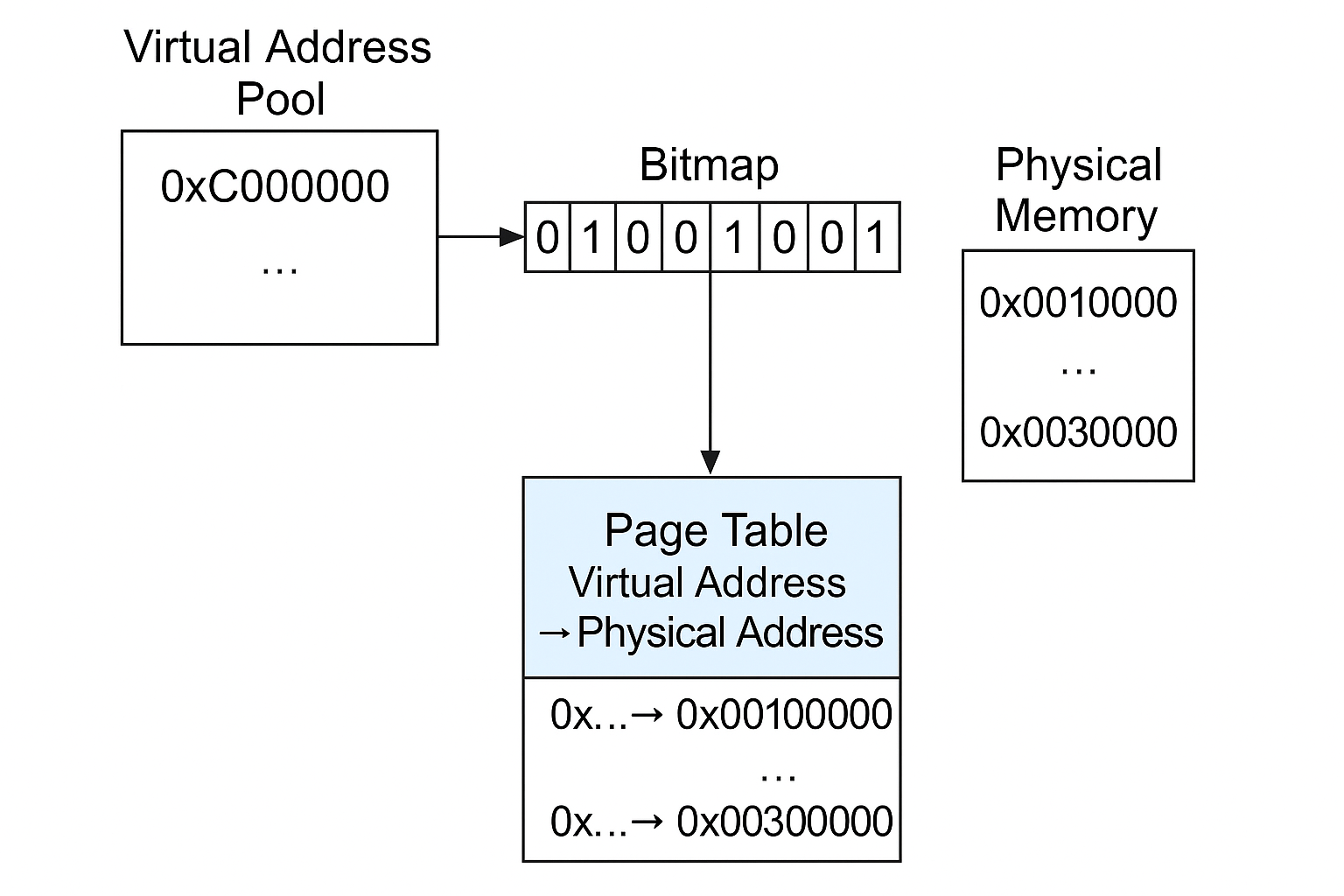
剩余的空闲物理内存则被划分为两个部分：一部分作为内核物理内存池（kernel\_pool），另一部分作为用户物理内存池（user\_pool），二者各占总空闲内存的大致一半。为了有效管理物理页的分配，SpiderOS 使用位图（Bitmap）作为标记机制。每一个比特位表示一页物理内存的占用状态：0 表示空闲，1 表示已分配。

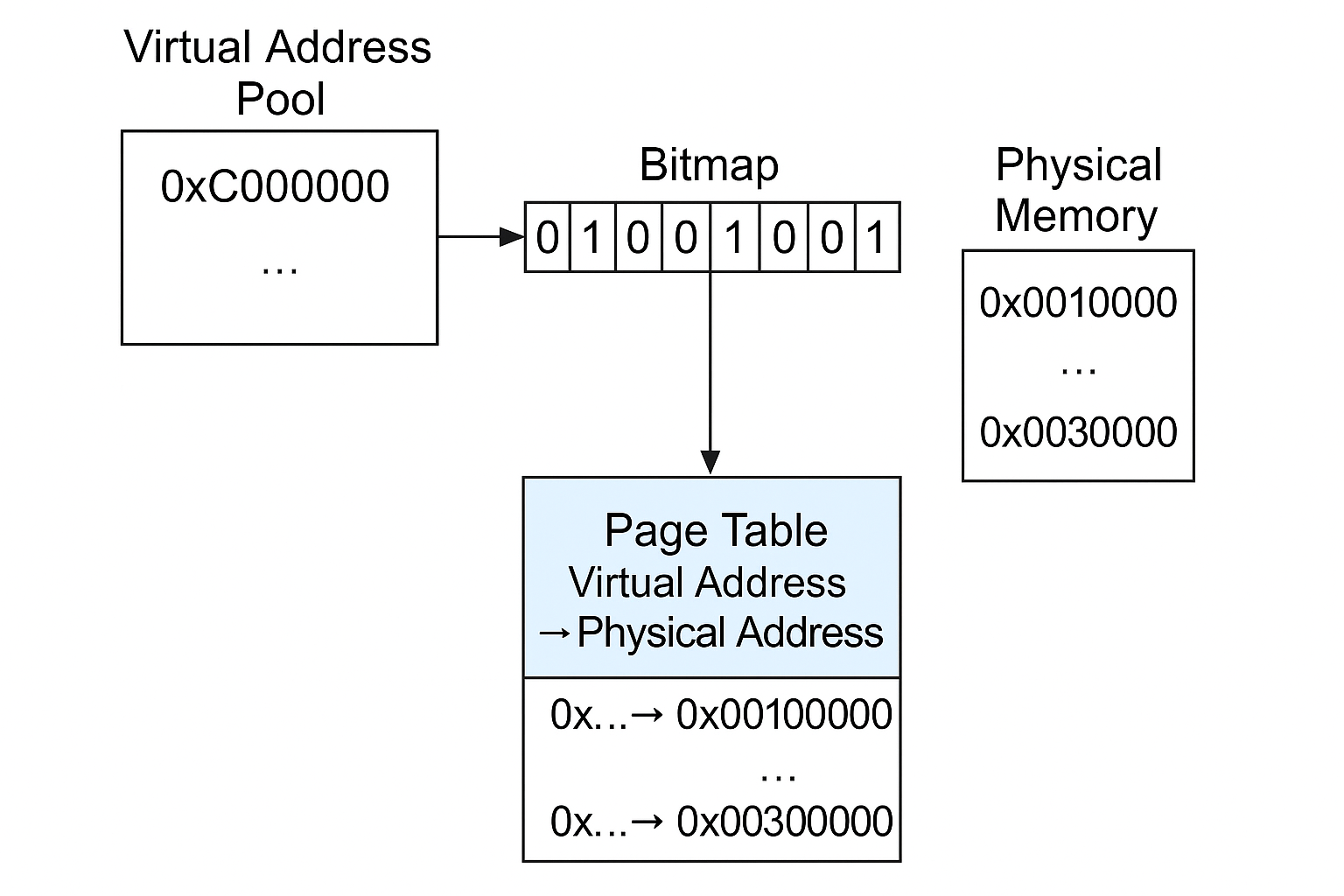
内核和用户内存池分别拥有独立的位图，内核位图起始地址存放在 MEM\_BITMAP\_BASE，用户位图紧随其后，避免了内存浪费。通过这种结构，内核能以非常小的开销，高效地跟踪系统中每一页内存的状态。



内存池结构图3-3

完成内存池初始化后，操作系统可通过get\_user\_pages来申请内存，调用get\_user\_pages后，内核首先会调用vaddr\_get()查找虚拟地址，若找到连续空闲的pg\_cnt个虚拟页，则循环调用oalloc()分配物理页并调用page\_table\_add()将虚拟地址与物理地址通过页表建立映射。(映射关系如图3-4)



映射关系图3-4

3.3.3 线程与进程调度机制设计

线程与进程的调度机制是实现多任务并发的重要基础。调度系统不仅负责为多个任务合理分配 CPU 资源，还要在多线程环境下确保系统的响应速度和任务切换的平滑性。为了达到这一目标，SpiderOS 采用了基于时间片轮转的调度策略，并结合线程状态管理、阻塞机制和优先级特性，实现了简单高效的任务调度框架。

在 SpiderOS 中，所有任务都以线程的形式存在，而进程则是拥有独立虚拟地址空间的线程扩展体。内核通过 task\_struct 结构体来描述每一个线程或进程的核心信息，包括进程 ID（pid）、线程状态（status）、内核栈指针（self\_kstack）、优先级（priority）以及用于调度的时间片（ticks）等字段。通过这套结构，操作系统可以准确描述并管理每一个正在运行或等待的任务。

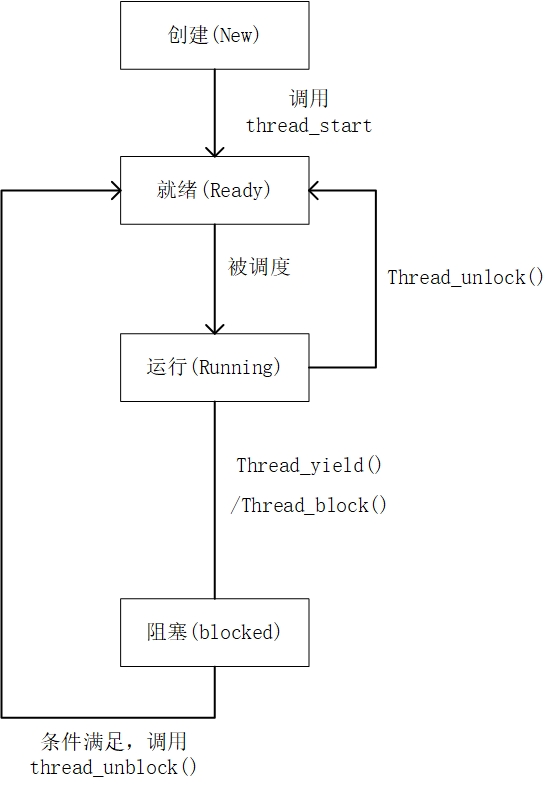
线程的创建流程清晰地遵循了初始化、栈布局配置、加入队列的顺序。当一个新线程通过 thread\_start() 创建时，系统会首先调用 init\_thread() 完成 PCB 结构的初始化，随后通过 thread\_create() 配置线程栈，最终将其加入到 thread\_ready\_list，等待调度器挑选执行。而主线程 main\_thread 则通过 make\_main\_thread() 完成特定的初始化，它不经过堆栈重新配置，而是直接使用已有的栈空间，完成就绪队列的挂载。

调度的核心由 schedule() 函数负责，该函数会在当前线程耗尽时间片或主动调用 thread\_yield() 时触发。调度器首先会将当前正在运行的线程重新加入就绪队列，并从 thread\_ready\_list 队列中选择下一个待运行的线程。通过 switch\_to() 完成线程上下文的切换，实现多任务并发运行的基础逻辑。每个线程的执行现场（寄存器和栈）都存储在其 PCB 中，确保切换时能够正确恢复执行状态(流程执行图3-5)

为了提升系统的资源利用效率，SpiderOS 设计了 idle\_thread 休眠线程。当就绪队列为空，调度器会自动唤醒 idle 线程进入休眠模式，通过 hlt 指令使 CPU 进入低功耗状态，直到新的中断或任务唤醒事件发生。这种设计不仅避免了 CPU 空转的浪费，也提升了整体系统的稳定性。

除了基本的时间片轮转调度，SpiderOS 还实现了阻塞与唤醒机制。当线程因资源不可用而无法继续执行时，可以通过 thread\_block() 将自身状态标记为 BLOCKED 并主动让出 CPU，待资源准备好后，通过 thread\_unblock() 将线程重新加入就绪队列，等待重新调度。这样，系统就实现了资源的同步等待与抢占机制，避免了死循环式的“忙等”浪费，提升了并发运行效率。

整个调度机制的设计，结合了简单直观的队列管理与灵活的状态切换策略，确保 SpiderOS 在运行时既能保证公平的任务切换，又具备良好的系统响应速度。这一机制不仅是内核多任务管理的基石，也是操作系统稳定运行的重要保障。



流程执行图3-5

3.3.4 输入输出系统

SpiderOS 的输入输出系统（I/O）是连接用户程序与硬件设备的重要桥梁，直接影响系统的交互效率和运行稳定性。操作系统中的 I/O 模块主要负责键盘输入、中断驱动、数据缓冲与控制台输出的实现，完成从底层硬件信号到用户可见字符的完整数据链路。

在 SpiderOS 中，I/O 系统由 keyboard.c、ioqueue.c 和 console.c 等模块组成。键盘模块通过配置中断向量，接收来自键盘硬件的扫描码，随后将其通过 ioqueue 环形队列缓冲，解决了输入设备速度与处理速度不匹配的问题，实现了异步输入流的稳定传递。当用户程序调用相关接口时，系统会从 ioqueue 中取出字符，实现用户输入的有序获取。

另一方面，输出部分通过 console.c 模块完成。该模块实现了字符的屏幕打印逻辑，底层采用直接操作显存的方式，将字符数据写入 0xb8000 地址所在的显存区域。同时结合 set\_cursor() 函数动态更新光标位置，模拟标准终端的行为，提升了用户体验。

此外，I/O 系统还对输入输出过程中的同步问题进行了设计，采用信号量与锁机制防止资源竞争，保证了并发环境下的输入输出正确性和安全性。尤其在多线程任务切换的场景下，SpiderOS 的 I/O 子系统通过对缓冲队列和打印操作的加锁，有效避免了线程竞争所导致的字符混乱。

3.3.5 用户进程

在 SpiderOS 中，用户进程的设计实现充分体现了操作系统内核与用户空间的隔离思想，保障了系统稳定性与进程独立性。用户进程不仅是用户程序运行的基本载体，同时也是内核资源调度与分配的核心单元。通过进程的引入，SpiderOS 实现了多用户程序并发运行的目标。

用户进程的创建主要由 process.c 模块完成，函数 process\_execute() 作为用户进程入口，负责完成用户态程序的内存空间分配、进程控制块（PCB）初始化、页表设置以及内核线程栈的准备工作。在进程创建阶段，SpiderOS 会首先调用内存管理模块，为用户进程单独申请用户态虚拟地址池，确保用户程序在运行时拥有独立的地址空间。同时，系统会分配一套独立的页目录，隔离用户态与内核态的地址映射关系，实现了基本的地址保护。

每一个用户进程都通过 task\_struct 结构与线程共享基础调度框架，但区别在于其 pgdir 成员指向单独的页表，而内核线程的该项为 NULL，这样实现了用户进程的虚拟地址空间与内核线程的物理内存安全隔离。此外，用户进程的 userprog\_vaddr 结构体用于记录用户态虚拟地址的分配状态，配合位图机制，实现了用户态内存的高效管理。

在用户进程的调度与运行过程中，SpiderOS 保持了统一的线程调度机制，用户进程与内核线程同属于 thread\_ready\_list，通过时间片轮转方式公平竞争 CPU 使用权。当用户进程被调度时，process\_activate() 函数会加载其页表，确保 CPU 访问的是用户进程自己的虚拟空间，而不是其他进程或内核的内存区域。

通过这种设计实现了用户进程的基本隔离、安全运行与高效调度，并为后续实现系统调用、用户态文件访问、进程创建与终止等功能提供了良好的基础。该设计遵循操作系统传统的内核态-用户态分离原则，同时结合分页机制实现了内存空间的隔离，提升了系统的稳定性与安全性。

3.3.6 文件系统

文件系统模块扮演着存储管理的核心角色。它不仅为用户进程提供了持久化存储接口，还通过抽象化的路径解析、文件描述符管理和块设备读写机制，实现了文件的有序组织与高效访问。

SpiderOS 的文件系统采用了简化的设计，整体架构围绕“块设备 - 文件系统 - 文件描述符 - 用户接口”四层展开。首先，硬盘的底层访问由 ide.c 提供，完成了扇区级别的数据读写操作，并通过分区扫描与设备描述符结构将硬盘设备抽象成逻辑上的 disk 对象。其上层通过 fs.c 负责文件系统的初始化与挂载，扫描引导区并识别有效的超级块 super\_block，为后续文件操作提供元数据支持。

文件的基本操作由 file.c 和 inode.c 模块共同完成。SpiderOS 将文件抽象为 inode 结构，inode 存储了文件的基本属性与磁盘块地址，便于快速定位文件内容所在的扇区位置。文件的打开、读写与关闭操作则通过 sys\_open、sys\_read、sys\_write 等系统调用提供，底层通过调用 file\_\* 系列函数完成 inode 操作与缓存管理。

目录的支持由 dir.c 提供，目录项在 SpiderOS 中被设计为特定结构体 dir\_entry，用于描述文件名和对应的 inode 编号，实现了路径与文件的映射关系。路径解析机制通过 sys\_open 和 sys\_mkdir 等接口完成，支持相对路径与绝对路径的解析，配合 cwd\_inode\_nr 字段，确保进程能够准确定位当前工作目录。

用户程序通过系统调用接口访问文件系统，内核则通过文件描述符表 fd\_table 完成文件句柄的映射。文件描述符机制屏蔽了底层设备与存储细节，使用户程序能够以统一的方式读写文件、设备和目录，提高了接口的通用性与易用性。

3.3.7 系统交互

在 SpiderOS 的设计中，用户与操作系统的交互主要通过命令行终端（Shell）完成。Shell 作为系统级的命令解释器，既是用户输入指令的入口，也是内核功能的直接展示窗口，承担着命令解析、任务执行、结果回显等核心职责，是连接用户空间与内核世界的桥梁。

SpiderOS 的 Shell 模块以简单实用为目标，围绕 my\_shell() 这一主循环函数展开设计。系统启动完成后，init 进程首先调用 my\_shell()，将其绑定至用户终端，进入交互等待状态。Shell 会首先打印提示符 [SpiderOS@hutaotao /]:~$ ，等待用户输入命令，随后解析并执行相应功能。

命令的解析流程由 cmd\_parse() 完成，用户输入的字符串会根据空格被拆分成命令与参数两部分，并根据 buildin\_cmd\_table 查找是否属于内置命令。若是内置指令，例如 ls、cd、pwd、mkdir 等，Shell 会直接调用对应的系统调用接口实现功能；如果输入命令不属于内置表，Shell 则尝试在文件系统中查找是否存在同名的可执行文件，若找到则通过 exec 系统调用加载并运行新进程。

SpiderOS 的 Shell 支持基础的路径解析功能，用户可使用相对路径或绝对路径切换工作目录，支持通过 pwd 命令查看当前路径，mkdir 创建目录，ls 查看文件列表，体现出一个简单文件系统的基本交互特性。此外，Shell 支持通过 ps 命令查看当前系统内运行的进程列表，配合内核调度机制，实现用户对进程状态的直观掌握。

Shell 的输入输出依赖于控制台驱动完成，用户的每次键盘输入都通过 keyboard.c 的中断处理机制被写入 I/O 缓冲队列，Shell 主循环从缓冲区读取并解析输入，完成指令的交互响应。输出部分则通过 put\_str、put\_char、put\_int 等函数，将命令执行的结果实时输出到控制台屏幕，完成人机交互的闭环。

第4章 SpiderOS功能验证与测试

4.1 内存管理功能验证

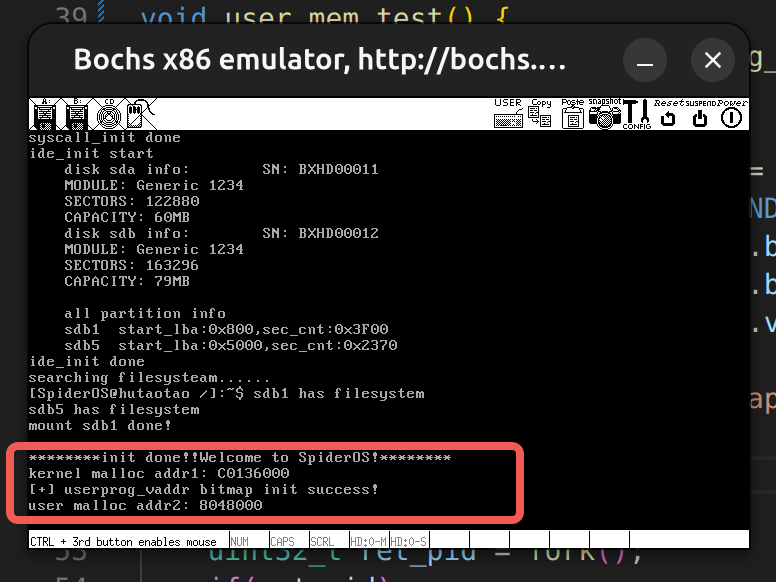
在 SpiderOS 内核的内存管理模块中，物理内存的使用被划分为两部分：**内核内存池（kernel\_pool）** 和 **用户内存池（user\_pool）**，二者通过位图机制管理物理页的分配与回收。为了验证内存池的划分是否正确、位图是否能正确标记页面的分配与释放，设计以下测试用例来验证 mem\_pool\_init() 完成内存池划分的正确性，以及 sys\_malloc() 是否能正确分配与释放内存，位图状态是否能准确反映。

**测试设计（代码见附录1）：**

为了确保 SpiderOS 内存管理模块的初始化与页框分配功能的正确性，本测试设计了针对内核内存池和用户内存池的分配流程验证。测试的核心目的是确认：

内核内存池通过 get\_kernel\_pages() 能够正确分配连续的物理内存页，  
由于用户进程的虚拟地址空间位图默认并未初始化，因此设计了 user\_mem\_test() 函数，手动对 userprog\_vaddr.vaddr\_bitmap 进行初始化，确保用户进程能够正常执行内存页的分配操作。

测试结果：

  
 内存分配测试结果

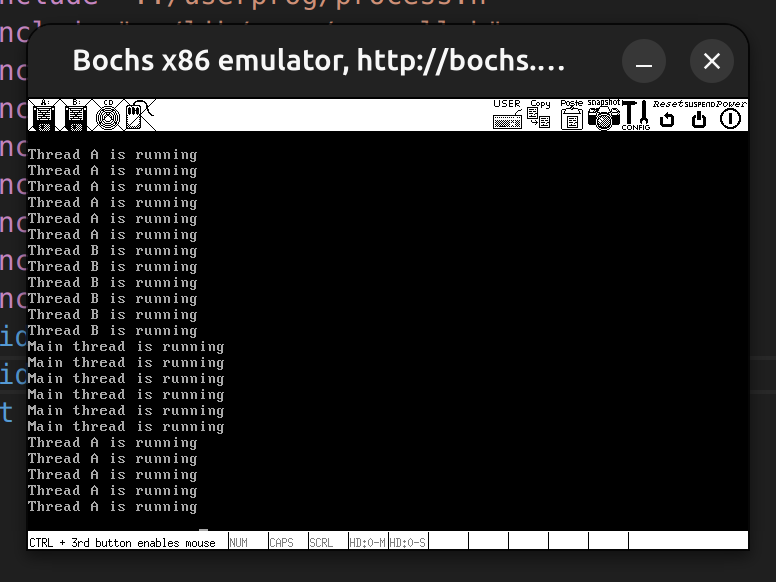
4.2 线程与进程调度机制验证

为了验证 SpiderOS 内核中线程与进程调度模块的稳定性与正确性，设计了一组基于实际代码的功能测试，覆盖线程创建、状态转换、调度切换等核心功能，确保系统具备基本的多任务并发运行能力。

测试设计（代码见附录2）：

SpiderOS的调度机制采用时间片轮转和阻塞唤醒相结合的策略，线程通过thread\_start()函数创建，并被插入就绪队列thread\_ready\_list，调度器在每次时钟中断中自动轮换，依次调度各个线程。

测试结果：



4.3 输入输出系统功能验证

在 SpiderOS 操作系统的输入输出系统中，主要涉及对控制台输出的管理和键盘输入的中断处理。输入输出模块不仅承担了用户与内核交互的桥梁作用，还负责在多线程并发访问控制台时保证输出内容的正确性和互斥性。为了验证控制台输出的互斥机制和键盘中断的输入响应，本节设计了简单的测试用例来验证 SpiderOS 的输入输出功能是否按预期运行。

测试设计（代码见附录3）：

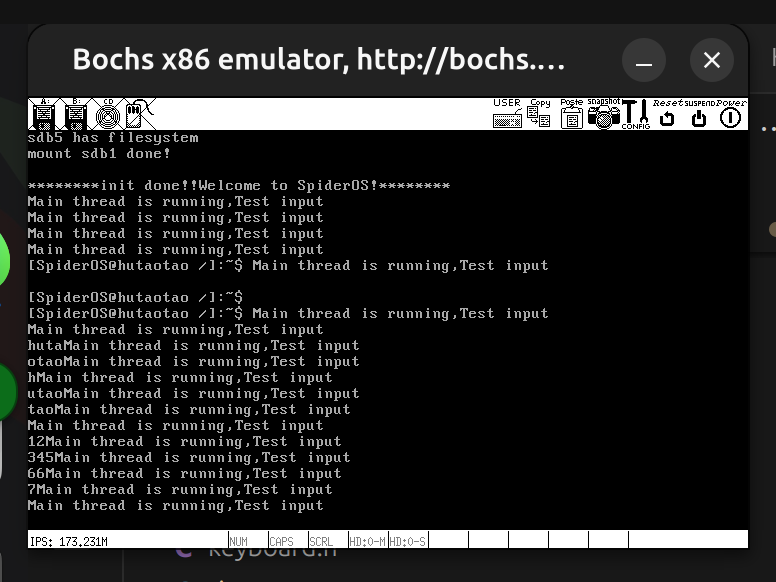
本次测试围绕 SpiderOS 的 console\_put\_str() 和 keyboard\_handler() 两个关键函数展开，主要测试两部分内容：

**控制台输出可行性验证**  
在Spideros启动后，主线程通过console\_put\_str()输出字符串，验证控制台输出是否稳定可靠。

**键盘输入中断测试**  
键盘输入模块通过注册中断处理函数 keyboard\_handler()，实现用户按键触发中断并将扫描码写入 ioqueue 环形缓冲区。测试通过在输入过程中观察是否能够正确捕获按键输入，并在控制台实时回显输入内容，验证中断处理流程和输入缓存逻辑是否稳定可靠。

测试结果：

如下图所示，可以看到主线程在循环输出"Main thread is running,Test input\n"的同时用户输入hutaotao，控制台正确的实时回显输入内容。测试结果正常。



4.4 文件系统功能验证

SpiderOS 文件系统的设计遵循了简洁而实用的原则，采用扇区读写方式和分区扫描机制，配合 inode、目录表以及块管理，完成对文件的创建、读写、关闭等基础操作。为了验证文件系统模块的正确性，主要围绕文件创建、读写、目录解析和分区挂载几个核心功能进行测试，确保 SpiderOS 的文件系统在正常条件下能够完成文件的基本操作。

测试设计（代码见附录5）：

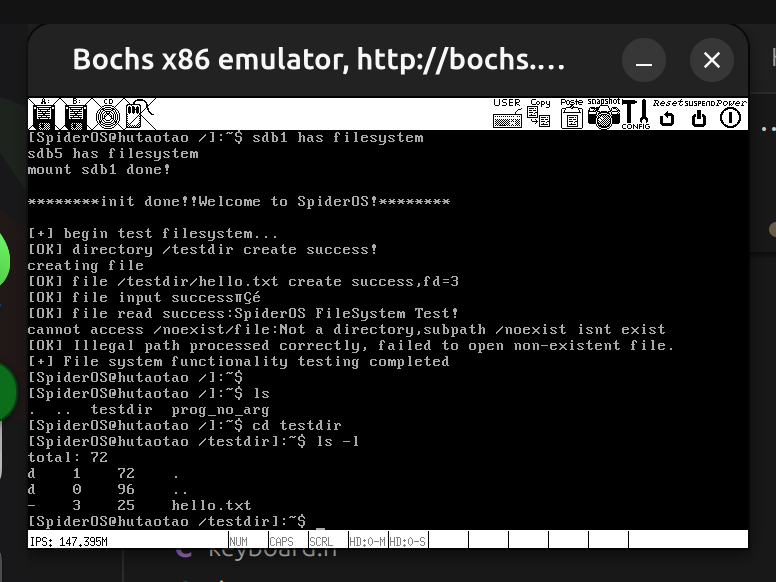
1.本次测试首先通过 sys\_mkdir() 创建一个新的目录 /testdir，随后在该目录下调用 sys\_open() 创建文件 /testdir/hello.txt，并使用 sys\_write() 写入测试数据，最后使用 sys\_read() 读取内容进行比对，确保数据完整一致，最后通过 sys\_close() 关闭文件句柄。

此外，为验证目录解析的正确性，测试用例还调用 sys\_open() 尝试访问不存在的路径，观察返回值是否正确处理错误路径，验证路径解析逻辑是否健壮。

2.

测试结果：

根据打印日志，可以见到文件系统正常执行并正常创建testdir目录和hello.txt文件，并且成功向文件写入25Bytes的数据。



4.6 系统交互功能验证

SpiderOS 的系统交互模块主要由 Shell 命令解释器与用户输入输出设备共同组成，实现了简易的命令行界面，用户可以通过键盘输入指令，操作系统解析后完成文件创建、目录切换、进程查看等基础命令操作。这一部分既体现了 SpiderOS 文件系统与进程调度机制的结合能力，也是操作系统完成用户交互的重要体现。

本次测试的核心目标，是验证 SpiderOS 的 Shell 是否能够正确接收用户输入，并完成命令解析、功能调用和结果回显。测试覆盖以下功能点：

1. 系统是否能正确显示提示符 [SpiderOS@hutaotao /]:~$，并等待用户输入。

2. 支持基础命令，如：

ls 列出当前目录文件；

cd 切换目录；

mkdir 创建目录；

rm 删除文件；

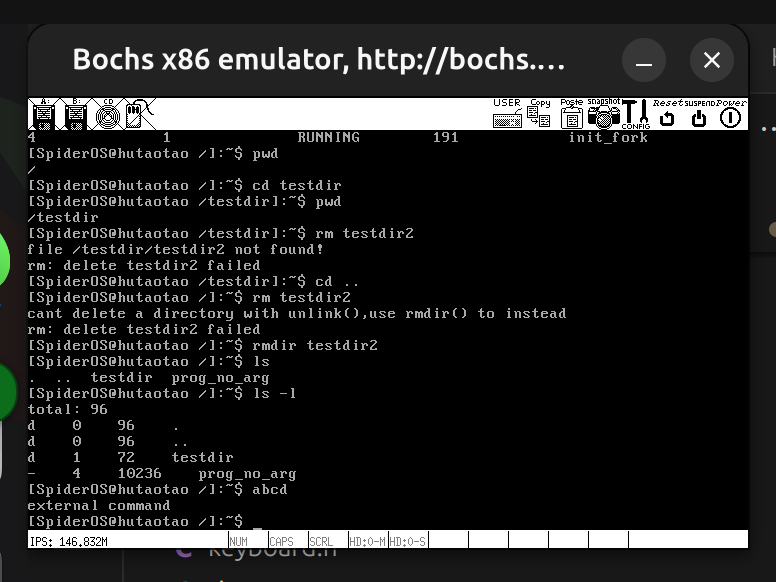
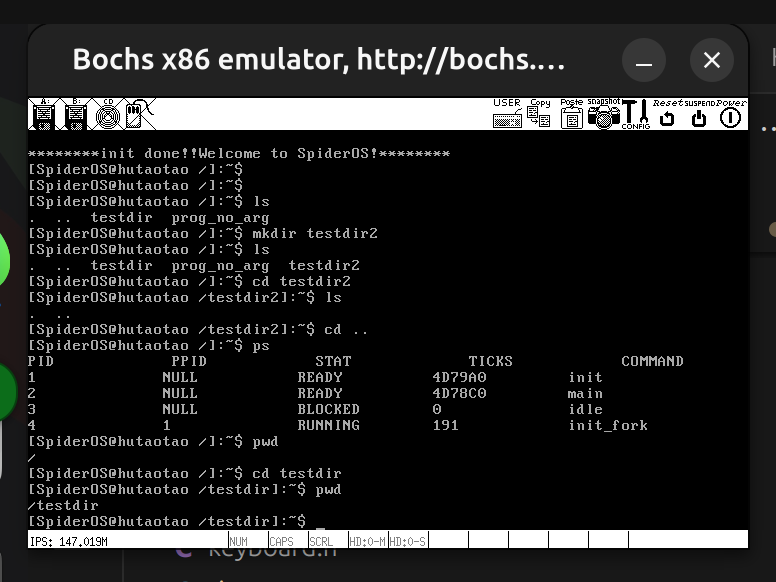
pwd 显示当前路径；

ps 查看进程列表。

1. 输入非法命令时，是否能正确回显提示“unknown command”。

测试结果：

根据以下测试图片，所有交互命令均正常运行并能处理非法命令。



**4.3.1 控制台输出打印功能测试  
4.3.2 键盘输入与中断驱动测试**

**4.4.1 用户进程创建与运行测试  
4.4.2 用户进程地址空间隔离性测试**

**4.5.1 分区扫描与挂载测试  
4.5.2 文件的读写与删除操作验证  
4.5.3 目录创建与路径解析测试**

4.6 系统交互功能验证  
**4.6.1 Shell命令解析与执行测试  
4.6.2 内置命令功能完整性测试**

第5章 总结与展望

5.1 研究总结

5.2 系统不足与后续改进

参考文献（不少于15篇，其中至少5篇英文）

[1] 彭星海. 基于x86架构的微内核操作系统的研究与实现[D]. 四川:电子科技大学,2020.

[2]程广辉,邓畅. 微内核操作系统的历史回顾与发展趋势分析[C]//上海市计算机学会第十届学术年会论文集. 2014:53-56.

（九）参考文献

参考文献一律放在文后，参考文献的书写格式要按现行国家标准规定。参考文献按文中出现的先后统一用阿拉伯数字进行自然编号，一般序码宜用方括号括起。

各类参考文献条目的编排格式及示例如下。

1. 专著著录格式

主要责任人.题名:其他题名信息[文献类型标识/文献载体标识].其他责任者.版本项.出版地:出版者,出版年:引文页码[引用日期].获取和访问路径.数字对象唯一标识符.

[1]PEEBLES P Z Jr.Probability,random variable,and random signal principles [M].4th ed. New York:McGraw Hill,2001.

2. 专著中析出的文献著录格式

析出文献主要责任者.析出文献题名[文献类型标识/文献载体标识].析出文献其他责任者//专著主要责任者.专著题名:其他题名信息.版本项.出版地:出版者,出版年:析出文献的页码[引用日期].获取和访问路径.数字对象唯一标识符.

[2]程根伟.1998年长江洪水的成因与减灾对策[M]//许厚泽,赵其国.长江流域洪涝灾害与科技对策.北京:科学出版社,1999:32-36.

[3]贾东琴,柯平.面向数字素养的高校图书馆数字服务体系研究[C]//中国图书馆学会年会论文集:2011年卷.北京:国家图书馆出版社,2011:45-52.

3. 连续出版物中的析出文献著录格式

析出文献主要责任者.析出文献题名[文献类型标识/文献载体标识].；连续出版物题名:其他题名信息,年,卷（期）:页码[引用日期]. 获取和访问路径.数字对象唯一标识符.

[4]于潇,刘义,柴跃廷,等.互联网药品可信交易环境中主体资质审核备案模式[J].清华大学学报（自然科学版）,2012,52（11）:1518.

4. 专利文献著录格式

专利申请者或所有者.专利题名:专利号[文献类型标识/文献载体标识].公告日期或公开日期[引用日期]. 获取和访问路径.数字对象唯一标识符.

[5]河北绿洲生态环境科技有限公司.一种荒漠化地区生态植被综合培育种植方法: 01129210.5[P/OL].2001-10-24[2002-05-28].http://211.152.9.47/sipoasp/zlijs/hyjs-yx-new.asp? recid=01129210.5&leixin=0.

5. 学位论文著录格式

[6]CALMS R B.Infrared spectroscopic studies on solid oxygen [D]. Berkeley:Univ.of California,1965.

6. 报告著录格式

[7]中国互联网络信息中心.第29次中国互联网络发展现状统计报告[R/OL].（2012-01-16）[2013-03-26]. http://www.cnnic.net.cn/hlwfzyj/hlwxzbg/201201/P020120709345264469680.pdf.

7. 报纸中析出文献著录格式

[8]丁文祥.数字革命与竞争国际化[N].中国青年报,2000-11-20(15).

8. 电子资源著录格式

主要责任者.题名:其他题名信息[文献类型标识/文献载体标识].出版地:出版者,出版年:引用页码(更新或修改日期). 获取和访问路径.数字对象唯一标识符.

[9]HOPKINSON A.Unimarc and metadata:Dublin core[EB/OL]. (2009-04-22)[2013-03-27].http://archive.ifla.org/IV/ifla64/138-16le.htm.

9. 标准的著录格式

[10]中国国家标准化管理委员会.卷烟 感官质量要求:GB 5606.4-2005[S].北京:中国标准出版社.2005.

艺术类专业如有特殊要求，可根据本《规范》，制定出适合本专业特点的毕业设计（论文）撰写规范，报校毕业设计（论文）工作领导小组审批后执行。

致 谢

致谢应以简短的文字对在课题研究和设计说明书（论文）撰写过程中曾直接给予帮助的人员或单位表示自己的谢意，这不仅是一种礼貌，也是对他人劳动的尊重，是治学者应有的思想作风，比如：本文是在导师XXX教授和XXX讲师的悉心指导下完成的，表示谢意！

感谢XX。

附录B： XX

附录是对于一些不宜放在正文中，但有参考价值的内容，可编入毕业设计（论文）的附录中，例如公式的推演、编写的程序等；当文章中引用的符号较多时，便于读者查阅，可以编写一个符号说明，注明符号代表的意义，也可作为附录的内容。一般附录放在全文最后。  
附录1：

int main(void)

{

init\_all();

intr\_enable();

void\* addr1 = get\_kernel\_pages(3);

put\_str("kernel malloc addr1: ");

put\_int((uint32\_t)addr1);

put\_char('\n');

user\_mem\_test();

void\* addr2 = get\_user\_pages(1);

put\_str("user malloc addr2: ");

put\_int((uint32\_t)addr2);

put\_char('\n');

while(1);

return 0;

}

void user\_mem\_test() {

struct task\_struct\* cur = running\_thread();

// 手动初始化用户空间虚拟地址位图

cur->userprog\_vaddr.vaddr\_start = USER\_VADDR\_START;

uint32\_t bitmap\_pg\_cnt = DIV\_ROUND\_UP((0xc0000000 - USER\_VADDR\_START) / PG\_SIZE / 8, PG\_SIZE);

cur->userprog\_vaddr.vaddr\_bitmap.bits = get\_kernel\_pages(bitmap\_pg\_cnt);

cur->userprog\_vaddr.vaddr\_bitmap.btmp\_bytes\_len = (0xc0000000 - USER\_VADDR\_START) / PG\_SIZE / 8;

bitmap\_init(&cur->userprog\_vaddr.vaddr\_bitmap);

put\_str("[+] userprog\_vaddr bitmap init success!\n");

}

附录2：

void thread\_a(void\* arg);

void thread\_b(void\* arg);

int main(void)

{

init\_all();

intr\_enable();

thread\_start("thread\_a", 31, thread\_a, NULL);

thread\_start("thread\_b", 31, thread\_b, NULL);

while (1) {

put\_str("Main thread is running\n");

}

return 0;

}

void init(void)

{

uint32\_t ret\_pid = fork();

if(ret\_pid)

while(1);

else

my\_shell();

PANIC("init: should not be here");

}

// 测试用线程A

void thread\_a(void\* arg) {

while (1) {

put\_str("Thread A is running\n");

}

}

// 测试用线程B

void thread\_b(void\* arg) {

while (1) {

put\_str("Thread B is running\n");

}

}

附录3：

int main(void)

{

init\_all();

intr\_enable();

while (1) {

delay(2000);

put\_str("Main thread is running,Test input\n");

}

return 0;

}

附录4：

void file\_system\_test() {

put\_str("\n[+] begin test filesystem...\n");

// 1️⃣ 测试目录创建

if (sys\_mkdir("/testdir") == 0) {

put\_str("[OK] directory /testdir create success!\n");

} else {

put\_str("[ERR] directory /testdir create fail!\n");

}

// 2️⃣ 测试文件创建与写入

int fd = sys\_open("/testdir/hello.txt", O\_CREAT | O\_RDWR);

if (fd != -1) {

put\_str("[OK] file /testdir/hello.txt create success,fd=");

put\_int(fd);

put\_char('\n');

char write\_buf[] = "SpiderOS FileSystem Test!";

if (sys\_write(fd, write\_buf, strlen(write\_buf)) == (int32\_t)strlen(write\_buf)) {

put\_str("[OK] file input success。\n");

} else {

put\_str("[ERR] file input fail!\n");

}

sys\_close(fd);

} else {

put\_str("[ERR] fail /testdir/hello.txt create fail！\n");

}

// 3️⃣ 测试文件读取

fd = sys\_open("/testdir/hello.txt", O\_RDWR);

if (fd != -1) {

char read\_buf[32] = {0};

sys\_read(fd, read\_buf, 31);

put\_str("[OK] file read success:");

put\_str(read\_buf);

put\_char('\n');

sys\_close(fd);

} else {

put\_str("[ERR] file read fail,can't open file。\n");

}

// 4️⃣ 测试非法路径

int wrong\_fd = sys\_open("/noexist/file", O\_RDWR);

if (wrong\_fd == -1) {

put\_str("[OK] Illegal path processed correctly, failed to open non-existent file.\n");

} else {

put\_str("[ERR] Illegal path error, file handle should be -1\n");

sys\_close(wrong\_fd);

}

put\_str("[+] File system functionality testing completed\n");

}