AIGC检测·全文报告单

NO:CNKIAIGC2025FG_202504100115238　　检测时间：2025-04-28 15:26:06

篇名：**自制操作系统的设计与实现**
作者：胡涛涛
单位：江南大学
文件名：自制操作系统的设计与实现.docx

全文检测结果　知网AIGC检测　https://cx.cnki.net

AI特征值：30.3%
AI特征字符数：6599
总字符数：21814

- AI特征显著 **(计入AI特征字符数)**
- AI特征疑似 **(未计入AI特征字符数)**
- 未标识部分

AIGC片段分布图

前部20%	中部60%	后部20%
AI特征值：20.3%	AI特征值：29.4%	AI特征值：42.7%
AI特征字符数：885	AI特征字符数：3851	AI特征字符数：1863

分段检测结果

序号	AI特征值	AI特征字符数 / 章节(部分)字符数	章节(部分)名称
1	10.6%	268 / 2518	中英文摘要等
2	42.2%	1513 / 3588	第1章绪论
3	19.8%	266 / 1342	第2章操作系统运行原理简述
4	21.1%	1723 / 8170	第3章系统结构设计与实现

5	41.4%	966 / 2335	第4章SpiderOS功能验证与测试
6	48.3%	1863 / 3861	第5章总结与展望

1. 中英文摘要等

AI特征值：10.6%　　AI特征字符数 / 章节(部分)字符数：268 / 2518

片段指标列表

序号	片段名称	字符数	AI特征	
1	片段1	268	显著	10.6%

原文内容

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，主要用途为学习研究，通过从自底向上逐步实现一套完整的操作系统，从而学习理解内核结构，进程调度，内存管理，文件系统，设备驱动，并提升开发者对操作系统体系结构的理解与动手能力。

本系统使用模块化的设计标准，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。并且各模块相互独立，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。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。

我们的开发过程从手动编写MBR和加载器开始，逐步实现了从BIOS的实模式到内核的保护模式转换。之后，进行了分页机制的初始化、内存池的划分以及虚拟地址空间的位图构建等关键的内存管理操作。在多任务方面，我们用双向链表管理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方式，确保了任务的公平调度。文件系统部分，我们完成了硬盘分区扫描、文件的读写操作、目录结构的解析，以及文件描述符的管理。用户可以在Shell命令行中轻松执行目录切换、新建文件和查看目录列表等基本操作。

10.6%(268)

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 the x86 architecture platform. The main purpose is to study and research, through the bottom-up step-by-step implementation of a complete set of operating system, so as to learn to

understand the kernel structure, process scheduling, memory management, file system, device drivers, and to enhance the developer's understanding of the operating system architecture and hands-on ability.

This system uses modular design standards, with clear boundaries of functional modules, the whole operating system is loaded from startup to kernel operation, and finally complete user interaction. 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.

Our development process started from writing MBR and loader manually, and gradually realized the conversion from real mode of BIOS to protected mode of kernel. After that, key memory management operations such as initialization of the paging mechanism, division of the memory pool, and bitmap construction of the virtual address space were performed. For multitasking, we managed the ready and blocking queues with bidirectional linked lists, which, combined with the active yield and interrupt-driven time-slice rotation methods, ensured a fair scheduling of tasks. In the file system section, we have accomplished hard disk partition scanning, file read/write operations, directory structure parsing, and file descriptor management. Users can easily perform basic operations such as directory switching, creating new files and viewing directory listings from the Shell command line.

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

2. 第1章绪论

AI特征值：42.2%　　AI特征字符数 / 章节(部分)字符数：1513 / 3588

片段指标列表

序号	片段名称	字符数	AI特征	
2	片段1	1254	显著	34.9%
3	片段2	259	显著	7.2%

原文内容

第1章绪论

1.1 研究背景与意义

操作系统，也就是我们常说的OS，是电脑里头非常核心的一部分软件，它负责管理硬件资源，比如处理器、内存、硬盘等等，还负责调度各种任务、驱动设备、支持文件系统，甚至还担着连接用户和硬件的桥梁角色。可以说，没有操作系统，电脑基本就没法用。在现代的电脑架构里，操作系统的稳定程度和可靠性直接关系到系统是否顺畅，所以它非常重要。

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越来越庞大，非常不便于阅读和学习。例如Linux第一个相对完整的版本在1991发布，当时的linux 0.11版本的内核才不到200K，而如今最新的4.20.13的Linux内核大小约100M[1]，翻了几百倍。可以体现增长速度可见一斑。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。

所以自主设计一个简单的操作系统很有必要，可以让开发者从零自下而上的逐步熟悉操作系统的核心架构以及运行的原理。并且在实践中深入学习理解原理，并在模块的实现中巩固理论知识。加深计算机体系和操作系统原理的认知理解。

除此之外操作系统的开发对个人能力提升也有很大的进步，独立开发操作系统需要对底层编程能力基础扎实，而且还要有纵观大局的模块化思维能力，并且开发者还需要有良好的代码规范以及异常处理来面对各种可能的问题。在实际的开发中，开发者还得面对很多关键的技术比如模块的划分，接口设计，数据结构选择以及资源同步等问题。这些挑战都能很好的锻炼开发者的个人能力。也可以更深刻的理解软硬件之间复杂的协作关系，把握系统启动和运行的机制。

因此自制操作系统在学习研究领域有很重要的作用，对于即将从事计算机行业的人来说尤其是系统开发，也有着很重要且积极的促进作用。

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。

二十世纪八十年代中期的时候，CarnegieMellon大学开发了第一款微内核的操作系统：Mach操作系统。它创新性的把之前内核态的服务都移动到了用户态，来实现更强大的模块化并且更加灵活。当然在那个计算机硬件资源匮乏的年代，硬件性

能有很大的限制，所以这一款操作系统在进程间通信(IPC)存在很大的开销，因此导致系统的整体性能不是很好。不过即使如此，它也对后面的诸多内核发展奠定了深远的基础，特别是在多线程支持和多处理器架构方面的探索(也就是现代操作系统)，这也是第一代微内核的创新性尝试。

尽管第一代微内核在性能方面有一些缺陷，但德国的计算机科学家约亨·利德克（Jochen Liedtke）在90年代初提出了L4微内核。L4大大简化了内核的功能，改善了IPC（进程间通信）机制，系统性能也因此得到了明显提升。特别是，L4采用寄存器传输信息的方法，降低了内存访问的开销，让IPC的速度比Mach快了20多倍。更厉害的是，L4的大小只有12KB，远小于Mach内核的300KB，充分展现了小型化内核在性能和资源使用上的优势。随着人们对信息安全的关注不断增强，第三代微内核在保证高性能的同时，也更重视系统的安全性。seL4是这阶段的代表之一，它在L4的基础上引入了能力空间（capability space）机制，确保每个进程只能访问自己被授权的资源，从而提升了安全等级。更重要的是，seL4还是全球第一个经过正式验证的微内核，使用数学方法验证了其8700行的C代码的正确性和安全性，广泛应用于航空航天和汽车电子等对安全性要求非常高的领域。

1.2.2 国内外微内核操作系统

国外对第三方操作系统的研究可以追溯得很早。20世纪60年代，著名的IBM推出了世界上第一个分时多任务系统CTSS，为现代操作系统里的进程控制和时间片调度提供了基础。后来，随着UNIX的出现和开源，更带动了Linux、FreeBSD等开源内核的发展，形成了如今开放、稳定，广泛应用于服务器、嵌入式系统和云计算的Linux系统。实际上Linux是个非常厉害的操作系统，很多国家和企业的操作系统都借用了它的内核或思想，比如鸿蒙OS、Android、SteamOS等等。过去十几年里，随着移动设备和嵌入式计算的需求不断增加，大家对轻量级操作系统的研究变得更加热烈。开源社区也涌现出了许多适合物联网设备的微内核，比如FreeRTOS和Zephyr，它们已经开始商用。这些系统一般都注重简化功能、提升稳定性，非常适合用来开发嵌入式和实时控制系统。与此同时，谷歌推出的Fuchsia操作系统，采用了全新的Zircon微内核架构，它打破了传统的UNIX框架，为未来的操作系统设计提供了新的思路。

中国的操作系统研发和实践也在不断发展。早期比较有代表性的是中科院计算技术研究所开发的银河操作系统和普华的 Linux。近几年，随着国产芯片的普及和新兴公司的支持，涌现出了像麒麟、鸿蒙这样的操作系统，它们面向桌面、移动设备和物联网等多个场景。这些新项目旨在让软硬件能更自主地运作，增强安全性、实现更好的互操作性，还推动相关环境的建设。同时，很多高校也开始推出一些实践性学习项目，比如南京大学的NEMU模拟器，鼓励学生通过编写和调试操作系统模块，培养软硬件协作的能力。如今，操作系统的教学不再只靠理论，而越来越偏重于基于具体项目的实践学习。国外的大学也有一些经典的入门操作系统案例，比如

34.9%(1254)

麻省理工学院的开源操作系统“xv6”和哈佛大学的“JOS”。这些课程主要介绍一些核心内容，比如虚拟内存管理、系统调用、文件系统和进程调度，帮助学生打下基础。

1.2 本文研究的主要内容

这个设计主要围绕x86架构平台，目标是实现一个简单的操作系统。开发者可以结合实际经验，深入了解操作系统的完整运行流程，包括引导加载、内核初始化、内存管理和文件系统的工作过程。最终，大家可以自己动手，打造一个具有基本功能的操作系统。以下是具体的研究内容介绍：

（1） 操作系统引导与启动机制

本系统从零开发，首先用汇编语言编写MBR主引导程序和Loader加载程序，在通过Bochs模拟X86环境完成硬盘引导区的读取和内存加载的过程。最后把内核加载到特定的位置并且跳转执行初始化函数，完成从硬件到内核的关键环节。

（2） 内核初始化与中断机制设计

内核初始化包括了GDT全局描述附表的初始化，IDT中断描述附表的初始化以及PIC可编程中断控制器的初始化，实现了软硬件中断的注册和管理。并支持键盘输入，时钟中断等交互。除此之外，内核还设计了统一的中断服务框架，能够打印异常信息。

（3） 内存管理模块的实现

内存管理是操作系统最核心的资源管理系统之一，本文中设计了位图管理的物理内存分配算法，并实现了页级内存的分配和释放。支持内核线程，用户进程的虚拟地址空间管理，还完成了malloc内存申请和释放功能，为进程以及文件系统提供了可靠的内存支持。

（4） 线程与进程管理机制

线程的设计包括完成了内核线程的创建，调度和切换功能。实现了TCB线程控制块和PCB进程控制块的设计，在调度方面，采用了简单的优先级轮转调度算法，支持简单的线程阻塞和唤醒，能够基本实现线程同步。不仅如此，用户进程还可以直接通过系统提供的接口来完成从内核态到用户态的任务切换。并且支持进程的fork和exec加载。

（5） 文件系统的设计与实现

文件系统模块可以扫描磁盘的分区并进行挂载，采用自己设计的简单文件系统格式，实现了创建、读取、写入和删除文件的功能。它还支持创建目录和浏览目录结构，形成基本的嵌套。系统通过接口提供给上层程序，让用户可以方便地进行文件操作。

（6） 系统调用接口设计

这篇文章介绍了用户空间到内核空间的系统调用机制，涵盖了常见的系统调用类型，比如进程管理、文件操作、内存分配和屏幕显示等。用户程序通过软中断触

7.2%(259)

发内核服务，实现用户态和内核态之间的安全切换，从而保证系统的稳定性和安全性。

（7） Shell命令行交互功能

为了让用户更方便地进行常用操作，我们设计了一个简单的Shell命令行解释器。这个工具支持像ls、pwd、mkdir、cd、rmdir这些基础命令，用户可以用它来创建文件、查看文件内容和切换目录，从而让系统的交互变得更加顺畅和直观。

3. 第2章操作系统运行原理简述

AI特征值：19.8%　　AI特征字符数 / 章节(部分)字符数：266 / 1342

片段指标列表

序号	片段名称	字符数	AI特征	
4	片段1	266	显著	19.8%

原文内容

第2章操作系统运行原理简述

2.1 x86架构计算机硬件简述

现代计算机最常见的架构为冯·诺依曼架构。如图2-1所示，这是一个x86计算机系统的硬件组成模型：

图2-1

传统的X86架构计算机系统中，硬件是由许多不同的子模块来组成，这些模块通过标准化的总线相互连接，相互配合协作来支撑操作系统以及应用程序的运行。如图2-1所示，现代操作系统主要包括了CPU,存储器，I/O桥,I/O总线.USB控制器，输入输出设备，图形适配器，磁盘控制器等等。其中CPU是整个系统的核心，也是大脑，负责指令的执行和数据的运算处理工作。它的内部继承了很多寄存器，用于存储变量和状态信息，并且可以通过总线接口和外部的模块进行通信。CPU通过系统总线连接到I/O桥，系统总线不仅承担着指令和数据的传输，还要负责协调多个模块的同步工作，比如存储器模块就通过存储器总线与I/O桥连接，在系统的启动和运行的时候CPU要频繁访问内存，这时候存储器总线就发挥了至关重要的作用。。I/O总线则与I/O桥相连接，用于连接各种输入和输出设备。不同的设备则需要专门的控制器连接到总线上，例如USB控制器可以连接键盘鼠标，图形适配器则需要连接显示器才可以进行图像的处理和输出，硬盘控制器则可以连接硬盘等设备，进行数据的存储和管理。其他的设备例如网卡，声卡等也可以通过I/O总线连接到整个系统。

2.2 操作系统的核心功能

操作系统是计算机中最基础也最重要的系统软件，它的主要作用是提供一套核心功能，让用户和各种应用程序可以顺利运行。操作系统帮助我们管控计算机的资源分配以及统筹调度。它的主要功能可以大致分为几个方面：

一、 进程管理进程是计算机在运行中的一个任务或程序，也是是系统资源的基本分配单位。操作系统则负责进程的创建，调度，执行以及终止。他会对每个进程进行分配必要的资源，比如CPU的时间片，内存空间以及文件的访问权限等等。还会根据调度算法来执行多进程并发执行。

二、 内存管理，内存（RAM）是计算机中非常重要的资源，由于ROM的读写速度较慢，而RAM是ROM速度的一百倍左右，所以电脑会把正在执行的资源加载到内存中，再由内存和CPU进行数据交换。而操作系统就负责内存的分配和回收。在现代的操作系统中，一般使用的是虚拟内存技术。把物理内存扩展成虚拟的地址空间，让每个进程都有他独自的内存空间。另外，操作系统还会采用分页和分段的技术，把数据和程序拆分成块再映射到物理内存，来达到数据隔离的效果。不过也可以通过共享内存来让多个进程共享一块内存区域，方便数据的交流。

三、 设备管理，操作系统还需要负责设备管理。例如键盘、鼠标、硬盘、显示器这些硬件设备，都会由操作系统统一调度。在硬件插入后，虚拟地址空间会有一块内存被提供用于访问硬件设备的入口。它会借助设备驱动程序，把硬件的复杂接口变得简单，让上层的应用可以以统一的方式访问不同的硬件设备。操作系统还会安排设备的使用顺序，管理缓存，以提高数据传输的效率和稳定性。

五、用户界面最后，操作系统需要提供一个用于交互的页面。这通常是通过命令行界面（CLI）或者图形界面（GUI）实现的。用户可以通过这些界面输入命令、管理文件、运行程序或者调整系统设置。

19.8%(266)

4. 第3章系统结构设计与实现

AI特征值：21.1%　　AI特征字符数 / 章节(部分)字符数：1723 / 8170

片段指标列表

序号	片段名称	字符数	AI特征	
5	片段1	230	显著	2.8%
6	片段2	337	疑似	4.1%
7	片段3	545	疑似	6.7%
8	片段4	1171	显著	14.3%
9	片段5	322	显著	3.9%

原文内容

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架构模拟器，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。Bochs在Ubuntu上很方便，因此SpiderOS采用的就是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

在我们熟悉的x86架构电脑里，开机启动操作系统的流程基本上可以理解为“固件 → 引导程序 → 操作系统内核”。当电脑刚通电时，主板上的BIOS会先进行一系列的自检和初始化（也就是POST），然后按照预设的启动顺序去读取硬盘、U盘等设备上的主引导扇区（MBR），把它加载到内存的0x7C00地址，然后把控制权传给这个MBR里的引导程序。接下来，引导程序才会启动整个系统。SpiderOS的引导过程也遵循这个经典的流程，因此引导程序主要由两个部分组成：主引导记录（MBR）和加载器（Loader）。MBR的任务其实很简单，就是调用BIOS中断，把磁盘上的Loader加载到内存的0x600位置，然后把控制交给Loader。Loader会执行一些更复杂的初始化，比如检测内存、切换到保护模式、设置分页机制，最后把内核加载到内存并开始运行。

开发操作系统时，除了启动步骤之外，还要根据硬件环境调整，合理安排内存

布局，确保CPU能顺利从实模式切换到保护模式。

3.2.2 启动加载器设计

在完成MBR的主引导程序后，Loader顺利的被加载到0x600的位置。然后Loader会进行更复杂的一些初始化流程，比如内存检测，从实模式切换保护模式（详见3.2.3），设置分页机制。最后会把内核加载到内存并开始运行。

对于大部分普通的电脑设备来说，加电后，CPU会先运行主板上的BIOS代码，完成硬件检测和初始化，然后按照预设的启动顺序进行引导。我们用Bochs模拟环境，首先创建一个空白的镜像文件hd40M.img，将主引导扇区（MBR）加载到这个镜像的第一个扇区。为了避免干扰或覆盖，系统会把Loader放在第3个扇区（LBA 2），而内核kernel则加载到第10个扇区。整个hd40M.img的结构可以参考图3-1。

hd40M.img

图 3-1

3.2.3 从实模式到保护模式

在使用x86架构的电脑时，一开机，CPU首先进入一种叫做实模式（Real Mode）的状态。在这个状态下，CPU操作的都是物理地址，由于寻址方式为16位的段寄存器的内容乘以16当作段基地址，加上16位的偏移地址形成20位的物理地址，所以它最多只能访问到1MB的物理内存。而且，实模式下还不支持现代操作系统的许多功能，比如内存保护、多任务处理和分页管理。因此，启动过程中，操作系统需要迅速的把CPU从实模式切换到保护模式（Protected Mode），这样才能为内核提供一个稳定可靠的硬件环境。

SpiderOS的启动加载器（Loader）在检测完内存后，会按照一定的步骤，从实模式平稳地切换到保护模式。一开始，BIOS会把主引导扇区（MBR）加载到内存的0x7C00地址，然后CPU会跳转到这个地址开始执行MBR中的代码。经过一番处理后，会调用硬盘读函数rd_disk_m_16，把硬盘上的Loader程序读取到内存中，然后用jmp指令跳转到Loader的内存位置，把控制权交给Loader。

Loader的加载地址通过宏定义LOADER_BASE_ADDR控制，在SpiderOS中设置的是0x900。Loader比MBR任务更复杂，它要进行内存检测，确认机器支持多大内存；接下来会设置全局描述符表（GDT），为在切换到保护模式后提供正确的段描述符和属性；最后，它会正式切换到保护模式，完成关键的转换。所有准备工作完成后，Loader会用jmp指令跳转到内核的main函数地址，也就是交还控制权给核心部分，从此进入C语言编写的世界。

3.3 内核核心模块设计与实现

顺利地完成了从实模式到保护模式的切换。当引导程序结束后，CPU的控制权就正式交给了内核代码，此时SpiderOS的内核会进行一系列的初始化。内核就像系统的大脑，主要负责处理用户的各种请求，协调硬件设备，确保整个系统平稳运行。

3.3.1 中断管理机制

中断机制在操作系统内核和硬件之间起着关键的桥梁作用。SpiderOS设计的中断子系统搭建了一个完整的响应体系，让硬件发出信号或出现故障时，CPU能快速而安全地切换到内核进行处理，这样系统才能保持平稳运行。至于x86架构，它为中断管理提供了硬件基础，那就是中断描述符表（IDT）。

中断描述符表（类似图3-2所示）中存放着129个中断向量，每个向量都对应一个中断门描述符，告诉CPU：当这个中断发生时，应该跳转到哪个地址去执行相关程序。SpiderOS用结构体gate_desc来抽象这个表项，里面包括中断服务函数的地址（包括低16位和高16位），段选择子，属性等信息。系统启动时，内核会准备好这个中断描述符表，填充每个表项的具体中断处理程序地址，通常由id_desc_init()函数完成。填充完毕后，通过lidt指令通知CPU表的起始位置和大小。此时，CPU已经可以处理中断了。当硬件设备（比如键盘输入、硬盘读写完成等）发出中断信号，CPU会根据中断向量自动从IDT中找到对应的中断门描述符，跳转到指定的中断处理程序。通常，我们会用汇编入口进入，然后保存现场、调用C层的general_intr_handler来统一处理。在这个过程中，CPU会自动切换堆栈、保护现场，确保用户程序的状态不被破坏。

gate_desc数据结构图 3-2

SpiderOS的中断管理系统中，有一个叫做register_handler()的函数接口，它让内核或驱动程序可以把特定的处理函数绑定到某个中断向量上。例如键盘驱动会把keyboard_handler()注册到键盘中断对应的中断向量上，硬盘驱动则会将intr_hd_handler()绑定到硬盘的中断向量。当硬件上发生事件的时候，用户无需直接干预，系统会自动调用对应的处理程序，帮助硬件和内核之间传递数据。为了防止多个中断处理相互干扰，SpiderOS还提供了intr_disable()和intr_enable()这两个函数，用来在一些关键操作时临时屏蔽中断。关闭中断后，内核可以安全地修改重要的数据结构，避免被其他中断打断导致的问题，等处理完毕，再恢复中断的状态。

3.3.2 内存管理系统

在设计SpiderOS的时候，内存管理模块扮演着关键的角色。它不仅保证了内核和用户程序的稳定运行，也是操作系统在资源分配和隔离方面的核心所在。

系统启动完成后，进入内核阶段并开启中断后，SpiderOS会立即调用mem_init()这个函数，启动内存管理系统的初始化过程。这个函数会读取存放在0x800地址的BIOS中的信息，确认目前系统的实际物理内存容量，然后把这个信息传递给mem_pool_init()来进行下一步的设置。

2.8%(230)

而mem_pool_init()是内存池的核心部分，它首先会根据页表的结构，计算出用来做分页机制的页表空间大小。默认情况下，操作系统会把低端1MB的空间（0x100000）和供页表使用的空间（256个4KB页）预留出来。这个区域由页表占用

，操作系统本身无法直接调度。剩余的空闲物理内存会被划分成两部分：一部分用作内核专用的物理内存池（kernel_pool），另一部分用作用户空间的内存池（user_pool），大概各占一半。为了让内存管理更高效，SpiderOS使用一种叫位图（Bitmap）的标记方式，每一位对应一页物理内存。0代表空闲，1代表已被分配。

内核和用户的内存池都拥有独立的位图，内核位图存放在MEM_BITMAP_BASE的地址，用户空间的位图紧随其后，这样可以节省空间。用这种方式，系统能够用很低的开销，快速准确地跟踪每一页内存的使用情况。

内存池结构图3-3

完成内存池初始化后，操作系统可以通过get_user_pages函数申请内存。当调用get_user_pages时，内核首先会用vaddr_get()函数查找虚拟地址。如果找到连续空闲的pg_cnt个虚拟页，就会循环调用oalloc()函数来分配对应的物理页，然后通过调用page_table_add()，在页表中建立虚拟地址与物理地址的映射关系（映射示意如图3-4所示）

映射关系图3-4

3.3.3 线程与进程调度机制设计

调度机制在实现多个任务同时运行时起着非常关键的作用。它不仅要合理分配CPU资源，确保每个任务都能得到处理，还要让系统在多线程环境下反应迅速，任务切换也要尽可能平滑。为达到这个目标，SpiderOS采用了基于时间片轮转的调度方案，同时结合了线程状态管理、阻塞机制和优先级设定，打造了一个既简单又高效的调度框架。在SpiderOS中，每个任务都作为一个线程存在，而进程则相当于是拥有自己独立虚拟空间的线程扩展。内核用结构体task_struct来描述每个任务的主要信息，包括进程ID（pid）、当前状态（status）、内核栈指针（self_kstack）、优先级（priority）以及调度用的时间片（ticks）等。

4.1%(337)

通过这套结构，操作系统可以准确地管理每一个正在运行或等待的任务。线程的创建流程也很清楚，主要包括初始化、栈布局配置以及加入调度队列。当一个新线程通过thread_start()函数创建时，系统会先调用init_thread()完成PCB（进程控制块）的初始化，然后用thread_create()设置好线程的栈，最后把它加入到ready_list（就绪队列）中，由调度器决定何时运行。而主线程main_thread则通过make_main_thread()进行特殊的初始化，不需要重新配置栈空间，而是直接使用已有的空间，并挂载到就绪队列中等待调度。。

调度的核心由 schedule() 函数负责，这个函数会在当前线程耗尽时间片，或者当你主动调用 thread_yield() 的时候触发。调度器会先把当前正在跑的线程放回就绪队列，然后从 thread_ready_list 里挑选下一个准备运行的线程。通过 switch_to() 来切换线程的执行环境，确保多任务能够顺畅地同时进行。每个线程的状态保存信息（像寄存器和栈）都存放在它的 PCB 里，这样切换时就能保证程序继续从正确的位置运行（看流程图3-5）。

为了让系统资源的使用更加高效，SpiderOS 设计了一个空闲线程，也叫 idle_thread。当就绪队列里没有任何任务时，调度器会自动唤醒这个空闲线程，让它进入休眠状态，然后用 hlt 指令让 CPU 低功耗模式。直到遇到新的中断或任务，空闲线程才会被重新唤醒。

除了基本的时间片轮转调度，SpiderOS 还增加了阻塞和唤醒的机制。当一个线程因为资源还没准备好而无法继续运行时，它可以调用 thread_block()，把自己标记为 BLOCKED，然后放弃 CPU。等到资源准备好后，再用 thread_unblock() 让它回到就绪队列，准备下一次调度。这样，系统既可以聪明地等待资源，又不会陷入忙等状态，从而提高了并发效率。整个调度机制结合了简单直观的队列管理和灵活的状态切换策略，不仅确保了蜘蛛操作系统的公平轮转，还能快速响应各种请求。这套机制不仅是内核多任务管理的基础，也是保证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（I/O）系统就像是连接用户程序和硬件设备之间的桥梁，直接影响着系统的交互效率和稳定性。这部分负责处理键盘输入、中断驱动、数据缓冲和控制台输出，完成从硬件信号到屏幕显示字符的整个过程。

在 SpiderOS 里，I/O 系统主要由 keyboard.c、ioqueue.c 和 console.c 这几个模块组成。 键盘模块通过设置中断向量，接收来自硬件的扫描码，然后把它们放到 ioqueue 的环形队列中进行缓冲，这样就避免了硬件输入速度和处理速度不一致的问题。这样一来，用户输入就可以以异步的方式稳定传输。当用户程序调用相关接口时，系统会从 ioqueue 取出字符，保证输入的顺序。而输出部分由 console.c 负责。它实现了把字符显示到屏幕上的功能，底层通过直接操作显存，把字符写入到内存地址 0xb8000。同时，它还会用 set_cursor() 动态更新光标位置，让终端的表现更自然，更顺畅。此外，I/O 系统在设计时也考虑了同步问题，采用信号量和锁机制，确保在多个任务同时进行输入和输出的操作时，数据不会出错。在多线程环境下，SpiderOS 的 I/O 子系统通过对缓冲区和打印操作加锁，有效避免了因为线程抢占带来的字符混乱。

3.3.5 用户进程

在完成以上的工作后，终于可以进入用户进程了，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。用户进程作为用户程序运行的基本载体，是内核资源的调度和分配的核心单元。通过进程的引入我们可以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 文件中模块完成，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的分配，进程控制块的初始化，页表设置以及一些内核线程的准备工作。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，来确保用户程序在运行时

6.7%(545)

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_struce数据结构和线程共享基础调度的框架，但是区别在于pgdir成员指向了单独的页表，而内核线程该项为NULL,这样就可以实现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。除此之外用户进程的userprog_vaddr结构体用来标记用户态虚拟地址的分配状态，来实现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。

3.3.6 文件系统

文件系统模块在整个操作系统中扮演着非常重要的角色，负责数据的存储和管理。为了实现既简洁又可靠的持久化存储，SpiderOS设计了基本的文件系统架构，涵盖了硬盘分区管理、文件的读写接口、目录的解析，以及管理文件信息的控制块（File Control Block，FCB）等关键部分。

在系统启动的早期，SpiderOS通过内核中的IDE设备模块识别硬盘，并用partition_scan()函数扫描硬盘上的主分区和逻辑分区，把有效的分区信息记录下来。系统会挂载默认的活跃分区，为后续的文件操作打好基础。每个分区由一个结构体存储，不仅包含起始LBA、扇区数，还内置超级块、块位图、inode位图和根目录信息，形成了对分区资源的管理单元。

文件系统采用一种简化的inode机制来管理每个文件。每次打开的文件都对应一个inode，记录文件的起始块、长度等重要信息。在系统启动时，调用filesys_init()函数来初始化整个文件系统。这包括读取存储在磁盘上的超级块、建立内存中的inode表、初始化文件表和目录结构。超级块保存了文件系统的基本信息，比如总的扇区数、inode的数量和数据区的起始位置，是系统正常运行的基础。

在文件操作的接口层面，SpiderOS提供了一系列系统调用，比如sys_open()、sys_read()、sys_write()和sys_close()，方便用户程序对文件进行操作。系统内部通过file.c模块处理具体的文件控制逻辑。例如，打开文件时会分配文件描述符、维护文件的偏移量，并进行块的映射计算。创建新文件时调用file_create()，而打开已有文件则用file_open()。

在路径解析方面，系统实现了基本的路径遍历逻辑。用户可以通过输入绝对路径或相对路径来访问文件，系统会调用search_file()函数，递归地逐级解析路径中的目录，最终找到目标文件或目录的inode。SpiderOS中的目录项（dir_entry）采用简单的结构，支持基本的目录读取和子文件查找操作。

14.3%(1171)

关于目录管理，SpiderOS支持sys_mkdir()来创建新目录，内部通过调用dir_create()分配磁盘空间，并且会更新父目录的目录项信息。同时，也支持sys_rmdir()删除空目录，以及sys_opendir()打开目录等常用操作。

为了保证读写的稳定性和多线程环境下的安全性，SpiderOS在访问文件或目录时引入了简单的锁机制。尤其是对分区位图和inode表的修改，都采用加锁方式进行保护，以避免多线程同时操作时出现数据冲突或破坏。

3.3.7 系统交互

用户和操作系统的互动主要靠命令行终端（Shell）来实现。Shell 就像是系统的一个指挥官，不仅是用户输入指令的地方，也能直接展现出内核的功能。它负责解析用户的命令、执行任务以及显示结果，是连接用户和核心系统的重要桥梁。SpiderOS 的 Shell 模块以简洁实用为目标，主要围绕一个叫做 my_shell() 的主循环函数来设计。当系统启动完毕后，init 进程会调用这个 my_shell()，把它绑定到用户的终端上，从而开始等待用户的指令。Shell 一开始会显示一个提示符 [SpiderOS@hutaotao /]:~$，然后等待用户输入命令，解析后执行相应的操作。

命令的解析是由 cmd_parse() 来完成，输入的字符串会被空格拆分成命令和参数两部分。Shell 会检查输入的命令是否属于内置命令（比如 ls、cd、pwd、mkdir 等），如果是，就直接调用相关系统调用实现功能；如果不是，Shell 会打印出”external command”来提示用户命令未知。SpiderOS 的 Shell 支持基础的路径处理，比如使用相对路径或绝对路径切换工作目录，用户可以通过 pwd 查看当前路径，使用 mkdir 创建新目录，使用 ls 查看文件列表，展示了一个简单文件系统的基本操作。

此外，Shell 还能通过 ps 命令查看系统中运行的进程列表，结合内核的调度机制，让用户能够直观了解系统状态。

3.9%(322)

5. 第4章SpiderOS功能验证与测试

AI特征值：41.4%　　AI特征字符数 / 章节(部分)字符数：966 / 2335

片段指标列表

序号	片段名称	字符数	AI特征	
10	片段1	966	显著	41.4%
11	片段2	287	疑似	12.3%
12	片段3	259	疑似	11.1%
13	片段4	287	疑似	12.3%

原文内容

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，物理内存被划分为两个部分：一个是内核内存池（kernel_pool），另一个是用户内存池（user_pool）。这两个内存池通过位图的方式来管理物理页面的分配和回收。为了确保内存池的划分正确、位图能准确反映页面的状态，我们设计了一些测试用例，主要用来验证 `mem_pool_init()` 是否正确完成了内存池的划分，以及 `sys_malloc()` 是否能正确地分配和释放内存，位图是否清楚地显示了这些操作的结果。

测试设计（代码见附录1）：

为了确保 SpiderOS 的内存管理模块在初始化和分配页框方面都能正常工作，这次测试主要是检查内核内存池和用户内存池的分配流程。测试的关键是确认，内核内存池能通过 get_kernel_pages() 方法成功分配一段连续的物理内存页。此外，由于用户进程的虚拟地址空间的位图默认没有初始化，我们特别设计了 user_mem_test() 函数，手动设置了 userprog_vaddr.vaddr_bitmap，确保用户进程可以正常进行内存页的分配操作。

测试结果：成功

内存分配测试结果

4.2 线程与进程调度机制验证

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，用来测试线程调度机制的核心功能，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）：

41.4%(966)

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。

12.3%(287)

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。

11.1%(259)

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。

3. 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。

12.3%(287)

6. 第5章总结与展望

AI特征值：48.3%　　AI特征字符数 / 章节(部分)字符数：1863 / 3861

片段指标列表

序号	片段名称	字符数	AI特征	
14	片段1	625	显著	16.2%
15	片段2	1238	显著	32.1%

原文内容

第5章总结与展望

5.1 研究总结

本项目基于X86架构自制了一个小型的操作系统SpiderOS,围绕着操作系统的核心机制和功能展开开发研究。通过实践来深入理解操作系统各个方面的整体运行机制。

在系统启动部分，本系统实现了从机器上电到加载MBR再到保护模式切换，加载内核等完整流程，来确保系统可以顺利的进入多任务的操作环境。除此之外通过手动设计中段模块，内存管理模块，线程和进程模块，文件系统模块来让我深入学习操作系统底层的相关知识，也为我后续从事的系统研发工作奠定了技术基础。至此虽然操作系统的设计告一段落，但是相关源码已上传至Github开源并在后续我会持

续完善。

5.2 系统不足与后续改进

尽管 SpiderOS 已实现了许多基本功能，但在实际应用中，系统仍存在一些不足之处，主要体现在以下几个方面：

Shell 功能简单

目前的 Shell 仅支持一些基础命令，比如创建文件和目录等，但功能相对单一，无法像主流操作系统那样支持管道、重定向等命令行功能，也不支持执行复杂脚本。因此，用户操作的灵活性和交互性有所局限。

无图形用户界面（GUI）

目前系统仅支持文本模式，没有提供图形用户界面，这让用户操作显得较为繁琐和单调。如果用户操作较多的情况下，界面的复杂度和使用体验会大大降低。

文件系统功能有限

SpiderOS 的文件系统目前还只是支持基本的文件和目录操作，功能上相对简化。比如缺乏高效的文件访问机制，文件的管理和访问速度较慢，不能满足大规模数据处理的需求。

缺少完善的系统调试功能

在调试和错误追踪方面，系统并没有提供完善的工具或机制，用户在进行系统开发或调试时，可能无法快速定位问题，缺乏高效的调试支持。

针对以上不足，未来的改进方向可以包括：

增强 Shell 的功能，支持更多的命令与操作，提高用户操作的灵活性。

设计并实现简单的图形用户界面（GUI），提升用户体验，使操作更直观。

改进文件系统，加入更高效的文件存储和访问机制，优化文件系统性能。

提供更完善的调试工具，帮助开发者在系统开发过程中更高效地排查和解决问题。

附录A： 测试集源码

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)
    {
```

32.1%(1238)

```
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");
// 1 □ 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");
    }
    // 2 □ 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");
    }
    // 3 □ 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");
    }
    // 4 □ 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");
    }
```

说明:

1、支持中、英文内容检测;

2、AI特征值=AI特征字符数/总字符数;

3、红色代表AI特征显著部分，计入AI特征字符数;

4、棕色代表AI特征疑似部分，未计入AI特征字符数;

5、检测结果仅供参考，最终判定是否存在学术不端行为时，需结合人工复核、机构审查以及具体学术政策的综合应用进行审慎判断。

cx.cnki.net

https://cx.cnki.net
知网个人AIGC检测服务