

小型内核设计文档

思路过程:

虽然学完了操作系统这门课程的理论，但是对于我们来说编写一个内核代码还是不容易，于是我们参考了《操作系统真相还原》这本书的一些内容，加上参考其他的一些网上资料和视频教学从零开始搭建一个操作系统内核。

我们在构建操作系统内核的过程中，采用了由底层到高层、逐步扩展功能的设计思路。最初，通过汇编语言编写引导程序（Bootloader），实现从硬件启动到保护模式的切换，并完成内核加载的基本流程。为了更高效地实现复杂功能，系统从汇编语言逐步过渡到 C 语言，使内核代码更加简洁、易维护。在内核加载完成后，首先解决内存管理问题，通过段式和页式内存管理机制，划分内核空间和用户空间，确保内存资源的高效利用，同时为后续功能提供了稳定的基础。

在内存管理完善的基础上，内核引入了多任务处理机制，通过任务切换和调度支持多个任务的并发运行。多任务的设计包括实现简单的线程和进程模型，调度算法的选择，以及任务间同步与通信机制（如信号量和锁），从而提升了系统的并发能力。在完成这些内核核心功能之后，开始着手实现数据的持久化存储功能。为此，设计并实现了一个简单的文件系统，支持文件的创建、读取、写入和删除，同时引入目录结构，支持嵌套目录管理，为用户提供了文件存储和管理的基础功能。

接下来，为了支持硬件交互，内核开发了设备驱动程序，包括键盘、鼠标、显示器和磁盘驱动。设备驱动程序的设计负责抽象硬件操作，提供统一的接口供上层调用，同时提升了系统的可扩展性，使内核能够适配更多类型的设备。在硬件支持和文件系统功能完善之后，我们进一步为系统实现了基本的用户界面。通过命令行界面（Shell），用户可以执行简单的命令与系统交互，管理文件、查看系统状态，并运行用户程序。这种逐步从底层硬件管理到高层用户交互功能的设计思路，不仅确保了系统的稳定性和灵活性，也为功能的扩展和优化提供了清晰的逻辑路径。

各个系统模块的设计方案：

具体的方案设计如下面的目录和下面的报告内容：

实现重点：

我们实现的重点就是操作系统内核应该具备的基本功能，比如文件管理，内存管理，设备驱动以及进程和线程管理的实现。最后，为了我们交互的方便，我们组还从零实现 `shell` 功能，能够实现命令的输入输出，解析命令，解析路径，以及 Linux 下的 `ls`、`cd`、`mkdir`、`ps`、`rm` 命令等等。

实现创新点：

1. 缓冲池设计

文件系统通过引入缓冲池（如 I/O 缓冲区 `io_buf`），在磁盘数据读写过程中临时存储数据，有效减少磁盘 I/O 次数并提升性能。缓冲池的设计允许将多次小规模读写操作合并为一次批量处理，从而提高效率。此外，在 `inode` 和目录项的跨扇区操作中，缓冲池整合了多个扇区的数据，避免复杂的跨扇区处理逻辑，简化了实现。同时，缓冲池还支持数据的预处理（如跳过无效目录项、更新元信息等），确保磁盘操作更高效，进一步提升了文件系统的整体运行速度。

2. 支持跨扇区的数据管理

文件系统针对 `inode` 和目录项可能跨越多个扇区的情况设计了专门的处理机制，实现了对大文件和复杂目录结构的支持。在文件存储方面，通过直接块和一级间接块的结合，文件系统可以管理超出单个扇区存储容量的大文件。在目录管理中，当目录项跨扇区时，利用缓冲区和块分配机制扩展目录存储空间，同时确保目录结构的完整性和操作的灵活性。这种设计使得文件系统能够高效处理超出单个扇区限制的大数据，并具备动态扩展能力。

3. 错误回滚与一致性机制

文件系统在操作过程中引入了错误回滚机制，确保在文件创建、删除或资源分配失败时系统的一致性。具体来说，在文件创建过程中，如果某个环节（如目录项写入或块分配）失败，会通过回滚机制释放已分配的 `inode` 和磁盘块，并更新相关位图状态，将系统恢复到操作前的状态。通过记录操作步骤（如

rollback_step)，能够按顺序撤销已完成的步骤，避免数据不一致或资源泄漏。该机制在保障文件系统数据完整性和可靠性方面具有重要意义，特别是在系统中断或操作失败时，显著提高了文件系统的健壮性。

4. 双向链表在内核中的灵活应用

内核使用了双向链表作为核心的数据结构，用于管理进程、分区、文件等多种队列。双向链表的设计保证了高效的插入、删除操作，同时提供了灵活的遍历方式，通过回调函数实现链表的条件查找。这种设计避免了传统数组实现的固定长度限制，提供了动态、灵活的队列管理能力，非常适合内核中对动态资源的高效管理需求。

5. 线程与任务的细粒度管理

内核设计了简化的 PCB（进程控制块）和线程栈，既支持内核线程的管理，也为用户态任务的扩展提供了基础。通过保存线程运行的上下文信息（如寄存器、函数指针和栈顶指针），实现了线程的灵活切换。首次调度和任务切换使用统一的机制，将用户自定义任务函数与内核线程调度逻辑结合，为任务管理提供了灵活的可扩展性。

目录

环境部署:	7
环境部署（准备工作）	7
sudo apt install nasm.....	7
引导加载流程	9
1. BIOS.....	9
2. MBR（主引导代码）	9
1） 初始化寄存器;.....	10
2） 清屏，通过显存打印队伍名称“HFUT 666”;	10
3） 加载 loader 程序到内存，过程如下：	10
1） 通过 0x1F2 端口设置需要读取的扇区数；	10
4） 通过 0x1F7，写入读命令；	10
3. loader(内核加载程序).....	11
1） 初始化环境	11
2） 获取系统内存布局	11
3） 切换到保护模式	12
4） 保护模式初始化	12
5） 设置分页机制	12
6） 内核初始化	14
7） 跳转到内核入口地址执行	14
内存管理.....	14
1. 内存池初始化.....	15
1） 分离内核和用户内存空间	15
2） 位图管理内存使用	15
3） 统一初始化流程	15
2. 虚拟地址管理.....	15
1） 虚拟地址分配	15
2） 虚拟地址释放	16
3） 用户进程虚拟地址隔离	17
3. 物理地址管理.....	17
1） 页框分配	17
2） 页框释放	17
3） 页表管理	17
4. 内存块管理.....	18
1） 内存分块与描述符设计	18
2） 内存块分配策略	19
3） 内存块释放策略	20
在内核空间实现线程.....	21
基础的数据结构.....	21
线程的实现.....	24
双向链表的数据结构设计	26
实现的功能	26

多线程调度模块设计与实现	27
多线程调度模块实现思路	29
任务调度器和任务切换模块设计与实现	29
进程管理.....	31
1. 进程的创建与复制	31
1) 复制父进程的 PCB 和内存资源	31
2) 复制父进程的用户内存空间	32
3) 构建子进程的内核栈	33
2. 进程的执行与用户空间管理	34
1) 用户进程创建与初始化	34
2) 用户进程的启动	34
3) 创建页目录	35
4) 初始化虚拟地址位图	35
3. 进程的调度与切换	35
1) 进程调度	35
2) 任务切换和 TSS 的更新	37
设备管理.....	38
1. 键盘驱动程序	38
1.1 原理简介	38
1.2 设计目标	39
1.3 系统设计	39
2. 环形缓冲区管理	40
2.1 设计目标	40
2.2 系统设计	40
3. 硬盘驱动程序	42
3.1 设计目标	42
3.2 核心功能实现	42
文件系统.....	46
1.创建超级块、inode 和目录项（基础的数据结构）	46
1.1 超级块（Super Block）	46
1.2 inode（索引节点）	47
1.3 目录项（Directory Entry）	47
索引结点与目录项的关系	48
2.创建文件系统（其实还是一些初始化的工作，初始化分区）	48
(1) 根据分区大小计算元信息的扇区数及位置	49
(2) 在内存中创建超级块	49
(3) 将超级块写入磁盘	50
(4) 将元信息写入磁盘上的各自位置	50
(5) 将根目录写入磁盘	50
3.文件系统初始化函数(filesys_init)	50
4.挂载分区	51
步骤详解	52
5.文件描述符:	53
文件描述符与 inode 关联关系	53

6.文件操作相关的基础函数.....	54
Inode 操作相关函数.....	54
文件相关函数.....	54
目录相关操作函数.....	54
路径解析相关函数.....	54
功能总结:	54
7.创建文件:	55
8.文件的操作.....	56
8.1 文件的打开.....	56
8.2 文件的关闭.....	57
8.3 文件写入.....	57
8.4 文件写入核心功能 file_write.....	58
8.5 读取文件.....	58
8.6 现文件读写指针定位.....	59
8.7 文件的删除功能:	59
实现思路.....	60
实现思路:	60
9. 目录的操作.....	61
9.1 创建目录.....	61
9.2 遍历目录:	62
9.3 删除目录:	62
实现思路.....	63
9.4 任务的工作目录.....	63
功能说明.....	63
Shell 功能.....	64
Shell 功能与实现思路.....	64
命令输入与快捷键支持 (readline).....	64
命令解析 (cmd_parse).....	65

环境部署：

环境部署（准备工作）

1. 编译器选用：主要采用 c 语言开发，选择 gcc 编译器；对于汇编语言编译器选用 nasm 编译器。

配置：sudo apt install gcc

sudo apt install nasm

注意：此处由于我学习参考的代码架构不支持采用高版本的 gcc 编译器运行，所以需要下载 4.8 以下的版本（这里我采用的是 gcc-4.4），否则只支持部分代码编译，不能全部都正常编译。

2. 虚拟机选取：选用 vmware (vmware 和 virtualBox 均可以，看个人)

3. 下载并配置 bochs (2.6.8) 用来模拟 x86 硬件平台（能达到以下运行效果）

首先在 bochs 中找到 bochsrs.txt 文档然后将其复制到 bochsrc.disk 中，然后将其中的内容改成如下：（注意修改路径）

megs : 32

romimage: file=/home/xht/xht/bochs/share/bochs/BIOS-bochs-latest

vgaromimage: file=/home/xht/xht/bochs/share/bochs/VGABIOS-lgpl-latest

boot: disk

log: bochs.out

mouse:enabled=0

keyboard:keymap=/home/xht/xht/bochs/share/bochs/keymaps/x11-pc-us.map

ata0:enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14

ata0-master:type=disk,path="hd60M.img", mode=flat,cylinders=121,heads=16,spt=63

ata0-slave:type=disk, path="hd80M.img", mode=flat,cylinders=162,heads=16,spt=63

先行创建软盘 hd60M.img 和从盘 hd80M.img, 并为 hd80M.img 分区（过程如下）

```

1. Create new floppy or hard disk image
2. Convert hard disk image to other format (mode)
3. Resize hard disk image
4. Commit 'undoable' redolog to base image
5. Disk image info

0. Quit

Please choose one [0] 1

Create image

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] hd

What kind of image should I create?
Please type flat, sparse, growing, vpc or vmware4. [flat]

Enter the hard disk size in megabytes, between 10 and 8257535
[10] 60

What should be the name of the image?
[c.img] hd60M.img

Creating hard disk image 'hd60M.img' with CHS=121/16/63

```

```

1. Create new floppy or hard disk image
2. Convert hard disk image to other format (mode)
3. Resize hard disk image
4. Commit 'undoable' redolog to base image
5. Disk image info

0. Quit

Please choose one [0] 1

Create image

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd]

What kind of image should I create?
Please type flat, sparse, growing, vpc or vmware4. [flat]

Enter the hard disk size in megabytes, between 10 and 8257535
[10] 80

What should be the name of the image?
[c.img] hd80M.img

Creating hard disk image 'hd80M.img' with CHS=162/16/63

```

分区效果如下：

```

xht@xht-VMware-Virtual-Platform: ~/xht/bochs
g  新建一份 GPT 分区表
G  新建一份空 GPT (IRIX) 分区表
o  create a new empty MBR (DOS) partition table
s  新建一份空 Sun 分区表

命令(输入 m 获取帮助): p
Disk ./hd80M.img: 79.73 MiB, 83607552 字节, 163296 个扇区
单元: 扇区 / 1 * 512 = 512 字节
扇区大小(逻辑/物理): 512 字节 / 512 字节
I/O 大小(最小/最佳): 512 字节 / 512 字节
磁盘标签类型: dos
磁盘标识符: 0xec8ba4da

设备      启动    起点    末尾    扇区    大小  Id 类型
./hd80M.img1  2048    18175   16128   7.9M  83  Linux
./hd80M.img4  18432   163295  144864  70.7M  5  扩展
./hd80M.img5  20480   29551   9072    4.4M  66  未知
./hd80M.img6  32768   45367   12600   6.2M  66  未知
./hd80M.img7  49152   70000   20849   10.2M  66  未知
./hd80M.img8  73728   130000  56273   27.5M  66  未知
./hd80M.img9  133120  163295  30176   14.7M  66  未知

```

至此运行代码所需的一些必要条件已经准备就绪（如果在 bochs 运行时出现 no bootable device 可能是上述软盘、从盘配置出现问题，小组其他成员出现过类似情况）

引导加载流程

1. BIOS

计算机上电后，CPU 的 cs: ip 寄存器被强制初始化为 0xF000: 0xFFFF0，即 BIOS 程序入口地址 0xFFFF0 处开始执行，此处为 16B 大小的跳转指令，将跳向 0xfe05b 处，这是 BIOS 代码真正开始的地方。

接下来 BIOS 程序将检测内存、显卡等外设信息，当检测通过，并初始化好硬件后，开始在内存中 0x000~0x3FF 处建立数据结构，中断向量表 IVT 并填写中断例程。

实模式下 1M 内存空间布局

起始	结 束	大 小	用 途
FFFF0	FFFFF	16B	BIOS 入口地址，此地址也属于 BIOS 代码，同样属于顶部的 640KB 字节。只是为了强调其入口地址才单独贴出来。此处 16 字节的内容是跳转指令 jmp f000: e05b
F0000	FFFFF	64KB-16B	系统 BIOS 范围是 F0000~FFFFF 共 640KB，为说明入口地址，将最上面的 16 字节从此处去掉了，所以此处终止地址是 0xFFFFF
C8000	FFFFF	160KB	映射硬件适配器的 ROM 或内存映射式 I/O
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	B7FFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	A7FFF	64KB	用于彩色显示适配器
9FC00	9FFFF	1KB	EBDA (Extended BIOS Data Area) 扩展 BIOS 数据区
7E00	9FBFF	622080B 约 608KB	可用区域
7C00	7DFF	512B	MBR 被 BIOS 加载到此处，共 512 字节
500	7BFF	30464B 约 30KB	可用区域
400	4FF	256B	BIOS Data Area (BIOS 数据区)
000	3FF	1KB	Interrupt Vector Table (中断向量表)

2. MBR (主引导代码)

BIOS 最后一项工作将校验启动盘中位于 0 盘 0 道 1 扇区的内容，如果此扇区末尾的两个字节分别是魔数 0x55 和 0xaa，BIOS 便认为此扇区中确实存在可执行的程序，便加载到物理地址 0x7c00，随后跳转到此地址，继续执行，即执行主引导程序 MBR。

在 MBR 引导扇区中的内容包括：

- (1) 446 字节的引导程序及参数；
- (2) 64 字节的分区表；
- (3) 2 字节结束标记 0x55 和 0xaa。

MBR 的主要任务是加载内核加载器(loader.s)到指定位置,并将控制权交给它。
在我们的 MBR 程序中,完成了如下工作:

- 1) 初始化寄存器;
- 2) 清屏,通过显存打印队伍名称“HFUT 666”;
- 3) 加载 loader 程序到内存,过程如下:

程序将通过读取磁盘的 2 号扇区,即 loader 程序所在的位置,将其写入内存,具体的操作是通过配置硬盘控制器的端口寄存器来实现的,各端口作用如下:

IO 端口		端口用途	
Primary 通道	Secondary 通道	读操作时	写操作时
Command Block registers			
0x1F0	0x170	Data	Data
0x1F1	0x171	Error	Features
0x1F2	0x172	Sector count	Sector count
0x1F3	0x173	LBA low	LBA low
0x1F4	0x174	LBA mid	LBA mid
0x1F5	0x175	LBA high	LBA high
0x1F6	0x176	Device	device
0x1F7	0x177	Status	Command
Control Block registers			
0x3F6	0x376	Alternate status	Device Control

- 1) 通过 0x1F2 端口设置需要读取的扇区数;
- 2) 将扇区的 LBA 地址写入 0x1F3 (LBA low), 0x1F4 (LBA mid), 0x1F5 (LBA high);
- 3) 通过 0x1F6, 配置 device 寄存器的值, 设置为 1110_0000, 其第 5 位和第 7 位固定为 1, 第 6 位为 1 表示启用 LBA;
- 4) 通过 0x1F7, 写入读命令;
- 5) 同一端口(0x1F7), 检测硬盘状态, 若硬盘控制器准备好数据传输且硬盘空闲, 则可以传输, 否则等待。通过检测该端口中两位二进制数的值来获取状态;
- 6) 若准备好, 从 0x1F0 端口读数据, 一个扇区有 512 字节, 每次读入一个字, 所以需要循环读取 256 次, 同时读取的数据放在内存的某个位置 (如起始地址为 0x600), 这就是 loader 程序在内存中的起始地址。

此后, MBR 程序将跳转到 0x600 处, 即把控制权交给 loader 程序。

3. loader(内核加载程序)

该程序完成的工作有：

1) 初始化环境

包括对栈和寄存器初始化，对内存信息缓冲区初始化。清空 EBX、ES 等寄存器，避免受上一程序的影响，并准备存储 BIOS 提供的内存布局信息。

2) 获取系统内存布局

通过调用 int 0x15 的 e820 功能，获取 ARDS（地址范围描述结构）。这是一个用于描述内存物理地址范围的结构体。其结构如下：

地址范围描述符结构 ARDS

字节偏移量	属性名称	描 述
0	BaseAddrLow	基地址的低 32 位
4	BaseAddrHigh	基地址的高 32 位
8	LengthLow	内存长度的低 32 位，以字节为单位
12	LengthHigh	内存长度的高 32 位，以字节为单位
16	Type	本段内存的类型

此中断例程的调用方法如下表：

BIOS 中断 0x15 子功能 0xE820 说明

调用或返回	寄存器或状态位	参 数 用 途
调用前输入	EAX	子功能号：EAX 寄存器用来指定子功能号，此处输入为 0xE820
	EBX	ARDS 后续值：内存信息需要按类型分多次返回，由于每次执行一次中断都只返回一种类型内存的 ARDS 结构，所以要记录下一个待返回的内存 ARDS，在下次中断调用时通过此值告诉 BIOS 该返回哪个 ARDS，这就是后续值的作用。第一次调用时一定要置为 0，EBX 具体值我们不用关注，字取决于具体 BIOS 的实现。每次中断返回后，BIOS 会更新此值
	ES: DI	ARDS 缓冲区：BIOS 将获取到的内存信息写入此寄存器指向的内存，每次都以 ARDS 格式返回
调用或返回	寄存器或状态位	参 数 用 途
调用前输入	ECX	ARDS 结构的字节大小：用来指示 BIOS 写入的字节数。调用者和 BIOS 都同时支持的大小是 20 字节，将来也许会扩展此结构
	EDX	固定为签名标记 0x534d4150，此十六进制数字是字符串 SMAP 的 ASCII 码；BIOS 将调用者正在请求的内存信息写入 ES: DI 寄存器所指向的 ARDS 缓冲区后，再用此签名校验其中的信息
返回后输出	CF 位	若 CF 位为 0 表示调用未出错，CF 为 1，表示调用出错
	EAX	字符串 SMAP 的 ASCII 码 0x534d4150
	ES:DI	ARDS 缓冲区地址，同输入值是一样的，返回时此结构中已经被 BIOS 填充了内存信息
	ECX	BIOS 写入到 ES:DI 所指向的 ARDS 结构中的字节数，BIOS 最小写入 20 字节
	EBX	后续值：下一个 ARDS 的位置。每次中断返回后，BIOS 会更新此值，BIOS 通过此值可以找到下一个待返回的 ARDS 结构，咱们不需要改变 EBX 的值，下一次中断调用时还会用到它。在 CF 位为 0 的情况下，若返回后的 EBX 值为 0，表示这是最后一个 ARDS 结构

若调用成功后，ARDS 数据存储到缓冲区 ards_buf。中断调用可能失败，因此需要检查是否成功。如果失败，会尝试备用方法获取内存信息。即依次尝试 e801 和 88h 功能获取基础内存信息。

在缓冲区 `ards_buf` 中已经存储了所有的 ARDS 记录，每条记录占用 20 字节。现在需要找到所有可用内存区域的最大结束地址。通过遍历 ARDS 表，读取每个描述符的基地址和长度，计算每个内存区域的结束地址并比较，保存系统中所有可用内存区域的最大结束地址。

3) 切换到保护模式

为了突破实模式下 1MB 的内存限制，启用内存保护机制和多任务支持，而实现对更大内存和复杂操作系统的管理，需要从实模式进入到保护模式，需执行以下三个步骤：

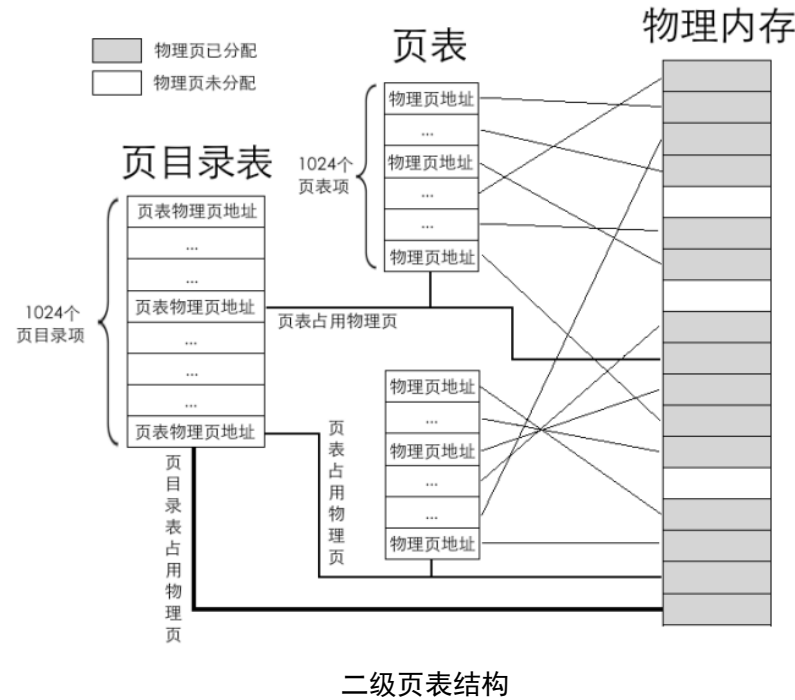
- (1) 开启 A20 地址线，允许访问 1MB 以上的内存；
- (2) 加载 `gdt`，为保护模式设置段描述符；
- (3) 将 `cr0` 的 `pe` 位置 1，表示切换到保护模式。

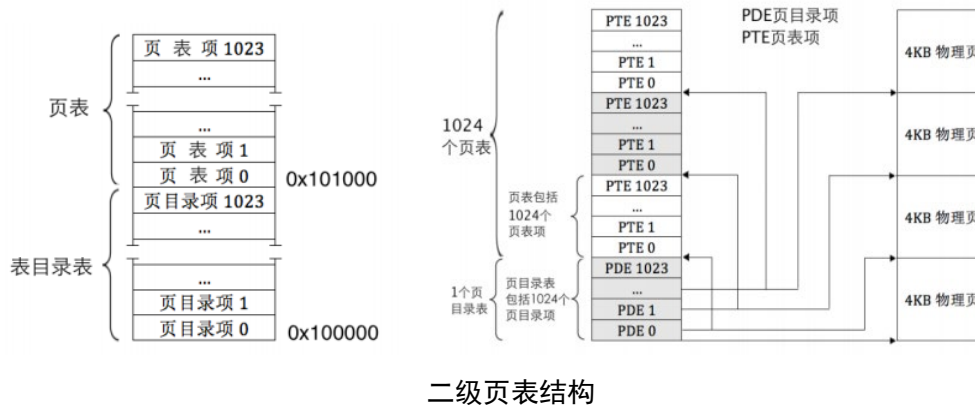
4) 保护模式初始化

设置段寄存器 `DS`、`ES`、`SS`，并初始化栈指针，然后从硬盘读取内核数据到内存缓冲区。

5) 设置分页机制

二级页表结构如下：





具体实现如下：

准备好页目录及页表

a) 建立页目录项：

首先初始化页目录表，避免未初始化的数据导致分页机制错误。然后设置第 0 项页目录项，使其指向第一个页表，映射虚拟地址 0x0~0x3FFFFFFF，覆盖低端 1MB 的物理内存，使 loader 的线性地址与分页后的虚拟地址一致。页目录项的值包括第一个页表地址，以及读写控制权限等标志位。

设置第 768 项页目录项，使其也指向第一个页表，映射虚拟地址 0xC0000000~0xC03FFFFFFF，覆盖低端 1MB 的物理内存。

设置第 1023 项页目录项使其指向页目录表自身，便于动态操作页表。

b) 建立页表：

创建第 0 个页表的页表项 (PTE)，第 0 个页表共需 256 个页表项（对应 1MB 空间，每个页框 4KB），设置每个页表项的地址和属性，起始物理地址：0x0，每次增加 0x1000（4KB），保证物理地址连续分配，属性包括读写权限等标志位，它将 0x0-0x1FFFFFF（虚拟地址）映射到 0x0-0x1FFFFFF（物理地址）。

最后创建其他页目录项，提前为内核空间（高 1GB）的所有页目录项分配页表地址。

启用分页

通过 `sgdt` 指令获取当前全局描述符表（GDT）的地址和大小。将段基址设置为高 1GB（`0xC0000000`）起始，配合分页机制映射虚拟地址到物理地址。调整 GDT 起始地址，确保段描述符的基地址指向高地址；然后更新栈指针，将栈指针移动到高 1GB 的虚拟内存区；最后设置页目录基地址（CR3），并启用分页机制（CR0）。

6) 内核初始化

初始化内核就是根据 `elf` 规范将内核文件中的段（segment）展开到（复制到）内存中的相应位置，过程如下：

首先，清零寄存器，防止之前程序的值对当前操作产生干扰。然后读取程序段表信息，在 ELF 文件头包含了段表的偏移、每个段的大小和段的数量。

遍历程序段表中的每个段，并将需要加载的段复制到内存目标地址。每个段的结构体包含类型，目标地址，源地址，大小四个关键字段。最后把程序中的段通过 `mem_cpy` 函数复制到段自身的虚拟地址处。

7) 跳转到内核入口地址执行

此后将进入内核程序开始运行。

内存管理

实现操作系统内存管理的核心功能，包括内存池初始化、虚拟地址管理、物理地址管理、页框分配与释放、内存块管理等模块。通过初始化内核和用户内存池，建立内存位图和页表映射，实现虚拟地址与物理地址的有效管理。同时支持按页分配与释放整页内存，满足大内存需求；通过内存块管理细分页框，支持小内存块的分配与回收。此外，代码提供了虚拟地址与物理地址之间的转换工具和辅助功能，确保内存分配和回收的高效性和可靠性，为操作系统的进程运行提供了稳健的内存管理机制。主要设计思想介绍如下：

1. 内存池初始化

1) 分离内核和用户内存空间

操作系统的内存管理需要区分内核和用户内存，以保证内核的安全性和稳定性，同时为用户进程提供足够的内存资源。因此，内存池初始化的核心思想是将所有空闲内存划分为 **内核内存池(kernel pool)** 和 **用户内存池(user pool)**，两者独立管理，互不干扰。内核池负责为内核分配内存(如中断处理、内核线程)，用户池负责为用户进程分配内存。

2) 位图管理内存使用

使用位图(bitmap)作为内存分配的核心数据结构，每一位表示一个页框的使用状态，0 表示空闲，1 表示已分配，使得其占用空间小，还能快速定位空闲页框。

每个内存池都维护一张位图，同时内核还额外维护了一个**虚拟地址池**位图，用于管理内核的虚拟地址分配。

3) 统一初始化流程

内存池初始化中，对内核池、用户池和虚拟地址池进行统一的初始化流程，包括地址计算、位图分配、内存分布打印等。

2. 虚拟地址管理

1) 虚拟地址分配

核心功能是扫描位图，找到指定数量的连续空闲虚拟页框，并返回起始虚拟地址。位图扫描(bitmap_scan)是关键部分，它通过检查位图中的连续位状态，快速定位可以分配的地址范围。代码描述如下：

```
int vaddr_start = 0, bit_idx_start = -1;
```

```

if (pf == PF_KERNEL) {
    bit_idx_start = bitmap_scan(&kernel_vaddr.vaddr_bitmap, pg_cnt);
    if (bit_idx_start == -1) return NULL;
    while (cnt < pg_cnt)
        bitmap_set(&kernel_vaddr.vaddr_bitmap, bit_idx_start + (cnt++), 1);
    vaddr_start = kernel_vaddr.vaddr_start + bit_idx_start * PG_SIZE;
}

```

首先，`bitmap_scan` 函数在内核虚拟地址位图 (`kernel_vaddr.vaddr_bitmap`) 中扫描，查找连续 `pg_cnt` 个空闲页框的起始索引。如果没有找到空闲页框，返回 `NULL`，表示分配失败。找到起始索引后，利用 `bitmap_set` 将位图对应的范围置为 1，表示这些页框已被占用。同时，根据位图的起始索引和虚拟地址池的起始地址 (`kernel_vaddr.vaddr_start`)，计算出分配的虚拟地址的起始位置。

2) 虚拟地址释放

通过位图索引，标记指定数量的虚拟页框为未分配状态（将位图置 0）。对于内核或用户进程的虚拟地址池，分别通过位图索引定位到要释放的地址范围。代码描述如下：

```

if (pf == PF_KERNEL) {
    bit_idx_start = (vaddr - kernel_vaddr.vaddr_start) / PG_SIZE;
    while (cnt < pg_cnt) {
        bitmap_set(&kernel_vaddr.vaddr_bitmap, bit_idx_start + (cnt++), 0);
    }
} else {
    struct task_struct* cur_thread = running_thread();
    bit_idx_start = (vaddr - cur_thread->userprog_vaddr.vaddr_start) / PG_SIZE;
    while (cnt < pg_cnt) {
        bitmap_set(&cur_thread->userprog_vaddr.vaddr_bitmap, bit_idx_start
+ (cnt++), 0);
    }
}

```


若回收的是内核空间的虚拟地址（PF_KERNEL），通过计算地址与内核虚拟地址起始地址的偏移量，确定页框在位图中的起始索引，并将位图中对应的页框标记为 0（空闲）。若回收的是用户进程的虚拟地址，则获取当前线程的用户虚拟地址池，通过相同方式计算位图索引，并将对应范围置为 0。

3) 用户进程虚拟地址隔离

每个用户进程通过 struct task_struct 维护自己的虚拟地址位图和起始地址。系统保证用户进程的虚拟地址空间不会超出其分配范围，也无法访问其他进程或内核空间。

```
if (cur->pgdir != NULL && pf == PF_USER) {  
    bit_idx = (vaddr - cur->userprog_vaddr.vaddr_start) / PG_SIZE;  
    ASSERT(bit_idx > 0);  
    bitmap_set(&cur->userprog_vaddr.vaddr_bitmap, bit_idx, 1);  
}
```

判断当前线程是否为用户进程（cur->pgdir != NULL）且分配内存池为用户空间（pf == PF_USER）。通过计算虚拟地址在用户虚拟地址池位图中的索引（bit_idx），确保索引有效（ASSERT），并将位图中对应位置标记为 1，表示该虚拟地址已被占用。这样实现了用户进程虚拟地址的分配和记录。

3. 物理地址管理

1) 页框分配

页框分配采用位图扫描算法，通过位图标记空闲的页框，并返回分配的物理地址。

2) 页框释放

根据物理地址判断页框属于内核池还是用户池，将对应位标记为 0，表示页框空闲。

3) 页表管理

动态分配页目录和页表，完成虚拟地址到物理地址的映射。

```
void page_table_add(void* _vaddr, void* _page_phyaddr) {
```

```

uint32_t vaddr = (uint32_t)_vaddr, page_phyaddr = (uint32_t)_page_phyaddr;
uint32_t* pde = pde_ptr(vaddr); // 获取页目录指针
uint32_t* pte = pte_ptr(vaddr); // 获取页表指针

if (!(*pde & 0x00000001)) { // 页目录不存在
    uint32_t pde_phyaddr = (uint32_t)palloc(&kernel_pool); // 分配页目录
    *pde = (pde_phyaddr | PG_US_U | PG_RW_W | PG_P_1);
    memset((void*)((int)pte & 0xffff000), 0, PG_SIZE); // 清空页表
}

ASSERT(!(*pte & 0x00000001)); // 确保页表项不存在
*pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1); // 设置页表项
}

```

首先，通过虚拟地址计算得到对应的页目录项指针和页表项指针。如果页目录项尚未存在（通过检查其最低位是否为 1 判断），则动态分配一个物理页框作为页表，并将其地址写入页目录项，同时设置权限位（用户级、可读写、存在）。接着，通过 `pte_ptr` 获取页表项指针，确保当前页表项不存在（避免覆盖），然后将物理地址写入页表项，并同样设置权限标志位。如果页目录不存在，会自动分配并初始化页表。

4. 内存块管理

1) 内存分块与描述符设计

每个页框被划分为多个固定大小的内存块，每个内存块的大小由内存块描述符 `mem_block_desc` 定义。内存块描述符记录了块大小（`block_size`）、每个页框内可容纳的块数（`block_per_arena`），以及管理这些块的空闲链表（`free_list`）。

分块大小按照 2 的倍数递增（例如 16B、32B、64B 等），满足不同大小的分配需求，避免过多碎片。

2) 内存块分配策略

对小于或等于 1024B 的分配请求，采用内存块管理。分配时根据请求的大小找到合适的块描述符，通过空闲链表快速获取一个空闲块。当一个块描述符的空闲链表为空时，分配新的页框，将整个页框划分为多个内存块，挂入该描述符的空闲链表中。

```
void* sys_malloc(uint32_t size) {
    if (size > 1024) { // 请求大于 1024B，直接分配页框
        uint32_t page_cnt = DIV_ROUND_UP(size + sizeof(struct arena),
        PG_SIZE);

        struct arena* a = malloc_page(PF_KERNEL, page_cnt);
        a->desc = NULL; // 标记为大内存分配
        a->cnt = page_cnt;
        a->large = true;
        return (void*)(a + 1);
    }

    uint8_t desc_idx = 0;
    while (size > k_block_descs[desc_idx].block_size) desc_idx++;

    struct mem_block_desc* desc = &k_block_descs[desc_idx];
    if (list_empty(&desc->free_list)) { // 空闲链表为空
        struct arena* a = malloc_page(PF_KERNEL, 1); // 分配一个页框
        a->desc = desc;
        a->cnt = desc->block_per_arena;
        uint32_t block_idx;
        for (block_idx = 0; block_idx < desc->block_per_arena; block_idx++) {
            struct mem_block* b = arena2block(a, block_idx);
            list_append(&desc->free_list, &b->free_elem);
        }
    }
}
```

```

    }

    struct mem_block* b = elem2entry(struct mem_block, free_elem,
list_pop(&desc->free_list));

    struct arena* a = block2arena(b);
    a->cnt--; // 减少空闲块数
    return (void*)b;
}

```

针对不同大小的内存请求采取不同策略：对于大于 1024 字节的请求，直接按页框分配，并标记为大内存分配；对于小于等于 1024 字节的请求，利用内存块管理，从空闲链表中分配内存块。如果链表为空，会动态分配一个页框并划分为多个内存块挂入链表中。分配完成后，更新内存块和页框的状态，返回分配的内存地址。

3) 内存块释放策略

每个内存块的地址对应于一个页框。通过内存块地址，确定所属的页框（arena），然后将内存块回收，挂回所属描述符的空闲链表。如果一个页框中的所有内存块都被释放，则释放整个页框。

```

void sys_free(void* ptr) {
    struct mem_block* b = ptr;
    struct arena* a = block2arena(b);

    if (a->large) { // 大块直接释放页框
        mfree_page(PF_KERNEL, a, a->cnt);
        return;
    }

    list_append(&a->desc->free_list, &b->free_elem);
    a->cnt++; // 增加空闲块数
    if (a->cnt == a->desc->block_per_arena) { // 所有块已空闲

```

```

uint32_t block_idx;
for (block_idx = 0; block_idx < a->desc->block_per_arena; block_idx++) {
    struct mem_block* b = arena2block(a, block_idx);
    list_remove(&b->free_elem);
}
mfree_page(PF_KERNEL, a, 1); // 释放页框
}
}

```

如果内存块所在的 arena 标记为大内存块(a->large)，直接通过 mfree_page 释放其占用的页框。如果是小内存块，将内存块重新加入其对应的空闲链表，并增加 arena 的空闲块计数(a->cnt)。如果 arena 中的所有内存块都空闲(a->cnt 等于块总数)，会逐一移除这些内存块并释放整个 arena 所占用的页框。

在内核空间实现线程

基础的数据结构

简单的进程控制块（PCB）

通过定义 struct task_struct，实现了最基础的 PCB，用于描述线程（或进程）的属性和状态信息。PCB 包含线程的内核栈指针 self_kstack、线程状态 status、优先级 priority、线程名 name 以及栈边界标记 stack_magic，后者用于检测栈溢出。

```

struct task_struct {
    uint32_t* self_kstack; // 各线程的内核栈顶指针
    enum task_status status; // 线程状态
    uint8_t priority; // 线程优先级
    char name[16]; // 线程名
    uint32_t stack_magic; // 栈的边界标记，用于检测栈溢出
};

```

线程栈的实现 (thread_stack)

设计并实现了 struct thread_stack，它是线程自己的栈，用于保存线程运行时的上下文信息以及首次运行时需要的函数和参数。线程栈通过结构体成员保存关键寄存器（如 ebp、ebx、edi、esi），并包含一个用于存储待执行函数地址的成员 eip，用于实现线程首次运行时的调度以及任务切换时的环境恢复。

```
struct thread_stack {
    uint32_t ebp;
    uint32_t ebx;
    uint32_t edi;
    uint32_t esi;

    void (*eip)(thread_func* func, void* func_arg); // 待执行的函数
    void (*unused_retaddr);                        // 占位的返回地址
    thread_func* function;                         // 函数名
    void* func_arg;                               // 函数参数
};
```

中断栈的实现 (intr_stack)

通过定义 struct intr_stack，实现了中断发生时用于保存上下文的栈结构。中断栈会在中断进入时保存寄存器和相关信息（如中断号 vec_no、段寄存器 cs 和 ss、指令指针 eip 等），并在中断退出时恢复这些信息，确保中断处理后线程能够正常运行。

```
struct intr_stack {
    uint32_t vec_no;    // 中断号
    uint32_t edi;       // 通用寄存器
    uint32_t esi;
    uint32_t ebp;
    uint32_t esp_dummy; // 被忽略的 esp
    uint32_t ebx;
    uint32_t edx;
    uint32_t ecx;
    uint32_t eax;
    uint32_t gs;        // 段寄存器
    uint32_t fs;
    uint32_t es;
    uint32_t ds;
```

```
uint32_t err_code;    // 错误代码（部分中断会压入）
void (*eip)(void);    // 中断返回地址
uint32_t cs;
uint32_t eflags;
void* esp;
uint32_t ss;
};
```

线程状态的管理

定义了线程的多种状态（如 `TASK_RUNNING`、`TASK_READY`、`TASK_BLOCKED` 等），用于表示线程在不同阶段的运行状态，为后续线程的调度和管理提供支持。

```
enum task_status {
    TASK_RUNNING,
    TASK_READY,
    TASK_BLOCKED,
    TASK_WAITING,
    TASK_HANGING,
    TASK_DIED
};
```

实现的思路

PCB 的设计与用途

`struct task_struct` 是线程的核心数据结构，记录了线程的运行状态和必要的信息。线程的内核栈指针 `self_kstack` 用于记录线程的当前栈顶位置，确保线程在被切换回处理器时能够正确恢复上下文信息。`status` 用于标记线程状态，配合调度器实现线程的状态转换。`priority` 作为线程调度的重要参考，决定线程的时间片分配。`stack_magic` 作为安全标记，用于检测栈是否发生溢出。

线程栈的作用

`struct thread_stack` 是线程运行时的重要栈结构。在线程首次运行时，栈中的 `eip` 指向内核线程函数 `kernel_thread`，而 `function` 和 `func_arg` 分别保存用户定义的待执行函数及其参数。通过汇编 `ret` 指令，将线程切换到 `kernel_thread` 执行。在任务切换时，`thread_stack` 的前四个寄存器成员（`ebp`、`ebx`、`edi`、`esi`）被用于保存和恢复主调函数的寄存器状态，确保任务切换的上下文完整性。

中断栈的设计与功能

`struct intr_stack` 是线程内核栈的一部分，用于在中断发生时保存线程的上下文信息。处理器会在进入中断时按照固定顺序将寄存器及相关信息压入中断栈。这一设计使得中断退出时能够通过逆序出栈恢复上下文，确保中断处理对线程执行流的影响最小。

首次调度的设计

首次调度的关键在于通过线程栈的 `eip` 将线程引导到 `kernel_thread` 函数执行。在 `kernel_thread` 内部，通过调用用户定义的函数 `function(func_arg)` 完成线程的初始化任务。这种设计通过汇编指令 `ret` 实现函数跳转，并且通过占位参数 `unused_retaddr` 确保线程栈布局符合 C 调用约定，避免栈帧错乱。

任务切换与线程状态管理

为任务切换准备了线程栈和中断栈两个关键结构。线程切换由汇编函数 `switch_to` 完成，通过保存和恢复线程栈的上下文实现线程间的无缝切换。线程状态的管理通过 `status` 标记实现，配合调度器在运行态、就绪态、阻塞态等状态之间进行切换。

设计的扩展性

该设计虽然是线程管理的初步实现，但其核心结构和机制（如 PCB、线程栈和中断栈）为后续功能扩展提供了基础。例如，可以进一步增加线程调度算法、线程同步机制、进程支持以及用户态与内核态的切换机制，使得系统具备更完善的多任务管理能力。

线程的实现

线程的实现主要围绕线程的创建、初始化、运行环境配置和首次调度展开。线程的核心是通过线程控制块（PCB）和线程栈协同工作，实现线程的独立运行。以下是具体的实现思路与流程：

线程的创建与初始化（thread_start）

线程的创建由 `thread_start` 函数完成。`thread_start` 首先调用 `get_kernel_pages` 为线程分配一页内核空间，其中一部分作为线程的 PCB，另一部分用作线程的内核栈。随后，通过调用 `init_thread` 函数，将线程的基本信息（如名称、优先级和状态）写入 PCB，状态初始设置为 `TASK_READY`。此外，栈顶指针 `self_kstack` 被设置为该页的顶部，为线程提供专属的内核栈，并在 PCB 中设置栈边界标记 `stack_magic`，以便检测栈溢出。

线程栈的初始化与运行环境配置（thread_create）

在 `thread_create` 函数中，为线程分配运行时的栈空间。线程栈的布局包括两个部分：中断栈 `intr_stack` 和线程栈 `thread_stack`。中断栈预留空间用于线程进入中断时保存寄存器状态，而线程栈用于保存线程的上下文及函数调用信息。在线程栈中设置入口函数 `kernel_thread`，并将用户定义的函数 `function` 和参数 `func_arg` 写入线程栈。此外，初始化必要的寄存器值（如 `ebp`、`ebx`、`edi` 和 `esi`）为 0，以保证线程运行的环境完整且稳定。

内核线程的执行（kernel_thread）

`kernel_thread` 是线程首次运行的入口函数，其地址被设置为线程的 `eip`。在线程首次被调度时，`kernel_thread` 会调用用户指定的函数 `function(func_arg)`，实现线程的实际任务。这种机制使用户可以灵活定义线程的行为，同时内核通过统一的入口函数管理线程的首次运行。

线程的运行流程

线程创建：调用 `thread_start`，分配内核空间，初始化 PCB 和内核栈。

运行环境配置：在 `thread_create` 中设置中断栈和线程栈，并配置入口函数 `kernel_thread`。

首次调度：通过汇编指令加载线程栈顶指针，恢复寄存器值，并通过 `ret` 跳转到线程入口函数。

任务执行：线程首次运行后，`kernel_thread` 调用用户定义的函数 `function(func_arg)`，完成指定的任务。

双向链表的数据结构设计

双向链表的数据结构通过两个核心结构体实现：`struct list_elem` 和 `struct list`。

- `struct list_elem` 表示链表中的单个结点，仅包含两个指针：`prev`（前驱指针）和 `next`（后继指针），用于连接链表中的结点，不存储具体的数据内容。
- `struct list` 表示整个链表，包含两个固定的结点 `head` 和 `tail`，分别作为链表的起始和结束标记。`head` 和 `tail` 本身不存储数据，仅起到链表结构标记的作用。

以下是双向链表的基本数据结构设计代码：

```
/* 链表结点结构 */struct list_elem {
    struct list_elem* prev; // 前驱结点
    struct list_elem* next; // 后继结点
};
/* 链表结构 */struct list {
    struct list_elem head; // 链表头部（固定，不存储数据）
    struct list_elem tail; // 链表尾部（固定，不存储数据）
};
```

这种设计使链表更加通用，可以灵活地用于不同场景下的队列和栈操作。

实现的功能

1. 链表初始化 (`list_init`)

功能：初始化一个链表，将链表置为空，仅包含头部和尾部标记结点。

2. 结点插入 (`list_insert_before`)

功能：将一个新结点插入到指定结点之前。

3. 在链表头部插入结点 (`list_push`)

功能：将结点插入到链表的头部（`head` 的后面），实现类似栈的后进先出功能。

4. 在链表尾部追加结点 (list_append)

功能：将结点追加到链表尾部 (tail 的前面)，实现类似队列的先进先出功能。

5. 删除结点 (list_remove)

功能：从链表中移除指定的结点。

6. 从链表头部弹出结点 (list_pop)

功能：移除并返回链表头部第一个结点 (head.next)，实现类似栈的出栈操作。

7. 查找指定结点 (elem_find)

功能：在链表中查找是否包含指定的目标结点，返回 true 或 false。

8. 条件查找结点 (list_traversal)

功能：通过遍历链表，调用回调函数逐个判断结点是否符合条件，返回第一个符合条件的结点指针。

9. 计算链表长度 (list_len)

功能：返回链表中结点的个数 (不包括 head 和 tail)。

10. 判断链表是否为空 (list_empty)

功能：判断链表是否为空 (head.next == tail)，空时返回 true，否则返回 false。

多线程调度模块设计与实现

多线程调度是操作系统的核心功能之一，其作用是让多个线程能够在有限的 CPU 资源上轮流执行。本模块基于简单的优先级调度算法，实现了线程的创建、管理和切换功能。通过扩展线程的控制块 (PCB) 以及引入全局线程队列，我们完成了从单线程到多线程的调度机制。

数据结构设计

多线程调度模块的核心是对线程 PCB 的扩展以及全局线程队列的引入。在 PCB 结构 `struct task_struct` 中新增了以下关键字段：

- `ticks`：表示线程在当前调度周期内的剩余时间片。每次线程被调度到 CPU 上运行时，`ticks` 初始化为线程优先级 `priority`，并在时钟中断触发时逐步递减，直至为 0，触发线程切换。
- `elapsed_ticks`：记录线程从创建到当前时刻消耗的总 CPU 时间，用于统计线程的运行情况。
- `general_tag` 和 `all_list_tag`：作为线程在不同队列中的唯一标识。`general_tag` 用于就绪队列 `thread_ready_list`，`all_list_tag` 用于全部线程队列 `thread_all_list`。
- `pgdir`：进程的页表地址。线程共享进程地址空间，故此字段为 NULL；而进程会将其页表地址存入此字段。

同时，引入两个全局线程队列：

- `thread_ready_list`：存储所有处于就绪状态的线程。调度器从此队列中选择下一个运行的线程。
- `thread_all_list`：存储系统中创建的所有线程，无论其处于何种状态，用于系统的统一管理和调试。

核心功能实现

功能实现

多线程调度模块实现了线程的创建、管理和切换，使多个线程能够在处理器上轮流执行。主线程通过 `make_main_thread` 函数被封装为普通线程，加入全部线程队列 `thread_all_list`，以便统一管理。新线程通过 `thread_start` 创建，初始化其 PCB 和栈后，加入就绪队列 `thread_ready_list` 和全部线程队列。线程首次运行由 `kernel_thread` 调用用户指定的函数完成，并在执行前开启中断，确保时钟中断能够正常调度线程。

调度机制基于时间片轮转和优先级。每个线程的时间片由其优先级决定 (`ticks = priority`)，运行时逐步递减至 0 后触发线程切换。时钟中断驱动调度器从 `thread_ready_list` 中选择下一个线程运行，实现多线程的公平竞争和轮转执行。

多线程调度模块实现思路

模块通过扩展线程 PCB 和引入全局队列实现线程管理。新增的 `ticks` 字段记录线程剩余时间片，`general_tag` 和 `all_list_tag` 用于将线程加入就绪队列和全部线程队列，轻量化队列操作并提高管理效率。主线程直接初始化已有的栈和 PCB，无需重新分配内存。新线程通过动态分配 PCB，并设置优先级和栈布局，保证调度时能正确执行用户函数。

调度基于优先级时间片算法，通过时钟中断触发线程切换。当前线程的时间片耗尽后，调度器将其加入就绪队列尾部，选择下一个线程运行。线程的首次运行通过预设栈布局调用用户函数，并开启中断以确保调度机制正常运作。整个模块通过统一的队列管理和调度算法，实现了多线程的高效切换与公平运行。

任务调度器和任务切换模块设计与实现

功能实现

任务调度器和任务切换模块的核心功能是实现多线程的任务调度与切换。调度器的职责是根据任务状态管理线程，确保线程能够按照优先级和时间片轮转调度。任务切换则负责保存当前线程的上下文环境并加载下一个线程的上下文，使多个线程能够在处理器上切换运行。

调度器 `schedule` 的功能是管理就绪队列 `thread_ready_list`，将当前线程（时间片耗尽时）重新加入队列末尾，并从队列中弹出下一个线程作为新的运行线程。当前线程的 `ticks`（剩余时间片）被重置为其优先级值，状态从

`TASK_RUNNING` 转为 `TASK_READY`，而下一个线程的状态被设置为 `TASK_RUNNING`。调度器通过函数 `switch_to` 切换到新的线程运行，完成任务切换。

时钟中断处理程序 `intr_timer_handler` 是任务调度的触发点。它负责递减当前线程的时间片 `ticks`，当时间片耗尽时调用调度器切换到其他线程。同时记录内核运行的总时钟滴答数 `ticks` 和当前线程的运行时间 `elapsed_ticks`，用于系统统计和分析。

任务切换由汇编函数 `switch_to` 实现。它接受两个参数，分别为当前线程和即将切换到的线程。`switch_to` 保存当前线程的上下文，包括 `esi`、`edi`、`ebx` 和 `ebp` 等寄存器的值，并将其存储到当前线程的 `PCB` 中；同时加载下一个线程的上下文，使其寄存器恢复到被换下时的状态。如果下一个线程是首次运行，则其返回地址指向 `kernel_thread`，由其执行用户指定的任务函数。

线程相关的初始化通过 `thread_init` 完成。此函数初始化了就绪队列和全部线程队列，并将主线程封装为普通线程，使其能够纳入调度管理体系中。通过封装主线程并初始化其他线程信息，调度器可以统一管理所有线程，实现完整的任务调度和切换功能。

实现思路

任务调度模块以队列为核心，管理线程的状态和调度顺序。所有待运行的线程被维护在就绪队列 `thread_ready_list` 中，通过先进先出的顺序轮流分配处理器执行。当线程时间片耗尽时，调度器将其重新加入队列末尾，并从队列中选择下一个线程运行。线程的优先级通过其时间片 `ticks` 体现，时间片越大，线程运行的时间越长。每次中断都会递减当前线程的 `ticks`，实现轮转调度。

调度器 `schedule` 的设计遵循两个原则：当线程因时间片耗尽被换下时，必须将其重新加入就绪队列，并重置时间片；当线程因其他原因（如阻塞）被换下时，不需要加入就绪队列，而是直接选择下一个线程运行。调度器通过读取就

绪队列的头部结点（线程标签 `general_tag`），将其转换为线程 `PCB`，并调用 `switch_to` 完成线程切换。

任务切换通过汇编函数 `switch_to` 实现。`switch_to` 保存当前线程的寄存器状态，并将栈指针 `esp` 保存到线程的 `PCB` 中，同时从即将运行的线程 `PCB` 中恢复栈指针和寄存器状态。线程首次运行时，栈中的返回地址指向 `kernel_thread`，由其负责调用用户指定的任务函数并执行。

时钟中断处理程序 `intr_timer_handler` 是调度器的触发机制。每次时钟中断都会更新系统时钟滴答数 `ticks`，递减当前线程的时间片 `ticks`，并在其耗尽时调用 `schedule`。为了保证线程切换的正确性，调度器要求中断必须被关闭，这由中断处理程序保证。

线程相关的初始化通过 `thread_init` 完成。此函数初始化两个全局线程队列，并封装主线程，使其具备线程的所有属性。这种设计使主线程与其他线程在调度上具有一致性，便于统一管理。

进程管理

进程管理这一模块中，包括进程的创建与复制、进程的执行与用户空间管理、进程的调度与切换等内容。通过复制父进程的资源为子进程提供独立的运行环境，并确保子进程能够正确执行。进程调度通过时间片轮转和优先级调度算法来公平分配 CPU 时间，而进程切换通过保存和恢复任务状态实现多任务的并发执行，任务状态段（TSS）则在任务切换中提供了必要的上下文保存和恢复机制。

1. 进程的创建与复制

1) 复制父进程的 `PCB` 和内存资源

通过 `memcpy` 复制父进程的 `PCB` 和内存空间，但对于内存部分的资源，

父子进程需要独立，因此通过分配新的内存位图和独立的内存资源给予进程。设置子进程的 pid 和 parent_pid，并确保其状态为就绪状态 TASK_READY。

```
int32_t copy_pcb_vaddrbitmap_stack0(struct task_struct* child_thread,
struct task_struct* parent_thread) {
    // 复制父进程的 PCB 和内存资源
    memcpy(child_thread, parent_thread, PG_SIZE);

    // 设置子进程的 PID 和状态
    child_thread->pid = fork_pid();
    child_thread->status = TASK_READY;
    child_thread->ticks = child_thread->priority;
    child_thread->parent_pid = parent_thread->pid;

    // 初始化块描述符等资源
    block_desc_init(child_thread->u_block_desc);

    // 分配新的虚拟内存位图给予进程
    uint32_t bitmap_pg_cnt = DIV_ROUND_UP((0xc0000000 -
USER_VADDR_START) / PG_SIZE / 8, PG_SIZE);
    void* vaddr_btmap = get_kernel_pages(bitmap_pg_cnt);
    memcpy(vaddr_btmap, child_thread->userprog_vaddr.vaddr_bitmap.bits,
bitmap_pg_cnt * PG_SIZE);
    child_thread->userprog_vaddr.vaddr_bitmap.bits = vaddr_btmap;

    // 设置子进程的名称
    strcat(child_thread->name, "_fork");

    return 0;
}
```

首先，通过 memcpy 将父进程的进程控制块（PCB）复制到子进程，并为子进程分配一个新的 PID。接着，设置子进程的状态为就绪，初始化子进程的内存块描述符，并为其分配新的虚拟内存位图，确保子进程的内存资源独立于父进程。此外，子进程的名称通过拼接 “_fork” 后缀来区分父进程。

2) 复制父进程的用户内存空间

子进程从父进程复制已分配的内存。根据父进程的虚拟地址位图，逐页复制内存内容，并确保在子进程中正确地设置页表，避免父子进程共享内存。

```
void copy_body_stack3(struct task_struct* parent_thread, struct
task_struct* child_thread, void* buf_page) {
    uint8_t* vaddr_btmap =
```



```

parent_thread->userprog_vaddr.vaddr_bitmap.bits;
    uint32_t btmp_bytes_len =
parent_thread->userprog_vaddr.vaddr_bitmap.btmp_bytes_len;
    uint32_t vaddr_start = parent_thread->userprog_vaddr.vaddr_start;

    // 根据父进程的虚拟地址位图，复制已分配的内存
    uint32_t idx_byte = 0, idx_bit = 0, prog_vaddr = 0;
    while (idx_byte < btmp_bytes_len) {
        if (vaddr_btmp[idx_byte]) {
            idx_bit = 0;
            while (idx_bit < 8) {
                if ((BITMAP_MASK << idx_bit) & vaddr_btmp[idx_byte])
                {
                    prog_vaddr = (idx_byte * 8 + idx_bit) * PG_SIZE +
vaddr_start;

                    memcpy(buf_page, (void*)prog_vaddr, PG_SIZE);
                    page_dir_activate(child_thread);
                    get_a_page_without_opvaddrbitmap(PF_USER,
prog_vaddr);

                    memcpy((void*)prog_vaddr, buf_page, PG_SIZE);
                    page_dir_activate(parent_thread);
                }
                ++idx_bit;
            }
            ++idx_byte;
        }
    }
}

```

通过遍历父进程的虚拟地址位图，找到已分配的内存页，并逐页复制到子进程。首先，根据父进程的位图，计算出每个已分配页的虚拟地址，并将该页的内容复制到临时缓冲区 `buf_page`。然后，切换到子进程的页表，通过 `get_a_page_without_opvaddrbitmap` 为子进程分配相应的页，并将缓冲区内容复制到子进程的内存中。最后，切换回父进程的页表，继续执行下一个内存页的复制。

3) 构建子进程的内核栈

确保它具有正确的返回地址、寄存器状态和程序计数器，保证子进程在中断返回时能够正确执行。

2. 进程的执行与用户空间管理

1) 用户进程创建与初始化

首先，创建一个新的用户进程，包括分配 PCB、设置名称、优先级、虚拟地址位图等。这些操作保证了每个进程都能够拥有独立的内存空间和运行环境。

将子进程的入口函数（start_process）设置为进程的执行起始点，并传递 filename 参数。这是为了确保进程能够从用户态正确启动。

2) 用户进程的启动

```
void start_process(void* filename) {
    void* function = filename_;
    struct task_struct* cur = running_thread(); // 获取当前正在运行的线程
    struct intr_stack* proc_stack = (struct
intr_stack*)((uint32_t)cur->self_kstack + sizeof(struct
thread_stack));
    proc_stack->edi = proc_stack->esi = proc_stack->ebp =
proc_stack->esp_dummy = 0;
    proc_stack->ebx = proc_stack->edx = proc_stack->ecx =
proc_stack->eax = 0;
    proc_stack->gs = 0;
    proc_stack->ds = proc_stack->es = proc_stack->fs =
SELECTOR_U_DATA; // 设置数据段选择子
    proc_stack->eip = function; // 设置进程入口函数
    proc_stack->cs = SELECTOR_U_CODE; // 设置代码段选择子
    proc_stack->eflags = (EFLAGS_IOPL_0 | EFLAGS_MBS | EFLAGS_IF_1);
// 设置标志寄存器
    proc_stack->esp = (void*)((uint32_t)get_a_page(PF_USER,
USER_STACK3_VADDR) + PG_SIZE); // 设置栈顶
    proc_stack->ss = SELECTOR_U_DATA; // 设置栈段选择子
    asm volatile ("movl %0,%esp;jmp intr_exit" :: "g"(proc_stack) :
"memory");
}
```

先获取当前线程的 PCB，并设置其栈信息。proc_stack 是进程的中断栈，它存储了进程执行时需要保存的寄存器值和其他状态信息。代码将 proc_stack 设置为 0 初始化，然后设置了进程的数据段选择子（ds, es, fs）、代码段选择子（cs）、进程入口函数（eip），以及栈顶（esp）。通过设置标志寄存器（eflags），确保进程能够启用中断。接着，调用 asm 指令将栈指针（esp）设置到

proc_stack, 并跳转到 intr_exit, 实现从内核模式切换到用户模式, 开始执行用户程序。

3) 创建页目录

每个进程都需要有一个独立的页目录, 为用户进程创建页目录, 并复制内核页目录的部分内容, 使得进程能访问内核空间。然后通过页目录和页表管理, 进程的虚拟地址被映射到实际的物理内存地址。页目录的最后一项指向其自身, 用于递归映射。

```
uint32_t* create_page_dir(void) {
    uint32_t* page_dir_vaddr = get_kernel_pages(1); // 分配一页内存来存储页目录
    if (page_dir_vaddr == NULL) {
        console_put_str("create_page_dir: get_kernel_page
failed!\n");
        return NULL;
    }

    memcpy((uint32_t*)((uint32_t)page_dir_vaddr + 0x300 * 4),
(uint32_t*)(0xffffffff000 + 0x300 * 4), 1024); // 复制内核页目录
    uint32_t new_page_dir_phy_addr =
addr_v2p((uint32_t)page_dir_vaddr);
    page_dir_vaddr[1023] = new_page_dir_phy_addr | PG_US_U | PG_RW_W
| PG_P_1; // 设置页目录的最后一项指向自身
    return page_dir_vaddr;
}
```

4) 初始化虚拟地址位图

需要为每个进程创建虚拟地址位图, 确保每个进程的虚拟地址空间可以独立管理, 避免多个进程之间的内存干扰, 并且用于跟踪虚拟地址空间中哪些页已经被分配, 哪些是空闲的。

3. 进程的调度与切换

1) 进程调度

实现了一个结合时间片轮转和优先级调度的调度算法。首先, 代码通过 `cur->ticks = cur->priority`; 将每个进程的时间片设置为其优先级, 优先级高的进程会获得更多的时间片。时间片用尽后, 进程将被挂起并放回到就绪队列中,

等待下一轮调度。这样，时间片轮转保证了进程间的公平性，每个进程都有机会在 CPU 上运行。

在调度过程中，首先会检查当前进程的状态。如果当前进程的状态是 TASK_RUNNING，则表示它还在执行，此时将其加入就绪队列 `thread_ready_list` 并将状态设置为 TASK_READY。通过这种方式，当前进程不会被丢弃，它会等待下次调度时继续运行。如果当前没有其他就绪的进程，操作系统会唤醒空闲线程 `idle_thread`，使 CPU 进入空闲状态，直到有新的进程可执行。

接下来，操作系统从就绪队列中弹出下一个就绪的进程，并将其状态设置为 TASK_RUNNING，表示它可以开始执行。通过 `process_activate(next)` 激活该进程的页目录，确保进程能够访问其独立的虚拟内存空间。最后，调用 `switch_to(cur, next)` 实现上下文切换，将 CPU 的控制权从当前进程切换到下一个进程。在上下文切换过程中，操作系统会保存当前进程的执行状态（如寄存器值、程序计数器等），并恢复下一个进程的状态，确保进程能从正确的地方继续执行。

```
void schedule(void) {
    ASSERT(intr_get_status() == INTR_OFF); // 确保在关闭中断的情况下进行调度

    struct task_struct* cur = running_thread(); // 获取当前正在运行的线程
    if (cur->status == TASK_RUNNING) {
        ASSERT(!elem_find(&thread_ready_list, &cur->general_tag));
        // 当前进程不在就绪队列中
        list_append(&thread_ready_list, &cur->general_tag); // 将当前进程加入就绪队列

        cur->status = TASK_READY; // 设置当前进程状态为就绪
        cur->ticks = cur->priority; // 重置当前进程的时间片
    }

    if (list_empty(&thread_ready_list)) {
        thread_unblock(idle_thread); // 如果没有就绪线程，则唤醒空闲线程
    }

    struct task_struct* thread_tag = list_pop(&thread_ready_list);
    // 获取下一个就绪线程
```

```

    struct task_struct* next = (struct
task_struct*)((uint32_t)thread_tag & 0xffffffff000); // 获取线程 PCB 地
址
    next->status = TASK_RUNNING; // 设置下一个线程的状态为运行中

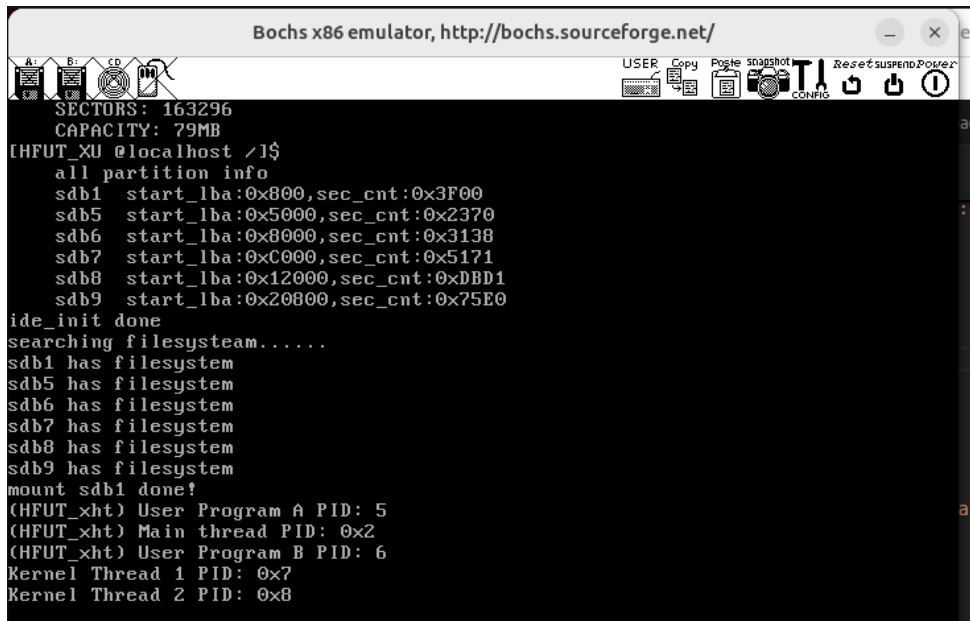
    process_activate(next); // 激活进程的页目录
    switch_to(cur, next); // 进行上下文切换
}

```

2) 任务切换和 TSS 的更新

设计思路核心在于保存当前任务的状态并恢复下一个任务的状态,从而实现多任务并发执行。每当任务切换时,操作系统需要将当前任务的上下文(如寄存器值、程序计数器等)保存到任务控制块(PCB)中,并从下一个任务的 PCB 中恢复其上下文。这样,任务能够在切换后继续执行。为了高效完成上下文切换,操作系统使用任务状态段(TSS)来存储每个任务的栈指针和其他关键状态信息,从而保证任务切换的快速性和准确性。

在任务切换时,操作系统通过更新任务状态段(TSS)中的 esp0 字段来保存当前任务的内核栈指针。每个任务有独立的内核栈,esp0 指向该任务的栈顶。任务切换时,操作系统调用 updata_tss_esp 函数,计算当前任务的内核栈位置,并更新 TSS 中的 esp0 字段。先计算出任务栈的顶部位置,然后将其赋值给 TSS 中的 esp0。在任务切换前,操作系统会更新当前任务和下一个任务的 esp0,确保它们的栈指针指向正确的位置,从而实现任务的上下文切换。



```
Bochs x86 emulator, http://bochs.sourceforge.net/
SECTORS: 163296
CAPACITY: 79MB
[HFUT_XU @localhost /l$
all partition info
sdb1 start_lba:0x800,sec_cnt:0x3F00
sdb5 start_lba:0x5000,sec_cnt:0x2370
sdb6 start_lba:0x8000,sec_cnt:0x3138
sdb7 start_lba:0xC000,sec_cnt:0x5171
sdb8 start_lba:0x12000,sec_cnt:0xDBD1
sdb9 start_lba:0x20800,sec_cnt:0x75E0
ide_init done
searching filesystem.....
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
(HFUT_xht) User Program A PID: 5
(HFUT_xht) Main thread PID: 0x2
(HFUT_xht) User Program B PID: 6
Kernel Thread 1 PID: 0x7
Kernel Thread 2 PID: 0x8
```

进程调度，创建测试

设备管理

1. 键盘驱动程序

1.1 原理简介

键盘通过中断向操作系统通报事件，硬件会在每次按键或释放时触发中断信号，CPU 通过中断向量表找到对应的中断处理程序，完成扫描码的读取和处理。因此，设置键盘驱动的第一步是绑定键盘中断向量到自定义的中断处理程序，这使得键盘输入能够即时被捕获和处理。

扫描码是键盘硬件生成的编码，用于表示按键的状态变化，包括通码（按键按下）和断码（按键释放）。每个键对应一个唯一的扫描码，当某些按键（如组合键）涉及多字节扫描码时，驱动程序需要结合硬件协议判断其完整性并正确解析。通过扫描码映射表（如 keymap），可以将这些硬件编码转换为可识别的字符，进而被用户程序读取。

特殊按键（如 Shift、Ctrl、Caps Lock）的状态需要通过标志位进行维护。其原理在于通过扫描码判断特殊按键的按下或释放事件，并记录其状态，供后续

扫描码解析使用。例如，当 Shift 按下时，驱动程序调整字符映射规则，使得按键产生大写字母或符号；Caps Lock 则通过切换标志位实现字母区的大小写模式切换。这些状态的实时维护和解析，确保了键盘功能的完整性和用户输入的准确性。

1.2 设计目标

本键盘驱动程序的设计目标是实现从键盘硬件捕获按键中断，读取扫描码并解析为对应的字符，同时支持功能键（如 Shift、Ctrl、Caps Lock）的状态管理，处理多字节扫描码的复杂情况，并通过线程安全的环形缓冲区缓存输入数据，确保在多任务操作系统中实现高效、可靠的键盘输入功能。

1.3 系统设计

1.3.1 数据结构

- 1) 环形缓冲区：用于存储解析后的按键字符，确保输入的线程安全。
- 2) 扫描码映射表：记录按键在普通状态和 Shift 状态下的字符。
- 3) 功能键状态变量：记录功能键是否被按下。

1.3.2 核心模块

1) 键盘初始化

键盘初始化的核心是注册中断处理程序和初始化相关的数据结构。首先，将键盘中断号（通常是 0x21）与自定义的键盘中断处理程序绑定，使得键盘硬件在按键事件发生时能够触发操作系统的中断处理。

其次，初始化环形缓冲区结构，完成缓冲区头尾指针的设置、锁的初始化以及生产者与消费者线程指针的清空，确保了缓冲区在多任务环境下的数据安全与同步。以及对键盘硬件进行必要的配置，如设置键盘控制器的相关寄存器以启用中断信号。

2) 中断处理

键盘按键事件发生时，硬件触发中断，CPU 会调用绑定的中断处理程序。在中断处理程序中，首先通过键盘数据端口（通常是 0x60）读取扫描码。通过最高位的标志判断扫描码是通码（按下）还是断码（释放）。如果扫描码为扩展码

（如多字节组合键的起始字节 0xe0），则标记为扩展状态，等待后续字节的处理。

对于通码，调用扫描码映射逻辑，将其转换为对应的字符或控制信号；对于断码，则更新特殊按键的状态（如 Shift 或 Ctrl 释放）。最后，处理程序会将解析后的字符或事件放入环形缓冲区，为用户程序提供后续读取使用。

3) 扫描码映射

通过预定义的扫描码表（keymap）进行映射，将扫描码索引到对应的字符。该表是一个二维数组，第一维表示未按下 Shift 键的普通字符，第二维表示按下 Shift 键的字符。对于功能键（如 Shift、Ctrl），通过在中断处理程序中维护状态标志位来影响映射过程，比如，Shift 键按下时，映射逻辑会选择 keymap 的第二维。

4) 线程同步

使用环形缓冲区作为输入存储区，并通过互斥锁和阻塞机制实现生产者-消费者模型。生产者是键盘中断处理程序，在按键事件发生时将解析出的字符放入缓冲区；消费者是用户线程，通过读取缓冲区获取输入数据。

2. 环形缓冲区管理

2.1 设计目标

- 1) **数据缓冲与存储：**设计一个固定大小的环形缓冲区，用于存储从生产者（如键盘中断处理程序）产生的数据，并提供给消费者（如用户程序）读取，确保数据不丢失且顺序正确。
- 2) **线程安全：**在多线程环境下，确保生产者与消费者对缓冲区的访问互不干扰，通过互斥锁和线程阻塞机制实现数据访问的安全性。
- 3) **高效资源利用：**利用环形结构，循环使用固定大小的缓冲区，避免动态内存分配和释放的开销，提高系统性能。

2.2 系统设计

2.2.1 数据结构

```
struct ioqueue {
```



```

char buf[bufsize];           // 固定大小的缓冲区
uint32_t head, tail;         // 头尾指针，分别表示写入和读取位置
struct task_struct* producer, *consumer; // 等待的生产者和消费者线程
struct lock lock;            // 用于线程同步的互斥锁
};

```

通过一个固定大小的数组 `buf` 存储数据，利用两个指针 `head` 和 `tail` 分别指向写入和读取的位置，实现数据的循环存取。为解决多线程环境中的竞争问题，缓冲区设计了互斥锁 `lock` 来确保线程安全，并记录阻塞的生产者和消费者线程 `producer` 和 `consumer`，当缓冲区满或空时，分别阻塞对应的线程，保证数据传输的高效和可靠性。

2.2.2 核心功能实现

1) 缓冲区初始化

包括初始化 `head` 和 `tail` 指针，清空生产者和消费者线程指针，以及初始化锁 `lock`。

2) 判断缓冲区状态

如果 `next_pos(head) == tail`，则缓冲区满。如果 `head == tail`，则缓冲区空。环形缓冲区通过取模实现索引循环，计算下一位置。

3) 数据读取

消费者从缓冲区中读取数据，如果缓冲区为空，需阻塞当前线程，等待生产者写入数据。否则从 `tail` 指针指向的位置读取数据，并将 `tail` 更新为下一个位置。此外，如果生产者线程因缓冲区满而阻塞，调用 `wakeup` 唤醒生产者。

4) 数据写入

生产者将数据写入缓冲区，如果缓冲区已满，也需阻塞当前线程，等待消费者读取数据，否则将数据写入 `head` 指针指向的位置，并将 `head` 更新为下一个位置。如果消费者线程因缓冲区空而阻塞，调用 `wakeup` 唤醒消费者。

5) 线程同步机制

设计两个函数实现线程阻塞和线程唤醒。如果缓冲区空（消费者）或满（生产者），阻塞当前线程。如果缓冲区状态变化，则唤醒阻塞的生产者或消费者。

3. 硬盘驱动程序

3.1 设计目标

- 1) **支持硬盘的自动识别：**通过硬盘控制器提供的接口，自动检测硬盘数量和属性信息。
- 2) **分区管理：**实现主分区和逻辑分区的扫描与记录。
- 3) **数据读写功能：**支持基于逻辑扇区地址（LBA）的扇区级数据读写操作。
- 4) **中断处理：**通过硬盘中断机制处理数据传输完成事件，提高效率。

3.2 核心功能实现

3.2.1 硬盘初始化

硬盘初始化主要目的是完成硬盘的检测、通道配置、设备识别和基本分区管理等工作，以下是硬盘初始化的详细实现和功能分解。

- 1) **硬盘数量检测：**通过 BIOS 提供的内存地址 0x475 获取。
- 2) **通道配置：**IDE 通道通过特定的端口进行通信。每个通道可以挂载两个设备（主盘和从盘），需要为每个通道设置端口基地址，中断号，互斥锁和信号量。
- 3) **硬盘设备初始化：**设置硬盘的通道和设备号（主盘为 0，从盘为 1），为硬盘设备分配名称，并且读取其基本信息（如序列号、型号和容量）。
- 4) **硬盘信息识别：**通过向硬盘发送 IDENTIFY 指令（0xEC），获取硬盘的基本信息，包括硬盘序列号，硬盘型号，总扇区数，从而计算硬盘容量。
- 5) **分区扫描：**读取分区表，记录主分区和逻辑分区信息。

3.2.2 硬盘分区扫描

分区信息存储在硬盘的主引导记录（MBR）或扩展分区引导扇区（EBR）的分区表中。每个分区表由 4 个分区表项组成，每个分区表项包含如下字段：

文件系统类型：指示分区类型，0x5 表示扩展分区，0x0 表示空分区。

起始扇区：分区的起始地址。

扇区数量：分区占用的总扇区数。

1) 分区表项的数据结构

```
struct partition_table_entry {  
    uint8_t bootable;           // 是否可引导  
    uint8_t start_head;        // 起始磁头号  
    uint8_t start_sec;         // 起始扇区号  
    uint8_t start_chs;         // 起始柱面号  
    uint8_t fs_type;           // 分区类型  
    uint8_t end_head;          // 结束磁头号  
    uint8_t end_sec;           // 结束扇区号  
    uint8_t end_chs;           // 结束柱面号  
    uint32_t start_lba;        // 起始 LBA 地址  
    uint32_t sec_cnt;          // 分区占用的总扇区数  
};
```

2) 分区扫描核心逻辑

从硬盘的第一个扇区（MBR）中读取主分区信息，如果分区表中存在扩展分区类型（fs_type == 0x5），则递归读取扩展分区的引导扇区，继续扫描其逻辑分区。

3) 分区信息记录

分区扫描完成后，将分区信息记录到全局分区队列中，以便操作系统的其他模块访问。

3.2.3 数据读写

进行数据读写的核心过程如下：

1) 选择目标硬盘

通过控制 reg_dev 寄存器来选择目标硬盘。

2) 设置目标扇区和扇区数

通过写入硬盘控制器的相关寄存器设置要访问的 LBA 和扇区数。

3) 发送读取命令或写入命令

向硬盘控制器发送 READ SECTOR 指令或 WRITE SECTOR 指令。

4) 等待读取或写入

读取时，检查硬盘状态寄存器，确保硬盘数据已准备好。写入时，确保硬盘可以接收数据。

5) 读取数据

核心代码如下：

```
void read_from_sector(struct disk* hd, void* buf, uint8_t sec_cnt) {
    uint32_t size_in_byte = sec_cnt == 0 ? 256 * 512 : sec_cnt * 512;
    insw(reg_data(hd->my_channel), buf, size_in_byte / 2); // 使用 insw 读取数据
    到内存
}
```

高效读取数据时，使用硬盘中断与信号量实现同步：

```
void ide_read(struct disk* hd, uint32_t lba, void* buf, uint32_t sec_cnt) {
    lock_acquire(&hd->my_channel->lock); // 加锁确保互斥
    select_disk(hd);                      // 选择硬盘

    uint32_t secs_op, secs_done = 0;
    while (secs_done < sec_cnt) {
        secs_op = (secs_done + 256 <= sec_cnt) ? 256 : sec_cnt - secs_done;
        select_sector(hd, lba + secs_done, secs_op);
        cmd_out(hd->my_channel, CMD_READ_SECTOR); // 发送读取命令

        sema_down(&hd->my_channel->disk_done); // 等待中断信号
        if (!busy_wait(hd)) { // 检查状态寄存器
            PANIC("Read sector failed!");
        }
        read_from_sector(hd, (void*)((uint32_t)buf + secs_done * 512), secs_op);
        secs_done += secs_op;
    }
    lock_release(&hd->my_channel->lock); // 释放锁
}
```

6) 写入数据

```
void write2sector(struct disk* hd, void* buf, uint8_t sec_cnt) {  
    uint32_t size_in_byte = sec_cnt == 0 ? 256 * 512 : sec_cnt * 512;  
    outsw(reg_data(hd->my_channel), buf, size_in_byte / 2); // 使用 outsw 写入数  
    据到硬盘  
}
```

同时也需要使用硬盘中断与信号量实现同步。

3.2.4 中断处理

通过硬件中断机制通知操作系统硬盘数据传输的完成，从而避免程序主动轮询等待硬盘响应，提高操作效率。具体包括：接收硬盘完成操作的中断信号，清除中断标志以保证后续中断正常触发；通过信号量机制唤醒因等待硬盘响应而阻塞的线程，实现线程与硬盘操作的同步。

1) 硬盘触发中断

硬盘完成数据传输（读写操作）后，通过触发中断信号通知处理器。中断信号通过 IRQ 通道 传递到 8259A 中断控制器，最终由 CPU 调用相应的中断处理程序。

2) 中断向量映射

硬盘中断通常由 IRQ14（主 IDE 通道）或 IRQ15（从 IDE 通道）触发，分别对应中断向量号 0x2E 和 0x2F。在初始化时，需要将硬盘中断号与具体的中断处理程序绑定。

3) 中断处理逻辑

首先，识别触发的中断号。然后通过读取硬盘的状态寄存器（reg_status）清除硬盘中断信号，还要通过信号量机制唤醒因等待硬盘操作而阻塞的线程，使其继续执行。核心代码如下：

```
void intr_hd_handler(uint8_t irq_no) {  
    ASSERT(irq_no == 0x2E || irq_no == 0x2F); // 硬盘中断向量号检查  
    uint8_t ch_no = irq_no - 0x20 - 0xE; // 计算通道号（0 表示主 IDE，1 表示
```

从 IDE)

```
struct ide_channel* channel = &channels[ch_no]; // 获取触发中断的 IDE 通道
ASSERT(channel->irq_no == irq_no); // 验证通道是否正确
if (channel->expecting_intr) { // 检查是否有等待的硬盘操作
    channel->expecting_intr = false; // 标记操作完成
    sema_up(&channel->disk_done); // 唤醒等待硬盘响应的线程
    inb(reg_status(channel)); // 读取状态寄存器，清除中断标志
}
}
```

4) 中断处理与线程同步

中断处理程序通过信号量实现线程同步。在硬盘操作发起后，线程调用 `sema_down` 等待信号量。如果信号量值为 0，则线程进入阻塞状态。硬盘操作完成后，中断处理程序调用 `sema_up` 增加信号量值，被阻塞的线程检测到信号量增加后，恢复执行。

文件系统

1.创建超级块、inode 和目录项（基础的数据结构）

（注：为方便写程序，我们代码中的数据块大小与扇区大小一致，即 1 块等于 1 扇区）

在文件系统的初始化过程中，需要创建三个核心数据结构：超级块（Super Block）、inode（索引节点）和目录项（Directory Entry）。这些数据结构是文件系统的基础，分别记录分区的元信息、文件的元信息以及文件名与 inode 的关联。下面是它们的定义和作用说明。

1.1 超级块（Super Block）

超级块是文件系统中用于记录分区全局信息的数据结构，主要包含分区的基本属性和数据布局。它包括的内容如下图所示：

魔数
数据块数量
inode 数量
分区起始扇区地址
空闲块位图地址
空闲块位图大小
inode 位图地址
inode 位图大小
inode 数组地址
inode 数组大小
根目录地址
根目录大小
空闲块起始地址
...

超级块的逻辑结构

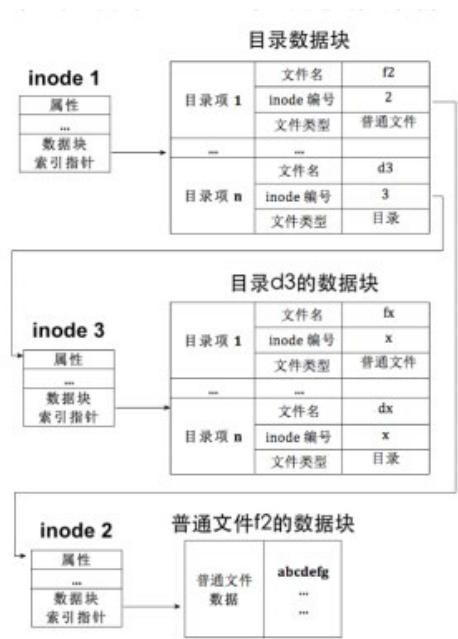
1.2 inode（索引节点）

inode 是文件系统中用于存储文件或目录元信息的核心数据结构，每个文件或目录都有一个唯一的 inode。它记录了文件或目录的大小（以字节为单位）、被打开的次数、写入限制标志（防止并发写操作）以及文件数据块的地址信息。文件的数据块地址包括 12 个直接数据块地址和 1 个一级间接块地址（支持最多 128 个间接数据块），总共支持 140 个数据块。

1.3 目录项（Directory Entry）

目录项（struct dir_entry）是文件系统中用于连接文件名与 inode 的数据结构，是目录的核心组成部分。目录项包含文件名、文件或目录对应的 inode 编号以及文件类型（普通文件或目录）。文件名的最大长度被限制为 16 个字符。

目录项与目录结构（struct dir）密切关联，其中目录结构在内存中使用，用于缓存目录数据和记录目录遍历的偏移量。目录项的作用是通过文件名快速定位 inode，从而找到文件或目录的元信息及数据块位置，它并不在磁盘上存在，只用于与目录相关的操作时，在内存中创建的结构，用过之后就释放了，不会回写到磁盘中。



索引结点与目录项的关系

2.创建文件系统(其实还是一些初始化的工作,初始化分区)

分区的空间分为以下几部分：

- **引导块:** 保留一个扇区，不实际使用。
- **超级块:** 用一个扇区存储分区的全局元信息。
- **inode 位图:** 记录 inode 的分配状态，每个 inode 对应一位。
- **inode 表:** 存储所有 inode 信息，每个 inode 的大小是固定的。
- **块位图:** 记录数据块的分配状态，每个块对应一位。
- **数据区:** 用于实际存储文件和目录的数据。

创建步骤如下：

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

(1) 根据分区大小计算元信息的扇区数及位置

引导块和超级块各固定占用一个扇区。`inode` 位图 的大小由最大 `inode` 数量 (4096) 决定，每个 `inode` 对应 1 位，其扇区数通过向上取整计算得出：

`inode_bitmap_sects` = `DIV_ROUND_UP(MAX_FILES_PER_PART, BITS_PER_SECTOR)`。

`inode` 表 的大小由 `inode` 数量和每个 `inode` 的固定大小 (`sizeof(struct inode)`) 计算得出。块位图 记录数据块的分配状态，通过计算空闲块数量并减去块位图占用的块数调整最终大小。数据区从 `inode` 表 结束的位置开始，用于存储文件和目录数据

(2) 在内存中创建超级块

超级块字段：

基本信息：

`magic`: 文件系统类型标识符 (0x23333333)。

`sec_cnt`: 分区的总扇区数。

`inode_cnt`: `inode` 的总数量。

`part_lba_base`: 分区的起始 LBA 地址。

元信息位置：

`block_bitmap_lba` 和 `block_bitmap_sects`: 块位图的起始地址和大小。

`inode_bitmap_lba` 和 `inode_bitmap_sects`: `inode` 位图的起始地址和大小。

`inode_table_lba` 和 `inode_table_sects`: `inode` 表的起始地址和大小。

`data_start_lba`: 数据区的起始地址。

根目录信息：

`root_inode_no`: 根目录的 `inode` 编号 (固定为 0)。

`dir_entry_size`: 目录项的大小。

(3) 将超级块写入磁盘

初始化完成后，超级块被写入引导块后的第一个扇区，即 `part->start_lba + 1`。

(4) 将元信息写入磁盘上的各自位置

在超级块写入后，文件系统的其他元信息（如块位图、inode 位图、inode 表）也需初始化并写入磁盘。由于这些元信息占用空间较大，需动态申请堆内存作为缓冲区，其大小由元信息的最大扇区数计算得出：

```
buf_size = max(block_bitmap_sects, inode_bitmap_sects,
inode_table_sects) * SECTOR_SIZE。
```

(5) 将根目录写入磁盘

最后一项工作是在根目录中写目录项“.”和“..”。任何目录都有这两个目录项，“.”表示当前目录，“..”表示上一级目录。每个目录项固定大小为 `dir_entry_size`，写入 `data_start_lba` 对应的第一个数据块。

3.文件系统初始化函数(filesys_init)

`filesys_init` 是文件系统的初始化函数，用于在磁盘上搜索分区，检测是否已有文件系统。如果分区不存在文件系统，或者检测到的文件系统不是本文件系统支持的类型，则对分区进行格式化，创建文件系统。这个函数在操作系统启动时调用，为文件系统的使用准备环境。

我们的内核对每个硬盘最多支持 12 个分区，4 个主分区和 8 个逻辑分区，因此在遍历硬盘分区时需要循环 12 次。`part` 用于指向每一个分区，当分区索引变量 `part_idx` 等于 4 时，这表示全部主分区都处理完了，将 `part` 指向硬盘的逻辑分区数组，也就是逻辑分区的第一个地址。同时进行格式化分区之前，我们先要判断分区是否存在，这是通过分区的 `sec_cnt` 是否为 0 来判断的，所以可以用此变量来判断，原因是分区 `part` 所在的硬盘作为全局数组 `channels` 的内嵌成员，全局变量会被初始化为 0。我们在扫描分区表的时候会把分区的信息写到 `part` 中，因此只要分区不存在，分区 `part` 中任意成员的值都会是 0，只是我们这里用 `sec_cnt` 来判断而已。如果我们在代码中判断出魔数为我

们设定的特定值， 就表示已经有文件系统，不再去格式它。否则后面要对此分区执行格式化。

4.挂载分区

与 Windows 不同，在 Linux 系统中，内核所在的分区是默认分区。自系统启动后，该分区会自动成为默认分区，并且它的根目录始终固定存在。如果需要其他分区，我们可以通过 `mount` 命令将这些分区挂载到默认分区的某个目录下，也就是“取出”这个分区供我们操作。虽然每个分区都有自己的根目录，但整个文件系统的根仍然是默认分区的根目录，所有分区都以它为起点连接起来。如果某个分区暂时不需要使用，还可以通过 `umount` 命令将其“收回”，也就是卸载掉。对于我们当前实现的挂载功能，操作方式相对简单，因为系统没有默认的路径树结构。挂载操作仅需要直接指定目标分区作为当前工作的分区，而不需要考虑将它挂载到某个特定的目录下。

挂载分区的功能由 `filesys_init` 函数与 `mount_partition` 函数共同完成。

整体流程如下：

- **确定默认分区：** 设置默认的分​​区名，并通过分区链表遍历找到与之匹配的目标分区。
- **查找目标分区：** 遍历分区链表，定位目标分区的指针并将其设置为当前分区。
- **加载超级块：** 从硬盘读取目标分区的超级块内容到内存，并初始化分区的超级块指针。
- **加载块位图：** 读取分区的块位图到内存，管理分区数据块的分配和释放状态。
- **加载 inode 位图：** 加载 inode 位图到内存，管理分区 inode 的分配和释放状态。
- **初始化分区其他结构：** 完成分区辅助结构（如打开 inode 列表）的初始化，为后续操作做好准备。

步骤详解

1. 确定默认分区

通过字符串 `default_part` 指定默认分区名，并调用 `list_traversal` 遍历分区链表 `partition_list`，将每个分区节点传递给回调函数 `mount_partition`。如果回调函数匹配到分区名称，则将分区指针赋值给全局变量 `cur_part`，标记为当前操作分区，并提前结束遍历；若遍历完成仍未匹配到，则说明默认分区不存在。

2. 查找目标分区

`mount_partition` 是 `list_traversal` 的回调函数，用于比较分区名与默认分区名是否匹配。如果匹配成功，则将目标分区的指针赋值给 `cur_part`，为后续操作做好准备；若不匹配，则继续遍历链表的下一个分区，直到找到目标分区或遍历结束。

3. 加载超级块

加载超级块是挂载分区的第一步元信息操作。首先分配缓冲区 `sb_buf` 和超级块存储内存 `cur_part->sb`，从目标分区的引导扇区之后读取超级块数据。通过复制关键字段（如块位图和 `inode` 位图的位置）到 `cur_part->sb`，完成超级块的初始化，为分区内元信息的加载提供支持。

4. 加载块位图

根据超级块记录的块位图起始位置和扇区数，分配相应内存，并从硬盘中加载块位图数据到内存。块位图用于记录分区中数据块的分配状态，是分区文件写入和管理的重要基础。

5. 加载 `inode` 位图

`inode` 位图用于跟踪 `inode` 的分配状态，加载方式与块位图类似。根据超级块中记录的起始位置和扇区数，从硬盘加载 `inode` 位图数据到内存，为文件和目录的管理提供支持。

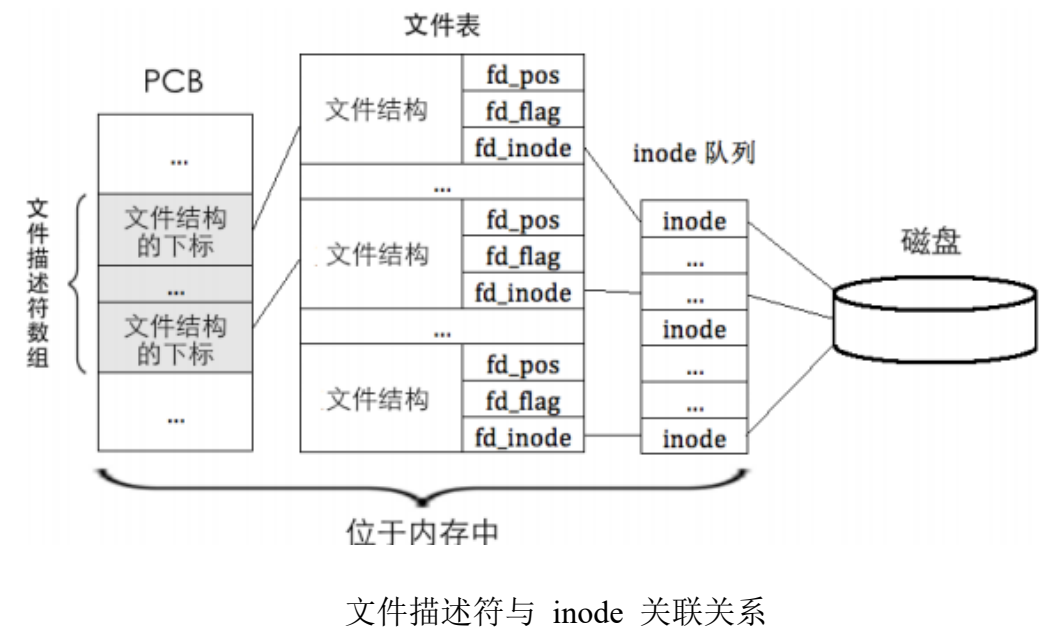
5.文件描述符：

⋮

fd_pos	// 文件偏移量
fd_flag	// 文件打开的标志如 O_CREAT
fd_inode	// inode 指针
...	

文件描述符逻辑结构

open 函数的返回值是一个数字，这个数字就是我们常说的文件描述符。文件描述符本质上只是一个整数，更具体地说，它是进程 PCB（进程控制块）中文件描述符数组的下标。这个数字并不表示“数量”，而是表示“位置”，它标明文件描述符在 PCB 的文件描述符数组中的具体位置。而文件描述符数组的每个元素会指向文件表中的某个文件结构，从而实现对文件的管理和访问。



创建文件描述符的过程就是依次在这三个数据结构中找到空位，并将相应的数据填充到这些空位中，最后返回文件描述符的位置地址。

6.文件操作相关的基础函数

Inode 操作相关函数

1. **inode_locate**: 定位指定 **inode** 在磁盘中的扇区位置和偏移量, 便于后续操作。
2. **inode_sync**: 将内存中的 **inode** 数据同步到磁盘, 更新文件元信息。
3. **inode_open**: 根据 **inode** 编号打开指定 **inode**, 加载到内存并返回指针。
4. **inode_close**: 关闭指定 **inode**, 减少打开计数, 释放相关资源。
5. **inode_init**: 初始化新建 **inode**, 设置初始状态, 如大小为 0, 块地址置 0。

文件相关函数

1. 文件表(**file_table**): 全局数组管理所有被打开的文件, 记录文件状态信息。
2. **get_free_slot_in_global**: 在文件表中寻找空闲位, 返回对应下标。
3. **pcb_fd_install**: 将全局文件表的条目安装到进程的文件描述符数组中。
4. **inode_bitmap_alloc**: 在 **inode** 位图中分配一个空闲 **inode**, 返回其编号。
5. **block_bitmap_alloc**: 在块位图中分配一个磁盘扇区, 返回其地址。
6. **bitmap_sync**: 将内存中的位图数据同步到硬盘, 确保一致性。

目录相关操作函数

1. **open_root_dir**: 打开指定分区的根目录, 初始化根目录相关信息。
2. **dir_open**: 打开指定目录的 **inode**, 返回目录指针。
3. **search_dir_entry**: 在目录中查找指定文件名的目录项, 返回其信息。
4. **dir_close**: 关闭指定目录并释放相关资源。
5. **create_dir_entry**: 初始化目录项结构, 设置文件名、**inode** 编号和类型。
6. **sync_dir_entry**: 将目录项写入父目录的空闲位置, 并同步到硬盘。

路径解析相关函数

1. **path_parse**: 解析路径中的最顶层名称, 并返回剩余路径。
2. **path_depth_cnt**: 计算路径深度(层级数)。
3. **search_file**: 搜索指定路径, 返回目标文件的 **inode** 编号, 并记录搜索状态。

功能总结:

通过以上函数的实现, 我们完成了文件和目录操作的核心功能, 包括文件的创建、删除、打开、关闭, 以及目录的搜索、遍历和管理, 同时实现了路径解析

和位图同步功能。这些函数共同构成了一个完整的文件系统操作框架，为文件和目录的高效管理提供了基础支持。

7.创建文件：

经过前期大量的准备工作，现在我们的操作系统已经具备了创建文件的能力，但是我们这里指的是创建普通文件，而不是目录文件，关于目录文件我们后面才会说到。

整理一下思路，下面是创建一个文件所需要做的工作：

(1) 文件的大小、位置等属性需要通过 inode 来描述，因此在创建文件时需要创建对应的 inode。这需要通过更新 inode_bitmap 来申请一个 inode 号，同时在 inode_table 数组中填充新的 inode 信息。

(2) 文件的实际存储位置由 inode->i_sectors 指定，这需要通过更新 block_bitmap 来申请可用块（在本实现中，1 块等于 1 扇区）。因此，分区数据区 data_start_lba 之后的某个扇区将会被分配。

(3) 新创建的文件必然存放在某个目录中，因此该目录的 inode->i_size 会增加一个目录项的大小。新增的文件对应的目录项需要写入该目录的 inode->i_sectors[] 所指向的某个扇区中。若原有扇区已满，可能需要申请新的扇区用于存储这些目录项。

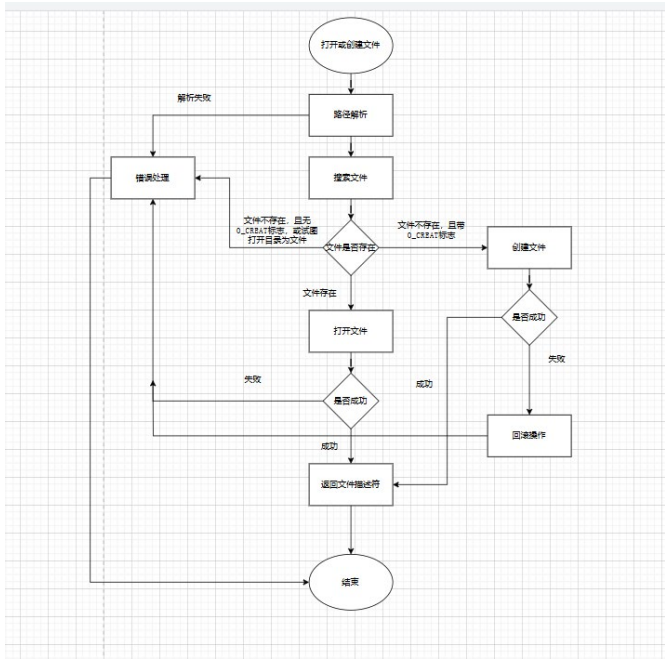
(4) 如果某一步操作失败，则需要回滚之前所有已成功完成的操作。

(5) 所有已经被修改的内存数据，包括 inode_bitmap、block_bitmap、新文件的 inode，以及文件所属目录的 inode，都需要同步更新到硬盘上。

整体流程如下：文件操作的整体流程包括路径解析、文件检查、资源分配、文件描述符分配，以及错误回滚机制。首先，通过路径解析逐层检查路径合法性并确认文件是否存在；若路径中存在错误，直接返回失败。文件创建时，会分配 inode 和磁盘块，更新元信息，同时在父目录生成目录项并同步到磁盘。在文件打开时，检查目标文件的合法性，并根据标志决定是否创建新文件或直接

返回文件描述符。若操作失败，则触发回滚机制，撤销已完成的步骤，确保文件系统一致性。文件系统初始化阶段加载根目录和全局文件表，为路径解析和文件操作提供基础支持。

至此，我们已经实现了文件的创建功能。



创建文件流程

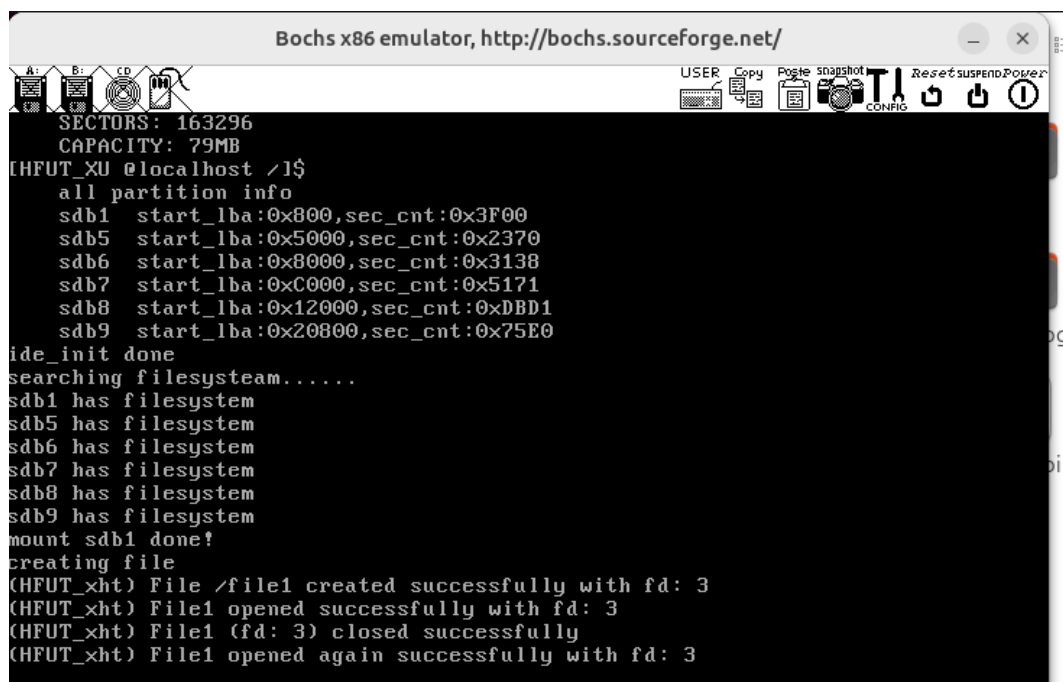
8.文件的操作

8.1 文件的打开

实现思路

文件的打开首先通过 `sys_open` 检查路径是否合法，并调用 `search_file` 判断目标文件是否存在。如果文件已存在且为普通文件，则调用 `file_open` 完成打开操作，将文件的 `inode` 加载到文件表 `file_table` 并初始化文件指针和标识符。当以写入模式打开文件时，通过设置 `write_deny` 避免多个进程同时写入。如果目标文件不存在但包含 `O_CREAT` 标志，则调用 `file_create`

创建文件后再打开。最后，将文件描述符安装到进程的文件描述符表 `fd_table` 中，并返回文件描述符。



```
Bochs x86 emulator, http://bochs.sourceforge.net/
SECTORS: 163296
CAPACITY: 79MB
[HFUT_XU @localhost /]$
all partition info
sdb1  start_lba:0x800,sec_cnt:0x3F00
sdb5  start_lba:0x5000,sec_cnt:0x2370
sdb6  start_lba:0x8000,sec_cnt:0x3138
sdb7  start_lba:0xC000,sec_cnt:0x5171
sdb8  start_lba:0x12000,sec_cnt:0xDBD1
sdb9  start_lba:0x20800,sec_cnt:0x75E0
ide_init done
searching filesystem.....
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
creating file
(HFUT_xht) File /file1 created successfully with fd: 3
(HFUT_xht) File1 opened successfully with fd: 3
(HFUT_xht) File1 (fd: 3) closed successfully
(HFUT_xht) File1 opened again successfully with fd: 3
```

文件的打开与关闭

8.2 文件的关闭

实现思路

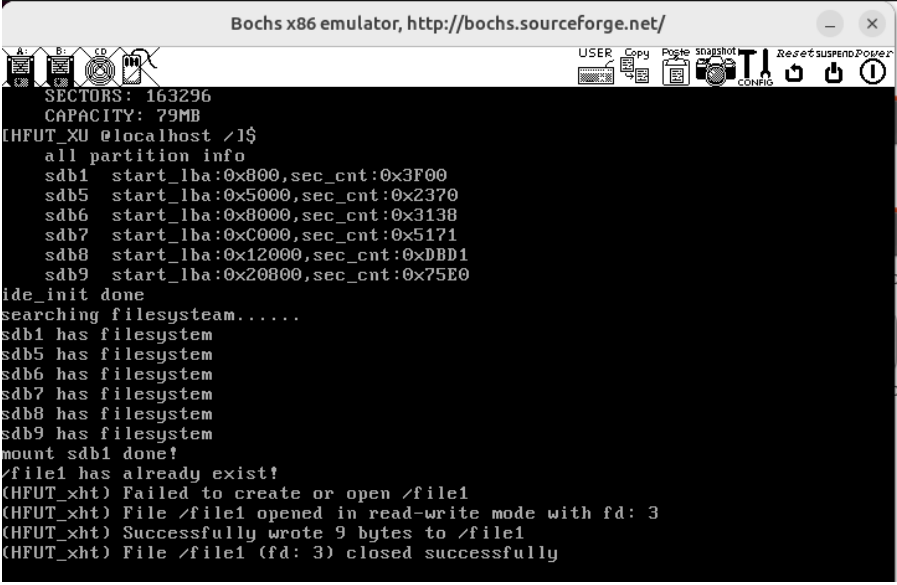
文件关闭时，`sys_close` 首先通过辅助函数 `fd_local2global` 将进程的本地文件描述符映射到全局文件表中的下标。随后调用 `file_close` 释放文件占用的 `inode` 资源，并将其从文件表中移除，同时重置 `write_deny` 标志以允许其他进程对文件进行写操作。最后，将进程本地文件描述符表中的对应项置为-1，表示该文件描述符位置已释放，整个关闭操作完成后返回执行结果。

8.3 文件写入

整体思路与流程

为了实现支持文件描述符的 `sys_write` 功能，本次的工作主要包括两个部分：首先在文件层次实现核心的 `file_write` 函数，用于具体完成文件数据的写入；然后

在系统调用层实现 `sys_write`，对用户接口进行适配和功能扩展。此外，还需要对周边函数（如 `write` 系统调用、`printf` 等）进行相应改动，以适应新功能的实现。



```
Bochs x86 emulator, http://bochs.sourceforge.net/
SECTORS: 163296
CAPACITY: 79MB
[HFUT_XU @localhost ~]$
all partition info
sdb1 start_lba:0x800,sec_cnt:0x3F00
sdb5 start_lba:0x5000,sec_cnt:0x2370
sdb6 start_lba:0x8000,sec_cnt:0x3138
sdb7 start_lba:0xC000,sec_cnt:0x5171
sdb8 start_lba:0x12000,sec_cnt:0xDBD1
sdb9 start_lba:0x20800,sec_cnt:0x75E0
ide_init done
searching filesystem.....
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
/file1 has already exist!
(HFUT_xht) Failed to create or open /file1
(HFUT_xht) File /file1 opened in read-write mode with fd: 3
(HFUT_xht) Successfully wrote 9 bytes to /file1
(HFUT_xht) File /file1 (fd: 3) closed successfully
```

文件的写入测试

8.4 文件写入核心功能 `file_write`

`file_write` 是文件写入的核心功能，负责将缓冲区的数据写入文件。首先，检查当前文件大小加待写数据是否超过系统限制（140 块，每块 512 字节），若超限则返回错误。接着，申请缓冲区 `io_buf` 进行硬盘操作，并为记录所有块地址申请 `all_blocks`。如果是首次写入，分配第一个数据块并记录到文件的 `i_sectors` 数组中，更新位图并同步到硬盘。接下来，根据需要写入的数据大小，判断是否需要新增块。若新增块，依据情况分配直接块、一级间接块或完全通过间接块进行存储，并将所有涉及的块地址存储到 `all_blocks` 中。数据写入时，按照计算出的块地址将数据分块写入硬盘，首次写入时先读取现有数据并追加。最后，更新文件的大小 `i_size` 和写入位置 `fd_pos`，同步文件的 `inode` 到硬盘，释放缓冲区并返回成功写入的字节数。

8.5 读取文件

整体思路

文件读取功能的实现包括两部分：核心文件读取功能 `file_read` 和系统调用封装 `sys_read`。文件读取与文件写入的逻辑类似，核心任务是根据文件描述符定位文件，并从文件中读取指定数量的数据到缓冲区。同时，根据文件的存储块分配结构，处理直接块和间接块的读取情况。系统调用 `sys_read` 则作为用户接口，简化用户操作，提供高层次的文件读取支持。至于 `sys_read` 暂时不细说，因为按照后面功能的完善，这个部分还会有不少改动。

8.6 现文件读写指针定位

在上面一个板块中，我们读完了文件之后，如果想要重头再开始读取文件，只能关闭文件在打开文件，这就很有些费时间，因此我们需要实现读写指针的定位，其实也就是确定文件读写时的起始偏移量而已。

实现思路

实现 `sys_lseek` 时，首先根据文件描述符 `fd` 找到对应的文件结构，并从中获取文件当前的读写位置 `fd_pos` 和文件大小 `i_size`。然后根据 `whence` 的不同，计算新的文件指针位置 `new_pos`：若 `whence` 为 `SEEK_SET`，则直接将 `offset` 作为新的 `fd_pos`；若为 `SEEK_CUR`，则将当前 `fd_pos` 加上 `offset`；若为 `SEEK_END`，则将文件大小加上 `offset` 作为新的 `fd_pos`。计算完成后，检查 `new_pos` 是否在合法范围内（0 到文件大小减 1 之间）。若超出范围，返回 -1 表示失败；若合法，则更新文件的 `fd_pos` 为 `new_pos`，并返回新的位置值。通过这一逻辑，`sys_lseek` 实现了文件指针位置的灵活调整，提供了对文件读写操作的精确控制。

8.7 文件的删除功能：

删除文件主要是对应两个工作，一个是回收文件对应的结点，第二个就是删除文件对应的目录

8.7.1 回收结点，主要包括下面这些工作：

- **位图释放：**回收 inode 和数据块对应的位图位。
- **块释放：**处理直接块、间接块，以及一级间接索引表块。
- **同步操作：**确保内存与磁盘数据的一致性。
- **关闭和清空：**释放与 inode 相关的内存资源，并清空表项（可选）。

实现思路

回收 inode 资源的关键是准确释放其关联的硬盘资源。在 `inode_delete` 中，首先通过 `inode_locate` 获取 inode 在硬盘中的位置，清零其数据后写回硬盘。`inode_release` 则负责回收 inode 关联的所有数据块资源，首先读取 inode 的直接块和一级间接块地址存入 `all_blocks` 数组，如果存在一级间接块表，还需回收其中记录的间接块及表本身。随后遍历 `all_blocks` 数组，对每个有效块在数据块位图中清零并同步到硬盘。最后，清除 inode 位图的使用标记并同步到硬盘，必要时调用 `inode_delete` 擦除 inode 表中的数据，确保其所有资源被彻底释放。

8.7.2 删除文件目录：

主要涉及以下这些工作：

1. **目录项删除：**从父目录的所有块中查找指定 inode 编号的目录项，并将其清空。
2. **块回收：**如果该目录项独占一个数据块，并且该块不是根目录的第一个块，则回收该块。
3. **目录 inode 更新：**目录的 inode 大小会减去一个目录项的大小，同时将更新同步到硬盘。
4. **硬盘同步：**将所有修改（如目录数据、块位图、inode）同步到硬盘，确保文件系统状态一致。

实现思路：

实现删除目录项时，首先遍历父目录 `inode` 的所有块，通过 `i_sectors` 收集直接块和间接块地址，并加载这些块的数据以搜索目录项。在每个块中，通过缓冲区 `io_buf` 查找与指定 `inode` 编号匹配的目录项。找到后，根据块中剩余目录项数量决定操作：若该块仅包含待删除的目录项且不是根目录的第一个块，则回收该块，在块位图中标记为未使用并同步更新，同时移除对应的块地址；若块中还有其他有效目录项，则仅清空当前目录项并将修改同步到硬盘。此外，确保根目录的第一个块不会被回收。删除完成后，更新目录 `inode` 的大小，减少一个目录项的大小，并同步到硬盘。如果未找到指定的目录项，则返回 `false` 表示删除失败。

9. 目录的操作

9.1 创建目录

创建目录要做的工作如下：

- (1) 确认待创建的新目录在文件系统上不存在。
- (2) 为新目录创建 `inode`。
- (3) 为新目录分配 1 个块存储该目录中的目录项。
- (4) 在新目录中创建两个目录项 “.” 和 “..”，这是每个目录都必须存在的两个目录项。
- (5) 在新目录的父目录中添加新目录的目录项。
- (6) 将以上资源的变更同步到硬盘。

我们将在 `mkdir` 的内核部分—`sys_mkdir` 中实现以上功能，

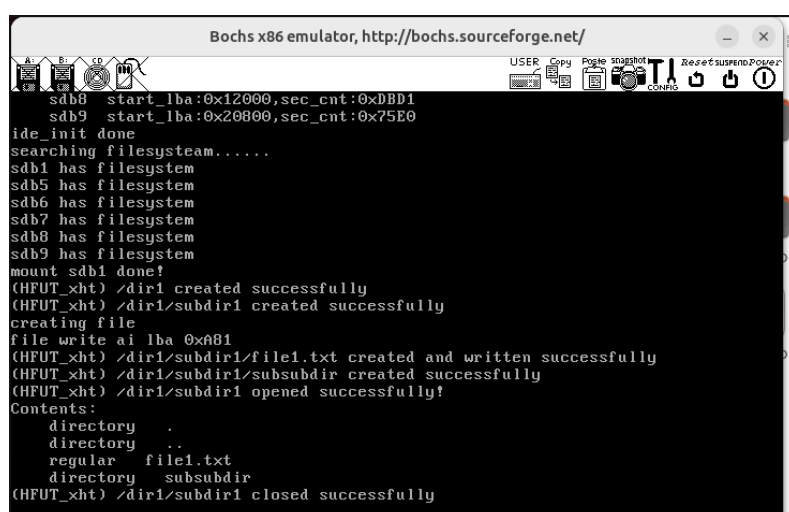
实现思路（`sys_mkdir`）

`sys_mkdir` 的实现首先通过调用 `search_file` 检查目标路径是否已存在，如果存在同名文件或目录，则直接返回错误。接着判断路径的中间目录是否存在，如果中间路径不存在，也直接返回错误。接下来为新目录分配一个 `inode`，并初始化 `inode` 的基本信息，然后从块位图中分配一个数据块，将新目录的 `.` 和 `..` 目录项写入该块，并将块地址存入新目录 `inode` 的 `i_sectors[0]` 中，同时将块位图的更改同步到硬盘。随后在父目录中添加新目录的目录项，更新父目录

inode 的大小，并将父目录 inode 的更改同步到硬盘。最后，将新目录 inode 的更改和 inode 位图的更新同步到硬盘，并关闭父目录释放相关资源。如果在任意步骤中失败，程序会进入回滚流程，根据记录的 `rollback_step` 回退到之前的状态，例如释放已分配的 inode 或块位图，关闭父目录等，确保文件系统的一致性。

9.2 遍历目录：

遍历目录的功能是读取目录中所有的目录项，包括文件和子目录的信息，支持顺序访问、多次遍历和从头重新读取的能力。在实现该功能前，需要通过 `sys_opendir` 和 `sys_closedir` 实现目录的打开与关闭。`sys_opendir` 验证路径是否合法且为目录，返回目录指针；`sys_closedir` 释放目录相关的资源，确保系统资源高效管理。目录项读取由 `sys_readdir` 完成，基于 `dir_read` 遍历 inode 管理的数据块，跳过空洞返回有效的目录项，同时更新游标 `dir_pos`，避免重复读取。如果目录为空或读取完毕，则返回 `NULL`。通过 `sys_rewinddir`，游标 `dir_pos` 可重置为 0，实现从头重新读取目录，无需关闭并重新打开，提高遍历效率和灵活性。



```
Bochs x86 emulator, http://bochs.sourceforge.net/
sdb0 start_lba:0x12000,sec_cnt:0xDBD1
sdb9 start_lba:0x20800,sec_cnt:0x75E0
ide_init done
searching filesystem.....
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
(HFUT_xht) /dir1 created successfully
(HFUT_xht) /dir1/subdir1 created successfully
creating file
file write ai lba 0xA01
(HFUT_xht) /dir1/subdir1/file1.txt created and written successfully
(HFUT_xht) /dir1/subdir1/subsubdir created successfully
(HFUT_xht) /dir1/subdir1 opened successfully!
Contents:
  directory  .
  directory  ..
  regular    file1.txt
  directory  subsubdir
(HFUT_xht) /dir1/subdir1 closed successfully
```

遍历目录测试

9.3 删除目录：

成功移除指定的空目录，并释放该目录在文件系统中占用的所有资源。如果目标目录中有文件或子目录，则删除操作会失败，并提示“目录非空”。在删除成功时，目录项会从父目录中移除，目录对应的 inode 和数据块资源也会被回收，同时保证文件系统结构的完整性和一致性。

目录删除测试

实现思路

删除目录的核心是确保只有空目录才能被删除。通过 `dir_is_empty` 判断目录是否为空，即检查目录是否仅包含基础目录项“.”和“..”。若目录为空，则调用 `dir_remove` 进行删除，先通过 `delete_dir_entry` 从父目录中删除对应目录项，再调用 `inode_release` 回收子目录的 inode 和资源。在 `sys_rmdir` 系统调用中，通过 `search_file` 检查路径是否存在并验证其是否为目录文件。如果路径不存在、目标是普通文件或目录非空，则直接返回错误；若路径是空目录，则调用 `dir_remove` 执行删除，并释放相关资源，最后返回结果。整个过程通过严格的判断和资源管理，确保操作的安全性和一致性。

9.4 任务的工作目录

在 Linux 中咱们经常会使用命令 `pwd` 来显示当前工作目录，还要用 `cd` 命令来改变工作目录，在我们的内核中也要实现这样的功能。

功能说明

显示当前工作目录（sys_getcwd, pwd）： 返回当前任务的工作目录的绝对路径，可以写入用户提供的缓冲区，或者动态分配内存后返回路径。

更改当前工作目录（sys_chdir, chdir）： 将当前任务的工作目录切换为用户指定的路径，若路径有效且为目录则切换成功，否则返回失败。

获得文件属性（sys_stat, ls）： 获取文件或目录的基本属性，包括 inode 编号、文件大小和文件类型，并将其填充到提供的缓冲区中。

改变当前工作目录（sys_chdir, getcwd）： 更改当前任务的工作目录，修改进程控制块中工作目录对应的 inode 编号，成功返回 0，失败返回 -1。

文件属性获取测试

Shell 功能

Shell 功能与实现思路

实现的功能

命令输入与快捷键支持（readline）

Shell 实现了用户输入命令的功能，允许用户通过键盘输入并实时显示输入内容。支持键盘快捷键，如 Ctrl+U 用于清除当前输入，Ctrl+L 用于清屏但保留当前输入，增强了用户的操作体验。

命令解析 (cmd_parse)

实现了命令解析功能，能够将用户输入的字符串按照空格等分隔符分割成命令和参数。解析结果存储在参数数组 argv 中，并返回参数数量 argc，为后续的命令执行提供支持。

路径解析与转换 (make_clear_abs_path 和 wash_path)

实现了路径解析功能，将用户输入的路径（可能包含相对路径、. 和 ..）转换为规范的绝对路径。支持相对路径基于当前工作目录转换为绝对路径，并去除多余符号，确保路径的合法性和正确性。

内建命令实现 (buildin_ls、buildin_cd、buildin_mkdir、buildin_rmdir、buildin_rm、buildin_pwd、buildin_ps、buildin_clear)

实现了一系列内建命令，包括 ls、cd、mkdir、rmdir、rm、pwd、ps 和 clear。这些命令通过内建函数直接执行，无需加载外部程序，完成文件系统操作、目录管理、进程信息显示等功能。

用户交互与提示符 (print_prompt)

Shell 提供了直观的用户交互界面，通过命令提示符显示当前路径、主机名和用户名（如 [HFUT_XU@localhost %s]\$，引导用户输入命令。支持实时反馈用户的操作结果，如命令输出或错误提示。

实现的思路

命令输入与快捷键

通过 readline 函数逐字符读取用户输入，实时将输入内容显示在屏幕上。对快

捷键的处理由上层 Shell 实现，Ctrl+U 的实现通过输出退格符清除输入，Ctrl+L 的实现通过调用清屏函数 `clear` 并重新输出提示符和输入内容。

命令解析

使用 `cmd_parse` 函数对用户输入的命令字符串进行解析。通过遍历输入字符串，逐个提取单词（命令和参数），将每个单词的指针存储到参数数组 `argv` 中。支持跳过多余空格，并处理参数数量限制，确保解析结果可靠且不溢出。

路径解析与转换

其实现通过 `make_clear_abs_path` 和 `wash_path` 函数完成。首先，`make_clear_abs_path` 会检查路径类型：如果是相对路径，就通过调用 `getcwd` 获取当前工作目录并拼接用户输入，生成绝对路径；如果是绝对路径，则直接处理。随后调用 `wash_path` 函数对路径进行清洗，该函数逐层解析路径，将路径中的 `.` 忽略不处理，将 `..` 回退到上一级目录（通过删除当前路径的最后一层实现），并将有效的路径层级逐一拼接到最终路径中，确保输出的是合法的绝对路径。整个解析过程保证了用户输入路径的正确性和简洁性，为后续命令的执行提供了稳定可靠的路径支持。

内建命令实现

内建命令以 `buildin_` 前缀命名，如 `buildin_ls` 和 `buildin_cd`。每个命令接收参数 `argc` 和 `argv`，调用对应的系统调用完成实际功能。在执行命令前，路径参数会通过 `make_clear_abs_path` 转换为绝对路径，确保路径输入的正确性。部分命令（如 `cd`）会更新当前工作目录缓存，用于刷新提示符。

用户交互与提示符

Shell 主程序通过循环实现交互，每次显示提示符，等待用户输入命令，解析命令并执行相应功能。提示符的实现依赖当前工作目录缓存 `cwd_cache`，在 `cd` 命令成功后更新缓存，确保提示符始终反映用户的当前目录状态。