Lab14 - Ext2 filesystem

0. 前言

实验概述

学习 Ext2 文件系统的结构、文件存储形式与相关代码；了解Ext2到ext4的演进，以及 rootfs、Initramfs 及其挂载机制。

实验代码

拉取实验代码：

```
$ git clone https://github.com/sdww0/sustech-os-lab.git && cd sustech-os-lab/lab14-fs
```

或更新实验代码：

```
# In sustech-os-lab
$ git pull && cd lab14-fs
```

本次实验新增/修改代码包括：

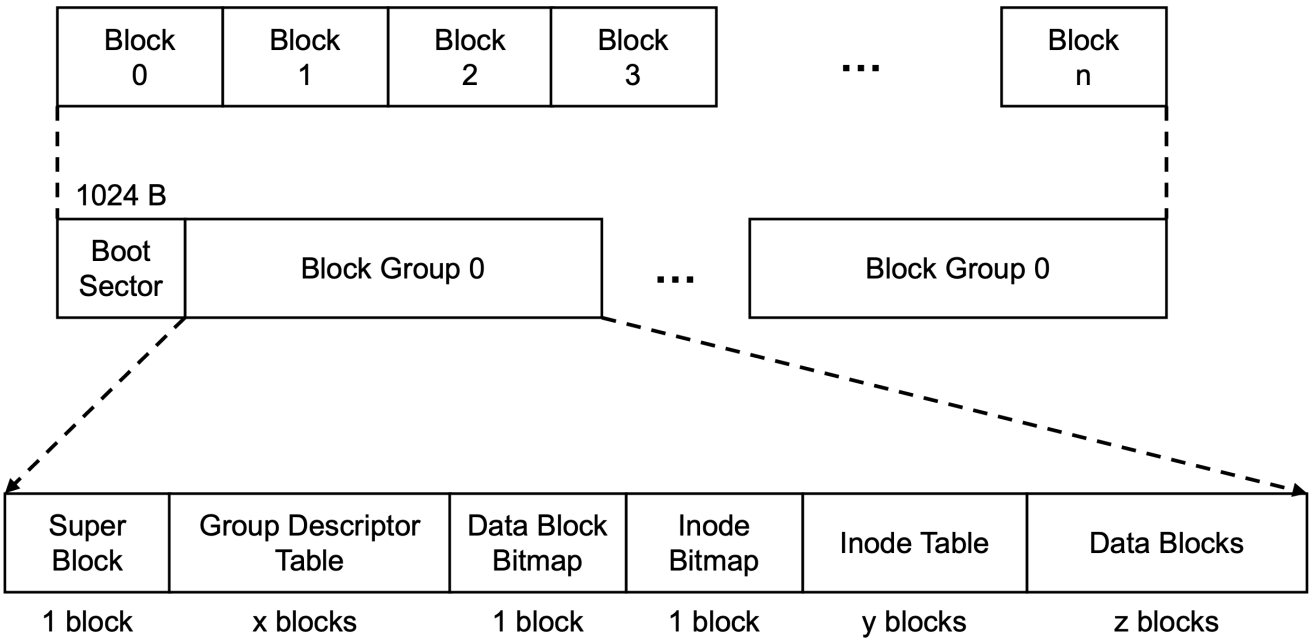
1. Ext2 文件系统实现
2. Ext2 文件系统格式化与挂载
3. 重构块设备接口，以适配 Ext2 的需求

1. Ext2文件系统

1.1 Ext2概览

Ext2（第二代扩展文件系统）是Linux内核于1993年引入的一款经典磁盘文件系统。它采用“块组”结构来组织磁盘空间，即将整个分区划分为多个连续的块组，每个块组内部都包含自己独立的超级块、Inode表、位图和数据块区域。作为Ext系列演进的基石，Ext2为其后续版本奠定了核心架构：Ext3在其基础上主要增加了日志功能以提升崩溃恢复能力；而现代广泛使用的Ext4则进一步扩展，支持更大的存储容量、更优的性能特性（如区段扩展、多块分配）以及更丰富的功能。

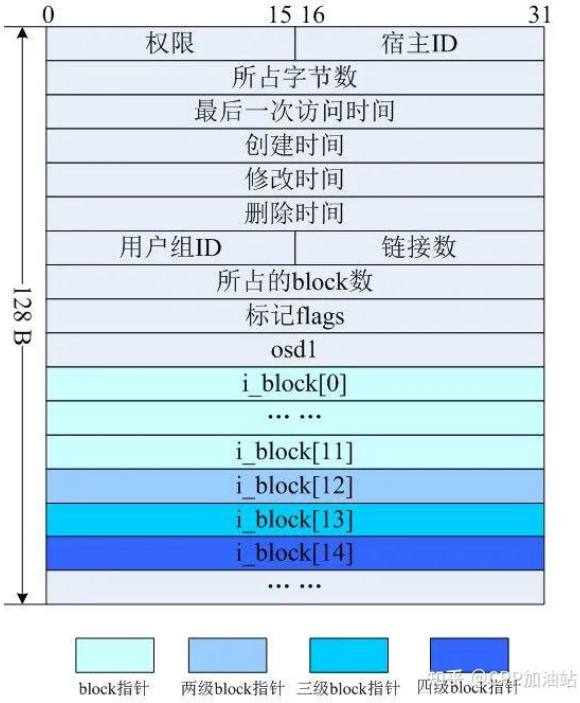
Ext2文件系统以“块”作为数据存储与分配的最小单元，块大小可在创建文件系统时配置，通常为1KB、2KB或4KB。一个完整的Ext2文件系统在磁盘上的物理布局如下图所示：



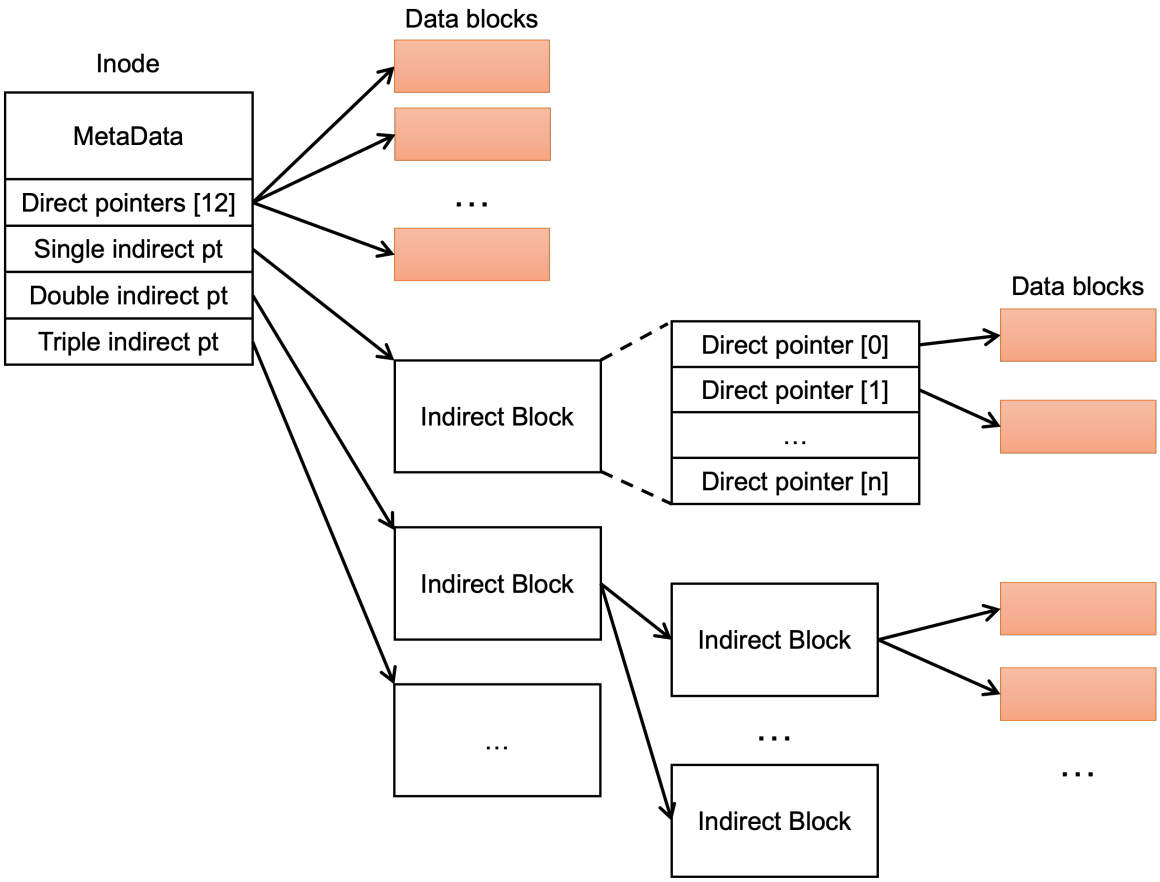
如图所示，Ext2文件系统起始部分是一个启动块，占用1024字节，随后是连续排列的多个块组。每个块组包含以下部分：

- **超级块**：存储整个文件系统的全局元数据，如块大小、块组总数、空闲块/Inode计数等。第一个块组（块组0）的超级块位于固定偏移1024字节处，其余块组中的超级块是其备份，用于灾难恢复。
- **块组描述符表**：一个由多个块组描述符组成的数组，每个描述符记录了对应块组的关键信息，例如块位图和Inode位图的起始块号、Inode表的起始块号等。
- **块位图**：一个比特位图，用于跟踪本块组内所有数据块的空闲与已使用状态（1个比特代表1个数据块）。
- **Inode位图**：一个比特位图，用于管理本块组内所有Inode的空闲状态。
- **Inode表**：一个连续存储的Inode数组，每个Inode的大小固定为128字节，用于存储文件或目录的元数据。
- **数据块**：实际存储文件内容或目录项列表的磁盘空间。

其中的核心概念**Inode**（索引节点）是Ext2管理文件实体的关键数据结构。每个Inode固定为128字节，包含了文件/目录的所有元信息，例如类型、权限、所有者、大小、时间戳（创建、修改、访问时间）以及指向其数据块的指针数组。其结构示意图如下（图源：知乎）：

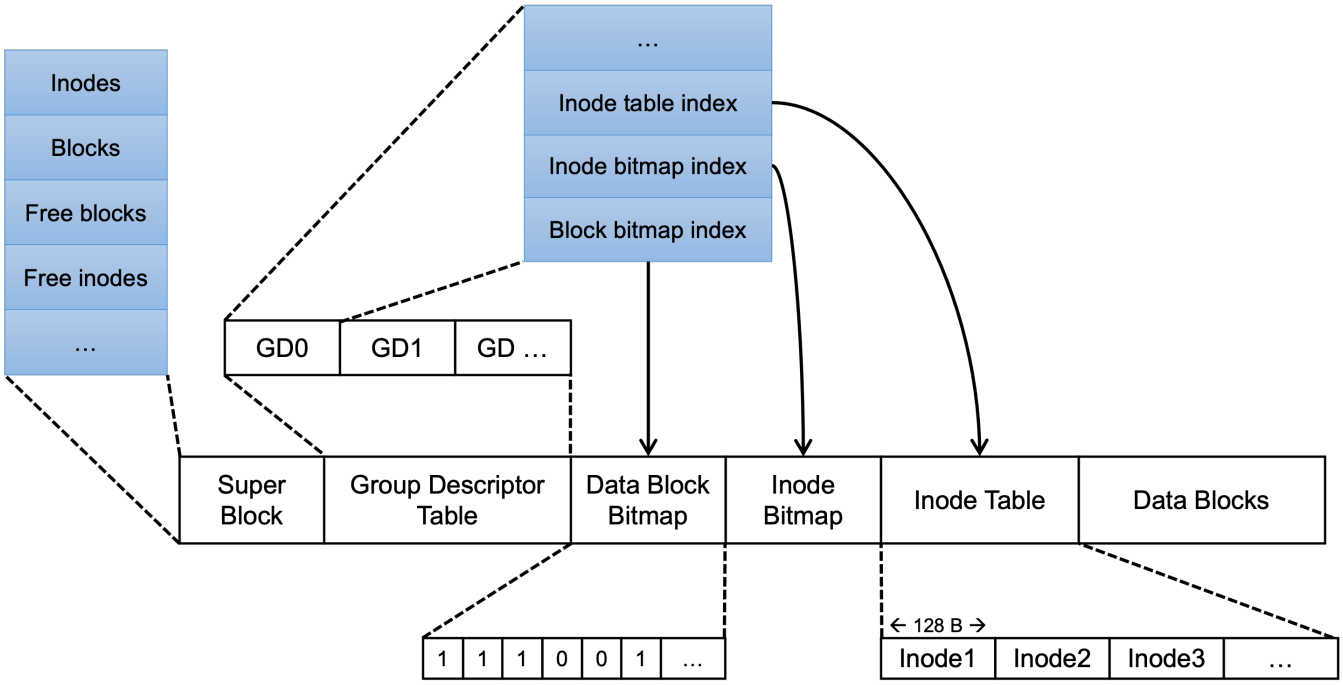


图中最后的 `i_block[xx]` 数组即数据块指针。**前12个为直接指针**，直接指向存储文件数据的数据块；**第13个为一级间接指针**，指向一个包含更多直接指针的块；**第14个为二级间接指针**；**第15个为三级间接指针**。其多级索引原理如下图所示：



1.2 Ext2初始化

操作系统挂载一个Ext2文件系统时，会执行一系列初始化操作以建立其内存中的管理结构，Ext2中块组初始化的相关结构如下：



对应的过程如下（函数为`fs::ext2::Ext2Fs::new()`）：

1. **读取超级块**：操作系统首先从磁盘的固定偏移（第一个块组的1024字节处）读取**超级块**。通过验证其中的魔法值（`0xEF53`）来确认这是一个有效的Ext2文件系统。
2. **加载块组描述符表**：根据超级块中的信息，定位并读取整个**块组描述符表**到内存。该表提供了所有块组关键元数据，如位图和Inode表位置。
3. **建立缓存**：将超级块与块组描述符缓存进内存中，并根据块组描述符，索引到块位图、Inode位图与Inode表。

块组描述符与超级块的定义如下（节选）：

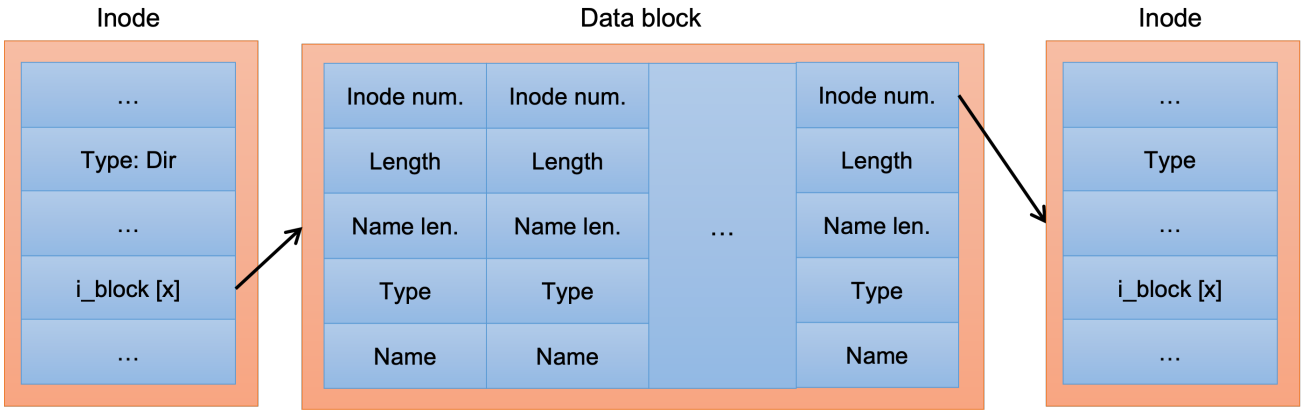
```
#[repr(C)]
#[derive(Clone, Copy, Debug, Pod)]
pub(super) struct RawGroupDescriptor {
    block_bitmap: u32,
    inode_bitmap: u32,
    inode_table: u32,
    free_blocks_count: u16,
    free_inodes_count: u16,
    dirs_count: u16,
    pad: u16,
    reserved: [u32; 3],
}

#[repr(C)]
#[derive(Clone, Copy, Debug, Pod, Default)]
pub struct RawSuperBlock {
    pub inodes_count: u32,
    pub blocks_count: u32,
    pub reserved_blocks_count: u32,
    pub free_blocks_count: u32,
    pub free_inodes_count: u32,
```

```
    pub first_data_block: u32,
    pub log_block_size: u32,
    pub log_frag_size: u32,
    pub blocks_per_group: u32,
    pub frags_per_group: u32,
    pub inodes_per_group: u32,
    pub mtime: u32,
    pub wtime: u32,
    pub mnt_count: u16,
    pub max_mnt_count: u16,
    pub magic: u16,
    ...
}
```

1.3 Directory的存储与索引

在Ext2中，**目录（Directory）是一种特殊类型的文件**，其Inode类型标记为`S_IFDIR`。目录的结构与索引如图所示：



目录的数据块中存储了一个线性表，包含了若干目录项，目录项的定义如下：

```
#[repr(C)]
#[derive(Debug, Clone, Copy, Pod)]
pub struct Ext2DirEntry {
    // Inode number
    ino: u32,
    // Length
    record_len: u16,
    // Name length
    name_len: u8,
    // Inode type
    type_: u8,
    // The name buffer
    name: [u8; MAX_NAME_LEN],
}
```

当用户希望查看一个文件（如`/home/user/file.txt`）是否存在时，操作系统通过以下步骤索引到该Inode：

1. **路径解析**：从根目录（默认为Inode 2）开始。读取根目录的Inode，根据其`i_block[]`指针找到存储根目录项的数据块。

2. **目录项遍历**：在根目录的数据块中，线性扫描目录项，查找名为`"home"`的条目，并获得其Inode号。
3. **递归查找**：读取`"home"`目录的Inode，进而读取其数据块，查找名为`"user"`的目录项，获得其Inode号。重复此过程，最终在`"user"`目录的数据块中找到名为`"file.txt"`的目录项，并获取目标文件的**Inode号**。

1.4 Ext2的读取与写入操作

Ext2 的文件读取操作较为简单：根据目录索引到目标文件后，利用 `i_block[xx]` 即可读取文件数据。

Ext2 的文件写入则稍复杂：若**不需要扩展文件大小**，只需覆盖目标数据块的内容，并更新单个 Inode 的元数据；若**需要扩展文件大小**，则需额外执行从块位图中查找并分配空闲数据块，以及更新全局元数据（如超级块等）的操作。

1.5 Ext2到Ext4的演进

Ext2 文件系统作为 Linux 早期广泛使用的格式，虽然稳定可靠，但在性能、可靠性和功能上逐渐显现出局限性。为适应不断增长的需求，Ext2 经历了两次重要升级：Ext3 与 Ext4。

Ext2 --> Ext3

Ext3 在 Ext2 基础上最核心的改进是引入了**日志（Journaling）功能**。在 Ext2 中，若系统在写入文件元数据（如 Inode、目录项）或实际数据时突然断电或崩溃，文件系统可能因部分更新未完成而处于不一致状态。此时必须通过耗时的全盘检查（`fsck`）来扫描并修复错误。

Ext3 的日志机制解决了这一问题。其原理是：在实际执行磁盘写入操作之前，先将这些操作的**“意图”**记录到一个专用的日志区域。这份日志如同一份操作清单，若所有操作顺利完成，相应的日志条目会被标记为完成并清理。若发生崩溃，则在下一次挂载时，系统只需读取并分析日志，然后根据记录重放（Redo）或撤销（Undo）相关操作，即可快速、安全地将文件系统恢复到崩溃前的一致状态，从而避免了全盘扫描的巨大开销。

Ext3 提供三种日志模式，以在安全性与性能之间进行权衡：

- **日志模式**：记录所有元数据与文件数据，安全性最高，但性能开销最大。
- **顺序模式**：仅记录元数据，并确保数据块先于元数据写入，平衡安全与效率。
- **回写模式**：仅记录元数据，且不严格保证写入顺序，性能最优

Ext3 --> Ext4

Ext4 是对 Ext3 的一次全面升级，旨在突破其在超大容量与高并发场景下的性能瓶颈。

Ext4 的一项核心改进是引入**区段（Extents）**，用以取代 Ext2/3 中传统的块映射方式。在 Ext2/3 中，大文件可能需要通过大量间接指针块来映射成千上万个数据块，导致元数据庞大且容易产生碎片。区段则将文件中连续的数据块描述为一个范围（例如“起始块号 1000，连续 200 个块”，其概念类似于内存管理中的“起始地址+长度”），仅需一个区段结构即可表示，从而大幅减少了元数据量。

此外，Ext4 还实现了多项分配策略：

- **多块分配**：允许一次性分配多个连续的数据块，减少频繁分配操作的开销与碎片产生。
- **延迟分配**：将数据块的实际分配决策推迟到数据即将写回磁盘的时刻，从而积累更多的写入请求，实现更优的连续性分配。

2. Initramfs, Initrd与rootfs

2.1 Initramfs与Initrd

回顾操作系统的启动流程，内核加载到内存并开始运行后，需要启动第一个用户空间的初始化进程。然而，该进程及其依赖的程序和库通常存储在磁盘上，而访问磁盘又需要加载对应的驱动程序。这就形成了一个悖论：内核完成自身初始化后，缺乏必要的驱动而无法直接从磁盘加载第一个用户进程。**Initramfs** 正是为解决此问题而设计的中间媒介。

在介绍 Initramfs 之前，需要先了解其前身：**Initrd**。早期内核使用 initrd 来存放启动所必需的核心文件。Initrd 基于 **ramdisk** 机制，它将一段固定大小的内存模拟为一个完整的块设备。这意味着：

- 它像物理硬盘一样有固定的容量限制。
- 需要先使用 `mke2fs` 等格式化工具将其格式化为某个磁盘文件系统（如 Ext2）。
- 由于被内核视为块设备，访问其中的文件仍需经过页缓存、缓冲区等额外的拷贝和管理开销，尽管数据本身就在内存中。

为了克服 initrd 的这些限制，Linux 随后实现了 **tmpfs/ramfs**。这是一种直接在内存中实现的文件系统，无需模拟块设备，因此没有固定大小的限制（可动态增长），并且避免了不必要的缓存复制开销。基于 tmpfs/ramfs，开发人员设计了 **Initramfs** 来替代 Initrd。两者的目标一致——作为临时的根文件系统，存放内核启动所必需的中间文件。关键区别在于：Initramfs 直接使用 tmpfs/ramfs 机制加载，而非模拟的块设备，因而更加高效和灵活。

你可以通过`ls -lha /boot`来找到当前Linux系统所使用的Initramfs，以下是示例输出：

```
$ ls -lha /boot
-rw-r--r--  1 root root 281K  9月  9 20:53 config-6.8.0-84-generic
-rw-r--r--  1 root root 281K 11月 20 21:46 config-6.8.0-90-generic
lrwxrwxrwx  1 root root   27 12月 16 06:24 initrd.img -> initrd.img-6.8.0-90-generic
-rw-r--r--  1 root root 177M 11月 13 06:35 initrd.img-6.8.0-84-generic
-rw-r--r--  1 root root 179M 12月 16 06:25 initrd.img-6.8.0-90-generic
lrwxrwxrwx  1 root root   27 12月 16 06:25 initrd.img.old -> initrd.img-6.8.0-84-generic
-rw-------  1 root root 8.4M  9月  9 20:53 System.map-6.8.0-84-generic
-rw-------  1 root root 8.4M 11月 20 21:46 System.map-6.8.0-90-generic
lrwxrwxrwx  1 root root   24 12月 16 06:24 vmlinuz -> vmlinuz-6.8.0-90-generic
-rw-------  1 root root  15M  9月  9 20:54 vmlinuz-6.8.0-84-generic
-rw-------  1 root root  15M 11月 20 21:52 vmlinuz-6.8.0-90-generic
lrwxrwxrwx  1 root root   24 12月 16 06:25 vmlinuz.old -> vmlinuz-6.8.0-84-generic
```

`initrd.img-*`便是安装好的Initramfs，其中`initrd.img`会指向当前系统启动所使用的Initramfs。

在我们的内核设计中，必要的应用程序会直接编译并**链接到内核二进制镜像内部**。这种方式与 Initramfs 有相似的目标，即在内核启动阶段为其提供可执行的初始程序包，但是直接编译连接是一种紧耦合的方式。

2.2 Rootfs与根目录

在挂载真正的磁盘根文件系统之前，内核会首先挂载一个称为 **rootfs** 的基础文件系统。rootfs 本质上就是基于 tmpfs/ramfs 实现的。挂载 rootfs 后，内核会将 Initramfs 镜像（通常是一个 cpio 文件）解压并加载到其中。此时，系统就获得了一个包含基础目录结构、工具和驱动模块的临时根环境。

在这个临时环境中，系统可以执行 Initramfs 中的初始化脚本（通常是 `/init`），来完成诸如加载磁盘控制器驱动程序、解密加密分区、识别逻辑卷（LVM）等操作。当根文件系统（例如 `/dev/sda1` 上的 ext4 文件系统）准备就绪，系统便会将其挂载并切换为新的根目录，覆盖掉临时的 rootfs。

至此，rootfs 的使命完成。虽然 rootfs 本身作为内核的一个实例不能被卸载，但其内部由 Initramfs 解压出来的所有文件和目录所占用的内存可以被回收释放。

你可以通过 `df -hT` 命令查看当前系统中各个文件系统的挂载情况，以下是一个示例：

```
$ df -ha
# Filesystem                                    Size  Used Avail Use% Mounted on
# sysfs                                            0     0     0    - /sys
# proc                                             0     0     0    - /proc
# udev                                            16G     0   16G   0% /dev
# devpts                                           0     0     0    - /dev/pts
# tmpfs                                          3.2G  2.1M  3.1G   1% /run
# rpool/ROOT/ubuntu_qrb5ih                       285G   23G  263G   8% /
# ...
```

输出中，挂载点为 `/` 的行即系统当前使用的真实根文件系统。本例中，它是一个 ZFS 文件系统（类型为 `zfs`），其存储池名称为 `rpool`。

若想进一步了解这个根文件系统具体位于哪个物理块设备，可以结合 `lsblk -f` 命令查看：

```
$ lsblk -f
# NAME        FSTYPE     FSVER LABEL UUID                                 FSAVAIL FSUSE%
MOUNTPOINTS
# ...
# nvme0n1
# ├─nvme0n1p1 vfat       FAT32       D811-A895                             496.6M     3%
/boot/grub
# │
/boot/efi
# ├─nvme0n1p2 swap       1           f4c5d00c-0eef-46a7-b598-1bbe39bc1480
[SWAP]
# ├─nvme0n1p3 zfs_member 5000  bpool 106102806978449528
# └─nvme0n1p4 zfs_member 5000  rpool 2795207562415207895
```

可以看到，设备 `nvme0n1p4` 是 `rpool` ZFS 存储池的一部分，这正对应着 `df` 命令中挂载到根目录 `/` 的文件系统。

3. 上手练习

请你修改代码，将rootfs从Ramfs改为Ext2，模拟将文件系统挂载覆盖rootfs这一流程，并在Makefile中在host挂载写入Ext2时新增一个hello.txt文件，并往里面写入你指定的内容，然后在内核中运行`read_ext2_file`应用程序。结果示例（`Hello, TEXT!`）：

```
Virtio MMIO device found at 0x10008000 with size 0x1000, device id 2, version: 1
Virtio MMIO device found at 0x10007000 with size 0x1000, device id 2, version: 1
Read from ext2: Hello, Ext2!
[INIT] Starting Shell...
Running Shell...
~ # read_ext2_file
 Running command: read_ext2_file
Content of hello: Hello, TEXT!
```