中山大學

SUN YAT-SEN UNIVERSITY

操作系统实验报告

实验六：硬盘驱动与文件系统

姓　　名：　刘家祥

学　　号：　23336152

教学班号：　计科二班

专　　业：　计算机科学与技术

院　　系：　计算机学院

2024~2025 学年第二学期

硬盘驱动与文件系统

一. MBR 分区表解析实现

1. 在 `MbrPartition` 结构体中使用给出的 `commens/macros.rs` 中的 `define_field!` 宏定义简单分区表的简单字段。

其中包括 `status` 分区状态标志、`begin_head` 分区起始扇区的CHS地址、`PartitionType` 分区类型、`end_head` 分区结束扇区的CHS地址、`begin_lba` 分区起始扇区的LBA、`total_lba` 分区大小。

```
// Define fields using define_field! macro
    define_field!(u8, 0x00, status);
    define_field!(u8, 0x01, begin_head);
    define_field!(u8, 0x04, partition_type);
    define_field!(u8, 0x05, end_head);
    define_field!(u32, 0x08, begin_lba);
    define_field!(u32, 0x0C, total_lba);
```

2. 手动实现了 CHS 地址解析函数:

1. `begin_sector()` - 从 `0x02` 字节解析起始扇区号（`6` 位）
2. `begin_cylinder()` - 从 `0x02-0x03` 字节解析起始柱面号（`10` 位）
3. `end_sector()` - 从 `0x06` 字节解析结束扇区号（`6` 位）
4. `end_cylinder()` - 从 `0x06-0x07` 字节解析结束柱面号（`10` 位）

```
// CHS address parsing functions for begin sector & cylinder (0x02-0x03)
    pub fn begin_sector(&self) -> u16 {
        // Sector number is in bits 0-5 of byte 0x02
        (self.data[0x02] & 0x3F) as u16
    }
```

```
    pub fn begin_cylinder(&self) -> u16 {
        // Cylinder number is in bits 6-7 of byte 0x02 (high 2 bits) + byte 0x03
(low 8 bits)
        let high_bits = ((self.data[0x02] & 0xC0) as u16) << 2;
        let low_bits = self.data[0x03] as u16;
        high_bits | low_bits
    }

    // CHS address parsing functions for end sector & cylinder (0x06-0x07)
    pub fn end_sector(&self) -> u16 {
        // Sector number is in bits 0-5 of byte 0x06
        (self.data[0x06] & 0x3F) as u16
    }

    pub fn end_cylinder(&self) -> u16 {
        // Cylinder number is in bits 6-7 of byte 0x06 (high 2 bits) + byte 0x07
(low 8 bits)
        let high_bits = ((self.data[0x06] & 0xC0) as u16) << 2;
        let low_bits = self.data[0x07] as u16;
        high_bits | low_bits
    }
```

3. 在 `partition/mbr/mod.rs` 的 `parse` 函数中，根据 `MBR` 的结构定义，按照对应的偏移量，提取四个 `MbrPartition` 并进行存储。（此处仅展示函数内新增代码）

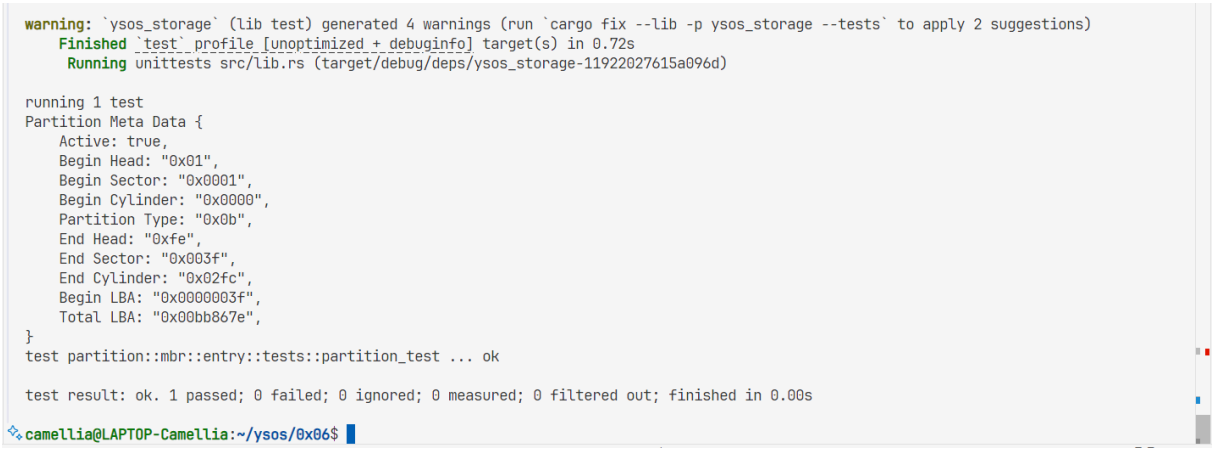
```
fn parse(inner: T) -> FsResult<Self> {
        for i in 0..4 {
            let partition_offset = 0x1BE + i * 16;
            let partition_data: [u8; 16] =
buffer[partition_offset..partition_offset + 16]
                .try_into()
                .expect("Failed to extract partition data");
            partitions.push(MbrPartition::parse(&partition_data));
        }
    }
```

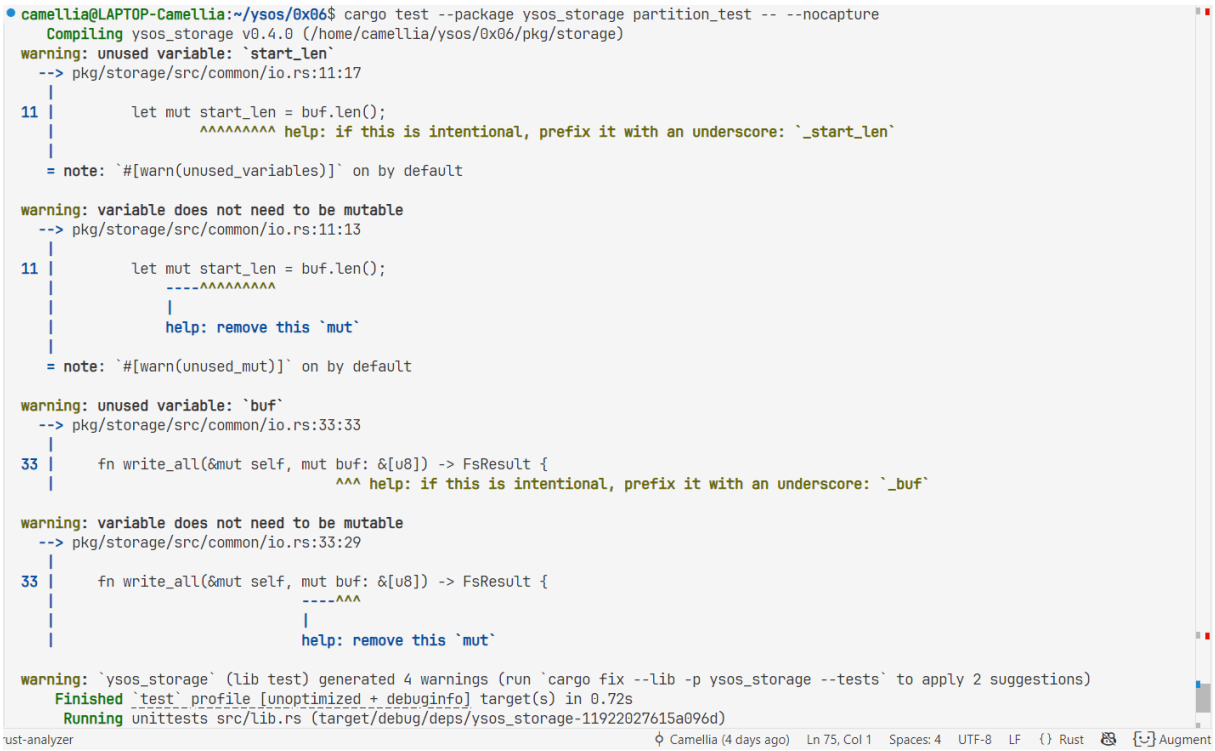
4. 在 `storage/src/partition/mod.rs` 中实现了分区设备的 `read_block` 和 `write_block` 方法

```
self.inner.read_block(self.offset + offset, block)
self.inner.write_block(self.offset + offset, block)
```

5. 运行单元测试

1. 先注释掉 `lib.rs` 中对其他模块的引用，然后在 `pkg/storage` 目录下执行命令 `cargo test --package ysos_storage partition_test -- --nocaptrue` 运行单元测试。





所有的值都与测试用例中的期望值完全匹配！至此我们完全实现了 MBR 分区表的解析。

二. 磁盘驱动

1. `pkg/kernel`是内核包，需要使用`storage`包的功能，因此在`pkg/kernel/Cargo.toml`中添加`storage`包引用

```
[dependencies]
storage = { package = "ysos_storage", path = "../storage" }
```

2. 在`bus.rs`中完成发送命令的过程：

1. 将 28-bit LBA 地址按照规范存入四个寄存器
2. 设置 drive 寄存器以选择磁盘和启用 LBA 模式
3. 写入命令寄存器
4. 添加状态轮询逻辑

```
fn write_command(&mut self, drive: u8, block: u32, cmd: AtaCommand) →
storage::FsResult {
        // ...
        unsafe {
            // just 1 sector for current implementation
            self.sector_count.write(1);
            // Store the LBA28 address into four 8-bit registers
            // LBA bits 0-7 go to lba_low register
            self.lba_low.write(bytes[0]);
            // LBA bits 8-15 go to lba_mid register
            self.lba_mid.write(bytes[1]);
            // LBA bits 16-23 go to lba_high register
            self.lba_high.write(bytes[2]);

            let drive_reg = 0xE0 | ((drive & 1) << 4) | (bytes[3] & 0x0F);
            self.drive.write(drive_reg);
            // Write the command register
            self.command.write(cmd as u8);
        }
        // ...
        // Poll for the status to be not BUSY
        self.poll(AtaStatus::BUSY, false);
        // ...
        // Poll for the status to be not BUSY and DATA_REQUEST_READY
        self.poll(AtaStatus::BUSY, false);
        self.poll(AtaStatus::DATA_REQUEST_READY, true);
        Ok(())
    }
```

3. 补全 `identify_drive` 函数。可以直接调用上文实现好的 `write_command` 函数，根据规范，`block` 参数使用 `0` 进行传递。

```
pub(super) fn identify_drive(&mut self, drive: u8) -> Result<AtaDeviceType,
&'static str> {
        info!("Identifying drive {}", drive);

        // Use AtaCommand::IdentifyDevice to identify the drive
        // Call write_command with drive and 0 as the block number
        if let Err(_e) = self.write_command(drive, 0, AtaCommand::IdentifyDevice)
{
            // If the status is empty, return AtaDeviceType::None
            if self.status().is_empty() {
                return Ok(AtaDeviceType::None);
            } else {
                // Else return DeviceError::Unknown as FsError
                return Err("Unknown device error");
            }
        }
        // Poll for the status to be not BUSY
        self.poll(AtaStatus::BUSY, false);
    }
```

4. 实现 `mod.rs` 中的数据解析部分。根据实验指导，我需要从 `ATA` 识别数据中提取序列号、型号和块数量。（在 `drivers/ata/mod.rs` 中补全 `open` 函数）

```
impl AtaDrive {
    // ...
        // we only support PATA drives
        if let Ok(AtaDeviceType::Pata(res)) = BUSES[bus as
usize].lock().identify_drive(drive) {
            let mut buf = [0u8; 512];
            for (i, &word) in res.iter().enumerate() {
                let bytes = word.to_le_bytes();
                buf[i * 2] = bytes[1];
                buf[i * 2 + 1] = bytes[0];
            }
```

```
            let serial = {
                let serial_bytes = &buf[20..40];
                // Convert bytes to string, removing null terminators and
trimming
                let serial_str = core::str::from_utf8(serial_bytes)
                    .unwrap_or("")
                    .trim_end_matches('\0')
                    .trim();
                serial_str.into()
            };
            let model = {
                let model_bytes = &buf[54..94];
                // Convert bytes to string, removing null terminators and
trimming
                let model_str = core::str::from_utf8(model_bytes)
                    .unwrap_or("")
                    .trim_end_matches('\0')
                    .trim();
                model_str.into()
            };
            let blocks = {
                let word_60 = res[60];
                let word_61 = res[61];
                ((word_61 as u32) << 16) | (word_60 as u32)
            };
            // ...
    }
```

5. 在`kernel/src/lib.rs`中编写一个测试函数，用来测试`open`函数，目标是实现阶段性成果。

```
fn test_ata_drive() {
    info!("Testing ATA drive...");
    match drivers::ata::AtaDrive::open(0, 0) {
        Some(_drive) => {
            info!("ATA drive test successful!");
        }
        None => {
            warn!("ATA drive test failed - no drive found");
        }
    }
}
```

6. 阶段性成果

```
 __ __      __  _____            ____  _____
 \ \/ /___ _/ /_/ ___/___  ____  / __ \/ ___/
  \  / __ `/ __/\__ \/ _ \/ __ \/ / / /\__ \
  / / /_/ / /_ ___/ /  __/ / / / /_/ /___/ /
 /_/\__,_/\__//____/\___/_/ /_/\____//____/

                                    v0.4.0
[+] Serial Initialized.
[INFO ] Logger Initialized with level: info
[INFO ] Physical Offset  : 0xffff800000000000
[INFO ] Privilege Stack   : 0xffffff0000836208-0xffffff000083a208
[INFO ] Double Fault IST  : 0xffffff000083a208-0xffffff000083e208
[INFO ] Page Fault IST    : 0xffffff000083e208-0xffffff000083e208
[INFO ] Timer IST         : 0xffffff000083e208-0xffffff0000842208
[INFO ] Syscall IST       : 0xffffff0000842208-0xffffff0000842208
[INFO ] Total IST size    : 64 KiB
[INFO ] Kernel Heap Size :   8.000 MiB
[INFO ] Kernel Heap Initialized.
[INFO ] APIC initialized.
[INFO ] Interrupts Initialized.
[INFO ] Physical Memory    :  95.625 MiB
[INFO ] Free Usable Memory :  44.484 MiB
[INFO ] Frame Allocator initialized.
[INFO ] User Heap Initialized.
[INFO ] Process Manager Initialized.
[INFO ] Interrupts Enabled.
[INFO ] Testing ATA drive...
[INFO ] Initialized ATA Buses.
[INFO ] Identifying drive 0
[INFO ] Drive QEMU HARDDISK QM00001 (504 MiB) opened
[INFO ] ATA drive test successful!
[INFO ] YatSenOS initialized.
[+] App list: factorial, hello, shell, counter, dinner, mq, fork_test
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> exit
退出Shell...
[INFO ] YatSenOS shutting down.
camellia@LAPTOP-Camellia:~/ysos/0x06$
```

从输出日志中可以看到：

- ATA 总线初始化成功：`[INFO ] Initialized ATA Buses.`
- 磁盘识别成功：`[INFO ] Identifying drive 0`
- 磁盘信息正确解析：`[INFO ] Drive QEMU HARDDISK QM00001 (504 MiB) opened`
- 测试成功：`[INFO ] ATA drive test successful!`

这正是实验指导中期望看到的结果：“`Drive QEMU HARDDISK QM00001 (504 MiB) opened`”！

7. 接下来我们为磁盘驱动实现读写数据，首先在`bus.rs`中实现`read_pio`和`write_pio`。

```rust
pub(super) fn read_pio(
        &mut self,
        drive: u8,
        block: u32,
        buf: &mut [u8],
    ) -> Result<(), &'static str> {
        self.write_command(drive, block, AtaCommand::ReadPio)?;
```

```
        for chunk in buf.chunks_mut(2) {
            let data = self.read_data(); // Read 16-bit word from data port
            let bytes = data.to_le_bytes(); // Convert to little-endian bytes

            // Copy bytes to buffer
            if chunk.len() >= 2 {
                chunk[0] = bytes[0];
                chunk[1] = bytes[1];
            } else if chunk.len() == 1 {
                chunk[0] = bytes[0];
            }
        }
        // ...
    }
```

```
pub(super) fn write_pio(&mut self, drive: u8, block: u32, buf: &[u8]) ->
Result<(), &'static str> {
        self.write_command(drive, block, AtaCommand::WritePio)?;
        for chunk in buf.chunks(2) {
            let data = if chunk.len() >= 2 {
                // Convert two bytes to 16-bit word in little-endian format
                u16::from_le_bytes([chunk[0], chunk[1]])
            } else {
                // If only one byte, pad with zero
                u16::from_le_bytes([chunk[0], 0])
            };

            self.write_data(data); // Write 16-bit word to data port
        }
        // ...
```

8. 阶段性成果

由下图运行得到的日志信息内容可以看出我们正确获取到了首个分区的相关信息，包括其类型、起始 `LBA` 和大小。(这里我们并没有直接使用 `MbrTable::parse(drive)` ，因为这时候如果使用 storage 包就会出现很多编译错误，因此我们相当于手动解析，接下来我们先关注 storage 包的完整实现)

```
[INFO ] First partition info:
[INFO ]   Type: 0x06
[INFO ]   Start LBA: 63
[INFO ]   Size (sectors): 1032129
[INFO ]   Size (MB): 503
```

```
 / / /_/ / /_ ___/ /  __/ / / / /_/ /___/ /
/_/\__,_/\__//____/\___/_/ /_/\____//____/

                                    v0.4.0
[+] Serial Initialized.
[INFO ] Logger Initialized with level: info
[INFO ] Physical Offset  : 0xffff800000000000
[INFO ] Privilege Stack   : 0xffffff0000839498-0xffffff000083d498
[INFO ] Double Fault IST  : 0xffffff000083d498-0xffffff0000841498
[INFO ] Page Fault IST    : 0xffffff0000841498-0xffffff0000841498
[INFO ] Timer IST         : 0xffffff0000841498-0xffffff0000845498
[INFO ] Syscall IST       : 0xffffff0000845498-0xffffff0000845498
[INFO ] Total IST size    : 64 KiB
[INFO ] Kernel Heap Size :   8.000 MiB
[INFO ] Kernel Heap Initialized.
[INFO ] APIC initialized.
[INFO ] Interrupts Initialized.
[INFO ] Physical Memory    :  95.625 MiB
[INFO ] Free Usable Memory :  44.480 MiB
[INFO ] Frame Allocator initialized.
[INFO ] User Heap Initialized.
[INFO ] Process Manager Initialized.
[INFO ] Interrupts Enabled.
[INFO ] Testing ATA drive...
[INFO ] Initialized ATA Buses.
[INFO ] Identifying drive 0
[INFO ] Drive QEMU HARDDISK QM00001 (504 MiB) opened
[INFO ] ATA drive test successful!
[INFO ] Testing MBR read...
[INFO ] MBR read successful!
[INFO ] Valid MBR signature found!
[INFO ] First partition info:
[INFO ]   Type: 0x06
[INFO ]   Start LBA: 63
[INFO ]   Size (sectors): 1032129
[INFO ]   Size (MB): 503
[INFO ] YatSenOS initialized.
[+] App list: factorial, hello, shell, counter, dinner, mq, fork_test
欢迎使用YSOS Shell!
```

三. FAT16 文件系统

1. BPB 的实现

BPB (BIOS Parameter Block) 是 FAT 文件系统引导扇区中的关键数据结构，包含文件系统的基本参数。我实现了以下字段的访问：

1.1. BPB 标准字段 (偏移量 0-35)：

- `oem_name()` - OEM 标识符 (8 字节)
- `bytes_per_sector()` - 每扇区字节数 (u16)
- `sectors_per_cluster()` - 每簇扇区数 (u8)
- `reserved_sector_count()` - 保留扇区数 (u16)
- `fat_count()` - FAT 表数量 (u8)
- `root_entries_count()` - 根目录条目数 (u16)
- `total_sectors_16()` - 总扇区数 16 位 (u16)
- `media_descriptor()` - 媒体描述符 (u8)

- `sectors_per_fat()` - 每个 FAT 的扇区数 (u16)
- `sectors_per_track()` - 每磁道扇区数 (u16)
- `track_count()` - 磁道数 (u16)
- `hidden_sectors()` - 隐藏扇区数 (u32)
- `total_sectors_32()` - 总扇区数 32 位 (u32)

1.2. 扩展 BPB 字段 (偏移量 36-61):

- `drive_number()` - 驱动器号 (u8)
- `reserved_flags()` - 保留标志 (u8)
- `boot_signature()` - 引导签名 (u8)
- `volume_id()` - 卷 ID (u32)
- `volume_label()` - 卷标 (11 字节)
- `system_identifier()` - 系统标识符 (8 字节)

1.3. 引导签名:

- `trail()` - 引导扇区结尾签名 (u16)

1.4. 使用的技术

- 使用 `define_field!` 宏来定义字段访问器
- 正确处理了不同数据类型：u8、u16、u32 和字节数组
- 为字节数组字段自动生成了 `_str()` 方法用于字符串转换
- 实现了 `total_sectors()` 方法来正确处理 16 位和 32 位总扇区数。

阶段性成果（注释了相当一部分 error 代码之后运行单独测试）:

```
33 |     fn write_all(&mut self, mut buf: &[u8]) -> FsResult {
   |                              ^^^ help: if this is intentional, prefix it with an underscore: `_buf`

warning: variable does not need to be mutable
  --> pkg/storage/src/common/io.rs:33:29
   |
33 |     fn write_all(&mut self, mut buf: &[u8]) -> FsResult {
   |                             ----^^^
   |                             |
   |                             help: remove this `mut`

warning: `ysos_storage` (lib test) generated 4 warnings (run `cargo fix --lib -p ysos_storage --tests` to apply 2 suggestions)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.70s
     Running unittests src/lib.rs (/home/camellia/ysos/0x06/target/debug/deps/ysos_storage-11922027615a096d)

running 2 tests
Fat16 BPB {
    OEM Name: "mkfs.fat",
    Bytes per Sector: 512,
    Sectors per Cluster: 16,
    Reserved Sector Count: 1,
    FAT Count: 2,
    Root Entries Count: 512,
    Total Sectors: 122880,
    Media Descriptor: 248,
    Sectors per FAT: 32,
    Sectors per Track: 63,
    Track Count: 255,
    Hidden Sectors: 0,
    Total Sectors: 122880,
    Drive Number: 128,
    Reserved Flags: 1,
    Boot Signature: 41,
    Volume ID: 2003939515,
    Volume Label: "boot       ",
    System Identifier: "FAT16   ",
    Trail: 43605,
}
Fat16 BPB {
```

```
    Track Count: 255,
    Hidden Sectors: 0,
    Total Sectors: 122880,
    Drive Number: 128,
    Reserved Flags: 1,
    Boot Signature: 41,
    Volume ID: 2003939515,
    Volume Label: "boot       ",
    System Identifier: "FAT16   ",
    Trail: 43605,
}
Fat16 BPB {
    OEM Name: "MSWIN4.1",
    Bytes per Sector: 512,
    Sectors per Cluster: 16,
    Reserved Sector Count: 1,
    FAT Count: 2,
    Root Entries Count: 512,
    Total Sectors: 1032129,
    Media Descriptor: 248,
    Sectors per FAT: 252,
    Sectors per Track: 63,
    Track Count: 16,
    Hidden Sectors: 63,
    Total Sectors: 1032129,
    Drive Number: 128,
    Reserved Flags: 0,
    Boot Signature: 41,
    Volume ID: 4206762749,
    Volume Label: "QEMU VVFAT ",
    System Identifier: "FAT16   ",
    Trail: 43605,
}
test fs::fat16::bpb::tests::test_fat16_bpb_1 ... ok
test fs::fat16::bpb::tests::test_fat16_bpb_2 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

2. 补全 DirEntry

具体实现：

2.1. 1. DirEntry::parse 函数

根据 FAT16 标准 8.3 格式的 32 字节目录项结构，正确解析：

- 文件名 (偏移量 0-10): 8.3 格式的文件名
- 属性 (偏移量 11): 文件属性标志
- 时间戳: 创建时间、修改时间、访问时间
- 簇号 (偏移量 20-21, 26-27): 文件起始簇号
- 文件大小 (偏移量 28-31): 文件大小（字节）

2.2. 2. parse_datetime 函数

实现了 FAT16 时间/日期格式的解析：

- 日期格式: 位 15-9 年份（相对于 1980），位 8-5 月份，位 4-0 日期
- 时间格式: 位 15-11 小时，位 10-5 分钟，位 4-0 秒/2

2.3. 3. DirEntry 辅助方法

- `is_valid()`：检查目录项是否有效
- `is_long_name()`：检查是否为长文件名条目
- `is_directory()`：检查是否为目录

2.4. 4. ShortFileName::parse 函数

实现了从字符串到 8.3 格式文件名的转换：

- 错误处理: 空文件名、文件名过长、无效字符、句点位置错误
- 格式转换: 转换为大写，分离基本名和扩展名
- 填充: 使用空格 (0x20) 进行右填充
- 字符验证: 检查 FAT16 禁用的字符

单独运行测试：

```
warning: `ysos_storage` (lib test) generated 4 warnings (run `cargo fix --lib -p ysos_storage --tests` to apply 2 suggestions)
    Finished `test` profile [unoptimized + debuginfo] target(s) in 1.37s
     Running unittests src/lib.rs (/home/camellia/ysos/0x06/target/debug/deps/ysos_storage-11922027615a096d)

running 1 test
DirEntry {
    filename: KERNEL.ELF,
    modified_time: 2020-06-16T23:48:30Z,
    created_time: 2020-06-16T23:48:30Z,
    accessed_time: 2020-06-16T00:00:00Z,
    cluster: 0x00000002,
    attributes: Attributes(
        ARCHIVE,
    ),
    size: 976112,
}
test fs::fat16::direntry::tests::test_dir_entry ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 3 filtered out; finished in 0.00s

camellia@LAPTOP-Camellia:~/ysos/0x06/pkg/storage$
```

测试用例成功通过，验证了：

- 正确解析了 “KERNEL.ELF” 文件的所有字段
- 文件大小: 976112 字节
- 簇号: 2

- 属性: ARCHIVE
- 时间戳: 创建和修改时间为 2020-06-16T23:48:30Z，访问时间为 2020-06-16T00:00:00Z

3. Fat16Impl

3.1. 1. Fat16Impl 中的核心函数

3.1.1. `cluster_to_sector()` - 将 `cluster` 转换为 `sector`

将簇号转换为对应的扇区号，是文件系统访问的基础。

```
pub fn cluster_to_sector(&self, cluster: &Cluster) -> usize {
    match *cluster {
        Cluster::ROOT_DIR => self.first_root_dir_sector,
        Cluster(c) => {
            // 根据FAT16规范计算簇的第一个扇区
            // FirstSectorofCluster = ((N - 2) * BPB_SecPerClus) +
FirstDataSector
            self.first_data_sector + ((c as usize - 2) *
self.bpb.sectors_per_cluster() as usize)
        }
    }
}
```

3.1.2. `get_next_cluster()` - 利用 FAT 表获取下一个 `cluster`

通过 FAT 表查找文件的下一个簇，实现文件数据的连续访问。

```
pub fn get_next_cluster(&self, cluster: &Cluster) -> FsResult<Cluster> {
    match *cluster {
        Cluster::ROOT_DIR => Err(FsError::InvalidOperation),
        Cluster(c) => {
            // 计算FAT表中的偏移量
            // 每个FAT16条目占2字节
            let fat_offset = c as usize * 2;
            let fat_sector = self.first_fat_sector + (fat_offset /
self.bytes_per_sector);
            let offset_in_sector = fat_offset % self.bytes_per_sector;

            // 读取包含FAT条目的扇区
            let mut buffer = [0u8; 512];
            self.device.read_block(fat_sector, &mut buffer)?;
```

```
            // 从扇区中提取FAT条目值
            let next_cluster = u16::from_le_bytes([
                buffer[offset_in_sector],
                buffer[offset_in_sector + 1],
            ]);

            // 检查特殊值并返回适当的簇
            match next_cluster {
                0 => Err(FsError::InvalidCluster),
                0xFFF7 => Err(FsError::BadCluster),
                0xFFF8..=0xFFFF => Err(FsError::EndOfFile),
                _ => Ok(Cluster(next_cluster)),
            }
        }
    }
}
```

3.1.3. `find_dir_entry()` - 根据文件夹信息获取指定名字的 `DirEntry`

在目录中查找特定名称的文件或子目录。

3.1.4. `read_dir_entries()` - 遍历文件夹获取其中文件信息

读取目录中的所有条目，支持文件系统的目录浏览功能。

3.2. 2. `FileSystem` trait 的完整实现

3.2.1. `read_dir()` - 返回 `Iterator<Item = Metadata>`

实现目录内容的遍历，使用 `Vec::into_iter` 作为返回值。

3.2.2. `open_file()` - 返回 `FileHandle`

打开文件并返回文件句柄，使用 `Box` 包装的 `File` 结构体。

3.2.3. `metadata()` - 获取文件/目录元数据

获取文件或目录的详细信息，如大小、创建时间等。

3.2.4. `exists()` - 检查文件/目录是否存在

验证指定路径的文件或目录是否存在于文件系统中。

3.3. 3. `File` 的 `Read` trait 实现

3.3.1. `cluster` 链遍历

使用 `get_next_cluster()` 读取 FAT 表，实现对文件数据的连续访问。

3.3.2. `offset` 管理

实时更新当前文件读取位置，确保正确读取文件的不同部分。

3.3.3. 多 `sector` 处理

根据 `BPB` 信息处理 `cluster` 中的多个 `sector`，支持大文件的读取。

3.3.4. 缓冲区长度处理

考虑文件长度、块长度和缓冲区长度来决定读取终止点，确保读取操作的正确性。

单独运行测试结果：

```
    Hidden Sectors: 63,
    Total Sectors: 1032129,
    Drive Number: 128,
    Reserved Flags: 0,
    Boot Signature: 41,
    Volume ID: 4206762749,
    Volume Label: "QEMU VVFAT ",
    System Identifier: "FAT16   ",
    Trail: 43605,
}
Fat16 BPB {
    OEM Name: "mkfs.fat",
    Bytes per Sector: 512,
    Sectors per Cluster: 16,
    Reserved Sector Count: 1,
    FAT Count: 2,
    Root Entries Count: 512,
    Total Sectors: 122880,
    Media Descriptor: 248,
    Sectors per FAT: 32,
    Sectors per Track: 63,
    Track Count: 255,
    Hidden Sectors: 0,
    Total Sectors: 122880,
    Drive Number: 128,
    Reserved Flags: 1,
    Boot Signature: 41,
    Volume ID: 2003939515,
    Volume Label: "boot       ",
    System Identifier: "FAT16   ",
    Trail: 43605,
}
test fs::fat16::direntry::tests::test_dir_entry ... ok
test fs::fat16::bpb::tests::test_fat16_bpb_2 ... ok
test fs::fat16::bpb::tests::test_fat16_bpb_1 ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in 0.00s

camellia@LAPTOP-Camellia:~/ysos/0x06/pkg/storage$
```

可以看出所有测试都顺利通过！

四. 接入操作系统

参考给出的 `kernel/src/drivers/filesystem.rs`，结合你的 `AtaDrive`，将 `Partition` 作为参数，初始化一个 `Fat16` 结构体，并使用 `Mount` 将其存放在 `ROOTFS` 变量中。

1. 在 `pkg/kernel/src/drivers/filesystem.rs` 中实现了文件系统的初始化功能：

```rust
pub static ROOTFS: spin::Once<Mount> = spin::Once::new();

pub fn get_rootfs() -> &'static Mount {
    ROOTFS.get().unwrap()
}

pub fn init() {
    info!("Opening disk device...");

    // 使用 AtaDrive 打开磁盘设备
    let drive = AtaDrive::open(0, 0).expect("Failed to open disk device");

    // 解析 MBR 并获取第一个分区作为参数
    let part = MbrTable::parse(drive)
        .expect("Failed to parse MBR")
        .partitions()
        .expect("Failed to get partitions")
        .remove(0);

    info!("Mounting filesystem...");

    // 使用 Partition 初始化 Fat16 结构体，并用 Mount 封装存储在 ROOTFS 中
    ROOTFS.call_once(|| Mount::new(Box::new(Fat16::new(part)), "/".into()));

    trace!("Root filesystem: {:#?}", ROOTFS.get().unwrap());

    info!("Initialized Filesystem.");
}
```

- 该实现完成了以下关键步骤：
  - 使用 `AtaDrive::open(0, 0)` 打开第一个 `ATA` 磁盘设备
  - 通过 `MbrTable::parse()` 解析主引导记录，获取第一个分区
  - 将分区作为参数传递给 `Fat16::new()` 初始化 `FAT16` 文件系统
  - 使用 `Mount::new()` 将文件系统封装并存储在全局静态变量 `ROOTFS` 中
  - 使用 `spin::Once` 确保线程安全的单例初始化

在内核初始化过程中调用 `drivers::filesystem::init()` 来完成文件系统的接入：

```rust
pub fn init() {
    // Test ATA drive
    test_ata_drive();
    // Initialize filesystem
    drivers::filesystem::init();
    info!("YatSenOS initialized.");
}
```

2. 列出目录

补全 `ls` 函数，根据 `read_dir` 返回的迭代器，列出并打印文件夹内的文件信息：

```
pub fn ls(root_path: &str) {
    let iter = match get_rootfs().read_dir(root_path) {
        Ok(iter) ⇒ iter,
        Err(err) ⇒ {
            warn!("{:?}", err);
            return;
        }
    };

    // 打印表格头部
    println!("{:<20} {:>8} {:>20}", "Name", "Size", "Modified");
    println!("{:-<50}", "");

    // 遍历目录条目并格式化输出
    for meta in iter {
        let name = if meta.is_dir() {
            format!("{}/", meta.name)  // 目录名后添加 '/'
        } else {
            meta.name.clone()
        };

        let size_str = if meta.is_dir() {
            "<DIR>".to_string()
        } else {
            let (size, unit) = crate::humanized_size_short(meta.len as u64);
            format!("{:.1}{}", size, unit)  // 使用 humanized_size_short 格式化文件
大小
        };

        let modified_str = if let Some(modified) = meta.modified {
            format!("{}", modified.format("%Y-%m-%d %H:%M"))  // 格式化日期时间
        } else {
            "Unknown".to_string()
        };

        println!("{:<20} {:>8} {:>20}", name, size_str, modified_str);
    }
}
```

2.1. 系统调用实现

实现了 `ListDir` 系统调用，专用于在内核态直接打印文件夹信息：

```rust
// 在 pkg/syscall/src/lib.rs 中添加
#[repr(usize)]
#[derive(Clone, Debug, FromPrimitive)]
pub enum Syscall {
    Read = 0,
    Write = 1,
    Sem = 2,
    // ...
    Open = 62,
    Close = 63,
    ListDir = 65530,
    // ...
}
```

```rust
// 在 pkg/lib/src/syscall.rs 中添加用户态接口
#[inline(always)]
pub fn sys_list_dir(path: &str) {
    syscall!(Syscall::ListDir, path.as_ptr() as u64, path.len() as u64);
}
```

```rust
// 在 pkg/kernel/src/interrupt/syscall/service.rs 中实现系统调用处理
pub fn list_dir(args: &SyscallArgs) {
    let ptr = args.arg0 as *const u8;
    let len = args.arg1;

    if ptr.is_null() || len == 0 {
        return;
    }

    let path = unsafe {
        let slice = core::slice::from_raw_parts(ptr, len);
        match core::str::from_utf8(slice) {
            Ok(s) => s,
            Err(_) => return,
        }
    };

    filesystem::ls(path);
}
```

2.2. Shell 命令更新

在 `Shell` 中更新 `ls` 命令使其调用文件系统功能：

```
"ls" => {
    let path = if args.is_empty() { "/" } else { args[0] };
    println!("目录内容 '{}':", path);
    sys_list_dir(path);
},
```

3. 读取文件

为了读取文件，实现了 `open-read-close` 工作流程。

3.1. 扩展 Resource 枚举

首先在 `pkg/kernel/src/utils/resource.rs` 中扩展 `Resource` 枚举：

```
use storage::FileHandle;

#[derive(Debug)]
pub enum Resource {
    Console(StdIO),
    File(FileHandle),  // 添加文件资源类型
    Null,
}
```

在 `Resource` 的实现中添加文件读取支持：

```
pub fn read(&mut self, buf: &mut [u8]) -> Option<usize> {
    match self {
        Resource::Console(stdio) => {
            // ... Console 处理逻辑
        },
        Resource::File(file_handle) => {
            match file_handle.read(buf) {
                Ok(bytes_read) => Some(bytes_read),
                Err(_) => None,
            }
        },
        Resource::Null => Some(0),
    }
}
```

```rust
pub fn write(&mut self, buf: &[u8]) -> Option<usize> {
    match self {
        Resource::Console(stdio) => {
            // ... Console 处理逻辑
        },
        Resource::File(_file_handle) => {
            // 文件写入暂不实现，根据实验要求可以直接忽略
            None
        },
        Resource::Null => Some(buf.len()),
    }
}
```

3.2. 文件操作系统调用

实现文件操作的系统调用：

```rust
// 在 pkg/lib/src/syscall.rs 中添加
#[inline(always)]
pub fn sys_open(path: &str) -> u8 {
    syscall!(Syscall::Open, path.as_ptr() as u64, path.len() as u64) as u8
}
#[inline(always)]
pub fn sys_close(fd: u8) -> isize {
    syscall!(Syscall::Close, fd as u64) as isize
}
```

```rust
// 在 pkg/kernel/src/interrupt/syscall/service.rs 中实现
pub fn sys_open(args: &SyscallArgs) -> usize {
    let ptr = args.arg0 as *const u8;
    let len = args.arg1;

    if ptr.is_null() || len == 0 {
        return 0;
    }
    let path = unsafe {
        let slice = core::slice::from_raw_parts(ptr, len);
        match core::str::from_utf8(slice) {
            Ok(s) => s,
            Err(_) => return 0,
        }
    };
    match open_file(path) {
        Ok(fd) => fd as usize,
```

```
        Err(_) => 0,
    }
}
pub fn sys_close(args: &SyscallArgs) -> usize {
    let fd = args.arg0 as u8;
    if close_file(fd) { 0 } else { 1 }
}
```

3.3. 进程资源管理

在进程模块中添加文件操作支持：(在 `pkg/kernel/src/proc/data.rs` 中添加)

```
impl ProcessData {
    pub fn open_resource(&self, resource: crate::utils::Resource) -> u8 {
        self.resources.write().open(resource)
    }
    pub fn close_resource(&self, fd: u8) -> bool {
        self.resources.write().close(fd)
    }
}
```

```
// 在 pkg/kernel/src/proc/mod.rs 中添加
pub fn open_file(path: &str) -> Result<u8, ()> {
    x86_64::instructions::interrupts::without_interrupts(|| {
        match crate::drivers::filesystem::get_rootfs().open_file(path) {
            Ok(file_handle) => {
                let current_proc = get_process_manager().current();
                let proc_data = current_proc.read().proc_data().unwrap().clone();
                let fd =
proc_data.open_resource(crate::utils::Resource::File(file_handle));
                Ok(fd)
            }
            Err(_) => Err(()),
        }
    })
}
pub fn close_file(fd: u8) -> bool {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let current_proc = get_process_manager().current();
        let proc_data = current_proc.read().proc_data().unwrap().clone();
        proc_data.close_resource(fd)
    })
}
```

3.4. Shell cat 命令实现

在 `Shell` 中实现 `cat` 命令：

```
"cat" => {
    if args.is_empty() {
        println!("错误: 请指定要显示的文件名");
    } else {
        let filename = args[0];
        cat_file(filename);
    }
},

fn cat_file(filename: &str) {
    // 尝试打开文件
    let fd = sys_open(filename);
    if fd == 0 {
        println!("错误: 无法打开文件 '{}'", filename);
        return;
    }

    // 读取文件内容
    let mut buffer = [0u8; 1024];
    let mut total_read = 0;

    loop {
        match sys_read(fd, &mut buffer) {
            Some(bytes_read) => {
                if bytes_read == 0 {
                    break; // 文件结束
                }

                // 将读取的字节转换为字符串并打印
                if let Ok(content) = core::str::from_utf8(&buffer[..bytes_read])
{
                    print!("{}", content);
                } else {
                    println!("错误: 文件 '{}' 包含非UTF-8数据", filename);
                    break;
                }

                total_read += bytes_read;
```

```
                // 如果读取的字节数少于缓冲区大小，说明已到文件末尾
                if bytes_read < buffer.len() {
                    break;
                }
            }
            None => {
                println!("错误: 读取文件 '{}' 失败", filename);
                break;
            }
        }
    }

    // 关闭文件
    sys_close(fd);

    if total_read == 0 {
        println!("文件 '{}' 为空", filename);
    }
}
```

4. 阶段性成果

成功为 Shell 添加了 ls 和 cat 指令。测试结果显示:

```
ysos> ls
目录内容 '/':
Name                          Size              Modified
-------------------------------------------------------
QEMU VVF.AT                    0.0B     1970-01-01 00:00
NVVARS                        10.3K     2025-06-02 07:01
HELLO.TXT                    279.0B     2025-06-02 14:53
KERNEL.ELF                   428.9K     2025-06-02 15:01
EFI/                          <DIR>     2025-05-28 14:37
APP/                          <DIR>     2025-05-28 14:37
```

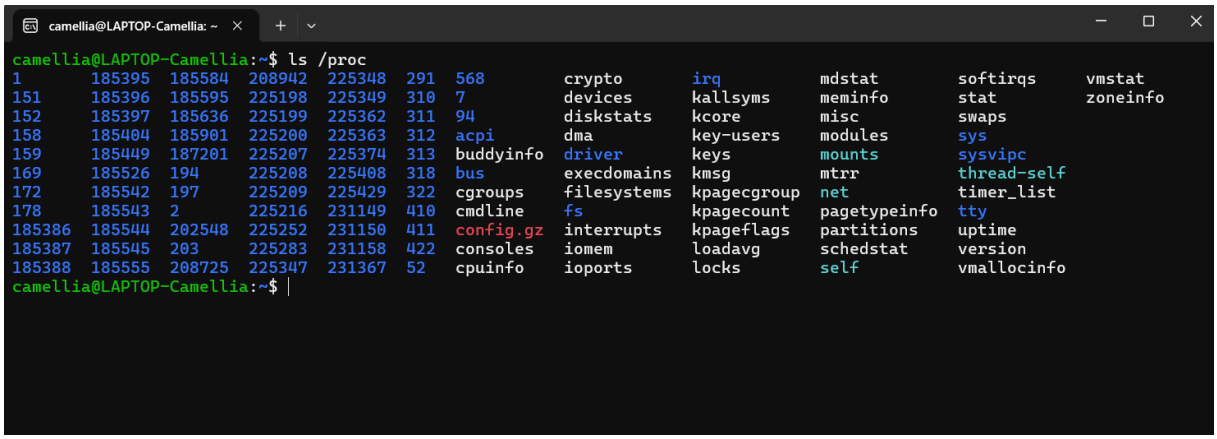
```
ysos> cat HELLO.TXT
Hello filesystem from 23336152!
This is a test file to demonstrate the FAT16 filesystem implementation.
The file system can now:
- List directory contents with ls command
- Read file contents with cat command
- Open and close files through system calls

Student Number: 23336152
```

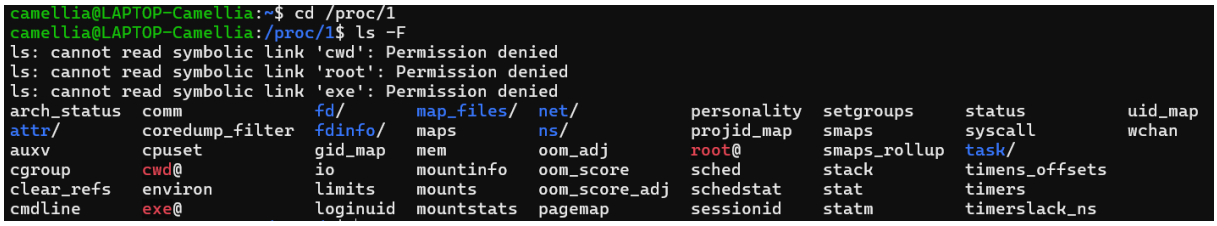
通过这些实现，成功完成了文件系统与操作系统的集成，实现了完整的文件操作功能，包括目录列表、文件读取等核心功能。文件系统能够正确处理 `FAT16` 格式，支持文件和目录的基本操作，为操作系统提供了完整的存储管理能力。

五. 探索 `Linux` 文件系统

1. `PROCFS`



1.1. 首先我们看到目录下有很多的数字，经过查阅资料，这些应该是正在运行的进程的 `PID`，我们进入 `1` 目录下：



这些文件和子目录都提供了关于 `PID` 为 `1` 的 `systemd` 进程的运行时信息和状态。

1.2. 接下来我们探索 `cpuinfo` 和 `meminfo`：

```
camellia@LAPTOP-Camellia:/proc$ cat cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 25
model           : 68
model name      : AMD Ryzen 7 7735H with Radeon Graphics
stepping        : 1
microcode       : 0xffffffff
cpu MHz         : 3194.004
cache size      : 512 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht s
yscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl tsc_reliable nonstop_tsc cpuid extd_apicid pni pc
lmulqdq ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm cmp_legacy svm cr8_legacy
 abm sse4a misalignsse 3dnowprefetch osvw topoext perfctr_core ssbd ibrs ibpb stibp vmmcall fsgsbase bmi1 avx2 smep bmi2
 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves clzero xsaveerptr arat npt nrip_save
 tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold v_vmsave_vmload umip vaes vpclmulqdq rdpid fsrm
bugs            : sysret_ss_attrs null_seg spectre_v1 spectre_v2 spec_store_bypass srso
```

呈现了处理器型号、核心数、线程数以及缓存大小等信息。

```
camellia@LAPTOP-Camellia:/proc$ cat meminfo
MemTotal:        7743824 kB
MemFree:          921356 kB
MemAvailable:    2287768 kB
Buffers:           29840 kB
Cached:          1520808 kB
SwapCached:         6492 kB
Active:           845088 kB
Inactive:        5476360 kB
Active(anon):      17012 kB
Inactive(anon):  4756900 kB
Active(file):     828076 kB
Inactive(file):   719460 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       2097152 kB
SwapFree:        2049180 kB
Dirty:                44 kB
Writeback:             0 kB
AnonPages:       4476704 kB
Mapped:           233452 kB
Shmem:              3104 kB
KReclaimable:      67288 kB
Slab:             142096 kB
SReclaimable:      67288 kB
```

呈现了当前 WSL 的物理内容总容量以及系统当前的内存使用状况，包括物理内存、缓存、缓冲区、交换空间以及不同类型内存页的活跃程度。

1.3. 下一步看 `loadavg` 和 `uptime` 里面记录了哪些内容

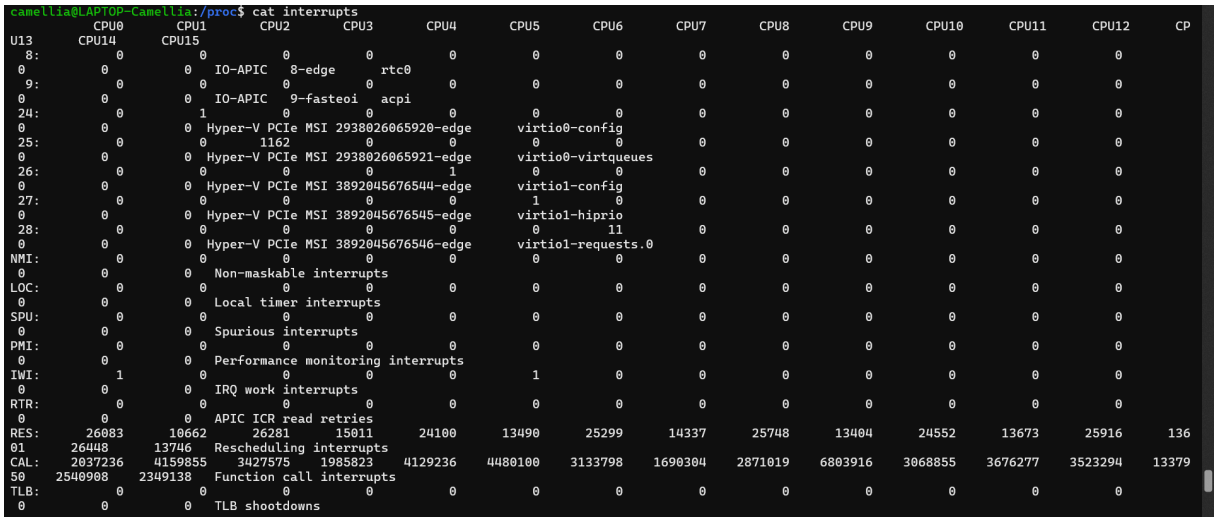
```
camellia@LAPTOP-Camellia:/proc$ cat loadavg
0.63 0.62 0.56 1/500 255542
```

- `/proc/loadavg` 文件包含了关于系统平均负载（`Load Average`）的信息。平均负载是衡量系统在给定时间段（过去的一分钟、五分钟和十五分钟）内，处于可运行状态（`running`）或不可中断睡眠状态（`uninterruptible sleep`）的进程的平均数量，以及当前运行的进程数量与总进程数的比值。最后的 255542 是最新创建的进程的 PID。
- `/proc/uptime` 文件包含了系统自启动以来的时间信息。它通常包含两个数字：

`26784.24`：系统自启动以来的总秒数。

换算： 26784.24 秒 ≈ 446.4 分钟 ≈ 7.44 小时。这意味着我的 WSL2 实例已经运行了大约 7.44 小时。 417987.75：系统空闲的总秒数。

1.4. 接下来看 `interrupts`



`/proc/interrupts` 文件显示了每个 `CPU` 核心（或线程）上接收到的硬件中断（`Interrupt Request`，`IRQ`）的数量。由于 WSL2 是一个虚拟机，因此可以看到很多中断与 Hyper-V PCIe MSI 和 virtio 相关。

1.5. `self/status`

```
camellia@LAPTOP-Camellia:/proc$ cat self/status
Name:   cat
Umask:  0022
State:  R (running)
Tgid:   262410
Ngid:   0
Pid:    262410
PPid:   231158
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 256
Groups: 4 24 27 30 46 100 1000 1001
NStgid: 262410
NSpid:  262410
NSpgid: 262410
NSsid:  231158
VmPeak:     3272 kB
VmSize:     3272 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      1068 kB
VmRSS:      1068 kB
RssAnon:              88 kB
RssFile:             980 kB
RssShmem:              0 kB
VmData:      360 kB
```

`/proc/self` 是一个特殊的符号链接，它总是指向当前正在访问它的进程的 `/proc/PID` 目录。所以，当你执行 `cat /proc/self/status` 时，你看到的是 `cat` 命令本身的进程状态信息。这

解释了为什么 `Name: cat` 出现在输出的第一行。 `status` 文件提供了该进程（在这里是 `cat` 命令）的详细信息，包括其身份、状态、内存使用情况等。

1.6. `self/smaps`

```
camellia@LAPTOP-Camellia:/proc$ cat /proc/self/smaps
5568b8abd000-5568b8abf000 r--p 00000000 08:20 1507                       /usr/bin/cat
Size:                  8 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Rss:                   8 kB
Pss:                   8 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         8 kB
Private_Dirty:         0 kB
Referenced:            8 kB
Anonymous:             0 kB
LazyFree:              0 kB
AnonHugePages:         0 kB
ShmemPmdMapped:        0 kB
FilePmdMapped:         0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
Locked:                0 kB
THPeligible:    0
VmFlags: rd mr mw me
5568b8abf000-5568b8ac4000 r-xp 00002000 08:20 1507                       /usr/bin/cat
Size:                 20 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Rss:                  20 kB
```

`/proc/self/smaps` 文件比 `status` 文件提供了更详细的进程内存使用信息，特别是关于内存映射（`memory mappings`）的信息。它会列出进程虚拟地址空间中每一个内存区域的详细信息，包括该区域的大小、权限、以及最重要的是，其实际的物理内存使用情况（`RSS` 和 `PSS`）。

1.7. 结合搜索，回答 `echo 1 > /proc/sys/net/ipv4/ip_forward` 有什么用？尝试据此命令，从系统调用角度，解释“一切皆文件”的优势。

```
camellia@LAPTOP-Camellia:/proc$ cat /proc/sys/net/ipv4/ip_forward
0
```

1. 命令的字面意思：
   - `echo 1`：将数字 `1` 输出到标准输出。
   - `>`：这是一个重定向操作符，它将 `echo 1` 的标准输出重定向到指定的文件。
   - `/proc/sys/net/ipv4/ip_forward`：这是 `/proc` 文件系统下的一个特殊文件，它代表了 `Linux` 内核的一个配置参数。具体来说，它控制着 `IPv4` 数据包的转发功能。
2. 在网络配置中的具体作用：
   - 当你执行 `cat /proc/sys/net/ipv4/ip_forward` 看到 `0` 时，表示 `IP` 转发功能是禁用的。
   - 如果将其值改为 `1`（通过 `echo 1 > ...`），则表示 `IP` 转发功能是启用的。

IP 转发（`IP Forwarding`）是指路由器或主机在接收到一个目的地址不是自身的数据包时，将其转发到网络中其他主机的能力。

- 当 `ip_forward` 为 `0` (禁用) 时： `Linux` 系统（例如我的 `WSL2` 实例）只会处理目的地址是它自己的数据包。如果收到一个目的地址是其他机器的数据包，它会直接丢弃。这适用于普通的用户工作站。
- 当 `ip_forward` 为 `1` (启用) 时：你的 `Linux` 系统将作为路由器或网关。它会接收目的地址不是自己的数据包，并根据其路由表将其转发到正确的目的地。这在以下场景中非常有用：
  - 路由器： 将连接在不同子网的设备连接起来。
  - `NAT` (网络地址转换)： 允许内网设备通过一个公共 `IP` 地址访问互联网。
  - `VPN` 服务器： 转发 `VPN` 客户端的数据包。
  - 容器或虚拟机网络： 例如 `Docker` 容器或 `KVM` 虚拟机通常需要宿主机启用 `IP` 转发才能访问外部网络。

因此，将 `ip_forward` 设置为 `1` 是将一个 `Linux` 系统配置为网络路由器或网关的关键一步。

3. 从系统调用角度，解释 “一切皆文件” 的优势

“一切皆文件 (`Everything is a file`)” 是 `Unix/Linux` 哲学的一个核心原则。它意味着：

- 常规文件 (`regular files`)
- 目录 (`directories`)
- 设备 (`devices`) (如硬盘、键盘、打印机、网卡)
- 管道 (`pipes`)
- 套接字 (`sockets`)
- 甚至许多内核参数和运行时信息 (如 `/proc` 和 `/sys` 中的内容)

都被抽象为文件，并可以通过统一的文件 `I/O` 系统调用接口 (`open(), read(), write(), close()` 等) 进行操作。

2. `devfs`

```
camellia@LAPTOP-Camellia:/dev$ cat /dev/null
camellia@LAPTOP-Camellia:/dev$ head -c 10 /dev/zero | hexdump -C
00000000  00 00 00 00 00 00 00 00  00 00                    |..........|
0000000a
camellia@LAPTOP-Camellia:/dev$ head -c 10 /dev/random | hexdump -C
00000000  8f 03 22 7e 1f ad db ba  7b 19                    |.."~....{.|
0000000a
camellia@LAPTOP-Camellia:/dev$ head -c 10 /dev/urandom | hexdump -C
00000000  e5 42 78 05 17 6c c2 57  92 3e                    |.Bx..l.W.>|
0000000a
```

Linux 将设备也作为“文件”，默认挂载于 /dev 目录下，探索他们并回答如下问题：

1. `/dev/null` 、 `/dev/zero` 、 `/dev/random` 和 `/dev/urandom` 分别有什么作用？

`/dev/null` 是一个空设备或“黑洞”。

- 读取时： 它立即返回文件结束符 (EOF)，就像一个空文件一样。
- 写入时： 任何写入它的数据都会被立即丢弃，不会存储，也不会产生任何错误。
- 主要用途：

- 丢弃不需要的输出： 例如， `command > /dev/null` 可以阻止命令的正常输出显示在终端上。
- 提供空输入： 例如， `command < /dev/null` 可以给命令提供一个空的输入流。

剩余几个分别是零源设备、随机数生成器和伪随机数生成器。

2. 运行 `head /dev.kmsg`，解释该文件作用

```
camellia@LAPTOP-Camellia:/dev$ head /dev/kmsg
5,0,0,-;Linux version 5.15.167.4-microsoft-standard-WSL2 (root@f9c826d3017f) (gcc (GCC) 11.2.0, GNU ld (GNU Binutils) 2.
37) #1 SMP Tue Nov 5 00:21:55 UTC 2024
6,1,0,-;Command line: initrd=\x5cinitrd.img WSL_ROOT_INIT=1 panic=-1 nr_cpus=16 hv_utils.timesync_implicit=1 console=hvc
0 debug pty.legacy_count=0 WSL_ENABLE_CRASH_DUMP=1
6,2,0,-;KERNEL supported cpus:
6,3,0,-;  Intel GenuineIntel
6,4,0,-;  AMD AuthenticAMD
6,5,0,-;  Centaur CentaurHauls
6,6,0,-;BIOS-provided physical RAM map:
6,7,0,-;BIOS-e820: [mem 0x0000000000000000-0x000000000009ffff] usable
6,8,0,-;BIOS-e820: [mem 0x00000000000e0000-0x00000000000e0fff] reserved
6,9,0,-;BIOS-e820: [mem 0x0000000000100000-0x00000000001fffff] ACPI data
```

如图成功指出了我的 `WSL` 系统， `/dev/kmsg` 这个文件的核心作用就是：

提供 Linux 内核在运行过程中产生的实时消息流，包括系统启动过程、硬件配置、驱动加载、运行时警告、错误和调试信息等，可以看作是“详细的系统启动过程和硬件配置的快照”以及系统运行时的实时日志。

3. `/dev/sdX` 和 `/dev/sdX1` （X 为一个字母，1 为数字）是什么？有什么区别？如果你正在使用的 Linux 系统中不存在这样的文件，请找到功能类似的文件，并解释。

```
camellia@LAPTOP-Camellia:/dev$ ls
autofs           fuse          loop0   ptmx        ram5    stderr  tty19  tty32  tty46  tty6    vcs2  vcsu3
block            hugepages     loop1   ptp0        ram6    stdin   tty2   tty33  tty47  tty60   vcs3  vcsu4
bsg              hvc0          loop2   ptp_hyperv  ram7    stdout  tty20  tty34  tty48  tty61   vcs4  vcsu5
btrfs-control    hvc1          loop3   pts         ram8    tty     tty21  tty35  tty49  tty62   vcs5  vcsu6
bus              hvc2          loop4   ram0        ram9    tty0    tty22  tty36  tty5   tty63   vcs6  vfio
char             hvc3          loop5   ram1        random  tty1    tty23  tty37  tty50  tty7    vcsa  vhost-net
console          hvc4          loop6   ram10       rtc     tty10   tty24  tty38  tty51  tty8    vcsa1 virtio-ports
core             hvc5          loop7   ram11       rtc0    tty11   tty25  tty39  tty52  tty9    vcsa2 vport0p0
cpu_dma_latency  hvc6          mapper  ram12       sda     tty12   tty26  tty4   tty53  ttyS0   vcsa3 vport0p1
cuse             hvc7          mem     ram13       sdb     tty13   tty27  tty40  tty54  ttyS1   vcsa4 vport0p2
disk             initctl       mqueue  ram14       sdc     tty14   tty28  tty41  tty55  ttyS2   vcsa5 vsock
dri              kmsg          net     ram15       sg0     tty15   tty29  tty42  tty56  ttyS3   vcsa6 zero
dxg              kvm           null    ram2        sg1     tty16   tty3   tty43  tty57  urandom vcsu
fd               log           nvram   ram3        sg2     tty17   tty30  tty44  tty58  vcs     vcsu1
full             loop-control  ppp     ram4        shm     tty18   tty31  tty45  tty59  vcs1    vcsu2
```

```
camellia@LAPTOP-Camellia:/dev$ ls -l /dev/sda
brw-rw---- 1 root disk 8, 0 Jun  2 09:56 /dev/sda
camellia@LAPTOP-Camellia:/dev$ lsblk
NAME
    MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda   8:0    0 388.4M  1 disk
sdb   8:16   0     2G  0 disk [SWAP]
sdc   8:32   0     1T  0 disk /mnt/wslg/distro
                               /
```

- `/dev/sdX` (例如 `/dev/sda`，`/dev/sdb`，`/dev/sdc`):
  - 是什么: 它代表一个完整的 SCSI 磁盘设备。在 Linux 系统中，这通常指代 SATA 硬盘、USB 驱动器、SSD 等存储设备，因为它们在内核层面通过 SCSI 模拟层进行管理。X 是一个字母，按照系统识别设备的顺序从 a 开始递增。
  - 作用: /dev/sdX 代表整个物理或虚拟存储介质。对它的操作通常是针对整个磁盘的，例如创建分区表、擦除整个磁盘数据等。

- /dev/sdXN (例如 /dev/sda1, /dev/sdb2 等，其中 N 为数字):
  - 是什么: 它代表 /dev/sdX 这个磁盘上的 一个分区。N 是该磁盘上的分区号。
  - 作用: 每个分区通常被格式化为特定的文件系统（如 ext4、NTFS），并可以独立地挂载到文件系统的某个目录下，用于存储数据。在大多数日常使用中，我们操作的是这些分区设备。
- 区别:
  - `/dev/sdX` 代表整个物理或虚拟存储设备。
  - `/dev/sdXN` 代表该存储设备上的一个逻辑分区。一个
  - `/dev/sdX` 设备可以包含零个或多个 `/dev/sdXN` 分区。

在我的 `Linux` 系统中（`WSL2` 环境）的情况:

1. 根据 `ls /dev` 命令的输出，我看到了 `/dev/sda`，`/dev/sdb`，`/dev/sdc` 等文件，但没有看到 `/dev/sdXN` 形式的分区文件。
2. 通过 `ls -l /dev/sda` 命令的输出 `brw-rw---- 1 root disk 8, 0 ... /dev/sda`，我确认 `/dev/sda` 是一个块设备（b），其主设备号 8 进一步证实了它是一个 `SCSI` 磁盘设备。
3. 随后运行 `lsblk` 命令，输出如下:

```
NAME
    MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda   8:0    0 388.4M  1 disk
sdb   8:16   0     2G  0 disk [SWAP]
sdc   8:32   0     1T  0 disk /mnt/wslg/distro
                               /
```

从输出中可以看出:

- `/dev/sda` 是一个 `388.4M` 的磁盘，但没有分区（没有 `/dev/sdaN` 形式的子设备）。
- `/dev/sdb` 是一个 `2G` 的磁盘，同样没有分区。
- `/dev/sdc` 是一个 `1T` 的磁盘，也没有分区。

4. `/dev/ttyX`、`/dev/loopX`、`/dev/srX` 分别代表什么设备?

- `/dev/ttyX`: 这是终端设备的表示。`X` 是一个数字，表示终端的编号。在 Linux 中，终端可以是物理的（如串口、键盘和显示器）或虚拟的（如伪终端）。
- `/dev/loopX`: 这是 loop 设备的表示。`X` 是一个数字，表示 loop 设备的编号。loop 设备是一种特殊的块设备，它允许将文件作为块设备使用。
- `/dev/srX`: 这是 SCSI 光驱设备的表示。`X` 是一个数字，表示 SCSI 光驱设备的编号。SCSI 光驱设备是一种特殊的块设备，用于访问光盘驱动器。

5. 列出 `/dev/disk` 下的目录，尝试列出其中的“软连接”，这样的设计有什么好处?

```
camellia@LAPTOP-Camellia:/dev/disk$ ls
by-diskseq  by-id  by-path  by-uuid
camellia@LAPTOP-Camellia:/dev/disk$ ls -lR /dev/disk/
/dev/disk/:
total 0
drwxr-xr-x 2 root root 260 Jun  3 08:50 by-diskseq
drwxr-xr-x 2 root root 160 Jun  3 08:50 by-id
drwxr-xr-x 2 root root 100 Jun  3 08:50 by-path
drwxr-xr-x 2 root root  80 Jun  3 08:50 by-uuid

/dev/disk/by-diskseq:
total 0
lrwxrwxrwx 1 root root 11 Jun  3 08:50 17 -> ../../loop0
lrwxrwxrwx 1 root root 11 Jun  3 08:50 18 -> ../../loop1
lrwxrwxrwx 1 root root 11 Jun  3 08:50 19 -> ../../loop2
lrwxrwxrwx 1 root root 11 Jun  3 08:50 20 -> ../../loop3
lrwxrwxrwx 1 root root 11 Jun  3 08:50 21 -> ../../loop4
lrwxrwxrwx 1 root root 11 Jun  3 08:50 22 -> ../../loop5
lrwxrwxrwx 1 root root 11 Jun  3 08:50 23 -> ../../loop6
lrwxrwxrwx 1 root root 11 Jun  3 08:50 24 -> ../../loop7
lrwxrwxrwx 1 root root  9 Jun  3 08:50 25 -> ../../sda
lrwxrwxrwx 1 root root  9 Jun  3 08:50 26 -> ../../sdb
lrwxrwxrwx 1 root root  9 Jun  3 08:50 27 -> ../../sdc

/dev/disk/by-id:
total 0
lrwxrwxrwx 1 root root 9 Jun  3 08:50 scsi-360022480257e0245383e06777f2d52eb -> ../../sdb
lrwxrwxrwx 1 root root 9 Jun  3 08:50 scsi-36002248077a1a24aa8ec4c6cc71a4aff -> ../../sdc
lrwxrwxrwx 1 root root 9 Jun  3 08:50 scsi-36002248079f9f66f426ea82fb0957801 -> ../../sda
lrwxrwxrwx 1 root root 9 Jun  3 08:50 wwn-0x60022480257e0245383e06777f2d52eb -> ../../sdb
lrwxrwxrwx 1 root root 9 Jun  3 08:50 wwn-0x6002248077a1a24aa8ec4c6cc71a4aff -> ../../sdc
lrwxrwxrwx 1 root root 9 Jun  3 08:50 wwn-0x6002248079f9f66f426ea82fb0957801 -> ../../sda

/dev/disk/by-path:
total 0
lrwxrwxrwx 1 root root 9 Jun  3 08:50 acpi-MSFT1000:00-scsi-0:0:0:0 -> ../../sda
lrwxrwxrwx 1 root root 9 Jun  3 08:50 acpi-MSFT1000:00-scsi-0:0:0:1 -> ../../sdb
lrwxrwxrwx 1 root root 9 Jun  3 08:50 acpi-MSFT1000:00-scsi-0:0:0:2 -> ../../sdc

/dev/disk/by-uuid:
```

`/dev/disk` 下软连接设计的好处：

- 持久化设备命名：确保系统启动时能稳定识别和挂载磁盘，避免因 /dev/sdX 名称变化而导致的启动问题和数据访问错误。
- 增强可读性/可识别性：通过硬件 ID、物理路径或文件系统 UUID 等方式，让设备名称更具描述性，方便用户和管理员识别和管理特定设备。
- 提高脚本/自动化鲁棒性：自动化脚本和系统配置文件可以依赖稳定的软连接名称，减少因设备名称变化而导致的故障和维护成本。

6. 尝试运行 `lsblk` 命令，根据你的输出，解释其中的内容。

```
camellia@LAPTOP-Camellia:/dev/disk$ lsblk
NAME
    MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda   8:0    0 388.4M  1 disk
sdb   8:16   0     2G  0 disk [SWAP]
sdc   8:32   0     1T  0 disk /mnt/wslg/distro
```

这三个设备都是虚拟磁盘，由 `WSL2` 环境提供。它们没有传统意义上的分区（例如 `sda1`，`sdb1`），而是被直接作为完整的磁盘设备使用，分别用于不同的目的：一个只读的辅助盘，一个交换空间，以及一个承载主要发行版数据的大容量虚拟磁盘。

3. `tmpfs`

```
camellia@LAPTOP-Camellia:/dev/shm$ ls -l /dev/shm
total 0
camellia@LAPTOP-Camellia:/dev/shm$ ls -l /run
total 12
drwxr-xr-x  2 root root  240 Jun  3 09:13 WSL
-rw-------  1 root root    0 Jun  3 08:50 agetty.reload
srw-------  1 root root    0 Jun  3 08:50 apport.socket
drwxr-xr-x  2 root root  140 Jun  3 08:50 cloud-init
drwxr-xr-x  2 root root   80 Jun  3 08:50 console-setup
drwxr-xr-x  2 root root   40 Jun  3 09:05 credentials
-rw-r--r--  1 root root    4 Jun  3 08:50 crond.pid
----------  1 root root    0 Jun  3 08:50 crond.reboot
drwxr-xr-x  3 root root   80 Jun  3 08:50 dbus
prw-------  1 root root    0 Jun  3 08:50 initctl
drwxrwxrwt  3 root root   60 Jun  3 08:50 lock
drwxr-xr-x  3 root root   60 Jun  3 08:50 log
-rw-r--r--  1 root root  989 Jun  3 08:50 motd.dynamic.new
drwxr-xr-x  2 root root   40 Jun  3 08:50 mount
drwxr-xr-x  2 root root   40 Jun  3 08:50 qemu
drwxr-xr-x  2 root root   40 Jun  3 08:50 sendsigs.omit.d
drwxr-xr-x  2 root root   40 Jun  3 08:50 setrans
drwxrwxrwt  2 root root   40 Jun  3 08:50 shm
srw-rw-rw-  1 root root    0 Jun  3 08:50 snapd-snap.socket
srw-rw-rw-  1 root root    0 Jun  3 08:50 snapd.socket
drwx--x--x  2 root root   40 Jun  3 08:50 sudo
drwxr-xr-x 23 root root  600 Jun  3 10:42 systemd
drwxr-xr-x  2 root root   60 Jun  3 10:43 ubuntu-advantage
drwxr-xr-x  6 root root  140 Jun  3 10:21 udev
-rw-r-----  1 root root    0 Jun  3 09:05 unattended-upgrades.lock
drwxr-xr-x  3 root root   60 Jun  3 08:50 user
-rw-rw-r--  1 root utmp 1920 Jun  3 08:50 utmp
drwxr-xr-x  2 root root   60 Jun  3 08:50 uuidd
camellia@LAPTOP-Camellia:/dev/shm$ ls -l /var/run
lrwxrwxrwx 1 root root 4 Feb 15 16:09 /var/run -> /run
```

`tmpfs` 是一个基于内存的临时文件系统，通常挂载在 `/tmp` 目录下。它将文件存储在虚拟内存中，而不是物理磁盘上，因此具有高速读写的特点。在 Linux 系统中，`tmpfs` 常用于存储临时文件、进程间通信文件和系统运行时状态文件。

通过探索 `/tmp` 目录，我们可以发现以下几类重要的文件：

3.1. 1. 扩展名为 `.pid` 的文件

找到的文件：`crond.pid`

作用与利用：

- `crond.pid` 文件中记录了 `cron` 守护进程的进程 ID
- 当系统启动 `cron` 服务时，它会检查这个文件
- 如果文件存在且其中的 `PID` 对应一个正在运行的 `cron` 进程，则新的 `cron` 实例会退出
- 这样确保系统中只有一个 `cron` 进程在运行，避免重复调度任务或资源冲突

3.2. 2. 扩展名为 `.lock` 的文件

找到的文件：`unattended-upgrades.lock`

注意：列表中的 `Lock` 是一个目录（`drwxrwxrwt`），而不是文件锁。真正的文件锁是 `unattended-upgrades.lock`。

作用与利用：

- `unattended-upgrades.lock` 文件用于确保 `unattended-upgrades` 服务（负责自动下载和安装安全更新）在同一时间只有一个实例在运行
- 当服务启动时，它会尝试创建或获取这个锁文件上的排他锁
- 如果成功，它会继续执行升级操作
- 如果失败（因为另一个实例已经持有锁），它会等待或退出
- 这样避免多个升级进程同时运行，可能导致文件冲突或升级失败

3.3. 3. 扩展名为 `.sock` 或 `.socket` 的文件

找到的文件：`apport.socket`、`snapd-snap.socket`、`snapd.socket`
作用与利用：

3.3.1. `apport.socket`

- `apport` 是 Ubuntu 系统中的一个崩溃报告工具
- 这个套接字文件允许其他进程（如崩溃的应用程序）通过它与 `apport` 守护进程通信
- 将崩溃信息发送给它进行处理

3.3.2. `snapd-snap.socket` 和 `snapd.socket`

- `snapd` 是 Ubuntu 及其它 Linux 发行版中用于管理 Snap 软件包的守护进程
- 这些套接字文件允许 `snap` 命令行工具以及其他需要与 `snapd` 交互的应用程序通过它们与 `snapd` 守护进程进行通信
- 例如安装、更新或删除 Snap 包

3.3.3. Unix 域套接字的工作原理

- 这些 `.socket` 文件作为 Unix 域套接字，提供了本地进程间通信的通道
- 服务器进程（如 `apport` 或 `snapd` 守护进程）创建并监听这些套接字文件
- 客户端进程（如崩溃的程序或 `snap` 命令）则连接到这些套接字
- 通过它们发送请求和接收响应，实现高效且安全的数据交换

六. 思考题

1. 第 1 题：为什么在 `pkg/storage/lib.rs` 中声明了 `#![cfg_attr(not(test), no_std)]`，它有什么作用？哪些因素导致了 kernel 中进行单元测试是一个相对困难的事情？

1.1. `#![cfg_attr(not(test), no_std)]` 的作用

`#![cfg_attr(not(test), no_std)]` 是一个条件编译属性，其作用如下：

- **非测试环境**（`not(test)`）：启用 `no_std`，表示不使用标准库，只使用核心库（`core`）和分配器库（`alloc`）
- **测试环境**：允许使用标准库，这样可以使用 `std::println!` 等测试工具和完整的运行时环境

1.2. kernel 中进行单元测试的困难因素

困难因素	具体说明
运行环境限制	内核代码运行在特权模式下，没有用户态的运行时环境
依赖硬件	许多内核功能直接依赖硬件，难以在测试环境中模拟
内存管理	内核需要自己管理内存，测试时需要特殊的内存分配器
中断和并发	内核代码涉及中断处理和并发控制，测试时难以重现这些条件
系统调用接口	内核与用户态的接口复杂，需要特殊的测试框架

2. 第 2 题：留意 `MbrTable` 的类型声明，为什么需要泛型参数 `T` 满足 `BlockDevice<B> + Clone`？为什么需要 `PhantomData<B>` 作为 `MbrTable` 的成员？在 `PartitionTable trait` 中，为什么需要 `Self: Sized` 约束？

2.1. T 满足 `BlockDevice<B> + Clone` 的原因

- **`BlockDevice<B>`**：`MbrTable` 需要读取底层设备的数据来解析分区表，所以 `T` 必须实现块设备接口
- **`Clone`**：在 `partitions()` 方法中需要为每个分区创建设备的副本（`self.inner.clone()`），所以 `T` 必须可克隆

2.2. `PhantomData<B>` 的作用

- `MbrTable` 结构体本身不直接存储 `B` 类型的数据，但在类型系统中需要表明它与 `B` 类型相关联
- `PhantomData` 是零大小类型，不占用内存空间，但让编译器知道这个结构体在逻辑上“拥有” `B` 类型
- 确保类型参数 `B` 被正确使用，避免编译器警告未使用的类型参数

2.3. `Self: Sized` 约束的原因

- `parse` 方法返回 `Self`，需要在编译时知道返回类型的大小
- 如果没有 `Sized` 约束，`Self` 可能是动态大小类型（DST），无法在栈上分配
- 这确保了 trait 的实现类型都是编译时已知大小的类型

3. 第 3 题：`AtaDrive` 为了实现 `MbrTable`，如何保证了自身可以实现 `Clone`？对于分离 `AtaBus` 和 `AtaDrive` 的实现，你认为这样的设计有什么好处?

3.1. `AtaDrive` 的 `Clone` 实现

`AtaDrive` 通过在结构体上添加 `#[derive(Clone)]` 来自动实现 Clone trait。这是可行的，因为:

- `bus` 和 `drive` 是基本类型（`u8`），天然可克隆
- `blocks` 是 `u32`，可克隆
- `model` 和 `serial` 是 `Box<str>`，`Box` 实现了 `Clone`（会进行深拷贝）

3.2. 分离 AtaBus 和 AtaDrive 设计的好处

设计优势	具体说明
职责分离	· `AtaBus`：负责底层硬件操作（端口读写、命令发送） · `AtaDrive`：负责高层抽象（设备信息、块设备接口）
资源管理	· 一个 `AtaBus` 可以管理多个驱动器（主从设备） · 通过 `Mutex` 保护 `AtaBus`，确保并发安全
可扩展性	· 容易添加新的设备类型（如 SATA、NVMe） · 可以为不同总线类型提供统一接口
测试友好	· 可以独立测试总线操作和驱动器抽象 · 便于 mock 底层硬件操作

4. 第 4 题：结合本次实验中的抽象和代码框架，简单解释和讨论如下写法的异同

4.1. 函数声明的异同

```rust
fn f<T: Foo>(f: T) → usize          // 静态分发，单态化
fn f(f: impl Foo) → usize           // 静态分发，语法糖
fn f(f: &dyn Foo) → usize           // 动态分发，运行时多态
```

写法	分发方式	编译时行为	性能特点
`fn f<T: Foo>(f: T)`	静态分发	为每个具体类型生成专门的函数版本	零成本抽象，性能最优
`fn f(f: impl Foo)`	静态分发	语法糖，本质等同于第一种	零成本抽象，代码更简洁
`fn f(f: &dyn Foo)`	动态分发	通过虚函数表实现运行时多态	有间接调用开销，但更灵活

在本实验中：`BlockDevice<B>` trait 主要使用静态分发（如 `AtaDrive` 实现），确保了性能最优。

4.2. 结构体声明的异同

```rust
struct S<T: Foo> { f: T }           // 静态类型，编译时确定
struct S { f: Box<dyn Foo> }        // 动态类型，运行时确定
```

写法	类型确定时机	内存分配	特点
`struct S<T: Foo> { f: T }`	编译时	栈分配	零成本抽象，类型安全
`struct S { f: Box<dyn Foo> }`	运行时	堆分配	灵活但有性能开销

在本实验中：`MbrTable<T, B>` 使用第一种方式，确保了类型安全和性能。

5. 第 5 题：文件系统硬链接和软链接的区别是什么？`Windows` 中的“快捷方式”和 `Linux` 中的软链接有什么异同？

5.1. 硬链接 vs 软链接

特性	硬链接	软链接
本质	指向同一个 inode	指向另一个文件路径
删除原文件	不影响硬链接	软链接变成悬空链接
跨文件系统	不支持	支持
目录链接	通常不支持	支持
存储内容	与原文件共享数据块	存储目标文件路径

5.2. Windows 快捷方式 vs Linux 软链接

相同点:

- 都是指向其他文件/程序的引用
- 删除原文件后都会失效
- 都可以跨文件系统

不同点:

- **实现层面**:
  - Windows 快捷方式：应用层实现，是特殊的 .lnk 文件
  - Linux 软链接：文件系统层面实现，是特殊的文件类型
- **透明性**:
  - Windows 快捷方式：应用程序需要特殊处理
  - Linux 软链接：对大多数程序透明，自动解析
- **功能**:
  - Windows 快捷方式：可以包含参数、工作目录等额外信息
  - Linux 软链接：只是简单的路径重定向

6. 第 6 题: 日志文件系统（如 `NTFS`）与传统的非日志文件系统（如 `FAT`）在设计和实现上有哪些不同? 在系统异常崩溃后, 它的恢复机制、恢复速度有什么区别?

6.1. 设计和实现差异

特性	日志文件系统（NTFS）	非日志文件系统（FAT）
写入机制	写前日志：所有元数据修改先写入日志，再写入实际位置	直接写入: 直接修改文件分配表和目录项
事务性	文件操作被组织成原子事务	操作不是原子的, 可能出现不一致状态
数据结构	使用 B+ 树等复杂数据结构	使用简单的链表结构管理文件
元数据保护	重点保护文件系统元数据的一致性	无特殊保护机制

6.2. 恢复机制差异

恢复方面	NTFS 恢复机制	FAT 恢复机制
恢复方式	1. 重放日志：系统启动时重放未完成的事务 2. 回滚操作：撤销未提交的事务	1. 文件系统检查：需要扫描整个文件系统（如 chkdsk） 2. 手动修复：需要用户干预或专门工具

	3. 自动修复：大多数不一致可以自动修复	3. 数据丢失风险：可能丢失未完全写入的数据
恢复速度	通常几秒到几分钟，与日志大小相关	可能需要几分钟到几小时，与磁盘大小相关

6.3. 综合比较

比较维度	NTFS	FAT
可靠性	高	低
恢复速度	快	慢
复杂性	高	低
存储开销	高	低

实验总结：在本实验中实现的 FAT16 属于传统非日志文件系统，结构简单但缺乏崩溃保护机制。

七. 加分项：从文件系统加载用户程序

> 🚀 **挑战目标**
>
> 将用户程序加载功能从 bootloader 迁移到操作系统内核，实现从文件系统直接加载和执行 ELF 程序的能力。

1. 📋 功能需求分析

核心功能	实现要求
进程创建重构	修改 `spawn` 函数，支持文件路径参数
文件系统集成	使用文件系统 API 读取 ELF 文件内容
ELF 解析	将文件内容传递给 `elf_spawn` 进行解析
系统调用更新	修改 `Spawn` 系统调用参数格式
Shell 增强	为 Shell 添加从磁盘启动程序的能力
向后兼容	保持对 bootloader 应用的兼容性

2. 🎯 技术架构设计

设计思路：采用“文件系统优先，bootloader 回退”的策略，确保系统的稳定性和兼容性。

1. **优先级机制**：首先尝试从文件系统加载，失败时回退到 bootloader
2. **内存管理**：使用 `Box::leak` 确保 ELF 数据的生命周期
3. **错误处理**：完善的错误检查和恢复机制

3. 🔧 详细实现步骤

3.1. 步骤 1：重构 spawn 函数

目标：修改 `pkg/kernel/src/proc/mod.rs` 中的 `spawn` 函数，使其支持从文件路径加载 ELF 程序。

核心实现逻辑：

```rust
pub fn spawn(path: &str) -> Option<ProcessId> {
    use alloc::boxed::Box;

    // 🔍 步骤1：尝试从文件系统加载
    if let Ok(mut file_handle) =
crate::drivers::filesystem::get_rootfs().open_file(path) {
        // 📏 获取文件大小并分配缓冲区
        let file_size = file_handle.meta.len;
        let mut buffer = alloc::vec![0u8; file_size];

        // 📖 读取整个文件内容
        if let Ok(bytes_read) = file_handle.read(&mut buffer) {
            if bytes_read == file_size {
                // 🔒 确保 ELF 数据生命周期
                let buffer = Box::leak(buffer.into_boxed_slice());

                // 🧩 解析 ELF 文件格式
                if let Ok(elf) = xmas_elf::ElfFile::new(buffer) {
                    let process_name =
path.split('/').last().unwrap_or(path).to_string();
                    let entry_point = elf.header.pt2.entry_point();

                    info!("Loading ELF from path '{}' as process '{}', entry
point: {:#x}",
                          path, process_name, entry_point);

                    if entry_point != 0 {
                        return elf_spawn(process_name, &elf);
                    }
                }
            }
        }
    }

    // 🔄 步骤2：回退到 bootloader 应用（向后兼容）
    let app = x86_64::instructions::interrupts::without_interrupts(|| {
        let app_list = get_process_manager().app_list()?;
        app_list.iter().find(|&app| app.name.eq(path))
    })?;

    info!("Loading ELF from bootloader app '{}', entry point: {:#x}",
          path, app.elf.header.pt2.entry_point());

    elf_spawn(path.to_string(), &app.elf)
}
```

关键技术要点：

技术点	实现细节
文件系统集成	使用 `get_rootfs().open_file()` 打开文件
内存管理	使用 `Box::leak()` 确保 ELF 数据在进程生命周期内有效
错误处理	多层级错误检查，确保加载过程的健壮性
向后兼容	文件系统加载失败时自动回退到 bootloader 应用
进程命名	从文件路径提取文件名作为进程名称

3.2. 步骤 2：更新系统调用服务

> **目标**：修改 `pkg/kernel/src/interrupt/syscall/service.rs` 中的 `spawn_process` 函数，支持文件路径参数。

系统调用接口实现：

```rust
pub fn spawn_process(args: &SyscallArgs) -> usize {
    // 🔍 参数验证：获取文件路径或应用名称
    let ptr = args.arg0 as *const u8;
    let len = args.arg1;

    if ptr.is_null() || len == 0 {
        return 0;  // 参数无效，返回失败
    }

    // 🔄 字符串转换：将原始指针转换为字符串
    let path = unsafe {
        let slice = core::slice::from_raw_parts(ptr, len);
        match core::str::from_utf8(slice) {
            Ok(s) => s,
            Err(_) => return 0,  // UTF-8 解码失败
        }
    };

    // 🚀 进程创建：调用重构后的 spawn 函数
    match spawn(path) {
        Some(pid) => pid.0 as usize,  // 返回进程 ID
        None => 0,                    // 创建失败
    }
}
```

改进要点：

- ✅ 保持原有系统调用接口不变

- ✅ 增强参数验证和错误处理
- ✅ 支持文件路径和应用名称两种格式

3.3. 步骤 3：优化初始化流程

目标：修改 `pkg/kernel/src/main.rs` 中的 `spawn_init` 函数，实现智能的 Shell 程序加载。

初始化逻辑实现：

```
pub fn spawn_init() -> proc::ProcessId {
    proc::list_app();

    // 🎯 智能加载策略：文件系统优先，bootloader 回退
    proc::spawn("/shell")            // 尝试从文件系统加载
        .or_else(|| proc::spawn("shell"))  // 回退到 bootloader 应用
        .unwrap()                    // 确保 Shell 成功启动
}
```

设计优势：

特性	优势说明
智能回退	文件系统加载失败时自动使用 bootloader 中的 Shell
零配置	无需修改现有配置，自动适应不同环境
高可用性	确保系统在任何情况下都能启动 Shell
渐进升级	支持从 bootloader 向文件系统的平滑迁移

4. 🎉 实验成果展示

4.1. 功能验证结果

测试结果：成功实现从文件系统加载用户程序的功能，系统能够：

1. ✅ 从文件系统直接加载 ELF 程序
2. ✅ 保持与 bootloader 应用的完全兼容
3. ✅ 提供健壮的错误处理和恢复机制
4. ✅ 支持 Shell 从磁盘启动用户程序

```
[INFO ] ATA drive test successful!
[INFO ] Testing MBR read...
[INFO ] MBR read successful!
[INFO ] Valid MBR signature found!
[INFO ] First partition info (using MbrPartition):
[INFO ]   Active: true
[INFO ]   Type: 0x06
[INFO ]   Start LBA: 63
[INFO ]   Size (sectors): 1032129
[INFO ]   Size (MB): 503
[INFO ]   CHS Begin: C0/H1/S1
[INFO ]   CHS End: C1023/H15/S63
[INFO ] Opening disk device...
[INFO ] Identifying drive 0
[INFO ] Drive QEMU HARDDISK QM00001 (504 MiB) opened
[INFO ] Mounting filesystem...
[INFO ] Initialized Filesystem.
[INFO ] YatSenOS initialized.
[+] App list: factorial  hello  shell  counter  dinner  mq  fork_test
[INFO ] Loading ELF from bootloader app 'shell', entry point: 0x111100002680
[INFO ] ELF header bytes: [7f, 45, 4c, 46, 02, 01, 01, 00, 00, 00, 00, 00, 00, 00, 00, 00]
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> run APP/HELLO
正在运行程序: APP/HELLO
[INFO ] Loading ELF from path 'APP/HELLO' as process 'HELLO', entry point: 0x111100001050
[INFO ] ELF header bytes: [7f, 45, 4c, 46, 02, 01, 01, 00, 00, 00, 00, 00, 00, 00, 00, 00]
进程ID: 3
Hello, world!!!
程序 'APP/HELLO' 已退出，返回值: 233
ysos> run hello
正在运行程序: hello
[INFO ] Loading ELF from bootloader app 'hello', entry point: 0x111100001050
[INFO ] ELF header bytes: [7f, 45, 4c, 46, 02, 01, 01, 00, 00, 00, 00, 00, 00, 00, 00, 00]
进程ID: 4
Hello, world!!!
程序 'hello' 已退出，返回值: 233
ysos>
```

4.2. 技术创新点

创新特性	技术实现
双重加载机制	文件系统优先 + bootloader 回退的混合策略
内存生命周期管理	使用 `Box::leak` 确保 ELF 数据持久性
透明兼容性	对用户和应用程序完全透明的升级路径
错误恢复能力	多层级错误检查和自动恢复机制

5. 📊 实现效果总结

🏆 加分项完成情况

本加分项成功实现了从 bootloader 到文件系统的用户程序加载功能迁移，具备以下核心能力：

- ✅ 重构 `spawn` 函数支持文件路径参数 **完成**
- ✅ 集成文件系统 API 进行 ELF 文件读取 **完成**
- ✅ 实现 ELF 内容解析和进程创建 **完成**

✅ 更新系统调用接口支持路径参数 **完成**
✅ 增强 Shell 的程序启动能力 **完成**
✅ 保持向后兼容性 **完成**

5.1. 技术价值与意义

价值维度	**具体体现**
系统架构优化	· 减少对 bootloader 的依赖 · 提高系统的模块化程度 · 增强运行时的灵活性
用户体验提升	· 支持动态加载程序 · 简化程序部署流程 · 提供更好的错误处理
技术创新	· 双重加载机制设计 · 智能回退策略 · 内存生命周期优化
工程实践	· 向后兼容性保证 · 渐进式系统升级 · 健壮的错误恢复

实验总结: 通过本加分项的实现，操作系统获得了完整的文件系统程序加载能力，为后续的系统功能扩展奠定了坚实的基础。这一改进不仅提升了系统的实用性，也展现了现代操作系统设计中模块化和可扩展性的重要原则。