



课程实验报告

0x06 硬盘驱动与文件系统

课程名称	操作系统原理实验
专业名称	计算机科学与技术
学生姓名	陈政宇
学生学号	23336003
实验地点	东校园-实验中心大楼 B201
实验成绩	
实验日期	2025 年 6 月 23 日

目录

1 实验目的	3
2 代码实现	3
2.1 MBR 分区表	3
2.2 磁盘驱动	5
2.2.1 命令发送	5
2.2.2 磁盘识别	6
2.2.3 读写数据	8
2.3 FAT16 文件系统	10
2.3.1 BPB	10
2.3.2 DirEntry	14
2.3.3 Fat16Impl	18
2.4 接入操作系统	29
2.4.1 ls 命令	29
2.4.2 读取文件	31
3 思考题	33
3.1 q1	33
3.2 q2	34
3.3 q3	35
3.4 q4	35
3.5 q5	36
3.6 q6	37

1 实验目的

1. 了解文件系统的概念、作用。
2. 实现块设备、磁盘、分区、文件系统的抽象。
3. 了解 ATA 硬盘的工作原理、实现基本的读写驱动。
4. 实现 FAT16 文件系统的读取和只读文件访问。

2 代码实现

2.1 MBR 分区表

☰ 补全分区表结构体定义

对于分区表项，需要你在 `partition/mbr/entry.rs` 中，补全对应的结构体定义。

笔者为大家提供了一个便捷的宏：`define_field`，它的定义可以在 `common/macros.rs` 中找到，并且为各位补有文档注释，以做说明如何使用。

同时，这里以 `MbrPartition` 的定义为例子，再做一些解释：

```
1 impl MbrPartition {  
2     // ...  
3     define_field!(u8, 0x00, status);  
4     // ...  
5 }
```

[Python](#)

这里的 `define_field!` 宏，接受三个参数，分别是字段的类型、字段的偏移量和字段的名称。它会自动为你生成一个 `status()` 的函数，用于获取字段的值。

字段的类型可以是 `u8`、`u16`、`u32`，分别对应 1、2、4 字节的整数；同时还有 `[u8; n]` 的类型，用于表示固定长度的字节数组，同时也会提供一个对应的从 `&[u8]` 转换为 `&str` 的函数。

你可以在下方的 `Debug trait` 的实现中看到这些函数的使用，你需要补全其中展示的全部函数，并尝试通过文件附带的单元测试。

对于 `0x01-0x03` 和 `0x05-0x07` 两组三字节的内容分别表示了开始和结束的 CHS 地址，包含三组内容：磁头号、扇区号和柱面号，分别占用 8、6 和 10 比特，因此无法使用 `define_field` 进行简单定义，需要你自行实现 `head`、`sector` 和 `cylinder` 所对应的函数，对 `data` 进行解析。

```

1  impl MbrPartition {
2      /// Parse a partition entry from the given data.
3      pub fn parse(data: &[u8; 16]) -> MbrPartition {
4          MbrPartition {
5              data: data.to_owned(),
6          }
7      }
8      // FIXME: define other fields in the MbrPartition
9      //      - use `define_field!` macro
10     //      - ensure you can pass the tests
11     //      - you may change the field names if you want
12     //
13     // NOTE: some fields are not aligned with byte.
14     //      define your functions to extract values:
15     //
16     //      0x02 - 0x03 begin sector & begin cylinder
17     //      0x06 - 0x07 end sector & end cylinder
18
19     // an example of how to define a field
20     // move your mouse on the `define_field!` to see the docs
21     define_field!(u8, 0x00, status);
22     define_field!(u8, 0x01, begin_head);
23     define_field!(u8, 0x04, partition_type);
24     define_field!(u8, 0x05, end_head);
25     define_field!(u32, 0x08, begin_lba);
26     define_field!(u32, 0x0C, total_lba);
27     pub fn begin_sector(&self) -> u16 {
28         (self.data[0x02] & 0x3f) as u16
29     }
30     pub fn begin_cylinder(&self) -> u16 {
31         ((self.data[0x02] as u16 & 0xc0) << 2) | (self.data[0x03] as u16)
32     }
33     pub fn end_sector(&self) -> u16 {
34         (self.data[0x06] & 0x3f) as u16
35     }
36     pub fn end_cylinder(&self) -> u16 {
37         ((self.data[0x06] & 0xc0) as u16) << 2 | self.data[0x07] as u16
38     }
39     pub fn is_active(&self) -> bool {
40         self.status() == 0x80
41     }
42 }

```

根据文档定义各个字段即可。单元测试结果如下（证明所有字段均正确定义）：

```
[ INFO] Drive QEMU HARDDISK QM00001 (120.00009 GiB) opened
ATA Drive opened successfully!
Drive info: AtaDrive {
    bus: 0,
    drive: 0,
    blocks: 251658432,
    model: "QEMU HARDDISK",
    serial: "QM00001",
}
Testing drive operations...
Drive has 251658432 blocks
Successfully read block 0
First 16 bytes: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ INFO] Identifying drive 1
[ WARN] Drive 0@1 is not a PATA drive
No secondary ATA Drive found
```

2.2 磁盘驱动

2.2.1 命令发送

☰ 补全 ATA 命令发送

在实现了 MBR 分区表解析后，笔者预计你对现有的代码结构已经有了一定的认知。现在，我们来实现 ATA 磁盘驱动，使得内核能够通过它访问“真实”的虚拟磁盘，并读取并解析其中的数据。

```
1  /// Writes the given command 🔒 Rust
2  ///
3  /// reference: https://wiki.osdev.org/ATA\_PIO\_Mode#28\_bit\_PIO
4  fn write_command(&mut self, drive: u8, block: u32, cmd: AtaCommand) ->
    storage::FsResult {
5      let bytes = block.to_le_bytes(); // a trick to convert u32 to [u8; 4]
6      unsafe {
7          // just 1 sector for current implementation
8          self.sector_count.write(1);
9
10         // DONE: store the LBA28 address into four 8-bit registers
11         //      - read the documentation for more information
12         //      - enable LBA28 mode by setting the drive register
```

```
13     self.lba_low.write(bytes[0]); // LBA bits 0-7
14     self.lba_mid.write(bytes[1]); // LBA bits 8-15
15     self.lba_high.write(bytes[2]); // LBA bits 16-23
16
17     // Enable LBA28 mode by setting the drive register
18     // bit 6 = 1 (LBA mode), bit 4 = drive number, bits 0-3 = LBA
19     // bits 24-27
20     self.drive.write(0xE0 | ((drive & 1) << 4) | (bytes[3] & 0x0F));
21
22     // DONE: write the command register (cmd as u8)
23     self.command.write(cmd as u8);
24 }
25
26 if self.status().is_empty() {
27     // unknown drive
28     return Err(storage::DeviceError::UnknownDevice.into());
29 }
30
31 // DONE: poll for the status to be not BUSY
32 self.poll(AtaStatus::BUSY, false);
33
34 if self.is_error() {
35     warn!("ATA error: {:?} command error", cmd);
36     self.debug();
37     return Err(storage::DeviceError::InvalidOperation.into());
38 }
39
40 // DONE: poll for the status to be not BUSY and DATA_REQUEST_READY
41 self.poll(AtaStatus::BUSY, false);
42 self.poll(AtaStatus::DATA_REQUEST_READY, true);
43 Ok(())
```

在上面的代码中，我们完成了 ATA 命令的发送。主要步骤如下：

1. 将当前块的 LBA 偏移分别存入四个寄存器中
2. 同时使用 drive 寄存器选择磁盘
3. 发送命令
4. 等待设备就绪，判断是否出错
5. 等待数据请求就绪

2.2.2 磁盘识别

☞ 补全 ATA 磁盘识别

在完成命令发送部分后，尝试补全 `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。

```

1  /// Identifies the drive at the given `drive` number (0 or 1). 📌 Rust
2  ///
3  /// reference: https://wiki.osdev.org/ATA\_PIO\_Mode#IDENTIFY\_command
4  pub(super) fn identify_drive(&mut self, drive: u8) ->
5      storage::FsResult<AtaDeviceType> {
6
7      // DONE: use `AtaCommand::IdentifyDevice` to identify the drive
8      //      - call `write_command` with `drive` and `0` as the block
9      //      - if the status is empty, return `AtaDeviceType::None`
10     //      - else return `DeviceError::Unknown` as `FsError`
11     if let Err(_) = self.write_command(drive, 0,
12         AtaCommand::IdentifyDevice) {
13         if self.status().is_empty() {
14             return Ok(AtaDeviceType::None);
15         } else {
16             return Err(storage::DeviceError::UnknownDevice.into());
17         }
18     }
19
20     // NONE: poll for the status to be not BUSY
21     self.poll(AtaStatus::BUSY, false);
22
23     Ok(match (self.cylinder_low(), self.cylinder_high()) {
24         (0x00, 0x00) => AtaDeviceType::Pata(Box::new([0u16; 256].map(|_|
25             self.read_data()))),
26         // ignore the data as we don't support following types
27         (0x14, 0xEB) => AtaDeviceType::PataPi,
28         (0x3C, 0xC3) => AtaDeviceType::Sata,
29         (0x69, 0x96) => AtaDeviceType::SataPi,
30         _ => AtaDeviceType::None,
31     })

```

ATA 磁盘的识别主要步骤如下：

1. 调用 `write_command` 函数发送 ATA 命令以识别驱动。
2. 检查状态寄存器，如果为空，则返回 `AtaDeviceType::None`。
3. 如果状态寄存器不为空，则根据 `cylinder_low` 和 `cylinder_high` 的值判断驱动类型，并返回相应的 `AtaDeviceType`。

2.2.3 读写数据

☰ 补全 ATA 数据读写

在编写 `write_command` 函数时，你或许会注意到 `sector_count` 寄存器直接被设置为了 1。

虽然 ATA 驱动支持一次读取多个扇区，但从抽象和实现角度，本实验还是采取了每次写指令只读一块的方式。由于 ATA 本身的速度就很慢，并且作为实验实现，这样能够更加灵活和便捷。

经过上述函数的统一，读写磁盘的操作变得十分简单：在使用 `write_command` 指明需要进行的操作后，从 `data` 寄存器中每次 16 位地与 `buf` 进行数据交互。

在为 Bus 实现了 `read_pio` 和 `write_pio` 之后，你还需要在 `drivers/ata/mod.rs` 中补全块设备的实现。

`AtaDrive` 通过 `bus` 和 `drive` 字段存储了对应的磁盘信息，`BUSES` 的定义已经为大家定义完善，你需要借助这些内容，补全 `impl BlockDevice for AtaDrive` 中对应的 `FIXME` 的内容。

```

1  impl BlockDevice<Block512> for AtaDrive { Rust
2      fn block_count(&self) -> storage::FsResult<usize> {
3          // DONE: return the block count
4          Ok(self.blocks as usize)
5      }
6
7      fn read_block(&self, offset: usize, block: &mut Block512) ->
        storage::FsResult {
8          // DONE: read the block
9          //      - use `BUSES` and `self` to get bus
10         //      - use `read_pio` to get data
11         BUSES[self.bus as usize]
12             .lock()
13             .read_pio(self.drive, offset as u32, block.as_mut())
14     }

```



```
15
16     fn write_block(&self, offset: usize, block: &Block512) ->
        storage::FsResult {
17         // DONE: write the block
18         //     - use `BUSES` and `self` to get bus
19         //     - use `write_pio` to write data
20         BUSES[self.bus as usize]
21         .lock()
22         .write_pio(self.drive, offset as u32, block.as_ref())
23     }
24 }
```

在上面的代码中，实现了 AtaDrive 的块设备 trait。主要步骤如下：

1. 使用 BUSES 数组和 self 获取对应的总线。
2. 使用 read_pio 方法读取数据到 block 中。
3. 使用 write_pio 方法将 block 中的数据写入磁盘。

✓ 阶段成果

在完成命令发送，并按照注释补全 identify_drive 函数后，你可以自行修改相关函数，测试 AtaDrive 的 open 函数。

在操作系统初始化结束后，使用 AtaDrive::open(0, 0) 获取磁盘信息，为了确保通过编译，可以先忽略 filesystem.rs。

如果顺利，你应该会看到 Drive QEMU HARDDISK QM00001 (504 MiB) opened 的日志字样。

在为 AtaDrive 实现了块设备的 trait 后，尝试使用 MbrTable::parse(drive) 解析磁盘分区表。

如果顺利，你应当能够正确获取首个分区的相关信息，包括其类型、起始 LBA 和大小。尝试添加日志来记录这些信息，并补充在报告中。

测试结果如下图所示

```
[ INFO] Partition 1: Found non-empty partition
[ INFO]   Status: 0x80 (Active/Bootable)
[ INFO]   Type: 0x06 (FAT16)
[ INFO]   Start LBA: 63
[ INFO]   Total blocks: 1032129
[ INFO]   Size: 503 MB (516064 KB)
[ INFO]   CHS Start: Cylinder 0, Head 1, Sector 1
[ INFO]   CHS End: Cylinder 1023, Head 15, Sector 63
[ INFO] === MBR Summary ===
[ INFO] Total partitions found: 1
[ INFO] Active partitions: 1
✓ MBR parsed successfully
[ INFO] Creating partition object for active partition 1
[ INFO]   Partition range: LBA 63 to LBA 1032191
[ INFO] Created 1 active partition objects
✓ Found 1 partition(s)
   Partition 1: 251658432 blocks (122880 MB)
[ INFO] YatSenOS shutting down.
```

2.3 FAT16 文件系统

2.3.1 BPB

☰ 补全 BPB 结构体

在实现了 ATA 驱动后，我们可以开始实现 FAT16 文件系统。首先，我们需要定义 BPB（BIOS Parameter Block）结构体。

在文件 `fs/fat16/bpb.rs` 中，使用 `define_field!` 宏定义 BPB 的各个字段。你需要根据 FAT16 规范，补全所有字段的定义。

注意：BPB 的偏移量和大小需要根据 FAT16 规范进行设置。

```
1  impl Fat16Bpb { Rust
2      /// Attempt to parse a Boot Parameter Block from a 512 byte sector.
3      pub fn new(data: &[u8]) -> FsResult<Fat16Bpb> {
4          let data = data.try_into().unwrap();
5          let bpb = Fat16Bpb { data };
6
7          if bpb.data.len() != 512 || bpb.trail() != 0xAA55 {
8              return Err(FsError::InvalidOperation);
9          }
}
```

```

10
11     Ok(bpb)
12 }
13
14 pub fn total_sectors(&self) -> u32 {
15     if self.total_sectors_16() == 0 {
16         self.total_sectors_32()
17     } else {
18         self.total_sectors_16() as u32
19     }
20 }
21
22 // DONE: define all the fields in the BPB
23 //     - use `define_field!` macro
24 //     - ensure you can pass the tests
25 //     - you may change the field names if you want
26 // 跳转指令和 OEM 标识符 (偏移 0x03-0x0A, 8 字节)
27 define_field!(u8, 8, 0x03, oem_name);
28 // 每扇区字节数 (偏移 0x0B, 2 字节)
29 define_field!(u16, 0x0B, bytes_per_sector);
30 // 每簇扇区数 (偏移 0x0D, 1 字节)
31 define_field!(u8, 0x0D, sectors_per_cluster);
32 // 保留扇区数 (偏移 0x0E, 2 字节)
33 define_field!(u16, 0x0E, reserved_sector_count);
34 // FAT 表数量 (偏移 0x10, 1 字节)
35 define_field!(u8, 0x10, fat_count);
36 // 根目录条目数 (偏移 0x11, 2 字节)
37 define_field!(u16, 0x11, root_entries_count);
38 // 总扇区数 (16位) (偏移 0x13, 2 字节)
39 define_field!(u16, 0x13, total_sectors_16);
40 // 媒体描述符 (偏移 0x15, 1 字节)
41 define_field!(u8, 0x15, media_descriptor);
42 // 每 FAT 扇区数 (偏移 0x16, 2 字节)
43 define_field!(u16, 0x16, sectors_per_fat);
44 // 每磁道扇区数 (偏移 0x18, 2 字节)
45 define_field!(u16, 0x18, sectors_per_track);
46 // 磁头数 (偏移 0x1A, 2 字节)
47 define_field!(u16, 0x1A, track_count);
48 // 隐藏扇区数 (偏移 0x1C, 4 字节)
49 define_field!(u32, 0x1C, hidden_sectors);
50 // 总扇区数 (32位) (偏移 0x20, 4 字节)
51 define_field!(u32, 0x20, total_sectors_32);

```

```

52 // 驱动器号 (偏移 0x24, 1 字节)
53 define_field!(u8, 0x24, drive_number);
54 // 保留标志 (偏移 0x25, 1 字节)
55 define_field!(u8, 0x25, reserved_flags);
56 // 引导签名 (偏移 0x26, 1 字节)
57 define_field!(u8, 0x26, boot_signature);
58 // 卷 ID (偏移 0x27, 4 字节)
59 define_field!(u32, 0x27, volume_id);
60 // 卷标 (偏移 0x2B, 11 字节)
61 define_field!([u8; 11], 0x2B, volume_label);
62 // 文件系统标识符 (偏移 0x36, 8 字节)
63 define_field!([u8; 8], 0x36, system_identifier);
64 // 引导签名 (偏移 0x1FE, 2 字节)
65 define_field!(u16, 0x1FE, trail);
66 }

```

各个段的详细信息已经在注释中给出。

✓ 阶段成果

在补全 BPB 结构体后，你可以使用单元测试来验证其正确性。

```

1  #[test] 🔒 Rust
2  fn test_fat16_bpb_2() {
3      // Taken from QEMU VVFAT
4      const DATA: [u8; 64] = hex_literal::hex!(
5          "EB 3E 90 4D 53 57 49 4E 34 2E 31 00 02 10 01 00
6          02 00 02 00 00 F8 FC 00 3F 00 10 00 3F 00 00 00
7          C1 BF 0F 00 80 00 29 FD 1A BE FA 51 45 4D 55 20
8          56 56 46 41 54 20 46 41 54 31 36 20 20 20 00 00"
9      );
10
11      let mut bpb_data = Vec::with_capacity(512);
12      bpb_data.extend_from_slice(&DATA);
13      bpb_data.resize(510, 0u8);
14      bpb_data.extend_from_slice(&[0x55, 0xAA]);
15
16      let bpb = Fat16Bpb::new(&bpb_data).unwrap();
17
18      assert_eq!(bpb.oem_name(), b"MSWIN4.1");
19      assert_eq!(bpb.oem_name_str(), "MSWIN4.1");
20      assert_eq!(bpb.bytes_per_sector(), 512);

```

```
21  assert_eq!(bpb.sectors_per_cluster(), 16);
22  assert_eq!(bpb.reserved_sector_count(), 1);
23  assert_eq!(bpb.fat_count(), 2);
24  assert_eq!(bpb.root_entries_count(), 512);
25  assert_eq!(bpb.total_sectors_16(), 0);
26  assert_eq!(bpb.media_descriptor(), 0xf8);
27  assert_eq!(bpb.sectors_per_fat(), 0xfc);
28  assert_eq!(bpb.sectors_per_track(), 63);
29  assert_eq!(bpb.track_count(), 16);
30  assert_eq!(bpb.hidden_sectors(), 63);
31  assert_eq!(bpb.total_sectors_32(), 0xfbfcd1);
32  assert_eq!(bpb.drive_number(), 128);
33  assert_eq!(bpb.reserved_flags(), 0);
34  assert_eq!(bpb.boot_signature(), 0x29);
35  assert_eq!(bpb.volume_id(), 0xfab1afd);
36  assert_eq!(bpb.volume_label(), b"QEMU VVFAT ");
37  assert_eq!(bpb.volume_label_str(), "QEMU VVFAT ");
38  assert_eq!(bpb.system_identifier(), b"FAT16 ");
39  assert_eq!(bpb.system_identifier_str(), "FAT16 ");
40
41  assert_eq!(bpb.total_sectors(), 0xfbfcd1);
42
43  println!("{:?}", bpb);
44 }
```

测试结果如下图所示：

```

successes:

---- fs::fat16::bpb::tests::test_fat16_bpb_2 stdout ----
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,
}

successes:
  fs::fat16::bpb::tests::test_fat16_bpb_2

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

```

2.3.2 DirEntry

☰ 补全 DirEntry 结构体

在实现了 BPB 结构体后，我们需要定义 DirEntry 结构体。DirEntry 用于表示 FAT16 文件系统目录项。

在文件 fs/fat16/direntry.rs 中，使用 define_field! 宏定义 DirEntry 的各个字段。你需根据 FAT16 规范，补全所有字段的定义。

与先前的 MBR 和 BPB 不同，这里的 DirEntry 并不持有 data 数据作为自身的字段，而是通过 parse 函数直接解析 &[u8]，并返回一个 DirEntry 的实例。

而对于 ShortFileName 类型，你则需要实现从日常使用的文件名到磁盘数据的转化函数，具体来说，你需要实现 parse 函数，将一个 &str 类型的文件名 foo.bar（忽略大小写）转化为 ShortFileName 类型，并存储 FOO BAR

注意：DirEntry 的偏移量和大小需根据 FAT16 规范进行设置。

```

1  /// Parse a short file name from a string
2  pub fn parse(name: &str) -> FsResult<ShortFileName> {

```

 Rust

```

3      // DONE: implement the parse function
4      //      use `FilenameError` and into `FsError`
5      //      use different error types for following conditions:
6      //
7      //      - use 0x20 ' ' for right padding
8      //      - check if the filename is empty
9      //      - check if the name & ext are too long
10     //      - period `.` means the start of the file extension
11     //      - check if the period is misplaced (after 8 characters)
12     //      - check if the filename contains invalid characters:
13     //          [0x00..=0x1F, 0x20, 0x22, 0x2A, 0x2B, 0x2C, 0x2F, 0x3A,
14     //          0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, 0x7C]
15     // 检查文件名是否为空
16     if name.is_empty() {
17         return Err(FilenameError::FilenameEmpty.into());
18     }
19
20     // 定义无效字符
21     const INVALID_CHARS: &[u8] = &[
22         0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
23         0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
24         0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19,
25         0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
26         0x20, // space
27         0x22, // "
28         0x2A, // *
29         0x2B, // +
30         0x2C, // ,
31         0x2F, // /
32         0x3A, // :
33         0x3B, // ;
34         0x3C, // <
35         0x3D, // =
36         0x3E, // >
37         0x3F, // ?
38         0x5B, // [
39         0x5C, // \
40         0x5D, // ]
41         0x7C, // |
42     ];
43
44     // 检查无效字符

```

```
43     for byte in name.bytes() {
44         if INVALID_CHARS.contains(&byte) {
45             return Err(FileNameError::InvalidCharacter.into());
46         }
47     }
48
49     // 转换为大写
50     let name_upper = name.to_uppercase();
51
52     // 查找点号位置
53     if let Some(dot_pos) = name_upper.find('.') {
54         // 有扩展名的情况
55         let basename = &name_upper[..dot_pos];
56         let extension = &name_upper[dot_pos + 1..];
57
58         // 检查基名长度
59         if basename.len() > 8 {
60             return Err(FileNameError::NameTooLong.into());
61         }
62
63         // 检查扩展名长度
64         // if extension.len() > 3 {
65         //     return Err(FileNameError::ExtensionTooLong.into());
66         // }
67
68         // 检查点号位置（不能在第9个字符之后）
69         if dot_pos > 8 {
70             return Err(FileNameError::MisplacedPeriod.into());
71         }
72
73         // 构建文件名数组
74         let mut name_array = [0x20u8; 8]; // 用空格填充
75         let mut ext_array = [0x20u8; 3]; // 用空格填充
76
77         // 复制基名
78         for (i, byte) in basename.bytes().enumerate() {
79             name_array[i] = byte;
80         }
81
82         // 复制扩展名
83         for (i, byte) in extension.bytes().enumerate() {
84             ext_array[i] = byte;
```



```
85         }
86
87         Ok(ShortFileName {
88             name: name_array,
89             ext: ext_array,
90         })
91     } else {
92         // 没有扩展名的情况
93         if name_upper.len() > 8 {
94             return Err(FilenameError::NameTooLong.into());
95         }
96
97         let mut name_array = [0x20u8; 8]; // 用空格填充
98         let ext_array = [0x20u8; 3];      // 扩展名全为空格
99
100        // 复制基名
101        for (i, byte) in name_upper.bytes().enumerate() {
102            name_array[i] = byte;
103        }
104
105        Ok(ShortFileName {
106            name: name_array,
107            ext: ext_array,
108        })
109    }
110 }
```

在上面的代码中，我们实现了从字符串解析 ShortFileName 的函数。主要步骤如下：

1. 检查文件名是否为空。
2. 定义无效字符列表，并检查文件名是否包含无效字符。
3. 将文件名转换为大写。
4. 查找点号位置，并根据点号位置分离基名和扩展名。
5. 检查基名和扩展名的长度是否符合 FAT16 规范。
6. 构建 ShortFileName 实例并返回。

✓ 阶段成果

在补全 DirEntry 结构体后，可以使用单元测试来验证其正确性。

测试结果如下图所示：

```

running 1 test
test fs::fat16::dirent::tests::test_dir_entry ... ok

successes:

---- fs::fat16::dirent::tests::test_dir_entry stdout ----
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,
}

successes:
  fs::fat16::dirent::tests::test_dir_entry

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

```

2.3.3 Fat16Impl

三 补全 Fat16Impl

在实现了上述文件系统的数据格式之后，你需要在 `fs/fat16/impls.rs` 中实现你需要的一系列函数，包括但不限于：

1. 将 `cluster: &Cluster` 转换为 `sector`
2. 根据当前 `cluster: &Cluster`，利用 FAT 表，获取下一个 `cluster`
3. 根据当前文件夹 `dir: &Directory` 信息，获取名字为 `name: &str` 的 `DirEntry`
4. 遍历文件夹 `dir: &Directory`，获取其中文件信息
5. 其他你可能需要的帮助函数
6. 在实现了一系列函数后，为 `impl FileSystem for Fat16` 补全实现：

`Iterator<Item = Metadata>` 可以简单利用 `Vec::into_iter` 作为返回值，不需要考虑懒求值。

`FileHandle` 的 `file` 部分直接使用 `fs/fat16/file.rs` 中定义的 `File` 结构体，并使用 `Box` 包装。

最后，为 `File` 实现 `Read trait`，需要注意：

- `cluster` 链需要使用上述函数读取 FAT 表进行获取。
- `offset` 用于记录当前文件读取到了什么位置，需要实时更新。
- 一个 `cluster` 中存在多个 `sector`，你需要根据 `bpb` 信息进行读取操作。

- buf 参数是不定长的，你需要考虑文件长度、块长度以及缓冲区长度，来决定什么时候终止读取。

```

1  impl Fat16Impl { Rust
2      pub fn new(inner: impl BlockDevice<Block512>) -> Self {
3          let mut block = Block::default();
4          let block_size = Block512::size();
5
6          inner.read_block(0, &mut block).unwrap();
7          let bpb = Fat16Bpb::new(block.as_ref()).unwrap();
8
9          trace!("Loading Fat16 Volume: {:#?}", bpb);
10
11         // HINT: FirstDataSector = BPB_ResvdSecCnt + (BPB_NumFATs *
12         //      FATSz) + RootDirSectors;
13
14         let fat_start = bpb.reserved_sector_count() as usize;
15
16         // let root_dir_size = { /* DONE: get the size of root dir from
17         //      bpb */ };
18         // 根目录区域大小 = (根目录条目数 * 32) / 每扇区字节数
19         let root_dir_size = ((bpb.root_entries_count() as usize * 32) +
20         //      (bpb.bytes_per_sector() as usize - 1)) /
21         //      bpb.bytes_per_sector() as usize;
22         // let first_root_dir_sector = { /* FIXME: calculate the first
23         //      root dir sector */ };
24         // 根目录起始扇区 = 保留扇区 + (FAT表数量 * 每FAT扇区数)
25         let first_root_dir_sector = fat_start +
26         //      (bpb.fat_count() as usize *
27         //      bpb.sectors_per_fat() as usize);
28
29         let first_data_sector = first_root_dir_sector + root_dir_size;
30
31         Self {
32             bpb,
33             inner: Box::new(inner),
34             fat_start,
35             first_data_sector,
36             first_root_dir_sector,
37         }
38     }
39 }

```

```

35     pub fn cluster_to_sector(&self, cluster: &Cluster) -> usize {
36         match *cluster {
37             Cluster::ROOT_DIR => self.first_root_dir_sector,
38             Cluster(c) => {
39                 // DONE: calculate the first sector of the cluster
40                 // HINT: FirstSectorofCluster = ((N - 2) *
41                     BPB_SecPerClus) + FirstDataSector;
42                 if c < 2 {
43                     // 簇号小于2是无效的
44                     self.first_data_sector
45                 } else {
46                     ((c - 2) as usize * self.bpb.sectors_per_cluster()
47                         as usize) + self.first_data_sector
48                 }
49             }
50         }
51         // DONE: YOU NEED TO IMPLEMENT THE FILE SYSTEM OPERATIONS HERE
52         // - read the FAT and get next cluster
53         // - traverse the cluster chain and read the data
54         // - parse the path
55         // - open the root directory
56         // - ...
57         // - finally, implement the FileSystem trait for Fat16 with
58         //   `self.handle`
59         /// 读取 FAT 表获取下一个簇
60         pub fn get_next_cluster(&self, cluster: u16) ->
61             FsResult<Option<u16>> {
62             // FAT16 中每个 FAT 条目占 2 字节
63             let fat_offset = cluster as usize * 2;
64             let sector_offset = fat_offset / self.bpb.bytes_per_sector() as
65                 usize;
66             let byte_offset = fat_offset % self.bpb.bytes_per_sector() as
67                 usize;
68             let mut block = Block::default();
69             self.inner.read_block(self.fat_start + sector_offset, &mut
70                 block)?;
71             let fat_entry = u16::from_le_bytes([

```

```

70         block.as_ref()[byte_offset],
71         block.as_ref()[byte_offset + 1]
72     ]);
73
74     match fat_entry {
75         0x0000 => Ok(None), // 空簇
76         0xFFF8..0xFFFF => Ok(None), // 簇链结束
77         _ => Ok(Some(fat_entry)), // 下一个簇
78     }
79 }
80
81 /// 读取目录条目
82 pub fn read_dir_entries(&self, cluster: &Cluster) ->
83     FsResult<Vec<DirEntry>> {
84     let mut entries = Vec::new();
85     let mut current_cluster = *cluster;
86
87     loop {
88         let sector = self.cluster_to_sector(&current_cluster);
89         let sectors_per_cluster = match current_cluster {
90             Cluster::ROOT_DIR => {
91                 // 根目录的大小
92                 ((self.bpb.root_entries_count() as usize * 32) +
93                  (self.bpb.bytes_per_sector() as usize - 1)) /
94                 self.bpb.bytes_per_sector() as usize
95             }
96             _ => self.bpb.sectors_per_cluster() as usize,
97         };
98
99         // 读取簇中的所有扇区
100         for i in 0..sectors_per_cluster {
101             let mut block = Block::default();
102             self.inner.read_block(sector + i, &mut block)?;
103
104             // 每个扇区可以包含多个目录条目 (512 / 32 = 16)
105             for j in 0..(self.bpb.bytes_per_sector() as usize /
106                          DirEntry::LEN) {
107                 let offset = j * DirEntry::LEN;
108                 let entry_data = &block.as_ref()[offset..offset +
109                                     DirEntry::LEN];
110
111                 // 检查是否到达目录结束

```

```
109         if entry_data[0] == 0x00 {
110             return Ok(entries);
111         }
112
113         // 跳过已删除的条目
114         if entry_data[0] == 0xE5 {
115             continue;
116         }
117
118         // 解析目录条目
119         if let Ok(entry) = DirEntry::parse(entry_data) {
120             if entry.is_valid() && !entry.is_long_name() {
121                 entries.push(entry);
122             }
123         }
124     }
125 }
126
127 // 对于根目录, 不需要跟随簇链
128 if let Cluster::ROOT_DIR = current_cluster {
129     break;
130 }
131
132 // 获取下一个簇
133 if let Cluster(c) = current_cluster {
134     if let Ok(Some(next_cluster)) = self.get_next_cluster(c
135         as u16) {
136         current_cluster = Cluster(next_cluster as u32);
137     } else {
138         break;
139     }
140 } else {
141     break;
142 }
143
144 Ok(entries)
145 }
146
147 /// 在目录中查找指定名称的条目
148 pub fn find_entry(&self, dir_cluster: &Cluster, name: &str) ->
    FsResult<Option<DirEntry>> {
```

```
149     let entries = self.read_dir_entries(dir_cluster)?;
150     let target_name = ShortFileName::parse(name)?;
151     if name.is_empty() {
152         return Ok(None); // 空名称不匹配任何条目
153     }
154     for entry in entries {
155         if entry.filename == target_name {
156             return Ok(Some(entry));
157         }
158     }
159
160     Ok(None)
161 }
162
163 /// 解析路径并查找文件/目录
164 pub fn find_path(&self, path: &str) -> FsResult<Option<DirEntry>> {
165     let path = path.trim_start_matches('/');
166     if path.is_empty() {
167         return Ok(None); // 根目录没有对应的 DirEntry
168     }
169     // info!("Finding path: {}", path);
170     let parts: Vec<&str> = path.split('/').collect();
171     let mut current_cluster = Cluster::ROOT_DIR;
172     let mut lenth = 0;
173     for (i, part) in parts.iter().enumerate() {
174         if part.is_empty() {
175             continue; // 跳过空部分
176         }
177         lenth += 1;
178     }
179     info!("Path parts: {:?}, length: {}", parts, lenth);
180     for (i, part) in parts.iter().enumerate() {
181         if let Some(entry) = self.find_entry(&current_cluster,
182             part)? {
183             info!("Found entry: {:?}", entry);
184             if i == lenth - 1 {
185                 // 找到目标文件/目录
186                 info!("Returning entry: {:?}", entry);
187                 return Ok(Some(entry));
188             } else {
189                 // 中间路径必须是目录
190                 if entry.is_directory() {
```

```

190             current_cluster = entry.cluster;
191         } else {
192             return Err(FsError::NotADirectory);
193         }
194     }
195 } else {
196     return Ok(None); // 路径不存在
197 }
198 }
199
200 Ok(None)
201 }
202
203 /// 读取文件数据
204 pub fn read_file_data(&self, start_cluster: u16, offset: usize, buf:
205     &mut [u8]) -> FsResult<usize> {
206     let mut current_cluster = start_cluster;
207     let mut bytes_read = 0;
208     let mut file_offset = 0;
209     let cluster_size = self.bpb.sectors_per_cluster() as usize *
210         self.bpb.bytes_per_sector() as usize;
211     // 跳过不需要的簇
212     while file_offset + cluster_size <= offset {
213         if let Ok(Some(next)) =
214             self.get_next_cluster(current_cluster) {
215                 current_cluster = next;
216                 file_offset += cluster_size;
217             } else {
218                 return Ok(0); // 文件结束
219             }
220     }
221     // 开始读取数据
222     while bytes_read < buf.len() {
223         let sector = self.cluster_to_sector(&Cluster(current_cluster
224             as u32));
225         // 读取簇中的所有扇区
226         for i in 0..self.bpb.sectors_per_cluster() as usize {
227             if bytes_read >= buf.len() {
228                 break;

```



```
228         }
229
230         let mut block = Block::default();
231         self.inner.read_block(sector + i, &mut block)?;
232
233         let sector_data = block.as_ref();
234         let sector_offset_in_file = file_offset + i *
235         self.bpb.bytes_per_sector() as usize;
236
237         // 计算在当前扇区中的起始位置
238         let start_in_sector = if offset > sector_offset_in_file
239         {
240             offset - sector_offset_in_file
241         } else {
242             0
243         };
244
245         // 计算要读取的字节数
246         let bytes_to_read = core::cmp::min(
247             self.bpb.bytes_per_sector() as usize -
248             start_in_sector,
249             buf.len() - bytes_read
250         );
251
252         if bytes_to_read > 0 {
253             buf[bytes_read..bytes_read + bytes_to_read]
254                 .copy_from_slice(&sector_data[start_in_sector
255                 + bytes_to_read]);
256             bytes_read += bytes_to_read;
257         }
258     }
259
260     file_offset += cluster_size;
261
262     // 获取下一个簇
263     if let Ok(Some(next)) =
264         self.get_next_cluster(current_cluster) {
265         current_cluster = next;
266     } else {
267         break; // 文件结束
268     }
269 }
```

```
264     }
265
266     Ok(bytes_read)
267 }
268 }
269
270 impl FileSystem for Fat16 {
271     fn read_dir(&self, path: &str) -> FsResult<Box<dyn Iterator<Item =
    Metadata> + Send>> {
272         // DONE: read dir and return an iterator for all entries
273         let cluster = if path == "/" || path.is_empty() {
274             Cluster::);
281                 }
282             } else {
283                 return Err(FsError::);
284             }
285         };
286
287         let entries = self.handle.read_dir_entries(&cluster)?;
288         let metadata_list: Vec<Metadata> = entries
289             .into_iter()
290             .map(|entry| {
291                 let entry_type = if entry.is_directory() {
292                     FileType::
```

```
305         })
306         .collect();
307
308         Ok(Box::new(metadata_list.into_iter()))
309     }
310
311     fn open_file(&self, path: &str) -> FsResult<FileHandle> {
312         // DONE: open file and return a file handle
313         if let Some(entry) = self.handle.find_path(path)? {
314             if entry.is_file() {
315                 let file = File::new(self.handle.clone(), entry.clone());
316
317                 // 从 DirEntry 创建 Metadata
318                 let metadata = Metadata::new(
319                     entry.filename.to_string(),
320                     FileType::File,
321                     entry.size as usize,
322                     Some(entry.created_time),
323                     Some(entry.modified_time),
324                     Some(entry.accessed_time),
325                 );
326
327                 Ok(FileHandle::new(metadata, Box::new(file)))
328             } else {
329                 Err(FsError::NotAFile)
330             }
331         } else {
332             Err(FsError::FileNotFound)
333         }
334     }
335
336     fn metadata(&self, path: &str) -> FsResult<Metadata> {
337         // DONE: read metadata of the file / dir
338         if path == "/" || path.is_empty() {
339             // 根目录的元数据
340             return Ok(Metadata::new(
341                 "/".to_string(),
342                 FileType::Directory,
343                 0,
344                 None, // 根目录通常没有创建时间
345                 None,
346                 None,
```

```

347         ));
348     }
349
350     if let Some(entry) = self.handle.find_path(path)? {
351         let entry_type = if entry.is_directory() {
352             FileType::Directory
353         } else {
354             FileType::File
355         };
356
357         Ok(Metadata::new(
358             entry.filename.to_string(),
359             entry_type,
360             entry.size as usize,
361             Some(entry.created_time),
362             Some(entry.modified_time),
363             Some(entry.accessed_time),
364         ))
365     } else {
366         Err(FsError::FileNotFound)
367     }
368 }
369
370 fn exists(&self, path: &str) -> FsResult<bool> {
371     // DONE: check if the file / dir exists
372     if path == "/" || path.is_empty() {
373         return Ok(true);
374     }
375
376     Ok(self.handle.find_path(path)?.is_some())
377 }
378 }

```

在上面的代码中，我们实现了 `Fat16Impl` 的主要功能：

1. 在构造函数中读取 BPB 信息，并计算 FAT 表和根目录的起始扇区。
2. 实现 `cluster_to_sector` 函数，将簇转换为扇区。
3. 实现 `get_next_cluster` 函数，从 FAT 表中获取下一个簇。
4. 实现 `read_dir_entries` 函数，读取指定簇中的目录条目。
5. 实现 `find_entry` 函数，在指定目录中查找指定名称的条目。
6. 实现 `find_path` 函数，解析路径并查找文件或目录。
7. 实现 `read_file_data` 函数，读取文件数据。

2.4 接入操作系统

≡ 接入操作系统

在实现了上述内容后，相信你已经迫不及待想要去进行调试你的文件系统是否能够正确运行了。

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  pub fn init() {
2      info!("Opening disk device...");
3
4      let drive = AtaDrive::open(0, 0).expect("Failed to open disk
5      device");
6
7      // only get the first partition
8      let part = MbrTable::parse(drive)
9      .expect("Failed to parse MBR")
10     .partitions()
11     .expect("Failed to get partitions")
12     .remove(0);
13
14     info!("Mounting filesystem...");
15
16     ROOTFS.call_once(|| Mount::new(Box::new(Fat16::new(part)),
17     "/".into()));
18
19     trace!("Root filesystem: {:#?}", ROOTFS.get().unwrap());
20
21     info!("Initialized Filesystem.");
22 }

```

2.4.1 ls 命令

≡ 实现 ls 命令

之后，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。

为了实现的便利，可以定义添加如下的系统调用，专用于在内核态直接打印文件夹信息，而不是将这些数据传递给用户态处理。

```
1 // path: &str (arg0 as *const u8, arg1 as len)
2 Syscall::ListDir => list_dir(&args),
```

 Rust

```
1 pub fn ls(root_path: &str) {
2     info!("Listing directory: {}", root_path);
3     let iter = match get_rootfs().read_dir(root_path) {
4         Ok(iter) => iter,
5         Err(err) => {
6             warn!("{:?}", err);
7             return;
8         }
9     };
10
11     // DONE: format and print the file metadata
12     // - use `for meta in iter` to iterate over the entries
13     // - use `crate::humanized_size_short` for file size
14     // - add '/' to the end of directory names
15     // - format the date as you like
16     // - do not forget to print the table header
17     // 打印表头
18     info!("Directory listing for: {}", root_path);
19     info!("{: <12} {: >10} {: >8} {: <20} {}",
20         "Type", "Size", "Name", "Modified", "Created");
21     info!("{}", "-".repeat(70));
22
23     // 遍历目录条目
24     for meta in iter {
25         let type_str = if meta.is_dir() { "DIR" } else { "FILE" };
26
27         // 格式化文件名，目录添加 '/' 后缀
28         let name_display = if meta.is_dir() {
29             format!("{}/", meta.name)
30         } else {
31             meta.name.clone()
32         };
33
34         // 格式化文件大小
35         let size_display = if meta.is_dir() {
```

 Rust

```

36         "-".to_string()
37     } else {
38         let (value, unit) = crate::humanized_size_short(meta.len as
39             u64);
40         format!("{:.1}{}", value, unit) // Format the tuple into a
41             String
42     };
43
44     // 格式化时间
45     let modified_str = match meta.modified {
46         Some(time) => format!("{}", time.format("%Y-%m-%d %H:%M")),
47         None => "N/A".to_string(),
48     };
49
50     let created_str = match meta.created {
51         Some(time) => format!("{}", time.format("%Y-%m-%d %H:%M")),
52         None => "N/A".to_string(),
53     };
54
55     info!("{:<12} {:>10} {:>8} {:<20} {}",
56         type_str,
57         size_display,
58         name_display,
59         modified_str,
60         created_str);
61 }

```

2.4.2 读取文件

☞ 实现 cat 命令

为了读取一个文件，约定一个用户态程序需要遵循 open - read - close 过程。

在 utils/resources.rs 中扩展 Resource 枚举：

```

1 pub enum Resource {
2     File(FileHandle),
3     // ...
4 }

```

 Rust

在对 Resource 的实现中，可以直接使用 file.read(buf) 进行读取，而对于写入操作，由于不做要求，你可以直接忽略。

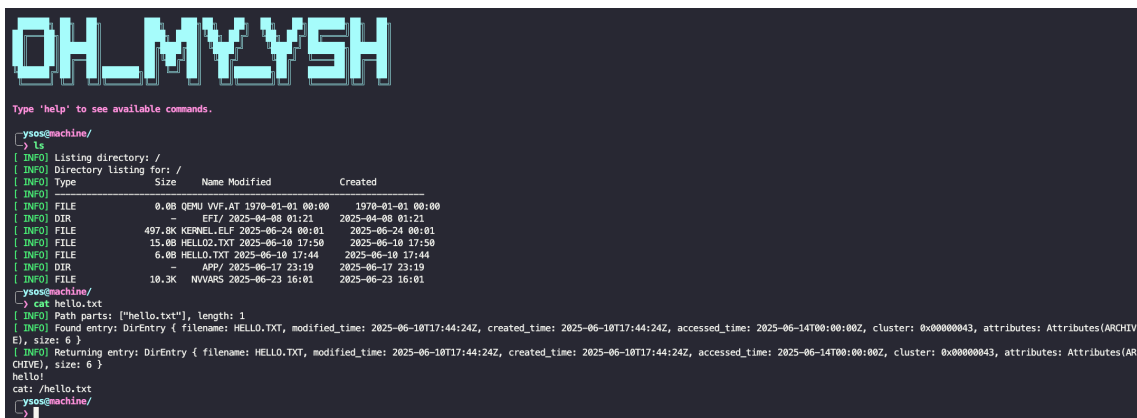
```
1  /// The `Read` trait allows for reading bytes from a source. 🔒 Rust
2  pub trait Read {
3      /// Pull some bytes from this source into the specified buffer,
4      /// returning
5      /// how many bytes were read.
6      fn read(&mut self, buf: &mut [u8]) -> FsResult<usize>;
7
8      /// Read all bytes until EOF in this source, placing them into `buf`.
9      fn read_all(&mut self, buf: &mut Vec<u8>) -> FsResult<usize> {
10         let mut total_bytes_read_in_call = 0;
11         loop {
12             // FIXME: read data into the buffer
13             // - extend the buffer if it's not big enough
14             // - break if the read returns 0 or Err
15             // - update the length of the buffer if data was read
16             if buf.capacity() == buf.len() {
17                 buf.reserve(1024);
18             }
19             // 获取 Vec 中未初始化部分的切片
20             let mut temp_buf = [0u8; 1024]; // 临时缓冲区
21             match self.read(&mut temp_buf) {
22                 Ok(0) => {
23                     // EOF
24                     break;
25                 }
26                 Ok(n) => {
27                     buf.extend_from_slice(&temp_buf[..n]);
28                     total_bytes_read_in_call += n;
29                 }
30                 Err(e) => {
31                     return Err(e);
32                 }
33             }
34         }
35         Ok(total_bytes_read_in_call)
36     }
37 }
```


✓ 阶段成果

在实现了 `ls` 和 `cat` 命令后，你可以在用户态运行这些命令来验证文件系统的正确性。

例如，运行 `ls /` 可以列出根目录下的所有文件和目录，而运行 `cat /path/to/file` 可以读取并打印文件内容。

测试结果如下图所示：



```
OH_MY_YSH
Type 'help' to see available commands.

ysos@machine/
> ls
[INFO] Listing directory: /
[INFO] Directory listing for: /
[INFO] Type      Size   Name Modified      Created
[INFO] -----
[INFO] FILE      0.0B GENU VVF.AT 1970-01-01 00:00 1970-01-01 00:00
[INFO] DIR        -     EFT/ 2025-04-08 01:21 2025-04-08 01:21
[INFO] FILE    497.8K KERNEL.ELF 2025-06-24 00:01 2025-06-24 00:01
[INFO] FILE    15.0B HELLO2.TXT 2025-06-10 17:50 2025-06-10 17:50
[INFO] FILE     6.0B HELLO1.TXT 2025-06-10 17:44 2025-06-10 17:44
[INFO] DIR        -     APP/ 2025-06-17 23:19 2025-06-17 23:19
[INFO] FILE    10.3K NVARS 2025-06-23 16:01 2025-06-23 16:01

ysos@machine/
> cat hello.txt
[INFO] Path parts: ["hello.txt"], length: 1
[INFO] Found entry: DirEntry { filename: HELLO.TXT, modified_time: 2025-06-10T17:44:24Z, created_time: 2025-06-10T17:44:24Z, accessed_time: 2025-06-14T00:00:00Z, cluster: 0x00000043, attributes: Attributes(ARCHIVE), size: 6 }
[INFO] Returning entry: DirEntry { filename: HELLO.TXT, modified_time: 2025-06-10T17:44:24Z, created_time: 2025-06-10T17:44:24Z, accessed_time: 2025-06-14T00:00:00Z, cluster: 0x00000043, attributes: Attributes(ARCHIVE), size: 6 }
hello!
cat: /hello.txt
ysos@machine/
>
```

3 思考题

3.1 q1

? Question

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？

这是一个条件编译属性，它的意思是：在正常编译时启用 `#![no_std]` 模式，但在运行单元测试时链接标准库 `std`。

这在内核或嵌入式开发中非常关键：

- 内核环境限制：操作系统内核不能依赖另一个操作系统提供的服务（如文件、网络、堆内存等），而这些都是 `std` 库提供的。因此，内核必须在 `no_std` 环境下编译。
- 测试便利性：在 `no_std` 环境下测试非常困难，因为缺少测试框架、打印输出、断言等基础设施。通过在测试时启用 `std`，开发者可以在宿主操作系统上利用完整的标准库和测试工具链来测试模块的纯逻辑部分，极大地简化了测试。

在内核中进行单元测试相对困难，主要原因包括：

- 无底层环境：内核本身就是最底层的软件，没有 `std` 库，没有现成的内存分配器、线程管理等。测试代码需要自己处理这些最基本的问题。
- 硬件依赖：许多内核代码直接与硬件交互（如读写 I/O 端口、处理中断）。在普通的测试环境中无法模拟这些硬件行为，必须使用专门的模拟器（如 QEMU）或在真实硬件上运行。
- 执行环境隔离：内核运行在特权级（Ring 0），一个错误（如无效的内存访问）可能导致整个系统崩溃，而不是像用户程序那样仅仅是进程崩溃，这使得调试和测试的风险更高。
- 缺乏测试框架：标准的测试框架依赖 `std` 库。在 `no_std` 环境下，需要使用专门的、功能有限的测试框架。

3.2 q2

? Question

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

- `T: BlockDevice`：此约束保证了泛型参数 `T` 是一个块设备，具备按块读取数据的能力。`B` 是块大小的泛型参数，这使得 `MbrTable` 可以适用于不同块大小的设备。
- `Clone`：此约束要求块设备类型 `T` 是可以被克隆的。当 `MbrTable` 解析出一个分区后，会为该分区创建一个逻辑设备视图（`Partition`）。这个 `Partition` 结构体需要拥有一个独立的底层设备实例，通过 `Clone` 可以安全地复制设备句柄（通常是轻量级的克隆，如克隆一个 `Arc` 指针），从而避免所有权冲突。

`PhantomData` 是一个零大小的标记类型。它的作用是告诉编译器，`MbrTable<T, B>` 这个结构体虽然没有一个类型为 `B` 的字段，但在逻辑上它“拥有”或“关联”着泛型参数 `B`。

在这个场景下，`B` 是 `BlockDevice trait` 的一部分。如果不加 `PhantomData`，编译器可能会报错，认为泛型参数 `B` 未被使用。通过添加它，我们明确了 `MbrTable` 的行为与块大小 `B` 相关，这对于类型检查和保证泛型约束的正确性至关重要。

在 `PartitionTable` trait 中，`Self: Sized` 约束意味着实现该 trait 的任何类型都必须具有在编译时已知的大小。

默认情况下，Rust 的 trait 会隐式地包含这个约束。这是因为如果一个 trait 的方法按值返回 Self（如 `fn new() -> Self`）或接收 Self 作为参数，编译器必须知道 Self 的大小才能在栈上创建或移动它。不带 Sized 约束的 trait（`trait MyTrait: ?Sized`）可以用来创建大小不定的 **trait object**（如 `&dyn MyTrait`）。

3.3 q3

question [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？]

AtaDrive 结构体通常包含指向其所属总线（AtaBus）的引用以及自身的标识信息。为了方便实现 Clone 并解决生命周期问题，常见的做法是让 AtaBus 被包裹在一个引用计数的智能指针 Arc 中。

AtaDrive 则持有这个 `Arc<AtaBus>`。Arc 本身实现了 Clone，它的克隆操作非常轻量，只是增加引用计数。因此，AtaDrive 的 Clone 实现只需克隆内部的 `Arc<AtaBus>` 并复制其他简单字段即可。

这种设计遵循了单一职责原则和组合优于继承的思想，好处显著：

- 模型映射现实：精确地模拟了物理硬件的结构。一个 ATA 总线（AtaBus）可以连接多个驱动器（AtaDrive）。
- 资源共享与管理：AtaBus 代表了共享的物理资源（如 I/O 端口）。将其实现为一个可共享的实体（如 `Arc<Mutex<AtaBus>>`），允许多个 AtaDrive 实例安全地并发访问总线。
- 模块化和可测试性：AtaBus 的逻辑（底层通信）和 AtaDrive 的逻辑（高层读写）可以独立开发和测试。
- 代码清晰：职责划分清晰，使得代码更容易理解和维护。

3.4 q4

? Question

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：

1. 函数声明：

- `fn f<T: Foo>(f: T) -> usize`
- `fn f(f: impl Foo) -> usize`
- `fn f(f: &dyn Foo) -> usize`

2. 结构体声明：

- `struct S<T: Foo> { f: T }`
- `struct S { f: Box<dyn Foo> }`

这几种写法代表了 Rust 中实现多态的两种主要方式：静态分发和动态分发。

- `fn f<T: Foo>(f: T) -> usize`
- `fn f(f: impl Foo) -> usize`

这两者在功能上几乎等价，都使用 **静态分发 (Static Dispatch)**。

- 机制：编译器会为调用 `f` 的每一种具体类型 `T` 生成一个专门的函数版本（单态化）。
- 优点：性能极高，无运行时开销，可进行深度优化。
- 缺点：可能导致最终的二进制文件体积增大。

- `fn f(f: &dyn Foo) -> usize`

这个写法使用 **动态分发 (Dynamic Dispatch)**。

- 机制：参数 `f` 是一个 **trait object**（胖指针），包含数据指针和虚函数表（vtable）指针。在运行时通过 vtable 查找并调用正确的方法。
- 优点：二进制文件体积更小，允许创建异构集合（如 `Vec<&dyn Foo>`）。
- 缺点：有轻微的运行时开销（指针间接调用），且阻止编译器内联。

- `struct S<T: Foo> { f: T }`

- 静态分发。结构体 `S` 的大小取决于具体的类型 `T`。`S<TypeA>` 和 `S<TypeB>` 是两个完全不同的类型。

- `struct S { f: Box<dyn Foo> }`

- 动态分发。字段 `f` 是一个指向堆上 trait object 的智能指针。结构体 `S` 的大小是固定的，与 `f` 内部存储的具体类型无关。

3.5 q5

? Question

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？

特性	硬链接 (Hard Link)	软链接 (Symbolic Link)
本质	一个文件内容 (inode) 的多个目录入口（文件名）。	一个特殊的文件，其内容是另一个文件的路径。
Inode	与原始文件共享同一个 inode。	拥有自己独立的 inode。

删除	删除链接只是减少 inode 的链接数。只有当链接数降为 0 时，文件数据才会被真正删除。	删除软链接本身，对原始文件没有影响。若原始文件被删除，软链接会失效。
跨文件系统	不可以。Inode 在单个文件系统中是唯一的。	可以。它只是一个路径字符串。
目标类型	通常不能指向目录。	可以指向文件或目录。

Windows “快捷方式” vs. Linux 软链接

- 相同点：
 - 都是一种“指针”，指向另一个文件或目录。
 - 如果目标被删除或移动，它们都会失效。
- 不同点：
 - 实现层面：软链接是 文件系统层面 的特性，对应用透明。快捷方式（.lnk）是 **Windows Shell** 层面 的特性，本质是一个普通文件，需要特定程序（如资源管理器）才能解析。
 - 功能：软链接只存储目标路径。快捷方式可以存储更丰富的信息，如启动参数、工作目录、图标等。
 - 兼容性：软链接是 POSIX 标准。快捷方式是 Windows 特有的。（注：现代 NTFS 文件系统也支持类似软链接的符号链接 mklink）。

3.6 q6

?

Question

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？

特性	非日志文件系统 (e.g., FAT)	日志文件系统 (e.g., NTFS, ext4)
写操作	直接在磁盘上的元数据和数据块上进行修改。	先将修改操作写入一个独立的日志区域，再将变更应用到磁盘。
核心组件	文件分配表 (FAT)、目录、数据区。	在非日志系统的基础上，增加了一个 日志 (Journal) 区域。
操作流程	直接更新元数据和数据。若中途断电，可能导致不一致。	1. Journaling : 将事务写入日志。 2. Commit : 标记事务已完整记录。 3. Checkpointing : 将日志中的变更实际写入磁盘。

异常崩溃后的恢复

- 非日志文件系统 (FAT):

- 恢复机制：必须进行 全盘扫描（如 `chkdsk` 或 `fsck`），检查并尝试修复所有元数据结构的不一致性。
- 恢复速度：非常慢。速度与磁盘容量和文件数量成正比，对于大容量磁盘可能需要数小时。

- 日志文件系统 (NTFS, ext4):

- 恢复机制：日志重放 (**Journal Replay**)。系统重启后，只需检查日志文件：
 - 对于已提交但未完全写入磁盘的事务，会根据日志记录“重放”这些操作，完成写入。
 - 对于未提交的事务，直接忽略。
- 恢复速度：非常快。系统只需处理通常很小的日志文件，恢复过程通常在数秒内完成，与磁盘总容量无关，能快速恢复到一致状态。