Lab9 - File Descriptor, IPC (Pipe), and VFS

0. 前言

实验概述

本次实验课将了解文件描述符（File Descriptor），管道（Pipe）作为进程间通信（IPC），以及虚拟文件系统（VFS）。

实验代码

拉取实验代码：

```
$ git clone https://github.com/sdww0/sustech-os-lab.git && cd sustech-os-lab/lab9-fd-ipc-vfs
```

或更新实验代码：

```
# In sustech-os-lab
$ git pull && cd lab9-fd-ipc-vfs
```

本次实验代码新增：

1. File Table，FileLike trait，与File Descriptor的实现：`src/fs/file_table.rs`，`src/fs/file.rs`
2. PIPE的实现：`src/fs/pipe.rs`，`src/syscall/pipe.rs`
3. VFS（包含Inode）框架与RamFS的实现：`src/fs/mod.rs`，`src/fs/ramfs.rs`
4. openat系统调用，打开某一文件

1. 文件描述符与文件表

在Linux编程中，访问文件需要先使用`open`函数打开文件，该函数返回一个整型数据，即文件描述符（File Descriptor）。文件描述符是操作系统为每个进程维护的文件表（File Table）的索引，用于跟踪进程打开的文件或其他资源。

文件描述符的后端不仅可以表示文件，还可以表示其他系统资源。Unix/Linux的设计哲学中包含一条：“一切皆文件”（Everything is a File），即**文件描述符可以统一表示各种资源**，例如通过网络套接字（socket）读写数据、与设备交互（device file）以及进程间通信（IPC）等。经典的文件描述符包括：标准输入（stdin，索引0）、标准输出（stdout，索引1）和标准错误（stderr，索引2）。

在之前的实现中，`read`和`write`系统调用忽略了传入的文件描述符，直接进行终端输入输出。这种方式不够通用，需要进一步抽象以支持对不同系统资源的访问。为此，我们引入了`FileLike` trait：

```
pub trait FileLike: Sync + Send {
    fn read(&self, writer: VmWriter) -> Result<usize>;
    fn write(&self, reader: VmReader) -> Result<usize>;
}
```

`read/write`接口即是对于系统资源的一个抽象。

引入`FileLike` trait后，我们便可以引入文件表`FileTable`，并修改`read`和`write`系统调用，使其获取当前进程的文件表并调用相应方法进行读写。为此，我们在PCB中新增了文件表，并在内部按顺序添加了stdin、stdout和stderr三种标准输入输出。`FileTable`的结构及初始化函数如下：

```
/// Represents an open file entry
pub struct FileEntry {
    file: Arc<dyn FileLike>,
}

/// File table structure
pub struct FileTable {
    table: Vec<Option<FileEntry>>,
    fds_in_use: usize,
}

impl FileTable {
    pub fn new_with_standard_io() -> Self {
        let mut table = Vec::new();
        table.push(Some(FileEntry {
            file: Arc::new(Stdin),
        }));
        table.push(Some(FileEntry {
            file: Arc::new(Stdout),
        }));
        table.push(Some(FileEntry {
            file: Arc::new(Stderr),
        }));
        FileTable {
            table,
            fds_in_use: 3,
        }
    }
}
```

文件表初始化时会添加标准输入输出，这些结构实现了`FileLike` trait，从而实现了与终端的交互。

2. IPC（PIPE）

进程间通信（Inter Process Communication，IPC）是**一种允许不同进程进行数据交互的机制**。最经典的IPC机制是管道（PIPE），在终端中经常使用。例如，使用`grep`命令搜索关键词时，会使用`ps | grep $PID`，其中`|`符号就是基于管道实现的。管道创建一个通道，将两个进程的输入输出连接起来，使上一个进程的输出作为下一个进程的输入。

管道会创建两个文件描述符，一个用于读取数据，另一个用于写入数据。它基于RingBuffer实现`FileLike` trait。大致流程如下图所示：

在系统中运行pipe程序，可以看到父子进程通过管道通信的流程：

```
[INIT] Starting Shell...
Running Shell...
~ # pipe
 Running command: pipe
Parent: Writing to pipe...
Parent: Write complete, waiting for child to finish...
Child: Waiting to read from pipe...
Child: Read 'Hello from parent!' from pipe
```

其他形式的IPC还包括信号（Signal）、共享内存（Shared Memory）和命名管道（Named Pipe）等。信号是一种常用的通知机制，用于通知进程某些事件的发生（类似于内核中断事件）。但由于信号与文件描述符无关，超出了本实验范围，因此未实现。

3. 文件系统，虚拟文件系统（VFS）与Inode

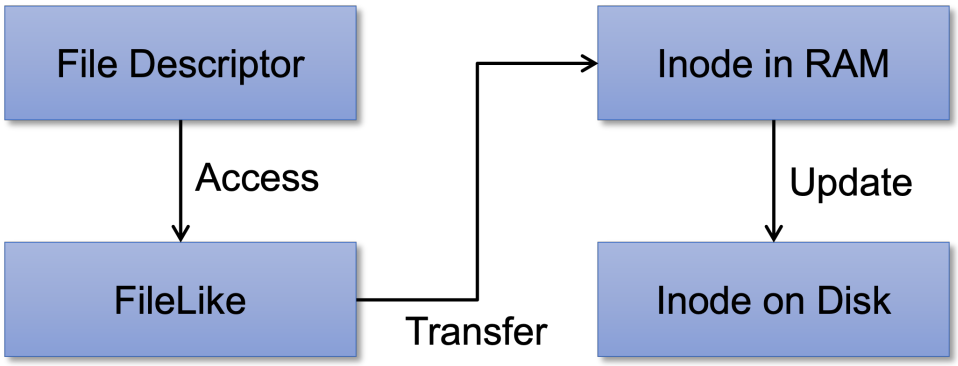
文件系统（如Ext4，ZFS，exFAT等）是一种存储和组织计算机数据的方法。它使用**文件**和**目录**代替了硬盘这类物理设备（**称为块设备**）的数据块概念，屏蔽了数据在硬盘等设备上的物理存储细节，用户只需通过目录路径和文件名即可访问数据，而无需关心其底层实现。同时，存储空间的分配与释放等管理工作，也由文件系统自动完成。

虚拟文件系统（Virtual File System，VFS）是操作系统内核中的一个抽象层，它通过定义一组通用的接口和数据结构，使得应用程序能够使用相同的系统调用访问不同类型的文件系统。实验代码为VFS引入了三个trait或结构：`FileSystem`、`Inode`和`InodeMeta`。这些定义了上层访问的接口和数据，其中最重要的是Inode（Index node）：

```
pub trait Inode: Send + Sync {
    fn open(self: Arc<Self>, name: String) -> Arc<dyn Inode>;
    fn read_at(&self, offset: usize, writer: VmWriter) -> Result<usize>;
    fn write_at(&self, offset: usize, reader: VmReader) -> Result<usize>;
    fn metadata(&self) -> &InodeMeta;
    fn size(&self) -> usize;
}
```

Inode是Unix类文件系统中用于描述文件系统对象（如文件或目录）的数据结构。Inode内部存储元数据（如文件大小、最后修改时间等）以及该对象在磁盘上的属性和位置（在更新到磁盘时使用）。

实验代码引入了`FileInode`作为包装器，将Inode包裹后实现`FileLike` trait，从而使上层在调用`read`/`write`时能够触发实际的文件读写。访问路径如下：



4. RamFS

RamFS是一种简单的文件系统，**基于内存实现文件操作**。与其他文件系统（如Ext4、exFAT）相比，RamFS实现简单，但具有易失性（因为数据存储在内存中，无法持久化到磁盘）。本实验代码提供了RamFS的基础支持，作为VFS实现的示例。它使用`Vec<u8>`存储数据，但目前不支持文件目录，只支持读写单个文件。相关结构如下：

```
pub struct RamInode {
    data: Mutex<Vec<u8>>,
    metadata: InodeMeta,
}

pub struct RamFS {
    root: Arc<RamInode>,
}
```

运行ramfs程序，能得到以下输出：

```
[INIT] Starting Shell...
Running Shell...
~ # ramfs
 Running command: ramfs
Content of hello: World
Content of world: World
```

5. 上手练习

在ramfs测试程序中，创建了两个文件然后分别写入了不同的内容，但当前的ramfs只支持单个文件，请你扩展当前ramfs的实现，新增目录功能，当`Inode::open`发现目录下没有文件时，创建一个普通文件并返回。支持完毕后终端输出应该为：

```
[INIT] Starting Shell...
Running Shell...
~ # ramfs
 Running command: ramfs
Content of hello: Hello
Content of world: World
```