RAM File System Shell

张哲恺(corax@smail.nju.edu.cn)

- RAM File System Shell
  - 前言
  - 简介
    - 背景知识
      - 文件系统
      - Shell
    - 项目概要
      - 任务说明
      - 前情提要
  - 开始你的项目
    - 获取项目框架
    - 项目框架导读
    - 项目引导
      - 前情提要
      - 文件系统部分
        - 数据结构设计
          - 文件节点
          - 文件描述符
        - init_ramfs
        - find
        - run_link
        - rrmdir
        - rmkdir
        - ropen
        - rseek
        - rread
        - rwrite
        - rclose
        - close_ramfs
      - Shell部分
        - 环境变量
        - init_shell
        - close_shell
        - sls
        - scat
        - smkdir
        - stouch
        - secho
        - swhich
    - 运行/测试说明
      - MacOS/Linux 用户
      - Windows 用户
  - 提交说明
    - 数据约定
      - 内存文件系统部分
      - Shell部分
    - 提交方式
      - MacOS/Linux 用户
      - Windows 用户

测试样例
样例1
样例2
样例3
样例4
样例5

前言

在本次项目中，同学们将会尝试完成与平时作业中不同的任务：你不是在一个文件中编写一个完整的程序，而是按照要求完善或实现分布在多个文件中的函数。测试代码中会调用你完成的这些函数，以检测是否完成了题目的要求。

如果你在完成作业的过程中感到困难或疑惑，不妨与出题助教联系以获得帮助，但请坚持独立自主完成本次作业😊。此外，为了方便大家完成作业，本次的文档中在各个函数中已经补充了去年项目中需要大家自行查询手册获取的内容（位于各个函数的分界线下方，如果你想更好地提升自己的能力，也可以选择不阅读引导中的内容，而是自行阅读手册并根据要求实现）。

简介

背景知识

文件系统

文件系统(File System)是操作系统的重要组成部分，当你打开Windows系统的资源管理器，你所看到的就是一个文件系统的`UI`界面。通过调用文件系统提供的接口，我们就可以将数据持久化到磁盘上。而C语言为我们提供了一系列接口，使程序员可以通过C语言的接口，基于操作系统对文件进行操作，例如：

```
FILE *fopen(const char *pathname, const char *mode);
void fclose(FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

我们要实现的文件系统参考了Linux的文件系统，在Linux环境下，它们是通过调用操作系统的以下接口来完成对应的功能的：

```
int open(const char *pathname, int flags);
int close(int fd);
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int unlink(const char *pathname);
```

注：上面的这些接口你在本次作业中**基本都不会用到，但阅读手册并熟悉它们的功能对你完成项目相当有益**，关于如何阅读它们的手册，我们会在下文提到。

我们要实现的文件系统结构与Linux的文件系统结构采用了一致的树形结构，具体描述如下：

在Linux的世界观中，文件系统中的所有对象都是文件。文件系统对每个文件的文件名长度是有约束的，在我们的文件系统中，**约定所有的basename（什么是basename？）需要满足≤32字节，否则视为不合法。**我们的文件系统在初始状态下只存在根目录/，文件系统中存在两类文件对象：**目录文件 (directory file)**和**常规文件 (regular file)。目录文件**下可以存放其他对象，而**常规文件**不可以，即在Linux文件系统的树形结构中，**常规文件**只能是叶节点。例如：

```
/
├── 1.txt       "/1.txt"
├── 2.txt       "/2.txt"
└── dir         "/dir"
    ├── 1.txt   "/dir/1.txt"
    └── 2.txt   "/dir/2.txt"
```

可以看到，在根目录下一共有3个项目：两个文件，一个目录dir，而dir下还可以拥有两个文件。右侧的字符串称为对象的“绝对路径”。

需要注意的是，在Linux系统下，如果绝对路径指示目录，则它的每一个/都可以被替换为多余的数个/，两者表意相同，且末尾也可以添加数个/，例如///dir////；如果绝对路径指示文件，则除了末尾不可以添加/，否则视为目录，路径中间的所有/都可以冗余，例如///dir///1.txt。你不妨自行在Linux环境下尝试这一点，但是需要指出的是，Linux系统会对/和//进行区分，但实际上却是同一目录，**在本次项目中，我们不考虑//的存在，一律视为/，同时，本次项目中的目录与文件名如果含有字母，数字和`.`之外的字符均视为非法取值。**

⚠在本次项目中，关于文件系统的部分沿用上述说明。并且不用考虑相对路径(`.`和`..`)，所有输入均为绝对路径。

Shell

Shell是**用户与操作系统内核之间的接口**，通过这个接口，你可以按照一定的方式获取操作系统提供的服务，其中很重要的一部分就是对文件系统的访问与操作。

什么是用户与操作系统内核之间的接口？现在同学们使用电脑基本上都是在通过图形化界面(Graphical User Interface)与操作系统内核进行交互，这就是一种用户与操作系统内核之间的接口。而在更早之前，计算机使用者使用字符界面与操作系统进行交互，现在Shell基本就特指这种字符界面的交互方式。举个例子，Linux系统常用的Shell有sh, bash等，Windows常用的Shell有powershell, cmd等。

项目概要

任务说明

在本次作业中，我们需要你在内存(Random Access Memory, RAM)上**模拟**一个文件系统，即：将数据“持久化”到内存上，而不是磁盘的主存(Memory)上，这是一个易失性的文件管理系统；同时，我们还需要你**模拟**一个不完整的Shell，实现对你模拟的文件系统的访问与操作，这包括一些基础命令，例如`ls`，`cat`等。

前情提要

经过今年的改版，本项目对Windows具备了更好的支持！至于具体的内容你可以参看后文的**运行/测试说明。**

如果你有额外的精力和时间，我们仍然推荐你尝试Linux系统！如何获取Linux环境：（推荐顺序从上到下）

1. 如果你有多余并且性能不算太差的电脑，我们十分推荐你备份数据后安装Linux操作系统获取原生体验（有一定难度）。
2. 你可以通过Windows官方提供的wsl(Windows Subsystem for Linux)来获取非原生Linux环境，尽管非原生，但也相当够用，注意使用此方式你几乎**只能使用Shell**进行交互（其他方式都可以使用图形界面和Shell），但这对你完成作业也有所帮助。你可以通过这个链接来查看安装教程，但过程中遇到的更多问题需要你自行STFW/RTFM。
3. 你可以通过安装虚拟机软件来获取原生Linux体验，但具体如何操作需要你自行检索。
4. 不要轻易尝试双系统（不推荐）。

开始你的项目

获取项目框架

我们为你准备了一个git仓库，请基于这个git仓库完成你的项目。如果你不会使用git，请自行检索并学着使用。

你可以使用下面的命令来获取项目框架：

```
git clone https://git.nju.edu.cn/Corax/ramfshell.git
```

请在默认的master分支上完成你的作业，最终OJ的评分也将以master分支上的内容为准。

在这个仓库中我们为你提供了一个自动编译脚本`Makefile`，并且为你配置好了记录自动追踪，请不要随意修改`Makefile`，除非你清楚你在干什么，你的提交记录将成为查重时证明独立完成的重要证据。

项目框架导读

项目的整体框架如下：

```
/ramfs-shell
├── .gitignore
├── Makefile
├── CMakeLists.txt
├── compile.ps1
├── README.md
├── main.c
├── include
|   ├── ramfs.h
|   └── shell.h
├── fs
|   └── ramfs.c
└── sh
    └── shell.c
```

1. .gitignore文件用于git仓库提交时，请不要随意修改，除非你知道你在做什么，否则可能出现提交失败的错误；
2. Makefile是我们为你提供的自动编译脚本，如前文所述；
3. CMakeLists.txt是为Clion及Cmake用户准备的编译脚本；
4. compile.ps1则是为Windows用户准备的编译脚本；
5. 你正在阅读的是README.md（也有可能是它导出的pdf）；
6. main.c用于进行测试；
7. include文件夹中包含了所有的头文件；
8. fs文件夹中包含了RAM File System的核心代码，你需要实现其中尚未完成的函数；
9. sh文件夹中包含了Shell的核心代码，你需要实现其中尚未完成的函数。

项目引导

前情提要

本引导中**数据结构设计部分的内容**旨在为完成本次项目有困难的同学提供思路，你可以自行选择是否按照这样的方式设计。

同时，需要完成内容中关于错误处理的描述只是必要的，你可以自行识别在该项目中还能处理哪些错误，以提高程序的鲁棒性。

文件系统部分

数据结构设计

文件节点

在内存文件系统中，你需要一个数据结构来存储一个**文件**的信息（我们沿用了Linux中“Everything is a file”的设计），包括这个**文件的类型**（是普通文件还是目录），**名称**；如果它是一个普通文件，则它的**文件内容**、**文件大小**也值得我们关注；如果它是一个目录，则它有哪些**子节点**也值得我们关注。这里我们给出一个可供参考的数据结构，它对应了上文中我们关注的文件信息，你可以试着自己理解这些字段的含义，**也可以根据自己的需要设计自己的数据结构**：

```
typedef struct node {
    enum { FILE_NODE, DIR_NODE } type;
    struct node *dirents;
    void *content;
    int nrde;
    int size;
    char *name;
} node;
```

文件描述符

在Linux系统中，当你打开一个文件，就会得到这个文件的一个文件描述符（File Descriptor，它是一个int类型的**非负整数**），用于对文件进行读写。对于文件描述符而言，它重要的属性包含：**读写性质**（支持对文件进行的操作，例如只读、只写等）、**偏移量**、**对应的文件**，这个类型我们已经为你声明在了下面的代码中，并在代码中定义了这样一个文件描述符的数组（这个文件系统中最多可以同时打开4096个文件描述符），你可以根据你的需要修改其结构。

```
typedef struct FD {
  bool used;
  int offset;
  int flags;
  node *f;
} FD;

#define NRFD 4096
FD fdesc[NRFD];
```

接下来我们将对**读写性质**、**偏移量**做出进一步的说明。

偏移量 (offset)

想象你用手指指着读一本书，offset 相当于你手指指向的位置。你每读一个字，手指就向前前进一个字；如果你想改写书本上的字，每改写一个字，手指也向前前进一个字。

每一个文件描述符都拥有一个偏移量，用来指示读和写操作的开始位置。这个偏移量对应的是文件描述符，而不是“文件”对象。也就是说如果两次打开同一个文件，你将得到两个不同的文件描述符，它们之间的偏移量相互独立，具体而言，你可以考虑下面的代码：

```
// 假设1.txt文件中的内容是helloworld
char buf[10];
int fd1 = open("/1.txt", O_RDONLY);
int fd2 = open("/1.txt", O_RDONLY);
read(fd1, buf, 6); //从fd1中读取6个字节存储到buf中
read(fd2, buf, 6); //从fd2中读取6个字节存储到buf中
// 两次读取的结果应该是相同的
```

需要注意的是在读取过程中，偏移量可以超过原文件的大小 (size)，即指到文件末尾之后的位置。但是一旦开始在文件末尾之后的位置开始写入，你就需要将文件进行扩容，并且用 `\0` 填充中间的间隙。

读写性质

如果你阅读了ramfs.h，你会发现其中存在这些常量：

```
#define O_APPEND 02000
#define O_CREAT 0100
#define O_TRUNC 01000
#define O_RDONLY 00
#define O_WRONLY 01
#define O_RDWR 02
```

这些标志常量就是用来指示读写性质的，其中以0开头的数字在C语言中表示8进制，它们的含义如下：

- O_APPEND (02000): 以追加模式打开文件，即打开文件后，文件描述符的偏移量指向文件的末尾，若不含此标志，则指向文件的开头；如果该标志单独出现默认可读
- O_CREAT (0100): 如果传入的文件路径不存在，就创建这个文件，但如果这个文件的父目录不存在，就创建失败；如果文件已存在就正常打开
- O_TRUNC (01000): 如果传入的文件路径是存在的文件，并且同时还带有可写（O_WRONLY和O_RDWR）的标志，就清空这个文件
- O_RDONLY (00): 以只读的方式打开文件
- O_WRONLY (01): 以只写的方式打开文件
- O_RDWR (02): 以可读可写的方式打开文件

聪明的你在阅读的过程中应该发现：前三个标志是可以和其他标志进行结合的，而它们在二进制表示下，1的位置总是不同的。很自然的，你会联想到它们可以通过**按位或**运算进行结合，而事实也正是如此。然而，有的标志之间的组合在语义上是矛盾的，我们将对这些组合进行说明：

- O_TRUNC | O_RDONLY在Linux系统中是Unspecified行为，在本次项目中我们约定此标志组合的行为为正常只读打开，而不清空文件
- O_RDWR | O_WRONLY取只写的语义
- 如果传入的参数没有指定任何标志，则默认只读（这与O_RDONLY取值为0也有关系）
- O_RDONLY | O_WRONLY取只写的语义

init_ramfs

该函数为RAM File System的初始化函数，你可以在其中**自由完成**对内存文件系统的初始化，比如创建根目录等，我们的测试代码中会最先调用此函数。

find

该函数为辅助函数，用于寻找pathname对应的文件节点，由于在Shell中也需要使用，所以提前为同学们声明，但它不对应任何文档，你可以**自由实现它的内容，但要方便你完成下面各函数的实现。**

run_link

该函数对应于前文提到的`unlink`函数，在Linux系统下你可以利用命令`man 2 unlink`查看其文档。

你可以自行搜索学习如何在该界面下浏览文档内容，也可以通过重定向`man 2 unlink > unlink.txt`并将unlink.txt文件复制到图形界面下更好地阅读，也可以在互联网上搜索相关文档（但你无法保证互联网上的描述是正确的，同理，请尽可能阅读手册原文而不要阅读翻译）。这段话在之后的函数中将不再赘述。

你需要实现的内容包含：仅文档中有关文件的部分，不需要考虑是否有文件描述符正在使用该文件，测试数据**保证不会再使用和已经被删去的文件相关的文件描述符。**请特别关注返回值描述，并对不合理的输入做合适的处理，具体而言，你需要针对文档中`EISDIR`和`ENOENT`（不需要处理dangling symbolic link，该内存文件系统中也不存在这个东西）的描述进行错误处理。

- 首先，你需要根据传入的绝对路径参数在你的文件系统中找到这个文件节点，如果无法找到，说明这个路径对应的文件节点不存在，你应该直接返回-1，对应上文的`ENOENT`
- 接下来，你需要判断找到的文件节点是否是**常规文件**，如果它是一个**目录文件**，则无法被这个函数删去，你应该直接返回-1，对应上文的`EISDIR`
- 在上述错误均不存在的情况下，你需要找到这个节点的父级节点，将这个节点从其子节点数组中删去，**请特别注意资源的回收**

rrmdir

该函数对应于前文提到的`rmdir`函数，在Linux系统下你可以利用命令`man 2 rmdir`查看其文档。

你需要实现的内容包含：文档中的完整描述（就一句话）。请特别关注返回值描述，并对不合理的输入做合适的处理，具体而言，你需要针对文档中`ENONENT`，`ENOTDIR`，`ENOTEMPTY`，`EACCESS`（因为我们的文件系统中并不包含权限，所以这个错误只用考虑一个不能被删除的特殊情况）的描述进行错误处理。

- 首先，你需要根据传入的绝对路径参数在你的文件系统中找到这个文件节点，如果无法找到，说明这个路径对应的文件节点不存在，你应该直接返回-1，对应上文的`ENOENT`
- 接下来，你需要判断这个文件节点是否是**目录文件**，如果它是一个**常规文件**，则无法被这个函数删去，你应该直接返回-1，对应上文的`ENOTDIR`
- 紧接着，你需要判断这个文件节点是否是特殊的不可删去的目录（根目录），如果是，则无法被这个函数删去，你应该直接返回-1，对应上文的`EACCESS`
- 最后，你需要判断这个文件节点的子节点数量是否为0，如果不是，则无法被这个函数删去，你应该直接返回-1，对应上文的`ENOTEMPTY`
- 在上述错误均不存在的情况下，你需要找到这个节点的父级节点，将这个节点从其子节点数组中删去，**请特别注意资源的回收**

rmkdir

该函数对应于前文提到的`mkdir`函数，在Linux系统下你可以利用命令`man 2 mkdir`查看其文档。

你需要实现的内容包含：仅文档中的第一句描述，因为我们的内存文件系统并未引入权限问题。请特别关注返回值描述，并对不合理的输入做合适的处理，具体而言，你需要针对文档中`EEXIST`，`EINVAL`，`ENOENT`，`ENOTDIR`(前一个，因为我们的文件系统未引入相对路径)和`ENAMETOOLONG`。

- 首先，你需要根据传入的绝对路径参数在你的文件系统中尝试找寻这个文件节点，如果能够找到，说明它已经存在，无法继续创建，你应该直接返回-1，对应上文的`EEXIST`
- 接下来，你需要尝试找到这个绝对路径对应的父级节点，如果无法找到，说明父级目录不存在，无法继续创建，你应该直接返回-1，对应上文的`ENOENT`
- 紧接着，你需要检查找到的父级节点是否是一个**目录节点**，如果不是，则无法继续创建，你应该直接返回-1，对应上文的`ENOTDIR`
- 最后，你需要检查要创建的目录的basename是否合规（参看上文对文件系统的描述，注意有**两种错误**），如果不合规，则无法继续创建，你应该直接返回-1，对应上文的`EINVAL`和`ENAMETOOLONG`
- 在上述错误均不存在的情况下，你需要在父级目录节点中加入这个新的目录文件节点

ropen

该函数对应于前文提到的`open`函数，在Linux系统下你可以利用命令`man 2 open`查看其文档。

你需要实现的内容包含：文档中的前两段描述和返回值描述，关于标志位部分的内容，前文已经为你总结了。你需要对不合理的输入做合适的处理，具体而言，你需要针对文档中`ENFILE`，`ENONENT`（前两个）的描述进行错误处理，请注意，为了简洁起见，你不需要对`EISDIR`进行处理，测试数据保证不会出现此类情况，**即不会在打开一个目录时携带不合法的标志位，没有保证数据中不会打开一个目录。对于打开目录得到的描述符，对它的读写操作都应该出错**（见后文）。进一步，如果这个函数需要创建文件，你需要参考上面的`rmkdir`对`EINVAL`，`ENOENT`，`ENOTDIR`，`ENAMETOOLONG`进行处理，注意我们允许重复创建一个**常规文件**，尽管第二次创建时**什么也不会发生**。

为了简化大家实现此过程，我们约定数据中可能存在的flag组合包括：

- O_CREAT：仅创建
- O_RDONLY: 仅可读
- O_WRONLY: 仅可写
- O_RDWR：可读写
- O_CREAT | O_RDWR：创建与读写
- O_CREAT | O_RDONLY：创建与只读
- O_CREAT | O_WRONLY：创建与只写
- O_CREAT | O_RDWR | O_WRONLY：创建与只写（见前文说明）
- O_APPEND | O_RDWR：追加可读写（**追加时只需要在open时设置offset即可，之后的write不需要再设置offset，可以参考样例4**）
- O_APPEND | O_WRONLY：追加与只写
- O_TRUNC | O_WRONLY | O_RDWR：覆盖与只写（见前文说明）

- 首先，你需要根据传入的绝对路径参数在你的文件系统中找到这个文件节点，如果无法找到，说明这个路径对应的文件节点不存在，你应该直接返回-1，对应上文的`ENOENT`
- 接下来，你需要尝试为这个文件分配一个**文件描述符**，按照标志位(flag)的规定执行对应的操作，如果无法分配文件描述符，则返回-1，对应上文的`ENFILE`（如何分配？请结合下文进行思考）
- 进一步，如果标志位中需要创建文件，请参考上面的`rmkdir`对`EINVAL`，`ENOENT`，`ENOTDIR`，`ENAMETOOLONG`进行处理，注意我们允许重复创建一个**常规文件**，尽管第二次创建时**什么也不会发生**。
- 在上述错误均不存在的情况下，返回上文中分配的文件描述符

rseek

该函数对应于前文提到的`lseek`函数，在Linux系统下你可以利用命令`man 2 lseek`查看其文档。

你需要实现的内容包含：文档中的第一段描述，即只需要考虑前三种whence，项目框架已经为你预设了这三种whence的常量值，你可以在ramfs.h文件中找到它们的定义。虽然测试数据保证输入不会包含无效的whence，但你仍然需要对不合理的输入做合适的处理，具体而言，你只需要处理`EINVAL`错误，其余情况返回0即可。

在实现这个函数之前，请思考rseek的第一个参数，即文件描述符的类型为什么是int，你要如何通过它找到对应的文件描述符？ropen的返回值是什么？它们之间有什么联系？

三种可能的whence包括：

- SEEK_SET: 将偏移量设置为offset
- SEEK_CUR: 将偏移量加上对应的offset
- SEEK_END: 将偏移量设置为文件的size + offset

rread

该函数对应于前文提到的`read`函数，在Linux系统下你可以利用命令`man 2 read`查看其文档。

你需要实现的内容包含：文档中的前两段描述，即不需要考虑读取0个字节的情况。请特别关注返回值描述，并对不合理的输入做合适的处理，具体而言，你需要针对文档中`EBADF`和`EISDIR`进行处理，请注意你的程序应该具备足够的鲁棒性。

- 首先，你需要检查文件描述符是否合法，其标志位是否包含可读的选项，如果上述条件有一个不满足，则无法继续读取，你应该直接返回-1，对应上文的`EBADF`
- 然后，你需要检查此文件描述符对应的文件是否是一个目录文件，如果是，则无法继续读取，你应该直接返回-1，对应上文的`EISDIR`
- 在上述错误均不存在的情况下，你需要执行读取文件的逻辑，注意添加文件描述符的偏移量

rwrite

该函数对应于前文提到的`write`函数，在Linux系统下你可以利用命令`man 2 write`查看其文档。

你需要实现的内容包含：文档中的前三段描述和返回值描述中的前两段。同时，你还需要对不合理的输入做合适的处理，具体而言，你需要针对文档中`EBADF`和`EISDIR`（文档中并没有具体的描述，按照read类似地处理即可）进行处理。

按照read类似地处理即可

rclose

该函数对应于前文提到的`close`函数，在Linux系统下你可以利用命令`man 2 close`查看其文档。

你需要实现的内容包含：将当前文件描述符标记为未使用即可，但是你需要注意已经被标记未使用的文件描述符再次被使用时会发生什么？（你不妨使用Linux提供的接口尝试一下）同时，你需要针对文档中的`EBADF`描述进行错误处理。

- 你只需要检查当前文件描述符是否合法，如果不合法，返回-1即可，对应上文的`EBADF`
- 否则将其标记为未使用即可

close_ramfs

该函数功能为回收内存文件系统，你可以自由完成其实现，但你需要保证在函数结束后，内存文件系统的空间已经被回收，并且root（根目录指针）指向空，同样的，在其他代码中请保证**你回收了不再需要使用的内存。**

Shell部分

这一部分的内容并不困难，但可以帮助你理解Shell是如何通过文件系统的api为用户提供服务的。出于工作量考虑，本次并没有引入交互式的Shell，避免了大家需要进一步进行输入解析等工作，但你可以尝试在项目结束后为它添加这个功能，以获得更真实的体验。

同时，由于下列命令都是非常常用且简单的命令，你只需要实现其最基础的功能，不用考虑额外参数，请在互联网上搜索它们的相关手册或使用方法，文档中将不再引导如何实现这些函数。

环境变量

此次项目中我们仅讨论`PATH`环境变量，它是一个类似于链表的结构，它的每一个节点都由一条指向目录的绝对路径组成，当用户在Shell中输入一个既不是绝对路径也不是相对路径，即不以`/`或`./`开头的命令时，Shell会在这个链表中逐一搜索，查看节点对应的绝对路径目录下是否存在相同名称的可执行文件，如果找到，就停止搜索，并执行对应的可执行文件。例如，ls命令实际上执行的是/usr/bin/ls。

在本次项目中，你的Shell需要先读取/home/ubuntu/.bashrc文件（即Shell的配置文件），识别并保存其中的内容。值得指出的是，Shell的配置文件语法并不完全相同，内容也并不只包含对环境变量的设置，感兴趣的同学可以自行研究。这里我们以bash的配置文件语法为标准，并保证.bashrc文件中仅包含对`PATH`的设置，其设置语法如下：

```
export PATH=/path/to
```

表示将PATH设置为`/path/to`，数据保证此设置至少在文件的开头出现一次。

```
export PATH=$PATH:/path/to/
```

该句表示在链表的尾部添加`/path/to/`节点，`$PATH`表示取PATH中实际保存的值。

```
export PATH=/path//to/:$PATH
```

该句表示在链表的首部添加`/path//to/`节点。

实际上`PATH`是一串由**绝对路径**组成，使用`:`进行分隔的符号串。

init_shell

在此函数中你需要完成对环境变量的读取与保存。我们的测试代码会在调用任何shell函数之前调用这个函数。

close_shell

与close_ramfs类似，你需要回收shell不再使用的内存。我们的测试代码会在不再调用shell函数之后，在close_ramfs之前调用这个函数。

sls

> 从此函数开始，在本地测试时，每个函数都会在开头以红色字体输出命令的具体内容，并将字体切换回黑色，便于你进行调试。同时你需要关注函数的返回值，具体如何返回请参考文档。

对应于`ls`命令，你可以使用命令`man ls`查看其文档，**我们保证传入参数仅含一个绝对地址**（如果传入的字符串是一个空串该怎么办？你不妨ls试一下，由于我们没有引入cd命令，所以shell的工作目录总是根目录），你需要打印命令执行的结果

具体而言，如果这个绝对地址中包含了不存在的文件/目录，你需要按如下格式输出错误信息`ls: cannot access '%s': No such file or directory\n`；如果这个绝对地址中包含了一个文件而非目录，例如/home/file/directory（其中file是一个文件），你需要按如下格式输出错误信息`ls: cannot access '%s': Not a directory\n`；否则，你需要输出这个文件/目录下的所有（子）文件，用空格隔开，不需要在意输出的顺序

scat

对应于`cat`命令，你可以使用命令`man cat`查看其文档，**我们保证传入参数仅含一个绝对地址**，你需要打印命令执行的结果

具体而言，如果这个绝对地址中包含了不存在的文件/目录，你需要按如下格式输出错误信息`cat: %s: No such file or directory\n`；如果这个绝对地址中包含了一个文件而非目录，例如/home/file/directory（其中file是一个文件），你需要按如下格式输出错误信息`cat: %s: Not a directory\n`；如果这个绝对地址最终指向一个目录，你需要按如下格式输出错误信息`cat: %s: Is a directory\n`。否则，你需要输出这个文件**所有**的内容（如果文件中途包含一个\0怎么办？）

smkdir

对应于`mkdir`命令，你可以使用命令`man mkdir`查看其文档，**我们保证传入参数仅含一个绝对地址**，你需要打印命令执行的结果

具体而言，如果这个绝对地址中包含了不存在的文件/目录，你需要按如下格式输出错误信息`mkdir: cannot create directory '%s': No such file or directory\n`；如果这个绝对地址中包含了一个文件而非目录，例如/home/file/directory（其中file是一个文件），你需要按如下格式输出错误信息`mkdir: cannot create directory '%s': Not a directory\n`；如果这个绝对地址最终指向的文件/目录已经存在，你需要按如下格式输出错误信息`mkdir: cannot create directory '%s': File exists\n`。否则，你需要在指定位置创建一个目录，不需要打印输出，正确地返回即可。

stouch

对应于`touch`命令，你可以使用命令`man touch`查看其文档，**我们保证传入参数仅含一个绝对地址**，你需要打印命令执行的结果，如果输入不合法，你的结果应该为`No such file or directory`。

具体而言，如果这个绝对地址中包含了不存在的文件/目录，你需要按如下格式输出错误信息`touch: cannot touch '%s': No such file or directory\n`；如果这个绝对地址中包含了一个文件而非目录，例如/home/file/directory（其中file是一个文件），你需要按如下格式输出错误信息`touch: cannot touch '%s': Not a directory\n`。如果这个绝对地址最终指向的文件/目录已经存在，你不妨尝试一下linux系统会怎么做。否则，你需要在指定位置创建一个文件，不需要打印输出，正确地返回即可。

secho

对应于`echo`命令，你可以使用命令`man echo`查看其文档，**我们保证传入参数仅含一个字符串**，该字符串只包含字母，数字，“$”和"\"，你需要打印命令执行的结果，提示：`echo`是如何处理环境变量和转义的？不妨自己试试看😊。

swhich

对应于`which`命令，你可以使用命令`man which`查看其文档，**我们保证传入参数仅含一个字符串**，该字符串只包含字母，数字和"."，你需要打印命令执行的结果，关于函数的返回值，你可以尝试自己使用which命令定位存在和不存在的命令，你可以使用`echo $?`命令查看上一条命令的返回值（linux的文档中也有相应的描述，**这个方法适用于所有你需要实现的shell函数**）。

运行/测试说明

MacOS/Linux 用户

你可以在main.c中编写测试代码，并通过`make run`命令运行测试。

Windows 用户

你可以在main.c中编写测试代码。如果你使用CLion，则直接加载这个项目并运行即可，需要注意CLion内置的运行界面并不支持显示`ANSI`码，因此可能会在你的运行结果中出现乱码，为了避免这个问题，你可以在运行后停止程序，然后手动在其内嵌终端中输入`./cmake-build-debug/ramfs-shell.exe`来运行程序；或者（不需要使用CLion）你也可以使用`./compile.ps1`运行编译脚本，然后运行`./ramfs-shell.exe`。

提交说明

数据约定

本题一共由15个测试用例组成，其中0-9为内存文件系统部分的测试，10-14为Shell部分的测试，其中第0个测试点和第14个测试点为诚信测试，即你几乎什么都不用干（第14个测试点需要你对不含$的字符串完成echo操作）就可以得分。

内存文件系统部分

整个文件系统同时存在的所有文件内容不会超过 512 MiB（不含已经删去的文件和数据），给予 1GiB 的内存限制。 同时存在的文件与目录不会超过 65536 个。 同时活跃着的文件描述符不会超过 4096 个。

对于所有数据点，文件操作读写的总字节数不会超过 10GiB。时限将给到一个非常可观的量级。错误将会分散在各个数据点中，你需要保证你的 API 能正确地判断错误的情况并按照要求的返回值退出。各数据点的性质：

1. 参考样例2
2. 根目录下少量文件创建 + ropen + rwrite + rclose
3. 在 2 的基础上，测试 O_APPEND，rseek
4. 在 3 的基础上扩大规模
5. 少量子目录创建（<= 5 层）+ 文件创建与随机读写
6. 在 5 的基础上，测试 rrmdir, runlink。
7. 大文件测试。多 fd 对少量大文件大量读写 + rseek + O_TRUNCATE
8. 复杂的文件树结构测试。大量的 O_CREAT，rmkdir, rrmdir, runlink。少量读写
9. 文件描述符管理测试。大量 ropen、rclose，多 fd 单文件

Shell部分

数据规模沿用内存文件系统部分的说明，下面对各测试点的性质进行说明：

10. 多层目录和文件的混合创建（smkdir <= 3 层），以及ls命令的实现（你不需要考虑ls的输出顺序，we have special judge）
11. 创建文件，读写文件，环境变量综合测试
12. 在11的基础上加强对环境变量的拷打（注意关注export语法新增了一条规则）
13. 集中测试各种错误的处理是否正确

提交方式

MacOS/Linux 用户

你可以在OJ平台上点击提交代码获取TOKEN，然后在Makefile中找到submit目标，在第四行添加（用你获取到的token替换`${your token}`）：

```
	$(eval TOKEN := ${your token})
```

然后在终端输入`make submit`即可提交。

或者你也可以使用下面的命令进行提交：

```
make submit TOKEN=${your token}
```

Windows 用户

你可以打包压缩项目文件夹并使用GUI进行提交。

测试样例

样例1

由于shell在文件系统的基础之上，所以放出较全面的文件系统样例：

main.c

```
#include "ramfs.h"
#include "shell.h"
#include <string.h>
#include <stdlib.h>
#include <assert.h>

int notin(int fd, int *fds, int n) {
    for (int i = 0; i < n; i++) {
    if (fds[i] == fd) return 0;
    }
    return 1;
}
int genfd(int *fds, int n) {
    for (int i = 0; i < 4096; i++) {
    if (notin(i, fds, n))
    return i;
    }
    return -1;
}
int main() {
    init_ramfs();
    int fd[10];
    int buf[10];
    assert(ropen("/abc==d", O_CREAT) == -1);
    assert((fd[0] = ropen("/0", O_RDONLY)) == -1);
    assert((fd[0] = ropen("/0", O_CREAT | O_WRONLY)) >= 0);
    assert((fd[1] = ropen("/1", O_CREAT | O_WRONLY)) >= 0);
    assert((fd[2] = ropen("/2", O_CREAT | O_WRONLY)) >= 0);
    assert((fd[3] = ropen("/3", O_CREAT | O_WRONLY)) >= 0);
    assert(rread(fd[0], buf, 1) == -1);
    assert(rread(fd[1], buf, 1) == -1);
    assert(rread(fd[2], buf, 1) == -1);
    assert(rread(fd[3], buf, 1) == -1);
    for (int i = 0; i < 100; i++) {
      assert(rwrite(fd[0], "\0\0\0\0\0", 5) == 5);
      assert(rwrite(fd[1], "hello", 5) == 5);
      assert(rwrite(fd[2], "world", 5) == 5);
```

```
      assert(rwrite(fd[3], "\x001\x002\x003\x0fe\x0ff", 5) == 5);
    }
    assert(rclose(fd[0]) == 0);
    assert(rclose(fd[1]) == 0);
    assert(rclose(fd[2]) == 0);
    assert(rclose(fd[3]) == 0);
    assert(rclose(genfd(fd, 4)) == -1);
    assert((fd[0] = ropen("/0", O_CREAT | O_RDONLY)) >= 0);
    assert((fd[1] = ropen("/1", O_CREAT | O_RDONLY)) >= 0);
    assert((fd[2] = ropen("/2", O_CREAT | O_RDONLY)) >= 0);
    assert((fd[3] = ropen("/3", O_CREAT | O_RDONLY)) >= 0);
    assert(rwrite(fd[0], buf, 1) == -1);
    assert(rwrite(fd[1], buf, 1) == -1);
    assert(rwrite(fd[2], buf, 1) == -1);
    assert(rwrite(fd[3], buf, 1) == -1);
    for (int i = 0; i < 50; i++) {
      assert(rread(fd[0], buf, 10) == 10);
      assert(memcmp(buf, "\0\0\0\0\0\0\0\0\0\0", 10) == 0);
      assert(rread(fd[1], buf, 10) == 10);
      assert(memcmp(buf, "hellohello", 10) == 0);
      assert(rread(fd[2], buf, 10) == 10);
      assert(memcmp(buf, "worldworld", 10) == 0);
      assert(rread(fd[3], buf, 10) == 10);
      assert(memcmp(buf, "\x001\x002\x003\x0fe\x0ff\x001\x002\x003\x0fe\x0ff", 10) == 0);
    }
    assert(rread(fd[0], buf, 10) == 0);
    assert(rread(fd[1], buf, 10) == 0);
    assert(rread(fd[2], buf, 10) == 0);
    assert(rread(fd[3], buf, 10) == 0);
    assert(rclose(fd[0]) == 0);
    assert(rclose(fd[1]) == 0);
    assert(rclose(fd[2]) == 0);
    assert(rclose(fd[3]) == 0);
    return 0;
}
```

期望输出：

除了commit信息之外没有任何输出

样例2

综合测试

main.c:

```
#include "ramfs.h"
#include "shell.h"
#include <string.h>
#include <stdlib.h>
#include <assert.h>

const char *content = "export PATH=/usr/bin/\n";
const char *ct = "export PATH=/home:$PATH";
```

```
int main() {
  init_ramfs();

  assert(rmkdir("/home") == 0);
  assert(rmkdir("//home") == -1);
  assert(rmkdir("/test/1") == -1);
  assert(rmkdir("/home/ubuntu") == 0);
  assert(rmkdir("/usr") == 0);
  assert(rmkdir("/usr/bin") == 0);
  assert(rwrite(ropen("/home///ubuntu//.bashrc", O_CREAT | O_WRONLY), content, strlen(content))
== strlen(content));

  int fd = ropen("/home/ubuntu/.bashrc", O_RDONLY);
  char buf[105] = {0};

  assert(rread(fd, buf, 100) == strlen(content));
  assert(!strcmp(buf, content));
  assert(rwrite(ropen("/home////ubuntu//.bashrc", O_WRONLY | O_APPEND), ct, strlen(ct)) ==
strlen(ct));
  memset(buf, 0, sizeof(buf));
  assert(rread(fd, buf, 100) == strlen(ct));
  assert(!strcmp(buf, ct));
  assert(rseek(fd, 0, SEEK_SET) == 0);
  memset(buf, 0, sizeof(buf));
  assert(rread(fd, buf, 100) == strlen(content) + strlen(ct));
  char ans[205] = {0};
  strcat(ans, content);
  strcat(ans, ct);
  assert(!strcmp(buf, ans));

  fd = ropen("/home/ubuntu/text.txt", O_CREAT | O_RDWR);
  assert(rwrite(fd, "hello", 5) == 5);
  assert(rseek(fd, 7, SEEK_SET) == 7);
  assert(rwrite(fd, "world", 5) == 5);
  char buf2[100] = {0};
  assert(rseek(fd, 0, SEEK_SET) == 0);
  assert(rread(fd, buf2, 100) == 12);
  assert(!memcmp(buf2, "hello\0\0world", 12));

  init_shell();

  assert(scat("/home/ubuntu/.bashrc") == 0);
  assert(stouch("/home/ls") == 0);
  assert(stouch("/home///ls") == 0);
  assert(swhich("ls") == 0);
  assert(stouch("/usr/bin/ls") == 0);
  assert(swhich("ls") == 0);
  assert(secho("hello world\\n") == 0);
  assert(secho("\\$PATH is $PATH") == 0);

  close_shell();
  close_ramfs();
}
```

期望输出：

```
cat /home/ubuntu/.bashrc
export PATH=/usr/bin/
export PATH=/home:$PATH
touch /home/ls
touch /home///ls
which ls
/home/ls
touch /usr/bin/ls
which ls
/home/ls
echo hello world\n
hello worldn
echo \$PATH is $PATH
$PATH is /home:/usr/bin/
```

样例3

为方便大家了解shell的命令都应该如何报错，这里给出一个样例

main.c:

```
#include "ramfs.h"
#include "shell.h"
#include <string.h>
#include <stdlib.h>
#include <assert.h>

int main() {
  init_ramfs();
  init_shell();

  assert(sls("/home") == 1);
  assert(scat("/home/ubuntu/.bashrc") == 1);
  assert(scat("/") == 1);
  assert(smkdir("/home") == 0);
  assert(smkdir("/test/1") == 1);
  assert(stouch("/home/1") == 0);
  assert(smkdir("/home/1/1") == 1);
  assert(stouch("/test/1") == 1);
  assert(swhich("notexist") == 1);

  close_shell();
  close_ramfs();
}
```

期望输出:

```
ls /home
ls: cannot access '/home': No such file or directory
```

```
cat /home/ubuntu/.bashrc
cat: /home/ubuntu/.bashrc: No such file or directory
cat /
cat: /: Is a directory
mkdir /home
mkdir /test/1
mkdir: cannot create directory '/test/1': No such file or directory
touch /home/1
mkdir /home/1/1
mkdir: cannot create directory '/home/1/1': Not a directory
touch /test/1
touch: cannot touch '/test/1': No such file or directory
which notexist
```

样例4

这是为O_APPEND和O_TRUNC标志位准备的简单样例

main.c:

```
#include "ramfs.h"
#include "shell.h"
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

char s[105] = "Hello World!\n";
int main() {
    init_ramfs();
    init_shell();
    int fd1 = ropen("/test", O_CREAT | O_RDWR | O_APPEND);
    rwrite(fd1, s, strlen(s));
    rseek(fd1, 2, SEEK_SET);
    rwrite(fd1, s, strlen(s));

    scat("/test");

    int fd2 = ropen("/test", O_TRUNC | O_RDWR);

    scat("/test");
    rwrite(fd2, s, strlen(s));

    scat("/test");

    close_shell();
    close_ramfs();
    return 0;
}
```

期望输出:

```
cat /test
HeHello World!

cat /test

cat /test
Hello World!
```

样例5

由于之前写漏了basename长度的要求，以及有许多同学可能忘记了对目录的open操作，因此补充一个basename长度和open目录的样例

main.c:

```
#include "ramfs.h"
#include "shell.h"
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

#define test(func, expect, ...) assert(func(__VA_ARGS__) == expect)
#define succopen(var, ...) assert((var = ropen(__VA_ARGS__)) >= 0)
#define failopen(var, ...) assert((var = ropen(__VA_ARGS__)) == -1)

int main() {
    init_ramfs();
    int fd;
    test(rmkdir, -1, "/00000000000000000000000000000001");

    test(rmkdir, 0, "/it");
    test(rmkdir, 0, "/it/has");
    test(rmkdir, 0, "/it/has/been");
    test(rmkdir, 0, "/it/has/been/a");
    test(rmkdir, 0, "/it/has/been/a/long");

    succopen(fd, "/it/has/been/a/long", O_CREAT);
    failopen(fd, "it/has/been/a/long", O_CREAT);
    char buf[105];
    test(rread, -1, fd, buf, 100);
    test(rwrite, -1, fd, "a", 1);
    test(rrmdir, -1, "/it/has/been");
    test(rrmdir, 0, "/it/has/been/a/long");
    test(rwrite, -1, fd, "a", 1);
    test(rread, -1, fd, buf, 100);

    init_shell();
    close_shell();
    close_ramfs();
    return 0;
```

```
    }
```

期望输出:

除了commit信息之外没有任何输出