RAM File System Shell

张哲恺(corax@smail.nju.edu.cn) 冯亚林(191850036@smail.nju.edu.cn)

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，不妨通过卷首的邮箱与助教取得联系与帮助，请坚持独立自主完成本次作业😊，在本文档中提到了`自行`的部分，除非你已经花费了相当的精力进行搜索与研究仍未解决，否则不要轻易提问，这无益于你自己的能力提升😔。

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);
```

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。

Linux的文件系统结构采用了树形结构，具体描述如下：

初始状态下只存在根目录/，文件系统中存在两类对象：目录和文件。目录下可以存放其他对象，而文件不可以，即在Linux文件系统的树形结构中，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。

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

我们推荐你在Linux操作系统上完成本次项目。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）。

⚠如果你坚持使用$Windows$环境，那么你遇到的所有问题都将由你自己负责并解决！

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/KYCoraxxx/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，你的修改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
├── 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，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. 你正在阅读的是README.md（也有可能是它导出的pdf）；
4. main.c用于进行测试；
5. include文件夹中包含了所有的头文件；
6. 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；
7. 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

本引导旨在为完成本次项目有困难的同学提供思路，为了在完成项目过程中学习到更多知识与能力，你应该**尽可能减少直接阅读**此部分中 `攻略` （会在后续内容中进行标注）的内容，**而是先阅读每个函数的实现要求，然后阅读引导中标注的官方文档，识别出有哪些具体内容是需要你自己实现的，接下来自己实现，遇到了问题再阅读攻略内容**。

攻略的具体内容暂时不会更新，后期视同学们的完成情况放出(会尽可能让选题并认真对待的同学完成作业拿到分数)

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），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**。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）。各个函数文档的查看方式大同小异，此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（不用设置errno），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。

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（不用设置errno），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` （只用考虑一个特殊情况）的描述进行错误处理。

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，因为该内存文件系统并未引入权限问题。请特别关注返回值描述（不用设置errno），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，后一个是因为该文件系统未引入相对路径)。

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（不用设置errno，但正因如此，为了区分不同的错误，你应该对返回值做出一些与文档描述不同的处理，下文即将提到这一点），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`EEXIST`和`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。

> **更正**：由于O_EXCL不在此次项目考虑范围之内，所以你实际上不需要对EEXIST进行处理；此外，你需要对EINVAL（文件名不合法的条项，即检测是否出现了数字、字母、.之外的字符）进行处理

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。

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（不用设置errno），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。

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（不用设置errno）。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。

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不可用即可，但是你需要注意已经被标记不可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。

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。

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中输入一个非绝对路径或相对路径，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。

> **更正**：注意这里的语义是非(绝对路径或相对路径)

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：

> **更新**：添加了下面这个设置环境变量的语法

```
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` 是一串由绝对地址组成，使用 `:` 进行分隔的符号串。

init_shell

> **更正**：读一下这句话就知道了

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**，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` ，你可以尝试自己执行ls命令以查看对于不合法输入的具体返回内容，此后不再赘述。

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，如果输入不合法，你的结果应该为 `No such file or directory` 或 `Is a directory` 。

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，如果输入不合法，你的结果应该为 `No such file or directory` 或 `File exists` 。

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` 。

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（文档中也有相应的描述）。

选做部分*

在内存文件系统的基础之上，你可以给Shell添加更多的功能，以使它更加完善，但鉴于项目的难度和工作量，下面的方向留给有兴趣的同学继续自行探索😉

管道

重定向

ping

...

运行/测试说明

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

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. 如原始的 main.c
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

> **更正**：在你提交代码之前，请务必注意将shell.c的前几行修改为如下：

```
#include "ramfs.h"
#include "shell.h"
#ifndef ONLINE_JUDGE
  #define print(...) printf("\033[31m");printf(__VA_ARGS__);printf("\033[0m");
#else
  #define print(...)
#endif
```

推荐在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。

测试样例

样例1

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));

  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2

> 更新了数据点的性质，看看大家的情况再更新

main.c:

```
```

期望输出: