- 题面/讲义/api手册
- 提交方式
- 分数分布
- 测试点
- 代码框架
- 参考实现

前言

这可能是大家第一次做类似的工作：你不是编写一个完整的程序，而是按照要求实现一些函数。这些函数将会被我们调用，以检测是否实现了所要求的功能

题面

文件系统是操作系统的重要组成部分。调用文件系统 api，我们可以轻松地将数据持久化到磁盘上。C 语言中为我们提供了一组 api，它们基于操作系统 api，允许我们对文件进行操作：

```
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);
...
```

在 Linux 操作系统中，上述的 C 文件操作 api 是基于一组操作系统的文件 api 实现的：

```
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);
...
```

现要求你实现一个内存文件系统（ramfs）。顾名思义，这个文件系统的所有数据并不持久化到磁盘上，而是保存到内存当中，是一个易失性的文件管理系统。

文件系统的规定

与 Linux 的树形结构一致。在初始状态下，只存在根目录 "/"。文件系统中存在两类对象，目录与文件。目录下可以存放其他对象，而文件不可以。即在树形结构中，文件只能是叶子节点。

例 (#):

```
/
├── 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”。

单个文件和目录名长度 <= 32 字节，仅包含字母、数字、英文句点。对于存在不合法的文件名的路径，你的文件系统 api 应当统一通过返回 -1 来拒绝此类操作。

路径长度 <= 1024 字节。（变相地说，文件系统的路径深度存在上限）。

文件系统 api 统一使用绝对路径，即以 '/' 开头。在未创建任何文件时，即存在 "/" 指向的根目录。该目录可打开，不可删除，其余性质与一般目录一致。

我们要求你实现如下的 api，以实现文件系统的管理。其具体行为将会在 api 说明部分阐释。

```c
int ropen(const char *pathname, int flags);
int rclose(int fd);
ssize_t rwrite(int fd, const void *buf, size_t count);
ssize_t rread(int fd, void *buf, size_t count);
off_t rseek(int fd, off_t offset, int whence);
int rmkdir(const char *pathname);
int rrmdir(const char *pathname);
int runlink(const char *pathname);
```

注意，我们要求你实现的是内存操作系统。故你的程序应当使用内存管理 api（malloc、free）来存放文件所需的数据结构，以及文件的所有内容。我们保证整个文件系统的所有文件内容不会超过 512 MiB，且给予 1GiB 的内存限制。请小心地管理好内存注意不要超限。

开始你的项目

我们为你准备了一个 git repo。请基于这个 git repo 进行你的项目。如果你不会 git，请学着使用。

在 git repo 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。你的修改记录将成为查重时证明独立完成的重要证据。

推荐在 Linux 操作系统中完成本作业。如果你要使用 Windows，产生的问题由你自己解决。

获取代码框架：

```
git clone "https://git.nju.edu.cn/tilnel/ramfs.git"
```

注意：请在默认的 master 分支上进行开发。最终 OJ 的评分也将以你的 master 分支为准。

你应当在 ramfs.c 中包含你的所有实现。评测机会用我们自己的 Makefile（和分发版本一致）、ramfs.h（和分发版本一致）、main.c（包含更强力的测试用例）进行编译运行。因此你对 ranfs.h 和 main.c 以及 Makefile 的修改在 OJ 上不会产生效果。

提交：

```
make submit TOKEN=${你的token}
```

请在题目中“打开代码编辑器”后，获取你的提交 token。注意在校园网环境下提交。然后你就能在提交列表中看到你的提交。

注意在 make submit 之前，你需要将最新的改动 commit。同样注意保持你的工作目录整洁，如果你的 git repo 超过 20MiB（这一定是因为你放了很多很多奇怪的玩意），则没有办法提交。

你的 git repo 中不应当包含各种形式的编译产生的中间文件、编译结果。我们的 Makefile 只会在 build 目录下产生文件，我们也会配置好 .gitignore 文件避免 track 这些文件。

API 手册

注意两个对象的定义：文件（file），目录（directory）

最重要的对象：文件描述符 (file descriptor)

它是所有**打开的文件和目录的标志**，为一个非负整数。在 Windows 操作系统中称之为“句柄”。我们使用路径打开一个文件或目录，操作系统就会为这一次文件的打开分配一个文件描述符，它就像是一个“把手”一样。我们用这个文件描述符来指示打开的文件，进行对文件的操作。如：

```
int fd = open("/1.txt", O_RDONLY);  // open 返回一个文件描述符
read(fd, buf, 5);                   // 从打开的 fd (1.txt) 中读取五个字节
```

```
int ropen(const char *pathname, int flags);
```

打开 ramfs 中的文件。如果成功，返回一个文件描述符（一个非负整数），用于标识这个文件。

如果打开失败，则返回一个 -1。

pathname 为一个字符串，为文件的绝对路径。对于所有存在的文件和目录，你的 ropen 调用都应当成功。特别地，在指示一个目录时，pathname 的末尾可以有多余的 '/'。pathname 中同样可以有冗余的 '/'。

例如，在上文的例 (#) 中，以下的绝对路径是合法的：

```
//dir/        =/dir
////dir       =/dir
/1.txt        =/1.txt
//dir/1.txt   =/dir/1.txt
```

以下的绝对路径是不存在的。

```
/3.txt
/1.txt/       (文件路径后不可以有多余的'/')
/di/r/1.txt   (不存在这个路径)
```

flag 指示打开方式，其取值有如下可能（或可以是它们的组合）：

注意，在 C 中，以 0 开头的数字采用 8 进制表示法。

```
O_APPEND   02000 以追加模式打开文件。即打开后，文件描述符的偏移量指向文件的末尾。若无此标志，则指向文件的开头
O_CREAT    0100 如果 pathname 不存在，就创建这个文件，但如果这个目录中的父目录不存在，则创建失败；如果存在则正常打开
O_TRUNC    01000 如果 pathname 是一个存在的文件，并且同时以可写方式（O_WRONLY/O_RDWR）打开了文件，则文件内容被清空
O_RDONLY   00 以只读方式打开
O_WRONLY   01 以只写方式打开
O_RDWR     02 以可读可写方式打开
```

这些标志位的组合方式是使用按位的或运算。即：

```
O_TRUNC | O_RDWR    （可读可写，打开时清空）
O_CREAT | O_WRONLY（若不存在，创建后以读写方式打开；否则以读写方式直接打开）
O_APPEND    （文件描述符的偏移量指向文件末尾，但因为只有此一个标志位，既不可读也不可写）
```

```
int rclose(int fd);
```

关闭打开的文件描述符，并返回 0。如果不存在一个打开的 fd，则返回 -1。

```
ssize_t rwrite(int fd, const void *buf, size_t count);
```

向 fd 中的***偏移量***（马上解释）位置写入以 buf 开始的至多 count 字节，覆盖文件原有的数据（如果 count 超过 buf 的大小，仍继续写入），将 fd 的***偏移量***后移 count，并返回实际成功写入的字节数。如果写入的位置超过了原来的文件末尾，则自动为该文件扩容。

如果 fd 不是一个可写的文件描述符，或 fd 指向的是一个目录，则返回 -1。

在本实验中，ramfs 中同时存在的文件大小不会超过限制。因此你的 rwrite 对于一个能够写入的文件，事实上总应返回 count。

```
ssize_t rread(int fd, void *buf, size_t count);
```

从 fd 中的***偏移量***位置读出至多 count 字节到 buf 指向的内存空间当中，将***偏移量***后移 count，并返回实际读出的字节数。因为可能会读到文件末尾，因此返回值有可能小于 count。

如果 fd 不是一个可读的文件描述符，或 fd 指向的是一个目录，则返回 -1。

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。举个例子：

```
char buf[6];
int fd1 = open("/1.txt", O_WRONLY | O_CREAT);
int fd2 = open("/1.txt", O_RDONLY);
write(fd1, "helloworld", 11);
read(fd2, buf, 6);
```

此时 buf 中将从文件的开头读到字符串"hello\0"。但如果换一种方式：

假设 "/1.txt" 中原来有数据 "helloworld\0"

```
char buf[6];
int fd = open("/1.txt", O_RDWR);
write(fd, "hello", 5);
read(fd, buf, 6);
```

此时，write 在读取时，将文件指针前移了 5 个字节。于是read在读取的时候，将会从第6个字节开始读取。也即，read 将会读到 "world\0"。对于同一个文件描述符，读取和写入操作是共享偏移量的；对于不同的文件描述符，它们的偏移量则是各自独立的。

对于 open 操作，如果没有 O_APPEND 标志来将偏移量指向末尾，那么默认指向文件开头。

如何自由地修改和获取文件描述符的偏移量呢?

```
off_t rseek(int fd, off_t offset, int whence);
```

这个函数用于修改 fd 表示的文件描述符的偏移量，并返回当前文件的实际偏移量。

whence有三种取值：

```
SEEK_SET 0   将文件描述符的偏移量设置到 offset 指向的位置
SEEK_CUR 1   将文件描述符的偏移量设置到 当前位置 + offset 字节的位置
SEEK_END 2   将文件描述符的偏移量设置到 文件末尾 + offset 字节的位置
```

rseek 允许将偏移量设置到文件末尾之后的位置，但是并不会改变文件的大小，直到它在这个位置写入了数据。在 超过文件末尾的地方写入了数据后，原来的文件末尾到实际写入位置之间可能出现一个空隙，我们规定应当以 "\0" 填充这段空间。

```
int rmkdir(const char *pathname);
```

创建目录，成功则返回 0。如果目录的父目录不存在或此路径已经存在，则失败返回 -1。

如，原来系统中只存在根目录 "/"，调用：`rmkdir("/path/to/dir")` 返回 -1。

```
int rrmdir(const char *pathname);
```

删除一个空目录，成功则返回 0。如果目录不存在或不为空，或 pathname 指向的不是目录，返回 -1。

```
int runlink(const char *pathname);
```

删除一个文件，成功则返回 0。如果文件不存在或 pathname 指向的不是文件，则返回 -1。

额外的一个 api：

```
int init_ramfs();
```

可以用于初始化你的文件系统。比如创建根目录。我们用于测试的 main() 将总会包含它。（要在里面做什么取决于你自己！）

我们的测试用例长什么样：

```
/* our main.c */
#include "ramfs.h"
#include <assert.h>
#include <string.h>

int main() {
  init_ramfs();     // 你的初始化操作
  assert(rmkdir("/dir") == 0);   // 应当成功
  assert(rmkdir("//dir") == -1);  // 应当给出 error，因为目录已存在
  assert(rmkdir("/a/b") == -1);  // 应当给出 error，因为父目录不存在
  int fd;
  assert((fd = ropen("//dir///////1.txt", O_CREAT | O_RDWR)) > 0);  // 创建文件应当成功
  assert(rwrite(fd, "hello", 5) == 5);  // 应当完整地写入
  assert(rseek(fd, 0, SEEK_CUR) == 5);  // 当前 fd 的偏移量应该为 5
  assert(rseek(fd, 0, SEEK_SET) == 0);  // 应当成功将 fd 的偏移量复位到文件开头
  char buf[7];
  assert(rread(fd, buf, 7) == 5);   // 只能读到 5 字节，因为文件只有 5 字节
  assert(memcmp(buf, "hello", 5) == 0);   // rread 应当确实读到 "hello" 5 个字节
  assert(rseek(fd, 3, SEEK_END) == 8);  // 文件大小为 5，向后 3 字节则是在第 8 字节
  assert(rwrite(fd, "world", 5) == 5);  // 再写 5 字节
  assert(rseek(fd, 5, SEEK_SET) == 5);  // 将偏移量重设到 5 字节
  assert(rread(fd, buf, 8) == 8);   // 在第 8 字节后写入了 5 字节，文件大小 13 字节；那么从第 5 字节后应当能成功读到 8 字节
  assert(memcmp(buf, "\0\0\0world", 8) == 0); // 3 字节的空隙应当默认填 0
  assert(rclose(fd) == 0); // 关闭打开的文件应当成功
  assert(rclose(fd + 1) == -1); //关闭未打开的文件应当失败
  return 0;
}
```

我们将会在框架代码中提供几份测试代码供大家参考。大家可以自由地改动测试代码，并使用 `make test` 进行测试。