综合实验二-简易文件系统设计-实验草稿

实验目的

实验内容

实验环境

Linux或其他操作系统

实验步骤及说明

1. 理论部分

1）学习书本文件管理和磁盘存储器管理的相关章节，理解文件和目录的树状组织结构。

1. 编程部分

三个参考代码说明：

file\_system：创建100M的共享内存，并将其划分为4MB的文件系统管理和96MB的文件数据。进程同步时在这个文件创建信号量，信号量集的创建数量与inode信号量相同即可。

system\_core：实现基本功能接口，对功能进行实现和封装。

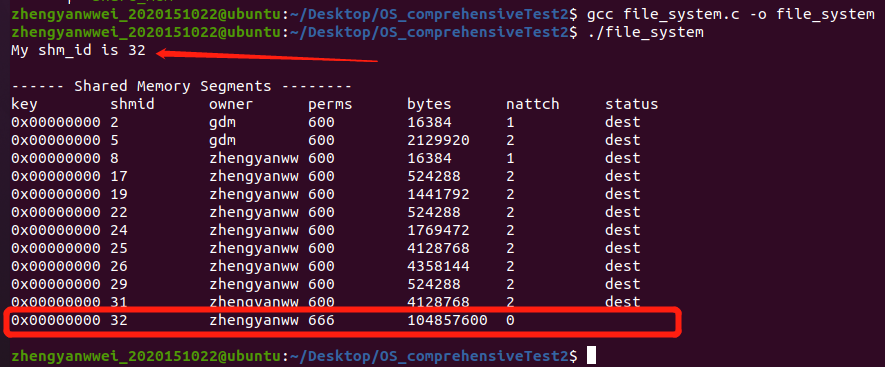
use\_my\_system：定义help

1. 创建一个100M的文件或者创建一个100M的共享内存。

创建100M的共享内存，并将其映射到进程空间

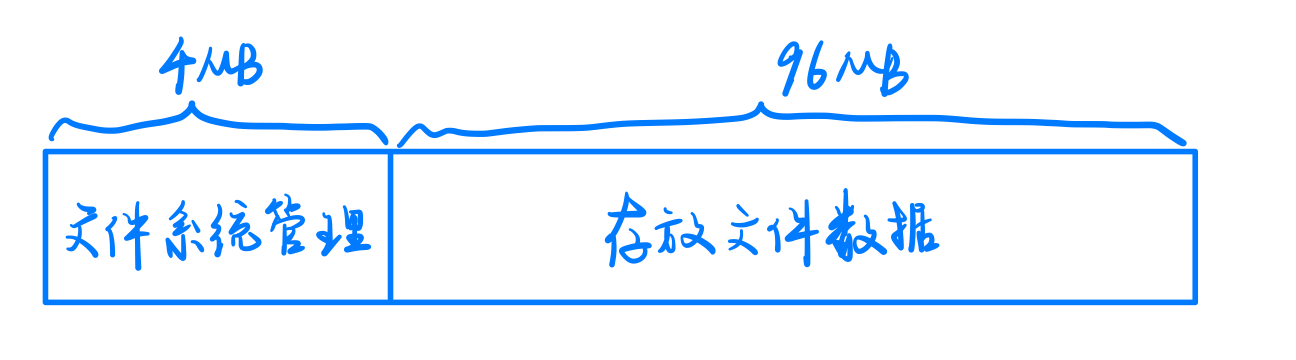
|  |
| --- |
| const size\_t MEM\_SIZE = 100 << 20 ; //100MB memories  int sem\_id;  int main(){  void \*share\_mem = NULL;  int shm\_id;  shm\_id = shmget(IPC\_PRIVATE, MEM\_SIZE, 0666); //获得共享内存id  printf("My shm\_id is %d\n",shm\_id);  system("ipcs -m");  share\_mem = shmat(shm\_id, 0, 0); //映射到进程空间  return 0;  } |

编译并运行一下上述程序，可以看到已经创建了一块大小为104857600bytes（100MB）的空间。



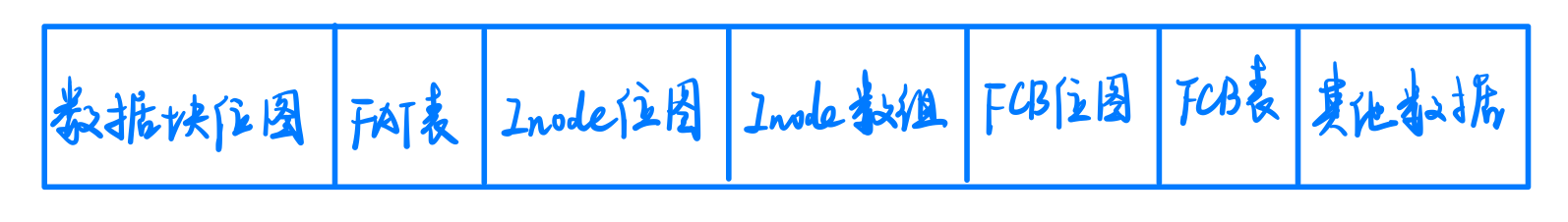
1. 尝试自行设计一个C语言小程序，使用步骤1分配的100M空间（mmap or 共享内存），然后假设这100M空间为一个空白磁盘，设计一个简单的文件系统管理这个空白磁盘，给出文件和目录管理的基本数据结构，并画出文件系统基本结构图，以及基本操作接口。

（1）对于步骤1中申请的100M共享内存，模拟FAT文件系统，首先从100M中拿出4MB作为文件管理系统的核心，存放位图、Inode数组等。



用于存放文件数据的96MB，按照4KB/页为单位划分，使用1bit对每一页是否被占用进行表示，这些信息会存放在4MB的文件系统管理中。

用于文件系统管理的4MB，其结构可以如下图所示：



其中，FAT表用于显示表达文件占用的数据页的跳转情况；Inode记录当前文件类型（包含目录和可读写文件）以及父节点、左右兄弟节点、子结点的inode编号以及对应的FCB编号；FCB记录文件名，每个文件和文件夹都占用一个inode，与inode一一对应。

1. 实验过程使用的常量声明：



//一个数据块的大小

#define BLOCK\_SIZE 4096

//96MB对应的24576个数据块

#define BLOCK\_NUM 24576

//每块对应1bit用于表示是否被占用，需要3072bytes

#define BLOCK\_MAP\_LENGTH 3072

//inode最多每块一个，对应数据块数量

#define INODE\_MAP\_LENGTH 3072

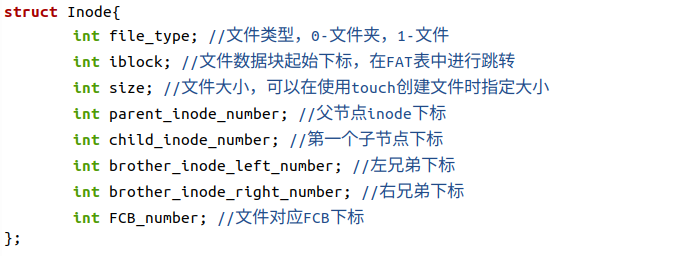
//考虑可能会有的软、硬链接，可以多开一倍

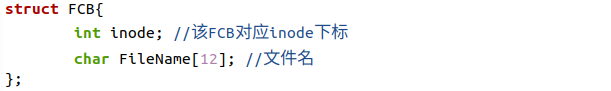
#define FCB\_MAP\_LENGTH 6144

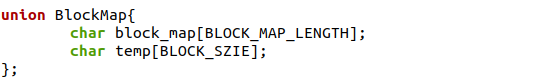
//系统管理核心占4MB，剩余96MB

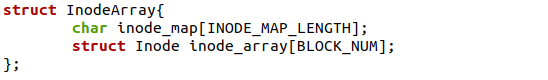
#define CORE\_SIZE (1<<22)

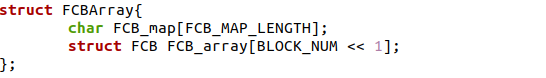
1. 用到的基本数据结构：

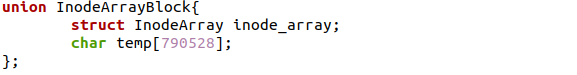


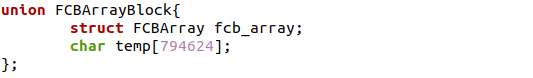


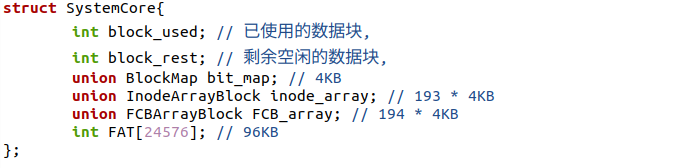


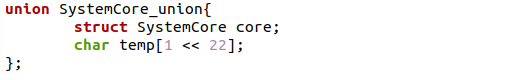


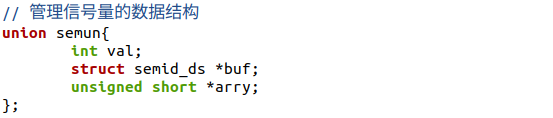












struct Inode{

int file\_type; //文件类型，0-文件夹，1-文件

int iblock; //文件数据块起始下标，在FAT表中进行跳转

int size; //文件大小，可以在使用touch创建文件时指定大小

int parent\_inode; //父节点inode下标

int child\_inode; //第一个子节点下标

int brother\_inode\_left; //左兄弟下标

int brother\_inode\_right; //右兄弟下标

int FCB\_num; //文件对应FCB下标

};

struct FCB{

int inode; //该FCB对应inode下标

char FileName[12]; //文件名

};

union BlockMap{

char block\_map[BLOCK\_MAP\_LENGTH];

char temp[BLOCK\_SZIE];

};

struct InodeArray{

char inode\_map[INODE\_MAP\_LENGTH];

struct Inode inode\_array[BLOCK\_NUM];

};

struct FCBArray{

char FCB\_map[FCB\_MAP\_LENGTH];

struct FCB FCB\_array[BLOCK\_NUM << 1];

};

union InodeArrayBlock{

struct InodeArray inode\_array;

char temp[790528];

};

union FCBArrayBlock{

struct FCBArray fcb\_array;

char temp[794624];

};

struct SystemCore{

int block\_used; // 已使用的数据块,

int block\_rest; // 剩余空闲的数据块,

union BlockMap bit\_map; // 4KB

union InodeArrayBlock inode\_array; // 193 \* 4KB

union FCBArrayBlock FCB\_array; // 194 \* 4KB

int FAT[24576]; // 96KB

};

union SystemCore\_union{

struct SystemCore core;

char temp[1 << 22];

};

// 管理信号量的数据结构

union semun{

int val;

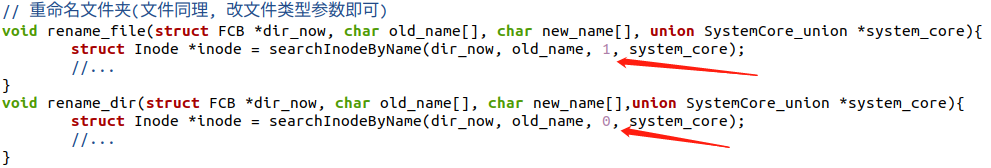
struct semid\_ds \*buf;

unsigned short \*arry;

};

1. 系统设置的一些基础功能接口：





//查看1byte即8bit中哪一位为0（0表示可用）

int searchBit(char byte){}

//查找一个位图中第一位为0的bit的下标（0表示可用）

int searchFirstBit(char \*bytes, int length){}

//设置一个位图中第pos位的bit

void setBit(char \*bytes, int pos, int val){}

//循环释放以iblock开始的数据块

void freeMemBlock(int iblock, union SystemCore\_union \*system\_core){}

//申请block\_num个数据块，自动连接FAT表，返回起始数据块下标

int allocMemBLock(int block\_num, union SystemCore\_union \*system\_core){}

//删除文件

void remove\_file(struct Inode \*inode, union SystemCore\_union \*system\_core){}

//根据文件名、文件大小创建文件

void touch(struct FCB \*dir\_now, char file\_name[], union SystemCore\_union \*system\_core, int file\_size){}

//在当前文件夹下创建名为dir\_name的文件夹

void mkdir(struct FCB \*dir\_now, char dir\_name[], union SystemCore\_union \*system\_core){}

//删除文件夹

void remove\_dir(struct Inode \*inode, union SystemCore\_union \*system\_core){}

//cd进入指定子目录

struct FCB\* cd(struct FCB \*dir\_new, char file\_name[], union SystemCore\_union \*system\_core){}

//列出当前文件夹下子目录和文件的信息（包括文件inode编号，文件类型，文件大小和文件名）

void ls(struct FCB \*dir\_now, union SystemCore\_union \*system\_core){}

//读取文件内容，嵌入data中

void readFile(struct Inode \*inode, union SystemCore\_union \*system\_core, char data[]){}

//把data中的内容写入文件对应的数据块

void writeFile(struct Inode \*inode, union SystemCore\_union \*system\_core, char data[]){}

// 重命名文件夹(文件同理, 改文件类型参数即可)

void rename\_file(struct FCB \*dir\_now, char old\_name[], char new\_name[], union SystemCore\_union \*system\_core){

struct Inode \*inode = searchInodeByName(dir\_now, old\_name, 1, system\_core);

//...

}

void rename\_dir(struct FCB \*dir\_now, char old\_name[], char new\_name[],union SystemCore\_union \*system\_core){

struct Inode \*inode = searchInodeByName(dir\_now, old\_name, 0, system\_core);

//...

}

1. 在步骤1的基础上实现部分文件操作接口操作，创建目录mkdir，删除目录rmdir，修改名称，创建文件open，修改文件。删除文件rm，查看文件系统目录结构ls。

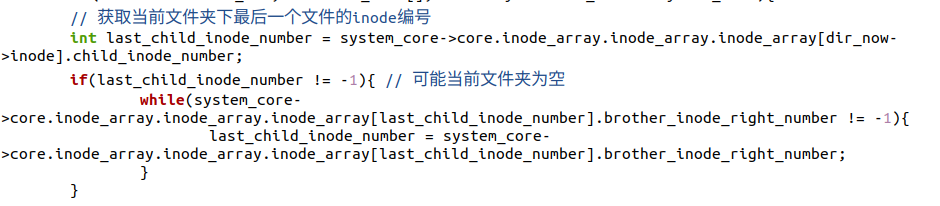
这里分成两部分进行说明：

* 1. **shell程序实现文件操作接口**

注明：全部在内存中实现。

1. 创建目录mkdir的实现

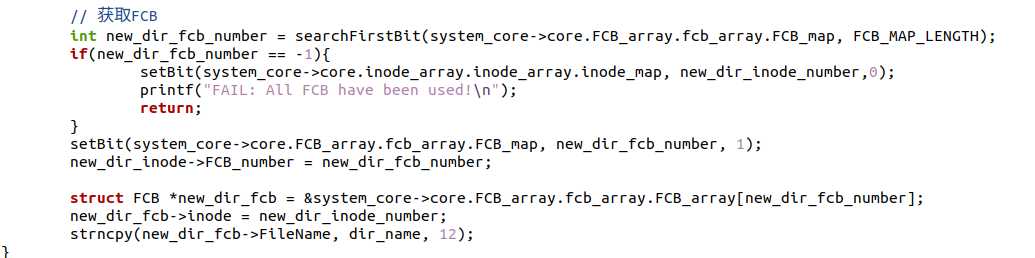
根据文件在文件管理系统中的存储特点，创建一个新的目录就需要分配对应的inode和FCB。首先设置循环获得当前文件夹下最后一个文件的inode编号。



然后为新文件夹获取可用的inode，根据上面得到的最后一个文件的inode编号对inode信息进行初始化。



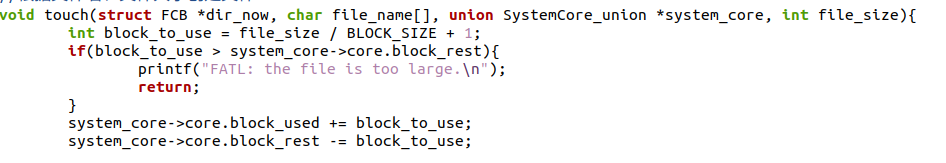
获取可用的FCB并初始化。



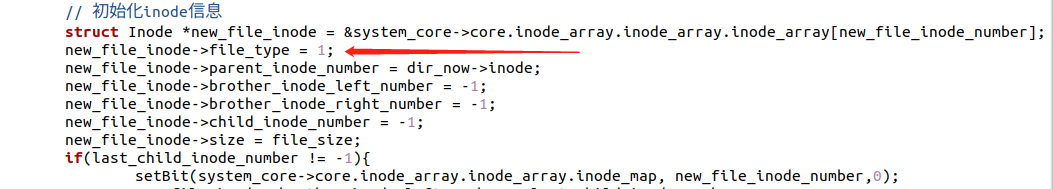
1. 根据文件名、文件大小创建文件touch的实现

与创建目录mkdir方法实现思路相同，主要是注意修改参数为文件对应的参数，下面仅展示不同的部分：

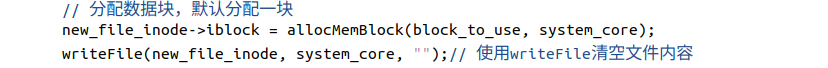
开始创建之前，判断当前剩余块是否够用。



初始化时，文件类型为1。

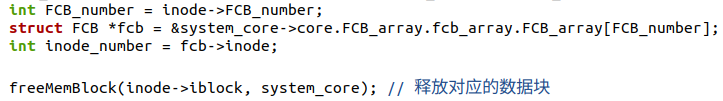


新文件装入当前文件夹后，分配数据块并清空。

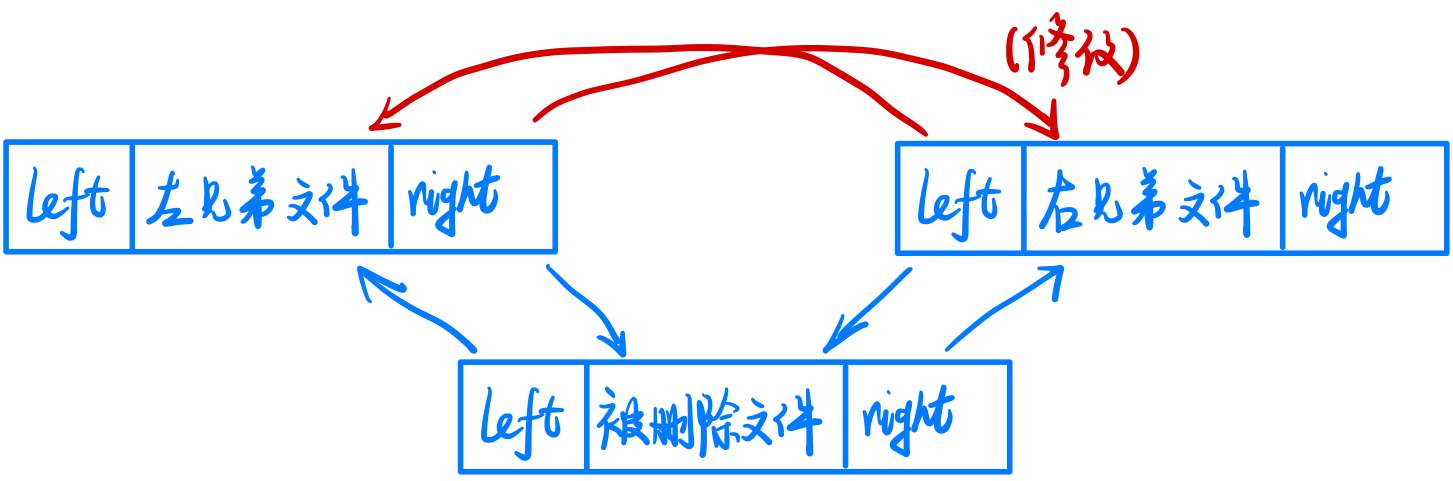
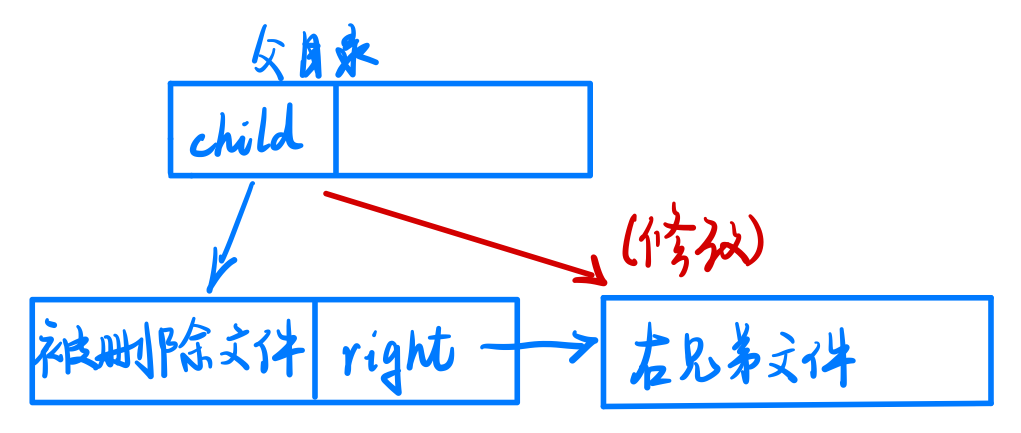


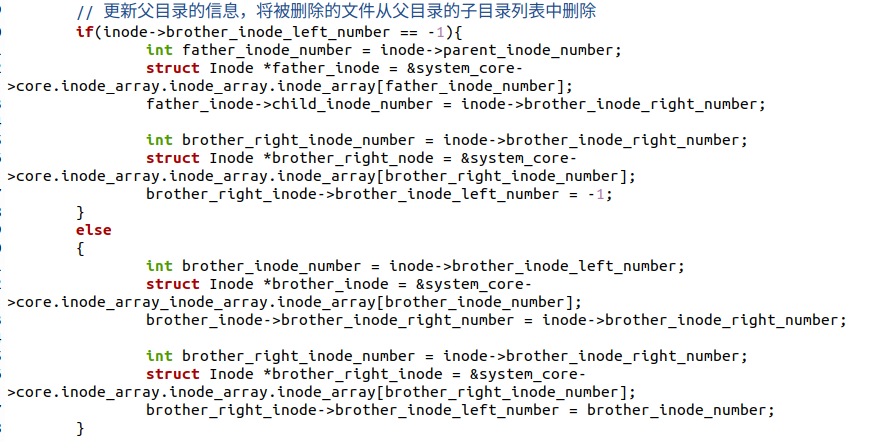
1. 删除文件rm的实现

借助位图和数据块的控制函数。首先从inode中获取FCB的编号和inode的编号，然后释放对应的数据块。

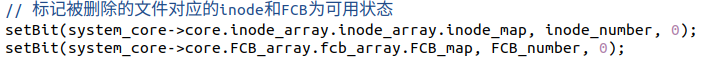


接着，根据被删除的文件的左右兄弟节点是否存在来更新父目录的信息：如果被删除的是父目录的第一个子目录，则修改父目录的第一个子节点为被删除的右兄弟的inode号；如果不是，则需要更新被删除文件左兄弟的右兄弟，把节点号改成被删除文件右兄弟的inode号，同时更新这个右兄弟的左兄弟为被删除文件的左兄弟。



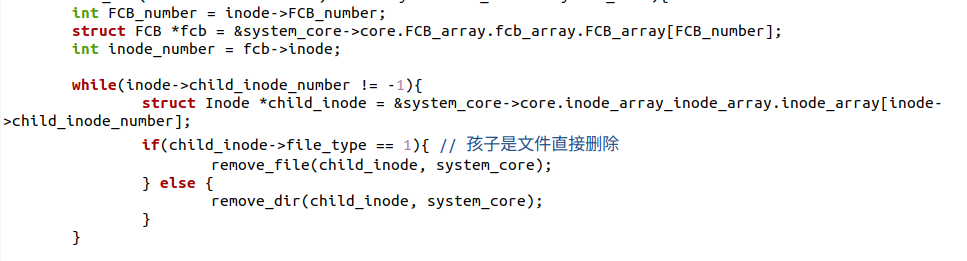


最后使用setBit函数将被删除文件对应的inode和FCB标记为可用状态。



1. 删除目录rmdir的实现

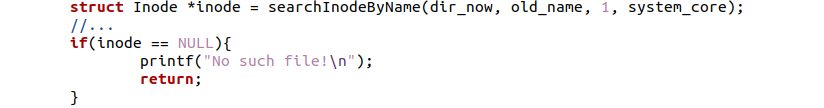
目录的删除与文件的删除的区别在于目录不一定为空。如果文件夹为空，直接删除即可。如果文件夹非空，则需要对其孩子进行遍历，如果孩子中存在空文件或空文件夹，同样可以直接删除，如果非空，则递归调用删除方法。



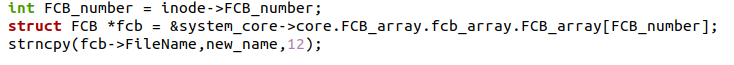
文件夹清空后，接下来的操作与上述删除文件的操作相同。

1. 修改文件名称（修改文件夹名操作方法相同，只需要修改文件类型的参数即可。其中，参数0表示文件夹，1表示文件）

首先根据获取的文件名，使用searchInodeByName查找文件的inode号，如果查找不到则打印错误信息。

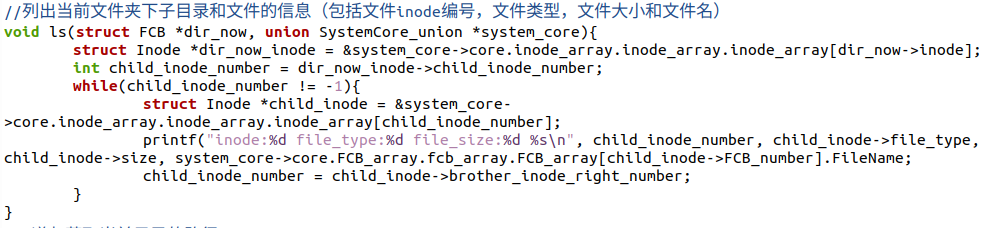


对于查找到的文件，获取其FCB号和FCB结构体指针，然后使用strncpy方法将新文件名复制到FCB结构体的FileName中，实现文件的重命名。



1. 查看文件系统目录结构ls的实现

ls命令的实现比较简单，获得当前目录的inode号，然后遍历当前目录的子节点即可。

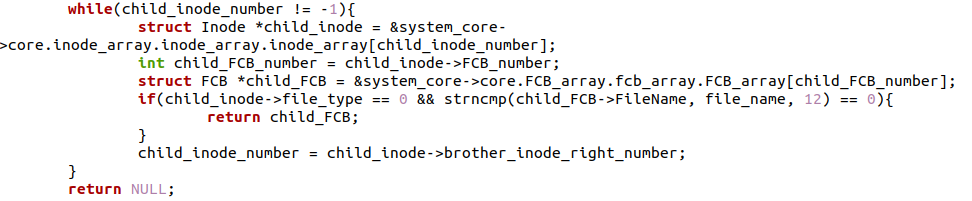


1. 进入某个指定目录cd的实现

首先从当前目录的inode中获得子目录的inode号，并将其进行存储。

1687106256299

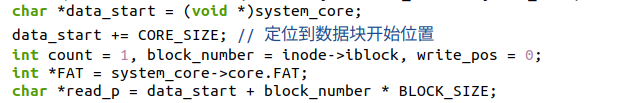
然后设置循环，在循环体中，首先获得当前子目录的inode号和FCB号，然后判断当前子目录是否是普通文件并判断是否与指定进入的目录名相同。如果是则返回当前的FCB指针，否则继续对下一个子目录进行判断。



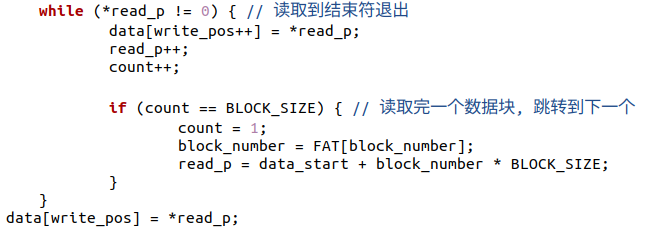
1. 读/写方法的实现

**读方法：**

定义指针data\_start指向数据块的起始位置（即跳过系统核心结构体的大小），初始化变量并获取系统核心结构体中的FAT数组指针。

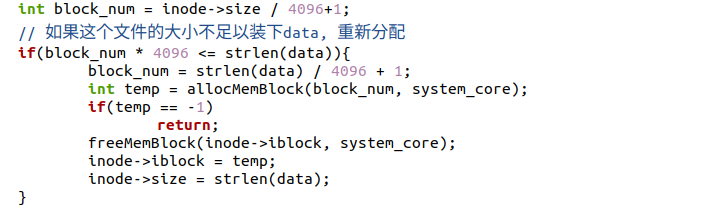


通过read\_p指针指向当前数据块起始位置，然后设置循环读取数据块内容直到读到结束符。最后将读取到的结束符添加到data数组中并返回读取到的数据。

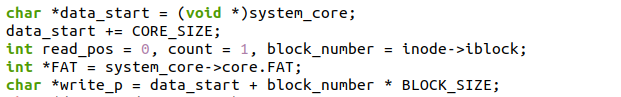


**写方法：**

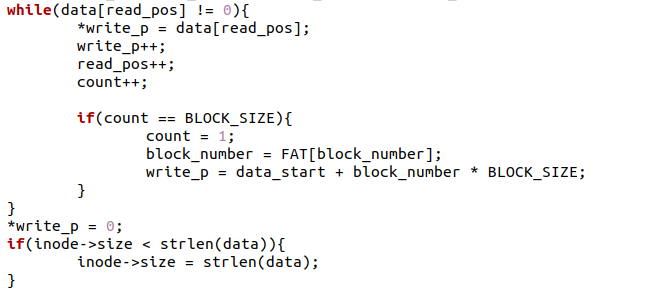
首先通过inode结构体中的size变量计算需要的数据块数量，然后判断当前文件剩余的空间是否足够写入data数组中的内容。如果不够则重新分配数据块并更新文件大小为data数组的长度。



然后同样进行读方法中的初始化工作。



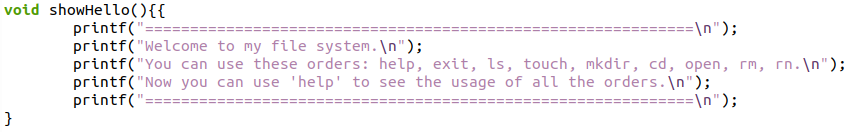
最后设置循环将data数组中的内容写入数据块，直到读到结束符。



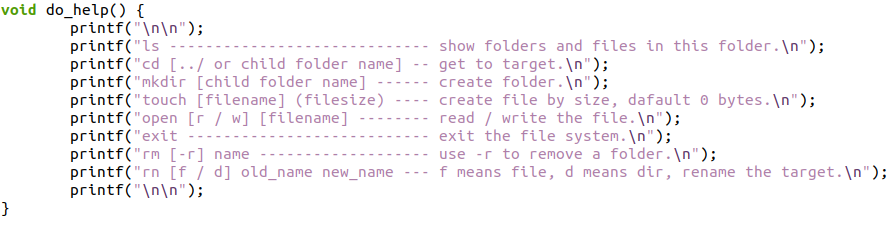
* 1. **编写程序，实现在文件系统中使用上述操作**

1. 预处理

编写一个showHello方法打印一些使用提示语



编写help方法对可操作指令的功能进行打印

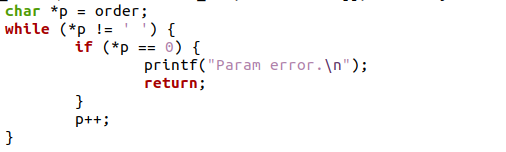


编写循环对输入的指令进行判断，如果是exit则打印相应的信息并退出循环，否则执行相应的操作



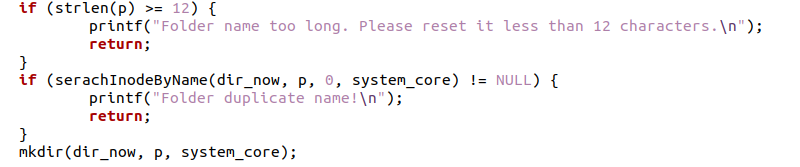
上述方法中，除了ls操作之外，其他操作后面包含其他内容，因此需要另外设置方法对这一部分进行判断。

1. 首先判断在指令后，是否存在表示其他内容的字符串。如果在遍历到空格之前就已经遍历到字符串的结尾，则输入不合法，输出Param error并返回。

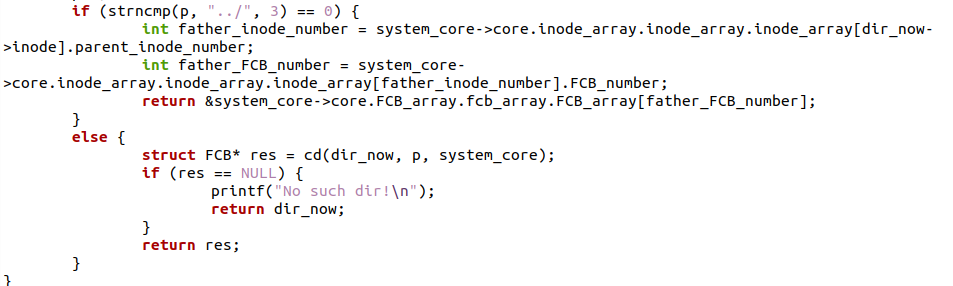


**接下来的判断具体到每个方法：**

①do\_mkdir：判断命令后的字符串所代表的文件名长度是否超过限制长度、所创建的文件夹是否已存在。通过不合理输入的筛选后调用mkdir方法实现操作。

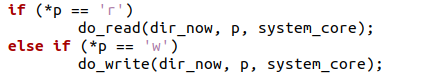


②do\_cd：判断要进入的文件夹名称是否是“../”，如果是，则通过当前目录的inode号找到父目录的inode号和对应的FCB号，如果不是，则调用cd进入指定的文件夹并获取返回值。

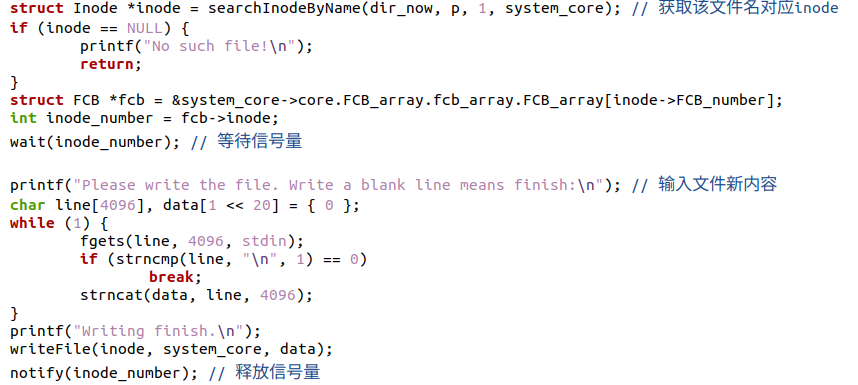


③do\_touch：判断文件名是否超过限定长度，不超过的话接着查找是否存在同名文件，如果不存在则使用touch方法进行创建。这里和上面do\_mkdir一样，不做展示。

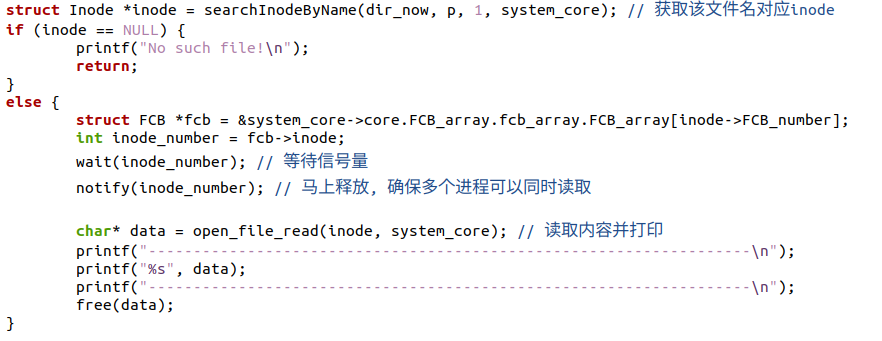
④do\_open：包含两种参数——r：读操作，w：写操作。所以首先需要在这个方法中对输入参数进行判断。



在写方法中：

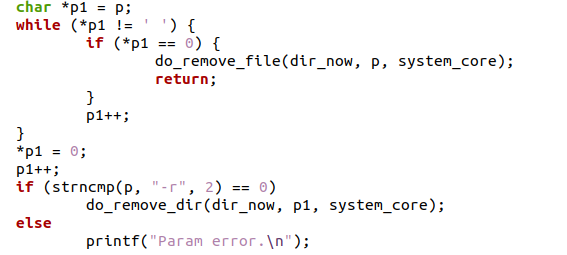


在读方法中：

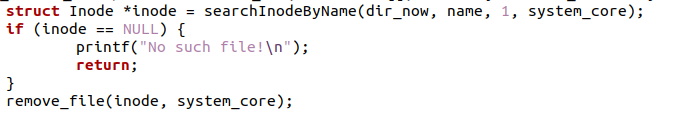


⑤do\_remove：这里同样包含两种参数——无参时：删除文件；-r：删除目录。

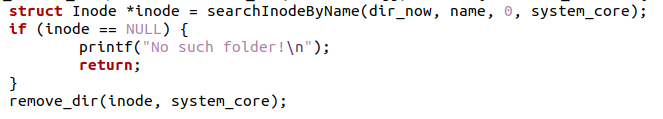
对输入指令参数的判断。



删除文件：

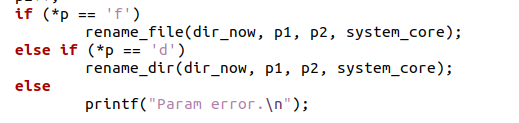


删除目录：



⑥do\_rename：这里同样包含两种参数——f：重命名文件；d：重命名目录。

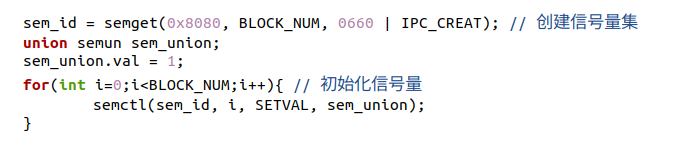
对输入指令参数的判断，然后直接调用shell程序中的方法。



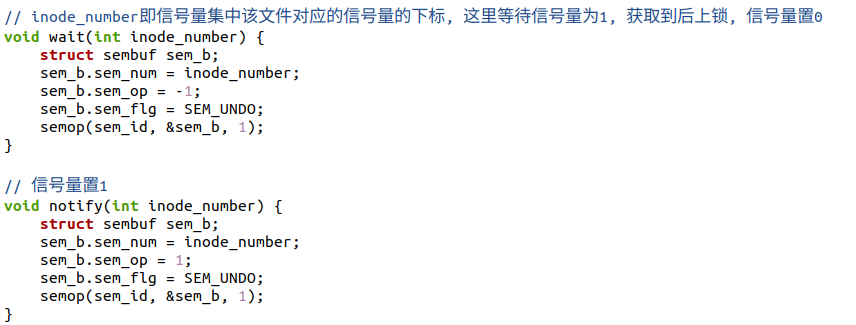
1. 参考进程同步的相关章节，通过信号量机制实现多个终端对上述文件系统的互斥访问，系统中的一个文件允许多个进程读，不允许写操作；或者只允许一个写操作，不允许读。

在上述功能的基础上添加对文件的读写操作。

1. 在步骤1申请内存的程序（这个程序类似一个生产者进程，那么使用系统的程序就可以看成一个消费者进程）中创建信号量集，在步骤2中我们提到文件和文件夹与inode一一对应，那么在创建信号量集的时候，使其数量和inode信号量相同即可。

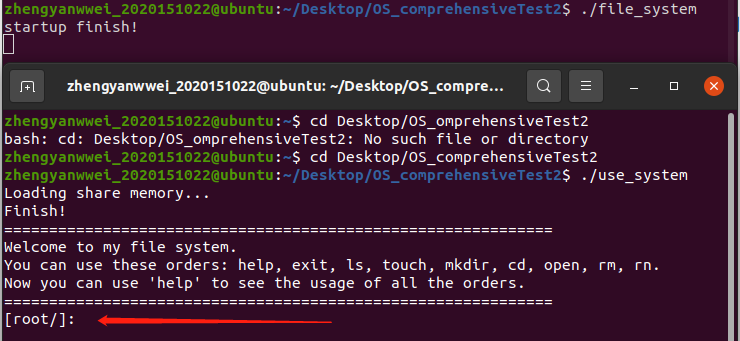


1. 在实现操作的shell文件中封装P/V操作的函数

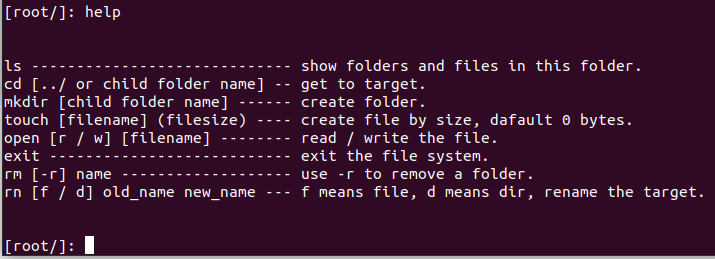


**运行：**

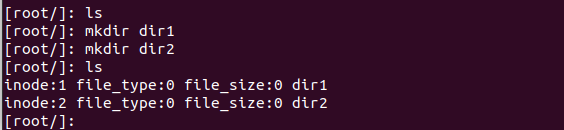
**打开终端窗口，尝试运行基本操作：**

****

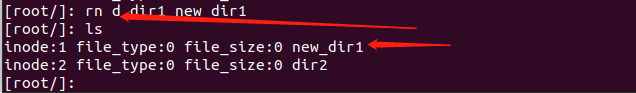
1. 输入help打印信息：



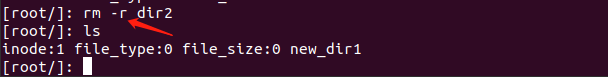
1. 创建目录mkdir：



1. 修改目录名rn d old\_folder\_name new\_folder\_name：



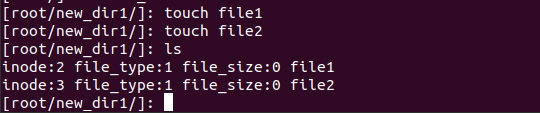
1. 删除目录rm -r folder\_name：



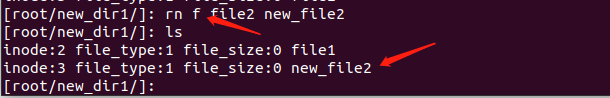
1. 进入目录new\_dir1：

1687157040171

1. 创建文件touch：

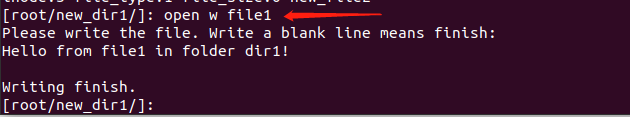


1. 修改文件名：

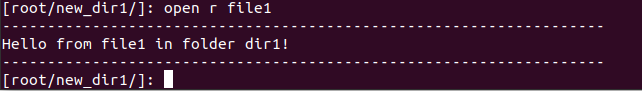


1. 读写操作：

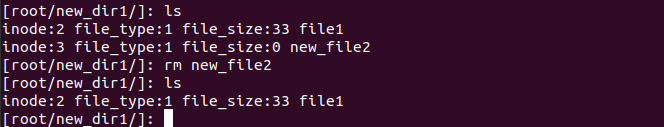
写文件：



读文件：



1. 删除文件：



1. 查看文件系统目录结构ls：

通过上面的截图可以看到该操作已经实现，打印信息为：

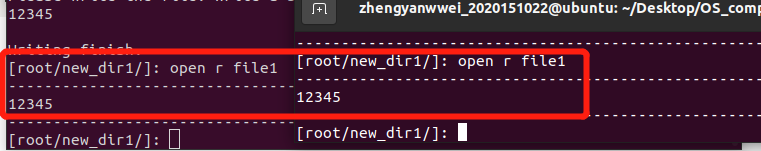
“inode:inode号 file\_type:文件类型（0-目录，1-文件） file\_size:文件大小 文件名”

1. 输入exit退出：

1687157508691

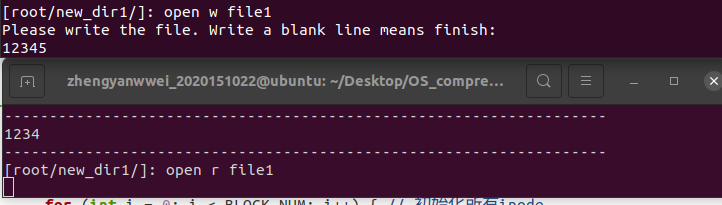
**打开新的终端窗口，尝试运行两个以上读写操作：**

1. 尝试同时读：



两个进程同时读。

1. 尝试写的同时进行读操作：



当上面的进程仍在执行写操作时，下面的进程执行读操作失败。