深 圳 大 学 实 验 报 告

课 程 名 称：操作系统

实验项目名称：综合实验 2 简易文件系统设计

学　　院：计算机与软件学院

专　　业：计算机科学与技术

指 导 教 师：阮元

报告人：刘睿辰　学号：**2018152051**　班级：数计班

实 验 时 间：**2021.6.21-2021.6.27**

实验报告提交时间：**2021.6.27**

教务处制

一、实验目的与要求：

综合利用文件管理的相关知识，结合对文件系统的认知，编写简易文件系统，加深操作系统中文件系统和资源共享的认识。

二、实验内容：

1. 可以使用 Linux 或其它 Unix 类操作系统；
2. 全面实践文件和目录操作及其组织结构；
3. 编写简易文件系统。

三、实验环境：

1. 硬件：桌面 PC；
2. 操作系统：Linux；
3. 实验平台：Ubuntu 18.04.

四、方法、步骤：

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。如图 1 所示，这是 Linux 系统的多级目录结构。

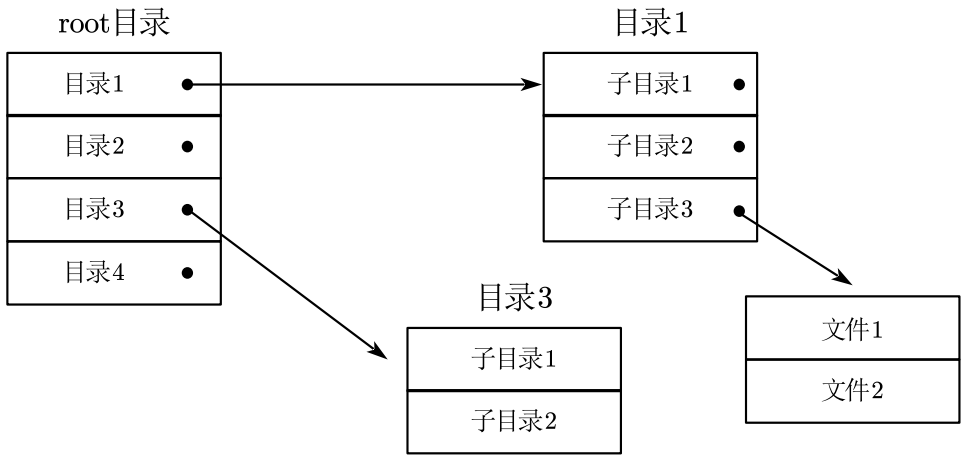

图 1. 多级系统结构（树型结构）

用户（或用户进程）要访问某个文件时要用文件路径名标识文件，文件路径名是个字符串。各级目录之间用“/”隔开。从根目录出发的路径称为绝对路径。显然，每次都从根目录开始查找，是很低效的。因此可以设置一个“当前目录”。当用户想要访问某个文件时，可以使用从当前目录出发的“相对路径”。可见，引入“当前目录”和“相对路径”后，磁盘的 I/O 的次数减少了。这就提升了访问文件的效率。树形目录结构可以很方便对文件进行分类，层次结构清晰，也能够更有效地进行文件的管理和保护。但是，树形结构不便于实现文件的共享。

2. 创建一个 100M 的文件或者创建一个 100M 的共享内存。

我们使用 malloc 函数来分配 100M 的内存。然后对于系统内的每一个盘块，如果没有被占用，就标记为 0，否则标记为 1；此外，由于储存位置图 bitmap 也需要空间，当盘块大小为 1KB 的时候，系统盘块个数为

$$\frac{100\times1024\times1024}{1024}=100\times1024$$

对于每一个盘块我们都需要进行记录，如果用 char 型变量来记录的话，就是需要 100×1024 bits 来进行记录，所以需要的盘块个数为

$$\frac{100\times1024}{1024}=100$$

所以前 100 个盘块应被标记为 1，代表从开始位置的 100 个盘块用来记录位视图。代码如下所示。

```
#define system_size 100*1024*1024           //系统大小，100M
#define block_size 1024                     //盘块大小，1KB
#define block_count system_size/block_size  //系统盘块数目，100*1024

char *startaddr;  //起始地址

/* initialize the system */
void initSystem()
{
    /* 分配 100M 空间 */
    startaddr = (char*)malloc(system_size * sizeof(char));
    //初始化盘块的位示图
    for(int i = 0; i < block_count; i++)
    {
        startaddr[i] = '0';
    }
    //用于存放位示图 bitmap 的空间已被占用，计算占用盘块数为 100
    int bitMapSize = block_count * sizeof(char) / block_size;

    //allocate from the start
    for(int i=0; i<bitMapSize; i++)
    {
        startaddr[i] = '1';   //盘块已被使用
    }
}
```

3. 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（共享内存或 mmap），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。
基本数据结构：目录单元+目录集合
本系统规定一个目录表只占用一个盘块，一个目录项大小为 64B，所以一个目录表中最多可含 15 个目录项，设置最大目录项DIRTABLE_MAX_SIZE 的值为 15。然后我们设置目录项，每一个目录项都有其名字，此外我们还需要其类型（文件或路径），然后设置其起始盘块，如下所示。

```
1.  //目录项结构：
2.  typedef struct dirUnit
3.  {
4.      char name[NUM];                 //文件名
5.      char type;                      //文件类型,0 为目录,1 为文件
6.      int start;                      //起始盘块
7.  }diru;
```

对于每一个盘块，内部都有一个目录表。目录表中应该目录项的个数diruAmount 。结构如下所示。

```
1.  //目录表结构：
2.  typedef struct dirTable
3.  {
4.      int diruAmount;                 //目录项数目
5.      diru dirs[DIRTABLE_MAX_SIZE];   //目录项列表
6.  }dirt;
```

规定了目录表之后，相当于文件在我们能访问的路径中被规定好了。但是实际上文件在内存中是在盘块中存储的，所以我们还需要规定文件控制块(File Control Block, FCB)，我们需要规定 FCB 中的文件大小（需要以盘块为单位），数据量大小，盘块索引号。因为涉及到文件读写，所以我们需要加入读指针。关于每一个文件都有一个链接，链接的个数我们记作 link，用于文件的共享，当文件的链接数为 0 时，系统可以回收文件的空间。此外，我们还要进行信号量的设置，这个我们在后面写入互斥的时候进行叙述。
FCB 的结构如下所示。

```
1.  //文件控制块结构：
2.  typedef struct FCB
3.  {
4.      int filesize;                   //文件大小，盘块为单位
5.      int datasize;                   //已写入的内容大小，字节为单位
6.      int block_index;                //文件数据起始盘块索引号
7.      int read_pointer;               //读指针，字节为单位
8.      int link;                       //文件链接数
9.      sem_t *write_semaphore;
10. }fcb;
```

由于采用的是连续分配方式，所以系统规定文件被创建时，必须给出文件的大小，而且后期也不能修改文件的大小。

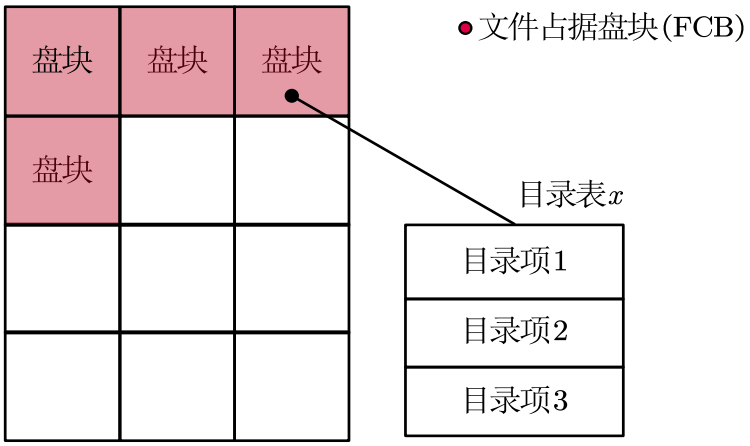

图 2. 文件占据系统图示

接口函数：按照题目中的要求，我们需要满足创建目录 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 这些功能，所以我们规定接口函数如下：

```
1.  //展示当前目录 ls
2.  void listDir();
3.  //切换目录 cd
4.  int cdDir(char dirName[]);
5.  //修改文件名或者目录名 mv
6.  int mv(char oldName[], char newName[]);////////////////////////////////
7.  //创建和删除文件，首先建立文件控制块
8.  //创建 FCB
9.  int createFCB(int fcbBlockNum, int fileBlockNum, int filesize);
10. int touch(char name[], int filesize);
11. //新建文件夹
12. int mkDir(char dirName[]);
13. //在目录表中加入目录项
14. int add(dirt* myDirTable, char name[], int type, int FCBBlockNum);
15. //删除文件
16. int rm(char name[]);
17. //释放文件控制块
18. int Free(int fcbBlock);
19. //在目录表中删除目录项
20. int deleteDirt(dirt* myDirTable, int unitIndex);
21. //删除目录 rmdir
22. int rmDir(char dirName[]);
23. //删除文件/目录项
24. int deleteFileInTable(dirt* myDirTable, int unitIndex);
```

```
//从目录中查找目录项目
int findUnitInTable(dirt* myDirTable, char unitName[]);
//读取文件内容
int Read(char name[], int length);
//编辑文件内容
int gedit(char name[], char content[]);
```

4. 在步骤 2 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。（40 分）注明：全部在内存中实现

1） mkdir 指令：创建目录功能

在构建目录的时候，我们分步来思考问题：

a) 首先我们应该分配一部分盘块，也就是应该得到新建的文件或文件路径的起始盘块编号，所以我们编写了 getBlock 函数，该函数返回一个起始盘块编号。代码如图 3 所示。

```
//磁盘分配
int getBlock(int blockSize)
{
    int startBlock = 0;
    int sum = 0;
    for(int i=0; i<block_count; i++)
    {
        if(startaddr[i] == '0')
        {
            if(sum == 0)
            {
                startBlock = i;  //获得该文件起始盘块号
            }
            sum++;
            if(sum == blockSize)
            {
                for(int j=startBlock;j<startBlock+blockSize;j++)
                {
                    startaddr[j] = '1';  //该部分应该被占用，置 1
                }
                return startBlock;
            }
        }
        else
        {
            sum = 0;
        }
```

```
    }
    return -1;
}
```

b) 接下来我们需要判断加入的文件是否存在，所以我们编写findUnitInTable函数来检查是否有重名文件。这部分的检查方法就是逐次与已经存在于目录表中的文件名称进行比较，如果重名的话就需要提示文件已经存在。findUnitInTable函数的编写如下所示。

```
//从目录中查找目录项目
int findUnitInTable(dirt* myDirTable, char unitName[])
{
    //获得目录表
    int diruAmount = myDirTable->diruAmount;
    int unitIndex = -1;
    for(int i=0; i<diruAmount; i++)
    {
        if(strcmp(unitName, myDirTable->dirs[i].name) == 0)
        {
            unitIndex = i;
        }
    }
    return unitIndex;
}
```

c) 如果新建的文件通过了是否重复，那么就需要将其加入目录表，这里我们就可以完成我们的add函数。代码比较简单，如下所示。

```
int add(dirt* myDirTable,char name[],int type,int FCBBlockNum)
{
    //获得目录表
    int diruAmount = myDirTable->diruAmount;
    //检测目录表示是否已满
    if(diruAmount == DIRTABLE_MAX_SIZE)
    {
        printf("内存已满！\n");
        return -1;
    }

    //是否存在同名文件
    if(findUnitInTable(myDirTable, name) != -1)
    {
        printf("文件已经存在！\n");
        return -1;
```

```
    }
    //构建新目录项
    diru* newdiru = &myDirTable->dirs[diruAmount];
    myDirTable->diruAmount++;//当前目录表的目录项数量+1
    //设置新目录项内容
    strcpy(newdiru->name, name);
    newdiru->type = type;
    newdiru->start = FCBBlockNum;

    return 0;
}
```

d) 最后我们来新建文件夹。我们在开始的时候需要规定一个最大文件名称长度，如果文件名称过长会提示“文件名称过长”这个提示信息。然后需要为目录表分配空间，并将目录作为目录项添加进目录表。

```
int mkDir(char dirName[])
{
    if(strlen(dirName) >= NUM)
    {
        printf("文件名称过长！\n");
        return -1;
    }
    //为目录表分配空间
    int dirBlock = getBlock(1);
    if(dirBlock == -1)
        return -1;
    //将目录作为目录项添加到当前目录
    if(add(current, dirName, 0, dirBlock) == -1)
        return -1;
    return 0;
}
```

2） ls 指令：展示当前文件夹的所有文件

在之前我们提到过，关于目录表，diruAmount 代表了当前的目录项的个数。所以当前路径下的文件个数就是diruAmount 。如果我们用current 来记录当前的路径，那么当前文件夹下的文件的个数就是current.diruAmount 。那么我们接下来要做的就是遍历这些文件，这些文件对应的就是 current.dirs[i]，每一个 current.dirs[i] 都是一个目录单元，主要输出他们的名字即可，即 current.dirs[i].name 。

代码如下所示。

```
void listDir()
```

```
{
    int unitAmount = current->diruAmount;
    printf("总计:%d\n", unitAmount);
    printf("文件名称\t 文件类别\t 文件大小\tFCB\t 起始块\n");

    for(int i=0; i<unitAmount; i++)
    {
        //获取目录项
        diru unitTemp = current->dirs[i];
        printf("%8s\t", unitTemp.name);
        if(unitTemp.type == 0)
        {
            char s1[100] = "文件夹";
            printf("%10s\t", s1);
        }
        else if (unitTemp.type == 1)
        {
            char s1[100] = "文件";
            printf("%8s\t", s1);
        }
        //该表项是文件，继续输出大小和起始盘块号
        if(unitTemp.type == 1)
        {
            int fcbBlock = unitTemp.start;
            fcb* fileFCB = (fcb*)getBlockAddr(fcbBlock);
            char s1[100] = {0};
            char s2[100] = {0};
            char s3[100] = {0};
            sprintf(s1, "%d", fileFCB->filesize);
            sprintf(s2, "%d", fcbBlock);
            sprintf(s3, "%d", fileFCB->block_index);

            printf("%12s\t%11s\t%s\n", s1,s2,s3);
        }
        else
        {
            int dirBlock = unitTemp.start;
            dirt* myTable = (dirt*)getBlockAddr(dirBlock);
            char s1[100] = {0};
            char s2[100] = {0};

            sprintf(s1, "%d", myTable->diruAmount);
            sprintf(s2, "%d", unitTemp.start);

```

```
46.             printf("%11s\t%11s\n",s1,s2);
47.         }
48.     }
49. }
```

在代码中标记红色的区域，是我们为了输出表项和表头保持对齐。在 Linux 系统中，为了将整型转为字符串型，我们使用了 sprintf 函数，因为一般情况下 itoa 函数属于非标准 C 语言，在 Linux 系统中并不能正常工作。

3） rmdir 指令：删除路径

删除文件需要注意一点，只有子路径才能被删除。当我们位于父文件夹的时候，我们不能删除父文件夹本身，只能删除父文件夹下面的某个子文件夹。所以在开始的时候要判断是否删除了父文件夹，如果输入的是父文件夹，则需要报错误信息，代码如下所示。

```
1.  //忽略系统的自动创建的父目录
2.      if(strcmp(dirName, "..") == 0)
3.      {
4.          printf("无法删除上层目录！\n");
5.          return -1;
6.      }
```

当要删除的路径是合法的时候，我们就再次调用 findUnitInTable 函数，从而找到这个路径的索引 index，然后 current.dirs[index] 就是我们要删除的路径。

这里需要注意，删除文件夹的时候，由于文件夹可能嵌套了文件夹，所以我们可以采用递归的方法来删除文件夹。假设要删除的文件索引是 unitIndex，那么对于它下面的 diruAmount 个子文件/子文件夹，我们都要一一进行删除。所以找到这个文件对应的位置之后，我们遍历目录表，对于每一个元素进行递归，直到删除的是文件，这就代表到达了最深处的子文件，然后释放这个位置的盘块空间，如下所示。

```
1.  int releaseBlock(int num, int size)
2.  {
3.      int end = num + size;
4.      //修改位示图盘块的位置为 0
5.      for(int i=num; i<end; i++)
6.      {
7.          startaddr[i] = '0';
8.      }
9.      return 0;
10. }
```

我们再来考虑如果删除的是单个的文件，直接调用这个 releaseBlock 函数是不行

的，因为我们需要更改文件的链接的个数，然后才能释放盘块，所以我们构建Free函数，先删除链接，再释放盘块，代码如下。

```
1.  //释放文件内存
2.  int Free(int fcbBlock)
3.  {
4.      fcb* myFCB = (fcb*)getBlockAddr(fcbBlock);
5.      myFCB->link--;  //链接数减一
6.      //无链接，删除文件
7.      if(myFCB->link == 0)
8.      {
9.          //释放文件的数据空间
10.         releaseBlock(myFCB->block_index, myFCB->filesize);
11.     }
12.     releaseBlock(fcbBlock, 1);
13.     return 0;
14. }
```

所以综合两部分的内容，我们可以编写出删除文件/目录项的代码，如下所示。

```
1.  //删除文件/目录项
2.  int deleteFileInTable(dirt* myDirTable, int unitIndex)
3.  {
4.     //查找文件
5.      diru myUnit = myDirTable->dirs[unitIndex];
6.      //判断类型
7.      if(myUnit.type == 0)//目录
8.      {
9.          //找到目录位置
10.         int fcbBlock = myUnit.start;
11.         dirt* table = (dirt*)getBlockAddr(fcbBlock);
12.         //递归删除目录下的所有文件
13.         printf("已删除%s\n", myUnit.name);
14.         int unitCount = table->diruAmount;
15.         for(int i=1; i<unitCount; i++)//忽略“..”
16.         {
17.             printf("删除%s\n", table->dirs[i].name);
18.             deleteFileInTable(table, i);
19.         }
20.         //释放目录表空间
21.         releaseBlock(fcbBlock, 1);
22.     }
23.     else
24.     {
```

```
        int fcbBlock = myUnit.start;
        Free(fcbBlock);
    }
    return 0;
}
```

这个函数deleteFileInTable可以用于删除目录，也可以用于删除文件。现在我们在位示图中清除了内容，还需要在目录表中删除文件，也就是删除目录表中的目录单元，这一步和数组元素删除很相似，用后面的元素依次覆盖掉前面的元素，然后总个数减一，代码如下所示。

```
//删除目录项
int deleteDirt(dirt* myDirTable, int unitIndex)
{
    //迁移覆盖
    int diruAmount = myDirTable->diruAmount;
    for(int i=unitIndex; i<diruAmount-1; i++)
    {
        myDirTable->dirs[i] = myDirTable->dirs[i+1];
    }
    myDirTable->diruAmount--;
    return 0;
}
```

所以总体来看，我们既要在位示图中删除目录单元，也要在目录表中删除。结合之前的分析，删除路径的代码如下所示。

```
//删除目录 rmdir
int rmDir(char dirName[])
{
    //忽略系统的自动创建的父目录
    if(strcmp(dirName, "..") == 0)
    {
        printf("无法删除上层目录！\n");
        return -1;
    }
    //查找文件
    int unitIndex = findUnitInTable(current, dirName);
    if(unitIndex == -1)
    {
        printf("没有该目录！请检查...\n");
        return -1;
    }
    diru myUnit = current->dirs[unitIndex];
```

```
    //判断类型
    if(myUnit.type == 0)//目录
    {
        deleteFileInTable(current, unitIndex);
    }
    else
    {
        printf("这不是一个目录！请检查...\n");
        return -1;
    }
    //从目录表中剔除
    deleteDirt(current, unitIndex);
    return 0;
}
```

4） mv 指令：修改文件名称

文件名称修改比较容易。我们只需要调用findUnitInTable 函数找到目录单元，然后用strcpy 函数来进行修改即可。代码如下所示。

```
//修改文件名或者目录名 mv
int mv(char oldName[], char newName[])
{
    int unitIndex = findUnitInTable(current, oldName);
    if(unitIndex == -1)
    {
        printf("file not found\n");
        return -1;
    }
    strcpy(current->dirs[unitIndex].name, newName);
    return 0;
}
```

5） open 指令：创建文件

这一步和 mkdir 指令的编写比较相近。但是我们需要注意，文件夹是没有真正的地址的，地址都是由文件来占据的，所以在这里我们需要设置块索引、文件大小，也就是找到文件控制块的存储位置。代码如下所示。

```
//创建 FCB
int createFCB(int fcbBlockNum, int fileBlockNum, int filesize)
{
    //找到 fcb 的存储位置
    fcb* currentFCB = (fcb*) getBlockAddr(fcbBlockNum);
    currentFCB->block_index = fileBlockNum;//文件数据起始位置
    currentFCB->filesize = filesize;//文件大小
```

```
8.      currentFCB->link = 1;//文件链接数
9.      currentFCB->datasize = 0;//文件已写入数据长度
10.     currentFCB->read_pointer = 0;//文件读指针
11.     return 0;
12. }
```

如果存储位置已找到且 FCB 可以被创建，就调用之前叙述过的add 函数将其添加进目录表，这样的话文件就被建立了，代码如下。

```
1.  //获得 FCB 的空间
2.  int fcbBlock = getBlock(1);
3.  if(fcbBlock == -1)
4.      return -1;
5.  //获取文件数据空间
6.  int FileBlock = getBlock(filesize);
7.  if(FileBlock == -1)
8.      return -1;
9.  //创建 FCB
10. if(createFCB(fcbBlock, FileBlock, filesize) == -1)
11.     return -1;
12. //添加到目录项
13. if(add(current, name, 1, fcbBlock) == -1)
14.     return -1;
15. return 0;
```

6） gedit 指令：修改文件

首先查找gedit 指令后面跟着的文件是否是存在的，如果不存在需要提示错误。查找文件就在目录表中查找即可，代码如下。

```
1.  int unitIndex = findUnitInTable(current, name);
2.  if(unitIndex == -1)
3.  {
4.      printf("没有找到该文件！请检查...\n");
5.      return -1;
6.  }
```

如果需要写入的文件是存在的，那么就找到这个文件的文件控制块，并得到地址。得到地址之后，从开始地址开始，写入我们输入的内容content 即可。注意，如果输入长度超过文件控制块大小的时候，我们就不能继续写入了，需要有提示信息。代码如下。

```
1.  //控制块
2.  int fcbBlock = current->dirs[unitIndex].start;
```

```
3.  fcb* myFCB = (fcb*)getBlockAddr(fcbBlock);
4.
5.  int contentLen = strlen(content);
6.  int filesize = myFCB->filesize * block_size;
7.  char* data = (char*)getBlockAddr(myFCB->block_index);
8.  for(int i=0; i<contentLen && myFCB->datasize<filesize; i++, myFCB->datasize++)
9.  {
10.     *(data+myFCB->datasize) = content[i];
11. }
12. printf("请按任意键退出...\n");
13. getchar();
14. if(myFCB->datasize == filesize)
15. {
16.     printf("无法继续写入！\n");
17. }
18. return 0;
```

7） rm 指令：删除文件

rm 指令用于删除文件，它的工作原理和rmdir 指令类似，需要使用Free 函数在位示图中修改盘块信息，也要在目录表中删除，也就是在内存中进行删除。这部分代码与rmdir 指令类似，在这里不作赘述。

8） cd 指令：变换当前文件目录

之前我们使用了current 记录存储了当前文件路径的目录表，那么我们需要使用cd 指令来改变当前操作的目录表。结合之前写过的函数，很明显我们需要通过findUnitInTable 函数来找到那个路径，然后才能进入。当然，我们也需要做警报处理，也就是存在文件目录不存在或者非文件目录的情况，这两种情况都要进行提示。代码如下所示。

```
1.  //目录项在目录位置
2.  int unitIndex = findUnitInTable(current, dirName);
3.  //不存在
4.  if(unitIndex == -1)
5.  {
6.      printf("没有该目录！请检查...\n");
7.      return -1;
8.  }
9.  if(current->dirs[unitIndex].type == 1)
10. {
11.     printf("这不是一个目录！请检查...\n");
12.     return -1;
13. }
```

找到目录单元在目录表中的位置unitIndex之后，current.dirs[unitIndex].start就代表了需要前往的路径的块号。知道块号之后就可以获得其块地址，将current修改为当前盘块的目录表。同时更新文件路径信息，也就是通过strcat将前后路径粘连到一起。更新current的代码如下。

```
//修改当前目录
int dirBlock = current->dirs[unitIndex].start;
current = (dirt*)getBlockAddr(dirBlock);
```

9） Read指令：文件读取

文件读取相对容易。首先用findUnitInTable函数找到对应文件，找不到就需要报出错误信息。然后找到对应的文件控制块，将接收字符串首地址设置为文件控制块的地址。将读指针初始化，随着读指针增加，将信息读入接收字符串。代码如下所示。

```
//控制块
int fcbBlock = current->dirs[unitIndex].start;
fcb* myFCB = (fcb*)getBlockAddr(fcbBlock);
myFCB->read_pointer = 0; //文件指针重置
//读数据
char* data = (char*)getBlockAddr(myFCB->block_index);
int datasize = myFCB->datasize;
printf("读取结果为(以#为终止符):");
for(int i=0; i<length && myFCB->read_pointer<datasize; i++, myFCB->read_pointer++)
{
    printf("%c", *(data+myFCB->read_pointer));
}
if(myFCB->read_pointer == datasize)//读到文件末尾用#表示
{
    printf("#");
}
printf("\n 请按任意键退出...\n");
getchar();
```

5. 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。（30分）

这是一个关于读写者问题的信号量控制。

我们设置只允许一个写操作，不允许读。

关于读写者问题的信号量控制，伪代码如下所示。

```
semaphore writer_semaphore = 1;
```

```
writer()
{
    while(1)
    {
        P(writer_semaphore);
        perform writing operation...
        V(writer_semaphore);
    }
}
```

所以我们需要对上面的gedit 指令进行修改。我们应该加上信号量设置，首先获得写者锁，然后先加锁使得别的进程不能写，待写进程结束之后，释放信号量。代码如下所示。

```
//获得写者锁
myFCB->write_semaphore = sem_open("write_semaphore", 0);

//上写者锁
sem_wait(myFCB->write_semaphore);

for(int i = 0; i < contentLen && myFCB->datasize < filesize; i++, myFCB->datasize++)
{
    *(data+myFCB->datasize) = content[i];
}

printf("请按任意键退出...\n");
getchar();

//释放写者锁
sem_post(myFCB->write_semaphore);
if(myFCB->datasize == filesize)
{
    printf("无法继续写入！\n");
}
return 0;
```

四、实验结论：

我们对文件系统进行测试。

1. 运行主程序，打印提示信息以及提供指令输入，如图 3 所示。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab/com-exp2$ ./shell
*********************************欢迎进入文件系统！ **********************************
以下指令可以帮助您使用该文件系统!
open + <string> + <length>        新建一个文件,文件名为<string>,长度为length
rm + <string>                     删除一个文件,文件名为<string>
mv + <string1> + <string2>        重命名一个文件/文件夹<string1>,新文件名为<string2>
mkdir + <string>                  新建一个文件夹,文件名为<string>
rmdir + <string>                  删除一个文件夹,文件名为<string>
gedit + <string> + <content>      编辑一个文件,文件名为<string>,内容为content
read + <string> + <length>        读取一个文件,文件名为<string>,长度为length
ls + <string>                     列举该文件夹下的所有文件
cd + <string>                     打开下一个目录<string>
cd ..                             返回上一个目录
quit                              退出文件系统
**********************************祝您使用愉快！ **********************************
/
```

图 3. 文件系统初始界面

2. mkdir 和open 指令结合ls 指令的测试。

   我们使用mkdir 指令建立一个文件夹并使用ls 指令查看构建是否成功。如图 4 所示，文件夹建立成功，且有文件重复的提示。

```
/mkdir doc1
/mkdir doc2
/mkdir doc2
文件已经存在!
/ls
总计:2
文件名称        文件类别        文件大小        FCB     起始块
    doc1         文件夹          1           101
    doc2         文件夹          1           102
/
```

图 4. mkdir 指令和ls 指令

我们再用open 指令建立一个大小为 100 的文件，如图 5 所示，文件类别已经发生了变化。ls 指令指出了文件的大小。此外，我们也建立一个同名文件，可以看到出现了文件重名的提示。

```
/open 1.txt 100
/ls
总计:3
文件名称        文件类别        文件大小        FCB     起始块
    doc1         文件夹          1           101
    doc2         文件夹          1           102
   1.txt          文件          100          104     105
/open 1.txt 50
文件已经存在!
/ls
总计:3
文件名称        文件类别        文件大小        FCB     起始块
    doc1         文件夹          1           101
    doc2         文件夹          1           102
   1.txt          文件          100          104     105
```

图 5. open 指令和ls 指令

3. cd 指令

我们再用cd 指令进入一个我们建立的文件夹，再用cd .. 指令返回上级目录，如图 6 所示。

```
/cd doc2
/doc2/ls
总计:1
文件名称        文件类别        文件大小        FCB     起始块
      ..         文件夹           3           100
/doc2/cd ..
/ls
总计:3
文件名称        文件类别        文件大小        FCB     起始块
    doc1         文件夹           2           101
    doc2         文件夹           1           102
   1.txt          文件           100          104     105
```

图 6. cd 指令和cd .. 指令

4. 删除文件rm 指令以及删除路径rmdir 指令

我们使用rmdir 指令在根目录中删除文件路径doc1 。为了体现递归删除的实现，我们首先使用cd 指令进入doc1 ，建立一个文件，如图 7 所示。

```
/cd doc1
/doc1/open 2.txt 100
/doc1/cd ..
```

图 7. 在待删除的文件夹中新建一个文件

然后在主目录中删除文件夹doc1 ，如图 8 所示，先删除了主路径，然后删除了路径内部的文件。

```
/rmdir doc1
已删除doc1
删除2.txt...
/ls
总计:2
文件名称        文件类别        文件大小        FCB     起始块
    doc2         文件夹           1           102
   1.txt          文件           100          104     105
/
```

图 8. 删除文件路径

接下来我们用rm 指令删除文件1.txt ，如图 9 所示，删除成功。

```
/rm 1.txt
/ls
总计:1
文件名称        文件类别        文件大小        FCB     起始块
    doc2         文件夹           1           102
/
```

图 9. 删除文件

5. mv 指令对文件和文件夹进行改名

如图 10 所示，我们对文件夹 2 进行更名（这个文件夹 2 和图 9 中并不是一个，本次是重启了系统之后的测试），可以看到文件夹已经成功更名，且之前对于我们的错误

输入进行了提示。

```
/mv 2 2-dup
文件未找到!
/mv doc2 doc2-dup
/ls
总计:1
文件名称        文件类别        文件大小        FCB      起始块
doc2-dup         文件夹           1             101
/
```

图 10. mv 指令对文件夹更名

现在我们进入这个文件夹新建一个文件，对其进行更名，看到这个指令对文件也是适用的。

```
/cd doc2-dup
/doc2-dup/open 1.txt 100
/doc2-dup/mv 1.txt 1-dup.txt
/doc2-dup/ls
总计:2
文件名称        文件类别        文件大小        FCB      起始块
      ..         文件夹           1             100
1-dup.txt        文件            100           102      103
/doc2-dup/
```

图 11. mv 指令对文件更名

6. gedit 指令编辑文件内容

接着图 11，我们编辑1－dup.txt 这个文件，将“name:1－dup.txt”这个内容写入。为了验证是否成功写入，我们调用read 函数读取文件内容，如图 12 所示，编辑操作是正常的。

```
/doc2-dup/gedit 1-dup.txt name:1-dup.txt
请按任意键退出...

/doc2-dup/Read 1-dup.txt 100
没有这个指令，请检查!
/doc2-dup/read 1-dup.txt 100
读取结果为(以#为终止符):name:1-dup.txt#
请按任意键退出...


/doc2-dup/
```

图 12. gedit 指令编辑文件内容

在读和写的最后过程中，都有一个“请按任意键退出”的过程。这个过程实际上是保证读进程和写进程都结束了才关闭文件，否则系统认为文件处于“打开”的状态，这时不允许有其它的操作。

五、实验体会：

本次实验我认为是操作系统所有实验中难度最高的一个。在这次实验中我们需要结合所学的文件系统知识来真正实现一个文件系统，难度还是非常高的，主要体现在代码量很大、设计思路不容易想等方面。本次实验做出来的文件系统能实现基本的功能，但是很明显该文件系统还有很多不足之处。

这次实验加深了我对文件系统的认识，也初步地了解了文件在内存中的管理方法。文件在磁盘中的管理方法还有很多种，本次实验采用的是位示图的方法，因为我认为位示图比较好理解，而且实现起来相对更容易一些。

指导教师批阅意见： 成绩评定： 指导教师签字： 年　　月　　日
备注：