操作系统实验报告 文件系统

16281042 李许增 安全 1601

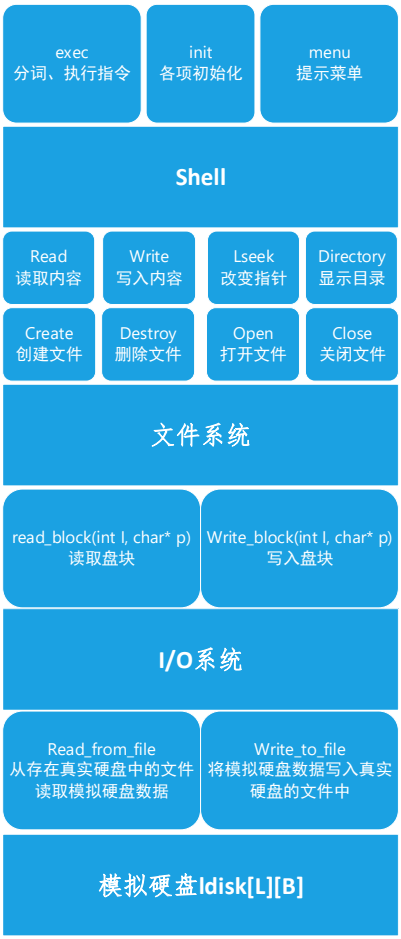
实验简介

本实验要求在假设的 I/O 系统之上开发一个简单的文件系统，这样做既能让实验者对文件系统有整体了解，又避免了涉及过多细节。用户通过 create，open，read 等命令与文件系统交互。文件系统把磁盘视为顺序编号的逻辑块序列，逻辑块的编号为 0 至 L - 1。I/O 系统利用内存中的数组模拟磁盘。

实验内容

整体架构

本次实验使用二维数组模拟硬盘，使用位图、文件描述符、目录来描述文件，打开文件表操作文件。所有定义放在 FileSystem.h 中，实现及主函数在 FileSystem.cpp 中，程序使用 C 语言完成。整体结构层次关系如下图：



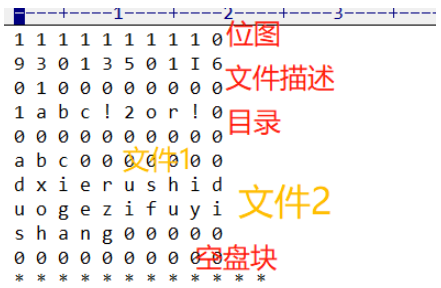
模拟硬盘

以 ldisk.txt 文件来存储 ldisk[L][B]的数据。本实验中设置 L 和 B 大小均为 10。文件中

每一盘块一行，最后附加终止符一行，共 11 行，每个数据之间空格分隔，共 20 列，总大小应为 243 字节。数组中初值为全零，存储在文件中以空格或者字符 0 来表示。硬盘文件示例如下图：

```
1 1 1 1 1 1 1 1 1 0
9 3 0 1 3 5 0 1 I 6
0 1 0 0 0 0 0 0 0 0
1 a b c ! 2 o r ! 0
0 0 0 0 0 0 0 0 0 0
a b c 0 0 0 0 0 0 0
d x i e r u s h i d
u o g e z i f u y i
s h a n g 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
* * * * * * * * * * * *
```

硬盘前 K 个盘块为系统保留盘，本次实验中 K = 3。其后的 2 个盘块用来存储目录文件，剩下的盘块自由存储文件。在系统保留盘中，盘块 0 存放位图，由于本次实验中硬盘块数和位数均为 10，故位图占满 0 号盘；盘块 1~2 存放文件描述符，动态存储，尚未占用时为 0，后续详细说明。由此可知，上图中硬盘表示的内容如下图所示：



系统变量定义如下：

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// L磁盘的存储块总数
#define L 10
// B表示每个存储块的长度
#define B 10
// K为系统保留盘数大小
#define K 3
//文件描述符数量
#define fdcpsize ((K-1)*B/4)
// 模拟磁盘
char ldisk[L][B];
```

位图

位图用来描述磁盘块的分配情况，共 10 位，0 表示未占用，1 表示已占用。创建或者删除文件，以及文件的长度发生变化时，文件系统都需要进行位图操作。

```c
//位图，0代表空，1代表被占用
int bmp[L];
```

文件描述符

前 k 个块的剩余部分包含一组文件描述符。每个文件描述符包含，文件长度和文件分配到的磁盘块号数组。在实验中我们可以把它设置为一个比较小的数，例如 3。磁盘块号数组中，0 位表示块号，1 位表示位号（偏移），2 位为状态位表示是否占用。

```
//文件描述符
struct Fdcp
{
    int len;//文件长度
    int fpos[3];//磁盘块号数组，0位为块号，1位为偏移，2位为状态，0为空闲1为占用
}fdcp[fdcpsize];
```

目录

我们的文件系统中仅设置一个目录，该目录包含文件系统中的所有文件。除了不需要显示地创建和删除之外，目录在很多方面和普通文件相像。目录对应 0 号文件描述符。初始状态下，目录中没有文件，所以，目录对应的描述符中记录的长度应为 0。每创建一个文件，目录文件的长度便增加一分。目录文件的内容由一系列的目录项组成，其中每个目录项由文件名和文件描述符序号组成。

本程序中目录设有目录总长代表目录文件大小，目录总数代表目录项个数。目录项中包含文件描述符序号和文件名，为了方便读写和判断，每个目录项以'!'作为结尾。

```
//目录
struct Index {
    int fdcpnum;//描述符序号
    char* fname;//文件名
    char end = '!';//每个目录尾
};
struct FIndex
{
    int size = 0;//目录总长
    int num = 0;//目录总数
    Index* index;//目录项
}findex;
```

打开文件表

文件系统维护一张打开文件表．打开文件表的长度固定，其表目包含读写缓冲区、读写指针、文件描述符号。文件被打开时，便在打开文件表中为其分配一个表目；文件被关闭时，其对应的表目被释放。

由于模拟磁盘空间有限，文件个数不会太多，故将打开文件表表目数量设定成为了固定值 5。此数值完全是自主设定和方便起见，可以任意改变，也可以设定为 oft 指针，分配时申请空间关闭时释放。

```
//打开文件表
struct OFT {
    char* buff; //读写缓冲区
    int rwpos = 0; //读写指针
    int fpos = 0; //文件描述符号
}oft[5];
```

I/O 系统

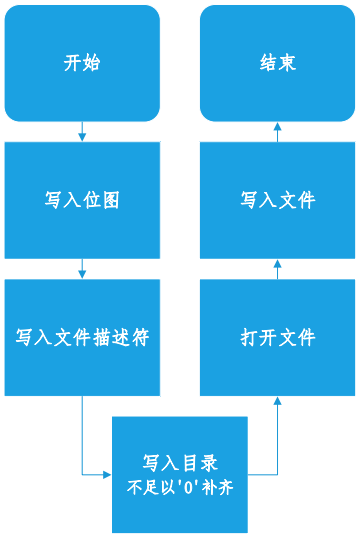
I/O 系统层面包含 4 个函数，两个读取，两个写入。该系统定义如下：

```
//I/O系统

// 该函数把逻辑块i的内容读入到指针p指向的内存位置，拷贝的字符个数为存储块的长度B
void read_block(int i, char* p);

// 该函数把指针p指向的内容写入逻辑块i，拷贝的字符个数为存储块的长度B
void write_block(int i, char* p);

// 存储到文件
void write_to_file();

/*把文件内容恢复到数组
*若文件为空，则返回-1
*若文件大小不对，返回-2
*否则返回0
*/
int read_from_file();
```

其中，指定逻辑块的读写较为简单，直接将模拟硬盘中的对应字符和指针 p 对应位置相互赋即可。注意到如果硬盘对应字符是初始值，或者是随机小于 0 的值，则用字符 0 来代替赋值。

void write_to_file()

存储到文件。该函数功能是将内存中的位图、目录、文件描述符等系统项存储到 ldisk 当中，同时将 ldisk 整体写入 ldisk.txt，即生成真实硬盘中的硬盘文件。流程如下：



写入位图较为简单，直接和 ldisk[0]赋值即可。

写入文件描述符，首先写入文件长度 len，再写入磁盘块号数组中的各项。由于描述符开始于 ldisk[1][0]，故可以计算出每个项目该写到磁盘何处，使用指针加偏移的方式写入磁盘。

```
//写入文件描述符
for (j = 0; j < fdcpsize; j++) {
    *(ldisk[1] + 4 * j) = fdcp[j].len + '0';
    *(ldisk[1] + 4 * j + 1) = fdcp[j].fpos[0] + '0';
    *(ldisk[1] + 4 * j + 2) = fdcp[j].fpos[1] + '0';
    *(ldisk[1] + 4 * j + 3) = fdcp[j].fpos[2] + '0';
}
```

目录从 ldisk[3][0]起始，可以仿照文件描述符的写入方式。根据目录总长 size，维护一个自 ldisk[3][0]算起的偏移指针。先写入描述符序号，再根据文件名长度，写入文件名，最后附加'!'作为每个目录项结尾。如果目录未满，则以'0'填充剩余磁盘位。

```
//写入目录
int p = 0;
for (i = 0; i < findex.num; i++) {
    *(ldisk[3] + p++) = findex.index[i].fdcpnum + '0';
    for (j = 0; j < strlen(findex.index[i].fname); j++, p++)
        *(ldisk[3] + p) = findex.index[i].fname[j];
    *(ldisk[3] + p++) = '!';
}
for (; p < 20; p++) {
    *(ldisk[3] + p) = '0';
}
```

打开和写入文件使用 freopen 函数 fclose 函数实现。最后将输出流交回控制台即可。

int read_from_file()

从 ldisk.txt 中读取硬盘内容。使用系统提供的 fseek 函数和 ftell 函数获取该文件的长度，当不满足上文提到的 243 个字节大小时判定为硬盘文件损坏，为空则返回-1，有内容但损坏则返回-2。其余交由 init 函数来进行处理，将在 shell 部分详细说明。

文件系统

文件系统位于 I/O 系统之上。文件系统需提供如下函数：create，destroy，open，read，write，lseek，directory。文件系统的定义如下：

```
// 文件系统

/*根据指定的文件名创建新文件
*成功，返回0
*无空闲描述符，返回-1
*目录分配不成功，返回-2
*没有空盘符，返回-3
*/
int create(char* filename);

/*删除指定文件
*删除成功，返回0
*没有该文件，返回-1
*/
int destroy(char* filename);

/*打开文件。该函数返回的索引号可用于后续的read,write,lseek,或close操作。
*打开成功，返回0
*没有该文件，返回-1
*打开文件已满，返回-2
*/
int open(char* filename);
```

```
/*关闭指定文件
*成功关闭，返回0
*文件未打开，返回-1
*/
int close(int index);

/*从指定文件顺序读入count个字节mem area指定的内存位置。读操作从文件的读写指针指示的位置开始
*成功读取，返回0
*文件未打开，返回-1
*到达文件尾，返回-2
*/
int read(int index, char* mem_area, int count);

/*把mem_area指定的内存位置开始的count个字节顺序写入指定文件。写操作从文件的读写指针指示的位置开始
*写入成功，返回0
*文件未打开，返回-1
*count过大导致覆盖后续文件，返回-2
*/
int write(int index, char* mem_area, int count);
```

```
/*把文件的读写指针移动到pos指定的位置。pos是一个整数，表示从文件开始位置的偏移量。
*文件打开时，读写指针自动设置为0。每次读写操作之后，它指向最后被访问的字节的下一个位置。
*lseek能够在不进行读写操作的情况下改变读写指针能位置。
*成功移动，返回0
*文件未打开，返回-1
*移动超过文件尾，返回-2
*/
int lseek(int index, int pos);

//列表显示所有文件及其长度
int directory();
```

int create(char* filename)

创建文件。找一个空闲文件描述符扫描 ldisk[0]～ldisk[k-1]；在文件目录里为新创建的文件分配一个目录项；在分配到的目录项里记录文件名及描述符编号。无空闲描述符，返回-1，目录分配不成功，返回-2，没有空盘符，返回-3。

根据 fdcpsize 顺序扫描文件描述符，当发现某一个描述符块号数组中状态位为 0，即空闲时，将其分配给此文件。

```
//寻找空描述符
for (i = 1; i < fdcpsize; i++) {
    if (fdcp[i].fpos[2] == 0) {
        startpos = i;
        fdcp[i].fpos[2] = 1;
        break;
    }
}
//如无空描述符，返回-1
if (startpos == 0)
    return -1;
```

因为已经给目录留好了空位，对于文件名大小设有限制（详见实验反思），故有空闲描述符即代表有空闲目录。只需将文件描述符、文件名等信息写入下一个目录项即可。将目录对应的文件描述符状态位置一，以免这是第一个用户文件时使用了目录文件的描述符。

```
//分配目录项
//如目录分配不成功，返回-2
findex.index = (Index*)realloc(findex.index, (findex.num + 1) * sizeof(Index));
if (findex.index == NULL)
    return -2;
else findex.num++;
```

```
//创建新目录
fdcp[0].fpos[2] = 1;
findex.index[findex.num - 1].fdcpnum = startpos;
findex.size += strlen(filename) + 2;
fdcp[0].len = findex.size;
findex.index[findex.num - 1].fname = (char*)malloc(strlen(filename) * sizeof(char));
strcpy(findex.index[findex.num - 1].fname, filename);
```

根据位图寻找空闲盘符分配磁盘空间，如无空盘符则返回-3，交由 shell 处理。此部分放在写入目录之前，故如没有空盘符目录中也不会生成对应目录项。

```
//分配空间
int have = 0;
for (i = 0; i < L; i++) {
    //扫描位图，寻找空盘符
    if (bmp[i] == 0) {
        fdcp[startpos].fpos[0] = i;
        fdcp[startpos].fpos[1] = 0;
        bmp[i] = 1;
        have = 1;
        break;
    }
}
//没有空盘符，返回-3
```

int destroy(char* filename)

删除文件。在目录里搜索该文件的描述符编号，删除该文件对应的目录项并更新位图，释放文件描述符，没有该文件，则返回-1，成功删除则返回 0。

扫描目录，使用 strcmp 函数来寻找文件名对应的文件。如果未找到则返回-1。找到后提取文件描述符序号。

根据其文件描述符，对应位图置空，对应磁盘处置零，释放文件描述符（状态位置零）。

```
//删除位图、文件、文件描述符
bmp[fdcp[fdcpnum].fpos[0]] = 0;
for (i = 0; i < fdcp[fdcpnum].len; i++) {
    ldisk[fdcp[fdcpnum].fpos[0]][fdcp[fdcpnum].fpos[1]+i] = '0';
}
fdcp[fdcpnum].len = 0;
for (i = 0; i < 3; i++)
    fdcp[fdcpnum].fpos[i] = 0;
```

根据文件名，计算需要删除的目录长度，删除对应目录项，更新目录总长，将其后的目录项前移。如果此时没有目录项，则将目录的文件描述符一同释放。

```
//删除目录
int deslen = 0;//需要删除的目录长度
findex.index[indexnum].fdcpnum = 0;
deslen = strlen(findex.index[indexnum].fname);
findex.index[indexnum].fname = NULL;
findex.size -= deslen + 2;
fdcp[0].len = findex.size;
//后续目录项前移
for (i = indexnum; i < findex.num; i++) {
    findex.index[i] = findex.index[i + 1];
}
findex.num--;
if (findex.num == 0)
    fdcp[0].fpos[2] = 0;
```

int open(char* filename)

读写缓冲区的大小等于一个磁盘存储块。打开文件时需要进行的操作如下：搜索目录找到文件对应的描述符编号；在打开文件表中分配一个表目；在分配到的表目中把读写指针置为 0 ，并记录描述符编号；读入文件的第一块到读写缓冲区中；返回分配到的表目在打开文件表中的索引号

同 destroy 函数一样，根据文件名在目录中搜索对应文件，如未找到则返回-1。

维护打开文件表。在分配表目时，同时判断已经打开的表目中是否含有该文件，如果已经打开，则返回-2，如果表目已满，则返回-3。但已经打开优先于表目已满。

```
//没有该文件，返回-1
if (status == 0)
    return -1;
for (i = 0; i < 5; i++) {
    //文件已经打开，返回-2
    if (oft[i].fpos == fdcpnum) {
        return -2;
    }
    if (oft[i].fpos == 0) {
        oftnum = i;
        break;
    }
}
//打开文件已满，返回-3
if (oftnum == -1)
    return -3;
```

成功分配表目以后，则给 buff 分配空间，读写指针置零，读入对应第一块磁盘数据。返回表目序号。

```
oft[oftnum].buff = (char*)malloc(B * sizeof(char));
oft[oftnum].fpos = fdcpnum;
oft[oftnum].rwpos = 0;
//读入文件盘块
read_block(fdcp[fdcpnum].fpos[0], oft[oftnum].buff);
return oftnum;
```

int close(int index)

关闭文件时需要进行的操作如下：把缓冲区的内容写入磁盘；释放该文件在打开文件表中对应的表目；返回状态信息。

如果对应表目的描述符为 0，则表示没有此打开的文件，返回-1。

如果 buff 区非空，则调用 write_block 函数将 buff 区写入磁盘，之后释放表目，将一切置空。

int read(int index, char* mem_area, int count)

同 close 相同，如果未打开则返回-1。

如果 buff 区为空，则分配 B 大小的空间。此操作用来防止刚进行过 write 操作，将 buff 区写入后置空，导致 read_block 报错的问题。

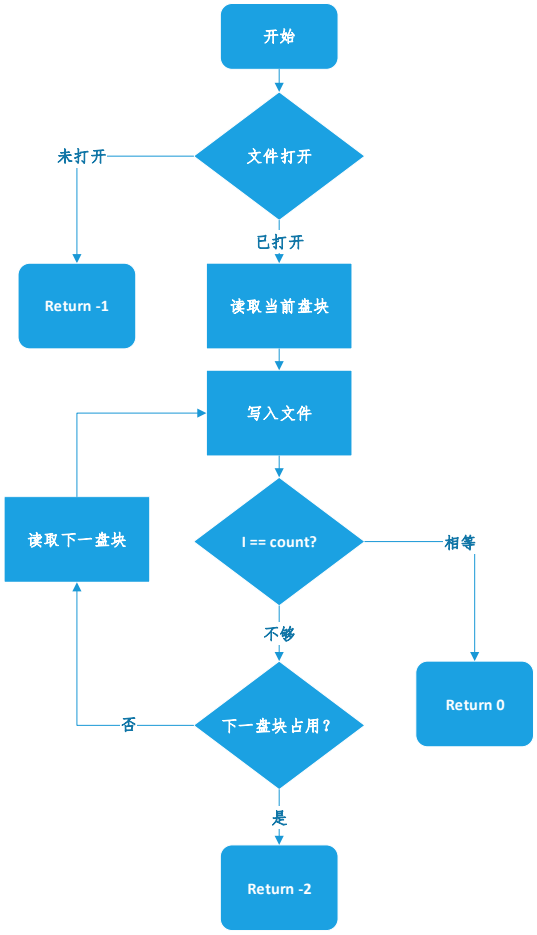
设置 int 型变量 ds:sp 表示文件读写位置，ds 为文件所在盘号，sp 为对应盘号的偏移。其中 sp 的数值为文件描述符盘块数组中的偏移，加上打开文件表中读写指针。

在不超过文件总长的情况下，按磁盘块读取文件。此时维护两个指针：mem_area 中的下一个字符位置 i，以及 sp。当 sp 超过盘块大小时，读取下一个盘块，并将 sp -= B。不断增加 i 的数值，直到达到 count 或者超过文件总长停止。最后更新文件读写指针。如果超过文件总长时也未达到 count，则返回-2，交由 shell 处理。

```
while (i < count) {
    read_block(ds++, oft[index].buff);
    for (; i < count && i + sp < B && i + oft[index].rwpos < filelen; i++)
        mem_area[i] = oft[index].buff[i + sp];
    //读够count，返回0
    if (i == count)
        break;
    //到达文件尾，返回-2
    if (i + oft[index].rwpos == filelen) {
        mem_area[i] = '\0';
        oft[index].rwpos = filelen;
        return -2;
    }
    sp -= B;
}
oft[index].rwpos += i;
mem_area[i] = '\0';
return 0;
```

int write(int index, char* mem_area) /(int index, char* mem_area, int count)

写操作的初始设置同读操作相同，文件未打开则返回-1，同样使用 ds:sp 来写入文件。函数流程如下：



不同的是文件长度获取到初值后，不作为终止的判断依据。当写满当前 buff 后，将目前的 buff 写入盘块，如果为写满 count，则判断下一个盘块是否被占用，如果占用返回-2，否则继续写入，直到写满 count 个字符。

```
while (i < count) {
    read_block(ds++, oft[index].buff);
    for (; i < count && sp < B; i++, sp++) {
        oft[index].buff[sp] = mem_area[i];
        oft[index].rwpos++;
    }
    write_block(ds - 1, oft[index].buff);
    bmp[ds - 1] = 1;
    //写够count长度，返回0
    if (i == count)
        break;
    //count过大，覆盖下一个文件，返回-2
    int a = ds - fdcp[oft[index].fpos].fpos[0];
    if (bmp[ds] != 0 && filelen - (a * B) <= 0) {
        fdcp[oft[index].fpos].len = oft[index].rwpos;
        oft[index].buff = NULL;
        return -2;
    }
    //尚未写满count，重新读入下一块磁盘
    sp -= B;
}
```

最后清空 buff，如果读写指针已经大于之前记录的文件长度，则将文件长度更新为读写指针的大小。

int lseek(int index, int pos)

更改读写指针。文件未打开则返回-1。同上述函数相同，设置好 sp 后，判断是否超过文件尾，是则返回-2，读写指针不移动，否则将读写指针移动到 pos 位置，返回 0。

int directory()

显示文件目录，包含目录序号，文件名，文件大小信息。

Shell

操纵文件系统的外壳程序或者一个菜单驱动系统。

系统定义如下：

```
//shell

//对内存、目录、描述符初始化
void init();

/*执行指令
*正常执行，返回0
*退出，返回-1
*错误指令，返回-2
*/
int exec(char str[]);

//菜单
void menu();
```

void init()

调用 I/O 系统中的 read_from_file()函数，根据其返回值进行操作。如果返回-1 或-2，表示磁盘文件错误，将一切系统置空。如果返回 0，说明磁盘文件正常，进行初始化。

位图和文件描述符的初始化较为简单，此两项长度和位置固定，直接从相应的磁盘位置读入即可。

```
//初始化位图
for (i = 0; i < B; i++) {
    bmp[i] = (int)ldisk[0][i] - '0';
}
//初始化文件描述符
for (i = 0; i < fdcpsize; i++) {
    fdcp[i].len = (int)(*(ldisk[1] + i * 4) - '0');
    for (int j = 0; j < 3; j++) {
        fdcp[i].fpos[j] = (int) * (ldisk[1] + j + 1 + i * 4) - '0';
    }
}
```

目录的初始化较为复杂。首先根据 fdcp[0]获取目录总长 size，维护一个 int 型变量 count，表示当前已经读取的目录长度。

随后初始化目录项。给目录项分配空间，同时将目录数量 num 加一。维护另一个 int 型变量 num，表示此目录项的长度，当对此目录项操作完毕后，count += num。维护 int 型变量 p，表示目前正在读取的位置。

如果 num == 0，表示正在读取新的目录项，则 p 位置的数据为当前目录项文件描述符序号，否则为文件名。使用字符型数组 tmp 暂存当前目录项的文件名。当 p 位置为'!'时表示目录项读取完毕，将 tmp 最后添加'\0'，赋值给 fname。

不断执行上段所述流程，知道 count 与 size 相等，此时完成目录初始化。

具体代码实现如下：

```
//初始化目录
int p = 0, count = 0;
findex.size = fdcp[0].len;
for (i = 0; count < fdcp[0].len - 1; i++) {
    int num = 0;
    j = 0;
    findex.index = (Index*)realloc(findex.index, ++findex.num * sizeof(Index));
    while (*(ldisk[3] + p) != '!') {
        if (num == 0) {
            findex.index[i].fdcpnum = *(ldisk[3] + p) - '0';
        }
        else
        {
            tmp[i][j] = *(ldisk[3] + p);
            j++;
        }
        p++;
        num++;
    }
    tmp[i][j] = '\0';
    findex.index[i].fname = tmp[i];
    p++; num++;
    count += num;
}
```

int exec(char str[])

分析 str，进行分词，执行指令。分词算法与 Unix Shell 实验中分词算法相同。创建字符数组 com[4][30]作为 Unix Shell 实验中的*args[]。当遇到 str 中空格时表示已经进入下一个指令参数，使用下一个字符数组存储。实现代码：

```
while (i < len) {
    for (j = 0; str[i] != ' ' && str[i] != '\0'; i++, j++) {
        com[k][j] = str[i];
    }
    i++;
    com[k][j] = '\0';
    k++;
}
```

根据 com[0]中字符串的不同，执行不同指令。指令内容包含所有文件系统中描述过的函数。对于不同的指令，有不同的函数调用和异常值处理。以 create 指令为例，其处理代码如下：

```
if (strcmp(com[0], "create") == 0) {
    switch (create(com[1])) {
    case 0:strcpy(rtn, com[1]); strcat(rtn, "创建成功"); break;
    case -1:strcpy(rtn, "无空闲描述符"); break;
    case -2:strcpy(rtn, "目录分配不成功"); break;
    case -3:strcpy(rtn, "没有空盘符"); break;
    default:strcpy(rtn, "未知情况"); break;
    }
}
```

其中 rtn 为字符型数组，用来存储提示信息。例如创建文件成功后，提示信息为对应文件名+创建成功。

如果 com[0]中内容不包含任何能够调用的函数，则提示“指令不支持”，返回-2；如果指令内容为 exit，则返回-1。此处的返回值都交由 menu 处理。

void menu()

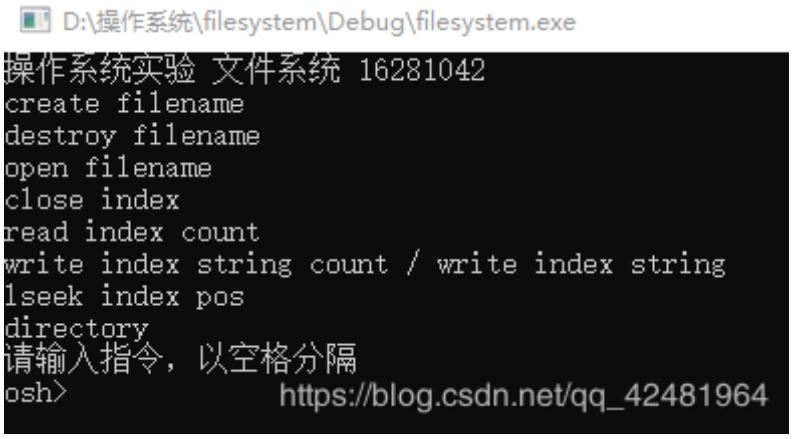
系统菜单。显示支持的指令格式，输入指令提示，个人信息。调用 exec(str)函数，当其返回-1 时，返回主函数，写入磁盘文件，程序结束。

```
void menu() {
    char com[50];
    int status = 1;
    printf("操作系统实验 文件系统 16281042\n");
    printf("create filename\ndestroy filename\nopen file
    printf("请输入指令，以空格分隔\n");
    while (status >= 0) {
        fflush(stdin);
        printf("osh>");
        gets_s(com);
        status = exec(com);
    }
}
```

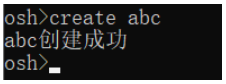
实验结果

当磁盘文件为空时

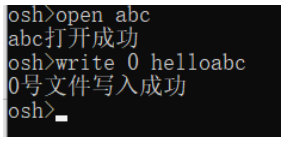
启动程序



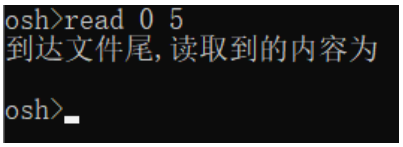
正确显示硬盘文件状态，显示指令格式及个人信息。
由于磁盘为空，应当创建文件。输入 create abc



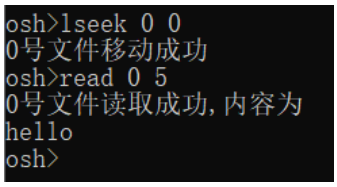
系统提示该文件创建成功。
打开该文件，写入内容：helloabc



系统提示操作成功
读取该文件的内容

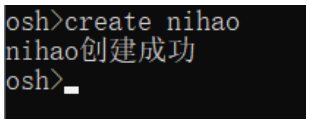


正确提示了错误信息。由于刚进行完写操作，读写指针在文件尾部，故应当先移动读写指针到文件头。再进行读操作。



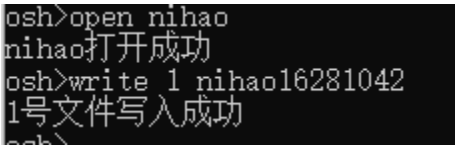
刚才对于 abc 文件写入了“helloabc”，将读写指针移动到 0，即文件头，读取后续 5 个字符，即为“hello”，读取正确。

在不关闭当前文件的情况下，新建另一文件 nihao

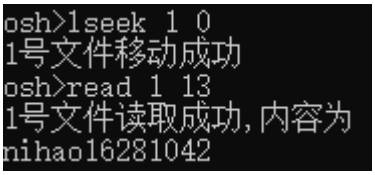


提示创建成功。

打开此文件，写入“nihao16281042”



系统提示写入成功。此字符串显然大于磁盘位数 10，读取文件验证写入成功。同样先移动读写指针到文件头。

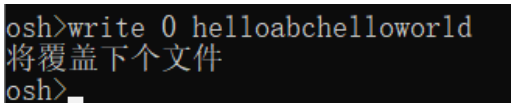


可以看到读取内容正确。

显示文件目录

可以看到文件名称及长度均正确。

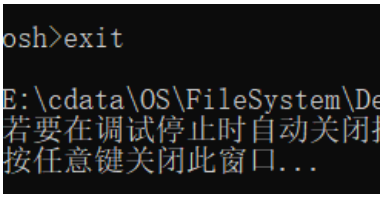
由于系统中优先将最近的空闲盘号分配给新创建的文件，故 nihao 必然在 abc 的紧接着的后一个磁盘块中。在 abc 中写入超过 10 字符的信息以验证系统正确性。



系统提示“将覆盖下个文件”，提示正确。此时移动读写指针到文件头，查看文件内容。

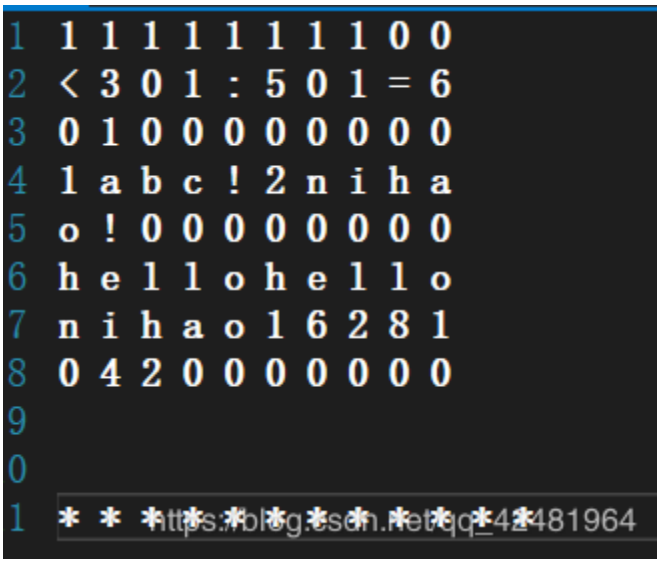
由于刚才写入时读写指针在当时的文件尾，故刚才写入是在 abc 后续添加。所以显示“hellohello”为正确值。显示目录，文件长正确。

输入 exit，生成 ldisk.txt 文件，查看磁盘中信息。



程序正常退出。

打开磁盘文件。内容如下：



首行为位图，剩余最后两块空盘，与实际磁盘情况符合；2~3 行为文件描述符，<对应 ASCII 码 60，与数字 0 相差 12，目录长度正确，盘块数组为 301，盘块 3 偏移 0 处开始，占用状态为 1 占用，目录文件描述正确；后续文件描述符：与 0 相差 10，=与 0 相差 13，均与实际文件长符合；4~5 为目录，共两项，文件描述符分别为 1、2，文件名正确，以！结尾，目录正确；后续文件中，文件内容与测试时相符。

至此，空盘情况下测试完毕。

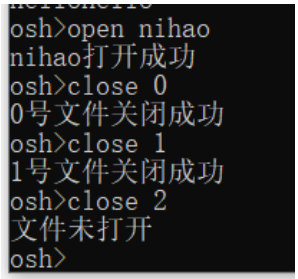
当磁盘文件正常时

完成上面空盘测试后，已经生成了正确的磁盘文件，此时再次运行程序，验证磁盘读

取。

先后查看目录，打开文件 abc，读取超过文件大小的文件内容，提示信息正确，显示内容与磁盘中内容相符。

打开文件 nihao，测试文件关闭。分别关闭文件 abc，nihao，和一个不存在的打开文件索引。



前两个文件关闭成功，后一个文件提示未打开。提示正确。

退出，查看磁盘文件

此时磁盘文件与上次文件相同。

至此，磁盘文件正常的测试完毕。

实验反思

1. 限于磁盘大小，原则上将文件名限制在 5 个字符以内。但对于文件名大小没有附加判断，可以超过此限制，后果是可能出现有空文件描述符但目录文件已经写满的情况。

2. 文件目录的初始化使用了很多的指针和提示性的变量，应该可以后续优化。
3. 由于指令分词时以空格作为判断符，故此方法写入文件时不能有空格。但磁盘读取时以 scanf(“%c ”….)的方法读入，故文件中可以出现空格。限于时间关系，没有单独写使用字符串变量调用 write 的方法，而是直接将输入的指令直接写入，但后续可以实现。

实验总结

文件系统实验十分复杂，我用了相当长的时间来自主完成此实验，让我对文件系统的实现和操作逻辑有了很深入的理解。

由于此实验相当复杂，希望能够在以后的课程实验设计中给此实验分配更长的时间，或者继续简化内容。