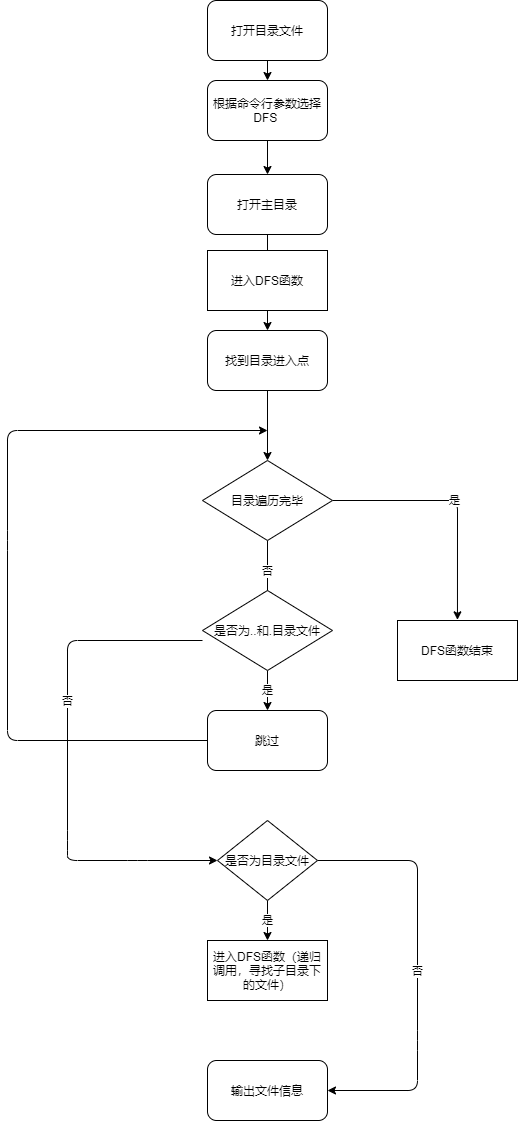
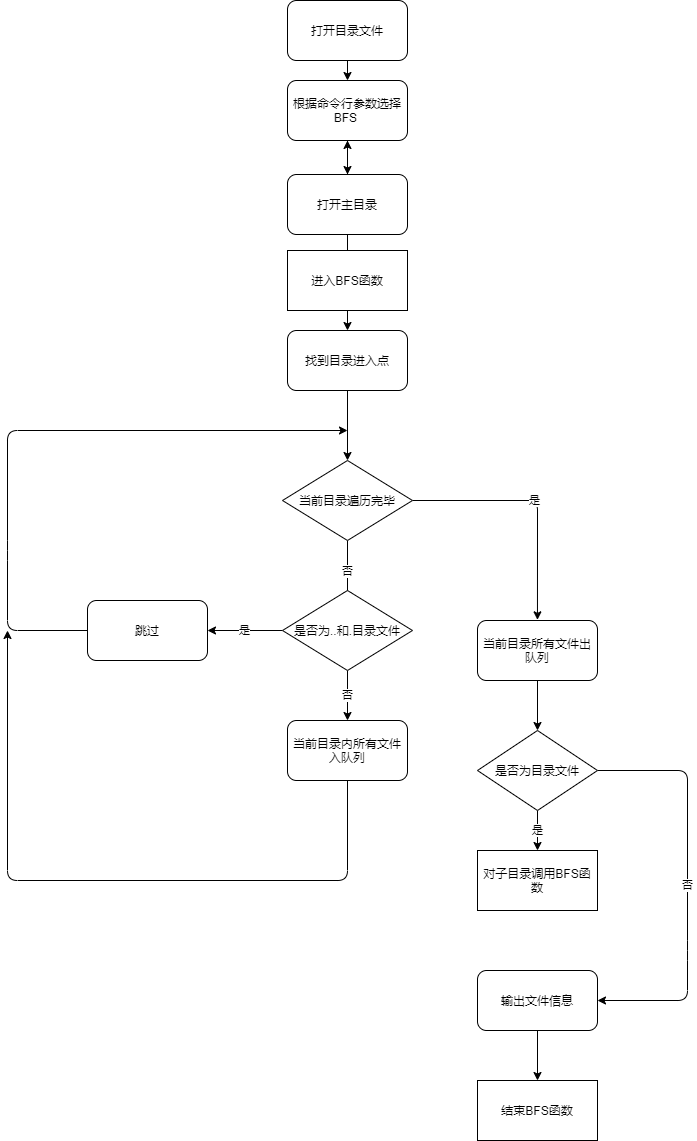
**文件系统扩展实验**

1. 实验名称：文件系统扩展实验
2. 实验要求：对【程序5\_9】进行扩展，要求参数为目录名，且其下至少有三层目录，分别用深度遍历及广度遍历两种方法对此目录进行遍历，输出此目录下所有文件的大小及修改时间。
3. 实验目的：熟悉linux文件系统的结构，即“一切皆文件”的设计思想，对于linux系统而言，目录同样是一个文件，甚至输出输入设备也被当作一个文件来处理；linux文件系统采用单个目录树的结构，不同于windows系统多个树根的设计思路。还需要掌握通过stat系统来获取文件相关信息的方法，以及如何通过readdir系统来遍历文件。其中还需要理解从目录中读取文件的过程，即通过dirent指针中的相对的偏移量、文件名等来找到目录下对应的文件。
4. 实验内容：程序的设计部分，

（1）DFS遍历文件目录的设计思路为，首先先了解DFS的思维，对于一个树而言，如果遍历到非叶子节点，那么往下遍历，一直到遍历到叶子节点为止，对于文件系统而言，可以理解为，遍历目录内的子目录文件和子文件，直到所有的非文件都被遍历完为止。对于DFS的实现，需要采用一种栈的结构来实现，将访问过的节点标记，入栈，等所有相关节点都被访问，就出栈，可以采用对文件名进行入栈出栈的操作，需要自己定义一个栈的数据结构，但我采用的是系统栈，也就是递归调用程序，一旦是目录文件，则递归地调用dfs函数来调用它的下层，一旦是非目录文件，则选择输出，具体的函数流程如下图所示



1. BFS与DFS的不同在于，DFS旨在不管有多少条岔路，先一条路走到底，不成功就返回上一个路口然后就选择下一条岔路，而BFS旨在面临一个路口时，把所有的岔路口都记下来，然后选择其中一个进入，然后将它的分路情况记录下来，然后再返回来进入另外一个岔路，BFS对于一颗树而言，是采用层序遍历的一个过程，对于一个文件目录而言也是如此，把主目录遍历完之后，才能把子目录遍历完，在实现方面，队列的数据结构正好满足了这种需求，实现方面，我先将各个各个目录下的所有文件（包括目录文件）入队列，这一层全部入队列之后，就进行出队列的操作，当出队列的是一个目录时，把它的子目录和子文件都入队列，当出队列的是一个文件时，输出它的相关信息即可，当然，对于不同层级的目录的操作也采用的递归的思路实现，程序的流程图如下所示：



1. 程序：

//@author:lazy1

//@email:674194901@qq.com

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

#include <time.h>

#include <string.h>

#include <ctype.h>

#define BFS 0

#define DFS 1

//队列的数据结构

typedef struct Quene

{

char \*filename[10];

int front;

int rear;

}Quene;

DIR \*dirp1;

struct dirent \*direntp;

struct stat statbuf;

int stats;

//在原有的函数上进行改进,用作BFS的check函数

static int GetFileSizeTime(const char \*filename)

{

struct stat statbuf;

if(stat(filename,&statbuf)==-1) //取 filename 的状态

{

printf("Get stat on %s Error:%s\n",filename,strerror(errno));

return(-1);

}

if(S\_ISDIR(statbuf.st\_mode))

return 1; //判断是否是目录文件

if(S\_ISREG(statbuf.st\_mode)) //判断是否是普通文件

printf("%ssize:%ldbytes\tmodifiedate: %s\n",filename,statbuf.st\_size,ctime(&statbuf.st\_mtime)); //输出文件的大小和最后修改时间

return 0;

}

//DFS,采用递归实现

void DirDfs(DIR \*dirp,char\*lastpath)

{

while((direntp = readdir(dirp))!=NULL)

{

char path[1024];

memset(path, 0, 1024);

strcpy(path, lastpath);

strcat(path, "/");

strcat(path, direntp -> d\_name);

if (strcmp(direntp->d\_name,"..")==0||strcmp(direntp->d\_name,".")==0)

{

continue;

}

if(stat(path,&statbuf)==-1)

{

printf("Get stat on %s Error:%s\n",path,strerror(errno));

break;

}

if(S\_ISDIR(statbuf.st\_mode))

{

DIR \*dirp2;

dirp2=opendir(path);

DirDfs(dirp2,path);

continue;

}

if(S\_ISREG(statbuf.st\_mode))

printf("%ssize:%ldbytes\tmodifiedat%s\n",path,statbuf.st\_size,ctime(&statbuf.st\_mtime));

}

return;

}

//初始化队列

Quene\* initQuene()

{

Quene \*Q=(Quene\*)malloc(sizeof(Quene));

for (int i = 0; i < 10; i++)

{

Q->filename[i]=(char\*)malloc(80);

}

Q->rear=Q->front=0;

return Q;

}

//BFS

void DirBfs(DIR \*dirp,char\* lastpath,Quene\*Q)

{

int ans;

while((direntp = readdir(dirp))!=NULL)

{

char path[1024];

memset(path, 0, 1024);

strcpy(path, lastpath);

strcat(path, "/");

strcat(path, direntp -> d\_name);

if(strcmp(direntp->d\_name,"..")!=0&&strcmp(direntp->d\_name,".")!=0)

{

//文件名入队列

// printf("%s\n",path);

strcpy(Q->filename[Q->rear],path);

// Q->filename[Q->rear]==path;

Q->rear++;

}

}

//出队列

while (Q->front!=Q->rear)

{

//如果是目录，子文件入队列

if(GetFileSizeTime(Q->filename[Q->front])==1)

{

DIR \*dirp2;

dirp2=opendir(Q->filename[Q->front]);

ans=Q->front;

Q->front++;

DirBfs(dirp2,Q->filename[ans],Q);

}

//如果是文件

else

{

Q->front++;

}

}

return;

}

int main(int argc,char \*\*argv)

{

//三个参数 最后一个用来选择遍历类型

if((dirp1=opendir(argv[1]))==NULL)

//打开目录，将打开的目录信息放至 dirp 中，若为空，则打开失败

{

printf("Open Directory %s Error:%s\n",argv[1],strerror(errno));

exit(1);

}

if(atof(argv[2])==DFS)

{

DirDfs(dirp1,argv[1]);

}

if(atof(argv[2])==BFS)

{

Quene\* Q=initQuene();

DirBfs(dirp1,argv[1],Q);

}

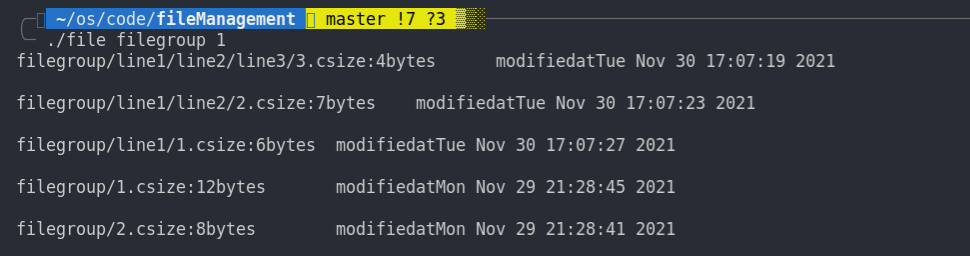
closedir(dirp1);

exit(1);

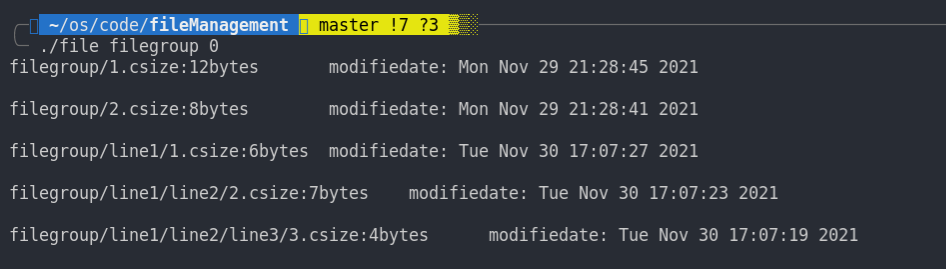
}

1. 运行结果：（进行反白处理后截图）

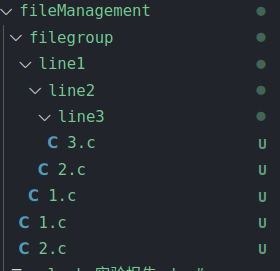
DFS:



BFS:



文件树形结构：



1. 实验总结：编程、调试过程中遇到的问题及解决办法。
2. 深度优先搜索的原理混淆：一开始的时候混淆了原理，以为DFS需要先把最底层目录的非目录文件输出，再逐一输出上层的文件，但是在遍历一个文件夹时，先遍历到的并不一定是目录文件，所以只要符合DFS的“一条路走到黑”原则即可
3. 如何实现的遍历文件，解决方式：我看了示例的代码，始终没有发现指针的移动，既然指针不移动如何通过循环遍历各个文件呢，于是我又去看了readdir的实现，发现这个函数调用完之后，文件指针参数会自动移到下一个文件（指针的传址特性），所以在遍历不同的文件目录时，就需要传递不同的指针来实现，每次进入一个新的目录，就需要开一个新的文件指针。
4. 出现刷屏的现象，判断是因为遍历到了一些 没有想到的文件，所以没有报错而是继续的循环。解决方式：当我把DFS的代码写好之后出现了刷屏的现象，原因是因为遍历到了其他的目录文件，导致文件路径出现了问题，影响了后续一系列的操作，但是ls命令并没有发现文件，这是因为linux系统中每个目录下面都有..和.的隐藏文件，用于相对路径相关的操作，也为很多命令带来了便捷，这里需要用ls -a命令来查看这两个文件，对DFS而言，遇到这两个文件时就需要continue进入下一层循环，对于BFS而言，这两个文件不入队列，问题就解决了。
5. BFS如何采用队列进行实现：BFS最常用的实现就是队列，所以我定义了一个队列结构，包含队首指针和队尾指针，里面的数据域采用一个指针数组来存储文件路径，出队时对于不同的路径逐一采用stat结构体来获取相关信息即可，出队列的条件就是两个指针进行了重合，由于不需要考虑队列空的情况，所以这里也简化了很多代码。
6. 这里还遇到的一个bug就是指针数组的初始化问题，对于linux系统而言产生了段错误，但是要查看具体的错误，需要先用unlimit命令来修改core文件的大小限制，这样就会产生core文件，根据文件的编号，就可以确实的知道错误的来源，方便解决bug，我当时的错误来源是没有对队列结构进行malloc分配内存就直接进行了赋值，因此产生了错误。
7. 代码的调试问题：对于递归的操作，理解时比较麻烦，所以需要采用gdb工具进行调试，调试时遇到了很多问题，如传参问题，需要采用set args命令，设置断点需要采用b命令，并且gdb的调试时默认不会对main以外的函数的具体执行过程进行调试的，所以就需要采用设置断点或者s命令来进入对应的函数内部（库函数不能使用，否则gdb会因为在文件夹下找不到函数的定义而报错）。