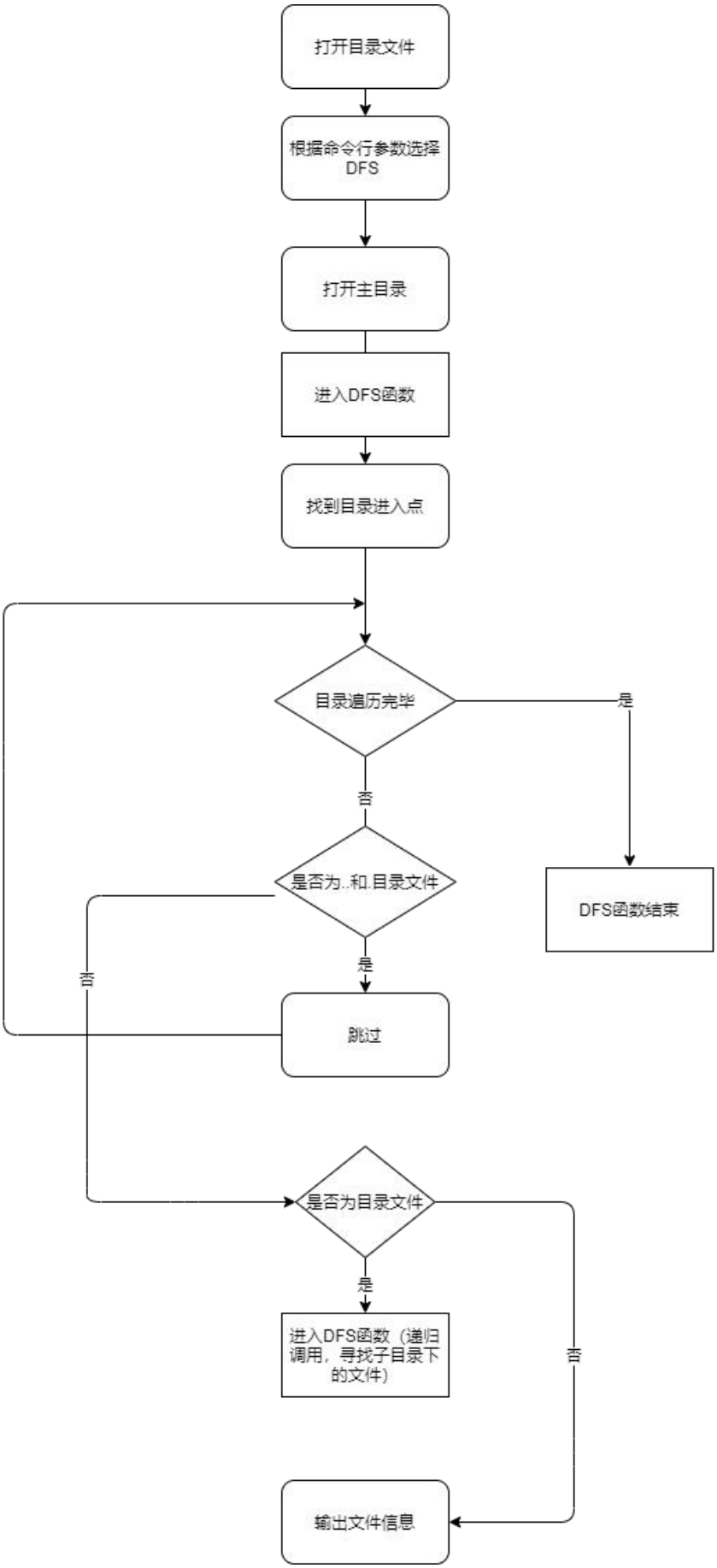
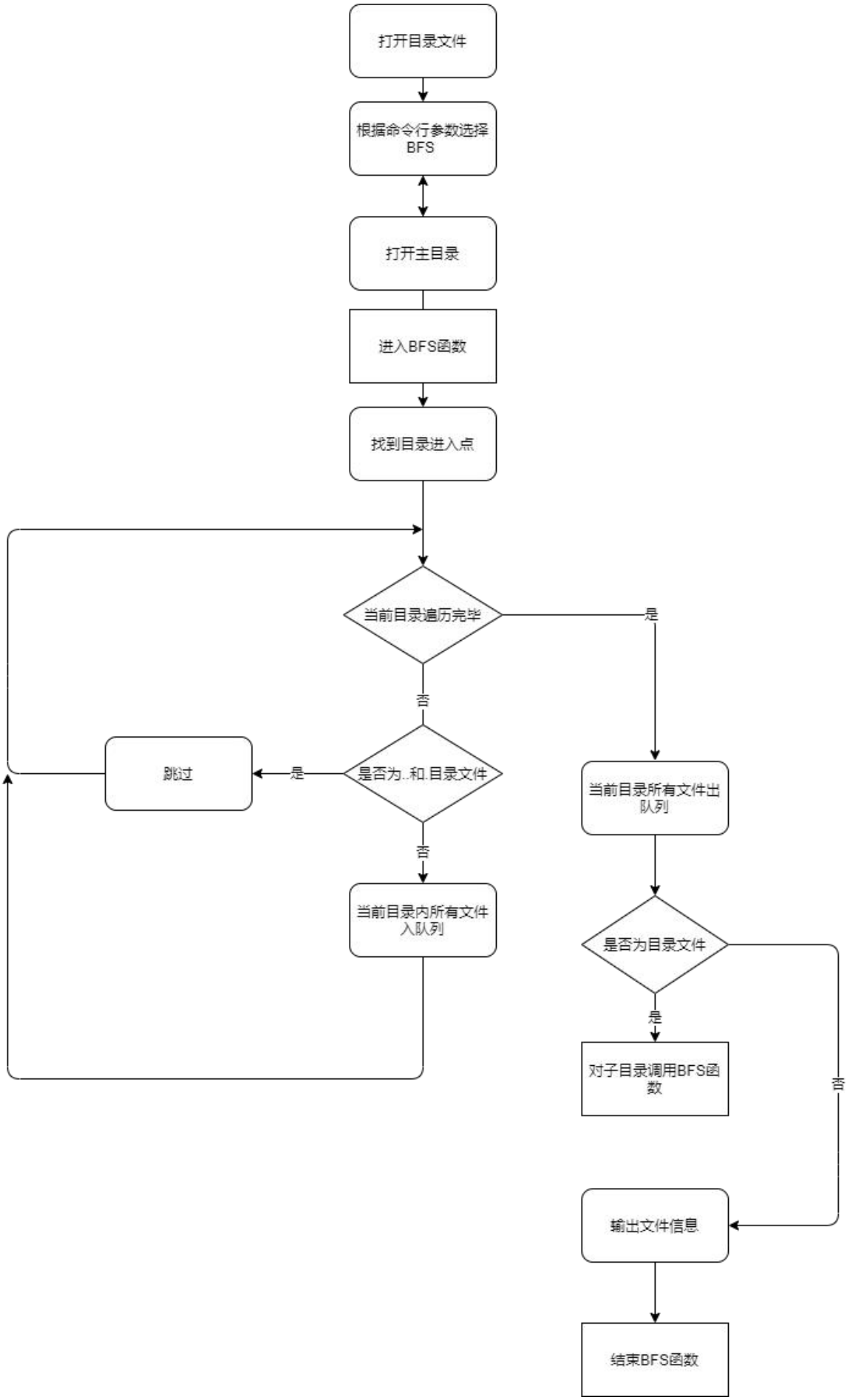
文件系统扩展实验

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


打开目录文件
根据命令行参数选择DFS
打开主目录
进入DFS函数
找到目录进入点
目录遍历完毕
是
否
是否为..和.目录文件
DFS函数结束
是
否
跳过
是否为目录文件
是
否
进入DFS函数（递归调用，寻找子目录下的文件）
输出文件信息

（2）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：


打开目录文件
根据命令行参数选择BFS
打开主目录
进入BFS函数
找到目录进入点
当前目录遍历完毕
是
否
是否为..和.目录文件
是
跳过
否
当前目录内所有文件入队列
当前目录所有文件出队列
是否为目录文件
是
对子目录调用BFS函数
否
输出文件信息
结束BFS函数

5. 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(&statb
uf.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);
```

}

6. 运行结果：（进行反白处理后截图）

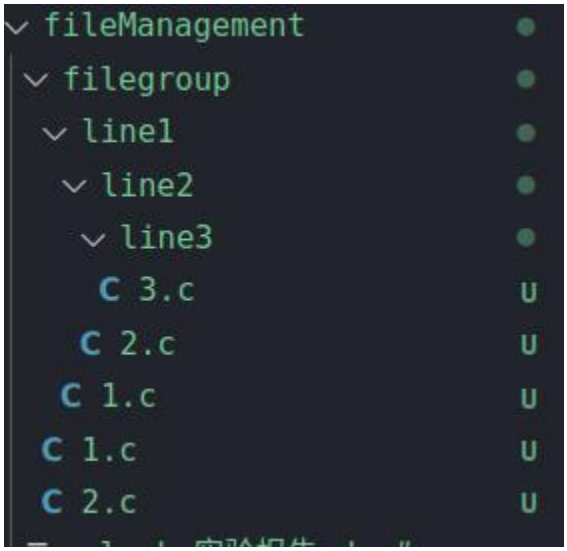
DFS:

```
~/os/code/fileManagement  master !7 ?3
./file filegroup 1
filegroup/line1/line2/line3/3.csize:4bytes      modifiedatTue Nov 30 17:07:19 2021

filegroup/line1/line2/2.csize:7bytes    modifiedatTue Nov 30 17:07:23 2021

filegroup/line1/1.csize:6bytes  modifiedatTue Nov 30 17:07:27 2021

filegroup/1.csize:12bytes       modifiedatMon Nov 29 21:28:45 2021

filegroup/2.csize:8bytes        modifiedatMon Nov 29 21:28:41 2021
```

BFS:

```
~/os/code/fileManagement  master !7 ?3
./file filegroup 0
filegroup/1.csize:12bytes       modifiedate: Mon Nov 29 21:28:45 2021

filegroup/2.csize:8bytes        modifiedate: Mon Nov 29 21:28:41 2021

filegroup/line1/1.csize:6bytes  modifiedate: Tue Nov 30 17:07:27 2021

filegroup/line1/line2/2.csize:7bytes    modifiedate: Tue Nov 30 17:07:23 2021

filegroup/line1/line2/line3/3.csize:4bytes      modifiedate: Tue Nov 30 17:07:19 2021
```

文件树形结构：



7. 实验总结：编程、调试过程中遇到的问题及解决办法。

（1）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

（2）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。

（3）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。

（4）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。

（5）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。

（6）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）。