Unix程序设计

实验（三）编写程序myfind

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2022年10月17日
学　　院	信息学院
课程名称	Unix程序设计

实验（三）编写程序myfind

目录

1 实验内容 **1**

1.1 命令语法 1

1.2 命令语义 1

2 程序设计与实现 **1**

2.1 程序设计 1

2.2 程序实现 1

3 程序测试 **2**

4 问题与改进 **2**

附录：代码清单 **4**

.1 Makefile 4

.2 myfind.c 4

1 实验内容

编写程序myfind

1.1 命令语法

myfind ⟨pathname⟩ [-comp ⟨filename⟩ | -name ⟨str⟩⋯]

1.2 命令语义

1. myfind ⟨pathname⟩的功能：除了具有与程序4-7相同的功能外，还要输出在⟨pathname⟩目录子树之下，文件长度不大于4096字节的常规文件，在所有允许访问的普通文件中所占的百分比。程序不允许打印出任何路径名。

2. myfind ⟨pathname⟩ -comp ⟨filename⟩的功能：⟨filename⟩是常规文件的路径名（非目录名，但是其路径可以包含目录）。命令仅仅输出在⟨pathname⟩目录子树之下，所有与⟨filename⟩文件内容一致的文件的绝对路径名。不允许输出任何其它的路径名，包括不可访问的路径名。

3. myfind ⟨pathname⟩ -name ⟨str⟩⋯的功能：⟨str⟩⋯是一个以空格分隔的文件名序列(不带路径)。命令输出在⟨pathname⟩目录子树之下，所有与⟨str⟩⋯序列中文件名相同的文件的绝对路径名。不允许输出不可访问的或无关的路径名。

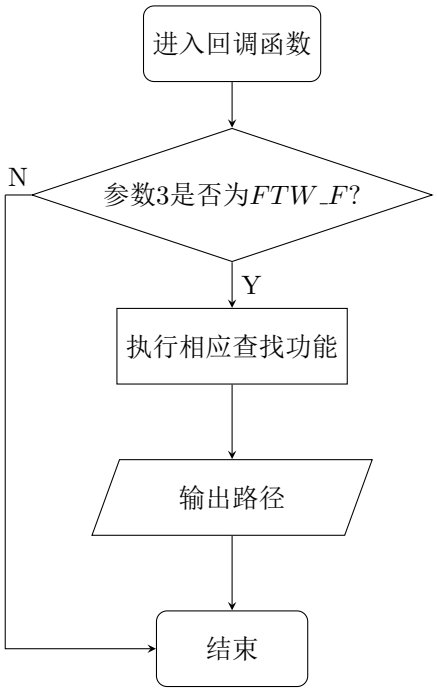
2 程序设计与实现

2.1 程序设计

首先简单地分析课本程序4-7。$myftw$接受一个路径和一个回调函数作为参数。它为路径分配内存，然后调用$dopath$完成路径遍历的实际功能。$dopath$中通过$lstat$获取文件的信息，然后调用回调函数。其中调用的第三个参数将遍历到的文件系统节点分为了文件、目录、不可读的目录、不能执行$stat$的文件4类。然后$dopath$对可读的目录进行递归调用。这样实现了对目录树的先序遍历。

由于$myftw$提供了基本可用的接口来在遍历目录树的时候进行自定义的操作，为了完成实验要求，就需要实现3个不同的回调函数，分别对应3个命令的功能。其中功能1只需要微调4-7中的$myfunc$。其它两个功能则需要在文件可读的情况下完成相应的操作。如图 1。

图 1: 流程图



2.2 程序实现

代码见附录：代码清单中的代码 2。任务1的实现较为简单，只需要在回调函数的参数3为FTW_F且文件类型是S_IFREG时，进一步判断文件的大小是否为4096即可。

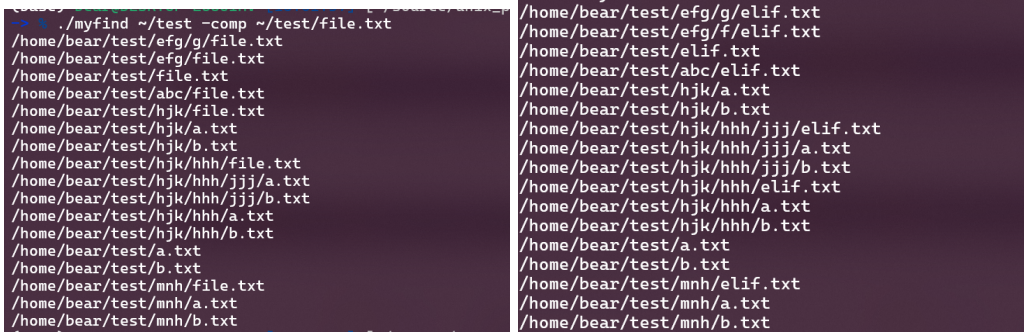
为了实现2和3，需要进行相对路径和绝对路径的转换，以及从路径中提取文件名。这里使用了两个POSIX标准的函数，分别是*realpath*和*basename*。*realpath*可以将相对路径转换为绝对路径，而*basename*可以从路径中提取文件名。

功能2需要对文件内容进行读取，然后逐字节比较是否一致；功能3需要先把参数复制到一个数组中，然后每次遍历到有效的文件就用*basename*获取文件名，然后遍历查找是否在给定的参数中存在。

3 程序测试

为了测试程序的正确性，首先需要准备测试用例。这里使用了一个多层嵌套的目录树，其中某些文件具有相同的内容，另一些文件有不同的内容，如图 3。运行结果如图 2。可见实现是正确的。

图 2: 运行结果



(a) 功能2 (b) 功能3

4 问题与改进

为了进一步提高实现功能3的效率，可以在查找文件名时使用字典树这一数据结构，避免多次重复比较前缀。例如图 4

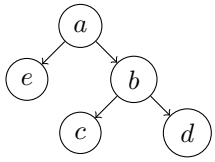
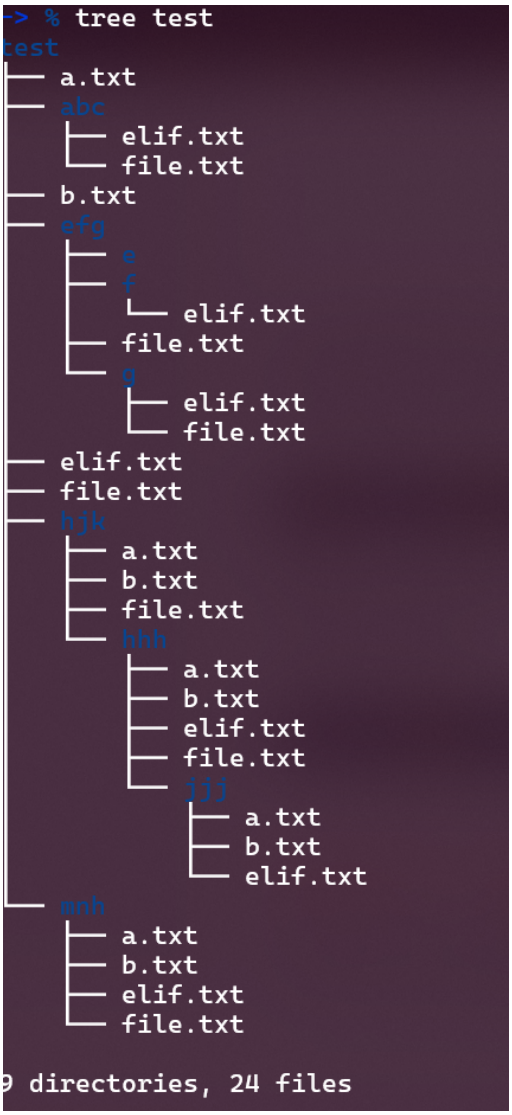


图 4: 字典树示例

图 4是为了查找文件名ae、abc、abd形成的字典树。在查找时，可以从根节点开始，逐层向下遍历。这样这些路径共同的前缀a、ab等都只比较了一遍，能够大大提升效率。

图 3: 测试目录树


-> % tree test
test
├── a.txt
├── abc
│   ├── elif.txt
│   └── file.txt
├── b.txt
├── efg
│   ├── e
│   ├── f
│   │   └── elif.txt
│   ├── file.txt
│   └── g
│   ├── elif.txt
│   └── file.txt
├── elif.txt
├── file.txt
├── hjk
│   ├── a.txt
│   ├── b.txt
│   ├── file.txt
│   └── hhh
│   ├── a.txt
│   ├── b.txt
│   ├── elif.txt
│   ├── file.txt
│   └── jjj
│   ├── a.txt
│   ├── b.txt
│   └── elif.txt
└── mnh
    ├── a.txt
    ├── b.txt
    ├── elif.txt
    └── file.txt
9 directories, 24 files

附录：代码清单

.1 Makefile

代码 1: Makefile

```
OBJS = main.o \
          error.o \
          pathalloc.o \

CC = gcc
CFLAGS = -Wall -g -std=c99

%.o: %.c
       $(CC) $(CFLAGS) -c $< -o $@

myfind: $(OBJS)
       $(CC) $(CFLAGS) $(OBJS) -o myfind

all: myfind

clean:
       rm -f *.o myfind

.PHONY: clean all
```

.2 myfind.c

代码 2: 程序实现

```
#include "apue.h"
#include <dirent.h>
#include <limits.h>

#include <libgen.h> // basename


typedef int (Callback)(const char*, const struct stat*, int);

static Callback simple_statistic;
static Callback content_compare;
static Callback name_compare;

static int myftw(char*, Callback*);

static int dopath(Callback*);

static int compare_file(const char* file1, const char* file2);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot, nless4k;

#define NARGS 16

static char* names[NARGS];
static int name_count = 0;

static char* comp_filename = NULL;
static struct stat comp_stat;

```

```
30 int main(int argc, char* argv[]) {
31     int ret;
32     if (!(argc == 2 || argc >= 4)) {
33         err_quit("usage: myfind <params>");
34     }
35 
36     if (argc == 2)
37     {
38         ret = myftw(argv[1], simple_statistic);
39         ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
40         if (ntot == 0) {
41             ntot = 1;
42         }
43 
44         printf("regular files  = %7ld, %5.2f %%\n", nreg, nreg * 100.0 / ntot);
45         printf("directories    = %7ld, %5.2f %%\n", ndir, ndir * 100.0 / ntot);
46         printf("block special  = %7ld, %5.2f %%\n", nblk, nblk * 100.0 / ntot);
47         printf("char special   = %7ld, %5.2f %%\n", nchr, nchr * 100.0 / ntot);
48         printf("FIFOs          = %7ld, %5.2f %%\n", nfifo, nfifo * 100.0 / ntot);
49         printf("symbolic links = %7ld, %5.2f %%\n", nslink, nslink * 100.0 / ntot);
50         printf("sockets        = %7ld, %5.2f %%\n", nsock, nsock * 100.0 / ntot);
51         printf("smaller than 4k= %7ld, %5.2f %%\n", nless4k, nless4k * 100.0 / ntot);
52 
53     }
54     else if (argc >= 4)
55     {
56         char* pathname = argv[1];
57         if (strcmp(argv[2], "-comp") == 0)
58         {
59             comp_filename = argv[3];
60             if (lstat(comp_filename, &comp_stat) < 0)
61             {
62                 err_sys("lstat error for %s", comp_filename);
63             }
64             ret = myftw(pathname, content_compare);
65         }
66         else if (strcmp(argv[2], "-name") == 0)
67         {
68             for (int i = 3; i < argc; i++)
69             {
70                 names[i - 3] = argv[i];
71                 name_count++;
72             }
73             ret = myftw(pathname, name_compare);
74         }
75         else
76         {
77             err_quit("usage: myfind <params>");
78         }
79     }
80 
81     return ret;
82 }
83 
84 #define FTW_F 1
```

```
85  #define FTW_D 2
86  #define FTW_DNR 3
87  #define FTW_NS 4
88
89  static char* fullpath;
90  static size_t pathlen;
91
92  static int myftw(char* pathname, Callback* func) {
93      fullpath = path_alloc(&pathlen);
94
95      if (pathlen <= strlen(pathname)) {
96          pathlen = strlen(pathname) * 2;
97          if ((fullpath = realloc(fullpath, pathlen)) == NULL) {
98              err_sys("realloc failed");
99          }
100     }
101     strcpy(fullpath, pathname);
102     return (dopath(func));
103 }
104
105 static int dopath(Callback* func) {
106     struct stat statbuf;
107     struct dirent* dirp;
108     DIR* dp;
109     int ret, n;
110     if (lstat(fullpath, &statbuf) < 0) {
111         return (func(fullpath, &statbuf, FTW_NS));
112     }
113     if (S_ISDIR(statbuf.st_mode) == 0) {
114         return (func(fullpath, &statbuf, FTW_F));
115     }
116
117     if ((ret = func(fullpath, &statbuf, FTW_D)) != 0) {
118         return (ret);
119     }
120     n = strlen(fullpath);
121     if (n + NAME_MAX + 2 > pathlen) {
122         pathlen *= 2;
123         if ((fullpath = realloc(fullpath, pathlen)) == NULL) {
124             err_sys("realloc failed");
125         }
126     }
127     fullpath[n++] = '/';
128     fullpath[n] = 0;
129     if ((dp = opendir(fullpath)) == NULL) {
130         return (func(fullpath, &statbuf, FTW_DNR));
131     }
132     while ((dirp = readdir(dp)) != NULL) {
133         if (strcmp(dirp->d_name, ".") == 0 || strcmp(dirp->d_name, "..") == 0) {
134             continue;
135         }
136         strcpy(&fullpath[n], dirp->d_name);
137         if ((ret = dopath(func)) != 0) {
138             break;
139         }
```

```
140     }
141     fullpath[n - 1] = 0;
142     if (closedir(dp) < 0) {
143         err_ret("can't close directory %s", fullpath);
144     }
145     return (ret);
146 }
147 
148 static int simple_statistic(const char* pathname, const struct stat* statptr, int type) {
149     switch (type) {
150     case FTW_F:
151         switch (statptr->st_mode & S_IFMT) {
152         case S_IFREG:
153             nreg++;
154             if (statptr->st_size <= 4096) {
155                 nless4k++;
156             }
157             break;
158         case S_IFBLK:
159             nblk++;
160             break;
161         case S_IFCHR:
162             nchr++;
163             break;
164         case S_IFIFO:
165             nfifo++;
166             break;
167         case S_IFLNK:
168             nslink++;
169             break;
170         case S_IFSOCK:
171             nsock++;
172             break;
173         case S_IFDIR:
174             err_dump("for S_IFDIR for %s", pathname);
175         }
176         break;
177     case FTW_D:
178         ndir++;
179         break;
180     case FTW_DNR:
181         err_ret("can't read directory %s", pathname);
182         break;
183     case FTW_NS:
184         err_ret("stat error for %s", pathname);
185         break;
186     default:
187         err_dump("unknown type %d for pathname %s", type, pathname);
188     }
189     return 0;
190 }
191 
192 static int content_compare(const char* pathname, const struct stat* statptr, int type) {
193     switch (type) {
194     case FTW_F:
```

```
195         switch (statptr->st_mode & S_IFMT) {
196         case S_IFREG:
197         case S_IFBLK:
198         case S_IFCHR:
199         case S_IFIFO:
200         case S_IFLNK:
201         case S_IFSOCK:
202             if (statptr->st_size > 0) {
203                 if (statptr->st_size == comp_stat.st_size &&
204                     compare_file(pathname, comp_filename)) {
205 
206                     char* real = realpath(pathname, NULL);
207                     printf("%s\n", real);
208                     free(real);
209                 }
210             }
211             break;
212         case S_IFDIR:
213             err_dump("for S_IFDIR for %s", pathname);
214         }
215         break;
216     case FTW_D:
217         break;
218     case FTW_DNR:
219         err_ret("can't read directory %s", pathname);
220         break;
221     case FTW_NS:
222         err_ret("stat error for %s", pathname);
223         break;
224     default:
225         err_dump("unknown type %d for pathname %s", type, pathname);
226     }
227     return 0;
228 }
229 
230 static int name_compare(const char* pathname, const struct stat* statptr, int type) {
231     switch (type) {
232     case FTW_F:
233         switch (statptr->st_mode & S_IFMT) {
234         case S_IFREG:
235         case S_IFBLK:
236         case S_IFCHR:
237         case S_IFIFO:
238         case S_IFLNK:
239         case S_IFSOCK:
240             for (int i = 0;i < name_count;i++)
241             {
242                 if (strcmp(basename(pathname), names[i]) == 0)
243                 {
244                     printf("%s\n", pathname);
245                 }
246             }
247             break;
248         case S_IFDIR:
249             err_dump("for S_IFDIR for %s", pathname);
```

```
250         }
251         break;
252     case FTW_D:
253         break;
254     case FTW_DNR:
255         err_ret("can't read directory %s", pathname);
256         break;
257     case FTW_NS:
258         err_ret("stat error for %s", pathname);
259         break;
260     default:
261         err_dump("unknown type %d for pathname %s", type, pathname);
262     }
263     return 0;
264 }
265 
266 static int compare_file(const char* file1, const char* file2)
267 {
268     FILE* fp1 = fopen(file1, "r");
269     FILE* fp2 = fopen(file2, "r");
270     if (fp1 == NULL || fp2 == NULL)
271     {
272         return 0;
273     }
274 
275     char* buf1 = malloc(4096);
276     if (!buf1)
277     {
278         return 0;
279     }
280 
281     char* buf2 = malloc(4096);
282     if (!buf2)
283     {
284         return 0;
285     }
286 
287     int ret = 1;
288     while (1)
289     {
290         int n1 = fread(buf1, 1, 4096, fp1);
291         int n2 = fread(buf2, 1, 4096, fp2);
292         if (n1 != n2)
293         {
294             ret = 0;
295             break;
296         }
297         if (n1 == 0)
298         {
299             break;
300         }
301         if (memcmp(buf1, buf2, n1) != 0)
302         {
303             ret = 0;
304             break;
```

```
305         }
306     }
307     return ret;
308 }
```