4.1

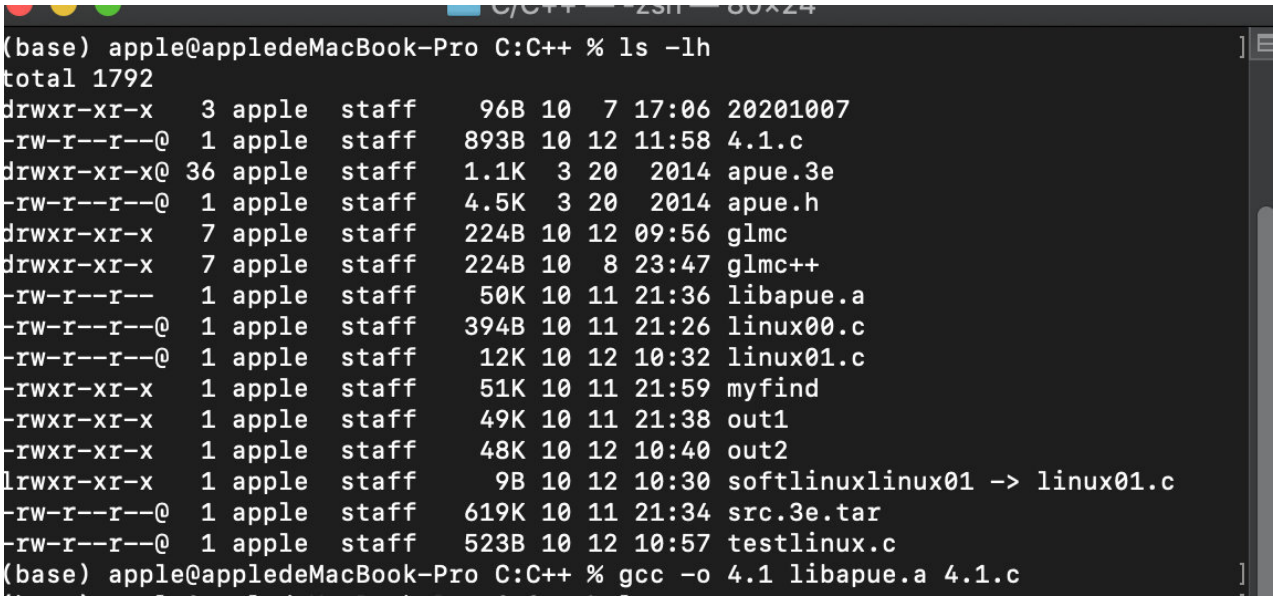
首先明白lstat和stat的细微区别：

针对此题，原有4.1.c中是lstat函数，运行时参数argc[1]是以创建的符号链接时，会返回该符号链接有关的信息，而不是符号链接引用的信息；如果是stat函数，运行时参数argc[1]是以创建的符号链接时，会返回符号链接引用的信息；

结论：lstat：输出symbolic link

stat：输出regular

验证一下：



如图，softlinuxlinux01是用ls -lh建立的软链接，指向linux01.c然后运行4.1.c(此时是用lstat函数)，之后手动修改为stat，然后重新编译运行对比如下：

```
(base) apple@appledeMacBook-Pro C:C++ % ./4.1 softlinuxlinux01
softlinuxlinux01: symbolic link
(base) apple@appledeMacBook-Pro C:C++ % gcc -o 4.1 libapue.a 4.1.c
(base) apple@appledeMacBook-Pro C:C++ % ./4.1 softlinuxlinux01
softlinuxlinux01: regular
```

验证结论 与预测无误

4.3

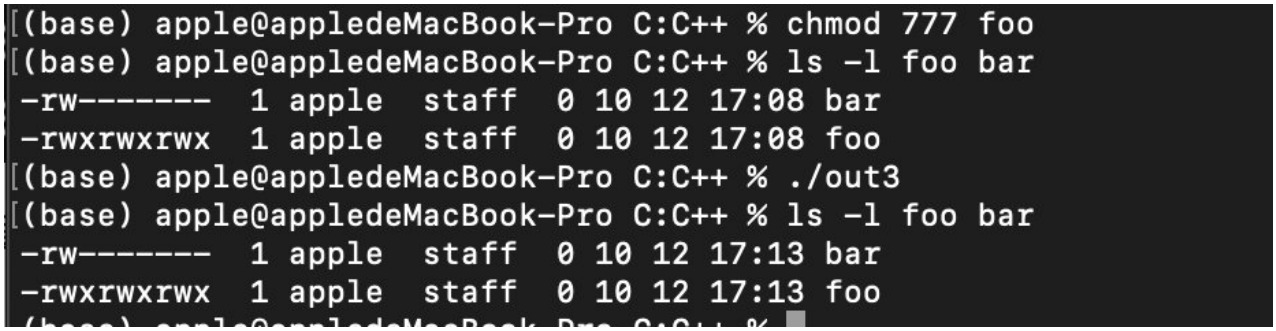
先chmod修改4.1.c的用户权限取消读，再调用ls -l查看发生改变，之后用cat命令试图查看该文件内容，发现权限不够，验证完毕。

```
(base) apple@appledeMacBook-Pro C:C++ % ls -l
total 1896
drwxr-xr-x   3 apple  staff      96 10  7 17:06 20201007
-rwxr-xr-x   1 apple  staff   50568 10 12 12:00 4.1
-rw-r--r--@  1 apple  staff     892 10 12 12:00 4.1.c
drwxr-xr-x@ 36 apple  staff    1152  3 20  2014 apue.3e
-rw-r--r--@  1 apple  staff    4631  3 20  2014 apue.h
drwxr-xr-x   7 apple  staff     224 10 12 09:56 glmc
drwxr-xr-x   7 apple  staff     224 10  8 23:47 glmc++
-rw-r--r--   1 apple  staff   51608 10 11 21:36 libapue.a
-rw-r--r--@  1 apple  staff     394 10 11 21:26 linux00.c
-rw-r--r--@  1 apple  staff   12305 10 12 10:32 linux01.c
-rwxr-xr-x   1 apple  staff   52072 10 11 21:59 myfind
-rwxr-xr-x   1 apple  staff   50680 10 11 21:38 out1
-rwxr-xr-x   1 apple  staff   49552 10 12 10:40 out2
lrwxr-xr-x   1 apple  staff       9 10 12 10:30 softlinuxlinux01 -> linux01.
-rw-r--r--@  1 apple  staff  633344 10 11 21:34 src.3e.tar
-rw-r--r--@  1 apple  staff     523 10 12 10:57 testlinux.c
(base) apple@appledeMacBook-Pro C:C++ % chmod u-r 4.1.c
(base) apple@appledeMacBook-Pro C:C++ % ls -l
total 1896
drwxr-xr-x   3 apple  staff      96 10  7 17:06 20201007
-rwxr-xr-x   1 apple  staff   50568 10 12 12:00 4.1
--w-r--r--   1 apple  staff     892 10 12 12:00 4.1.c
drwxr-xr-x@ 36 apple  staff    1152  3 20  2014 apue.3e
-rw-r--r--@  1 apple  staff    4631  3 20  2014 apue.h
drwxr-xr-x   7 apple  staff     224 10 12 09:56 glmc
drwxr-xr-x   7 apple  staff     224 10  8 23:47 glmc++
-rw-r--r--   1 apple  staff   51608 10 11 21:36 libapue.a
-rw-r--r--@  1 apple  staff     394 10 11 21:26 linux00.c
-rw-r--r--@  1 apple  staff   12305 10 12 10:32 linux01.c
-rwxr-xr-x   1 apple  staff   52072 10 11 21:59 myfind
-rwxr-xr-x   1 apple  staff   50680 10 11 21:38 out1
-rwxr-xr-x   1 apple  staff   49552 10 12 10:40 out2
lrwxr-xr-x   1 apple  staff       9 10 12 10:30 softlinuxlinux01 -> linux01.
-rw-r--r--@  1 apple  staff  633344 10 11 21:34 src.3e.tar
-rw-r--r--@  1 apple  staff     523 10 12 10:57 testlinux.c
(base) apple@appledeMacBook-Pro C:C++ % cat 4.1.c
cat: 4.1.c: Permission denied
```

4.4 因为采用create函数创建，带有O_TRUNC参数，会将原有文件长度截断为0，虽然create 以只写方式打开创建文件，在程序中两次修改unmask屏蔽字，但是文件是在之前创建的，是不会修改已经创建的文件的权限位，证明如下：

```
(base) apple@appledeMacBook-Pro C:C++ % ls -l foo bar
-rw-------  1 apple  staff  19 10 12 17:08 bar
-rw-rw-rw-  1 apple  staff   7 10 12 17:08 foo
(base) apple@appledeMacBook-Pro C:C++ % ./out3
(base) apple@appledeMacBook-Pro C:C++ % ls -l foo bar
-rw-------  1 apple  staff  0 10 12 17:08 bar
-rw-rw-rw-  1 apple  staff  0 10 12 17:08 foo
```

运行已经编译好的out3，发现原有两个文件长度被截断为0，文件权限位不改变，（当然你要说原来程序设置两个文件权限位也是这样的，没有验证）那我们可以继续如下尝试：



修改 foo 777权限位，然后运行发现权限位还没有发生变化，验证完毕。

4.5 此题不需要写程序～

目录的长度从来不会是0，因为它总是包含 .和 . .两项。符号连接的长度指其路径名包含的字符数，由于路径名中至少有一个字符，所以长度也不为 0。

4.7

当创建新的core文件时，内核对其存取许可权有一个默认设置，在本例中是 -rw-r- - r- -。

这一默认值可能会可能不会被umask的值修改。

在cat core 重定向到core.copy时shell对创建的重定向的新文件也有一个默认的访问许可权，本例中为-rw-rw-rw-。这个值总是被当前的umask修改，在本例中umask为02,所以原本core.copy创建出来权限与shell新文件默认许可权限一样，只是被禁止了其他组的w权限 为-rw-rw-r-

4.11

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <dirent.h>
#include <errno.h>
#include <sys/resource.h>
#include <signal.h>

#ifndef MAX_NAME
#define MAX_NAME 1024
#endif

#define YH_FILE             1
#define YH_ISDIR            2
#define YH_STAT_FAIL        3
#define YH_OPEN_DIR_FAIL    4
#define YH_MEMORY_FAIL      5
#define YH_CLOSE_DIR_FAIL   6
```

```c
static size_t max_path_len;
static char *max_path_buf;
static size_t ndir,  // directory
              nchr,  // special character
              nsock, // socket
              nreg,  // regular
              nlnk,  // symbolic link
              nfifo, // fifo
              nblk;  // special block

static int __handle_one_file(const char *pathname, struct stat *st, int flag) {
    switch (flag) {
        case YH_FILE:
            if (S_ISREG(st->st_mode))       nreg++;
            else if (S_ISCHR(st->st_mode))  nchr++;
            else if (S_ISBLK(st->st_mode))  nblk++;
            else if (S_ISLNK(st->st_mode))  nlnk++;
            else if (S_ISSOCK(st->st_mode)) nsock++;
            else if (S_ISFIFO(st->st_mode)) nfifo++;
            else {
                printf("%s unkonw fail\n", pathname);
                return -1;
            }
            break;
        case YH_ISDIR:
            ndir++;
            break;
        case YH_STAT_FAIL:
            printf("%s get stat fail : %s\n", pathname, strerror(errno));
            return -2;
        case YH_OPEN_DIR_FAIL:
            printf("%s open dir fail : %s\n", pathname, strerror(errno));
            return -3;
        case YH_MEMORY_FAIL:
            printf("realloc fail, current level : %s\n", pathname);
            exit(1);
        case YH_CLOSE_DIR_FAIL:
            printf("close dir fail, current level : %s\n", pathname);
            return -5;
        default:
            return -6;
    }
    return 0;
}

static int recursivly_go_deep_from(const char *pathname) {

    struct stat st;
    if (lstat(pathname, &st) < 0)
```

```
        return __handle_one_file(pathname, &st, YH_STAT_FAIL);

    if (S_ISDIR(st.st_mode) == 0)
        return __handle_one_file(pathname, &st, YH_FILE);

    int ret = 0;
    if ((ret = __handle_one_file(pathname, &st, YH_ISDIR)) < 0)
        return ret;

    /* 不用再扩充数组以容纳路径，
       因为最大文件名是有限制的 MAX_NAME */

    DIR *dp = NULL;
    struct dirent *dir_ren = NULL;
    if ((dp = opendir(max_path_buf)) == NULL) {
        __handle_one_file(pathname, &st, YH_OPEN_DIR_FAIL);
    } else {
        /* 先打开目录文件，再切换工作目录，
           为什么呢？举例子：工作目录为 /usr/local/bin，打开 bin 目录会失败 */
        chdir(pathname);
        while ((dir_ren = readdir(dp)) != NULL) {
            if (strcmp(dir_ren->d_name, ".")  == 0 ||
                strcmp(dir_ren->d_name, "..") == 0)
                continue;
            strncpy(max_path_buf, dir_ren->d_name, strlen(dir_ren->d_name));
            max_path_buf[strlen(dir_ren->d_name)] = 0;
            if (recursivly_go_deep_from(max_path_buf) != 0)
                break;
        }
        chdir("..");
    }

    if (dp != NULL && closedir(dp) < 0)
        return __handle_one_file(pathname, &st, YH_CLOSE_DIR_FAIL);

    return 0;
}

int myftw(const char *pathname) {
    if (pathname == NULL) {
        printf("pathname is NULL!\n");
        return 1;
    }
    max_path_buf = malloc(MAX_NAME);
    max_path_len = MAX_NAME;
    strncpy(max_path_buf, pathname, strlen(pathname));
    max_path_buf[strlen(pathname)] = 0;
    int ret = recursivly_go_deep_from(pathname);
    free(max_path_buf);
```

```
    max_path_buf = NULL;
    max_path_len = 0;
    return ret;
}

void sig_int(int signo) {
    printf("\n【操作被打断!】\n"
           "目录 : %zd\n"
           "普通文件 : %zd\n"
           "fifo : %zd\n"
           "socket : %zd\n"
           "块特殊设备 : %zd\n"
           "字符特殊设备 : %zd\n"
           "符号链接 : %zd\n", ndir, nreg, nfifo, nsock, nblk, nchr, nlnk);
    exit(2);
}

void (*yh_signal(int signo, void (*sig_handler)(int))) (int) {
    struct sigaction new_act, old_act;
    new_act.sa_mask = 0;
    new_act.sa_handler = sig_handler;
    if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
    new_act.sa_flags |= SA_INTERRUPT;
#endif
    } else {
        new_act.sa_flags |= SA_RESTART;
    }
    if (sigaction(signo, &new_act, &old_act) < 0)
        return SIG_ERR;
    return old_act.sa_handler;
}

int main(int argc, char *argv[]) {

    if (argc != 2) {
        printf("./a.out [beg_file_path]\n");
        exit(1);
    }

    setbuf(stdout, NULL);

    if (yh_signal(SIGINT, sig_int) == SIG_ERR) {
        perror("signal error");
        exit(1);
    }

    if (myftw(argv[1]) < 0) {
        printf("myftw fail\n");
```

```
        exit(1);
    }

    printf("目录 : %zd\n"
           "普通文件 : %zd\n"
           "fifo : %zd\n"
           "socket : %zd\n"
           "块特殊设备 : %zd\n"
           "字符特殊设备 : %zd\n"
           "符号链接 : %zd\n", ndir, nreg, nfifo, nsock, nblk, nchr, nlnk);

    return 0;
}
```

对书上源代码作如下改进：

- 不会在打开目录失败时跳出当前目录层级，而是打印一条错误信息，然后忽略之。
- 根据传入的起始路径不同，运行时间可能过长，导致用户不耐烦会按 `Ctrl + C` 中断程序，因此对该 `SIGINT` 信号进行捕获，在捕获函数中打印当前统计到的信息。

  踩坑如下：

  - `strncpy(char *dst, const char *src, size_t src_len)` 不会在 `dst` 后自动为你设置一个 `\0`
  - `strncat(char *dst, const char *src, size_t src_len)` 会自动去找 `dst` 的 `\0`，并且也会在拼接完成后，在 `dst` 的最后为你设置一个 `\0`
  - 如果当前工作目录为 `/usr/bin`，那么 `chdir("bin")` 和 `opendir("bin")` 的调用都会失败。
  - `lstat` 会自动判断传入路径是【相对路径】还是【绝对路径】。