System Calls for File Operations

8

Abstract

This chapter covers system calls for file operations. It explains the role of system calls and the online manual pages of Linux. It shows how to use system calls for file operations. It lists and explains the most commonly used system calls for file operations. It explains hard link and symbolic link files. It explains the stat system call in detail. Based on the stat information, it develops a ls-like program to display directory contents and file information. Next, it explains the open-close-lseek system calls and file descriptors. Then it shows how to use read-write system calls to read-write file contents. Based on these, it shows how to use system calls to display and copy files. It shows how to develop a selective file copying program which behaves like a simplified dd utility program of Linux. The programming project is to use Linux system calls to implement a C program, which recursively copies a directory to a destination. The purpose of the project is for the reader to practice the design of hierarchical program structures and use stat(), open(), read(), write() system calls for file operations.

8.1 Systems Calls

In an operating system, a process runs in two different modes; kernel mode and user mode, denoted by Kmode and Umode for short. While in Umode, a process has very limited privileges. It can not do anything that requires special privileges. Privileged operations must be done in Kmode. System call, or syscall for short, is a mechanism which allows a process to enter Kmode to perform operations not allowed in Umode. Operations such as fork child process, exec to change execution image, even termination must all be done in Kernel. In this chapter, we shall discuss syscalls for file operations in Unix/Linux.

8.2 System Call Man Pages

In Unix and most versions of Linux, the online manual (man) pages are maintained in the /usr/man/ directory (Goldt et al. 1995; Kerrisk 2010, 2017). In Ubuntu Linux, they are in the /usr/share/man

K. C. Wang, *Systems Programming in Unix/Linux*, https://doi.org/10.1007/978-3-319-92429-8_8

directory. All the syscall man pages are listed in the man2 subdirectory. The sh command **man 2 NAME** displays the man pages of the syscall NAME. For example,

```
man 2 stat  : display man pages of stat(), fstat() and lstat() syscalls
man 2 open: display man pages of open() syscall
man 2 read: display man pages of read() syscall, etc.
```

Many syscalls require specific included header files, which are listed in the **SYNOPSIS** part of the man pages. Without proper header files, the C compiler may generate many warnings due to mismatches in syscall function name types. Some syscalls may also require specific data structures as parameters, which must be present as described in the man pages.

8.3 System Calls for File Operations

Syscalls must be issued from a program. Their usage is just like ordinary function calls. Each syscall is a library function, which assembles the syscall parameters and ultimately issues a syscall to the OS kernel.

```
int syscall(int a, int b, int c, int d);
```

where the first parameter a is the syscall number, and b, c, d are parameters to the corresponding kernel function. In Intel x86 based Linux, syscalls are implemented by the INT 0x80 assembly instruction, which causes the CPU to switch from User mode to Kernel mode. The kernel's syscall handler routes the call to a corresponding kernel function based on the syscall number. When the process finishes executing the kernel function, it returns to User mode with the desired results. A return value >=0 means SUCCESS, -1 means FAILED. In case of failure, an errno variable (in errno.h) records the error number, which can be mapped to a string describing the error reason. The following example shows how to use some of simple syscalls.

Example C8.1. mkdir, chdir, getcwd, syscalls.

```
/************ C8.1.c file ************/
#include <stdio.h>
#include <errno.h>
int main()
{
    char buf[256], *s;
    int r;
    r = mkdir("newdir", 0766); // mkdir syscall
    if (r < 0)
       printf("errno=%d : %s\n", errno, strerror(errno));
    r = chdir("newdir");       // cd into newdir
    s = getcwd(buf, 256);      // get CWD string into buf[ ]
    printf("CWD = %s\n", s);
}
```

The program issues a **mkdir()** syscall to make a new directory. The mkdir() syscall requires a pathname and a permission (0766 in octal). If newdir does not exist, the syscall would succeed with a 0 return value. If we run the program more than once, it should fail on the second or any successive runs with a return value -1 since the directory already exists. In that case, the program would print the message

```
errno=17 : File exists
```

In addition to mkdir(), the program also illustrates the usage of chdir() and getcwd() syscalls.

EXERCISE: Modify the C8.1 program to make many directories in one run, e.g.

```
mymkdir dir1 dir2 dir3, ... dirn
```

Hint: write main() as main(int argc, char *argv[ ])

Simple System Calls: The following lists some of the simple syscalls for file operations. The reader is encouraged to write C programs to use and test them.

access : check user permissions for a file.

```
int access(char *pathname, int mode);
```

chdir : change directory

```
int chdir(const char *path);
```

chmod : change permissions of a file

```
int chmod(char *path, mode_t mode);
```

chown : **c**hange owner of file

```
int chown(char *name, int uid, int gid);
```

chroot : **c**hange (logical) root directory to pathname

```
int chroot(char *pathname);
```

getcwd : get absolute pathname of CWD

```
char *getcwd(char *buf,  int size);
```

mkdir : create a directory

```
int mkdir(char *pathname, mode_t mode);
```

rmdir : remove a directory (must be empty)

```
int rmdir(char *pathname);
```

link : hard link new filename to old filename

```
int link(char *oldpath, char *newpath);
```

unlink : decrement file's link count; delete file if link count reaches 0

```
int unlink(char *pathname);
```

symlink : create a symbolic link for a file

```
int symlink(char *oldpath, char *newpath);
```

rename : change the name of a file

```
int rename(char *oldpath, char *newpath);
```

utime : change access and modification times of file

```
int utime(char *pathname,  struct utimebuf *time)
```

The following syscalls require superuser privilege.

mount : attach a file system to a mounting point directory

```
int  mount(char *specialfile, char *mountDir);
```

umount : detach a mounted file system

```
int umount(char *dir);
```

mknod : make special files

```
int mknod(char *path, int mode, int device);
```

8.4 Commonly used system Calls

In this section, we shall discuss some of the most commonly used syscalls for file operations. These include

stat : get file status information

```
int stat(char *filename, struct stat *buf)
int fstat(int filedes, struct stat *buf)
int lstat(char *filename, struct stat *buf)
```

open : open a file for READ, WRITE, APPEND

```
int open(char *file, int flags, int mode)
```

close : close an opened file descriptor

```
int close(int fd)
```

read : read from an opened file descriptor

```
int read(int fd, char buf[ ], int count)
```

write : write to an opened file descriptor

```
int write(int fd, char buf[ ], int count)
```

lseek : reposition R|W offset of a file descriptor

```
int lseek(int fd, int offset, int whence)
```

dup : duplicate file descriptor into the lowest available descriptor number

```
int dup(int oldfd);
```

dup2 : duplicate oldfd into newfd; close newfd first if it was open

```
int  dup2(int oldfd, int newfd)
```

link : hard link newfile to oldfile

```
int link(char *oldPath, char *newPath)
```

unlink : unlink a file; delete file if file's link count reaches 0

```
int unlink(char *pathname);
```

symlink : create a symbolic link

```
int symlink(char *target, char *newpath)
```

readlink: read contents of a symbolic link file

```
int readlink(char *path, char *buf, int bufsize)
```

umask : set file creation mask; file permissions will be (mask & ~umask)

```
int umask(int umask);
```

8.5 Link Files

In Unix/Linux, every file has a pathname. However, Unix/Linux allows different pathnames to represent the same file. These are called LINK files. There are two kinds of links, HARD link and SOFT or symbolic link.

8.5.1 Hard Link Files

HARD Links: The command

```
ln  oldpath  newpath
```

creates a HARD link from newpath to oldpath. The corresponding syscall is

```
link(char *oldpath, char *newpath)
```

Hard linked files share the same file representation data structure (inode) in the file system. The file's links count records the number of hard links to the same inode. Hard links can only be applied to non-directory files. Otherwise, it may create loops in the file system name space, which is not allowed. Conversely, the syscall

```
unlink(char *pathname)
```

decrements the links count of a file. The file is truly removed if the links count becomes 0. This is what the rm (file) command does. If a file contains very important information, it would be a good idea to create many hard links to the file to prevent it from being deleted accidentally.

8.5.2 Symbolic Link Files

SOFT Links: The command

```
ln -s  oldpath newpath    # ln command with the -s flag
```

creates a SOFT or **Symbolic link** from newpath to oldpath. The corresponding syscall is

```
symlink(char *oldpath, char *newpath)
```

The newpath is a regular file of the LNK type containing the oldpath string. It acts as a detour sign, directing access to the linked target file. Unlike hard links, soft links can be applied to any file, including directories. Soft links are useful in the following situations.

(1). To access to a very long and often used pathname by a shorter name, e.g.

```
x -> aVeryLongPathnameFile
```

(2). Link standard dynamic library names to actual versions of dynamic libraries, e.g.

```
libc.so.6 -> libc.2.7.so
```

When changing the actual dynamic library to a different version, the library installing program only needs to change the (soft) link to point to the newly installed library.

One drawback of soft link is that the target file may no longer exist. If so, the detour sign might direct the poor driver to fall off a cliff. In Linux, such dangers are displayed in the appropriate color of dark RED by the ls command, alerting the user that the link is broken. Also, if foo -> /a/b/c is a soft link, the open("foo", 0) syscall will open the linked file /a/b/c, not the link file itself. So the open()/read() syscalls can not read soft link files. Instead, the **readlink** syscall must be used to read the contents of soft link files.

8.6 The stat Systen Call

The syscalls, **stat/lstat/fstat**, return the information of a file. The command **man 2 stat** displays the man pages of the stat system call, which are shown below for discussions.

8.6.1 Stat File Status

8.6.1.1 STAT(2) Linux Programmer's Manual STAT(2)

NAME

```
stat, fstat, lstat - get file status
```

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

DESCRIPTION
These functions return information about the specified file. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file.

stat stats the file pointed to by filename and fills in buf with stat information.
lstat is identical to stat, except in the case of a symbolic link, where the link itself is stated, not the file that it refers to. So the difference between stat and lstat is: stat follows link but lstat does not.
fstat is identical to stat, only the open file pointed to by filedes (as returned by open(2)) is stated in place of filename.

8.6.2 The stat Structure

All the stat syscalls return information in a stat structure, which contains the following fields:

```
struct stat{
   dev_t     st_dev;      /* device */
   ino_t     st_ino;      /* inode */
   mode_t    st_mode;     /* protection */
   nlink_t   st_nlink;    /* number of hard links */
   uid_t     st_uid;      /* user ID of owner */
   gid_t     st_gid;      /* group ID of owner */
   dev_t     st_rdev;     /* device type (if inode device) */
   off_t     st_size;     /* total size, in bytes */
   u32       st_blksize;  /* blocksize for filesystem I/O */
   u32       st_blocks;   /* number of blocks allocated */
   time_t    st_atime;    /* time of last access */
   time_t    st_mtime;    /* time of last modification */
   time_t    st_ctime;    /* time of last change */
};
```

The st_size field is the size of the file in bytes. The size of a symlink is the length of the pathname it contains, without a trailing NULL.

The value st_blocks gives the size of the file in 512-byte blocks. (This may be smaller than st_size/512, e.g. when the file has holes.) The value st_blksize gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Not all of the Linux file systems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the st_atime field. (See ‘noatime’ in mount(8).)

The field st_atime is changed by file accesses, e.g. by exec(2), mknod(2), pipe(2), utime(2) and read (2) (of more than zero bytes). Other routines, like mmap(2), may or may not update st_atime.

The field st_mtime is changed by file modifications, e.g. by mknod(2), truncate(2), utime(2) and write(2) (of more than zero bytes). Moreover, st_mtime of a directory is changed by the creation or deletion of files in that directory. The st_mtime field is not changed for changes in owner, group, hard link count, or mode.

The field st_ctime is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The following POSIX macros are defined to check the file type:

```
S_ISREG(m)  is it a regular file?
S_ISDIR(m)  directory?
S_ISCHR(m)  character device?
S_ISBLK(m)  block device?
S_ISFIFO(m) fifo?
S_ISLNK(m)  symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m) socket? (Not in POSIX.1-1996.)
```

The following flags are defined for the st_mode field:

```
S_IFMT     0170000   bitmask for the file type bitfields
S_IFSOCK   0140000   socket
S_IFLNK    0120000   symbolic link
S_IFREG    0100000   regular file
S_IFBLK    0060000   block device
S_IFDIR    0040000   directory
S_IFCHR    0020000   character device
S_IFIFO    0010000   fifo

S_ISUID    0004000   set UID bit
S_ISGID    0002000   set GID bit (see below)
S_ISVTX    0001000   sticky bit (see below)

S_IRWXU    00700     mask for file owner permissions
S_IRUSR    00400     owner has read permission
S_IWUSR    00200     owner has write permission
S_IXUSR    00100     owner has execute permission
S_IRWXG    00070     mask for group permissions
S_IRGRP    00040     group has read permission
S_IWGRP    00020     group has write permission
S_IXGRP    00010     group has execute permission
S_IRWXO    00007     mask for permissions for others (not in group)
S_IROTH    00004     others have read permission
S_IWOTH    00002     others have write permission
S_IXOTH    00001     others have execute permission
```

RETURN VALUE: On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

SEE ALSO chmod(2), chown(2), readlink(2), utime(2)

8.6.3 Stat and File Inode

Stat and file inode: First, we clarify how stat works. Every file has a unique **inode** data structure, which contains all the information of the file. The following shows the inode structure of EXT2 file systems in Linux.

```
struct ext2_inode{
   u16  i_mode;
   u16  i_uid;
   u32  i_size;
   u32  i_atime;
   u32  i_ctime;
   u32  i_mtime;
   u32  i_dtime;
   u16  i_gid;
   u16  i_links_count;
```

```
    u32  i_blocks;
    u32  i_flags;
    u32  i_reserved1;
    u32  i_block[15];
    u32  pad[7];
}; // inode=128 bytes in ext2/3 FS; 256 bytes in ext4
```

Each inode has a unique inode number (ino) on a storage device. Each device is identified by a pair of (major, minor) device numbers, e.g. 0x0302 means /dev/hda2, 0x0803 means /dev/sda3, etc. The stat syscall simply finds the file's inode and copies information from inode into the stat structure, except st_dev and st_ino, which are the file's device and inode numbers, respectively. In Unix/Linux, all time fields are the number of seconds elapsed since 0 hour, 0 minute, 0 second of January 1, 1970. They can be converted to calendar form by the library function ctime(&time).

8.6.4 File Type and Permissions

In the stat structure, most fields are self-explanatory. Only the st_mode field needs some explanation:

```
    mode_t   st_mode;          /* copied from i_mode of INODE */
```

The TYPE of st_mode is a u16 (16 bits).The 16 bits have the following meaning:

```
|Type|   |permissions|
----------------------
|tttt|fff|uuu|ggg|ooo|
----------------------
```

The leading 4 bits are file types, which can be interpreted as (in octal)

```
S_IFMT     0170000   bitmask for the file type bitfields
S_IFSOCK   0140000   socket
S_IFLNK    0120000   symbolic link
S_IFREG    0100000   regular file
S_IFBLK    0060000   block device
S_IFDIR    0040000   directory
S_IFCHR    0020000   character device
S_IFIFO    0010000   fifo
```

As of now, the man pages of all Unix-like systems still use octal numbers, which has its roots dated back to the old PDP-11 era in the 1970's. For convenience, we shall redefine them in HEX, which are much more readable, e.g.

```
S_IFDIR    0x4000    directory
S_IFREG    0x8000    regular file
S_IFLNK    0xA000    symbolic link
```

The next 3 bits of st_mode are flags, which indicate special usage of the file

```
S_ISUID    0004000   set UID bit
S_ISGID    0002000   set GID bit
S_ISVTX    0001000   sticky bit
```

We shall show the meaning and usage of setuid programs later. The remaining 9 bits are permission bits for file protection. They are divided into 3 categories by the (effective) uid and gid of the process:

```
owner  group  other
 rwx    rwx    rwx
```

By interpreting these bits, we may display the st_mode field as

```
-rwxr-xr-x        (REG file with r,x but w by owner only)
drwxr-xr-x        (DIR with r,x, but w by owner only)
lrw-r--r--        (LNK file with permissions)
```

where the first letter **(-|d|l)** shows the file type and the next 9 chars are based on the permission bits. Each char is printed as r|w|x if the bit is 1 or – if the bit is 0. For directory files, the x bit means whether access (cd into) to the directory is allowed or not.

8.6.5 Opendir-Readdir Functions

A directory is also a file. We should be able to open a directory for READ, then read and display its contents just like any other ordinary file. However, depending on the file system, the contents of a directory file may vary. As a result, the user may not be able to read and interpret the contents of directory files properly. For this reason, POSIX specifies the following interface functions to directory files.

```
#include <dirent.h>
DIR *open(dirPath); // open a directory named dirPath for READ
struct dirent *readdir(DIR *dp); // return a dirent pointer
```

In Linux, the dirent structure is

```
struct dirent{
    u32 d_ino; // inode number
    u16 d_reclen;
    char d_name[ ]
}
```

In the dirent structure, only the d_name field is mandated by POSIX. The other fields are system dependent. opendir() returns a DIR pointer dirp. Each readdir(dirp) call return a dirent pointer to an dirent structure of the next entry in the directory. It returns a NULL pointer when there are no more entries in the directory. We illustrate their usage by an example. The following code segment prints all the file names in a directory.

```
#include <dirent.h>
struct dirent *ep;
DIR *dp = opendir("dirname");
while (ep = readdir(dp)){
    printf("name=%s ", ep->d_name);
}
```

The reader may consult the **man 3** pages of **opendir and readddir** for more details.

8.6.6 Readlink Function

Linux's open() syscall follow symlinks. It is therefore not possible to open a symlink file and read its contents. In order to read the contents of symlink files, we must use the readlink syscal, which is

```
int readlink(char *pathname, char buf[ ], int bufsize);
```

It copies the contents of a symlink file into buf[ ] of bufsize, and returns the actual number of bytes copied.

8.6.7 The ls Program

The following shows a simple ls program which behaves like the **ls –l** command of Linux. The purpose here is not to re-invent the wheel by writing yet another ls program. Rather, it is intended to show how to use the various syscalls to display information of files under a directory. By studying the example program code, the reader should also be able to figure out how Linux's ls command is implemented.

```
/************* myls.c file **********/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>
#include <sys/types.h>
#include <dirent.h>

struct stat mystat, *sp;
char *t1 = "xwrxwrxwr-------";
char *t2 = "----------------";

int ls_file(char *fname)
{
  struct stat fstat, *sp;
  int r, i;
  char ftime[64];
  sp = &fstat;
  if ( (r = lstat(fname, &fstat)) < 0){
     printf("can't stat %s\n", fname);
     exit(1);
```

```
  }
  if ((sp->st_mode & 0xF000) == 0x8000) // if (S_ISREG())
     printf("%c",'-');
  if ((sp->st_mode & 0xF000) == 0x4000) // if (S_ISDIR())
     printf("%c",'d');
  if ((sp->st_mode & 0xF000) == 0xA000) // if (S_ISLNK())
     printf("%c",'l');
  for (i=8; i >= 0; i--){
    if (sp->st_mode & (1 << i)) // print r|w|x
      printf("%c", t1[i]);
    else
      printf("%c", t2[i]);      // or print -
  }
  printf("%4d ",sp->st_nlink);  // link count
  printf("%4d ",sp->st_gid);    // gid
  printf("%4d ",sp->st_uid);    // uid
  printf("%8d ",sp->st_size);   // file size
  // print time
  strcpy(ftime, ctime(&sp->st_ctime)); // print time in calendar form
  ftime[strlen(ftime)-1] = 0;   // kill \n at end
  printf("%s  ",ftime);
  // print name
  printf("%s", basename(fname)); // print file basename
  // print -> linkname if symbolic file
  if ((sp->st_mode & 0xF000)== 0xA000){
     // use readlink() to read linkname
     printf(" -> %s", linkname); // print linked name
  }
  printf("\n");
}

int ls_dir(char *dname)
{
  // use opendir(), readdir(); then call ls_file(name)
}

int main(int argc, char *argv[])
{
  struct stat mystat, *sp = &mystat;
  int r;
  char *filename, path[1024], cwd[256];
  filename = "./";        // default to CWD
  if (argc > 1)
     filename = argv[1];  // if specified a filename
  if (r = lstat(filename, sp) < 0){
     printf("no such file %s\n", filename);
     exit(1);
  }
  strcpy(path, filename);
  if (path[0] != '/'){ // filename is relative : get CWD path
```

```
    getcwd(cwd, 256);
    strcpy(path, cwd); strcat(path, "/"); strcat(path,filename);
  }
  if (S_ISDIR(sp->st_mode))
    ls_dir(path);
  else
    ls_file(path);
}
```

Exercise 1: fill in the missing code in the above example program, i.e. the ls_dir() and readlink() functions, to make it work for any directory.

8.7 open-close-lseek System Calls

open : open a file for READ, WRITE, APPEND

```
int open(char *file, int flags, int mode);
```

close : close an opened file descriptor

```
int close(int fd);
```

read : read from an opened file descriptor

```
int read(int fd, char buf[ ], int count);
```

write : write to an opened file descriptor

```
int write(int fd, char buf[ ], int count);
```

lseek : reposition the byte offset of a file descriptor to offset from whence

```
int lseek(int fd, int offset, int whence);
```

umask: set file creation mask; file permissions will be (mask & ~umask)

8.7.1 Open File and File Descriptor

```
#include <sys/type.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(char *pathname, int flags, mode_t mode)
```

Open() opens a file for READ, WRITE or APPEND. It returns a **file descriptor**, which is the lowest available file descriptor number of the process, for use in subsequent read(), write(), lseek() and close() syscalls. The flags field must include one of the following access mode O_RDONLY, O_WRONLY, or O_RDWR. In addition, flags may be bit-wise ORed with other flags O_CREAT, O_APPEND, O_TRUNC, O_CLOEXEC, etc. All these symbolic constants are defined in the **fcntl.h** header file. The optional mode field specifies the permissions of the file in Octal. The permissions of a newly created file or directory are the specified permissions bit-wise ANDed with ~umask, where **umask** is set in the login profile as (octal) 022, which amounts to deleting the w (write) permission bits for non-owners. The umask can be changed by the **umask()** syscall. **Creat()** is equivalent to open() with flags equal to O_CREAT|O_WRONLY|O_TRUNC, which creates a file if it does not exist, opens it for write and truncates the file size to zero.

8.7.2 Close File Descriptor

```
#include <unistd.h>
int close(inf fd);
```

Close() closes the specified file descriptor fd, which can be reused to open another file.

8.7.3 lseek File Descriptor

```
#include <sys/type.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

In Linux, **off_t** is defined as u64. When a file is opened for read or write, its RW-pointer is initialized to 0, so that read|write starts from the beginning of the file. After each read |write of n bytes, the RW-pointer is advanced by n bytes for the next read|write. lssek() repositions the RW-pointer to the specified offset, allowing the next read|write from the specified byte position. The whence parameter specifies SEEK_SET (from file beginning), SEEK_CUR (current RW-pointer plus offset) SEEK_END (file size plus offset).

8.8 Read() System Call

```
#include <unistd.h>
int read(int fd, void *buf, int nbytes);
```

read() reads nbytes from an opened file descriptor into buf[ ] in user space. The return value is the actual number of bytes read or -1 if read() failed, e.g. due to invalid fd. Note that the buf[ ] area must have enough space to receive nbytes, and the return value may be less than nbytes, e.g. if the file size is less than nbytes or when the file has no more data to read. Note also that the return value is an integer, not any End-Of-File (EOF) symbol since there is no EOF symbol in a file. EOF is a special integer value (-1) returned by I/O library functions when a FILE stream has no more data.

8.9 Write() System Call

```
#include <unistd.h>
int write(int fd, void *buf, int nbytes);
```

write() writes nbytes from buf[ ] in user space to the file descriptor, which must be opened for write, read-write or append mode. The return value is the actual number of bytes written, which usually equal to nbytes, or -1 if write() failed, e.g. due to invalid fd or fd is opened for read-only, etc.

Example: The following code segment uses open(), read(), lseek(), write() and close() syscalls. It copies the first 1KB bytes of a file to byte 2048.

```
char buf[1024];
int fd=open("file", O_RDWR);  // open file for READ-WRITE
read(fd, buf[ ], 1024);       // read first 1KB into buf[ ]
lseek(fd, 2048, SEEK_SET);    // lseek to byte 2048
write(fd, buf, 1024);         // write 1024 bytes
close(fd);                    // close fd
```

8.10 File Operation Example Programs

Syscalls are suitable for file I/O operations on large blocks of data, i.e. operations that do not need lines, chars or structured records, etc. The following sections show some example programs that use syscalls for file operations.

8.10.1 Display File Contents

Example 8.2: Display File Contents. This program behaves somewhat like the Linux cat command, which displays the contents of a file to stdout. If no filename is specified, it gets inputs from the default stdin.

```
/********* C8.2 file ********/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define BLKSIZE 4096
int main(int argc, char *argv[ ])
{
  int fd, i, m, n;
  char buf[BLKSIZE], dummy;
  fd = 0; // default to stdin
  if (argc > 1){
     fd = open(argv[1], O_RDONLY);
```

```
    if (fd < 0) exit(1);
  }
  while (n = read(fd, buf, BLKSIZE)){
    m = write(1, buf, n);
  }
}
```

When running the program with no file name, it collects inputs from fd=0, which is the standard input stream stdin. To terminate the program, enter Control-D (0x04), which is the default EOF on stdin. When running the program with a filename, it first opens the file for read. Then it uses a while loop to read and display the file contents until read() returns 0, indicating the file has no more data. In each iteration, it reads up to 4KB chars into a buf[ ] and writes the n chars to the file descriptor 1. In Unix/Linux files, lines are terminated by the LF=\n char. If a file descriptor refers to a terminal special file, the pseudo terminal emulation program automatically adds a \r for each \n char in order to produce the right visual effect. If the file descriptor refers to an ordinary file, no extra \r chars will be added to the outputs.

8.10.2 Copy Files

Example 8.3: Copy Files. This example program behaves like the Linux **cp src dest** command, which copies a src file to a dest file.

```
/******** c8.3.c file *******/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define BLKSIZE 4096
int main(int argc, char *argv[ ])
{
  int fd, gd, n, total=0;
  char buf[BLKSIZE];
  if (argc < 3) exit(1);  // usage a.out src dest
  if ((fd = (open(argv[1], O_RDONLY)) < 0)
     exit(2);
  if ((gd = open(argv[2],O_WRONLY|O_CREAT)) < 0)
     exit(3);
  while (n = read(fd, buf, BLKSIZE)){
     write(gd, buf, n);
     total += n;
  }
  printf("total bytes copied=%d\n", total);
  close(fd); close(gd);
}
```

Exercise 1. The Example program c8.3 has a serious flaw in that we should never copy a file to itself. Besides wasting time, the reader may also find out the reason by looking at the program code. If the src and dest files are the same file, rather than copying, it would just truncate the file size to 0. Modify the program to ensure that src and dest are not the same file. Note that different filenames may refer to the same file due to hard links.

HINT: stat both pathnames and compare their (st_dev, st_ino).

8.10.3 Selective File Copy

Example 8.4: Selective File Copy: This example program is a refinement of the simple file copying program in Example 8.3. It behaves like the Linux **dd** command, which copies selected parts of files. The reader may consult the man pages of dd to find out its full capabilities. Since our objective here is to show how to use syscalls for file operations, the example program is a simplified version of dd. The program runs as follows.

```
a.out if=in of=out bs=size count=k [skip=m] [seek=n] [conv=notrunc]
```

where [ ] denote optional entries. If these entries are specified, skip=m means skip m blocks of the input file, seek=n means step forward n blocks of the output file before writing and conv=notrunc means do not truncate the output file if it already exits. The default values of skip and seek are 0, i.e. no skip or seek. For simplicity, we assume the command line parameters contain no white spaces, which simplifies command-line parsing. The only new feature of this program is to parse the command line parameters to set file names, counting variables and flags. For example, if conv=notrunc is specified, the target file must be opened without truncating the file. Second, if skip and seek are specified, opened files must use lseek() to set the RW offsets accordingly.

```
/************ c8.4.c file **********/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

char in[128], out[128], buf[4096];
int bsize, count, skip, seek, trnc;
int records, bytes;
// parse command line parameters and set variables
int parse(char *s)
{
  char cmd[128], parm[128];
  char *p = index(s, '=');
  s[p-s] = 0; // tokenize cmd=parm by '='
  strcpy(cmd, s);
  strcpy(parm, p+1);
  if (!strcmp(cmd, "if"))
     strcpy(in, parm);
```

```
  if (!strcmp(cmd, "of"))
     strcpy(out, parm);
  if (!strcmp(cmd, "bs")
     bsize = atoi(parm);
  if (!strcmp(cmd, "count"))
     count = atoi(parm);
  if (!strcmp(cmd, "skip"))
     skip = atoi(parm);
  if (!strcmp(cmd, "seek"))
     seek = atoi(parm);
  if (!strcmp(cmd, "conv")){
     if (!strcmp(parm, "notrunc"))
        trnc = 0;
  }
}

int main(int argc, char *argv[])
{
  int fd, gd, n, i;
  if (argc < 3){
     printf("Usage: a.out if of ....\n"); exit(1);
  }
  in[0] = out[0] = 0;              // null file names
  bsize = count = skip = seek = 0; // all 0 to start
  trnc = 1;                        // default = trunc
  for (i=1; i<argc; i++){
      parse(argv[i]);
  }
  // error checkings
  if (in[0]==0) || out[0]==0{
     printf("need in/out files\n"); exit(2);
  }
  if (bsize==0 || count==0){
     printf("need bsize and count\n"); exit(3);
  }
  // ADD: exit if in and out are the same file
  if ((fd = open(in, O_RDONLY)) < 0){
     printf("open %s error\n", in); exit(4);
  }
  if (skip) lseek(fd, skip*bsize, SEEK_SET);
  if (trnc) // truncate out file
     gd = open(out, O_WRONLY|O_CREAT|O_TRUNC);
  else
     gd = open(out, O_WRONLY|O_CREAT); // no truncate
  if (gd < 0){
     printf("open %s error\n", out); exit(5);
  }
  if (seek) lseek(gd, seek*bsize, SEEK_SET);

  records = bytes = 0;
```

```
  while (n = read(fd, buf, bsize)){
    write(gd, buf, n);
    records++; bytes += n;
    count--;              // dec count by 1
    if (count==0) break;
  }
  printf("records=%d bytes=%d copied\n", records, bytes);
}
```

8.11 Programming Project: Recursive Copy Files using System Calls

The programming project is to use Linux system calls to write a C program, mycp, which recursively copies src to dest. It should work exactly the same as the Linux command

```
cp –r src dest
```

which recursively copies src to dest.

8.11.1 Hints and Helps

1. Analyze the various conditions whether copy is allowed. The following are some example cases, but they are incomplete. The reader should complete the case analyses before attempting to develop any code.
   (1). src must exist but dest may or may not exist.
   (2). If src is a file, dest may not exist, is a file or a directory.
   (3). If src is a directory, dest must be an existing or non-existing directory.
   (4). If src is a DIR and dest does not exist, create dest DIR and copy src to dest.
   (5). If src is a DIR and dest is an existing DIR: do not copy if dest is a descendant of src. Otherwise, copy src into dest/, i.e. as dest/(basename(src))
   (6). Never copy a file or directory to itself.
   Whenever in doubt, run the Linux **cp –r src dest** command and compare the results.

2. Organize the project program into 3 layers:
   Base case: cpf2f(file 1, filef2): copy file 1 to file 2; handle REG and LNK files
   Middle case: cpf2d(file, dir): copy file into an existing dir
   Top case: cpd2d(dir1,dir2): copy (recursively) dir1 to dir2

8.11.2 Sample Solution

Sample solution of the programming project can be downloaded from the book's website. Source code for instructors is also available upon request from the author.

8.12 Summary

This chapter covers system calls for file operations. It explains the role of system calls and the online manual pages of Linux. It shows how to use system calls for file operations. It lists and explains the most commonly used system calls for file operations. It explains hard link and symbolic link files. It explains the stat system call in detail. Based on the stat information, it develops a ls-like program to display directory contents and file information. Next, it explains the open-close-lseek system calls and file descriptors. Then it shows how to use read-write system calls to read-write file contents. Based on these, it shows how to use system calls to display and copy files. It shows how to develop a selective file copying program which behaves like a simplified dd utility program of Linux. The programming project is to use Linux system calls to implement a C program, which recursively copies a directory to a destination. The purpose of the project is for the reader to practice on the design of hierarchical program structures and use stat(), open(), read(), write() system calls for file operations.

References

Goldt, S. Van Der Meer, S., Burkett, S., Welsh, M., The Linux Programmer's Guide-The Linux Documentation Project, 1995

Kerrisk, M., The Linux Programming Interface, No Starch Press, Inc., 2010

Kerrisk, M. The Linux man-pages project, https://www.kernel.org/doc/man-pages/, 2017