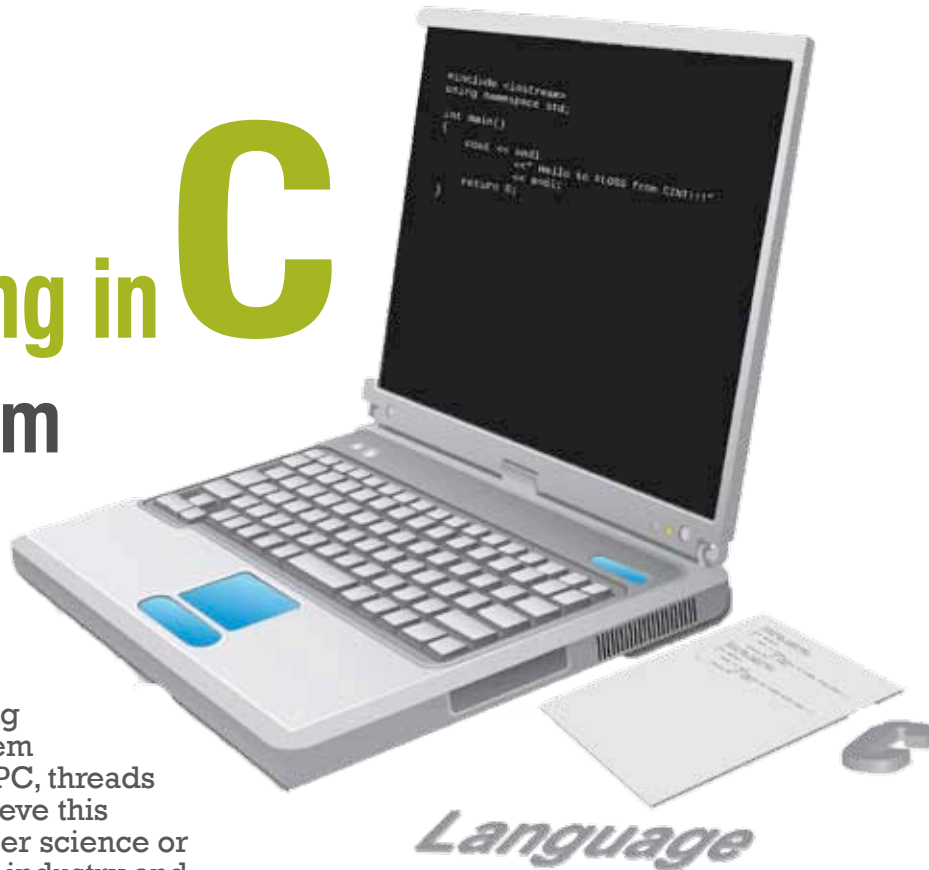


System Programming in C

The File-system Interface



In this series of articles, let us explore some important systems programming concepts in Linux, like file-system interaction, process concepts, IPC, threads and socket programming. I believe this series will be useful for computer science or IT students, freshers joining the industry and for professionals who have done this on other systems. I assume that readers are familiar with the C language and the GCC compiler on Linux, and have some basic general OS knowledge.

Part—1

In this article, we will look at some important file-system interaction-related concepts in Linux. Two types of interactions are very common—accessing or manipulating file contents through an editor, for instance; and accessing or manipulating file attributes, like listing files with the `ls` command. So here's an overview of inodes, hard and soft links, and some system call interfaces for files and directories, related to the two aspects mentioned. I will also relate these concepts to C standard library functions, so that you get an idea of how functions like `fopen`, `fclose`, `fgets`, etc, are implemented by library vendors for a particular OS.

Files, directories, inodes and links

Every file or directory in Linux has a unique number called the *inode number*. To view it, use `ls -li` (`l` is long listing format; `i` is for the inode number) at the command prompt.

The first column lists the inode number of the file/directory.

The inode number is an index to a very large array of *inode structures*. Individual elements store the file's attributes (like the type of file, access permissions, the owner, number of hard links, the file size, time stamps, etc) and the addresses of the file's data blocks.

Directories have entries called directory entries, which typically have a file or a directory name, and the inode number. For example, if you create a file called *file1* in your current working directory, a free inode (say *i1*) will be allocated for it. Thus, a new directory entry of the form (*file1*, *i1*) will be created in the current directory.

Hard links (we'll come to soft links soon) are references to a particular inode. Now, if we have another directory entry (say *file2*, *i1*) that points to the *same* inode *i1*, then *file2* is referred to as a *hard link* to *file1*; both entries point to the same inode structure, and refer to the

same data. Links are usually created with the `ln` command; to create the *file2* hard link to *file1*, the command would be `ln file1 file2`. After this, let us list the directory contents, with inodes, as shown below:

```
bash-3.00$ ls -li
total 2
2889771 -rw-r--r-- 2 user1 grp1 5 Apr 27 2012 file1
2889771 -rw-r--r-- 2 user1 grp1 5 Apr 27 2012 file2
```

You can see that both files have the same inode number 2889771. The third column shows the number of hard links—in the above case, it's two for both, since the one inode has two directory entries referencing it.

To notice something different, create a sub-directory, *dir1*. Do a listing again; you'll see that the link count of the directory is 2, and not 1, as for a regular file:

```
2889767 drwxr-xr-x 2 user1 grp1 512 Apr 26 2012 dir1
```

Why does even an 'empty' new directory have a link count of 2? Because of the following reasons:

- Its parent directory contains a directory entry for the name-to-inode mapping—that's one reference.
- In the directory itself, there is always created a `.` (dot) entry signifying the current directory, referencing the same inode for the directory—that's the second.

Now, there's also a default `..` (dot-dot, parent directory) entry created, and so obviously the parent directory's link count also increases by 1.

Now let's look at *soft links*, also called *symbolic links*. These are actually files containing the path to a target (linked file) and are created by adding the `-s` parameter to the `ln` command. For example:

```
bash-3.00$ ln -s file1 file3
bash-3.00$ ls -ltri
total 3
2889771 -rw-r--r-- 2 user1 grp1 5 Apr 27 2012 file2
2889771 -rw-r--r-- 2 user1 grp1 5 Apr 27 2012 file1
2889769 lrwxrwxrwx 1 user1 grp1 5 Apr 27 2012 file3 -> file1
```

Here, *file3* is a soft link to *file1*; it's a separate file, with its own inode number. Note that if we try to open *file3* with an editor, *file1*'s content is what we'll actually see—most programs follow symbolic links to get at the target files. To view the actual contents of the link file, not the target, use a command like *readlink*, as follows:

```
bash-3.00$ readlink file3
file1
```

Symbolic links are more flexible than hard links: you

cannot create a hard link on one file-system to a file on another file-system (say, a different HDD partition mounted by the OS). You can, however, create symbolic links across file-systems.

Accessing file attributes with C

Here is an example of accessing file/directory attributes, which is called *my_dirtraverse.c*:

```
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<dirent.h>
#include<sys/stat.h>
#include<sys/types.h>
void traversedir (char *);
int main () {
    char cdir[1024];
    /*get the path of current working directory using getcwd*/
    getcwd(cdir,1024);
    traversedir(cdir);
    return 0;
}
void traversedir( char *dir) {
    struct dirent *dp; /*dp is a pointer to a
    structure representing directory Entry*/
    struct stat sbuf; /*This is a structure, which contains the
    attributes
    of a file or directory*/
    char temppath [1024];
    DIR *dfd; /* DIR - Assume it is a structure which contains the file
    descriptor for a open directory and the directory entry*/
    printf("Directory %s\n",dir);
    /*Trying to open a directory using opendir function, which
    returns
    a pointer to DIR structure*/
    if ((dfd=opendir (dir))==NULL) {
        printf ("Cannot open %s",dir);
        return;
    }
    /*In a while Loop read the Directory Entries using
    readdir*/
    while ((dp=readdir (dfd))!=NULL) {
        /*every Directory is having the dot & dot dot
        entries*/
        if (strcmp (dp->d_name,".")==0) {
            continue;
        }
        else if (strcmp(dp->d_name,"..")==0) {
            continue;
        }
        else {
            /*checking whether it is a directory*/
```

```

/*before Stat make a absolute path name*/
strcpy (temppath,dir);
strcat (temppath, "/");
strcat (temppath, dp->d_name);
/*stat - populates the structure with the
attributes of the file/directory passed as the
first argument*/
stat (temppath,&sbuf);
/*Checking if it is a directory, you can
check many other things too */
if (S_ISDIR(sbuf.st_mode)) {
    /*Again call this function to
    walk down this directory*/
    traversedir (temppath);
}
}
}
closedir (dfd);
}

```

Now, if you compile this code and run it, it outputs the absolute pathnames of all directories starting from the current directory. The comments in the code will help you understand the basic algorithm.

You can consider modifying the code to add error-handling and more features. Please also read the manual pages for the functions used. You can experiment with these in your own code.

File access and manipulation: System calls vs library functions

Whenever you access a file, the OS (to be more specific, a system call) internally converts the filename to the inode number to get its attributes or data. The data structures used internally are called the user file descriptor table, the file table and the inode table. You need to remember that whenever your program opens a file, a file descriptor (a non-negative number) is returned to your program, which you can use to read/write the file's data. By default, three descriptors—0 (standard input), 1 (standard output) and 2 (standard error) are automatically allocated to every running program. When you open a file using library functions, instead of a file descriptor, you use a file pointer. A simple program will help you understand these concepts better, so here's a program to copy a file using library functions – it is called *mycpy_lib.c*:

```

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv []) {
    FILE *sfp,*dfp;
    int ch;

```

```

if (argc < 3) {
    printf ("Usage:mycpy sourcefile destinationfile\n");
    exit (1);
}
if ((dfp=fopen (argv[2],"w")) ==NULL) {
    printf ("Cannot open %s", argv[2]);
    exit (1);
}
if ((sfp = fopen(argv[1],"r"))==NULL) {
    printf ("Cannot open %s", argv[1]);
    exit (1);
}
while ((ch = fgetc ( sfp )) != EOF) {
    printf ("%c", (char)ch);
    fputc (ch, dfp);
}
fclose (sfp);
fclose (dfp);
return 0;
}

```

Compile (*gcc mycpy_lib.c -o mycpy*) and run it with two command-line arguments—an existing file, and a name for the copy. Then note the functions used, like *fopen*, *fgetc*, *fputc* and *fclose*. *fopen* returns a file pointer (to a structure named *FILE*), which is used as an argument to all the other functions. You can assume that for every open file in your process, there is an associated *FILE* structure. The function *fgetc* returns a character read from the file, while *fputc* writes a character to the file.

You can see that the *while* loop runs till the end of the file has been reached. How does the library function *fgetc* know that the file has ended? The code below, *mycpy_sys.c*, which uses system calls instead of library functions, will help answer these questions:

```

#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<errno.h>
#include<unistd.h>
#define BLOCKSIZE 100

int main (int argc, char *argv []) {
    int sfd, dfd, bytesread, ret = 0;
    char buf [BLOCKSIZE];
    if (argc < 3) {
        printf ("Usage:mycpy sourcefile
destinationfile\n");
        exit (1);

```

```

    }
    /*Opening a file(that is getting the file descriptor), whose
    name was specified in command line argument 1 in read only mode
    (specified by the symbolic constant O_RDONLY)*/
    sfd = open(argv[1],O_RDONLY);
    if (0 > sfd) {
        printf ("Cannot open File for reading \n");
        exit (1);
    }
    /*Open the destination file in Write Only Mode(O_WRONLY), O_CREAT-
    symbolic constant denotes that it will be created, O_TRUNC-data
    will be truncated, if already existing, S_IRWXU-specifies the
    permissions (read/write/execute for user)*/
    dfd = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,S_IRWXU);
    if (0 > dfd) {
        printf ("Cant open file for writing\n");
        exit (1);
    }
    while ((bytesread = read(sfd, buf, BLOCKSIZE))>0) {
        ret = write (dfd, buf, bytesread);
        if (-1 == ret) {
            printf ("File Write Error\n");
            exit (1);
        }
    }
    /*Closing the file descriptors explicitly*/
    close (sfd);
    close (dfd);
    return 0;
}

```

Now, note that here, the *open*, *read*, *write* and *close* functions have been used instead of *fopen*, *fgetc*, *fputc* and *fclose*. These actually translate to direct system calls. Instead of a file pointer from *fopen*, *open* returns a file descriptor number. The arguments of the *read* and *write* functions are of the general form: file descriptor, an array from where to write (in case of write) or into which to read data (in case of read), and third, the requested number of bytes to be read or to be written. The return values of *read* and *write* are the number of bytes successfully read or written. Here we do not have functions like *fgetc/fputc*, *fgets/fputs*, *fscanf/fprintf* or *fread/fwrite*, which can deal with characters, lines and formatted data and structures, respectively; *read* and *write* understand a file as a sequence of bytes only.

All standard library functions normally call the system calls internally; *fopen* should internally call *open*, or *fgetc* call *read*, etc. You can use *strace* (which traces system calls) to validate this. Compile *mycpy_lib.c* and run with source and destination file arguments, under *strace*, as follows:

```
bash-3.00$ strace ./a.out file1 file2
```

```
open ("file2", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
```


```
open ("file1", O_RDONLY) = 4
```

The numbers on the right are the return values (in this case, file descriptors) of the *open* function, which was called internally by *fopen*.

Now, what about end-of-file handling? When there is no more data to be read, the *read* function returns a value of 0; this is received by *fgetc*, which then returns *EOF* (typically -1, but we should not assume a value, for portability) to the caller. Thus, the system need not store any special character to mark the end-of-file—just the number of bytes in the file.

So it is now clear that a file pointer is a higher-level abstraction than the file descriptor, and the end-of-file is not a special character stored in a file.

So what are the advantages and disadvantages of using system calls rather than library functions? The latter are obviously more portable; *fopen*, *fgetc*, *fputc*, *fclose*, etc, are supposed to be available on a wide variety of platforms, but system calls like *open*, *read*, *write* and *close* are available for a particular OS, say Linux. However, using system calls can make the program run faster on a specific OS.

In this article, I have briefly covered the basic C language file-system interface in Linux. Please read the manual pages to learn more functions like *link*, *symlink*, *unlink*, *lseek*, *dup*, *access*, *stat*, *mkdir*, *rmdir*. Do also try implementing your own *ls* or *cat* command; that will help enhance your programming skills too.  **END**

References

- [1] Bach, Maurice J: 'The Design of the Unix Operating System', PHI
- [2] Stevens, W Richard: 'Unix Network Programming: Interprocess Communication', Vol-2, 2nd Edition, Pearson Education
- [3] Stevens, W Richard and Rago, Stephen A: 'Advanced Programming in the Unix Environment', 2nd Edition, Pearson Education
- [4] Linux Manual Pages
- [5] You can download the source code of this article from http://www.linuxforu.com/article_source_code/july12/system_programming.zip

Acknowledgement

I would like to thank Tanmoy Bandyopadhyay and Vikas Nagpal for their help in reviewing this article.

By: Swati Mukhopadhyay

The author is having more than 12 years of experience in academics and corporate training. Her interest areas are Digital logic, Computer architecture, Computer Networking, Data Structures, Linux and programming languages like C, C++. For any queries or feedback, she can be reached at swati.mukerjee@gmail.com or <http://swati.mukhopadhyay.wordpress.com/>.