Working With Files

Introduction

Files

1. In UNIX system or UNIX-like system, all input and output are done by reading or writing files, because all peripheral devices, even keyboard and screen are files in the file system.

2. This means that a single homogeneous interface handles all communication between a program and peripheral devices.In general, programs can use disk files, serial ports, printers and other devices in exactly the same way as they would.

3. UNIX provides a simple and consistent interface to operating system services and to devices.

Directories

A directory is a file that contains a list of file names and some indication of where they are located.

The location is an index into another table called the "inode list".

The inode for a file is where all the information of a file is kept on the disk, e.g. file's creation/modification time, permission, length, etc. except the file's name.

A directory entry generally consists of only two items, the file name and an inode number.

The structure of directory is defined in $< sys/dir.h >$ as followed,

```
/* conditional processor */
/* check if DIRSIZ is defined or not */
#ifndef DIRSIZ
/* if not, then define it */
```

```
#define DIRSIZ 14
/* end of the conditional processor*/
#endif

/* the structure of directory */
struct directory
{
/* type of ino_t, inode number*/
ino_t d_ino;

 /* directory name, an array of size DIRSIZ */
char d_name[DIRSIZ];
}
```

UNIX system

User space: $\boxed{\text{user program}}$

$\boxed{\text{library}}$

Kernel space: $\boxed{\text{System Calls}}$

$\boxed{\text{kernel}}$

$\boxed{\text{Devices Driver}}$

$\boxed{\text{Hardware}}$

This is a hierarchy view of Unix system, where various file functions exist in uer space and kernel space in the different levels, and complete different jobs.

1. System calls

   The Unix system provides its service through a set of system calls, which are in effect functions within the operating system that may be called by user programs.

   It is sometimes important to employ system calls to achieve maximum efficiency, or to access some facilities that are not in library.

   And parts of standard library can be implemented with system calls.

2. Device drivers

   The Kernel consists of a number of device drivers, which are a collection of lower-level interfaces for controlling system hardware.

Device driver encapsulate the details of hardware dependent features. For example, a device driver for a tape device will give instructions that how to start the tape, wind it forward and backward,etc.

3. Library functions

   To provide a higher level interface to devices and disk files, Unix provides a number of standard libraries, e.g.$< stdio.h >$, $< time.h >$, $< stdlib.h >$, $< math.h >$, etc.

   Using Library functions, we can avoid the penalty in making system calls. That is Unix has to switch from running user code to executing the kernel code and back again, so the system calls are more expensive than function calls.

File descriptors

What is file descriptor?

If you read and write a file, the system will check:

1. Does this file exit?

2. Do you have permission to access it?

If all is well, it will return to the program **a small non-negative integer**, and this is file descriptor.

convention

when the shell run a program, three files are open, with file descriptors 0, 1, and 2, called the standard input, the standard output, and standard error.If a program reads 0, writes 1 and 2, it can do input and output without worrying about opening files.

How it works?

Whenever input or output is to be done on the file, the file descriptor is used instead of the name to identify the file.

For example:

```
ls -l >lsoutput.txt
```

In this case, the shell interpret the command of ls -l and reads into file descriptors 0 and writes to the named file(lsoutput.txt). And file descriptor 2 remains attached to the screen, so error message can go there.

Standard I/O Library

Standard input and output

1. standard input

   The simplest mechanism is to read one character at a time from the standard input, normally keyboard.

   ```
   int getchar(void)
   ```

   getchar returns the next input character each time it is called, or EOF when encounters end of life.

2. standard output

   The function

   ```
   int putchar(int)
   ```

is used for output: putchar(c) puts the character c on the standard output, which is by default the screen. putchar returns the character written, or EOF if an error occurs.

Each source file refers to an input/output library function must contain the line:

```
 #include <stdio.h>
```

" < ... >" tells compiler to search the header file in a standard place, e.g /usr/include.

One example: A program converts its input to lower case.

```
#include <stdio.h>
#include <ctype.h>

main()
```

```
{
int c;

while((c = getchar()) !=EOF)
putchar(tolower(c));
return 0;
}
```

Here, function tolower() is defined in $< ctype.h >$; it converts an upper case letter to lower case, and returns other characters untouched.

Type in this program and compile it. Execute this program to see how the system catch your input from keyboard and convert your input into lower case, and show the result on the screen.

File Access

The standard input/output concerns the system devices, which are automatically defined by local operating system.

How to write a program that access a file which is not directly connected to the program?

try this on the shell:

```
cat hello.c program.c
```

this command print the contents of file hello.c and program.c on the standard output, e.g. screen.

1. How to open a file?

   The function **fopen** is used for open a file. It takes the name of the file as an argument, and returns a file pointer to used in later on reads or writes of the file.

   file pointer points to a structure that contains information about file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred.

```
 fp = fopen(name, mode)
```

The first argument is the name of the file, the second argument is the mode, such as “r”, “w”, “a”.

Reading an non-exist file is an error!!! Writing or appending an non-exist file may create a new file. If any errors occurs, the fopen will return NULL.

2. How to read and write a file?

   The function

   ```
    int getc(FILE *fp)
   ```

   returns the next character from the stream referred to by fp; and returns EOF for end of file or error.

The function

```
 int putc(int c, FILE *fp)
```

writes the character c to the file fp and returns the character written, or EOF if an error occurs.

The FILE is a structure declared in $< stdio.h >$. Its usage is as followed:

```
FILE *fp;
FILE *fopen(char *name, char *mode)
```

This says that fp a pointer to a FILE, and fopen returns a pointer to a FILE.

3. Error handling -stderr and Exit

   Sometimes trouble comes out that a file is not accessible for some reason. We need to know the error occurs where, and easier for debugging.

When a C program is started, the operating system is responsible for opening three files and providing file pointers for them. These files are stdin and stdout, and stderr, and they are declared in $< stdio.h >$.

Normally, stdin connects to the keyboard, stdout and stderr connect to the screen, stdin and stdout may be redirected to files or pipes as we talked before.

4. A program for implementing cat

What is cat? It is not your pet. Try command “cat file1, file2”, to see what happens on your screen.

```
#include<stdio.h>

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

```
FILE *fp;
void filecopy(FILE *, FILE *);
char *prog = argv[0];

if(argc==1)
filecopy(stdin, stdout);
else
while(--argc > 0)
if((fp = fopen(*++argv, ‘‘r’’))==NULL){
fprintf(stderr, ‘‘%s: can’t open %s\n",
prog, *argv);
exit(1);
}else{ filecopy(fp, stdout);
fclose(fp);
}
if(ferror(stdout)){
fprintf(stderr,’’%s: error writing stdout\n’’,
exit(2);
}
exit(0);
}

void filecopy(FILE *ifp, FILE *ofp)
```

```
{
int c;

while((c=getc(ifp)) !=EOF)
putc(c, ofp);
}
```

The program signals errors two ways:

First, the diagnostic produced by fprintf goes onto stderr, so it finds way to the screen instead of disappearing down a pipeline or into an output file.

Second, the program uses the standard library function exit, with argument of non-zero integer to indicate an error.

The function $int\ ferror(FILE * fp)$ returns non-zero if an error occurred on the output stream fp. This situation is very rare, but it may happen, for example, the disk fills up, no enough space for writing on.

Type in this program and compile it.

Execute this program:

(a) Give commands

```
ls -l >lsoutput.txt,
ps >psoutput.txt
```

to create two files: lsoutput.txt, psoutput.txt.

(b) Execute this program with the two files as arguments.

Line input/output

```
char *fgets(char *line, int maxline, FILE *fp)
```

fgets reads the next input line (including the new line) in file **fp** into the character arrary **line**; at most maxline-1 characters will be read.

Normally, **fgets** returns **line**; on end of the file or error it returns NULL.

```
char *fputs(char *line, FILE *fp)
```

fputs writes a string(which need not contain a newline) to a file, it returns EOF if an error occurs, and zero otherwise.

Formatted input and output

- Formatted output

  ```
  int printf(char *format, arg_1, arg_2)
  ```

  printf converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

  The commonly used format are:

- Formatted input

```
int scanf(char *format, ...)
```

The function **scanf** is the input analog of **printf**, providing many of the same conversion facilities in the opposite direction.**scanf** reads characters from the standard input, interprets then according to the specification in **format**, and store the results through the remaining arguments. But be aware, the arguments to **scanf** must be pointers. The most common error is writing

```
scanf(``%d'', n);
```

instead of writing

```
scanf(``%d'', &n);
```

The common used format are :

For example: /* A small calculator */

```
#include <stdio.h>
```

```
main()
{
double sum, v;
sum = 0;
while(scanf("%lf", &v) == 1)
printf("\t%.2f\n", sum += v);
return 0;
}
```

scanf stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successful matched and assigned input items. This can be used to decided how many item were found.

Low-level File Access

Low level I/O -read and write

The system calls **read** and **write** we used before in shell script are accessed through two functions:

```
int n_read = read(int fd, char *buf, int n)

int n_written = write(int fd, char *buf, int n)
```

The first argument is a file descriptor, the second argument is a character array where the data is to go or come from. The third argument is the number of bytes to be transferred. Each call returns a count of the number of bytes transferred.

On reading, the number of bytes returned may be less than the number requested. A return value of zero bytes implies the end of file, and -1 indicates an error.

For writing, the return value is the number of bytes written, and this exactly equal to the number requested. Otherwise, it is an error.

How many numbers of bytes can be read or written in one call? **Any number**. But the most common value is 1, which means one character at a time. We call this pattern as **unbuffered**. And also the block size, e.g. 1024, 4096 that correspond to a physical block size on a peripheral device.

Of course, reading or writing a large size of chunk of file is more efficient than byte by byte, because less system calls. (Do you remember the system call penalty?)

- How they can be used?

  /* A program copy input to output */

  ```
  # include <unistd.h>
  # define BUFSIZ 14
  ```

```
main()
{
  char buf[BUFSIZ];
  int n;

  while((n==read(0, buf, BUFSIZ)) >0)
       write(1, buf, n);
  return 0;
}
```

This program will copy your input from the keyboard to anywhere of the buffer. You may not see any result on the screen.

How it works? Can you comment on the statements? How you can make it show you the result on the screen? (hint: change the fp in write function!)

/* A program of implementing getchar */

```
#include <stdio.h>
```

```
#include <unistd.h>
#define BUFSIZ 1024

int getchar(void)
{

  static char buf[BUFSIZ];
  static char *bufp = buf;
  static int n =0;

  if (n == 0){
    n= read(0, buf, sizeof buf);
    bufp =buf;
  }

  return (--n >= 0)?
         (unsigned char)*bufp++: EOF;
}
```

This is just a function declaration. How do you compile it?(Hope you remember!)

```
 gcc -c getchar.c
```

/* A program combine this getchar with the lowercase program */

```
#include <stdio.h>
#include <ctype.h>
#undef getchar

main()
{
int c;

while((c = getchar()) !=EOF)
putchar(tolower(c));
return 0;
}
```

What does “undef” means? How do you compile it and execute it?

1. `gcc -c lowercase.c`

2. `gcc -o lowercase.o getchar.o`

Open, Creat, Close, Unlink

- open **open** is rather like the **fopen**, except that instead of returning a file pointer, it returns a file descriptor, which is just an int. **open** returns -1 if any error occurs. This function is used as followed,

  ```
  #include<fcntl.h>

  int fd;
  int open(char *name, int flags, int perms);
  fd=open(name, flags, perms);
  ```

  The first argument is a character string containing the filename. The second argument, flags, is an int that specifies how the file is to be opened; the main values are:

  ```
  O_RDONLY: open for read only
  O_WRONLY: open for write only
  O_RDWR: open for both reading and writing
  ```

  These constants are defined in $< fcntl.h >$in UNIX system V. The perms argument for **open** is always 0.

So, open an existing file for reading can be coded as this:

```
fd= open(name, O_RDONLY, 0);
```

- creat

  It is an error to try to open a file that does not exist. The system call creat is provided to create new files, or to re-write old one.

  ```
  int creat(char *name, int perms);
  fd= creat(name, perms);
  ```

  It returns a file descriptor if it was able to create the file, and -1 if not. If the file already exists, **creat** will truncate it to zero length, thereby discarding its previous contents; it is not an error to **creat** a file that already exists.

  About the **perms**, the UNIX system provides a set of options for permission of control read, write and access for the owner of the file, for the owner's group, and for all others. For example, you have the command of changing the file permission on the shell.

```
chmod a+xr fist.sh
```

The common permission values are defined in$< sys/stat.h >$ as followed:

```
 read, write, execute permission for user
S_IRUSR, S_IWUSR, S_IXUSR,
 read, write, execute permission for group
S_IRGRP, S_IWGRP, S_IXGRP,
 read, write, execute permission for others
S_IROTH, S_IWOTH, S_IXOTH.
```

- close

  The function **close(int fd)** breaks the connection between the file descriptor and an open file, and frees the file descriptor for use with some other file; it corresponds to fclose in the standard library except that there is no buffer to flush.

- unlink

  The function **unlink(char *name)** removes the file from the file system. It corresponds to the standard library function **remove**.

- Many versions of copying files

  We can use **open, read, write** system calls to implement copy one file to another file.

  And we copy the file by using larger block. The following program copies file in 1k block, again use the system calls as above.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>

main(int argc, char *argv[])
{
 int f1, f2, n;
 char block[1024];

if(argc !=3)

fprintf(stderr, "Usage: cp from to ");

 if((f1 = open(argv[1], O_RDONLY, 0)) == -1)
fprintf(stderr, "cp: can't open %s",argv[1]);

 if ((f2 = open(argv[2], O_WRONLY|O_CREAT,
S_IRUSR|S_IWUSR)) == -1)
fprintf(stderr, "cp: can't creat %s",argv[2]);

 while((n = read (f1, block,sizeof(block)))>0)
```

```
     if(write(f2, block, n) != n)
     fprintf(stderr, "cp: write erro on file %s",
    argv[2]);

     exit(0);
    }
```

Random Access -lseek

Each **read, write** takes place at a position in the file right after the previous one, which means that it access the file sequentially and contiguous. Using **lseek**, we can get any position in a file relative to the original position.

```
long lseek(int fd, long offset, int origin);
```

This function means that set current position whose file descriptor is **fd** to the **offset**, which is taken relative to the location specified by **origin**.

Listing your directory

1. A program list out the subdirectories of the current directory.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>

void printdir (char *dir)
{/*function takes dir name as argument*/

/*define a pointer to DIR structure*/
    DIR *dp;
/*define a pointer to dirent structure*/
    struct dirent *entry;
/*define a structure of status info*/
    struct stat statbuf;

/*if opendir returns a null pointer,
then it can't open a directory*/
 if((dp = opendir(dir)) == NULL) {
  printf("cannot open directory: %s\n,", dir);
  return;
 }
/*else go to the current dir*/
  chdir(dir);

/*readir() reads from pointer dp and returns
a pointer (entry).If entry is not null, then
calls function lstat() to get the status info
of a member(d_name) in this entry, which was
stored in the location of statbuf(&statbuf)*/

  while((entry = readdir(dp)) != NULL) {
    lstat(entry -> d_name, &statbuf);

/*if the status is a directory,then print out
this member d_name*/

    if(S_ISDIR(statbuf.st_mode)) {
      printf("%s/\n",entry -> d_name);
    }
/*if the status is not a directory,
then ignore*/
```

```
        else continue;
      }
    /*close the directory,the pointer of the
    dir is passed as an argument, in which the
    dir is stored*/
         closedir(dp);
    }
    int main(int argc, char *argv[])
    {
      /* set current dir as topdir*/
      char *topdir = ".";
      if(argc>=2)
        topdir=argv[1];

      printf("Directory scan of %s\n", topdir);
      printdir(topdir);
      printf("done.\n");

      exit(0);
    }
```

2. An extension list of all the dirs under subdirs, but it ignores the "." and ".." directory names.

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>

void printdir (char *dir, int depth)
{/*function takes dir name as argument*/

/*define a pointer to DIR structure*/
    DIR *dp;
/*define a pointer to dirent structure*/
    struct dirent *entry;
/*define a structure of status info*/
    struct stat statbuf;

/*if opendir returns a null pointer,
then it can’t open a directory*/
 if((dp = opendir(dir)) == NULL) {
  printf("cannot open directory: %s\n,", dir);
  return;
 }
/*else go to the current dir*/
  chdir(dir);

/*readir() reads from pointer dp and returns
a pointer (entry).If entry is not null, then
calls function lstat() to get the status info
of a member(d_name) in this entry, which was
stored in the location of statbuf(&statbuf)*/

  while((entry = readdir(dp)) != NULL) {
    lstat(entry -> d_name, &statbuf);

/*if the status is a directory,then print out
this member d_name*/

    if(S_ISDIR(statbuf.st_mode)) {

/*but call for string comparson strcmp() to
detect d_name which are equal to "."and "..",
then ignore them*/

    if(strcmp(".", entry -> d_name) == 0 ||
       strcmp ("..", entry -> d_name) == 0 )
    continue;

/*printout the list of d_name with the format
  having a number(depth) of blank space.*/

printf("%.*s%s/\n",depth, " ",entry->d_name);

/*recursively call the function printdir() to
  printout all the dirs of subdirs*/
      printdir(entry -> d_name, depth+4);
    }
/*if status is not a directory, then ignore*/
    else continue;
  }
  /* go to parent dirs, start from there*/
  chdir("..");

/*close the directory,the pointer of the
dir is passed as an argument, in which the
dir is stored*/
     closedir(dp);
}
```

```
int main(int argc, char *argv[])
{
  /* set current dir as topdir*/
  char *topdir = ".";
  if(argc>=2)
    topdir=argv[1];

  printf("Directory scan of %s\n", topdir);
  printdir(topdir,0);
  printf("done.\n");

  exit(0);
}
```

THE END