Library I/O Functions

9

Abstract

This chapter covers library I/O functions. It explains the roles of library I/O functions and their advantages over system calls. It uses sample programs to show the relationship between library I/O functions and system calls, and it explains their similarities and fundamental differences. It describes the algorithms of library I/O functions in detail. These include the algorithms of fread, fwrite and fclose, with an emphasis on their interactions with read, write and close system calls. It describes the different modes of library I/O functions, which include char mode, line mode, structured records mode and formatted I/O operations. It explains the buffering schemes of file streams and shows the effects of different buffering schemes by example programs. It explains functions with varying parameters and how to access parameters using stdarg macros.

The programming project is to integrate the principles and programming techniques of the chapter to implement a printf-like function for formatted printing of chars, strings and numbers in different number bases in accordance with a format string. The basis of the printf-like function is putchar() of Linux, but it works exactly the same as the library function printf(). The purpose of the project is to let the reader understand how library I/O functions are implemented.

9.1 Library I/O Functions

System calls are the basis of file operations, but they only support read/write of chunks of data. In practice, a user program may want to read/write files in logical units most suited to the application, e.g. as lines, chars, structured records, etc. which are not supported by system calls. Library I/O functions are a set of file operation functions, which provide both user convenience and overall efficiency (GNU I/O on streams 2017; GNU libc 2017; GNU Library Reference Manual 2017).

9.2 Library I/O Functions vs. System Calls

Almost every operating system that supports C programming also provides library functions for file I/O. In Unix/Linux, library I/O functions are built on top of system calls. In order to illustrate their intimate relationship, we first list a few of them for comparison.

K. C. Wang, *Systems Programming in Unix/Linux*, https://doi.org/10.1007/978-3-319-92429-8_9

System Call Functions: open(), read(), write(), lseek(), close();
Library I/O Functions: fopen(), fread(), fwrite(), fseek(), fclose();

From their strong similarities, the reader can almost guess that every library I/O function has its root in a corresponding system call function. This is indeed the case as fopen() relies on open(), fread() relies on read(), etc. The following C programs illustrate their similarities and differences.

Example 9.1: Display File Contents

```
        System Calls                    |     Library I/O Functions
----------------------------------------|---------------------------------
#include <fcntl.h>                      | #include <stdio.h>
int main(int argc, char *argv[ ])       | int main(int argc, char *argv[ ])
{                                       | {
1. int fd;                              |   FILE *fp;
   int i, n;                            |
   char buf[4096];                      |   int c;   // for EOF of stdin
   if (argc < 2) exit(1);               |   if (argc < 2) exit(1);
2. fd = open(argv[1], O_RDONLY);        |   fp = fopen(argv[1], "r");
   if (fd < 0) exit(2);                 |   if (fp==0) exit(2);
3. while (n = read(fd, buf, 4096)){     |   while ((c = fgetc(fp))!= EOF){
     for (i=0; i<n; i++){               |
       write(1, &buf[i], 1);            |     putchar(c);
     }                                  |
   }                                    |   }
}                                       | }
-------------------------------------------------------------------------
```

The left-hand side shows a program that uses system calls. The right-hand side shows a similar program that uses library I/O functions. Both programs print the contents of a file to the display screen. The two programs look similar but there are fundamental differences between them.

Line 1: In the system call program, the file descriptor fd is an integer. In the library I/O program, fp is a FILE stream pointer.

Line 2: The system call open() opens a file for read and returns an integer file descriptor fd, or -1 if open() fails. The library I/O function fopen() returns a FILE structure pointer, or NULL if fopen() fails.

Line 3: The system call program uses a while loop to read/write the file contents. In each iteration, it issues a read() system call to read up to 4KB chars into a buf[ ]. Then it writes each char from buf[ ] to the file descriptor 1, which is the standard output of the process. As pointed out before, using system calls to write one byte at a time is grossly inefficient. In contrast, the library I/O program simply uses fgetc(fp) to get chars from the FILE stream, output them by putchar() until EOF.

Besides the slight differences in syntax and functions used, there are some fundamental differences between the two programs, which are explained in more detail below.

Line 2: fopen() issues an open() system call to get a file descriptor fd. If the open() call fails, it returns a NULL pointer. Otherwise, it allocates a FILE structure in the program's **heap area.** The FILE

structure contains an internal buffer char fbuf[BLKSIZE] and an integer fd field. It records the file descriptor returned by open() in the FILE structure, initializes fbuf[ ] as empty, and returns the address of the FILE structure as fp.

Line 3: fgetc(c, fp) tries to get a char from the file stream fp. If the fbuf[ ] in the FILE structure is empty, it issues a read(fd, fbuf, BLKSIZE) system call to read BLKSIZE bytes from the file, where BLKSIZE matches the file system block size. Then it returns a char from fbuf[ ]. Subsequently, fgetc() returns a char from fbuf[ ] as long as it still has data. Thus, the library I/O read functions issues read() syscalls only to refill the fbuf[ ] and they transfer data from the OS kernel to user space always in BLKSZISE. Similar remarks also apply to library I/O write functions.

Exercise 9.1: In the system call program of Example 9.1, writing each char by a system call is very inefficient. Replace the for loop by a single write() system call.

Example 9.2: Copy files: Again, we list two versions of the program side by side to illustrate their similarities and differences

```
        System Calls                          Library I/O Functions
--------------------------------------|------------------------------
#include <fcntl.h>                    | #include <stdio.h>
#define BLKSIZE 4096                  | #define BLKSIZE 4096
int fd, gd;                           | FILE *fp, *gp;
char buf[4096];                       | char buf[4096];
                                      |
int main(int argc,char *argv[])       | int main(int argc,char *argv[])
{                                     | {
  int n, total=0;                     |  int n, total=0;
  if (argc < 3) exit(1);              |  if (argc < 3) exit(1);
  // check for same file              |  // check for same file
  fd = open(argv[1], O_RDONLY);       |  fp = fopen(argv[1], "r");
  if (fd < 0) exit(2);                |  if (fp == NULL) exit(2);
  gd = open(argv[2],O_WRONLY|O_CREAT);|  gp = fopen(argv[2], "w");
  if (gd < 0) exit(3);                |  if (gp == NULL) exit(3);
  while(n=read(fd, buf, BLKSIZE))     |   while(n=fread(buf,1,BLKSIZE,fp))
  {                                   |  {
      write(gd, buf, n);              |     fwrite(buf, 1, n, gp);
      total += n;                     |     total += n;
  }                                   |  }
  printf("total=%d\n",total);         |  printf("total = %d\n", total);
  close(fd); close(gd);               |  fclose(fp); fclose(gp);
 }                                    | }
 ------------------------------------------------------------------------
```

Both programs copy a src file to a dest file. Since the system call program was already explained in Chapter 6, we shall only discuss the program that uses library I/O functions.

(1). fopen() uses a string for Mode, where "r" for READ, "w" for WRITE. It returns a pointer to a FILE structure. fopen() first issues an open() system call to open the file to get a file descriptor number fd. If the open() system call fails, fopen() returns a NULL pointer. Otherwise, it allocates a FILE structure in the program's heap area. Each FILE structure contains an internal buffer, fbuf

[BLKSIZE], whose size usually matches the BLKSIZE of the file system. In addition, it also has pointers, counters and status variables for manipulating the fbuf[ ] contents and a fd field, which stores the file descriptor from open(). It initializes the FILE structure and returns fp which points at the FILE structure. It is important to note that the FILE structure is in the process' User mode Image. This means that calls to library I/O functions are ordinary function calls, not system calls.

(2). The programs terminates if any of the fopen() calls has failed. As mentioned above, fopen() returns a NULL pointer on failure, e.g. if the file can not be opened for the indicated mode.

(3). Then it uses a while loop to copy the file contents. Each iteration of the while loop tries to read BLKSIZE bytes from the source file, and write n bytes to the target file, where n is the returned value from fread(). The general forms of fread() and fwrite() are

```
int n = fread(buffer, size, nitems, FILEptr);
int n = fwrite(buffer,size, nitems, FILEptr);
```

where size is the record size in bytes, nitems is the number of records to be read or written, and n is the actual number of records read or written. These functions are intended for read/write structured data objects. For example, assume that the buffer area contains data objects of structured records

```
struct record{.....}
```

We may use

```
n = fwrite(buffer, sizeof(struct record), nitem, FILEptr);
```

to write nitem records to a file. Similarly,

```
n =   fread(buffer, sizeof(struct record), nitem, FILEptr);
```

reads nitem records from a file.

The above program tries to read/write BLKSIZE bytes at a time. So, it has size = 1 and nitems = BLKSIZE. As a matter of fact, any combination of size and nitems such that size*nitems = BLKSIZE would also work. However, using a size > 1 may cause problem on the last fread() because the file may have fewer than size bytes left. In that case, the returned n is zero even if it has read some data. To deal with the "tail" part of the source file, we may add the following lines of code after the while loop:

```
fseek(fp, total, SEEK_SET);  // fseek to byte total
n = fread(buf, 1, size, fp); // read remaining bytes
    fwrite(buf,1, n, gp);    // write to dest file
total += n;
```

fseek() works exactly the same as lseek(). It positions the file's R|W pointer to the byte location total. From there, we read the file as 1-byte objects. This will read all the remaining bytes, if any, and write them to the target file.

(4). After the copying is complete, both files are closed by fclose(FILE *p).

9.3 Algorithms of Library I/O Functions

9.3.1 Algorithm of fread

The algorithm of fread() is as follows:

(1). On the first call to fread(), the FILE structure's buffer is empty. fread() uses the saved file descriptor fd to issue a

```
n = read(fd, fbuffer, BLKSIZE);
```

system call to fill the internal fbuf[ ] with a block of data. Then, it initializes fbuf[ ]'s pointers, counters and status variables to indicate that there is a block of data in the internal buffer. It then tries to satisfy the fread() call from the internal buffer by copying data to the program's buffer area. If the internal buffer does not have enough data, it issues additional read() system call to fill the internal buffer, transfer data from the internal buffer to the program buffer, until the needed number of bytes is satisfied (or the file has no more data). After copying data to the program's buffer area, it updates the internal buffer's pointers, counters, etc. making it ready for next fread() request. It then returns the actual number of data objects read.

(2). On each subsequent call to fread(), it tries to satisfy the call from the FILE structure's internal buffer. It issues a read() system call to refill the internal buffer whenever the buffer becomes empty. Thus, fread() accepts calls from user program on one side and issues read() system calls to the OS kernel on the other. Except for the read() system calls, all processing of fread() are performed inside the user mode image. It enters the OS kernel only when needed and it does so in a way that matches the file system for best efficiency. It provides automatic buffering mechanism so that user programs do not have to worry about such detailed operations.

9.3.2 Algorithm of fwrite

The algorithm of fwrite() is similar to that of fread() except for the data transfer direction. Initially the FILE structure's internal buffer is empty. On each call to fwrite(), it writes data to the internal buffer, and adjusts the buffer's pointers, counters and status variable to keep track of the number of bytes in the buffer. If the buffer becomes full, it issues a write() system call to write the entire buffer to the OS kernel.

9.3.3 Algorithm of fclose

fclose() first flushes the local buffer of the FILE stream if the file was opened for WRITE. Then it issues a close(fd) system call to close the file descriptor in the FILE structure. Finally it frees the FILE structure and resets the FILE pointer to NULL.

9.4 Use Library I/O Function or System Call

Based on the above discussions, we can now answer the question of when to use system calls or library functions to do file I/O. fread() relies on read() to copy data from Kernel to an internal buffer, from which it copies data to the program's buffer area. So it transfers data twice. In contrast, read() copies data from Kernel directly to the program's buffer area, which copies only once. Thus, for read/write data in units of BLKSIZEs, read() is inherently more efficient than fread() because it only needs one copying operation instead of two. Similar remarks are also applicable to write() and fwrite().

It is noted that in some implementations of fread() and fwrite(), e.g. in the GNU libc library, if the requested size is in units of BLKSIZE, they may use system calls to transfer data in BLKSIZE from kernel to user specified buffer directly. Even so, using library I/O functions still require additional function calls. Therefore, in the above example, the program that uses system calls is actually more efficient than the one that uses library I/O functions. However, if the read/write is not in units of BLKSIZE, fread() and fwrite() may be far more efficient. For example, if we insists on RlW one byte at a time, fread() and fwrite() would be far better because they enter the OS kernel only to fill or flush the internal buffer, not on every byte. Here, we have implicitly assumed that entering Kernel mode is more expensive than staying in User mode, which is indeed true.

9.5 Library I/O Modes

The Mode parameter in fopen() may be specified as
"r", "w", "a" : for READ, WRITE, APPEND.

Each of the mode string may include a + sign, which means for both READ, WRITE, and in the cases of WRITE or APPEND, create the file if it does not exist.
"r+" : for R/W, without truncating the file.
"w+": for R/W, but truncate the file first; create if file does not exist.
"a+" : for R/W by append; create if file does not exist.

9.5.1 Char Mode I/O

```
int fgetc(FILE *fp):           // get a char from fp, cast to int.
int ungetc(int c, FILE *fp);   // push a previously char got by fgetc()
                                  back to stream
int fputc(int c, FILE *fp);    // put a char to fp
```

Note that fgetc() returns an integer, not a char. This is because it must return EOF on end of file. The EOF symbol is normally an integer -1, which distinguishes it from any char from the FILE stream.

For fp = stdin or stdout, c = getchar(); putchar(c); may be used instead. For run time efficiency, getchar() and putchar() are often not the shortened versions of getc() and putc(). Instead, they may be implemented as macros in order to avoid an extra function call.

Example 9.3: Char Mode I/O

```
(1). /* file copy using getc(), putc() */
   #include <stdio.h>
   FILE *fp,*gp;
   int main()
   {
      int c;  /* for testing EOF */
      fp=fopen("source", "r");
      gp=fopen("target", "w");
      while ( (c=getc(fp)) != EOF )
          putc(c,gp);
      fclose(fp); fclose(gp);
    }
```

Exercise 9.2: Write a C program which converts letters in a text file from lowercase to uppercase.

Exercise 9.3: Write a C program which counts the number of lines in a text file.

Exercise 9.4: Write a C program which counts the number of words in a text file. Words are sequences of chars separated by white spaces.

Exercise 9.5: Linux's man pages are compressed gz files. Use gunzip to uncompress a man page file. The uncompressed man page file is a text file but it is almost unreadable because it contains many special chars and char sequences for the man program to use. Assume ls.1 is the uncompressed man page of ls.1.gz. Analyze the uncompressed ls.1 file to find out what are the special chars. Then write a C program which eliminates all the special chars, making it a pure text file.

9.5.2 Line mode I/O

char *fgets(char *buf, int size, FILE *fp): read a line (with a \n at end) of at most size chars from fp to buf .

int fputs(char *buf, FILE *fp): write a line from buf to fp.

Example 9.4: Line Mode I/O

```
#include <stdio.h>
FILE *fp,*gp;
char buf[256]; char *s="this is a string";
int main()
{
  fp = fopen("src",  "r");
  gp = fopen("dest", "w");
  fgets(buf, 256, fp); // read a line of up to 255 chars to buf
  fputs(buf, gp);      // write line to destination file
}
```

When fp is stdin or stdout, the following functions may also be used but they are not shortened versions of fgets() and fputs().

```
gets(char *buf);      // input line from stdin but without checking length
puts(char *buf);      // write line to stdout
```

9.5.3 Formatted I/O

These are perhaps the most commonly used I/O functions.

Foratted Inputs: (FMT=format string)

```
scanf(char *FMT, &items);          // from stdin
fscanf(fp, char *FMT, &items);     // from file stream
```

Formatted Outputs:

```
printf(char *FMT, items);          // to stdout
fprintf(fp, char *FMT, items);     // to file stream
```

9.5.4 In-memory Conversion Functions

```
sscanf(buf, FMT, &items);     // input from buf[ ] in memory
sprintf(buf,FMT, items);      // print to buf[ ] in memroy
```

Note that sscanf() and sprintf() are not I/O functions but in-memory data conversion functions. For example, atoi() is a standard library function, which converts an string of ASCII digits to integer, but most Unix/Linux system do not have an itoa() function because the conversion can be done by sprintf (), so it is not needed.

9.5.5 Other Library I/O Functions

fseek(), ftell(), rewind()	:	change read/write byte position in file stream
feof(), ferr(), fileno()	:	test file stream status
fdopen()	:	open a file stream by a file descriptor
freopen()	:	reopen an existing stream by a new name
setbuf(), setvbuf()	:	set buffering scheme
popen()	:	create a pipe, fork a subprocess to invoke the sh

9.5.6 Restriction on Mixed fread-fwrite

When a file stream is for both R|W, there are restrictions on the use of mixed fread() and fwrite() calls. The specification requires at least one fseek() or ftell() between every pair of fread() and fwrite().

Example 9.5: Mixed fread-fwrite: This program yields different results when run under HP-UX and Linux.

```
#include <stdio.h>
FILE fp; char buf[1024];
int main()
{
  fp = fopen("file", "r+");    // for both R/W
  fread(buf, 1, 20, fp);       // read 20 bytes
  fwrite(buf,1, 20, fp);       // write to the same file
}
```

Linux gives the right result, which modifies the bytes from 20 to 39. HP-UX appends 40 bytes to the end of the original file. The difference stems from the non-uniform treatment of R/W pointers in the two systems. Recall that fread()/fwrite() issue read()/write() system calls to fill/flush the internal buffer. While read()/write() use the R/W pointer in the file's OFTE, fread()/fwrite() use the local buffer's R/W pointer in the FILE structure. Without a fseek() to synchronize these two pointers, the results depend on how are they used in the implementations. In order to avoid any inconsistencies, follow the suggestions of the man pages. For the Example 9.5 program, the results become identical (and correct) if we insert a line

```
fseek(fp, (long)20, 0);
```

between the fread() and fwrite() lines.

9.6 File Stream Buffering

Every file stream has a FILE structure, which contains an internal buffer. Read from or write to a file stream goes through the internal buffer of the FILE structure. A file stream may employ one of three kinds of buffering schemes.

. **unbuffered:** characters written to or read from an unbuffered stream are transmitted individually to or from the file as soon as possible. For example, the file stream stderr is usually unbuffered. Any outputs to stderr will appear immediately.
. **line buffered:** characters written to a line buffered stream are transmitted in blocks when a newline char is encountered. For example, the file stream stdout is usually line buffered. It outputs data line by line.
. **fully buffered:** characters written to or read from a fully buffered stream are transmitted to or from the file in block size. This is the normal buffering scheme of file streams.

After creating a file stream by fopen() and before any operation has been performed on it, the user may issue a

```
setvbuf(FILE *stream, char *buf,  int node,  int size)
```

call to set the buffer area (buf), buffer size (size) and buffering scheme (mode), which must be one of the macros

_IONBUF: unbuffered
_IOLBUF: line buffered
_IOFBUF: fully buffered

In addition, there are other setbuf() functions, which are variants of setvbuf(). The reader may consult the man pages of setvbuf for more details.

For line or fully buffered streams, **fflush(stream)** can be used to flush out the stream's buffer immediately. We illustrate the different buffering schemes by an example.

Example 9.6. FILE stream buffering: Consider the following C program

```
#include <stdio.h>
int main()
{
  (1).  // setvbuf(stdout, NULL, _IONBF, 0);
        while(1){
  (2).     printf("hi ");  // not a line yet
  (3).     // fflush(stdout);
           sleep(1);       // sleep for 1 second
         }
}
```

When running the above program, no outputs will appear immediately despite the printf() statement at line (2) on each second. This is because stdout is line buffered. Outputs will appear only when the printed chars fill the internal buffer of stdout, at which time, all the previously printed chars will appear at once. So, if we do not write lines to stdout, it behaves as a fully buffered stream. If we uncomment line (3), which flushes out stdout, each print will appear immediately despite it is not a line yet. If we uncomment line (1), which sets stdout to unbuffered, then the printing will appear on each second.

9.7 Functions with Varying Parameters

Among the library I/O functions, printf() is rather unique in that it can be called with a varying number of parameters of different types. This is permissible because the original C was not a type-checking language. Current C and C++ enforce type-checking but both still allow functions with varying number of parameters for convenience. Such functions must be declared with at least one argument, followed by 3 dots, as in

```
int func(int m, int n . . .)          // n = last specified parameter
```

Inside the function, parameters can be accessed by the C library macros

```
void va_start(va_list ap, last);  // start param list from last parameter
type va_arg(va_list ap, type);    //  type = next parameter type
va_end(va_list ap);               // clear parameter list
```

We illustrate the usage of these macros by an example:

Example 9.7: Access parameter list using stdarg Macros.

```
/******* Example of accessing varying parameter list ********/
#include <stdio.h>
#include <stdarg>     // need this for va_list type

// assume: func() is called with m integers, followed by n strings
int func(int m, int n . . .) // n = last known parameter
{
     int i;
(1). va_list ap;         // define a va_list ap
(2). va_start(ap, n);    // start parameter list from last param n
     for (i=0; i<m; i++)
(3).   printf("%d ", va_arg(ap, int));   // extract int params
     for (i=0; i<n, i++)
(4).   printf("%s ", va_arg(ap, char *)) // extra char* params
(5).   va_end(ap);
}
int main()
{
   func(3, 2, 1, 2, 3, "test", "ok");
}
```

In the example program, we assume that the function func() will be called with 2 known parameters, int m, int n, followed by m integers and n strings.

Line (1) defines a va_list ap variable.
Line (2) starts the parameter list from the last known parameter (n).
Line (3) uses va_arg(ap, **int**) to extract the next m parameters as integers.
Line (4) uses va_arg(ap, char *) to extract the next n parameters as strings.
Line (5) ends the parameter list by resetting the ap list to NULL.

The reader may compile and run the example program. It should print **1 2 3 test OK.**

9.8 Programming Project: Printf-like Function

The programming project is to write a printf()-like function for formatted printing of chars, strings, unsigned integers, signed integers in decimal and unsigned integers in HEX. The objective of the programming project is to let the reader understand how library I/O functions are implemented. In the case of printf(), which can print a varying number of items of different types, the basic operation is to print a single char.

9.8.1 Project Specification

In Linux, putchar(char c) prints a char. Use only putchar() to implement a function

```
int myprintf(char *fmt, . . .)
```

for formatted printing of other parameters, where fmt is a format string containing

```
%c : print char
%s : print string
%u : print unsigned integer
%d : print signed integer
%x : print unsigned integer in HEX
```

For simplicity, we shall ignore field width and precision. Just print the parameters, if any, as specified in the format string. Note that the number and type of the items to be printed are implicitly specified by the number of % symbols in the format string.

9.8.2 Base Code of Project

(1). The reader should implement a prints(char *s) function for printing strings.

(2). The following shows a printu() function, which prints unsigned integers in decimal.

```
char *ctable = "0123456789ABCDEF";
int BASE = 10; // for printing numbers in decimal
int rpu(unsigned int x)
{
    char c;
    if (x){
       c = ctable[x % BASE];
       rpu(x / BASE);
       putchar(c);
    }
}
int printu(unsigned int x)
{
   (x==0)? putchar('0') : rpu(x);
   putchar(' ');
}
```

The function rpu(x) recursively generates the digits of x % 10 in ASCII and prints them on the return path. For example, if x=123, the digits are generated in the order of '3', 2', '1', which are printed as '1', '2', '3' as they should.

(3). With the printu() function, the reader should be able to implement a printd() function to print signed integers.
(4). Implement a printx() function to print unsigned integers in HEX (for addresses).
(5). Assume that we have printc(), prints(), printd(), printu() and printx() functions. Then implement myprintf(char *fmt, . . .) by the following algorithm.

9.8.3 Algorithm of myprintf()

Assume that the format string fmt = "char=**%c** string=**%s** integer=**%d** u32=**%x**\n". It implies that there are 4 additional parameters of char, char *, int, unsigned int, types respectively. The algorithm of myprint() is as follows.

(1). Scan the format string fmt. Print any char that's not %. For each '\n' char, print an extra '\r'char.
(2). When encounter a '%', get the next char, which must be one of 'c', 's', 'u', 'd' or 'x'. Use va_arg (ap, type) to extract the corresponding parameter. Then call the print function by the parameter type.
(3). The algorithm ends when scanning the fmt string ends.

Exercise 9.6: Assume 32-bit GCC. Implement the myprintf(char *fmt, . . .) function by the following algorithm and explain WHY the algorithm works.

```
int myprintf(char *fmt, . . .)
{
   char *cp = fmt;
   int  *ip = (int *)&fmt + 1;
   // Use cp to scan the format string for %TYPE symbols;
   // Use ip to access and print each item by TYPE;
}
```

9.8.4 Project Refinements

(1). Define tab key as 8 spaces. Add %t to the format string for tab keys.
(2). Modify %u, %d and %x to include a width field, e.g. %8d prints an integer in a space of 8 chars and right-justified, etc.

9.8.5 Project Demonstration and Sample Solutions

Demonstrate the myprintf() function by driver programs. Sample solution to the programming project is available at the book's website for downloading. Source code of the project for instructors is available on request.

9.9 Summary

This chapter covers library I/O functions. It explains the roles of library I/O functions and their advantages over system calls. It uses sample programs to show the relationship between library I/O functions and system calls, and it explains their similarities and fundamental differences. It describes the algorithms of library I/O functions in detail. These include the algorithms of fread, fwrite and fclose, with an emphasis on their interactions with read, write and close system calls. It describes the different modes of library I/O functions, which include char mode, line mode, structured records mode and formatted I/O operations. It explains the buffering schemes of file streams and shows the effects of different buffering schemes by example programs. It explains functions with varying parameters and how to access parameters using stdarg macros.

The programming project is to integrate the principles and programming techniques of the chapter to implement a printf-like function for formatted printing of chars, strings and numbers in different number bases in accordance with a format string. The basis of the printf-like function is putchar() of Linux, but it works exactly the same as the library function printf(). The purpose of the project is to let the reader understand how library I/O functions are implemented.

Problems

1. Given the following C programs

```
---------- System call ---------------- Library I/O  -------------
#include <fcntl.h>                  |   #include <stdio.h>
char buf[4096];                     |   char buf[4096];
#define SIZE 1                      |   #define SIZE 1
int main()                          |   int main()
{                                   |   {
  int i;                            |     int i;
  int fd = open("file", O_RDONLY);  |     FILE *fp = fopen("file","r")
  for (i=0; i<100; i++){            |      for (i=0; i<100; i++)
    read(fd, buf, SIZE);            |       fread(buf, SIZE, 1, fp);
  }                                 |     }
}                                   |   }
-------------------------------------------------------------------
```

   (1). Which program would run faster? WHY?
   (2). Change SIZE to 4096 which is the file block size of Linux. Which program would run faster? WHY?
2. When copying files, we should never copy a file to itself.
   (1). WHY?
   (2). Modify the Example 9.2 programs to prevent copying a file to itself. Note that comparing the file names may not work because two distinct file names could be the same file due to hard links. So the problem is how to tell whether two pathnames are the same file. HINT: stat system call.
3. When copying directories, we should never copy a directory into itself. Assume that /a/b/c/ is a directory. It is permissible to copy /a/b/c into /a, but not /a/b into /a/b/c/.
   (1). WHY?
   (2). How to determine a directory is inside another directory?
   (3). Write C code to determine whether a directory is inside itself.

References

The GNU C Library:I/O on Streams, www.gnu.org/s/libc/manual/html_node/I_002fO-Overview.html, 2017
The GNU C Library (glibc), www.gnu.org/software/libc/, 2017
The GNU C Library Reference Manual, www.gnu.org/software/libc/manual/pdf/libc.pdf, 2017