Standard I/O Library

System Programming

2019 여름 계절학기

한양대학교 공과대학 컴퓨터소프트웨어학부
홍석준

I. Standard I/O Library

- The standard I/O library by Dennis Ritchie around 1975 based on the Portable I/O library by Mike Lesk
- The ISO C standard
- Additional interfaces by the Single UNIX Specification
- Easy-to-use library for buffer allocation, I/O in optimal-sized chunks, etc.

I. Streams and `FILE` Objects

- **When we open or create a file with the standard I/O library, a stream is associated with the file.**
- **The standard I/O function `fopen` returns a pointer to a FILE object, i.e. a *file pointer*, containing all the information to manage the stream.**
  - The file descriptor, a pointer to a buffer, the size of the buffer, # of characters in the buffer, an error flag, and the like.
- **Three predefined streams: `stdin`, `stdout`, and `stderr`**

I. Buffering

- **To use the minimum number of `read` and `write` calls.**
- **Three types: fully buffered, line buffered, and unbuffered**

- **Fully buffered**
  - Actual I/O takes place, when the buffer is filled.
  - The buffer is obtained by the first I/O function on a stream.
  - The buffer can be flushed (i.e., writing out its contents.)
    - Automatically when the buffer fills, or
    - `fflush` flushes a stream.

I. Buffering

- **Line buffered**
  - Actual I/O is performed, when a new line char is encountered on input or output.
  - Typically used on a stream to refer to a terminal.
  - Two caveats
    - Actual I/O might take place if the buffer is filled before a newline.
    - All line-buffered output streams are flushed, if input is requested from an unbuffered or a line-buffered stream.

- **Unbuffered**
  - No buffering
  - The `stderr` is normally unbuffered.

I. Buffering

- **ISO C requirements**
  - `stdin` and `stdout` are fully buffered, iff it is not an interactive device.
  - `stderr` is never fully buffered.
- **In most implemtations,**
  - `stderr` is always unbuffered.
  - All other streams are line buffered if referring to a terminal device; otherwise, they are fully buffered.
- **These defaults can be changed by either of `setbuf` and `setvbuf`.**

I. Buffering

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
int setvbuf(FILE *fp, char *buf, int mode, size_t
  size);
```

- **To enable/disable buffering, we set `setbuf`'s *buf* to non-null/`NULL`.**
- **`setvbuf`**
  - *mode*: `_IOFBF`, `_IOLBF`, and `_IONBF`.
  - Optional *buf* and *size* arguments

```
#include <stdio.h>
int fflush(FILE *fp);
```

- **A stream or all output streams, if *fp* is `NULL`, are flushed.**

I. Buffering

Function	*mode*	*buf*	Buffer and length	Type of buffering
`setbuf`		non-null	user *buf* of length `BUFSIZ`	fully buffered or line buffered
		`NULL`	(no buffer)	unbuffered
`setvbuf`	`_IOFBF`	non-null	user *buf* of length *size*	fully buffered
		`NULL`	system buffer of appropriate length	
	`_IOLBF`	non-null	user *buf* of length *size*	line buffered
		`NULL`	System buffer of appropriate length	
	`_IONBF`	(ignored)	(no buffer)	unbuffered

I. Opening a Stream

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *type);
FILE *freopen(const char *pathname, const char *type, FILE
   *fp);
FILE *fdopen(int filedes, const char *type);
```

- `freopen` **opens a file on a specified stream, closing the stream first, if it's already open. (typically used to open the file as one of** `stdin`, `stdout`, **and** `stderr`.**)**

- `fdopen` **is often used with pipes and network communication channels. (no** `fopen` **for these special files, so we have to call the device-specific function to obtain a file descriptor, and then** `fdopen`**)**

I. Opening a Stream

- **`b` stands for a binary file, but no effect for Unix.**
- **With `fdopen`,**
  - No truncation for opening for write, since it has already been opened.
  - Append mode can not create the file, since it already exists.

type	Description
`r` or `rb`	open for reading
`w` or `wb`	truncate to 0 length or create for writing
`a` or `ab`	append; open for writing at EOF, or create for writing
`r+`, `r+b` or `rb+`	open for reading and writing
`w+`, `w+b` or `wb+`	truncate to 0 length or create for reading and writing
`a+`, `a+b` or `ab+`	open or create for reading and writing at EOF

I. Opening a Stream

```
#include <stdio.h>

int fclose(FILE *fp);
```

- Any buffered output data is flushed, and an automatically allocated buffer is released.
- When a process terminates, all I/O streams are flushed and closed.

❑ Unformatted I/O

- Character-at-a-time I/O, e.g. `getc`
- Line-at-a-time I/O, e.g. `fgets` and `fputs`
- Direct I/O, e.g. `fread` and `fwrite`
  - a.k.a. binary I/O, object-at-a-time, record-oriented I/O, or structure-oriented I/O
  - Read or write some number of objects, often used for binary files

I. Reading and Writing a Stream

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

- All three return the next char (as an `unsigned char` converted to an `int`) if OK, EOF on end of file or error.
- `getchar` is equivalent to `getc(stdin)`
- `getc` can be implemented as a macro, but `fgetc` is guaranteed to be a function.

```
#include <stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
void clearerr(FILE *fp);  /* clear both flags */
```

- The *getc* functions return the same value whether an error occurs or the end of file is reached. We call `ferror` or `feof` to distinguish between the two.

I. Reading and Writing a Stream

```
#include <stdio.h>
int ungetc(int c, FILE *fp);
```

- Pushback of a char
- does not need to be the same char read.

-
```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

- `putchar(c) = putc(c,stdout)`

- `putc` **can be implemented as a macro, but** `fputc` **must as a function.**

I. Line-at-a-Time I/O

```
#include <stdio.h>

char *fgets(char *buf, int n, FILE *fp);

char *gets(char *buf);
```

- Both return `buf` if OK, `NULL` on end of file or error.
- `fgets` read up to `n`-1 chars, including a newline, and the buffer is null-terminated.
- `gets` is a deprecated function subject to buffer overflow, if the line is longer than the buffer, and it does not store a new line.

I. Line-at-a-Time I/O

```
#include <stdio.h>
int fputs(const char *str, FILE *fp);
int puts(const char *str);
```

- `fputs` **write the null-terminated string to the stream.**
  - No need to be line-at-a-time output, since the string need not contain a newline at the last char (which is usually the case.)

- `puts` **writes the null-terminated string to the standard output, and then a newline.**

I. Standard I/O Efficiency

```
#include "apue.h"
#define BUFFSIZE 4096
int main(void) {
        int n;
        char buf[BUFFSIZE];
        while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
                if (write(STDOUT_FILENO, buf, n) != n)
                        err_sys("write error");
        if (n < 0)
                err_sys("read error");
        exit(0);
}
```

Fig 3.5 - `read()` and `write()`

```
#include "apue.h"
int main(void) {
        int c;
        while ((c = getc(stdin)) != EOF)
                if (putc(c, stdout) == EOF)
                  err_sys("output error");
        if (ferror(stdin))
                err_sys("input error");
        exit(0);
}
```

Fig 5.4 - `getc()` and `putc()`

```
#include "apue.h"
int main(void) {
        char buf[MAXLINE];
        while (fgets(buf, MAXLINE, stdin) != NULL)
                if (fputs(buf, stdout) == EOF)
                        err_sys("output error");
        if (ferror(stdin))
                err_sys("input error");
        exit(0); }
```

Fig 5.5 – `fgets()` and `fputs()`

I. Standard I/O Efficiency

- **Copying a 98.5 MB file from `stdin` to `stdout`**
- **User CPU time dominated by a loop which is executed**
  - 100 million times in the character-at-a-time versions.
  - 3,144,984 times in the line-at-a-time version.
  - 12,611 times in the `read` version (for a buffer size of 8192.)
- **System CPU time is the same as before, because the standard I/O routines performs buffering using the optimal I/O size.**

Function	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Bytes of program text
best time from Figure 3.5	0.01	0.18	6.67	
`fgets, fputs`	2.59	0.19	7.15	139
`getc, putc`	10.84	0.27	12.07	120
`fgetc, fputc`	10.44	0.27	11.42	120
single byte time from Fig3.5	124.89	161.65	288.64	

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

❑ Read or write objects

– Example 1

```
float data[10]
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
   err_sys("fwrite error");
```

– Example 2

```
struct {
   short count;
   long total;
   char name[NAMESIZE];
} item;
if (fwrite(&item, sizeof(item), 1, fp) != 1)
   err_sys("fwrite error");
```

I. Binary I/O

- **`fread` returns the number of objects less than `nobj` on the end of file or error**
- **If `fwrite` returns the value less than the `nobj`, an error has occurred.**
- **Portability problem with binary I/O**
  - The offset of a member within a structure may differ between compilers and systems (i.e. different alignments)
  - Byte ordering for multibyte integers and floating-point values

I. Positioning a Stream

```
#include <stdio.h>
long ftell(FILE *fp);
int fseek(FILE *fp, long offset, int whence);
void rewind(FILE *fp);
```

- `whence`: **SEEK_SET, SEEK_CUR, SEEK_END that are the same as** `lseek`
- **A stream is set to the beginning of the file with** `rewind`**.**

```
#include <stdio.h>
off_t ftello(FILE *fp);
int fseeko(FILE *fp, off_t offset, int whence);
```

- **Introduced in the Single UNIX Specification**
- **The type of the** `offset` **is** `off_t` **instead of** `long`**.**

I. Positioning a Stream

```
#include <stdio.h>

int fgetpos(FILE *fp, fpos_t *pos)

int fsetpos(FILE *fp, const fpos_t *pos);
```

- **New with the ISO C standard**
- `fgetpos` **stores the file's current position indicator in** `pos`**, which can be used in a later call to** `fsetpos` **to reposition the stream.**

I. Formatted I/O

```
#include <stdio.h>
int printf(const char *format, …);
int fprintf(FILE *fp, const char *format, …);
int sprintf(char *buf, const char *format, …);
int snprintf(char *buf, size_t n, const char *format, …);

#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg);
int vfprintf(FILE *fp, const char *format, va_list arg);
int vsprinf(char *buf, const char *format, va_list arg);
int vsnprinf(char *buf, size_t n, const char *format, va_list arg);

#include <stdio.h>
int scanf(const char *format, …);
int fscanf(FILE *fp, const char *format, …);
int sscanf(const char *buf, const char *format, …);

#include <stdarg.h>
#include <stdio.h>
int vscanf(const char *format, va_list arg);
int vfscanf(FILE *fp, const char *format, va_list arg);
int vsscanf(const char *buf, const char *format, va_list arg);
```

I. Implementation Details

```
#include <stdio.h>

int fileno(FILE *fp);
```

- **It returns the file descriptor associated with the stream (if you want to call `dup`, `fcntl`, etc.)**

- **Print buffering for various standard I/O streams (Figure 5.11)**
  - `stdin` and `stdout`: line buffered
  - `stderr`: unbuffered
  - Regular files: fully buffered

I. Print buffering for various standard I/O streams

Fig 5.11

```
#include "apue.h"
void pr_stdio(const char *, FILE *);
int main(void) {
  FILE *fp;
  fputs("enter any character\n", stdout);
  if (getchar() == EOF) err_sys("getchar
error");
  fputs("one line to standard error\n", stderr);
  pr_stdio("stdin", stdin);
  pr_stdio("stdout", stdout);
  pr_stdio("stderr", stderr);

  if ((fp = fopen("/etc/motd", "r")) == NULL)
     err_sys("fopen error");
  if (getc(fp) == EOF) err_sys("getc error");

  pr_stdio("/etc/motd", fp);
  exit(0);
}
```

```
void pr_stdio(const char *name, FILE *fp)
{
  printf("stream = %s, ", name);
  /* The following is nonportable. */
  if (fp->_IO_file_flags & _IO_UNBUFFERED)
     printf("unbuffered");
  else if (fp->_IO_file_flags & _IO_LINE_BUF)
     printf("line buffered");
  else /* if neither of above */
  printf("fully buffered");
  printf(", buffer size = %d\n",
          fp->_IO_buf_end - fp->_IO_buf_base);
}
```

I. Print buffering for various standard I/O streams

Fig 5.11

```
$ ./a.out stdin, stdout, and stderr connected to terminal
enter any character
we type a newline
one line to standard error
stream = stdin, line buffered, buffer size = 1024
stream = stdout, line buffered, buffer size = 1024
stream = stderr, unbuffered, buffer size = 1
stream = /etc/motd, fully buffered, buffer size = 4096
$ ./a.out < /etc/termcap > std.out 2> std.err
$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 4096
stream = stdout, fully buffered, buffer size = 4096
stream = stderr, unbuffered, buffer size = 1
stream = /etc/motd, fully buffered, buffer size = 4096
```

I. Temporary Files

```
#include <stdio.h>
char *tmpnam(char *ptr);
FILE *tmpfile(void);
```

- **`tmpnam` generates a unique file name using the path prefix `P_tmpdir`.**
  - If *ptr* is `NULL`, the generated pathname is stored in an internal static area.
  - else an array of at least `L_tmpnam` is assumed.
- **`tmpfile` creates a temporary file (type `wb+`) that is automatically removed when it is closed or on program termination.**
  - Equivalent to (`tmpnam`, then create the file, and immediately `unlink` it)

Figure 5.12 – Demonstrate `tmpnam` and `tmpfile` functions

```
#include "apue.h"
int main(void) {
  char name[L_tmpnam], line[MAXLINE];
  FILE *fp;
  printf("%s\n", tmpnam(NULL)); /* first temp name */
  tmpnam(name); /* second temp name */
  printf("%s\n", name);

  if ((fp = tmpfile()) == NULL) /* create temp file */
    err_sys("tmpfile error");
  fputs("one line of output\n", fp); /* write to temp file */
  rewind(fp); /* then read it back */
  if (fgets(line, sizeof(line), fp) == NULL)
    err_sys("fgets error");
  fputs(line, stdout); /* print the line we wrote */
  exit(0);
}
```

```
$ ./a.out
/tmp/fileC1Icwc
/tmp/filemSkHSe
one line of output
```

Q and A