

第6章 | 标准I/O库 |

上一章介绍了基于文件描述符的使用底层系统调用的 I/O（输入/输出）操作，本章将介绍基于流的使用 C 标准库函数进行的 I/O 操作。不仅在 Linux，在很多操作系统上都实现了标准 I/O 库，该库由 ANSI C 标准说明。标准 I/O 库是在系统调用函数基础上构造的，它处理很多细节（例如缓存分配）以优化执行 I/O。与基于文件描述符的 I/O 相比，基于流的 I/O 更加简单、方便，也更加高效。因而在 Linux 环境 C 程序的编写中，基于流的 I/O 使用更为广泛。本章内容包括：

- 流和文件指针
- 缓存
- 流的打开和关闭
- 基于字符和行的 I/O
- 二进制 I/O
- 定位流
- 格式化 I/O
- 临时文件
- 文件流和文件描述符

6.1 流和文件指针

在上一章中，所有 I/O 函数都是针对文件描述符的。当打开一个文件时，即返回一个文件描述符，然后该文件描述符就用于后续的 I/O 操作。而对于标准 I/O 库，它们的操作是围绕流(stream)进行的。在 Linux 系统中，文件和设备都可被看作是数据流。这里所谓的数据流指的是无结构的字节序列。当用标准 I/O 库打开或创建一个文件时，即将一个流与一个文件结合起来。

标准 I/O 库提供了函数 `fopen` 用于打开一个流。当打开一个流时，该函数会返回一个指向 FILE 对象的指针，即文件指针（类型为 `FILE*`）。FILE 对象通常是一个结构，它包含了 I/O 库为管理该流所需要的所有信息：用于实际 I/O 的文件描述符，指向流缓存的指针，缓存的长度，当前在缓存中的字符数，出错标志等。但一般应用程序没有必要关心 FILE 结构体的各成员值，使用流时，只需将 FILE 指针作为参数传递给每个标准 I/O 函数。

Linux 对一个进程预定义了三个流：标准输入流、标准输出流和标准错误输出流，它们自动地为进程打开并可用。这三个标准 I/O 流通过在头文件`<stdio.h>`中预定义的文件指针 `stdin`、`stdout` 和 `stderr` 加以引用。它们与上一章所介绍的由文件描述符 `STDIN_FILENO`、`STDOUT_FILENO` 和

STDERR_FILENO 所表示的标准输入、标准输出和标准错误输出相对应。

6.2 缓存

标准 I/O 提供缓存的目的是尽可能减少使用 `read` 和 `write` 调用的次数，以提高 I/O 的效率。标准 I/O 也对每个 I/O 流自动进行缓存管理，免除了由应用程序考虑这一点所带来的麻烦。

标准 I/O 提供了三种类型的缓存。

1. 全缓存

使用全缓存时，只有当标准 I/O 缓存填满后才进行实际 I/O 操作。对磁盘文件的标准 I/O 操作一般是实施全缓存的。在一个流上执行第一次 I/O 操作时，相关标准 I/O 函数通常调用 `malloc` 函数分配所需使用的缓存。

术语刷新（`flush`）用于说明标准 I/O 缓存的写操作。缓存可由标准 I/O 例程自动刷新（例如当填满一个缓存时），或者可以调用函数 `fflush` 显式刷新一个流。

2. 行缓存

使用行缓存时，标准 I/O 库会在输入和输出中遇到换行符时执行实际 I/O 操作。当流涉及一个终端时（例如标准输入和标准输出），典型地使用行缓存。

对于行缓存有两个限制。第一，因为行的缓存的长度是固定的，所以只要填满了缓存，即使还没有写一个换行符，也会进行 I/O 操作。第二，任何时候只要通过标准 I/O 库要求从一个不带缓存的流或一个行缓存的流（它预先要求从内核得到数据，所需的数据可能已在该缓存中）得到输入数据，那么就会造成刷新所有行缓存输出流。

3. 不带缓存

即不对字符进行缓存。如果用标准 I/O 函数写若干字符到不带缓存的流中，则相当于用 `write` 系统调用函数将这些字符写至相关联的打开文件上。标准错误输出流 `stderr` 通常是不带缓存的，这就使得出错信息可以尽快显示出来，而不管它们是否含有一个新行字符。

ANSI C 规定了下列缓存特征。

- 当且仅当标准输入和标准输出并不涉及交互作用设备时，它们才是全缓存的。
- 标准错误输出绝对不会是全缓存的。

6.3 流的打开和关闭

标准 I/O 库提供了 `fopen` 系列函数用以创建或打开流文件，提供了 `fclose` 函数关闭已打开的流文件。

6.3.1 打开流

使用 `fopen` 系列函数可以创建或打开流文件，这些函数的原型如下。

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
```

```
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

打开一个文件后，即在文件指针（FILE *）和文件之间建立了一个关联。这三个函数的区别如下。

(1) fopen 打开路径名由 pathname 指定的一个文件。

(2) freopen 在由 stream 指定的流上打开一个指定的文件（其路径名由 pathname 指定），如若该流已经打开，则先关闭该流。此函数一般用于将一个指定的文件打开为一个预定义的标准流：标准输入、标准输出或标准错误输出。

(3) fdopen 取一个现存的文件描述符（可通过底层系统调用 open、dup、dup2、fcntl 或 pipe 等函数得到），并使一个标准的 I/O 流与该描述符相结合。此函数常用于由创建管道和网络通信通道函数获得的描述符。因为这些特殊类型的文件不能用标准 I/O 函数 fopen 打开，而必须先调用设备专用函数以获得一个文件描述符，然后用 fdopen 使一个标准 I/O 流与该描述符相结合。fdopen 函数不是 ANSI C 的标准函数，而是属于 POSIX.1 的标准。

这三个函数的各参数和返回值的含义如下。

1. path 要打开或创建的文件的名字

2. mode 对该 I/O 流的读、写方式，ANSI C 规定了 15 种不同的可能值

- r 或 rb 以读方式打开
- w 或 wb 以写方式打开或创建，并将文件长度截为 0
- a 或 ab 以写方式打开，新内容追加在文件尾
- r+ 或 r+b 或 rb+ 以更新方式打开（读和写）
- w+ 或 w+b 或 wb+ 以更新方式打开，并将文件长度截为 0
- a+ 或 a+b 或 ab+ 以更新方式打开，新内容追加在文件尾

注意：① 字符 b 的作用是区分文本文件和二进制文件。但 Linux 内核并不对这两种文件进行区分，所以在 Linux 系统环境下字符 b 作为 mode 的一部分实际上并无作用。② 对于 fdopen，因为该描述符已被打开，即所引用的文件必已存在，所以 fdopen 以写方式打开并不会创建该文件或截短该文件。

3. fd 待关联的底层文件描述符

4. stream 待关联的流文件指针，若该流已打开则会被先关闭

5. 返回值

- 成功时返回流文件指针
- 失败时返回 NULL

能否成功打开文件流受打开方式的限制，如表 6-1 所示。另外，进程可打开的文件流的数量有一个上限，该上限值由 stdio.h 中定义的 FOPEN_MAX 常量指定。

表 6-1 打开一个标准 I/O 流的六种不同的方式的限制

限 制	r	w	a	r+	w+	a+
文件必须已存在	✓			✓		
擦除文件以前的内容		✓			✓	
流可以读	✓			✓		
流可以写		✓	✓	✓	✓	✓
流只可在尾端处写			✓			✓

在指定 w 或 a 类型创建一个新文件时，无法指定该文件的存取许可权位。POSIX.1 要求以这种方式创建的文件具有下列存取许可权。

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

除非流引用终端设备，否则系统默认它被打开时是全缓存的。若流引用终端设备，则该流是行缓存的。

6.3.2 关闭流

在使用完流文件后，应调用 fclose 函数关闭流。fclose 函数的原型如下所示。

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

fclose 唯一的参数 fp 就是由 fopen 系列函数返回的流文件指针。成功时函数返回 0，失败返回 EOF(-1)。

在文件流被关闭之前，fclose 会刷新缓存中的输出数据，但缓存中的输入数据被丢弃。如果标准 I/O 库已经为该流自动分配了一个缓存，则释放此缓存。

当一个进程正常终止时（直接调用 exit 函数，或从 main 函数返回），则所有带未写缓存数据的标准 I/O 流都被刷新，所有打开的标准 I/O 流都被关闭。

打开和关闭流的一般用法如下。

```
FILE *fp = NULL; /* 定义文件指针 */
fp = fopen("filename", "r"); /* 打开文件 */
if(fp == NULL) /* 判断是否成功打开 */
{
    printf("Cannot open file.\n");
    exit(-1); /* 打开失败则退出 */
}
..... /* 读写文件 */
fclose(fp); /* 关闭文件 */
```

6.4 基于字符和行的 I/O

在打开一个流以后，可采用三种不同类型的非格式化 I/O 对其进行读、写操作。

- (1) 每次一个字符的 I/O。
 - (2) 每次一行的 I/O。以换行符标示一行的终止。
 - (3) 二进制 I/O。每次 I/O 操作读或写一定数量的对象，而每个对象具有指定的长度。
- 本节先介绍前两类 I/O。

6.4.1 字符 I/O

6.4.1.1 读字符

使用以下三个函数可一次读取一个字符。

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

```
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
```

函数 `getchar` 等同于 `getc(stdin)`。前两个函数的区别是 `getc` 可被实现为宏，而 `fgetc` 则一定是一个函数。这意味着调用 `fgetc` 所需时间很可能长于调用 `getc`。这些函数的每一次调用会使当前读写位置向文件尾部移动一个字符，即每次读取的是上次读取的位置之后的字符。

参数 `stream` 表示已打开并准备从其中读取数据的流。成功时，三个函数均返回所读取到的字符，出错或已到达文件尾端时返回 `EOF`。

应当注意，这三个函数以 `unsignedchar` 类型转换为 `int` 的方式返回下一个字符。说明为不带符号的理由是，即使所读字节的最高位为 1 也不会返回负数。要求整型返回值的理由是，这样就可以返回已发生错误或已到达文件尾端的指示值 `EOF`（其值一般是 -1）。

在读一个输入流时，经常会用到回送字符操作。回送字符操作通常用于需要根据下一个字符的值来决定如何处理当前字符的情况。处理这种情况时，需要在读出下一个字符以后，能将其送回流缓冲区中，以便下一次输入时再返回该字符。标准 I/O 库提供了 `ungetc` 函数以支持字符回送操作。该函数的原型如下。

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

参数 `c` 是要送回流中的字符，`stream` 是所操作的流。成功时返回 `c`，失败时返回 `EOF`。

送回到流中的字符以后又可从流中读出，但读出字符的顺序与送回的顺序相反。回送的字符，不一定必须是上一次读到的字符，但是不能回送 `EOF`。

当已经到达文件尾端时，仍可以回送一个字符。下次读将返回该字符，再次读才返回 `EOF`。之所以能这样做的原因是一次成功的 `ungetc` 调用会清除该流的文件结束指示。

6.4.1.2 判断流结束或出错

在大多数 `FILE` 对象的实现中都为每个流保持了两个标志：文件结束标志和出错标志。`feof` 和 `ferror` 函数分别根据这两个标志来判断流是否结束或流是否出错。因为到达文件尾端和出错时，三个字符读取函数的返回值都是 `EOF`，所以应当通过调用 `feof` 或 `ferror` 函数区分这两种情况。这两个函数的原型如下。

```
#include <stdio.h>
int feof(FILE *stream);
int ferror(FILE *stream);
```

参数 `stream` 为要判断是否已到文件尾或出错的流。当流已到文件尾时，`feof` 返回非 0（真），否则返回 0（假）。当流出错时，`ferror` 返回非 0（真），否则返回 0（假）。

调用 `clearerr` 函数可以清除 `FILE` 对象中的文件结束标志和出错标志。该函数原型如下。

```
#include <stdio.h>
void clearerr(FILE *stream);
```

参数 `stream` 为要清除标志的流，函数不带返回值。

6.4.1.3 写字符

使用以下三个函数可一次写入一个字符。

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

```
int putc(int c, FILE *stream);
int putchar(int c);
```

与字符输入函数一样, `putchar(c)`等同于`putc(c,stdout)`。`putc`可被实现为宏, 而`fputc`则一定是函数。这些函数的每一次调用会使当前读写位置向文件尾部移动一个字符, 即每次写入的位置是上次写入的位置之后的一个字节。

参数`stream`表示已打开并准备往其中写入数据的流。成功时, 三个函数均返回所写入的字符, 出错时返回`EOF`。

程序清单 6-1 给出了一个字符 I/O 的示例。该示例程序反复从标准输入读取字符并将其保存于一个文本文件, 直到用户按下`Ctrl+D`键为止。读取完用户输入后, 程序最后读取并显示所保存的文件的内容。

程序清单 6-1 ex_io_char.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main( )
5 {
6     FILE *fp;
7     int c;
8     char* filename = "1.txt";
9
10    if((fp = fopen(filename,"w")) == NULL)
11    {
12        printf("Can't open %s for writing.\n", filename);
13        exit(-1);
14    }
15    while ((c = getchar()) != EOF) /* 按 Ctrl-D 键产生 EOF 值 */
16        fputc(c, fp);
17    fclose(fp);
18
19    printf("Output file content:\n");
20    if((fp = fopen(filename,"r")) == NULL)
21    {
22        printf("Can't open %s\n for reading.", filename);
23        exit(-1);
24    }
25    while ((c = fgetc(fp)) != EOF)
26        putchar(c);
27    fclose(fp);
28
29    return 0;
30 }
```

程序第 10 行以写的方式在当前目录创建并打开“1.txt”文件, 第 15~16 行循环调用`getchar()`从标准输入读取字符, 并调用`fputc`将该字符写入到文件中。第 17 行关闭流。

程序第 20 行以读的方式打开当前目录下的“1.txt”文件, 第 25~26 行循环调用`fgetc()`从文件中读取字符, 并调用`putchar`将该字符输出到标准输出设备。第 27 行关闭流。

在命令行编译并运行该程序, 结果如下。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_io_char ex_io_char.c
```

```
jianglinmei@ubuntu:~/c$ ./ex_io_char
This is an example for character I/O
Good, over! [注: 在此回车后按下 Ctrl+D 终止输入]
Output file content:
This is an example for character I/O
Good, over!
jianglinmei@ubuntu:~/c$ ls 1.txt
1.txt
```

6.4.2 行 I/O

6.4.2.1 读行

使用以下两个函数可从流中一次读取一行文本。

```
#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
```

这两个函数的各参数和返回值的含义如下：

1. s 输入缓存
2. size 输入缓存大小
3. stream 流文件指针
4. 返回值
 - 成功时返回指向缓存的指针
 - 失败或处于文件尾端时返回 NULL，同时设置 errno

函数 fgets 从指定的流读，而 gets 从标准输入（即 stdin）读。每调用一次 fgets 会使当前读写位置向文件尾部移动所读取到的字符数，以保证每次读取的是上次读取之后的位置。

对于 fgets，必须指定缓存的长度 size。此函数一直读到下一个换行符 ('\n') 为止，但是不超过 n-1 个字符，读入的字符被送入缓存。该缓存总是以 null 字符结尾。如果该行（包括最后一个换行符）的字符数超过 n-1，则只返回一个不完整的行，而且缓存仍以 null 字符结尾。对 fgets 的下一次调用会继续读取该行剩余的字符。

gets 是一个不推荐使用的函数。问题是调用者在使用 gets 时不能指定缓存的长度。这样就可能造成缓存越界（若该行长于缓存长度），从而产生不可预料的后果。这种缺陷曾被利用，造成 1988 年的因特网蠕虫事件。

另外，fgets 会将读取到的换行符送入缓存，而 gets 并不保存换行符。

6.4.2.2 写行

使用以下两个函数可向流中一次写入一行文本。

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

这两个函数的各参数和返回值的含义如下。

1. s 待输出的字符串，应以 null 终止
2. stream 流文件指针
3. 返回值
 - 成功时返回非负数

- 失败时返回 EOF(-1)，并设置 errno

函数 fgets 从指定的流读，而 gets 从标准输入（即 stdin）读。每调用一次 fgets 会使当前读写位置向文件尾部移动所读取到的字符数，以保证每次读取的是上次读取之后的位置。

函数 fputs 将一个以 null 符终止的字符串写到指定的流，终止符 null 本身不写出。通常，在 null 符之前是一个换行符，但并不要求在 null 符之前一定是换行符。所以，fputs 并不一定每次输出一行。

函数 puts 将一个以 null 符终止的字符串写到标准输出，终止符本身不写出。但是，puts 在输出指定字符串后又附加地将一个换行符写到标准输出。

puts 并不像它所对应的 gets 那样不安全。但还是应避免使用它，以免需要记住它在最后又加上了一个换行符。如果总是使用 fgets 和 fputs，那么就会熟知在每行终止处必须自己加一个换行符。

程序清单 6-2 给出了一个行 I/O 的示例。该示例程序反复从标准输入读取行并将其保存于一个文本文件，直到用户输入 quit 为止。读取完用户输入后，程序最后读取并显示所保存的文件的内容。

程序清单 6-2 ex_io_line.c

```

1 #include "stdio.h"
2 #include "string.h"
3 #include "stdlib.h"
4 #define BUFSIZE    80
5
6 int main()
7 {
8     FILE *fp;
9     char filename[BUFSIZE], str[BUFSIZE];
10    printf("Please input filename: ");
11    gets(filename); /* 可用 fgets(filename, BUFSIZE, stdin); 代替 */
12
13    if ((fp = fopen(filename, "w")) == NULL)
14    {
15        printf("Can't open file [%s] for writing.\n", filename);
16        exit(-1);
17    }
18
19    printf("Please input filename content:\n");
20    /* 输入 quit 则结束 */
21    while (strcmp(gets(str), "quit") != 0)
22    {
23        fputs(str, fp);
24        fputc('\n', fp);
25    }
26    fclose(fp);
27
28    if ((fp = fopen(filename, "r")) == NULL)
29    {
30        printf("Can't open file [%s] for reading.\n", filename);
31        exit(-1);
32    }
33

```

```

34     printf("\nFilename content:\n");
35     /* 从文件读取字符串，返回 NULL 表示已读完 */
36     while ((fgets(str, BUFSIZE, fp)) != NULL)
37         puts(str); /* 可试用 fputs(str, stdout); 代替 */
38     fclose(fp);
39
40     return 0;
41 }

```

程序第 11 行通过 gets() 函数读取用户输入的文件名，然后在第 13 行以写的方式打开该文件，第 21 ~ 25 行循环调用 gets() 从标准输入读取一行字符串，并调用 fputs 将该字符串写入到文件中。因为 gets() 读入的串不包行换行符，所以第 24 行调 fputc() 函数向文件补充写入一个换行符。程序第 26 行关闭文件输出流。

程序第 28 行以读的方式打开上面写入了数据的文件，第 36 ~ 37 行循环调用 fgets() 从文件中读取行，并调用 puts 将读取到的行输出到标准输出设备。第 38 行关闭文件输入流。

在命令行编译并运行该程序，结果如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_io_line ex_io_line.c
jianglinmei@ubuntu:~/c$ ./ex_io_line
Please input filename: 1.txt
Please input filename content:
This is the first line.
This is the second line.
quit [注：输入 quit 退出]

Filename content:
This is the first line.

This is the second line.

```

注意，因为 puts 会在输出结果末尾自动输出换行符，所以输出的文件内容行末都多了一个空行。读者可试用 fputs(str, stdout) 替代 puts(str)，以查看结果有何不同。

6.5 二进制 I/O

二进制 I/O 也称直接 I/O、一次一个对象 I/O、面向记录的 I/O 或面向结构的 I/O。每次 I/O 操作读或写一定数量的对象，而每个对象具有指定的长度。常用于从二进制文件中读或向二进制文件中写一个结构。

6.5.1 读二进制流

使用 fread 函数可以进行二进制数据块的输入。函数原型如下。

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

函数 fread 的各参数和返回值的含义如下。

1. ptr 输入缓存

2. size 数据块的大小
3. nitems 数据块的个数
4. stream 流文件指针
5. 返回值

返回实际读取到的数据块的个数。如果此数字小于 nitems，则表示出错或到达文件尾端，应调用 ferror 或 feof 以判断究竟是哪一种情况

这里出现的 size_t 类型实际上是对无符号整型或无符号长整型的 typedef。

6.5.2 写二进制流

使用 fwrite 函数可以进行二进制数据块的输出。函数原型如下。

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

函数 fwrite 的各参数和返回值的含义如下。

1. ptr 待输出数据的首地址
2. size 数据块的大小
3. nitems 数据块的个数
4. stream 流文件指针
5. 返回值

- 返回实际输出的数据块的个数。如果此数字小于 nitems，则表示出错

6.5.3 二进制 I/O 的常见用法

二进制 I/O 有三种常见的用法。

(1) 读或写一个二进制数组。此时，指定 size 为每个数组元素的长度，nitems 为欲写的元素个数。例如：

```
float data[10];
..... /* 设置 data 各元素的值 */
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    printf("fwrite error");
```

这段代码调用 fwrite 向文件流写入 data 数组从第 3 个(下标为 2)元素起的连续 4 个元素(数据块个数为 4)，每个数据块的大小为一个元素(一个浮点数)的大小，如果写入的元素个数不为 4，则说明发生了错误。

(2) 读或写一个结构。此时，指定 size 为结构体的大小，nitems 为 1。例如：

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

这段代码调用 fwrite 向文件流写入结构体变量 item，数据块个数为 1，返回值如果不为 1 则

说明发生了错误。

(3) 以上二者的结合。例如：

```
struct {
    short count;
    long total;
    char name[NAMESIZE]
} item[10];
..... /* 设置 data 各元素的值 */
if (fwrite(item, sizeof(item), 10, fp) != 10)
    printf("fwrite error");
```

这段代码调用 fwrite 向文件流写入结构体类型的数组，每个数据块的大小为数组元素（一个结构体）的大小，数据块个数为 10，返回值如果不为 10 则说明发生了错误。

程序清单 6-3 给出了一个二进制 I/O 的示例。该示例程序实现了一个简单的文件复制功能。

程序清单 6-3 ex_io_copy.c

```
1 #include "stdio.h"
2 #include "stdlib.h"
3 #define BUFSIZE     80
4
5 int main(int argc, char* argv[])
6 {
7     FILE *fpIn, *fpOut;
8     int nRead, nWrite;
9     unsigned char buf[BUFSIZE];
10
11    if(argc != 3)
12    {
13        printf("Usage: %s <source_file> <destination_file>\n", argv[0]);
14        return -1;
15    }
16
17    if ((fpIn = fopen(argv[1], "r")) == NULL)
18    {
19        printf("Can't open file [%s] for reading.\n", argv[1]);
20        return -1;
21    }
22
23    if ((fpOut = fopen(argv[2], "w")) == NULL)
24    {
25        printf("Can't open file [%s] for writing.\n", argv[2]);
26        fclose(fpIn);
27        return -1;
28    }
29
30    while (!feof(fpIn))
31    {
32        nRead = fread(buf, sizeof(unsigned char), BUFSIZE, fpIn);
33        if(nRead < BUFSIZE)
34        {
35            if(ferror(fpIn))
36            {
37                printf("Copy failed.\n");
38            }
39        }
40    }
41
42    if(fclose(fpOut) != 0)
43    {
44        printf("Close failed.\n");
45    }
46}
```

```

38         break;
39     }
40 }
41
42 if(nRead > 0)
43 {
44     nWrite = fwrite(buf, sizeof(unsigned char), nRead, fpOut);
45     if(nWrite != nRead)
46     {
47         printf("Copy failed.\n");
48         break;
49     }
50 }
51 }
52
53 fclose(fpOut);
54 fclose(fpIn);
55
56 return 0;
55 }

```

程序第 11 ~ 15 行判断用户提供的命令行参数的合法性，参数个数不为 3 则提示正确的使用方法。

程序第 17 行以读的方式打开源文件，第 23 行以写的方式打开目标文件。

程序第 30 ~ 51 行为一个 while 循环，反复读取源文件的内容并将读取到的内容写入目标文件，这个循环是程序实现文件复制的主体部分。第 32 行调用 fread() 函数从源文件读取 BUFSIZE 个字节，第 33 ~ 40 行判断读取到的字节数是否少于 BUFSIZE。如果少于 BUFSIZE 则继续调用 perror() 函数判断流是否出错，如果出错则输出提示并退出循环。如果 fread() 函数读取到的字节数大于 0，则在第 44 行调用 fwrite() 函数将读取到的数据写入目标文件。第 45 ~ 49 行判断写入的字节数是否等于读取到的字节数。如果不等则说明写入出错，程序输出提示并退出循环。

在命令行编译并运行该程序，命令如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_io_copy ex_io_copy.c
jianglinmei@ubuntu:~/c$ ./ex_io_copy a.txt a.txt.bak

```

6.6 定位流

可以调用 fseek 定位流文件的读写指针。该函数的原型如下。

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

函数 fseek 的各参数和返回值的含义如下。

1. stream 流文件指针
2. offset 位移量
3. whence 指定位移量相对于何处开始。whence 可以取如下三个常量。
 - SEEK_SET (值为 0) 文件开始位置
 - SEEK_CUR (值为 1) 文件读写指针当前位置

- SEEK_END (值为 2) 文件结束位置

4. 返回值

- 若成功返回 0
- 若出错返回 -1 , 错误值记录在 errno

一次成功的 `fseek()` 调用会清除流结束标志，并会撤销已调用的 `ungetc()` 对流的影响。

调用函数 `rewind()` 可以将一个流的读写指针设置到文件的起始位置。其原型如下。

```
#include <stdio.h>
void rewind(FILE *stream);
```

函数 `rewind` 的唯一参数是已打开的流文件指针。调用 `rewind(fp)` 基本等同于调用 `fseek(stream, 0L, SEEK_SET)`。稍微不同的是，`rewind` 函数在将读写指针设置到文件的起始位置的同时会将错误指示器 `errno` 清 0。

调用 `ftell()` 函数可以获得一个流的读写指针的当前位置。该函数的原型如下。

```
#include <stdio.h>
long ftell(FILE *stream);
```

函数 `ftell` 的唯一参数是已打开的流文件指针。若成功，该函数返回读写指针当前相对于文件起始位置的位移量；若出错，则返回 -1，错误值记录在 `errno`。

6.7 格式化 I/O

格式化 I/O 的作用是：从输入流读取字符串并以指定的格式转换成内存中的二进制数据，或者将内存中的二进制数据以指定的格式转换成字符串并将转换后的字符串写入输出流。学习格式化 I/O 的重点在于掌握格式化控制串的用法。

6.7.1 格式化输出

可用 `printf` 系列函数进行格式化输出处理，它们的原型如下。

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

格式化输出各函数的各参数和返回值的含义如下。

1. stream 流文件指针
2. format 格式输出控制串，下文将详细介绍
3. .. 输出表项（项数根据 format 中的转换控制符可变）
4. str 字符串缓冲区
5. size 字符串缓冲区的大小
6. 返回值
 - 若成功，返回格式化后的字符数（不包含'\0'）
 - 若出错返回一个负数

函数 printf 将格式化数据写到标准输出，fprintf 写至指定的流，sprintf 和 snprintf 将格式化的字符送入 str 缓冲区中。sprintf 在该数组的尾端自动加一个 null 字节，但该字节不包括在返回值中。

snprintf 最多将 size 个字节（含'\0'）写入 str 缓冲区中，并且保证输出的最后一个字节是 null。snprintf 的返回值若大于或等于 size，则说明格式化的结果在输出时被截断了。例如：

```
char buf[5];
/* 以下语句执行后, n 的值为 7, buf = {'a', 'b', 'c', 'd', '\0'} */
int n = snprintf(buf, 5, "abcdefg");
```

由于 sprintf 不能指定缓冲区的大小，很容易造成缓冲区溢出，因此应优先使用 snprintf。在格式化控制串中包含两类信息。

1. 转换控制符：%[修饰符]格式字符 —— 指定输出格式
2. 普通字符或转义字符 —— 原样输出

例如：

```
printf("%c => %d\n", 65, 65); /* 结果为: A => 65 */
```

这里，“%c”和“%d”是转换控制符，“=>”是普通字符，“\n”是转义字符。

应当注意，在调用格式化输出函数时，格式字符的个数应等于输出表项的个数。

常用的格式字符如表 6-2 所示。

表 6-2 常用的格式字符

格式字符	说 明
c	输出一个字符
d 或 i	输出带符号的十进制整数（正数不输出符号）
o	以八进制无符号形式输出整数（不输出前导符 0）
x 或 X	以十六进制无符号形式输出整数（不输出前导符 0x 或 0X）。对于 0x 用 abcdef 输出；对于 0X，用 ABCDEF 输出
u	按无符号的十进制形式输出整数
f	以[-]mmm.ddd 带小数点的形式输出单精度和双精度数，d 的个数由精度指定。隐含的精度为 6，若指定的精度为 0，小数部分（包括小数点）都不输出
e 或 E	以[-]m.ddddde ± xx 或[-]m.dddddE ± xx 的形式输出单精度和双精度数。d 的个数由精度指定，隐含的精度为 6，若指定的精度为 0，小数部分（包括小数点）都不输出。用 E 时，指数以大写“E”表示
g 或 G	由系统决定采用%f 格式还是采用%e 格式，以使输出宽度最小。不输出无意义的 0。用 G 时，若以指数形式输出，则指数以大写表示
s	输出字符串中的字符，直到遇到'\0'，或者输出由精度指定的字符数

可用的格式化修饰符如表 6-3 所示。

表 6-3 格式化输出修饰符

格式字符	说 明
m	宽度修饰（一个 10 进制数）用以指定数据的输出域宽（占几个字符）。如果指定宽度小于数据需要的实际宽度，则数据左边补空格（默认右对齐），补够指定的宽度

续表

格式字符	说 明
.n	精度修饰(点号后跟一个10进制数)。对浮点数以%f或%e格式输出,指定小数点后的位数(四舍五入);对浮点数以%g格式输出,指定尾数部分的有效位数(四舍五入);对字符串,指定最多输出几个字符
-	在输出域宽内左对齐
+	在有符号正数前输出“+”号
0	输出数值时,若输出域宽不足,以0而不是空格填充
#	在八进制数前输出前导的0,在十六进制数前输出前导的0x或0X
l	用在d、i、o、x、u前,指定输出精度为long型。用在f、e、g前,指定输出精度为double型

下面以一些示例来说明格式化输出的具体使用方法。

1. 例 1

```
float f=123.456;
char ch= 'a';
printf("%f,%8f,%8.1f,%2f,%2e\n",f,f,f,f,f);
printf("%3c\n",ch);
```

输出结果为(这里为清晰起见,以“□”代替空格):

```
123.456001,123.456001,□□□123.5,123.46,1.2e+02
□□a
```

2. 例 2

```
static char a[]="Hello,world!"
printf("%s\n%15s\n%10.5s\n%2.5s\n%3s\n",a,a,a,a,a);
```

输出结果为:

```
Hello,world!
□□□Hello,world!
□□□□□Hello
Hello
Hel
```

3. 例 3

```
float f=123.456;
static char c[]="Hello,world!";
printf("%10.2f,%-10.1f\n",f,f);
printf("%10.5s,%-10.3s\n",c,c);
```

输出结果为:

```
□□□□123.46,123.5□□□□□
□□□□□Hello,Hel□□□□□□□
```

4. 例 4

```
/* 假设在16位机上, int为两个字节, long为四个字节 */
```

```
long a = 65536;
printf("%d, %8ld\n", a, a);
```

输出结果为：

0, 00000065536

5. 例5

```
int a=1234;
float f=123.456;
printf("%010.2f\n",f);
printf("%0+8d\n",a);
printf("%0+10.2f\n",f);
```

输出结果为：

```
0000123.46
+0001234
+000123.46
```

6. 例6

```
int a=123;
printf("%o, %#o, %X, %#X\n",a,a,a,a);
```

输出结果为：

173, 0173, 7B, 0X7B

6.7.2 格式化输入

可用 scanf 系列函数进行格式化输出处理，它们的原型如下。

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

格式化输入各函数的各参数和返回值的含义如下。

1. stream 流文件指针
2. format 格式输入控制串，下文将详细介绍
3. .. 输入地址表项（项数根据 format 中的转换控制符可变）
4. str 字符串缓冲区
5. 返回值
 - 若成功，返回实际得到输入的项数
 - 若出错返回 EOF，错误值记录在 errno

函数 scanf 将从标准输入读取格式化数据，fscanf 从指定的流读，sscanf 从 str 缓冲区中读取格式化数据。这里要注意和格式化输出函数不同的一点，以“...”表示的输入地址表项，应当指定内存的地址，如变量地址、数组名等。

和格式化输出函数一样，在格式化输入函数的格式化控制串中也包含两类信息。

1. 转换控制符：%[修饰符]格式字符 —— 指定输入格式

2. 普通字符或转义字符

—— 需原样输入，但不被赋值

例如：

```
char a; int b;
sscanf("A => 65", "%c => %d", &a, &b);
/* 结果 a 被赋值为'A'，b 被赋值为 65 */
```

这里，“%c” 和 “%d” 是转换控制符，“=>”是普通字符。

格式化输入函数常用的格式字符同格式化输出，如表 6-2 所示。可用的格式化输入修饰符如表 6-4 所示。

表 6-4 格式化输入修饰符

格式字符	说 明
h	用于 d、i、o、x、u 前，指定输入为 short 型或 unsigned short 型
l	用于 d、i、o、x、u 前，指定输入为 long 型或 unsigned long 型。用于 e、f、g 前，指定输入为 double 型
m	批定输入域宽（最大字符数）
*	抑制符，指定输入项读入后不赋值给变量

应当注意，在调用格式化输入函数时，格式字符的个数减去抑制符的个数应等于输入地址表项的个数。

下面的示例说明了格式化输入的具体使用方法。

```
scanf("%4d%2d%2d", &yy, &mm, &dd);
/* 输入:20031015，则 yy = 2003, mm = 10, dd = 15 */
scanf( "%3d%*4d%f" ,&k,&f);
/* 输入: 12345678765.43，则 k = 123, f = 8765.43 */
scanf("%3c%2c",&c1,&c2);
/* 输入: abcde，则 c1 = 'a', c2 = 'd' */
scanf( "%c%c%c" ,&c1,&c2,&c3);
/* 输入: a□b□c，则 c1 = 'a', c2 = '□', c3 = 'b' */
scanf("%d%c%f",&a,&b,&c);
/* 输入 1234a123o.26，则 a = 1234, b = 'a', c = 123 */
```

6.8 临时文件

标准 I/O 库提供了两个函数以帮助创建临时文件，它们的原型如下。

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

函数 tmpnam 产生一个与现在文件名不同的一个有效路径名字符串。每次调用它时，它都产生一个不同的路径名，最多调用次数是 TMP_MAX。TMP_MAX 定义在<stdio.h>中。

如果 s 为 NULL，则所产生的路径名存放在一个静态区中，指向该静态区的指针作为函数值

返回。下一次再调用 tmpnam 时，会重写该静态区。

如果 s 不是 NULL，则认为它指向长度至少是 L_tmpnam 个字符的数组（常数 L_tmpnam 定义在头文件<stdio.h>中）。所产生的路径名存放在该数组中，s 也作为函数值返回。

tmpfile 创建并打开一个临时的二进制文件流（类型 wb+），在关闭该文件流或程序结束时将自动删除这个文件。tmpfile 函数的一般实现是先调用 tmpnam 产生一个唯一的路径名，然后在使用完后用 unlink 函数删除。

程序清单 6-4 说明了这两个函数的应用。

程序清单 6-4 ex_io_tmp.c

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char    name[L_tmpnam], line[BUFSIZ];
6     FILE   *fp;
7
8     printf("%s\n", tmpnam(NULL));
9
10    tmpnam(name);
11    printf("%s\n", name);
12
13    if( (fp = tmpfile()) == NULL )
14    {
15        printf("Fail to create temp file.\n");
16        return -1;
17    }
18
19    fputs("one line of output\n", fp);
20    rewind(fp);
21    if( fgets(line, sizeof(line), fp) == NULL )
22    {
23        printf("Fail to read file.\n");
24        fclose(fp);
25        return -1;
26    }
27    fputs(line, stdout);
28
29    return 0;
30 }
```

程序第 8 行创建了第一个临时文件，并将文件名输出。

程序第 10 行创建了第二个临时文件，在第 11 行将产生的文件名输出。

程序第 13 行创建并打开了一个临时文件。第 19 行向该临时文件写入一行文本，第 20 行反卷文件读写指针到文件头，然后在第 21 行将前面写入的文本读出保存于 line 数组。第 27 行输出从临时文件读出的文本。

在命令行编译并运行该程序，命令如下。

```

jianglinmei@ubuntu:~/c$ vi ex_io_tmp.c
jianglinmei@ubuntu:~/c$ gcc -o ex_io_tmp ex_io_tmp.c
jianglinmei@ubuntu:~/c$ ./ex_io_tmp
/tmp/filerPJeuP
```

```
/tmp/fileFgsiUF
one line of output
```

6.9 文件流和文件描述符

每个文件流都和一个底层文件描述符相关联，可以通过调用 `fileno` 函数获得相关联的文件描述符。函数 `fileno` 的原型如下。

```
#include <stdio.h>
int fileno(FILE *stream);
```

函数 `fileno` 的唯一参数是已打开的文件流指针。函数执行成功时返回一个非负的文件描述符；失败时返回-1，并设置 `errno` 指示具体错误。

另外，如 6.3.1 小节所述，可以通过 `fdopen` 函数获取一个和已打开的文件描述符相关联的流文件指针。

6.10 小结

本章首先介绍了标准 I/O 库中流和文件指针的概念，以及流和底层文件描述符的关系。随后介绍了流操作时缓存的作用和分类。本章的主体部分是流的具体操作，包括流的打开、关闭、读、写和定位。流的读写又分基于字符和行的 I/O、二进制 I/O 和格式化 I/O，对这些类型的标准库 I/O 本章均以典型的示例给予了说明。本章最后介绍了临时文件的操作方法，以及文件流和文件描述符相互关联的方法。

6.11 习题

- (1) 请简述什么是文件指针，它和文件描述符有何关系。
- (2) 标准 I/O 提供了哪些类型的缓存？它们有何区别。
- (3) 请编写一个程序，使用标准 I/O 函数在用户的主目录下创建一个文件“rect.txt”。然后用基于字符的 I/O 函数向该文件写入一个 $n \times n$ 的由 1 ~ n^2 之间的数字组成的方形，如下所示。

```
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
```

文件写入成功之后，再用基于字符的 I/O 函数读取文件内容，并将内容输出到屏幕。

- (4) 请编写程序，分别用基于行的 I/O 函数和格式化 I/O 函数代替基于字符的 I/O 函数实现第 3 题的要求。
- (5) 程序清单 6-5 所示程序用于维护学生信息，学生信息在内存中用一个链表表示。请补充

完成函数 toDate、insert、save、load 和 release，各函数的功能参见注释。

程序清单 6-5 list_student.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define TRUE      1
#define FALSE     0
#define MAX_ID_LEN    10
#define MAX_NAME_LEN   30
#define FILE_NAME    "data"

typedef int BOOL;

typedef struct _DATE{
    int year;           // 年
    int month;          // 月
    int day;            // 日
} DATE_T;

typedef struct _STUDENT{
    char id[MAX_ID_LEN]; // 学号
    char name[MAX_NAME_LEN]; // 姓名
    DATE_T birthday; // 生日
    int height; // 身高
    struct _STUDENT * next; // 链表指针
} STUDENT_T;

// 字符串转成日期，成功返回 TRUE，否则返回 FALSE
BOOL toDate(DATE_T * date, char * szBuf)
{
}

// 在链表中插入一个节点，保持学号从小到大的顺序
void insert(STUDENT_T ** head, STUDENT_T * data)
{
}

// 保存学生信息到宏 FILE_NAME 定义的文件中
void save(STUDENT_T * head)
{
}

// 从宏 FILE_NAME 定义的文件中读取学生信息
void load(STUDENT_T ** head)
{}
```

```

// 输出学生信息
void output(STUDENT_T * head)
{
}

// 释放学生信息链表
void release(STUDENT_T * head)
{
}

// 创建学生信息链表
STUDENT_T * create()
{
    STUDENT_T *head = NULL, *p = NULL;
    char szBuf[1024];

    while(1)
    {
        p = (STUDENT_T *)malloc(sizeof(STUDENT_T));
        if(!p)
        {
            printf("Memory overflow!\n");
            break;
        }

        printf("Please input id (0 for exit): ");
        scanf("%s", p->id);
        if(strcmp(p->id, "0") == 0)
        {
            break;
        }
        printf("Please input name: ");
        scanf("%s", p->name);
        do
        {
            printf("Please input date (format of yyyy-mm-dd): ");
            scanf("%s", szBuf);
            if(toDate(&p->birthday, szBuf))
            {
                break;
            }
            else
            {
                printf("Invalid date format\n");
            }
        }while(1);
        printf("Please input height (unit of cm): ");
        scanf("%d", &p->height);

        if(!head)
        {
            head = p;
        }
        else
        {
            STUDENT_T *temp = head;
            while(temp->next)
                temp = temp->next;
            temp->next = p;
            p->prev = temp;
        }
    }
}

```

```
    }
else
{
    insert(&head, p);
}
}

return head;
}

int main()
{
    STUDENT_T * head = create();
    if(!head)
    {
        printf("Create list failed!\n");
        return -1;
    }

    printf("The student info created: \n");
    output(head);
    save(head);
    release(head);

    load(&head);
    printf("The student info loaded: \n");
    output(head);

    release(head);
}
```

第7章

进程和信号

进程是操作系统中最基本和重要的概念。进程是程序的一次执行，运行在自己的虚拟地址空间，是系统资源调度的基本单位。当执行程序时，系统创建进程，分配 CPU 和内存等资源。进程结束时，系统回收这些资源。本章将介绍进程的基本概念、Linux 进程环境和 Linux 进程控制，包括创建新进程、执行程序和进程终止。信号是软件中断，提供了一种处理异步事件的方法。很多比较重要的应用程序都需处理信号。本章先对信号机制进行综述，并说明每种信号的一般用法，然后介绍 Linux 程序中对信号的处理方法。本章内容包括：

- 进程的概念
- 进程环境
- 进程的结构
- 进程控制
- 信号的概念
- 信号的发送和捕获

7.1 进程的基本概念

7.1.1 什么是进程

进程的概念是 20 世纪 60 年代初首先由麻省理工学院的 MULTICS 系统和 IBM 公司的 CTSS/360 系统引入的。很难给进程下一个简单而明确的定义，有人认为进程是“一个其中运行着一个或多个线程的地址空间和这些线程所需要的系统资源”，也有人认为进程就是“正在运行的程序”。

进程是操作系统中最基本、重要的概念，是多道程序系统出现后，为了刻画系统内部出现的动态情况，描述系统内部各道程序的活动规律引进的一个概念，所有采用多道程序设计的操作系统都建立在进程的基础上。从理论角度看，进程是对正在运行的程序过程的抽象。从实现角度看，进程是一种数据结构，目的在于清晰地刻画动态系统的内在规律，有效管理和调度进入计算机系统主存储器运行的程序。

进程和程序是既联系又区别的。程序是指令的有序集合，其本身没有任何运行的含义，是一个静态的概念。而进程是一个具有独立功能的程序关于某个数据集合在处理机上的一次执行过程，它是一个动态的概念。

进程可以申请和拥有系统资源，是一个活动的实体。它不只是程序的代码，还包括当前的活

动（通过程序计数器的值和寄存器的内容来表示）。进程由程序代码、数据、变量（占用着系统内存）、打开的文件（文件描述符）和环境组成。进程具有以下特征。

- 动态性：进程的实质是程序的一次执行过程，进程是动态产生、动态消亡的。
- 并发性：任何进程都可以同其他进程一起并发执行。
- 独立性：进程能够独立运行，是系统分配和调度资源的基本单位。
- 异步性：因进程的独立性，各进程按各自独立的、不可预知的速度向前推进。
- 结构性：从结构上看，进程大体上由程序段、数据段和进程控制块三部分组成。

7.1.2 Linux 进程环境

7.1.2.1 程序的入口

C 程序总是从 main 函数开始执行。main 函数的原型是如下。

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

其中，argc 是命令行参数的数目，argv 是指向命令参数的各指针所构成的数组。

当内核启动 C 程序时，首先调用一个特殊的启动例程（编译连接程序将该例程设置为可执行程序的起始地址）。启动例程从内核取得命令行参数和环境变量值，然后调用 main 函数，并将命令行参数传递给它。

程序清单 7-1 所示程序将其所有命令行参数都回送到标准输出上。

程序清单 7-1 ex_main.c

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     int i;
6     for(i = 0; i < argc; i++)
7         printf("argv[%d]: %s\n", i, argv[i]);
8     return 0;
9 }
```

ANSIC 和 POSIX.1 都要求 argv[argc] 是一个空指针。因此将程序清单 7-1 中的 for 循环改写为：for(i=0; argv[i] != NULL; i++)，程序具有相关的效果。

在命令行编译并运行该程序，相关命令和结果如下。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_main ex_main.c
jianglinmei@ubuntu:~/c$ ./ex_main argument1 Alice LAST
argv[0]: ./ex_main
argv[1]: argument1
argv[2]: Alice
argv[3]: LAST
```

7.1.2.2 进程的终止

进程可能正常终止，也可能异常终止，共有以下 5 种方式。

- (1) 正常终止：① 从 main 返回；② 调用 exit；③ 调用 _exit。
- (2) 异常终止：① 被一个系统信号终止；② 调用 abort，它产生 SIGABRT 信号，是上一种异常终止的特例。

上节提及的启动例程会在 main 函数返回后立即调用 exit 函数。如果将启动例程以 C 代码形式表示（实际上该例程常常用汇编语言编写），则它调用 main 函数的形式可能如下。

```
exit(main(argc,argv));
```

7.1.2.3 exit 和 _exit 函数

exit 和 _exit 函数用于正常终止一个程序：_exit 立即进入内核，exit 则先执行一些清除处理（包括调用执行各终止处理程序，关闭所有标准 I/O 流等），然后进入内核。exit 属于标准库函数，而 _exit 是底层系统调用，它们的原型如下。

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

exit 函数会执行一个标准 I/O 库的清除关闭操作，即对所有打开的流调用 fclose 函数。exit 和 _exit 都带一个整型参数，称之为退出状态（exit status）。大多数 Shell 都提供检查一个进程终止状态的方法。如果(a)调用这些函数时不带终止状态，或(b)main 执行了一个无返回值的 return 语句，或(c)main 执行隐式返回，则该进程的终止状态是未定义的。

7.1.2.4 atexit 函数

ANSI C 规定，一个进程可以登记多至 32 个由 exit 自动调用的函数，这些函数被称为终止处理程序（exit handler）。登记终止处理程序要使用 atexit 函数，它的函数原型如下。

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

atexit 的参数是一个函数地址，atexit 函数成功返回 0，失败返回非 0 值。exit 函数以登记这些函数的相反顺序调用它们。同一函数如若登记多次，则也将被调用多次。

图 7-1 显示了 C 程序的启动/终止的方式和过程，注意图中 exit 和 _exit 的区别。

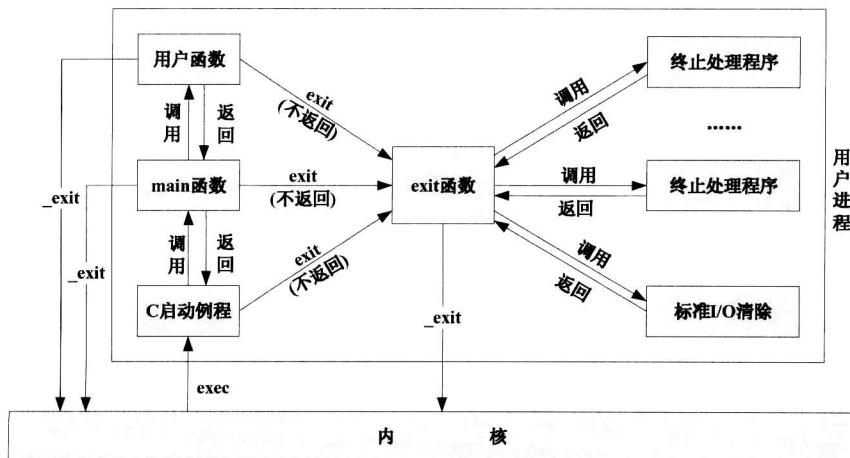


图 7-1 C 程序的启动/终止的方式和过程

内核使程序执行的唯一方法是调用一个 exec 函数（本章稍后介绍）。进程主动终止的唯一方法是显式或隐式地（调用 exit）调用 _exit。进程也可能被一个系统信号异常终止（图 7-1 中没有显示）。

程序清单 7-2 说明了 atexit 函数的使用方式。

程序清单 7-2 ex_atexit.c

```

1 #include <stdio.h>
2
3 static void my_exit1(void);
4 static void my_exit2(void);
5
6 int main(void)
7 {
8     if(atexit(my_exit2) != 0)
9     {
10         printf("register my_exit2 failed\n");
11         return -1;
12     }
13
14     if(atexit(my_exit1) != 0)
15     {
16         printf("register my_exit1 failed\n");
17         return -1;
18     }
19
20     printf("main is done\n");
21     return 0;
22 }
23
24 static void my_exit1(void)
25 {
26     printf("first exit handler\n");
27 }
28
29 static void my_exit2(void)
30 {
31     printf("second exit handler\n");
32 }
```

程序第 8 行和第 14 行先后登记了两个终止处理程序（函数），它们在 main 函数退出后被 exit 函数调用。

在命令行编译并运行该程序，结果如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_atexit ex_atexit.c
jianglinmei@ubuntu:~/c$ ./ex_atexit
main is done
first exit handler
second exit handler
```

7.1.2.5 环境表

每个进程在启动时都能接收到一张环境表。与命令行参数表一样，环境表也是一个字符指针数组，其中每个指针包含一个以 null 结束的字符串的地址。全局变量 `environ` 记录了该指针数组的地址。

```
extern char **environ;
```

全局变量 `environ` 称为环境指针，其所指向的数组称为环境表，数组中的每个指针指向的字

字符串称为环境字符串。环境字符串具有约定的形如“name=value”的格式。例如，具有5个环境字符串的环境表如图7-2所示。

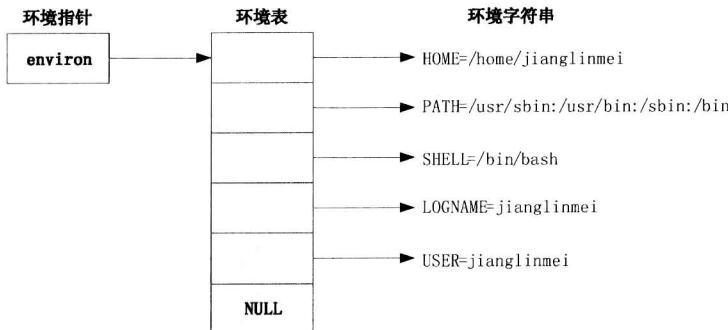


图7-2 含5个环境字符串的环境表

7.1.2.6 C程序的存储空间布局

C程序的存储空间一般由下列几部分组成。

(1) 正文段。由CPU执行的机器指令构成。通常，正文段共享的，所以同时启动一个程序的多个进程（如：运行两个bash Shell），在内存中只有一个正文段的副本。另外，正文段常常是只读的，这可以防止程序由于意外事故而修改其自身的指令。

(2) 初始化数据段。常称为数据段，由程序中已赋初值的静态变量构成。例如，如果有处于C程序中任何函数之外的定义：

```
int gNum = 100;
```

则此变量将以初值100存放在初始化数据段中。

(3) 非初始化数据段。常称为bss段，这一名称来源于早期汇编程序的一个操作符，意思是“block started by symbol（由符号开始的块）”。在程序开始执行之前，内核将此段初始化为0。例如，如果有处于C程序中任何函数之外的定义：

```
char *gName[100];
```

则此变量将存放在非初始化数据段中。

(4) 栈。自动变量以及每次函数调用时所需保存的场景信息（如返回地址、寄存器值）存放在栈中。另外，函数调用时也使用栈来传递参数，被调用的函数则在栈上为其自动变量和临时变量分配存储空间。

(5) 堆。通常在堆中进行动态存储分配。堆位于非初始化数据段顶和栈底之间。

图7-3显示了这些段的一种典型布局方式。注意，这是程序的逻辑布局，并不要求一个具体实现一定以这种方式安排其存储空间。

从图7-3还可注意到未初始化数据段的内容并不存放在磁盘程序文件中。需要存放在磁盘程序文件中的段只有正文段和初始化数据段。

使用size命令可查看一个可执行程序的正文段、数据段和bss段的长度（字节数）。例如：

```
jianglinmei@ubuntu:~/c$ size /bin/bash
text      data      bss      dec          hex      filename
799889     18452    20232   838573       ccbad    /bin/bash
```

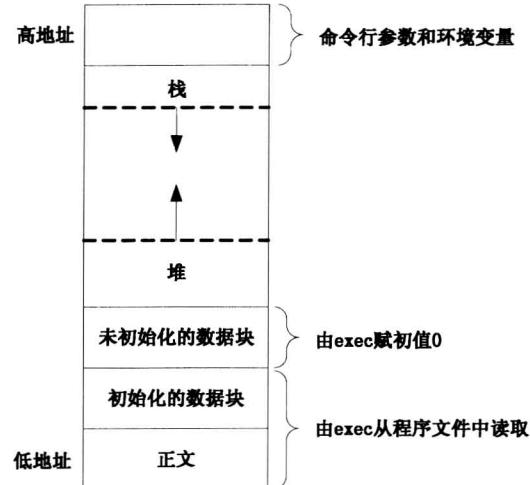


图 7-3 典型的存储空间布局

其中第 4 和第 5 列分别以十进制和十六进制显示各段的总长度。

7.1.2.7 静态库和共享库

所谓库，就是可复用的二进制可执行代码的有序集合。

Linux 下有两种类型的库，即静态库和共享库（又称动态库）。两者的一个重要区别在于其中的代码被载入的时刻不同：使用静态库的程序，在编译连接过程即载入静态库的代码并将静态库的代码置入编译出的可执行程序；而使用共享库的程序，在编译连接过程仅对共享库作简单引用，在程序运行时才将共享库的代码载入内存。

共享库的一个优点是，不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的副本。程序第一次执行或者第一次调用某个库函数时，用动态连接方法将程序与共享库函数相连接。这减少了每个可执行文件的长度，但增加了一些运行时间开销。共享库的另一个优点是可以用库函数的新版本代替老版本而无需对使用该库的程序重新编译和连接（假定参数的数目和类型都没有发生改变）。

在 Linux 下，库文件一般放在/usr/lib 和 /lib 下。静态库的名字一般为 libxxxx.a，其中xxxx是该库的名称。动态库的名字一般为 libxxxx.so.major.minor，其中xxxx是该库的名称，major 是主版本号，minor 是次版本号。

可使用 ar 命令将多个二进制目标代码文件打包成静态库。下面以一个示例来介绍静态库的建立和使用过程。先建立程序清单 7-3 至 7-5 所列静态库程序源文件。

程序清单 7-3 ex_static.h

```
1 extern int sum(int a, int b);
2 extern int average(int a, int b);
```

程序清单 7-4 ex_static1.c

```
1 int sum(int a, int b)
2 {
3     return a + b;
4 }
```

程序清单 7-5 ex_static2.c

```

1 int average(int a, int b)
2 {
3     return (a + b) / 2;
4 }
```

然后，在命令行编译生成目标代码并使用 ar 命令打包，如下所示。

```

jianglinmei@ubuntu:~/c$ gcc -c -o ex_static1.o ex_static1.c
jianglinmei@ubuntu:~/c$ gcc -c -o ex_static2.o ex_static2.c
jianglinmei@ubuntu:~/c$ ar rcs libmystatic.a ex_static1.o ex_static2.o
```

这样，即建立了一个静态库 mystatic，其库文件名为 libmystatic.a。可使用 Linux 下的 nm 命令来列出库中的符号清单，如下所示。

```

jianglinmei@ubuntu:~/c$ nm libmystatic.a

ex_static1.o:
00000000 T sum

ex_static2.o:
00000000 T average
```

对于每一个符号，nm 列出了它的值、类型和名称。这里的值“00000000”是符号在相应存储空间的段偏移，类型“T”表示正文段。

下面建立程序清单 7-6 所示代码，其中调用了上面建立的库文件中的函数。

程序清单 7-6 ex_test_static.c

```

1 #include <stdio.h>
2 #include "ex_static.h"
3
4 int main(void)
5 {
6     printf("3 + 5 = %d\n", sum(3, 5));
7     printf("average of 3 and 5 is: %d\n", average(3, 5));
8     return 0;
9 }
```

在命令行编译运行该程序，如下所示。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_test_static ex_test_static.c -L. -lmystatic
jianglinmei@ubuntu:~/c$ ./ex_test_static
3 + 5 = 8
average of 3 and 5 is: 4
```

Linux 下可使用 gcc 的编译选项-shared 将使用 gcc -c -fPIC 命令生成的目标文件打包成共享库。

使用 Linux 命令 ldd 可以查看一个二进制可执行程序或共享库所依赖的共享库。例如：

```

jianglinmei@ubuntu:~/c$ ldd ex_test_static
linux-gate.so.1 => (0x006c4000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0x0026d000)
/lib/ld-linux.so.2 (0x00fa8000)
```

7.2 进程的结构

7.2.1 进程控制块和进程表

Linux 内核是通过一个称为**进程控制块**的数据结构来对并发执行的进程进行控制和管理的。进程控制块的英文缩写是 PCB (Process Control Block), 在 Linux 内核中, PCB 由一个 task_struct 结构体定义。PCB 通常是系统内存占用区中的一个连续存区, 它存放着内核用于描述进程情况及控制进程运行所需全部信息。列举部分信息如下。

- (1) 进程标识符: 每个进程都有一个唯一的标识符 PID。
- (2) 进程当前状态: 说明进程当前所处的状态。为了管理的方便, 系统设计时会将相同的状态的进程组成一个队列, 如就绪进程队列, 根据等待的事件不同组成的等待打印机队列、等待磁盘 I/O 完成队列, 等等。
- (3) 进程的程序和数据地址: 用于将 PCB 与其程序和数据联系起来。
- (4) 进程资源清单: 列出所拥有的除 CPU 外的资源记录, 如拥有的 I/O 设备, 打开的文件列表等。
- (5) 进程优先级: 进程的优先级反映进程的紧迫程度, 通常由用户指定或系统设置。
- (6) CPU 现场保护区: 当进程因某种原因不能继续占用 CPU 时(如等待打印机), 释放 CPU, 这时就要将 CPU 的各种状态信息保护起来, 以便将来再次得到 CPU 时恢复各种状态, 继续运行。
- (7) 用于实现进程间通信所需的信息。
- (8) 其他信息: 如父进程的 PID、有效用户 ID、有效组 ID、进程占用 CPU 的时间、进程退出码、当前目录节点、执行文件节点等。

Linux 内核将所有进程控制块组织成指针数组形式, 形如:

```
struct task_struct *task[NR_TASK];
```

该指针数组即**进程表**, 其中记录了指向各 PCB 的指针。NR_TASK 规定了最多可同时运行进程的个数。在近期 Linux 版本中的 PCB 组成一个环形结构, 系统中实际存在的进程数由其定义的全局变量 nr_task 来动态记录。Linux 内核以 PID 作为进程表项(即进程控制块)的索引, 以此来管理系统中的各进程。

7.2.2 进程标识

如上一小节所述, 每个进程都有一个唯一进程标识 PID。在 Linux 中, PID 是一个非负整数。因为进程 ID 标识符总是唯一的, 常将其用做其他标识符的一部分以保证其唯一性。

在 Linux 中, 有三个进程具有其特殊性。

- (1) PID 为 0 调度进程。该进程是内核的一部分, 因此也被称为交换进程或系统进程。
- (2) PID 为 1 的 init 进程。该进程在系统自举过程结束时由内核调用, 其对应的程序文件是 /sbin/init。init 进程是由内核启动并运行的第一个用户进程(与交换进程不同, 它不是内核中的系统进程), 负责在内核自举后启动一个 Linux 系统, 它通常读与系统有关的初始化文件(/etc/rc* 文件), 并将系统引导到某一个状态(例如多用户)。init 进程决不会终止。它是一个普通的用户

进程，但是它以超级用户特权运行。

(3) PID 为 2 的 kthreadd 内核进程。该进程也是一个内核线程。内核经常需要在后台执行一些操作，这种任务可以通过内核线程（kernel thread）完成。内核线程是独立运行在内核空间的标准进程，和普通的进程间的区别在于内核线程没有独立的地址空间，从来不切换到用户空间去。kthreadd 由内核从 init 进程产生，用于衍生出其他的内核线程。

一般进程都是由一个“父进程”创建的，被父进程创建的进程称为“子进程”。PID 为 0 的交换进程是其他所有进程的祖先进程。init 进程是所有其他用户进程的祖先进程。kthreadd 内核线程是其他所有内核线程的父进程。使用“pstree”命令和“ps ax -o pid,ppid,command”命令可以清楚地查看进程间的父子关系。例如：

```
jianglinmei@ubuntu:~$ ps ax -o pid,ppid,command
  PID  PPID COMMAND
    1      0 /sbin/init
    2      0 [kthreadd]
    3      2 [ksoftirqd/0]
    5      2 [kworker/u:0]
    6      2 [migration/0]
    7      2 [migration/1]
.....
 644      1 /usr/sbin/sshd -D
 773      1 /usr/sbin/irqbalance
 774      1 cron
 775      1 atd
.....
18408  644 sshd: jianglinmei [priv]
18573 18408 sshd: jianglinmei@pts/0
18574 18573 -bash
```

除了 PID 以外，每个进程还有一些其他标识符。下列函数可返回这些标识符。

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void); /* 返回：调用进程的进程 ID */
pid_t getppid(void); /* 返回：调用进程的父进程 ID */
uid_t getuid(void); /* 返回：调用进程的实际用户 ID */
uid_t geteuid(void); /* 返回：调用进程的有效用户 ID */
gid_t getgid(void); /* 返回：调用进程的实际组 ID */
gid_t getegid(void); /* 返回：调用进程的有效组 ID */
```

7.2.3 进程的状态

Linux 是一个多用户、多任务的分时操作系统，可以同时运行多个用户的多个程序，即多道程序同时运行。在多道程序系统中，进程在处理器上交替运行，状态也不断地发生变化。图 7-4 说明了 Linux 进程状态和各状态间的转换关系。

1. 运行状态

指正在 CPU 中运行或者就绪的状态，包括：内核运行态、用户运行态、就绪态。Linux 内核并不对此三种状态进行区分。

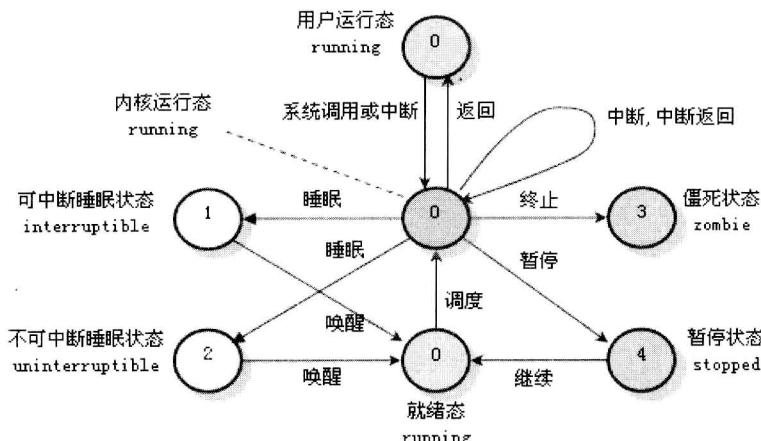


图 7-4 讲程状态和转换关系

2. 可中断睡眠状态

当进程处于可中断等待状态时，系统不会调度该进程执行。当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号，都可以唤醒进程转换到就绪状态。

3. 不可中断睡眠状态

不可中断，指的并不是 CPU 不响应外部硬件的中断，而是指进程不响应异步信号。处于该状态的进程不响应信号，只有使用 `wake_up()` 函数明确唤醒才能转换到就绪状态。该状态被设计用于保护内核的某些处理流程不被打断，或者在进行某些 I/O 操作时避免进程与设备交互的过程被打断，造成设备陷入不可控的状态。

4. 暂停状态

当进程收到信号 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 时就会进入暂停状态。可向其发送 SIGCONT 信号让进程转换到可运行状态。

5. 僵死状态 (TASK ZOMBIE)

当进程已停止运行，但其父进程还没有询问其状态时，则称该进程处于僵死状态。

当一个进程的运行时间片用完后，系统调度程序就会切换到其他的进程去执行。而当进程在内核态运行时需要等待某个资源，该进程就会自愿放弃 CPU 的使用权，而让调度程序去执行其他进程，自己进入睡眠状态。

只有当进程从“内核运行态”转移到“睡眠状态”时，内核才会进行进程切换操作。在内核态下运行的进程不能被其他进程抢占，而且一个进程不能改变另一个进程的状态。为了避免进程切换时造成内核数据错误，内核在执行临界区代码时会禁止一切中断。

可在 Linux 命令行下使用 “ps ax -o pid,stat,command” 命令查看进程所处的状态。例如：

```
jianglinmei@ubuntu:~$ ps ax -o pid,stat,command
PID STAT   COMMAND
  1 Ss      /sbin/init
  37 SN     [ksmd]
 892 Ssl    gdm-binary
2049 Sst+   bash
 8204 S<    udevd --daemon
12682 R+    ps ax
```

其中，中间的 STAT 列显示了各进程的当前状态，该列的每一个字符代表一种状态。常见的状态字符的含义如表 7-1 所示。

表 7-1

常见的进程状态字符

STAT 字符	说 明
S	睡眠。通常是在等待某个事件的发生
R	运行/可运行，即在运行队列中，处于正在运行或即将运行状态
D	不可中断的睡眠（等待，不响应异步信号）。通常是在等待输入或输出完成
T	停止。通常是被 Shell 作业控制所停止，或处于调试器控制下
Z	僵尸（zombie）进程
N	低优先级任务
s	进程是会话期首进程
+	进程属于前台进程组
I	进程是多线程的
<	高优先级任务

7.3 进 程 控 制

7.3.1 system 函数

在进程中执行另一个程序的一个简单方法是调用标准库函数 system，其原型如下。

```
#include <stdlib.h>
int system(const char *command);
```

system 函数以一个命令字符串 command 作为其参数，该命令字符串的格式和 Shell 命令行中使用的命令一样。system 函数运行 command 命令并等待该命令完成，本质是执行 “/bin/sh -c command”。system 函数调用成功时返回相应命令的退出状态码，如果无法启动 Shell 则返回 127，发生其他错误时返回 -1。

使用 system 函数并非启动其他进程的理想手段，因其必须先启动一个 Shell，再使用该 Shell 执行相应的命令。

下面以程序清单 7-7 所示程序来说明 system 函数的使用。

程序清单 7-7 ex_system.c

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     printf("Running ps with system\n");
7     system("ps -ef");
8     printf("Done.\n");
9     exit(0);
10 }
```

程序第7行调用了 system 函数，执行“ps -ef”命令列出当前系统的进程。
在命令行编译并运行该程序，命令如下。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_system ex_system.c
jianglinmei@ubuntu:~/c$ ./ex_system
Running ps with system
UID      PID  PPID  C STIME TTY          TIME CMD
root      1      0  0 Nov16 ?        00:00:01 /sbin/init
...
1001    20397 20396  0 18:53 pts/1    00:00:01 -bash
1001    20706 20397  0 19:13 pts/1    00:00:00 ./ex_system
1001    20707 20706  0 19:13 pts/1    00:00:00 sh -c ps -ef
1001    20708 20707  0 19:13 pts/1    00:00:00 ps -ef
Done.
```

从显示结果中可以看出，bash 创建 ex_system 进程，ex_system 进程创建了 sh 进程，然后 sh 进程创建最终的 ps 进程。

7.3.2 exec 函数

和 system 函数类似，可以调用 exec 系列函数以执行另外一个程序。这些函数的原型如下。

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
int execve(const char *path, char *const argv[],
           char *const envp[]);
```

和 system 函数需启动一个 Shell 来执行运行新的进程不同，当一个进程调用一种 exec 函数时，该进程将完全由新程序替换，新程序从其 main 函数开始执行。因为调用 exec 并不创建新进程，只是用另一个新程序替换了当前进程的正文、数据、堆和栈段，所以调用 exec 前后进程 ID 并未改变。

exec 系列函数各参数和返回值的含义如下。

1. path 待运行的程序全路径名（命令字符串）
2. file 待运行的程序名，通过 PATH 环境变量搜索其路径
3. arg 命令参数
4. ... 可选的一到多个命令参数，要求最后一个必须是 NULL
5. argv 命令参数指针数组
6. envp 传递给待运行程序的环境变量指针数组
7. 返回值
 - 成功时不返回（原程序不再执行）
 - 出错时返回-1，并设置 errno 变量

这些函数之间的第一个区别是，函数名包含字母 p(表示“path”)的 execp、execvp 和 execvpe 函数取文件名 file 作为第一个参数，其他函数则取路径名 path 作为第一个参数。当指定 file 作为参数时：如果 file 中包含 /，则就将其视为路径名，否则就按 PATH 环境变量的设定，在相关目录中搜寻可执行文件。如果 execp、execvp 和 execvpe 在 PATH 所设路径中找到了一个可执行文件，但是该文件不是由编译连接程序产生的机器可执行代码文件，则认为该文件是一个 Shell 脚本，于是会试着调用 /bin/sh，并以该 filename 作为 Shell 的输入。

第二个区别与参数表的传递有关。函数名中包含字母 l(表示“list”)的 execl、execlp 和 execle 要求将新程序的每个命令行参数都作为一个单独的参数，然后在最后一个命令行参数后附加一个空指针参数结尾。函数名中包含字母 v(表示“vector”)的另外三个函数 execv、execvp 和 execve 则要求先构造一个指向各参数的指针数组，数组的最后一个元素也必须是空指针，然后以该数组地址作为这三个函数的参数。

最后一个区别与向新程序传递环境表相关。函数名以字母 e(表示“environment”)结尾的三个函数 execle、execvpe 和 execve 可以传递一个指向环境字符串指针数组的指针，该指针数组也必须以空指针作为其最后一个元素。其他四个函数则无法传递环境变量，只能使用调用进程中的全局 environ 变量为新程序复制现存的环境。

这些函数中，前 6 个函数是通过 execve 函数实现的，称为 execve 函数的前端。execvpe 函数是 GNU 的扩展。

程序清单 7-8 说明各个 exec 函数的基本用法。

程序清单 7-8 exec 函数的基本用法

```
#include <unistd.h>
/* 命令行参数指针数组范例，注意数组应以命令名本身（即 argv[0]）作为第一个元素 */
char *const ps_argv[] = {"ps", "ax", 0};

/* 环境变量指针数组范例 */
char *const ps_envp[] = {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* 各个 exec 函数的调用方法范例，以下函数不可能同时成功调用，
 * 因一旦一个调用成功就不会返回，其后的语句将不再有机会得到执行。 */
execl("/bin/ps", "ps", "ax", 0);           /* 假设 ps 命令在 /bin 目录下 */
execlp("ps", "ps", "ax", 0);                /* 假设 /bin 目录在 PATH 环境变量中 */
execle("/bin/ps", "ps", "ax", 0, ps_envp);
execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execvpe("ps", ps_argv, ps_envp);
execve("/bin/ps", ps_argv, ps_envp);
```

前面曾提及在执行 exec 后，进程 ID 没有改变。除此之外，执行新程序的进程还保持了原进程的下列特征。

- (1) 进程 ID 和父进程 ID。
- (2) 实际用户 ID 和实际组 ID。
- (3) 添加组 ID。
- (4) 进程组 ID。
- (5) 会话期 ID。

- (6) 控制终端。
- (7) 闹钟尚余留的时间。
- (8) 当前工作目录。
- (9) 根目录。
- (10) 文件权限创建屏蔽字。
- (11) 文件锁。
- (12) 进程信号屏蔽。
- (13) 未决信号。
- (14) 资源限制。

(15) 用户态运行时间、内核态运行时间、子进程用户态运行时间和子进程内核态运行时间。

对打开文件的处理与每个描述符的 exec 关闭标志值有关。进程中每个打开的描述符都有一个 exec 关闭标志 (FD_CLOEXEC)。若此标志设置，则在执行 exec 时关闭该描述符，否则该描述符仍打开。除非特地用 fcntl 设置了该标志，否则系统的默认操作是在 exec 后仍保持这种描述符打开。

POSIX.1 标准明确要求在 exec 时关闭打开的目录流。通常在 opendir 函数内部调用即调用了 fcntl 函数为对应于打开目录流的描述符设置 exec 关闭标志。

注意，在 exec 前后实际用户 ID 和实际组 ID 保持不变，而有效 ID 是否改变则取决于所执行程序的文件 SUID 位和 SGID 位是否设置。如果新程序的 SUID 位已设置，则有效用户 ID 变成程序文件所有者的 ID，否则有效用户 ID 不变。对组 ID 的处理方式与此相同。

程序清单 7-9 说明 exec 函数的效用。

程序清单 7-9 ex_exec.c

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main()
5 {
6     printf("Running ps with execlp\n");
7     execlp("ps", "ps", "ax", NULL);
8     printf("Done.\n");
9     exit(0);
10 }
```

程序清单 7-9 中的代码和程序清单 7-7 的代码基本一样，只是在第 7 行用 execlp 函数代替了 system 函数。

在命令行编译并运行该程序，命令如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_exec ex_exec.c
jianglinmei@ubuntu:~/c$ ./ex_exec
Running ps with execlp
UID      PID  PPID C STIME TTY          TIME CMD
root      1      0  0 Nov16 ?        00:00:01 /sbin/init
.....
1001    21758 21757 1 22:28 pts/2    00:00:01 -bash
1001    21874 21758 0 22:29 pts/2    00:00:00 ps -ef
```

从显示结果中可以看出，并不存在原始的 ex_exec 进程，bash 进程就是 ps 进程的父进程。

7.3.3 fork 函数

一个现存进程创建一个新进程的唯一方法是调用 fork 或 vfork 函数（7.2.2 节介绍的三个特殊进程除外，vfork 函数在下一小节介绍）。fork 函数的原型如下。

```
#include <unistd.h>
pid_t fork(void);
```

由 fork 创建的新进程被称为子进程（child process）。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是 0，而父进程的返回值则是子进程的 PID（因交换进程的 PID 为 0，所以一个子进程的进程 ID 不可能为 0）。在子进程中可以调用 getppid 函数获得父进程的 PID。当 fork 失败时，将返回 -1，并设置全局变量 errno。fork 典型错误为：

- (1) E_AGAIN —— 子进程超过 CHILD_MAX 限制。
- (2) ENOMEM —— 进程表无足够空间。

在调用 fork 函数之后，子进程和父进程继续执行 fork 之后的指令。

一般来说，在 fork 之后是父进程先执行还是子进程先执行是不确定的，父进程和子进程是完全独立运行的。如果要求父、子进程之间相互同步，则要求采用某种形式的进程间通信机制。

子进程是父进程的复制品。例如，子进程获得父进程数据空间、堆和栈的复制品。注意，这是子进程所拥有的拷贝，父、子进程并不共享这些存储空间。如果正文段是只读的，则父、子进程共享正文段。

但是，Linux 下的 fork 函数并不对父进程的数据段、堆和栈进行完全拷贝，而是使用了写时复制（Copy-On-Write,COW）的技术，让由父、子进程共享这些区域，而且内核将它们的存取许可权改变为只读的。当有进程试图修改这些区域时，才由内核为有关部分做一个拷贝。

fork 创建的子进程继承了父进程的以下属性。

- (1) 已打开的文件描述符。
- (2) 实际用户 ID、实际组 ID、有效用户 ID、有效组 ID。
- (3) 添加组 ID。
- (4) 进程组 ID。
- (5) 对话期 ID。
- (6) 控制终端。
- (7) SUID 标志和 SGID 标志。
- (8) 当前工作目录。
- (9) 根目录。
- (10) 文件方式创建屏蔽字。
- (11) 信号屏蔽和排列。
- (12) 已打开的文件描述符的执行时关闭标志。
- (13) 环境。
- (14) 连接的共享存储段。
- (15) 资源限制。

父、子进程之间的区别是：

- (1) fork 的返回值。

(2) 进程 ID。

(3) 不同的父进程 ID。

(4) 子进程的用户态运行时间、内核态运行时间、子进程用户态运行时间和子进程内核态运行时间被设置为 0。

(5) 父进程设置的锁，子进程不继承。

(6) 子进程的未决告警被清除。

(7) 子进程的未决信号集设置为空集。

`fork` 有以下两种用法。

(1) 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。这在网络服务进程中是常见的——父进程等待委托者的服务请求，当请求到达时，父进程调用 `fork`，使子进程处理此请求，父进程则继续等待下一个服务请求。

(2) 一个进程要执行一个不同的程序。这对 Shell 是常见的情况。在这种情况下，子进程在从 `fork` 返回后立即调用 `exec`。当然，子进程在 `fork` 和 `exec` 之间可以更改自己的属性，如 I/O 重新定向、用户 ID、信号排列等。

程序清单 7-10 说明了 `fork` 函数的用法。

程序清单 7-10 ex_fork.c

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 int main()
6 {
7     pid_t pid;           /* 用于保存 PID */
8     char *message;      /* 用于保存消息字符串 */
9     int n = 2;           /* 计数变量 */
10
11    printf("fork program starting\n");
12    pid = fork();        /* 创建子进程 */
13    switch(pid)
14    {
15        case -1:          /* 出错 */
16            perror("fork failed");
17            exit(1);
18        case 0:            /* 子进程 */
19            message = "This is the child";
20            n = 5;
21            break;
22        default:           /* 父进程 */
23            message = "This is the parent";
24            n++;
25            break;
26    }
27
28    for(; n > 0; n--) {
29        puts(message);
30        sleep(1);
31    }

```

```

32
33     exit(0);
34 }

```

程序第 12 行调用 fork 函数创建子进程，返回值保存于变量 pid。第 13~26 行间的 switch 语句根据 fork 的返回值对出错情况、父进程和子进程作不同的处理。第 28~31 行的 for 循环根据父、子进程设置的计数值不同分别多次输出各自的消息字符串，该循环在父进程中循环 3 次，在子进程中循环 5 次，其中的 puts 语句总共执行了 8 次。

在命令行编译并运行该程序，命令和结果如下。

```

jianglinmei@ubuntu:~/c$ ./ex_fork
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
jianglinmei@ubuntu:~/c$ This is the child
This is the child

```

注意，程序运行结果中，有两条 “This is the child” 出现在了命令行提示符 “\$” 之后，这是因为父进程比子进程更早退出，回到了 Shell。

7.3.4 vfork 函数

vfork 是另一个可以创建子进程的函数，与 fork 函数最大的一个区别是，vfork 函数创建子进程后会阻塞父进程，其原型如下。

```
#include<sys/types.h>
#include<unistd.h>
pid_t vfork(void);
```

vfork 函数被调用一次，也会返回两次，返回值的含义也和 fork 完全一样。vfork 创建的新进程的目的是 exec 另一个程序。

vfork 与 fork 一样都创建一个子进程，但是它并不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用 exec 或 exit，于是也就不会存访该地址空间。但在子进程调用 exec 或 exit 之前，它在父进程的空间中运行。这种工作方式在一定程度上提高了效率。

vfork 和 fork 之间的另一个区别是：vfork 保证子进程先运行，在子进程调用 exec 或 exit 之后父进程才可能被调度运行。

程序清单 7-11 说明了 vfork 函数的功用。

程序清单 7-11 ex_vfork.c

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 int main()
6 {
7     pid_t pid;      /* 用于保存 PID */

```

```

8   char *message;      /* 用于保存消息字符串 */
9   int n = 2;           /* 计数变量 */
10
11  printf("fork program starting\n");
12  pid = vfork();       /* 用 vfork 创建子进程 */
13  switch(pid)
14  {
15  case -1:            /* 出错 */
16      perror("fork failed");
17      exit(1);
18  case 0:              /* 子进程 */
19      message = "This is the child";
20      n = 5;
21      break;
22  default:             /* 父进程 */
23      message = "This is the parent";
24      n++;
25      break;
26  }
27
28  for(; n > 0; n--) {
29      puts(message);
30      sleep(1);
31  }
32
33  exit(0);
34 }

```

除第 12 行使用 vfork 代替 fork 以外，本程序其他代码与程序清单 7-10 中的代码完全一致。在命令行编译并运行该程序，命令如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_vfork ex_vfork.c
jianglinmei@ubuntu:~/c$ ./ex_vfork
fork program starting
This is the child
This is the parent

```

从程序的运行结果可以看到，子进程的信息完全输出运行完毕之后，父进程才得到运行并输出自己的信息。另外，子进程中的 for 循环执行了 5 次，而父进程中的 for 循环只执行了 1 次，为什么？如上所述，在子进程调用 exec 或 exit 之前，它在父进程的空间中运行，即子进程共享了父进程的变量。子进程运行完之后，变量 n 的值为 0。这时开始运行父进程在 vfork 之后的代码。父进程在第 24 行代码处将 n 的值增加了 1，因此其后的 for 循环执行 1 次。

7.3.5 进程的终止状态

在 7.1.2.2 小节曾描述，进程有 3 种正常终止方式和两种异常终止方式。不管进程如何终止，最后都会执行内核中的同一段代码。这段代码为相应进程关闭所有已打开的文件描述符，释放它所使用的内存等。

对任意一种终止情形，父进程都应当得到通知，知道子进程是如何终止的。对于正常终止的情况，传向 `exit` 或 `_exit` 的参数，或 `main` 函数的返回值，指明了它们的退出状态（`exit status`），内核以该“退出状态”作为进程的“终止状态”。在异常终止情况下，内核（不是进程本身）会产生一个指示其异常终止原因的终止状态（`termination status`）。终止进程的父进程可使用 `wait` 或 `waitpid` 函数（在下一节介绍）取得其终止状态。

上面介绍了子进程将其终止状态返回给父进程。但是如果父进程在子进程之前终止，会发生什么情况呢？答案是，对于父进程已经终止的所有进程，它们的父进程都改变为 `init` 进程，称这些进程被 `init` 进程领养。这种处理方法保证了每个进程都有一个父进程。

另外，如果子进程在父进程之前终止，那么父进程如何得到子进程的终止状态呢？答案是，进程终止的时候，内核并未马上释放其进程控制块（PCB），而是在其中保留了一定量的信息，所以当终止进程的父进程调用 `wait` 或 `waitpid` 时，可以得到这些信息。这些信息至少包括进程 ID、该进程的终止状态、以及该进程使用的 CPU 时间总量。

一个已经终止、但是其父进程尚未对其进行善后处理（获取终止子进程的有关信息、释放它仍占用的资源）的进程被称为僵尸进程（zombie）。

7.3.6 wait 和 waitpid 函数

父进程可以调用 `wait` 或 `waitpid` 函数等待子进程的结束。这两个函数的原型如下。

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

`wait` 函数等待任一子进程的结束，`waitpid` 函数则可等待指定的子进程的结束。这两个函数各参数和返回值的含义如下。

1. `status` 输出参数，用于获取子进程的退出状态。如果调用者不关心终止状态，则可将该参数指定为空指针
2. `pid` 要等待的子进程的 PID，值为 -1 时意为任一子进程
3. `options` 常见的选项为 `WNOHANG`，意为不阻塞调用者进程
4. 返回值
 - 成功时，返回已结束子进程的 PID；对于 `waitpid`，若指定了 `WNOHANG` 选项，但没有子进程终止，则返回 0
 - 出错时返回 -1，并设置 `errno` 变量

在父进程中调用 `wait` 或 `waitpid` 可能发生以下情况。

- (1) 阻塞（如果其所有子进程都还在运行）。
- (2) 带子进程的终止状态立即返回（如果一个子进程已终止，正等待父进程获取其终止状态）。
- (3) 出错立即返回（如果它没有任何子进程）。

当一个进程终止（正常或异常）时，内核会向其父进程发送 `SIGCHLD` 信号。父进程的默认处理是忽略该信号，但也可以设置一个信号发生时即被调用执行的回调函数以捕获该信号（关于信号处理在 7.4 节介绍）。如果父进程在捕获 `SIGCHLD` 信号的回调函数中调用 `wait`，则可期望 `wait` 会立即返回。但是如果在一个任意时刻调用 `wait`，则进程可能会阻塞。

`waitpid` 和 `wait` 区别如下。

(1) 在一个子进程终止前, wait 的调用者一定阻塞; 而 waitpid 可使用 WNOHANG 选项让调用者不阻塞。

(2) waitpid 可等待任一个子进程的终止, 也可以等待指定子进程的终止。这和传递给 pid 参数的值有关。

- ① pid == -1 等待任一子进程。
- ② pid > 0 等待其进程 ID 与 pid 相等的子进程。
- ③ pid == 0 等待其组 ID 等于调用进程的组 ID 的任一子进程。
- ④ pid < -1 等待其组 ID 等于 pid 的绝对值的任一子进程。

POSIX.1 规定终止状态用定义在<sys/wait.h>中的各个宏来查看。有三个互斥的宏可用来取得进程终止的原因, 它们的名字都以 WIF 开始。基于这三个宏中哪一个值是真, 就可选用其他宏来取得终止状态、信号编号等。表 7-2 列出了这些宏的含义。

表 7-2 用于解释进程退出状态的宏

宏	说 明
WIFEXITED(status)	如果子进程正常结束, 则取非零值
WEXITSTATUS(status)	如果 WIFEXITED 非零, 则得到子进程的退出码
WIFSIGNALED(status)	如果子进程因未捕获的信号而终止, 则取非零值
WTERMSIG(status)	如果 WIFSIGNALED 非零, 则得到引起子进程终止的信号代码
WIFSTOPPED(status)	如果子进程已意外终止, 则取非零值
WSTOPSIG(status)	如果 WIFSTOPPED 非零, 则得到引起子进程终止的信号代码

程序清单 7-12 说明了 wait 函数的使用方法。

程序清单 7-12 ex_wait.c

```

1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     pid_t pid;          /* 用于保存 PID */
10    char *message;      /* 用于保存消息字符串 */
11    int n = 2;           /* 计数变量 */
12    int exit_code;      /* 退出状态 */
13
14    printf("fork program starting\n");
15    pid = fork();        /* 创建子进程 */
16
17    switch(pid)
18    {
19        case -1:
20            perror("fork failed");
21            exit(1);
22        case 0:

```

```

23     message = "This is the child";
24     n = 5;
25     exit_code = 37; /* 子进程退出状态设为 37 */
26     break;
27 default:
28     message = "This is the parent";
29     n++;
30     exit_code = 0; /* 父进程退出状态设为 0 */
31     break;
32 }
33
34 for(; n > 0; n--)
35 {
36     puts(message);
37     sleep(1);
38 }
39
40 if (pid != 0)      /* 父进程 */
41 {
42     int stat_val;
43     pid_t child_pid;
44
45     child_pid = wait(&stat_val); /* 等待子进程退出并获取其退出状态 */
46
47     printf("Child has finished: PID = %d\n", child_pid);
48     if(WIFEXITED(stat_val))      /* 判断是否正常退出 */
49         printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
50     else
51         printf("Child terminated abnormally\n");
52 }
53
54 exit(exit_code);
55 }
```

本程序在程序清单 7-10 的基础上作了一些改动。在第 25 行和第 30 行分别设置了子进程和父进程的退出状态。第 40~52 行代码的作用是，在父进程中等待子进程的退出并获取其退出状态，输出退出的子进程的 PID，若正常退出则一并输出其退出状态值。

在命令行编译并运行该程序，命令和运行结果如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_wait ex_wait.c
jianglinmei@ubuntu:~/c$ ./ex_wait
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 24664
Child exited with code 37
```

7.4 信 号

7.4.1 简介

信号是软件中断，是 Linux 系统为响应某些条件而产生的一个事件。信号提供了一种处理异步事件的方法，可作为进程间通信的一种机制，由一个进程发送给另一个进程。

每个信号都有一个以 SIG 开头名称，例如 SIGINT、SIGALRM 等。这些信号名称在系统的头文件中（包含 signal.h 可引用）以宏的形式定义，对应一个正整数（信号编号）。

有以下情形可以产生一个信号。

(1) 当用户在终端键入某些组合键时产生信号。例如：在终端上按 Ctrl+C 组合键通常产生中断信号（SIGINT），这是一种常用的终止一个已失去控制的进程的方法。

(2) 硬件异常产生信号。例如：除数为 0、非法的内存访问等。这类异常一般由硬件检测到并将其通知内核，然后内核产生适当的信号发送给正在运行的进程。例如，向执行了非法的内存访问的进程发送一个 SIGSEGV 信号。

(3) 进程调用 kill 函数将信号发送给另一个进程或进程组。

(4) 用户在 Shell 命令行，使用 kill 命令将信号发送给其他进程。实际上 kill 命令只是 kill 函数的一个命令行接口，两者本质上是同一种情形。在命令行使用“kill -<信号名> <PID>”命令可向指定进程发送指定信号，使用“killall -<信号名> <命令名>”可向所有运行<命令名>的进程发送指定信号。

(5) 当检测到某种软件异常时产生信号。例如：在网络连接中接收到与约定的波特率不符的数据时产生 SIGURG 信号、向管道写数据时没有与之对应的读进程时产生 SIGPIPE 信号、设置的闹钟时间已经超时时产生 SIGALRM 信号等。

信号是一种典型的异步事件，产生信号的事件对进程而言是随机出现的。当信号产生时，可有三种处理方式：忽略、捕获或执行默认操作。

(1) 忽略信号。大多数信号都可使用这种方式进行处理。但 SIGKILL 和 SIGSTOP 这两种信号决不能被忽略，因为它们为超级用户提供了一种使进程终止或停止的可靠方法。另外，某些由硬件异常产生的信号（例如非法的内存访问或除以 0）也不应被忽略，如果忽略则进程的行为是未定义的。

(2) 捕获信号。为此进程必须事先设置响应信号的回调函数，让内核在信号产生时调用该函数。

(3) 执行系统默认动作。对表 7-3 所列的信号，系统采取的默认动作是立即终止该进程。表 7-4 则列出了 Linux 环境下一些其他常见的信号。

表 7-3 未捕获时会引起进程终止的信号

信号名称	说 明
SIGABORT	调用 abort 函数时产生，进程将异常终止
SIGALRM	超时。一般由 alarm 设置的定时器产生
SIGFPE	浮点运算异常，如除以 0 和浮点溢出

续表

信号名称	说 明
SIGHUP	终端关闭或断开连接。由处于非连接状态的终端发给控制进程或由控制进程在自身结束时发给每个前台进程
SIGILL	非法指令。通常由一个崩溃的程序或无效的共享内存模块引起
SIGINT	程序终止，一般由从终端敲入的中断字符（Ctrl + C）产生
SIGKILL	终止进程(此信号不能被捕获或忽略)，一般在 Shell 中用它来强制终止异常进程
SIGPIPE	向管道写数据时没有与之对应的读进程时产生
SIGQUIT	程序退出。一般由终端敲入的退出字符（Ctrl + \）产生
SIGSEGV	无效内存段访问。一般是因为对内存中的无效地址进行读写引起，如数组越界、解引用无效指针
SIGTERM	kill 命令默认发送的信号，要求进程结束运行。UNIX 在关机时也用此信号要求服务停止运行
SIGUSR1	用户定义信号 1，用于进程间通信
SIGUSR2	用户定义信号 2，用于进程间通信

表 7-4

未捕获时不会引起进程终止的常见信号

信号名称	说 明
SIGCHLD	子进程停止或退出时产生，默认被忽略
SIGCONT	如果进程被暂停则继续执行
SIGSTOP	停止执行（此信号不能被捕获或忽略）
SIGTSTP	终端挂起。通常因按下 Ctrl + Z 组合键而产生
SIGTTIN	后台进程尝试读操作。Shell 用以表明后台进程因需要从终端读取输入而暂停运行
SIGTTOU	后台进程尝试写操作。Shell 用以表明后台进程因需要产生输出而暂停运行

7.4.2 捕获信号

最简单的捕获信号的方式是调用 signal 函数。该函数的原型如下。

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

signal 函数各参数和返回值的含义如下。

1. signum 准备捕获或忽略的信号

2. handler 捕获到信号后由系统调用的回调函数。可以设为以下特殊值：

- SIG_IGN 忽略信号
- SIG_DFL 恢复默认行为

3. 返回值

● 成功时返回原先定义信号处理函数，如果原先未定义则返回 SIG_ERR，并设置 errno 为一正数值

● 出错时，如果给出的是一个无效的信号、不可捕获或不可忽略的信号，则返回 SIG_ERR，并设置 errno 为 EINVAL

进程启动时，所有信号的状态都为系统默认或忽略。调用 exec 函数会将原先设置为要捕捉的

信号都更改为默认动作。这很容易理解：一个进程原先要捕捉的信号，当其执行一个新程序后，就自然地不能再捕捉了，因为原先设定的信号捕捉函数的地址新程序中已无意义。当一个进程调用 fork 创建子进程时，其子进程继承了父进程的信号处理方式。因为子进程在开始时复制了父进程存储映像，所以信号捕捉函数的地址在子进程中是有意义的。

Shell 程序对信号的处理有其特殊性，它会自动将后台进程中对中断和退出信号的处理方式设置为忽略。这样，当用户在命令行按 Ctrl+C 或 Ctrl+\ 组合键时就不会影响到后台进程。否则的话，使用 Ctrl+C 或 Ctrl+\ 组合键将不但会终止前台进程，也会终止所有后台进程。

交互式程序通常会采用下列形式的代码来捕获 SIGINT 信号（对应 Ctrl+C）和 SIGQUIT 信号（对应 Ctrl+\）。

```
int sig_int(), sig_quit();
if(signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if(signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

这样处理后，仅当不忽略 SIGINT 和 SIGQUIT 时，进程才捕获它们。从 signal 的这两个调用中可以看到 signal 函数的限制：不改变信号的处理方式就无法确定信号的当前处理方式。

程序清单 7-13 说明了 signal 函数的使用方法。

程序清单 7-13 ex_signal.c

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void gotit(int sig)           /* 定义信号回调函数 */
6 {
7     printf("signal %d is captured.\n", sig);
8     signal(SIGINT, SIG_DFL);      /* 设置回默认动作 */
9 }
10
11 int main()
12 {
13     signal(SIGINT, gotit);       /* 设置 SIGINT 信号的回调函数 */
14     while(1)
15     {
16         printf("Hello World!\n");
17         sleep(1);
18     }
19 }
```

本程序非常简短，首先在第 5 行定义了一个函数 gotit，然后在主函数 main 中（第 13 行）将其设置为 SIGINT 信号（由用户在终端按 Ctrl+C 组合键产生）的回调函数。第 14~18 行的 while 循环是一个死循环没有退出条件，因此该程序只能通过信号来终止。

在命令行编译并运行该程序，命令及结果如下。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_signal ex_signal.c
jianglinmei@ubuntu:~/c$ ./ex_signal
Hello World!
Hello World!
```

```

^Csignal 2 is captured.
Hello World!
Hello World!
^C

```

从程序执行结果可见，用户第一次按下 Ctrl+C 组合键时，输出了程序清单 7-13 第 7 行打印的信息，说明程序捕获到了 SIGINT 信号。用户第二次按下 Ctrl+C 组合键时，程序终止。这是因为程序清单 7-13 第 8 行将对 SIGINT 信号的处理设置回了默认动作。

7.4.3 发送信号

7.4.3.1 kill 函数

一个进程可以调用 kill 函数给自己或其他进程发送信号。kill 函数的原型如下。

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

kill 函数各参数和返回值的含义如下。

1. pid 接收信号的进程 PID
2. sig 要发送的信号
3. 返回值 成功时返回 0；失败时返回 -1，并设置 errno 全局变量。常见的 errno 值为：
 - EINVAL 给定的信号无效
 - EPERM 发送进程权限不够
 - ESRCH 目标进程不存在

使用 kill 发送信号时，发送方进程应具有相应的权限，必须满足以下两个条件之一：

- (1) 接收信号进程和发送信号进程的所有者相同。
- (2) 发送信号的进程的所有者是超级用户。

程序清单 7-14 说明了 kill 函数的使用方法。

程序清单 7-14 ex_kill.c

```

1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     pid_t pid;
10    char *message;
11    int n = 2;
12
13    printf("fork program starting\n");
14    pid = fork(); /* 创建子进程 */
15
16    switch(pid)
17    {
18        case -1:
19            perror("fork failed");

```

```

20     exit(1);
21 case 0:
22     message = "This is the child";
23     n = 5;
24     break;
25 default:
26     message = "This is the parent";
27     n++;
28     break;
29 }
30
31 for(; n > 0; n--)
32 {
33     puts(message);
34     sleep(1);
35 }
36
37 if (pid != 0)
38 {
39     kill(pid, SIGINT); /* 向子进程发送 SIGINT 信号 */
40 }
41
42 return 0;
43 }

```

本程序在程序清单 7-10 的基础上添加了第 37~40 行的代码。这段代码的作用是在父进程中向子进程发送 SIGINT 信号。

在命令行编译并运行该程序，命令及结果如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_kill ex_kill.c
jianglinmei@ubuntu:~/c$ ./ex_kill
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child

```

比较程序清单 7-10 的输出结果可以看出，子进程的 for 循环只循环了 3 次（比设定的 n=5 次少了 2 次），这是因为父进程发送的 SIGINT 信号使子进程中途终止了。

7.4.3.2 alarm、pause 和 sleep 函数

使用 alarm 函数可以为进程设置一个定时器，在设定的时间到达时，产生 SIGALRM 信号。如果不忽略也不捕获此信号，其默认动作是终止进程。alarm 函数的原型如下。

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

alarm 函数的参数和返回值的含义如下。

1. seconds 指定几秒后产生 SIGALRM 信号，0 表示取消设置

2. 返回值 成功时返回以前设置的定时器时间的余留秒数；失败时返回 0

每个进程只能有一个定时器。在进程内再次调用 alarm 时，原定时器时间的余留秒数作为本

次 alarm 函数调用的值返回。以前登记的定时器时间则被新值取代。另外，alarm 设置的定时器并非周期性的定时器，即调用一次只产生一次 SIGALRM 信号。如果要实现每隔一定时间都周期性地产生 SIGALRM 的功能，必须在其信号处理程序中再次调用 alarm 函数。

应注意，经过指定秒后，内核产生 SIGALRM 信号，但由于进程调度的延迟，进程真正处理该信号还需一段额外的时间，即，alarm 函数设置的定时器并不能精确定时执行用户设定的操作。

调用 pause 函数可使调用进程挂起直至捕捉到一个信号，其原型如下。

```
#include <unistd.h>
int pause(void);
```

调用 pause 函数将使用进程挂起，直到进程捕获到一个信号并从该信号的信号处理程序中返回时，pause 函数才返回。在此情况下 pause 的返回 -1，并设置全局变量 errno 的值为 EINTR。

使用 alarm 和 pause，进程可使自己睡眠一段指定的时间。程序清单 7-14 中的 mysleep 函数实现了这一功能。

程序清单 7-15 说明了 kill 函数的使用方法。

程序清单 7-15 ex_alarm_pause.c

```
1 #include <unistd.h>
2 #include <signal.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 static void onTimer(int sig)          /* SIGALRM 信号处理程序 */
7 {
8     printf("The timer is timeout.\n");
9 }
10
11 static unsigned int mysleep(unsigned int seconds)
12 {
13     if (signal(SIGALRM, onTimer) == SIG_ERR)
14     {
15         printf("Fail to call signal()\n");
16         return seconds;
17     }
18
19     alarm(seconds);                  /* 设置定时器时间 */
20     pause();                        /* 暂停，等待信号的产生 */
21
22     return alarm(0);                /* 关闭定时器，返回定时器余留秒数 */
23 }
24
25 int main()
26 {
27     printf("Begin...\n");
28     mysleep(10);
29     printf("End...\n");
30
31     return 0;
32 }
```

程序 6~9 行定义了 SIGALRM 信号的信号处理程序（回调函数）。

程序第 11 行 ~ 第 23 行定义了一个睡眠函数。在第 13 行设置 onTimer 为 SIGALRM 信号的信号处理程序，第 19 行设置定时器，第 20 行调用 pause 函数使用进程暂停。当定时时间到达时，onTimer 函数被执行。当进程从 onTimer 函数中返回时，pause 函数即返回，进程继续往下执行。最后第 22 行关闭定时器并返回定时器余留秒数。

在命令行编译并运行该程序，命令及结果如下。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_alarm_pause ex_alarm_pause.c
jianglinmei@ubuntu:~/c$ ./ex_alarm_pause
Begin...
The timer is timeout.
End...
```

观察程序输出结果的过程，可以发现在输出“Begin...”之后，程序暂停了 10 秒左右，然后继续输出其他信息并退出。

程序清单 7-15 所示的 mysleep 函数虽能实现让进程睡眠若干秒的功能，但还存在以下缺陷。

- (1) 如果调用者已设置了定时器，则它会被 mysleep 函数中的第一次 alarm 调用消除。
- (2) mysleep 函数中修改了对 SIGALRM 的设置。作为一个公用函数应在该函数被调用时先保存原设置，并在该函数返回前恢复其设置。
- (3) 在调用 alarm 和 pause 之间有一个竞态条件。在一个繁忙的系统中，alarm 可能在调用 pause 之前超时，并调用了信号处理程序。如果发生了这种情况，而在调用 pause 后，再没有捕捉到其他信号，调用者将永远被挂起。

Linux 提供了一个更严谨的 sleep 函数实现了睡眠若干秒的功能。sleep 函数的原型如下。

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

sleep 函数使调用进程挂起直到：已经过了 seconds 所指定秒数，或者捕捉到一个信号并从信号处理程序返回。

如同 alarm 函数一样，sleep 函数的返回值为所设时间的余留秒数。

7.4.3.3 abort 函数

abort 函数的功能是使程序异常终止，其原型如下。

```
#include <stdlib.h>
void abort(void);
```

abort 函数将 SIGABRT 信号发送给调用进程。进程不应忽略此信号。进程捕捉 SIGABRT 后可在进程终止之前执行必要的清理操作。当信号处理程序返回时，abort 即终止进程。

7.4.4 信号集

POSIX.1 标准定义了数据类型 sigset_t 以存放一个信号集（由多个信号构成的集合），并且定义了五个处理信号集的函数。在 Linux 中，包含头文件 signal.h 头文件即可引用 sigset_t 和这五个信号集处理函数。这些函数的原型如下。

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
```

```
int sigdelset(sigset_t *set, int sig);
int sigismember(const sigset_t *set, int sig);
```

各函数的参数和返回值的含义如下。

1. set 信号集

2. sig 信号

3. 返回值

- 成功时, `sigismember` 函数返回 1 (表示“是”) 或 0 (表示“否”); 其他函数返回 0
- 失败时返回 -1, 并设置 `errno` 变量, 只有一个可能的错误代码 `EINVAL`, 表示给定的信号无效

函数 `sigemptyset` 初始化由 `set` 指向的信号集, 使其排除所有信号。函数 `sigfillset` 初始化由 `set` 指向的信号集, 使其包括所有信号。所有应用程序在使用信号集前, 应该对该信号集调用 `sigemptyset` 或 `sigfillset` 一次。

一旦已经初始化了一个信号集, 就可在该信号集中增、删特定的信号。函数 `sigaddset` 将一个信号添加到信号集中, `sigdelset` 则从信号集中删除一个信号。`sigismember` 则用于判断一个信号是否为信号集中的一员。

7.4.5 `sigaction` 函数

`sigaction` 是一个比 `signal` 更健壮的用于捕获信号的编程接口, 其原型如下。

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oldact);
```

`sigaction` 函数的参数和返回值的含义如下。

1. `sig` 准备捕获或忽略的信号

2. `act` 将要设置的信号处理动作

3. `oldact` 用于取回原先的信号处理动作

4. 返回值 成功时返回 0; 失败时返回 -1, 并设置 `errno` 变量。如果给出的是一个无效的信号或不可捕获或不可忽略的信号, `errno` 为 `EINVAL`

`sigaction` 函数的 `act` 和 `oldact` 参数是一个结构体指针, 该结构体具有如下形式的定义。

```
struct sigaction {
    void (*sa_handler)(int); /* 信号处理函数, 同 signal */
    sigset_t sa_mask;        /* 回调过程中将被屏蔽的信号集 */
    int sa_flags;            /* 可决定回调行为的位标志值 */
    ....                   /* 未列出的其他不重要的成员 */
}
```

如果 `act` 指针非空, 则表示要修改 `sig` 信号的处理动作。如果 `oldact` 指针非空, 则系统在其中返回该信号的原先动作。

当要更改信号动作时, 如果 `act` 参数的 `sa_handler` 指向一个信号捕捉函数(不是常数 `SIG_IGN` 或 `SIG_DFL`), 则 `sa_mask` 字段说明了一个信号集。在调用信号捕捉函数之前, 该信号集被加入到进程的信号屏蔽字中。仅当从信号捕获函数(也称“信号处理程序”)中返回时才将进程的信号屏蔽字恢复为原先值。这样, 在调用信号处理程序时就能阻塞某些信号。在信号处理程序被调用

时，系统建立的新信号屏蔽字会自动包括正被递送的信号。因此保证了在处理一个给定的信号时，如果这种信号再次发生，那么它会被阻塞到对前一个信号的处理结束为止。系统在同一类信号产生多次的情况下，通常并不将它们排队，所以如果在某种信号被阻塞时，它产生了五次，那么对这种信号解除阻塞后，其信号处理函数通常只会被调用一次。

一旦对给定的信号设置了一个动作，那么在用 `sigaction` 改变它之前，该设置就一直有效。

`sigaction` 结构的 `sa_flags` 字段用于设置对信号进行处理的可选项。常用选项及其含义如下。

1. `SA_NOCLDSTOP` 子进程停止时不产生 `SIGCHLD` 信号
2. `SA_RESETHAND` 在信号处理函数入口处将对此信号的处理方式重置为 `SIG_DFL`
3. `SA_RESTART` 重启可中断的函数而不是给出 `EINTR` 错误
4. `SA_NODEFER` 捕获到信号时不将它添加到信号屏蔽字中，即不自动阻塞当前捕获到的信号

使用 `SA_RESTART` 标志的理由是，程序中使用的许多系统调用都是可中断的，当接收到一个信号时，它们将返回一个错误并将 `errno` 设置为 `EINTR`，以此表明函数是因为一个信号而返回的。在设置了 `SA_RESTART` 标志的情况下，在信号处理函数执行完后被中断的系统调用将被重启。

下面对程序清单 7-13 的程序进行改写，用 `sigaction` 代替 `signal` 完成相似的功能，改写后的代码如程序清单 7-16 所示。

程序清单 7-16 ex_sigaction.c

```

1 #include <signal.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void gotit(int sig)          /* 定义信号回调函数 */
6 {
7     printf("signal %d is captured.\n", sig);
8
9     sleep(20);                /* 睡眠 20 秒 */
10 }
11
12 int main()
13 {
14     struct sigaction act;
15
16     act.sa_handler = gotit;
17     sigemptyset(&act.sa_mask);
18     sigaddset(&act.sa_mask, SIGQUIT);    /* 在信号回调函数中屏蔽 SIGQUIT 页 */
19     act.sa_flags = 0;
20
21     sigaction(SIGINT, &act, 0);        /* 设置 SIGINT 信号的回调函数 */
22     while(1) {
23         printf("Hello World!\n");
24         sleep(1);
25     }
26 }
```

相比于 `signal` 稍为复杂的是，为了将 `gotit` 函数设置为信号处理程序，在第 14 行先定义了一个 `struct sigaction` 结构体类型的变量 `act`，然后在第 16 行将 `gotit` 函数指针设置为 `act` 变量的

sa_handler 成员的值。

在第 18 行调用 `sigaddset` 函数将 SIGQUIT 信号（按 Ctrl+\ 组合键时产生）添加到 SIGINT 信号回调函数的屏蔽信号集中，当进程因捕获到 SIGINT 信号而进入信号回调函数 `gotit` 中时，进程将暂不响应 SIGQUIT 信号，而是在退出 `gotit` 函数时才响应。为了清晰地观察到这种效果，在 `gotit` 函数中（第 9 行）调用了 `sleep` 函数让进程睡眠 20 秒。

在第 21 行调用 `sigaction` 函数时，第三个参数值设为 0，意为不关心信号原先的处理动作是什么。

在命令行编译并运行该程序，命令及结果如下。

```
jianglinmei@ubuntu:~/c$ ./ex_sigaction
Hello World!
Hello World!
^Csignal 2 is captured.
^C退出
```

从程序执行结果可见，当按下 Ctrl+C 组合键时，输出了程序清单 7-16 第 7 行打印的信息，说明程序捕获到了 SIGINT 信号。此时，按下 Ctrl+\ 时，程序不会马上终止，这是因为程序清单 7-16 第 18 行将 SIGQUIT 信号添加到了屏蔽信号集中。

7.5 小结

本章首先介绍了进程的基本概念，较为详细地描述了 Linux 进程运行的相关环境，包括程序的入口、出口、环境表以及程序的存储空间布局等。随后介绍了进程的数据结构，在 Linux 环境下，进程采用了 PCB 结构组成的进程表来管理，其中 PID 是进程的唯一标识。进程可能处于不同的状态，进程在一定条件下可在不同的状态之间转换。在本章中间部分，重点介绍了 Linux 中用于进程控制的相关函数，掌握这些函数的使用方法是学习多进程编程技术的基础功课。对初学者而言，应深入理解 `fork` 函数一次调用两次返回的特点。本章最后部分较为详细地介绍了 Linux 中信号的概念和操作方法。对于这部分的知识，应注意掌握信号的机制重于掌握信号处理函数的使用。

7.6 习题

- (1) 请简述什么是进程，它和程序的关系如何？进程具有哪些特征？
- (2) 请编写程序，逆向输出用户从命令行传递给进程的参数。
- (3) 请简述 `exit` 和 `_exit` 函数的区别。
- (4) 请编写程序，在程序中用 `atexit` 函数登记一个终止处理程序。在主函数中打开一个文件流指针，然后在终止处理程序中关闭该文件流指针。
- (5) 请编写程序，输出进程的所有环境字符串。
- (6) 请简述 C 程序的存储空间一般由哪些部分组成。
- (7) Linux 静态库和共享库有哪些区别？请试建立一个自己的静态库。