

第 8 章

进程间通信

上一章介绍了如何创建进程和控制进程，以及如何在进程发送信号。采用信号机制可以通知进程有某个事件发生，但是无法给进程传递数据。也就是说，进程之间尚无法进行有效的信息交换。本章将介绍进程之间相互通信的其他技术，介绍如何在进程间交换信息，以及如何对多进程的数据访问进行同步。本章内容包括：

- IPC 简介
- 管道
- 命名管道（FIFO）
- 信号量
- 共享内存
- 消息队列

8.1 IPC 简介

关于进程间通信，类 UNIX 系统有一个专门的术语：IPC (Inter Process Communication)。

进程间通信就是在不同进程之间传递或交换信息。因此，要实现进程间通信，必须要有某个不同进程都能访问的存放数据的“公共场所”，即数据存放介质。

那么，不同进程之间存在着什么双方都可以访问的介质呢？

进程的用户空间是互相独立的，一般而言是不能互相访问的，唯一的例外是共享存储区。系统空间是公用，但是只有内核具有访问权限。除此以外，就只有外设了。在这个意义上，两个进程可以通过磁盘上的普通文件、或者通过“注册表”（Windows 系统）、或者通过第三方数据库进行信息的相互交换。广义上，通过外设进行的这种进程间的信息交换也是进程间通信的手段，但是术语 IPC 不以此为“进程间通信”。

Linux 环境下常用的 IPC 技术主要分三种：管道、System V IPC 和套接字。

管道有两种类型。①（普通）管道，通常有两个局限：一是半双工，只能单向传送数据，二是只能在同源进程（在进程创建上具有亲缘关系）间使用；② 命名管道，又称 FIFO 队列，它去除了普通管道的第二种限制，即可以在不相关的进程之间进行通信。

有三种类型的 System V IPC（后文将简称为 SysV IPC）通信技术。它们是消息队列、信号量和共享内存。这三种 IPC 在实现上具有很大的相似性。之所以称它们为 System V IPC，是因为它们最初是在 AT&T 公司的 System V.2 UNIX 版本中引入的。

套接字主要用于网络通信，可以在不同主机的进程间相互交换信息。因篇幅所限，本书对其不作专门介绍。

在上述的所有 IPC 通信技术中，管道、命名管道（FIFO）、消息队列、信号量和共享内存这五种技术是只能用于同一台主机的各个进程间的 IPC，也是一般意义上所指的 IPC。

8.2 管道

管道是出现最早的一种 IPC 技术，而且所有的类 UNIX 系统都提供这种进程间通信机制。如上一小节所述，管道有两个局限：

- (1) 管道实现的是半双工通信，即数据只能在一个方向上流动。
- (2) 只能在具有公共祖先的进程之间使用管道进行通信。通常，由一个进程创建管道，然后调用 fork 创建子进程，随后在父、子进程应用该管道进行通信。

8.2.1 pipe 函数

在 Linux 下，通过调用 pipe 函数来创建一个管道。该函数的原型如下。

```
#include <unistd.h>
int pipe(int filedes[2]);
```

pipe 函数参数和返回值的含义如下。

1. filedes 用于返回文件描述符的数组
2. 返回值
 - 成功时返回 0
 - 出错时返回 -1，并设置 errno 变量

pipe 函数通过参数 filedes 返回两个文件描述符。其中，filedes[0] 为读而打开，filedes[1] 为写而打开。filedes[1] 的输出将作为 filedes[0] 的输入。

通常，调用 pipe 函数的进程会接着调用 fork，这样就创建了一个父进程与子进程之间 IPC 通道，如图 8-1 所示。

fork 之后有两个操作选择，这取决于所需建立的管道的数据流向。对于从父进程到子进程的管道，父进程关闭管道的读端 (fd[0])，子进程则关闭写端 (fd[1])，如图 8-2 所示。反之，对于从子进程到父进程的管道，父进程关闭管道的写端 (fd[1])，子进程则关闭读端 (fd[0])。

在父、子进程各关闭管道的一端之后，双方即可分别调用 read (对于读进程) 或 write (对于写进程) 函数对未关闭的文件描述符进行读、写操作，从而实现 IPC 通信。在通信过程中应注意以下读、写规则。

(1) 当读一个写端已被关闭的管道时，在所有数据都被读取后，read 返回 0，以指示到了文件结束处。

(2) 如果写一个读端已被关闭的管道，则产生 SIGPIPE 信号。如果忽略该信号或者捕获该信号并从其处理程序返回，则 write 出错返回，errno 设置为 EPIPE。

在写管道时，已写但尚未被读走的字节数应小于或等于 PIPE_BUF (Linux 中一般是 4096 字节) 所规定的缓存的大小。

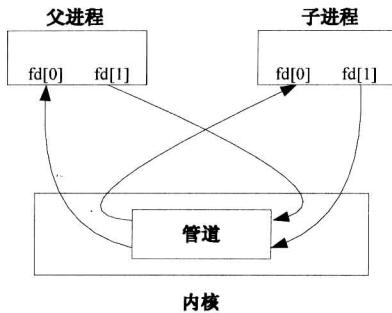


图 8-1 fork 之后的管道

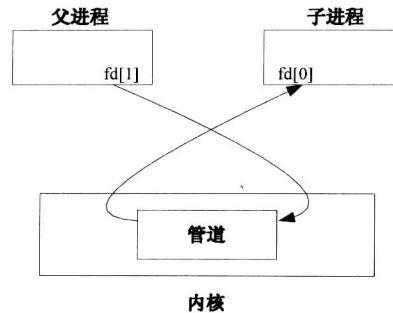


图 8-2 从父进程到子进程的管道

程序清单 8-1 说明了利用管道进行 IPC 通信的基本方法。

程序清单 8-1 ex_pipe.c

```

1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <string.h>
6
7 int main(int argc, char *argv[])
8 {
9     int pipefd[2];
10    pid_t cpid;
11    char buf;
12
13    if (argc != 2) {
14        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
15        exit(EXIT_FAILURE);
16    }
17
18    if (pipe(pipefd) == -1) { /* 创建管道 */
19        perror("pipe");
20        exit(EXIT_FAILURE);
21    }
22
23    cpid = fork();           /* 创建子进程 */
24    if (cpid == -1) {
25        perror("fork");
26        exit(EXIT_FAILURE);
27    }
28
29    if (cpid == 0) {         /* 子进程读管道 */
30        close(pipefd[1]);   /* 关闭写端 */
31
32        while (read(pipefd[0], &buf, 1) > 0)
33            write(STDOUT_FILENO, &buf, 1);
34
35        write(STDOUT_FILENO, "\n", 1);
36        close(pipefd[0]);
37        exit(EXIT_SUCCESS);

```

```

38 } else { /* 父进程将 argv[1] 写到管道 */
39     close(pipefd[0]); /* 关闭读端 */
40     write(pipefd[1], argv[1], strlen(argv[1]));
41     close(pipefd[1]); /* 关闭写端, 读端将读到 EOF */
42     wait(NULL); /* 等待子进程退出 */
43     exit(EXIT_SUCCESS);
44 }
45 }

```

本程序实现了基本的管道 IPC 功能, 父进程将用户从命令行传入的参数字符串传送给子进程, 然后子进程将收到的字符串输出到标准输出。

在程序的第 41 行, 父进程关闭了写文件描述符之后, 在程序的第 32 条, 子进程的 `read` 函数将返回 0, 从而退出相应的 `while` 循环。

在命令行编译并运行本程序, 命令和运行结果如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_pipe ex_pipe.c
jianglinmei@ubuntu:~/c$ ./ex_pipe "this is a string from father"
this is a string from father

```

8.2.2 `popen` 和 `pclose` 函数

如同 Shell 中使用的管道线 “|” 的作用一样, 在程序中常用管道实现将一个进程的输出连接到另一个进程的输入的操作。为此, 标准 I/O 库操作提供了两个函数 `popen` 和 `pclose` 以实现此功能。

`popen` 和 `pclose` 函数所实现的操作是: 创建一个管道, `fork` 一个子进程, 然后关闭管道的不使用端, 在子进程中 `exec` 一个 Shell 以执行一条命令, 然后等待命令的终止。它们的原型如下。

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

这两个函数的各参数和返回值的含义如下。

1. `command` 将在子进程中执行的命令行字符串
2. `type` 打开方式, 应为 “r” 或 “w” 二者之一
3. 返回值
 - 成功时, `popen` 返回一个文件流指针, `pclose` 返回 `command` 的终止状态
 - 出错时, `popen` 返回 `NULL`, `pclose` 返回 -1

函数 `popen` 先调用 `fork` 函数创建子进程, 然后调用 `exec` 函数以执行 `command`, 最后返回一个标准 I/O 文件指针。如果 `type` 是 “r”, 表示本进程可通过返回的文件指针读, 该文件指针连接到 `command` 的标准输出。如果 `type` 是 “w”, 表示本进程可通过返回的文件指针写, 该文件指针连接到 `command` 的标准输入。也即, `type` 参数的含义和与 `fopen` 函数的表示打开方式的参数的含义是一样的。另外, 默认情况下, 由 `popen` 打开的文件流是全缓存的。

函数 `pclose` 关闭标准 I/O 流, 并等待 `command` 命令执行结束, 最后返回 Shell 的终止状态(如果无法启动 Shell 则返回 127, 否则返回 `command` 命令的终止状态)。注意, 不应使用 `fclose` 来关闭 `popen` 打开的文件流。

`popen` 函数执行 `command` 的方式和 `system` 函数执行 `command` 的方式是一样的, 即相当于在

Shell 下执行 “sh -c command”。

程序清单 8-2 是一个简单的过滤程序，其功能是将从标准输入读取的所有字符转换成小写后写入标准输出。

程序清单 8-2 ex_popen_filter.c

```

1 #include <ctype.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(void)
6 {
7     int c;
8     while( (c = getchar()) != EOF ) {
9         if( isupper(c) )
10            c = tolower(c);
11         if( putchar(c) == EOF ) {
12             printf("fail in putchar()!\n");
13             exit(-1);
14         }
15         if(c == '\n')
16             fflush(stdout);
17     }
18     exit(0);
19 }
```

程序第 8 行，循环读取标准输入的字符，直到读取到 EOF（用户按下 Ctrl+D 组合键）为止。第 9 行判断所读取的字符是否是大写字母，如果是则在第 10 行将其转换为小写字母，随后在第 11 行将转换后的字符输出到标准输出。第 15~16 行，在读取到换行符时刷新缓存。

在命令行编译该程序，命令如下。

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

程序清单 8-3 调用程序清单 8-2 编译后的可执行程序 ex_popen_filter，用以说明通过 popen 和 pclose 函数使用管道进行 IPC 通信的基本方法。

程序清单 8-3 ex_popen.c

```

1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(void)
6 {
7     char    line[BUFSIZ];
8     FILE*   fpRead;
9
10    if( (fpRead = popen("./ex_popen_filter", "r")) == NULL) {
11        printf("Fail to call popen()!\n");
12        exit(-1);
13    }
14
15    for( ; ; ) {
16        fputs("PipeIn> ", stdout);
17        fflush(stdout);
```

```

18     if( fgets(line, BUFSIZ, fpRead) == NULL ) /* 读管道 */
19         break;
20     if( fputs(line, stdout) == EOF ) {
21         printf("Fail to call fputs()!\n");
22         exit(-1);
23     }
24 }
25
26 if(pclose(fpRead) == -1) {
27     printf("Fail to call pclose()!\n");
28     exit(-1);
29 }
30
31 putchar('\n');
32 exit(0);
33 }

```

程序第 10 行调用 `popen` 打开管道，在子进程中执行程序 `ex_popen_filter`。第 15 ~ 24 行的 for 循环反复从管道读取数据并将读取到的数据写入标准输出。第 16 行输出一条提示，因为标准输出通常是按行进行缓存的，而此处输出的提示并不包含换行符，所以要在第 17 行调用 `fflush` 将其立即输出。注意，在程序清单 8-2 的第 16 行调用 `fflush` 是因为 `popen` 实施的是全缓存，所以在读取到换行符时要刷新缓存，以便将数据及时送入管道。

在命令行编译本程序，命令和运行结果如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_popen ex_popen.c
jianglinmei@ubuntu:~/c$ ./ex_popen
PipeIn> Hello, world!
hello, world!
PipeIn> Data from FILTER.
data from filter.
PipeIn> [注: 按下 Ctrl + D]

```

8.3 命名管道 (FIFO)

命名管道又称 FIFO (First Input First Output)，即先入先出队列。如前所述，普通管道只能由相关进程使用，由它们共同的祖先进程创建管道。命名管道则克服了这个缺点，不相关的进程也能通过命名管道进行数据交换。

本书第 1.3.2.3 小节提到过，FIFO 也是一种文件类型。使用 `stat` 结构(见 5.2.5.1 小节)的 `st_mode` 成员的可判断文件是否是 FIFO 类型。

FIFO 的路径名存在于文件系统中，创建命名管道类似于创建文件。程序中应调用 `mkfifo` 函数来创建一个命名管道，该函数的原型如下。

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

函数的各参数和返回值的含义如下。

1. `pathname` 要打开或创建的文件的名字

2. mode 存取访问权限，同 open 函数的 mode 参数（见 5.2.2.1 小节）

3. 返回值 成功时返回 0，出错时返回 -1 并设置 errno

使用 mkfifo 创建的 FIFO 文件的所有者是当前进程的有效用户，文件的所属组是当前进程的有效组或父目录的所属组（与文件系统的类型和挂载选项有关）。

一旦用 mkfifo 创建了一个 FIFO 文件，就可用 open 函数打开它。此外，一般的文件 I/O 函数，比如 close、read、write、unlink 等，均可用于 FIFO。

当用 open 函数打开一个 FIFO 时，是否使用非阻塞标志（O_NONBLOCK）有以下影响。

(1) 未指定 O_NONBLOCK，则以读方式打开 FIFO 将阻塞到某个其他进程以写方式打开该 FIFO。反之，以写方式打开 FIFO 将阻塞到某个其他进程以读方式打开它。

(2) 指定了 O_NONBLOCK，则以读方式打开 FIFO 会立即返回。但是，如果在没有进程已经以读方式打开一个 FIFO 的情况下，就以写的方式打开该 FIFO，则相应的 open 函数将以失败返回，并设置 errno 为 ENXIO。

另外，与普通管道相同，如果向一个读端已关闭的 FIFO 写数据将产生信号 SIGPIPE。反之，如果某个 FIFO 的最后一个写进程关闭了该 FIFO，则读取该 FIFO 将得到一个文件结束标志。

当有多个写进程对应一个 FIFO 时（这是常见情况），需考虑原子写操作，已写但尚未被读走的字节数应小于或等于 PIPE_BUF。

程序清单 8-4 和 8-5 说明了 FIFO 的基本使用方法。

程序清单 8-4 ex_fifo.c

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int main(void)
9 {
10     int fd;
11     char buf;
12     char *fifofile = "/tmp/myfifo1234";
13
14     if(access(fifofile, F_OK) != 0) {          /* 文件不存在 */
15         if(mkfifo(fifofile, 0744) == -1) {      /* 创建 FIFO */
16             printf("Fail to create fifo.\n");
17             exit(EXIT_FAILURE);
18         }
19     }
20
21     if( (fd = open(fifofile, O_RDONLY)) == -1) {
22         printf("Fail to open fifo.\n");
23         exit(EXIT_FAILURE);
24     }
25
26     while (read(fd, &buf, 1) > 0)           /* 读 FIFO */
27         write(STDOUT_FILENO, &buf, 1);
28     write(STDOUT_FILENO, "\n", 1);
29 }
```

```

30     close(fd);
31     unlink(fifofile);           /* 删除 FIFO 文件 */
32     exit(EXIT_SUCCESS);
33 }

```

程序第 12 行定义了一个 FIFO 文件名字符串，第 14 行调用 access 函数判断管道文件是否存在，如不存在则在第 15 行创建管道文件。第 21 行，以读方式打开 FIFO 文件，然后在第 26 行循环读取 FIFO 中的字符，并在第 27 行将读取到的字符写入标准输出。

上面是读取 FIFO 的程序，下面来看写 FIFO 的程序。

程序清单 8-5 ex_fifo_writer.c

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 int main(int argc, char* argv[])
10 {
11     int fd;
12     char *fifofile = "/tmp/myfifo1234";
13
14     if (argc != 2) {
15         fprintf(stderr, "Usage: %s <string>\n", argv[0]);
16         exit(EXIT_FAILURE);
17     }
18
19     if( (fd = open(fifofile, O_WRONLY)) == -1) {
20         printf("Fail to open fifo.\n");
21         exit(EXIT_FAILURE);
22     }
23
24     write(fd, argv[1], strlen(argv[1])); /* 写 FIFO */
25
26     close(fd);                      /* 关闭写端, 读端将读到 EOF */
27     exit(EXIT_SUCCESS);
28 }

```

程序第 19 行调用 open 以写方式打开 FIFO，第 24 行将命令行参数字符串写入 FIFO。当程序运行到第 26 行关闭了写 FIFO 的文件描述符后，读 FIFO 的进程中 read 函数将返回 EOF（程序清单 8-4 第 26 行）。

在命令行编译并运行以上两个程序，命令和运行结果如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_fifo ex_fifo.c
jianglinmei@ubuntu:~/c$ gcc -o ex_fifo_writer ex_fifo_writer.c
jianglinmei@ubuntu:~/c$ ./ex_fifo &           [注: 在后台运行 ex_fifo]
[1] 31381
jianglinmei@ubuntu:~/c$ ./ex_fifo_writer "Hello, FIFO."
Hello, FIFO.
[1]+  完成                  ./ex_fifo

```

8.4 SysV IPC

本小节介绍三种 SysV IPC (信号量、共享内存和消息队列) 之间的共同特性。

- 标识符和关键字

对每种 SysV IPC 结构, 内核都有一个标识符(非负整数)予以标识和引用。与文件描述符不同的是, SysV IPC 标识符是进程无关并连续递增的。当一个 IPC 结构被创建以后, 与这种结构相关的标识符就加 1, 直至达到一个整型数的最大正值后才回转到 0。即使在 IPC 结构被删除后该值也不会丢失。为此, Linux 提供了 ipcs 命令和 ipcrm 命令分别用以查看系统中的 SysV IPC 和删除系统中遗留的 SysV IPC。

无论何时创建 SysV IPC 结构(调用 semget、shmget 或 msgget), 都必须指定一个关键字(key), 关键字的数据类型由系统规定为 key_t, 通常在头文件<sys/types.h>中规定为长整型。关键字由内核转换成标识符。

三个 get 函数(semget、shmget 和 msgget)都带两个的参数: 一个 key_t 型的 key 和一个整型的 flag。如果要创建一个新的 IPC 结构, 则必须指定 key 为 IPC_PRIVATE (一个特殊的键值, 它总是用于创建一个新队列), 或在 flag 中设置 IPC_CREAT 位。如果要访问现存的 IPC 结构, key 必须与创建该 IPC 时所指定的关键字一致, 并且不应指定 IPC_CREAT。

一般采用以下两种方法在进程间共用一个 SysV IPC 结构。

(1) 创建一个 IPC 结构时以 IPC_PRIVATE 作为参数, 然后将返回的标识符存放在某处(例如一个文件)以便其他进程取用。或者在父进程指定 IPC_PRIVATE 创建一个新 IPC 结构, 所返回的标识符在 fork 后可由子进程使用, 然后子进程可将此标识符作为 exec 函数的一个参数传给一个新进程。

(2) 指定一个具体的关键字。这种方法的问题是该关键字可能已与一个 IPC 结构相结合。在此情况下, 如果在 get 函数(semget、shmget 和 msgget)的 flag 参数中同时指定 IPC_CREAT 和 IPC_EXCL 位, 则返回 EEXIST。此时应删除已存在的 IPC 结构, 然后再创建它。

- 访问权限结构

SysV IPC 为每一个 IPC 结构都设置了一个 ipc_perm 结构。该结构规定了访问权限和所有者。

```
struct ipc_perm {
    key_t      __key;        /* 键 */
    uid_t      uid;          /* 所有者有效用户 ID */
    gid_t      gid;          /* 所有者有效组 ID */
    uid_t      cuid;         /* 创建者有效用户 ID */
    gid_t      cgid;         /* 创建者有效组 ID */
    unsigned short mode;     /* 访问权限 */
    unsigned short __seq;    /* 序列号 */
};
```

在创建 IPC 结构时, 除 seq 以外的所有字段都会被赋初值。IPC 结构的创建进程或超级用户的进程可以调用 ctl 函数(semctl、shmctl 或 msgctl)修改 uid、gid 和 mode 字段。mode 字段的值类似于 open 函数的 mode 参数(见 5.2.2.1 节), 但是对于任何 IPC 结构都不存在执行权限。

8.5 信号量

8.5.1 简介

信号量是一个计数器，用于多进程存取共享资源时的同步操作。使用信号量执行共享资源的访问控制应遵循“获取”和“释放”规则。

需获取共享资源时，测试控制该资源的信号量。若信号量的值为正，则进程可以使用该资源，将信号量值减1，表示它使用了一个资源单位；若此信号量的值为0，则进程进入睡眠状态，直至信号量值大于0时被唤醒，返回测试。

使用完共享资源时，使该信号量值增1。如果有进程正在睡眠并等待此信号量，则唤醒它们。

以上获取和释放信号量的操作分别被称为“P操作”和“V操作”。为了正确地实现信号量，信号量的测试、增1和减1操作应当是原子操作。为此，信号量通常是在内核中实现的。

最常用的信号量是二值信号量，它控制单个资源，初始值为1。但是，一般而言，信号量的初值可以是任一正值，该值说明有多少个共享资源单位可供使用。

但是，SysV的信号量比上述的要复杂得多，因为：① SysV以信号量集（多个信号量组成的集合）而非单一的信号量来处理信号量；② 创建信号量（semget）与对其赋初值（semctl）分开；③ 即使已没有进程在使用某个信号量集，它仍然存在于系统，因此必须在使用完后，手动清除已创建的信号量集。

8.5.2 semget 函数

semget函数创建一个新的信号量集或是获得一个已存在的信号量集，其原型如下。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

函数各参数和返回值的含义如下。

1. key 用于标识一个信号量集的键，通常为一整数
 2. nsems 信号量的数目
 3. semflg 标志位，指定IPC_CREAT表示创建信号量集，此时可与IPC_EXCL按位或（含义见8.4节）。在创建新的信号量集时其低9位用于指定访问权限
 4. 返回值 成功时，返回信号量集的标识符（一个非负整数）。出错时，返回-1并设置errno。
常见的错误值如下：
- EACCES 无访问权限
 - EEXIST 在同时指定IPC_CREAT和IPC_EXCL时，具有指定键的信号量集已存在
 - EINVAL nsems<0 或大于信号量数的极限值SEMMSL
 - ENOENT 键不存在

8.5.3 semop 函数

semop 函数用来改变信号量的值，其原型如下。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

函数各参数和返回值的含义如下。

1. semid 由 semget 返回的标识符
2. sops 指向一个结构体数组首元素的指针。该数组的每个元素指示了要对信号量集中哪个信号量做何操作
3. nsops 结构体数组的元素个数
4. 返回值 成功时，返回 0。出错时，返回 -1 并设置 errno

其中，sops 所指示的结构数组的每一个结构至少包含下列成员。

```
struct sembuf {
    short sem_num;           /* 信号量在信号量集中的序号 */
    short sem_op;            /* 要做的操作 */
    short sem_flg;           /* 选项标志 */
}
```

成员 sem_num 用于指定对信号量集中的哪个信号量进行操作，通常为 0，除非在使用一个信号量数组。

成员 sem_op 指定要对信号量做何操作，即信号量的变化量值。通常情况下中使用两个值：-1 表示执行 P 操作，用来等待一个信号量变为可用；+1 是表示执行 V 操作，用来通知一个信号量可在用。

成员 sem_flg 通常设置为 SEM_UNDO。该标志让操作系统跟踪当前进程对信号量所做的改变，如果占有信号量的进程终止时没有将其释放，操作系统将自动释放该信号量。

sops 所指示的所有动作会按数组元素的顺序全部“原子性”地执行，从而可以避免多个信号量的同时使用所引起的竞争条件。

8.5.4 semctl 函数

semctl 函数允许对信号量集进行多种类型的控制，其原型如下。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

函数各参数和返回值的含义如下。

1. semid 由 semget 返回的标识符
2. semnum 信号量在信号量集中的序号，当 cmd 为 IPC_RMID 时无效
3. cmd 要执行的动作
4. ... 依赖于 cmd 参数的可选参数

5. 返回值 成功时，返回一个非负数。出错时，返回-1 并设置 errno

当使用“...”所指示的参数时，用户必须自己定义一个如下形式的联合体并以该类型的变量作为第四个参数。

```
union semun {
    int             val;      /* cmd 为 SETVAL 时，指定设定值 */
    struct semid_ds *buf;    /* cmd 为 IPC_STAT 或 IPC_SET 时有效 */
    unsigned short   *array;   /* cmd 为 GETALL 或 SETALL 时有效 */
    struct seminfo   *_buf;   /* cmd 为 IPC_INFO 时有效 */
};
```

semctl 函数可用的 cmd 值很多，但最为常用的 cmd 值是：SETVAL 和 IPC_RMID。^① SETVAL 用于初始化信号量的值（该值表示可用共享资源的数量），应通过第四个参数 semun 联合体的 val 成员来传递该值，并且应在第一次使用信号量集之前设置。^② IPC_RMID 用于删除一个信号量集，信号量集的所有者或创建才有权删除。如果没有显示删除信号量集，它会一直存在于系统中。信号量集是有限资源，所以及时删除是十分必要的。

8.5.5 信号量的应用

最常用的信号量是最简单的二值信号量。本小节以程序清单 8-6 所示的二值信号量的例子来说明信号量的使用方法。

程序清单 8-6 ex_semaphore.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/sem.h>
7
8 static int set_semvalue(void);
9 static int semaphore_p(void);
10 static int semaphore_v(void);
11 static void del_sem_set(void);
12
13 /* 定义自己的 semun 联合体 */
14 union semun {
15     int             val;
16     struct semid_ds *buf;
17     unsigned short   *array;
18     struct seminfo   *_buf;
19 };
20
21 /* 定义全局变量 sem_id 保存信号量集的标识符 */
22 static int sem_id;
23
24 int main()
25 {
26     int i;
27     pid_t pid;
28     char ch;
```

```

29
30     /* 创建信号量集 */
31     sem_id = semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);
32     if(sem_id == -1) {
33         fprintf(stderr, "Failed to create semaphore set. \n");
34         exit(EXIT_FAILURE);
35     }
36     if(!set_semvalue()) {          /* 设置信号量的值 */
37         fprintf(stderr, "Failed to initialize semaphore\n");
38         exit(EXIT_FAILURE);
39     }
40
41     pid = fork();                /* 创建子进程 */
42     switch(pid)
43     {
44     case -1:
45         del_sem_set();           /* 删除信号量集 */
46         exit(EXIT_FAILURE);
47     case 0:                      /* 子进程 */
48         ch = 'O';
49         break;
50     default:                     /* 父进程 */
51         ch = 'X';
52         break;
53     }
54
55     srand((unsigned int) getpid()); /* 为随机数播种 */
56
57     for(i=0; i < 10; i++) {        /* 使用信号量控制临界区 */
58         semaphore_p();            /* P 操作，获取信号量 */
59         printf("%c", ch);         /* 在父进程中输出 X，在子进程中输出 O */
60         fflush(stdout);
61         sleep(rand() % 4);
62
63         printf("%c", ch);
64         fflush(stdout);
65         sleep(1);
66         semaphore_v();           /* V 操作，释放信号量 */
67     }
68
69     if(pid > 0) {                /* 父进程 */
70         wait(NULL);              /* 等待子进程退出 */
71         del_sem_set();           /* 删除信号量集 */
72     }
73
74     printf("\n%d - finished\n", getpid());
75     exit(EXIT_SUCCESS);
76 }
77
78 /* 设置信号量的值 */
79 static int set_semvalue(void)
80 {

```

```

81 union semun sem_union;
82
83 sem_union.val = 1;
84 if(semctl(sem_id, 0, SETVAL, sem_union) == -1)
85     return 0;
86
87 return 1;
88 }
89
90 /* P 操作, 获取信号量*/
91 static int semaphore_p(void)
92 {
93     struct sembuf sem_b;
94
95     sem_b.sem_num = 0;
96     sem_b.sem_op = -1;
97     sem_b.sem_flg = SEM_UNDO;
98     if(semop(sem_id, &sem_b, 1) == -1) {
99         fprintf(stderr, "semaphore_p failed/n");
100        return 0;
101    }
102
103    return 1;
104 }
105
106 /* V 操作, 释放信号量 */
107 static int semaphore_v(void)
108 {
109     struct sembuf sem_b;
110
111     sem_b.sem_num = 0;
112     sem_b.sem_op = 1;
113     sem_b.sem_flg = SEM_UNDO;
114     if(semop(sem_id, &sem_b, 1) == -1) {
115         fprintf(stderr, "semaphore_v failed/n");
116        return 0;
117    }
118
119    return 1;
120 }
121
122 /* 删除信号量集 */
123 static void del_sem_set(void)
124 {
125     union semun sem_union;
126
127     if(semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
128         fprintf(stderr, "Failed to delete semaphore/n");
129 }

```

这是本书到目前为止最长的一个程序。本程序首先创建一个信号量集（第 31 行），并对信号量集中的信号量值进行初始化（第 36 行）。然后调用 fork 创建子进程（第 41 行）。接下来，在父进程中设置待输出字符为'X'（第 51 行），在子进程中设置待输出字符为'O'（第 48 行）。在随后的 for 循环中（第 57~67 行）使用信号量保证每次循环的原子性，先执行 P 操作（第 58 行）申请资

源，输出字符，然后等待随机的 0~4 秒，再输出字符，等待 1 秒后执行 V 操作（第 66 行）释放资源，再进行下一次循环。最后，在父进程中，等待子进程退出并删除信号量集（第 71 行）。

程序中：设置信号量的值的功能封装于函数 `set_semvalue` 中；获取信号量的功能封装于函数 `semaphore_p` 中；释放信号量的功能封装于函数 `semaphore_v` 中；删除信号量集的功能封装于函数 `del_sem_set` 中。这使得代码清晰且易于复用。

在命令行编译并运行本程序，命令及运行结果如下。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_semaphore ex_semaphore.c
jianglinmei@ubuntu:~/c$ ./ex_semaphore
XXOOXXOOXXOOXXXXXOOXXOOXXOOXXOOXXOOXXOOOO
2707 - finished

2706 - finished
```

从运行结果可见，X 和 O 总是成对出现，这说明处于 P/V 之间的临界区代码是以原子的方式运行的。读者可以试将第 58 和第 66 行注释后再编译运行，比较运行结果有何不同。

8.6 共享内存

8.6.1 简介

共享内存允许两个或多个进程访问同一块存储区，就如同 `malloc()` 函数向不同进程返回了指向同一块内存区域的指针。当一个进程改变了这块地址中的内容的时候，其他进程都会察觉到这个更改。

访问共享内存区域和访问进程独有的内存区域一样快，并不需要通过系统调用或者其他需要切入内核的过程来完成。同时它也避免了对数据的各种不必要的复制。所以这是最快的一种 IPC。

使用共享内存必须注意多个进程之间对同一内存段的同步存取。由于系统内核没有对访问共享内存进行同步，所以程序员必须提供自己的同步措施。例如，在数据被写入之前不允许进程从共享内存中读取信息、不允许两个进程同时向同一个共享内存地址写入数据等。解决这个问题的常用方法是使用上一节介绍的信号量进行互斥访问。

要使用一块共享内存，进程必须首先分配它。随后需要访问这个共享内存段的每一个进程都必须将这个共享内存绑定到自己的地址空间中。在完成通信后，所有进程则将共享内存与自己的地址空间分离开来，并且由一个进程释放该共享内存段。

在 Linux 系统中，每个进程都有自己独立的虚拟内存空间。物理内存和虚拟内存均以分页的形式进行管理，系统页面的大小一般是固定的（Linux 下为 4KB，该值可通过 `getpagesize` 函数得到）。每个进程都维护一个从物理页面地址到虚拟页面地址之间的映射。分配共享内存其实就是在内核分配一个或多个新的内存页面（因此，共享内存段的大小都必须是系统页面大小的整数倍）。一个进程如需使用这个共享内存段，则必须建立进程本身的虚拟地址到共享内存页面地址之间的映射，该过程称为绑定。当对共享内存的使用结束之后，解除这个映射关系，该过程称为分离。最后，在共享内存段使用完毕时，必须有一个（且只能是一个）进程负责释放这些被共享的内存页面。

内核为每个共享内存段设置了一个 shmid_ds 结构用以管理共享内存。该结构的定义如下。

```
struct shmid_ds {
    struct ipc_perm shm_perm;          /* 所有者和权限标识 */
    size_t        shm_segsz;           /* 以字节为单位的段的长度 */
    time_t        shm_atime;           /* 最后绑定时间 */
    time_t        shm_dtime;           /* 最后分离时间 */
    time_t        shm_ctime;           /* 最后更改时间 */
    pid_t         shm_cpid;            /* 创建者进程的 PID */
    pid_t         shm_lpid;             /* 最后绑定或分离进程的 PID */
    shmat_t       shm_nattch;          /* 绑定数 */
    ...
};
```

8.6.2 shmget 函数

shmget 函数分配一个共享内存段，其原型如下。

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

函数各参数和返回值的含义如下。

1. key 用于标识一个共享内存键，通常为一整数
2. size 共享内存的字节数
3. shmflg 标志位，指定 IPC_CREAT 表示创建信号量集，此时可与 IPC_EXCL 按位或（含义见 8.4 节）。在创建新的共享内存时其低 9 位用于指定访问权限
4. 返回值 成功时，返回共享内存的标识符（一个非负整数）。出错时，返回 -1 并设置 errno。
常见的错误值如下：
 - EACCES 无访问权限
 - EEXIST 在同时指定 IPC_CREAT 和 IPC_EXCL 时，具有指定键的信号量集已存在
 - EINVAL size 小于 SHMMIN 或大于 SHMMAX，或 key 所指共享内存已存在，但 size 比已存在的共享内存更大
 - ENOENT 键不存在

8.6.3 shmat 和 shmdt 函数

shmat 函数将共享内存绑定到当前进程的内存空间，shmdt 函数将已绑定的共享内存与当前进程的内存空间相分离。它们的原型如下。

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

函数各参数和返回值的含义如下。

1. shmid 由 shmget 返回的标识符
2. shmaddr 绑定的地址（本进程地址空间的地址）

3. `shmflg` 标志位，常用的值为 `SHM_RND` 和 `SHM_RDONLY`
4. 返回值 对于 `shmat`, 成功时返回共享内存的地址；出错时，返回(`void*`)`-1` 并设置 `errno`；
对于 `shmdt`, 成功时返回 `0`, 失败时返回`-1` 并设置 `errno`

如果 `shmaddr` 指定为 `NULL`, 系统将选择一个合适的地址来绑定共享内存。

如果 `shmaddr` 不为 `NULL`, `shmflg` 指定了 `SHM_RND`, 则实际绑定地址为 `shmaddr` 向下舍入到最近的 `SHMLBA` (`Segment low boundary address`) 的倍数的位置。目前, Linux 下 `SHMLBA` 等于 `PAGE_SIZE` (内存页面大小)。否则, `shmaddr` 必须为一页对齐 (即页面大小的整数倍) 的地址。

如果 `shmflg` 指定了 `SHM_RDONLY`, 则共享内存以只读的方式绑定到本进程的地址空间, 进程应对该共享内存具有读的权限。否则共享内存以读、写的方式绑定到本进程的地址空间, 进程应对该共享内存具有读和写的权限。没有只写的绑定方式。

`shmat` 如果成功会更改与共享内存关联的 `shmid_ds` 结构, 将其 `shm_atime` 成员 设置为当前时间, `shm_lpid` 成员设置为当前进程的 PID, `shm_nattch` 成员的值则增 1。

传递给 `shmdt` 函数的 `shmaddr` 参数必须是本进程的由 `shmat` 函数返回的地址。`shmdt` 如果成功也会更改与共享内存关联的 `shmid_ds` 结构, 将其 `shm_dtime` 成员设置为当前时间, `shm_lpid` 成员设置为当前进程的 PID, `shm_nattch` 成员的值则减 1。如果 `shm_nattch` 的值变成了 0, 且共享内存段标记为删除, 则相应的共享内存段被删除。

在调用 `fork` 创建一个子进程的情况下, 子进程将继承已绑定的共享内存段。而在调用 `exec` 系列函数后, 所有已绑定的共享内存段会与新进程分离。当进程调用 `_exit` 函数退出的时候, 所有已绑定的共享内存段也会自动从进程分离出去。

8.6.4 `shmctl` 函数

`shmctl` 函数允许对共享内存进行多种类型的控制, 其原型如下。

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

函数各参数和返回值的含义如下。

1. `shmid` 由 `shmget` 返回的标识符
2. `cmd` 要执行的动作
3. `buf` 用于设置 (`cmd` 为 `IPC_SET`) 或获取 (`cmd` 为 `IPC_STAT`) 共享内存段的信息
4. 返回值 成功时返回非负数, 失败时返回`-1` 并设置 `errno`

`shmctl` 常用的 `cmd` 命令是 `IPC_RMID`, 用于删除一个共享内存段。在进程中调用 `exit` 和 `exec` 会使进程分离共享内存段, 但不会删除这个内存段。因此, 在结束使用每个共享内存段的时候都应当使用 `shmctl` 的 `IPC_RMID` 命令进行释放, 以免以后再申请内享内存时超过系统所允许的共享内存段的总数限制。

8.6.5 共享内存的应用

因为系统内核没有对共享内存的访问进行同步, 所以共享内存常结合信号量一起使用。程序清单 8-7 的基本功能是父进程循环随机产生大写字母, 并将其通过共享内存传递给子进程, 子进程读取到该字母后将其转换为相应的小写字母并将该小写字母传递给父进程, 最后父进程输出读取到的小写字母。该程序说明了共享内存结合信号量的基本使用方法。

程序清单 8-7 ex_sharememory.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/sem.h>
7
8 static int set_semvalue(void);
9 static int semaphore_p(void);
10 static int semaphore_v(void);
11 static void del_sem_set(void);
12
13 /* 定义自己的 semun 联合体 */
14 union semun {
15     int             val;
16     struct semid_ds *buf;
17     unsigned short   array;
18     struct seminfo   __buf;
19 };
20
21 /* 定义全局变量 sem_id 保存信号量集的标识符 */
22 static int sem_id;
23 /* 定义全局变量 shm_id 保存共享内存的标识符 */
24 static int shm_id;
25
26 int main()
27 {
28     int i;
29     pid_t pid;
30     char ch1, ch2;
31     char* pData = NULL;
32
33     /* 创建信号量集 */
34     sem_id = semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);
35     if(sem_id == -1) {
36         fprintf(stderr, "Failed to create semaphore set. \n");
37         exit(EXIT_FAILURE);
38     }
39     if(!set_semvalue()) {           /* 设置信号量的值 */
40         fprintf(stderr, "Failed to initialize semaphore\n");
41         exit(EXIT_FAILURE);
42     }
43     shm_id = shmget(IPC_PRIVATE, 4096, 0666 | IPC_CREAT);
44     if(shm_id == -1) {
45         fprintf(stderr, "Failed to create sharememory. \n");
46         del_sem_set();
47         exit(EXIT_FAILURE);
48     }
49
50     pid = fork();                  /* 创建子进程 */
51     if(pid == -1) {
52         perror("fork failed");
```

```

53     shmctl(shm_id, IPC_RMID, 0);           /* 删除共享内存 */
54     del_sem_set();
55     exit(EXIT_FAILURE);
56 }
57 else {
58     srand((unsigned int) getpid());        /* 为随机数播种 */
59     pData = (char*) shmat(shm_id, 0, 0);   /* 绑定 */
60
61     if (pid == 0) {                      /* 子进程 */
62         do {
63             semaphore_p();
64             ch1 = *pData;                  /* 读 */
65             ch2 = *(pData + 1);
66             if(ch2 == '@') {
67                 *pData = tolower(ch1);    /* 写 */
68                 *(pData + 1) = '#';
69             }
70             if(ch1 == 'Z') break;
71             semaphore_v();
72
73             sleep(1);
74         }while(1);
75     }
76     else {                                /* 父进程 */
77         for(i=0; i < 26; i++) {
78             semaphore_p();
79             *pData = 'A' + rand() % 26;   /* 写 */
80             if(i == 25) *pData = 'Z';
81             printf("%c", *pData);
82             *(pData + 1) = '@';
83             semaphore_v();
84             sleep(1);
85
86             do {
87                 semaphore_p();
88                 ch1 = *pData;              /* 读 */
89                 ch2 = *(pData + 1);
90                 if(ch2 == '#') {
91                     printf("%c", ch1);
92                     fflush(stdout);
93                     semaphore_v();
94                     break;
95                 }
96                 semaphore_v();
97             }while(1);
98         }
99     }
100
101     shmdt(pData);                      /* 分离 */
102 }
103
104     if(pid > 0) {                      /* 父进程 */

```

```

105     wait(NULL);           /* 等待子进程退出 */
106     shmctl(shm_id, IPC_RMID, 0); /* 删除共享内存 */
107     del_sem_set();          /* 删除信号量集 */
108 }
109
110 printf("\n%d - finished\n", getpid());
111 exit(EXIT_SUCCESS);
112 }
113 ..... /* main 函数后面的代码与程序清单 8-6 的代码完全一样, 故略去! */

```

程序第 43 行创建共享内存, 指定内存大小为 4096 (一个内存页), 权限为 0666 (所有用户可读写)。第 53 行和第 106 行删除已创建的共享内存, 第 53 行是因为 fork 失败中途退出而删除共享内存, 第 106 行是所有操作成功后在父进程中删除共享内存。

程序第 59 行将共享内存分别绑定到父子进程的地址空间, 该绑定地址由指针变量 pData 指向。

父子进程使用共享内存的第一和第二个字节进行通信。第一个字节存放相互传递的数据(大、小写字母)。第二个字节为通信的“旗语”, 父进程以字符'@'说明自己向共享内存写入的数据, 子进程以字符'#'说明自己向共享内存写入的数据。

程序第 62~74 行的 do...while 循环为子进程中的代码。在该循环中, 子进程获取信号量, 然后反复读取共享内存的第一个和第二个字节分别保存于变量 ch1 和 ch2。如果读到的第二个字节为'@'符号, 说明父进程已向共享内存写入的数据, 即将第一个字节的字母转换为小写然后写入共享内存的第一个字节, 同时将字符'#'写入共享内存的第二个字节以通知父进程转换完毕。第 72 行判断如果读取到了字母'Z', 则退出循环。最后在第 71 行释放信号量。

程序第 77~98 行的 for 循环为父进程中的代码。在该循环中, 父进程获取信号量, 然后随机产生一个大写字母, 保存到共享内存的第一个字节, 如果是最后一次循环则将字母'Z'保存到共享内存的第一个字节, 同时将字符'@'写入共享内存的第二个字节以通知子进程产生了新的字母, 然后释放信号量以便子进程获取信号量进行转换操作。随后, 在第 86~97 行的 do...while 循环中等待子进程的返回, 一旦读取共享内存的第二个字节的值为'#'即输出共享内存的第一个字节所存的字符并退出等待。

程序第 101 行, 在所有转换均完成后, 将共享内存从父子进程中分离出去。

在命令行编译并运行本程序, 命令及运行结果如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_sharememory ex_sharememory.c
jianglinmei@ubuntu:~/c$ ./ex_sharememory
QqUuXxVvTtCcEeKkLlBbCcQqQqRrEeYyYyKkOoGgMmOoLlRrXx
1808 - finished
Zz
1807 - finished

```

8.7 消息队列

8.7.1 简介

消息队列提供了一种在两个不相关的进程间传递数据的有效方法。使用消息队列可以从一个

进程向另一个进程发送数据块。每个数据块有一个最大长度的限制(由宏 `MSGMAX` 定义),系统中所有队列所包含的全部数据块的总长度也有一个上限值(由宏 `MSGMNB` 定义)。

消息队列独立于发送和接收进程而存在。如果没有在程序中删除消息队列,即使所有使用消息队列的进程都退出了,该消息队列和队列中的内容都不会被删除。它们余留在系统中直至有某个进程调用 `msgrecv` 读消息或调用 `msgctl` 删除消息队列,或者由用户执行 `ipcrm` 命令删除消息队列。这种行为和普通管道不同,当最后一个访问管道的进程终止时,管道就被完全删除了。和 FIFO 也有所不同,虽然当最后一个引用 FIFO 的进程终止时其名字仍保留在系统中,直至显式地删除它,但是留在 FIFO 中的数据却会在进程终止时全部删除。

消息队列是消息的链接表,存放在内核中并由消息队列标识符标识。

与共享内存相似,内核为每个消息队列设置了一个 `shmid_ds` 结构用以管理消息队列。该结构的定义如下。

```
struct msqid_ds {
    struct ipc_perm msg_perm;      /* 所有者和权限标识 */
    time_t        msg_stime;       /* 最后一次发送消息的时间 */
    time_t        msg_rtime;       /* 最后一次接收消息的时间 */
    time_t        msg_ctime;       /* 最后改变时间 */
    unsigned long _msg_cbytes;     /* 队列中当前数据字节数 */
    msgqnum_t     msg_qnum;        /* 队列中当前消息数 */
    msglen_t      msg_qbytes;      /* 队列允许的最大字节数 */
    pid_t         msg_lspid;       /* 最后发送消息的进程的 PID */
    pid_t         msg_lrpid;       /* 最后接收消息的进程的 PID */
};
```

8.7.2 msgget 函数

`msgget` 函数创建或获取一个消息队列,其原型如下。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

函数各参数和返回值的含义如下。

1. `key` 用于标识一个消息队列的键,通常为一整数
2. `shmflg` 标志位,指定 `IPC_CREAT` 表示创建信号量集,此时可与 `IPC_EXCL` 按位或(含义见 8.4 节)。在创建新的消息队列时其低 9 位用于指定访问权限
3. 返回值 成功时,返回消息队列的标识符(一个非负整数)。出错时,返回 -1 并设置 `errno`。常见的错误值如下:
 - `EACCES` 无访问权限
 - `EEXIST` 在同时指定 `IPC_CREAT` 和 `IPC_EXCL` 时,具有指定键的消息队列已存在
 - `ENOENT` 键不存在

调用 `msgget` 函数时,参数 `msgflg` 指定 `IPC_CREAT` 而未指定 `IPC_EXCL`,如果具有指定键的消息队列已存在,则只是忽略创建动作,而不会出错。

8.7.3 msgsnd 函数

msgsnd 函数把一条消息添加到消息队列中，其原型如下。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

函数各参数和返回值的含义如下。

1. msgid 由 msgget 返回的消息队列标识符
2. msgp 指向要发送的消息的缓冲区的指针
3. msgsz 消息长度。这个长度不包括长整形成员变量的长度（见下面的说明）
4. msgflg 标志。指定 IPC_NOWAIT 时，表示当队列满或达到系统限制时，函数立即返回（返回值为 -1），不发送消息
5. 返回值 成功时返回 0；失败时返回 -1，并设置 errno 变量

应当说明的是，msgsnd 所发送的消息受两方面的约束：① 长度必须小于系统规定的上限；

- ② 必须以一个长整形成员变量开始，接收函数以此成员来确定消息的类型。一般以下列形式来定义一个消息结构及该结构类型的变量。

```
struct my_message {
    long int message_type;      /* 消息类型 */
    <anydatatype> data;        /* 要发送的数据，可为任意数据类型 */
} msg;
```

在这样定义一个消息结构变量后，msgsnd 的 msgp 参数要指定为 &msg，msgsz 参数指定为 msg.data 的长度（字节数）。

8.7.4 msgrcv 函数

msgrcv 函数从一个消息队列获取消息，其原型如下。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
                int msgflg);
```

函数各参数和返回值的含义如下。

1. msgid 由 msgget 返回的消息队列标识符
2. msgp 指向准备接收消息的缓冲区的指针
3. msgsz 消息长度。这个长度不包括长整形成员变量的长度。msgp 所指向的缓冲区应大于或等于此长度
4. msgtyp 一个长整数。若值为 0，获取队列中的第一个可用消息；若值大于 0，获取具有相同类型的第一个消息；若小于 0，获取消息类型小于或等于其绝对值的第一个消息
5. msgflg 标志。指定 IPC_NOWAIT 时，表示当没有相应类型消息时，函数立即返回

(返回值为 -1), 不接收消息

6. 返回值 成功时返回 0; 失败时返回 -1, 并设置 errno 变量

8.7.5 msgctl 函数

msgctl 函数直接控制消息队列, 可对消息队列做多种操作, 其原型如下。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

函数各参数和返回值的含义如下。

1. msgid 由 msgget 返回的消息队列标识符

2. cmd 要采取的动作

- IPC_STAT 将内核所管理的消息队列的当前属性值复制到 buf (msqid_ds 结构) 中
 - IPC_SET 如果进程有足够的权限, 就把内核所管理的消息队列的当前属性值设置为 buf (msqid_ds 结构) 各成员的值
 - IPC_RMID 删除消息队列
3. buf 缓冲区, 作用视 cmd 而定
4. 返回值 成功时返回 0; 失败时返回 -1, 并设置 errno 变量。如果在进程正阻塞于 msgsnd 或 msgrcv 中等待时删除消息队列, 则这两个函数将以失败返回

8.7.6 消息队列的应用

本小节以程序清单 8-8 和程序清单 8-9 所示的例子来说明消息队列的使用方法。其中程序清单 8-8 的功能是从消息队列中接收一个字符串, 程序清单 8-9 的功能是将字符串发送到消息队列中。

程序清单 8-8 ex_msgrcv.c

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <errno.h>
5 #include <unistd.h>
6 #include <sys/msg.h>
7 /* 定义消息结构 */
8 struct my_msg_st {
9     long int my_msg_type; /* 长整型的消息类型 */
10    char some_text[BUFSIZ]; /* 接收的数据 */
11 };
12
13 int main()
14 {
15     int running = 1;
16     int msgid;
17     struct my_msg_st some_data;
18     long int msg_to_receive = 0;
19
20     /* 获取或建立消息队列 */
21     msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
```

```

22     if (msgid == -1) {
23         fprintf(stderr, "msgget failed with error: %d\n", errno);
24         exit(EXIT_FAILURE);
25     }
26
27     while(running) {
28         /*接收消息*/
29         if (msgrecv(msgid, (void *)&some_data, BUFSIZ,
30                     msg_to_receive, 0) == -1) {
31             fprintf(stderr, "msgrecv failed with error: %d\n", errno);
32             exit(EXIT_FAILURE);
33         }
34         printf("You wrote: %s", some_data.some_text);
35         if (strncmp(some_data.some_text, "end", 3) == 0) {
36             running = 0;
37         }
38     }
39     /*删除消息队列*/
40     if (msgctl(msgid, IPC_RMID, 0) == -1) {
41         fprintf(stderr, "msgctl(IPC_RMID) failed\n");
42         exit(EXIT_FAILURE);
43     }
44     exit(EXIT_SUCCESS);
45 }

```

程序第 8~11 行定义了消息结构。成员 `my_msg_type` 表示消息类型，该成员是必须的而且类型固定要求为长整型；成员 `some_text` 则用于保存实际接收到的数据。

程序第 21 行获取或创建消息队列，固定键为 1234，权限为所有用户可读、写。

程序第 27~38 行的 `while` 循环反复从消息队列接收消息，直到接到“`end`”字符串则退出循环。其中第 29 行调用 `msgrecv` 接收任意消息（第 4 个参数 `msg_to_receive` 的值为 0）。第 35 行判断接收到的是否是“`end`”。

程序第 40 行，删除用完的消息队列，及时回收资源。

程序清单 8-9 ex_mssnd.c

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <errno.h>
5 #include <unistd.h>
6 #include <sys/msg.h>
7
8 #define MAX_TEXT 512
9 /* 定义消息结构 */
10 struct my_msg_st {
11     long int my_msg_type;      /* 长整型的消息类型 */
12     char some_text[MAX_TEXT];   /* 传送的数据 */
13 };
14
15 int main()
16 {
17     int running = 1;
18     struct my_msg_st some_data;

```

```

19     int msgid;
20     char buffer[BUFSIZ];
21     msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
22     if (msgid == -1) {
23         fprintf(stderr, "msgget failed with error: %d\n", errno);
24         exit(EXIT_FAILURE);
25     }
26
27     while(running) {
28         printf("Enter some text: ");
29         fgets(buffer, BUFSIZ, stdin);
30         some_data.my_msg_type = 1;
31         strcpy(some_data.some_text, buffer);
32         /* 发送消息 */
33         if (msgsnd(msgid, (void *)&some_data, MAX_TEXT, 0) == -1) {
34             fprintf(stderr, "msgsnd failed\n");
35             exit(EXIT_FAILURE);
36         }
37         if (strncmp(buffer, "end", 3) == 0) {
38             running = 0;
39         }
40     }
41
42     exit(EXIT_SUCCESS);
43 }
```

程序第 10~13 行定义了消息结构，各成员的含义和程序清单 8-8 所示相同。

程序第 21 行获取或创建消息队列，固定键为 1234，权限为所有用户可读、写。

程序第 27~49 行的 while 循环反复读取用户从键盘输入的字符串，直到用户输入“end”为止，然后将读取到的字符串发送到消息队列。其中第 33 行调用 msgsnd 发送消息，指定消息类型为 1。第 37 行判断用户输入的字符串是否是“end”。

在命令行编译并运行以上两个程序，命令及运行结果如下。

```

jianglinmei@ubuntu:~/c$ ./ex_msgrcv &           [在后台运行接收程序]
[1] 2408
jianglinmei@ubuntu:~/c$ ./ex_msgsnd
Enter some text: Hello, message queue.
You wrote: Hello, message queue.
Enter some text: It's very easy to use it to
You wrote: It's very easy to use it to
Enter some text: transfer message between process.
You wrote: transfer message between process.
Enter some text: end
You wrote: end
[1]+  完成                  ./ex_msgrcv
```

8.8 小结

本章首先介绍了进程间通信的相关概念和术语，简要说明了 Linux 中常见的 IPC 的种类和各自的特点。随后介绍了最为通用的 IPC 机制——普通管道的两种创建和使用方法，一种是直接用

pipe 函数创建并用 read/write 读写, 另一种则是采用 popen 打开管道文件流并以文件流的方式进行读写。FIFO 管道是对普通管道的一种改进, 本章以实例介绍了在不相关的进程之间使用 FIFO 进行通信的方法。随后, 本章介绍了 SysV 的几种 IPC 机制的特点, 先后介绍了 SysV IPC 中的信号量、共享内存和消息队列的特点和相关函数的使用方法, 并一一举例说明了各种 IPC 的具体应用。

8.9 习 题

- (1) 简述 Linux 环境下有哪些常用的 IPC 技术。
- (2) 是不是所有的 IPC 技术都只能用于同一主机的进程之间相互关换信息?
- (3) 请简述管道和命名管道有何区别。
- (4) 请简述 SysV IPC 有何特点, 使用 SysV IPC 应注意哪些事项?
- (5) 什么是原子操作? 使用信号量时, 哪些动作应当是原子操作?
- (6) 使用共享内存进行 IPC 通信时, 可采用什么方法进行同步? 为何要进行同步?
- (7) 当所有使用消息队列的进程都退出以后, 消息队列中的数据会自动清除吗? 有哪些方法可以从系统中删除消息队列?
- (8) 请编写程序, 在程序中调用 fork 创建子进程, 然后在父进程中读取 argv[1]所指的文本文件, 将文件内容通过管道传送给子进程, 再由子进程将收到的文件内容的各行按字典序排序后输出到屏幕。
- (9) 分别使用 FIFO、共享内存和消息队列代替管道来实现第 (8) 题的要求。