第4章 进程间通信与同步

操作系统为进程提供了必要的隔离，使得进程内部获得“封闭”的“可再现”执行环境。但是也有很多场合需要进程间交互、协调完成任务，这就需要进程间通信手段以及同步手段。通信手段用于进程间的数据交换，而同步用于控制各自的执行步伐形成前后因果或互斥执行关系。本章将和读者一起编程实践，观察和感受 Linux 提供的各种进程间通信手段。

4.1. 进程间通信

只要数据能突破进程空间的隔离，完成数据在进程间传递就可以实现进程间通信。广义上来说，两个进程通过磁盘文件传递数据也是进程间通信手段，同理利用网络连接传递数据也是进程间通信。不过我们这里讨论更狭义的传统感念上的进程通信，也就是常说的管道、消息队列、共享内存等机制。

4.1.1. 管道

进程间的管道通信有两种形式，无名管道用于父子进程间，命名管可以用于任意进程间——命名管道在文件系统中有可访问的路径名。管道通信方式主要用于单向通信，如果需要双向通信则建立两条相反方向的管道。管道实质是由内核管理的一个缓冲区（一边由进程写入，另一边由进程读出），因此要注意，如果缓冲区满了则写管道的进程将会阻塞。另外管道内部没有显式的格式和边界，需要自行处理消息边界，如果多进程间共享还需要处理传送目标等工作。

无名管道

管道（pipe），或称无名管道，是所有 Unix 都提供的一种进程间通信机制。管道是单向的信道，进程从管道的写端口写入数据，需要数据的进程从读端口中获取数据，数据在管道中按到达顺序流动。Unix 命令中使用“|”来连接两个命令时使用的就是管道，例如“ls | more”将 ls 命令的标准输出内容写入到管道中，管道的输出内容作为 more 命令的标准输入。注意，重定向技术虽然看起来和管道很相似，例如“ls > temp”，但重定向并不使用管道。

shell 程序在处理 ls|more 时，首先调用 pipe()系统调用，创建管道（其核心就是一个内存缓冲区）并返回一对文件描述符，比如文件描述符 3（用于管道的读端）和文件描述符 4（用于管道的写端）。然后调用两次 fork()分别创建两个子进程（后面会替换成 ls 和 more 两个进程），此时两个子进程继承了 shell 的所打开的文件资源（含文件描述符 3 和 4），此时情形参见图 4-1。最后 shell 进程用 close()关闭文件并释放描述符 3 和 4（图 4-1 中用打叉的虚线表示关闭）。

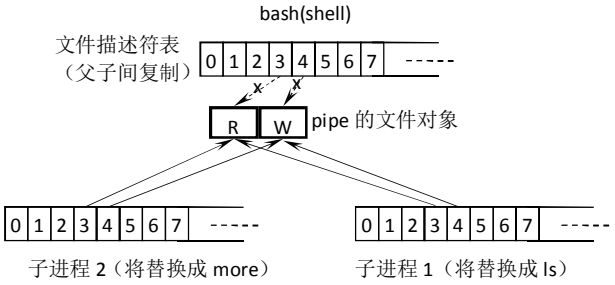

图 4-1 shell 调用 pipe()并创建两个子进程

图 4-2 表明子进程 1 通过 dup2(4,1)，将文件描述符 4 拷贝到文件描述符 1（标准输出文件），然后用 close()关闭文件 3、4 并释放相应的描述符，最后执行 execve()系统调用来执行 ls 程序，于是 ls 的标准输出（对应文件描述符 1）内容都写入到管道的写端。

相似地，子进程 2 通过 dup2(3,0)将文件描述符 3 拷贝到文件描述符 0（标准输入文件），然后用 close()关闭文件 3、4 并释放相应的描述符，最后执行 execve()系统调用来执行 more 程序，于是 more 从标准输入（对应文件描述符 0）读取的数据实际上就是管道的读端提供的数据——即 ls 写入的数据。具体细节如图 4-2 所示，此时管道起到连接两个进程输入输出的作用。

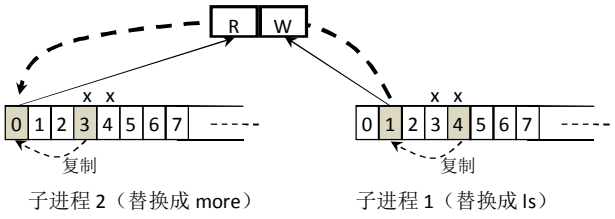

图 4-2 ls 与 more 通过管道进行通信

上述例子比较简单，仅仅在两个进程之间使用管道。实际上管道比较灵活，能看到管道所对应的文件描述符的进程之间都可以使用，但是务必注意同步关系。另外用户要关闭不使用的管道端口，否则可能会出现异常情况。C 库中的用户态函数 popen()和 pclose()对 pipe()系统调用进行了封装，更方便和安全。下面代码 4-1 以 pipe()为例展示父子进程间使用管道进行通信的方法，pipe()将通过两个文件描述符（整数）来指代管道缓冲区的读端和写端（代码中用 fds[]变量记录）。其中父进程关闭管道的读端 fds[0]并往管道的写端 fds[1]写出信息，子进程关闭了管道的写端 fds[1]并从管道的读端 fds[0]读回信息。

代码 4-1 pipe-demo.c 代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
int main()
{
    pid_t pid = 0;
```

```
    int fds[2];
    char buf[128];
    int nwr = 0;

    pipe(fds);                                  //在 fork()前执行

    pid = fork();
    if(pid < 0)
    {
        printf("Fork error!\n");
        return -1;
    }else if(pid == 0)
    {
        printf("This is child process, pid = %d\n", getpid());
        printf("Child:waiting for message...\n");
        close(fds[1]);                          //关闭写端 fds[1]
        nwr = read(fds[0], buf, sizeof(buf));   //从读端 fds[0]读入数据
        printf("Child:received\"%s\"\n", buf);
    }else{
        printf("This is parent process, pid = %d\n", getpid());
        printf("Parent:sending message...\n");
        close(fds[0]);                          //关闭写端 fds[0]
        strcpy(buf, "Message from parent! ");
        nwr = write(fds[1], buf, sizeof(buf));  //往写端 fds[1]写出数据
        printf("Parent:send %d bytes to child.\n", nwr);
    }
    return 0;
}
```

屏显 4-1 是 pipe-demo 运行的输出，其表明父进程发送了消息到管道，自进程成功接受到了“Message from parent”。

屏显 4-1 pipe-demo 的输出

```
[lqm@localhost ~]$ pipe-demo
This is parent process, pid = 8824
Parent:sending message...
Parent:send 128 bytes to child.
This is child process, pid = 8825
Child:waiting for message...
Child:received"Message from parent! "
[lqm@localhost ~]$
```

命名管道（FIFO）

前面提到的无名管道有一个主要缺点，只能通过父子进程之间（及其后代）使用文件描述符的继承来访问，无法在任意的进程之间使用。命名管道（named pipe）或者叫 FIFO 则突破了这个限制。可以说 FIFO 就是无名管道的升级版——有可访问的磁盘索引节点，即 FIFO 文件将出现在目录树中（不像无名管道那样只存在于 pipefs 特殊文件系统中）。

下面我们用 mkfilo 命令来创建命名管道 os-exp-fifo，如屏显 4-2 所示，其中 ls 命令查看时可以看出其类型是管道“p”。

屏显 4-2 mkfifo 创建命名管道

```
[lqm@localhost ~]$ mkfifo os-exp-fifo
[lqm@localhost ~]$ ls -l os-exp-fifo
prw-rw-r--. 1 lqm lqm 0 3月  18 09:01 os-exp-fifo
[lqm@localhost ~]$
```

此时可以用 cat os-exp-fifo 命令尝试从管道中读入数据，但是此时管道中还没有写入任何数据，因此 cat 将进入阻塞状态，如图 4-3 所示。

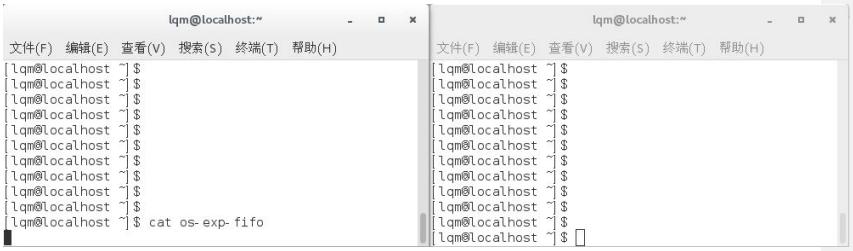

图 4-3 用 cat 尝试读取空的管道文件（阻塞）

如果此时在另一个终端上，用“echo Hello, Named PIPE! >os-exp-fifo”相关到写入数据，则 cat 会被唤醒并读入管道数据回显字符串“Hello, Named PIPE!”，如图 4-4 所示。

图 4-4 用 echo 向管道写入数据（cat 被唤醒并显示管道文件的内容）

在程序中使用命名管道的方法和普通文件差不多，只是创建时不能像普通文件那样直接用 open()创建，而是需要使用 mkfio()函数。函数 mkfio()的函数原型如下：

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char * pathname, mode_t mode);
```

函数 mkfifo()会依参数 pathname 创建特殊的 FIFO 文件（如果已经存在则创建失败），而参数 mode 为该文件的权限 （umask 值也会影响到 FIFO 文件的权限）。mkfifo()建立的 FIFO 文件后，本进程或其他进程都可以用读写一般文件的方式存取。关于文件访问权限和读写操作可以参见第 6 章。

4.1.2. System V IPC

Linux 的进程通信继承了 System V IPC。System V IPC 指的是 AT&T 在 System V.2 发行版中引入的三种进程间通信工具:

(1)信号量，用来管理对共享资源的访问

(2)共享内存，用来高效地实现进程间的数据共享

(3)消息队列，用来实现进程间数据的传递。

我们把这三种工具统称为System V IPC的对象，每个对象都具有一个唯一的IPC标识符ID。为了使不同的进程能够获取同一个 IPC 对象，必须提供一个 IPC 关键字（IPC key），内核负责把 IPC 关键字转换成 IPC 标识符 ID。下面我们观察这三种 IPC 工具。

在 Linux 中执行 ipcs 命令可以查看到当前系统中所有的 System V IPC 对象，如屏显 4-3 所示。此时系统中还没有创建消息队列和信号量数组（或称信号量集），有 3 段共享内存区。

屏显 4-3 ipcs 命令的输出

```
--------- 消息队列 -----------
键         msqid      拥有者  权限     已用字节数 消息

------------ 共享内存段 --------------
键         shmid      拥有者  权限     字节     nattch     状态
0x00000000 131072     lqm        600        524288     2          目标
0x00000000 163841     lqm        600        4194304    2          目标
0x00000000 327682     lqm        600        4194304    2          目标
0x00000000 524292     lqm        600        2097152    2          目标

--------- 信号量数组 -----------
键         semid      拥有者  权限     nsems

[lqm@localhost ~]$
```

查看这些 IPC 对象时还可以带上参数，ipcs -a 是默认的输出全部信息、ipcs -m 显示共享内存的信息、ipcs -q 显示消息队列的信息、ipcs -s 显示信号量集的信息。另外用还有一些格式控制的参数，–t 将会输出带时间信息、-p 将输出进程 PID 信息、-c 将输出创建者/拥有者的 PID、-l 输出相关的限制条件。例如用 ipcs -ql 将显示消息队列的限制条件，如屏显 4-4 所示。

屏显 4-4 ipcs –ql 的输出

```
[lqm@localhost ~]$ ipcs -ql

---------- 消息限制 -----------
系统最大队列数量 = 1982
最大消息尺寸（字节）= 8192
默认的队列最大尺寸（字节）= 16384

[lqm@localhost ~]$
```

删除这些 IPC 对象的命令是 ipcrm，它会将与 IPC 对象及其相关联的数据也一起删除， 管理员或者 IPC 对象的创建者才能执行删除操作。该命令可以使用 IPC 键或者 IPC 的 ID 来指定 IPC 对象：ipcrm -M shmkey 删除用 shmkey 创建的共享内存段而 ipcrm -m shmid 删除用 shmid 标识的共享内存段、ipcrm -Q msgkey 删除用 msqkey 创建的消息队列而 ipcrm -q msqid 删除用 msqid 标识的消息队列、ipcrm -S semkey 删除用 semkey 创建的信号而 ipcrm -s semid 删除用 semid 标识的信号。

消息队列

消息队列有些项邮政中的邮箱，里面的消息有点像信件——有信封以及写有内容的信纸。由于各条消息可以通过类型（type）进行区分，因此可以用于多个进程间通信。比如一个任务分派进程，创建了若干个执行子进程，不管是父进程发送分派任务的消息，还是子进程发送任

务执行的消息，都将 type 设置为目标进程的 PID，目标进程只接收消息类型为 type 的消息就实现了子进程只接收自己的任务，父进程只接收任务结果。

下面给出一个多功能的消息队列操作程序 msgtool.c（代码 4-2），读者从中学习如何使用消息队列来进行进程间通信。由于 msgtool 每次启动都是以新的进程形式运行，因此各次运行间使相互独立的，为了能访问到指定的消息队列，我们需一个外部的“键”转换为内部的消息队列的 ID，例如此处使用的是当前目录“.”并通过 frok()转换成内部 ID。msgtool.c 主函数中首先打开(或创建)一个消息队列 msgget key, IPC_CREAT|0660)，第一个参数就是前面通过 ftok()将键转换而来的消息队列 ID，第二个参数类似于文件打开的参数——IPC_CREAT 表示消息队列若是还不存在则创建一个新的、0660 表示创建者及其同组用户可以读（收）也可以写（发）消息。然后根据命令行参数调用不同操作函数，如果是“s”则调用 send_message()发送一条消息，“r”则调用 read_message()接受一条消息 ，如果是“d”则调用 remove_queue()删除指定的消息队列， 最后如果是“m”则调用 change_queue_mode()改变消息队列的访问模式。

其中发送消息的核心函数是 msgsnd()，第一个参数是消息队列的 ID，第二个参数时被发送消息的起始地址（消息的第一个成员是一个整数用于指出消息类型），第三个参数时消息长度，第四个参数指定写消息时的一些行为（此例子用 0）；接受消息的函数是 msgrcv()，第一个参数用于指定消息队列的 ID，第二个参数是接受缓冲区地址，第三个参数指出希望接受的消息类型（0 表示接受任意类型的一条消息，>0 表示接受指定类型的消息，<0 则表示接受类型数值小于该数字绝对值的一条消息）；删除和修改访问模式都是使用 msgctl()（分别指出操作为 IPC_RMID 或 IPC_SET）。

代码 4-2 msgtool.c 代码

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

#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_SEND_SIZE 80

struct mymsgbuf {                               //消息的结构体
        long mtype;                             //消息类型
        char mtext[MAX_SEND_SIZE];              //消息内容
};

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);

int main(int argc, char *argv[])
{
        key_t key;
        int msgqueue_id;
        struct mymsgbuf qbuf;

        if(argc == 1)
```

```
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 'm');

        /* Open the queue - create if necessary */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1)

        {
                perror("msgget");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case    's':    send_message(msgqueue_id,    (struct    mymsgbuf
*)&qbuf,atol(argv[2]), argv[3]);
                break;
                case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                break;
                case 'd': remove_queue(msgqueue_id);
                break;
                case 'm': change_queue_mode(msgqueue_id, argv[2]);
                break;

                default: usage();

        }

        return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Send a message to the queue */
        printf("Sending a message \n");
        qbuf->mtype = type;                     //填写消息的类型
        strcpy(qbuf->mtext, text);              //填写消息内容

        if((msgsnd(qid, (struct msgbuf *)qbuf,
                strlen(qbuf->mtext)+1, 0)) ==-1)
        {
                perror("msgsnd");
                exit(1);
        }
        return;
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
        /* Read a message from the queue */
        printf("Reading a message \n");
        qbuf->mtype = type;
        msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);

        printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
        return;
}
```

```
void remove_queue(int qid)
{
        /* Remove the queue */
        msgctl(qid, IPC_RMID, 0);
        return;
}

void change_queue_mode(int qid, char *mode)
{
        struct msqid_ds myqueue_ds;

        /* Get current info */
        msgctl(qid, IPC_STAT, &myqueue_ds);

        /* Convert and load the mode */
        sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);

        /* Update the mode */
        msgctl(qid, IPC_SET, &myqueue_ds);
        return;
}

void usage(void)
{
        fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
        fprintf(stderr, "nUSAGE: msgtool (s)end \n");
        fprintf(stderr, "        msgtool (r)ecv \n");
        fprintf(stderr, "        msgtool (d)elete\n");
        fprintf(stderr, "        msgtool (m)ode \n");
        exit(1);
}
```

下面我们来执行 msgtool s 1 Hello,my_msg_queue!以便发送类型为 1 的消息，然后用 ipcs –q 查看到新创建了一个消息队列(ID 为 0x6d00e3e3)，里面有 20 个字节(Hello,my_msg_queue!)的 1 条消息。此时再执行 msgtool –r 1（是另一个进程了） 读走类型为 1 的消息，然后再用 ipcs –q 可以看到该消息队列为空（0 字节）了。上述操作的输出如屏显 4-5 所示。

屏显 4-5 msgtool 的执行结果

```
[lqm@localhost ~]$ msgtool s 1 Hello,my_msq_queue!
Sending a message
[lqm@localhost ~]$ ipcs -q

--------- 消息队列 -----------
键        msqid      拥有者  权限     已用字节数 消息
0x6d00e3e3 0          lqm        660        20           1

[lqm@localhost ~]$ msgtool r 1
Reading a message
Type: 1 Text: Hello,my_msq_queue!
[lqm@localhost ~]$ ipcs -q

--------- 消息队列 -----------
键        msqid      拥有者  权限     已用字节数 消息
0x6d00e3e3 0          lqm        660        0            0

[lqm@localhost ~]$
```

另外，读者可以执行 msgtool r 删除指定的消息队列，或者用 msgtool m 修改该消息队列的访问模式（例如用 666 允许任意进程使用该消息队列）。

共享内存

System V IPC 的共享内存是由内核提供的一段内存，可以映射到多个进程的续存空间上，从而通过内存上的读写操作而完成进程间的数据共享。我们首先来看看如何创建共享内存的，示例代码如代码 4-3 所示，它创建了一个 4096 字节的共享内存区。shmget()的第一个参数 IPC_PRIVATE（=0，表示创建新的共享内存），第二个参数是共享内存区的大小，第三个是访问模式。虽然也可以像前面的消息队列的例子那样通过 ftok()将键值转换成 ID，但这里没有指定 ID，而是创建共享内存后由系统返回一个 ID 值（后面的进程要使用该共享内存时需要指定该 ID）。

代码 4-3 shmget-demo.c 代码

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>

#define BUFSZ 4096

int main ( void )
{
        int shm_id;

        shm_id=shmget(IPC_PRIVATE, BUFSZ, 0666 ) ;    //创建共享内存
        if (shm_id < 0 ) {
                perror( "shmget fail!\n" ) ;
                exit ( 1 );
        }

        printf ( "Successfully created segment : %d \n", shm_id ) ;
        system( "ipcs -m");                 //执行 ipcs  -m 命令，显示系统的共享内存信息
        return 0;
}
```

执行 shmget-demo 程序，其输出入所示。输出结果表明新创建的共享内存的 ID 为 819203，长度为 4096 字节，当前还没有进程将他映射到自己的进程空间（nattch 列为 0）。

屏显 4-6 shmget-demo 的输出

```
[lqm@localhost ~]$ shm-demo
Successfully created segment : 884739

------------ 共享内存段 --------------
键         shmid      拥有者  权限     字节     nattch     状态
0x00000000 131072     lqm        600        524288     2          目标
0x00000000 163841     lqm        600        4194304    2          目标
0x00000000 327682     lqm        600        4194304    2          目标
0x00000000 884739     lqm        666        4096       0
0x00000000 524292     lqm        600        2097152    2          目标
```

下面展示另一个进程通过影射该共享内存而使用它的过程，具体如代码 4-4 所示。

代码 4-4 shmatt-write-demo.c 代码

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main ( int argc, char *argv[] )
{
        int shm_id ;
        char * shm_buf;

        if ( argc != 2 ){
                printf ( "USAGE: atshm <identifier>" );
                exit (1 );
        }

        shm_id = atoi(argv[1]);

        if ( (shm_buf = shmat( shm_id, 0, 0)) < (char *) 0 ){  //映射共享内存到进程空间
                perror ( "shmat fail!\n" );
                exit (1);
        }

        printf ( " segment attached at %p\n", shm_buf );
        system("ipcs -m");                                      //显示共享内存信息
        strcpy(shm_buf,"Hello shared memory!\n");
        getchar();

         if ( (shmdt(shm_buf)) < 0 ) {                          //解除共享内存的映射
                perror ( "shmdt");
                exit(1);
        }

        printf ( "segment detached \n" );
        system ( "ipcs -m " );

        getchar();
        exit ( 0 );
}
```

我们运行 shatt-demo 819203（命令行参数中指出共享内存的 ID 为 819203），其第一段输出结果如屏显 4-7 所示。完成共享内存的映射后，shmatt-write-demo 往共享内存中写入一个字符串“Hello shared memory!”。shmatt-write-demo 还通过 system()执行了“ipcs -m”，因此也输出了当前的共享内存信息，可以看到 ID 为 819203 的共享内存已经有被映射了一次（nattach 列为 1）。

屏显 4-7 shmatt-write-demo 运行时的输出（1）

```
[lqm@localhost ~]$ shmatt-write-demo 884739
 segment attached at 0x7f89e55fc000

------------ 共享内存段 --------------
键         shmid      拥有者  权限     字节     nattch     状态
0x00000000 131072     lqm        600        524288     2          目标
0x00000000 163841     lqm        600        4194304    2          目标
0x00000000 327682     lqm        600        4194304    2          目标
0x00000000 884739     lqm        666        4096       1
```

```
0x00000000 524292     lqm        600        2097152    2          目标
```

屏显 4-7 第二行信息是该进程将共享内存映射到了进程空间 0x7f10ddf3a000 位置的地方，此时在另外一个终端上观察该进程的进程空间，可以看到相应位置有一个新的区域（7f89e55fc000-7f89e55fd000）。

屏显 4-8 shmatt-write-demo 映射共享内存时的进程布局

```
[lqm@localhost ~]$ ps -a
  PID TTY          TIME CMD
20138 pts/2    00:00:00 shmatt-write-de
20186 pts/0    00:00:00 ps
[lqm@localhost ~]$ cat /proc/20138/maps
00400000-00401000 r-xp 00000000 fd:00 14409889                           /home/lqm/shmatt-write-demo
00600000-00601000 r--p 00000000 fd:00 14409889                           /home/lqm/shmatt-write-demo
00601000-00602000 rw-p 00001000 fd:00 14409889                           /home/lqm/shmatt-write-demo
7f89e501e000-7f89e51d4000 r-xp 00000000 fd:00 262222                     /usr/lib64/libc-2.17.so
7f89e51d4000-7f89e53d4000 ---p 001b6000 fd:00 262222                     /usr/lib64/libc-2.17.so
7f89e53d4000-7f89e53d8000 r--p 001b6000 fd:00 262222                     /usr/lib64/libc-2.17.so
7f89e53d8000-7f89e53da000 rw-p 001ba000 fd:00 262222                     /usr/lib64/libc-2.17.so
7f89e53da000-7f89e53df000 rw-p 00000000 00:00 0
7f89e53df000-7f89e53ff000 r-xp 00000000 fd:00 262215                     /usr/lib64/ld-2.17.so
7f89e55e7000-7f89e55ea000 rw-p 00000000 00:00 0
7f89e55fa000-7f89e55fc000 rw-p 00000000 00:00 0
7f89e55fc000-7f89e55fd000 rw-s 00000000 00:04 884739                     /SYSV00000000 (deleted)
7f89e55fd000-7f89e55fe000 rw-p 00000000 00:00 0
7f89e55fe000-7f89e55ff000 r--p 0001f000 fd:00 262215                     /usr/lib64/ld-2.17.so
7f89e55ff000-7f89e5600000 rw-p 00020000 fd:00 262215                     /usr/lib64/ld-2.17.so
7f89e5600000-7f89e5601000 rw-p 00000000 00:00 0
7fff9b6e6000-7fff9b707000 rw-p 00000000 00:00 0                          [stack]
7fff9b7cf000-7fff9b7d1000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[lqm@localhost ~]$
```

击键回车后 shmat-write-demo 将解除共享内存的映射，此时 ipcs –m 显示对应的共享内存区没有人使用（nattch 列为 0），如屏显 4-9 所示。此时如果检查进程布局，将发现 7f89e55fc000-7f89e55fd000 区间的虚存已经没有了。

屏显 4-9 shmatt-demo 运行时的输出（2）

```
...
segment detached

------------ 共享内存段 --------------
键         shmid      拥有者  权限     字节     nattch     状态
0x00000000 131072     lqm        600        524288     2          目标
0x00000000 163841     lqm        600        4194304    2          目标
0x00000000 327682     lqm        600        4194304    2          目标
0x00000000 884739     lqm        666        4096       0
0x00000000 524292     lqm        600        2097152    2          目标
...
```

此时再尝试用另一个程序去映射该共享内存并从中读取数据，如代码 4-5 所示。

代码 4-5 shmatt-read-demo.c 代码

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>

int main ( int argc, char *argv[] )
{
        int shm_id ;
        char * shm_buf;

        if ( argc != 2 ){
                printf ( "USAGE: atshm <identifier>" );
                exit (1 );
        }

        shm_id = atoi(argv[1]);

        if ( (shm_buf = shmat( shm_id, 0, 0)) < (char *) 0 ){
                perror ( "shmat fail!\n" );
                exit (1);
        }

        printf ( " segment attached at %p\n", shm_buf );
        system("ipcs -m");
        printf("The string in SHM is :%s\n",shm_buf); //将共享内存区的内容打印出来
        getchar();

         if ( (shmdt(shm_buf)) < 0 ) {
                perror ( "shmdt");
                exit(1);
        }

        printf ( "segment detached \n" );
        system ( "ipcs -m " );

        getchar();
        exit ( 0 );
}
```

虽然创建该共享内存的进程已经结束了，可是 shmatt-read-demo 映射 ID 为的共享内存后，仍读出了原来写入的字符串，如所示。

屏显 4-10 shmatt-read-demo 的部分输出

```
[lqm@localhost ~]$ shmatt-read-demo 884739
 segment attached at 0x7f1f35ade000

------------ 共享内存段 --------------
键         shmid      拥有者  权限     字节     nattch     状态
0x00000000 131072     lqm        600        524288     2          目标
0x00000000 163841     lqm        600        4194304    2          目标
0x00000000 327682     lqm        600        4194304    2          目标
0x00000000 884739     lqm        666        4096       1
0x00000000 524292     lqm        600        2097152    2          目标

The string in SHM is :Hello shared memory!
…
```

从上面实验看出共享内存是比较灵活的通信方式，不需要像管道那要用文件接口 read()、write()等函数，也不需要像消息队列那样用 msgsend()/msgrcv()等函数来操作，直接用内存指针方式就可以操作。虽然实验中没有验证其容量，但是共享内存的容量远比管道和消息队列大。

信号量数组/信号量集

批注 [PR12]: 需要补充完整

在操作系统原理性课程中我们以及学习过信号量和信号量集机制。Linux 支持的 System V IPC 中的信号量实际上是信号量数组（信号量集），一次可以创建多个信号量。创建或者获得信号量集之后，可以对各个信号量进行 P/V 操作（或者称 up/down 操作），进程进行 P/V 操作时遵循信号的同步约束关系——由操作系统完成进程的阻塞或唤醒。下面我们来感受 Linux 编程中信号量集上的同步操作。

4.2. 进程间同步

Linux 同时支持 System V IPC 中的信号量集和 POSIX 信号量。前者常用于进程间通信、是基于内核实现的（不随进程结束而消失）；而后者是常用于线程间同步、方便使用且仅含一个信号量。POSIX 信号量分成有名信号量和无名信号量，前者和一个文件的路径名相关联，创建后不随进程结束而消失（可用于进程间通信），反之无名信号量则只在进程生命周期内存在且只能在该进程创建的线程间使用。

上述两种信号量的编程接口函数是很容易被区分：对于所有 System V 信号量函数，在它们的名字里面没有下划线（例如，有 semget()而不是 sem_get()），然而所有的的 POSIX 信号量函数都有一个下划线（例如，有 sem_post()而不是 sempost()）。

Linux 操作系统内核内部也有多个并发的执行流，它们之间使用内核的信号量，和这里讨论的用户态信号量又不相同。

4.2.1. System V IPC 信号量集

进程间的 System V IPC 信号量集的同步机制已经在前面 System V IPC 中和进程间通信主题（4.1.2 小节）一并讨论过，这里不再重复。

4.2.2. POSXI 信号量

POSIX 信号量又分成有名信号量和无名信号量，前者可以用于在多个进程间或多个线程间的同步，无名信号量只能用于线程间同步。两者的创建函数不同，但是相应的 P/V 操作（up/down）操作函数是一样的。

有名信号量

有名信号量由于可以通过标识来访问，因此可以同时用于进程间同步和线程间同步。下面我们来看看 POSIX 有名信号的使用过程和进程呈现出来的同步关系。有名信号量的创建使用 sem_open()完成，其函数原型如下：

```
#include <semaphore.h>
sem_t *sem_open(const char *name,int oflag,mode_t mode,unsigned int value);
```

代码 4-6 的 psem-named-open.c 中先用 sem_open()创建了一个信号量，该信号量由一个字符串所标识（代码忠是从命令行读入的一个文件名字符串），代码中使用了 O_CREAT 标志

（如果信号量还不存在则创建它）并将信号量初值置为 1。注意，出于代码的简洁考虑，并没有对 sem_open()的返回值进行处理，因此可能隐含创建失败的情况。

代码 4-6 psem-named-open.c 代码

```
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
        sem_t *sem;

        if(argc!=2)
        {
                printf("please input a file name to act as the ID of the sem!\n");
                exit(1);
        }
        sem=sem_open(argv[1],O_CREAT,0644,1);          //创建一个命名的 POSIX 信号量
        exit(0);
}
```

然后用 gcc psem-named-open.c –o psem-named-open –lthread（参数-lpthread 用于指出链接时所用的线程库）完成编译，然后运行 psem-named-open()。如果没有输入作为标识的文件名字符串，则给出体系要求用户输入；如果输入一个文件名字符串，正常情况将完成创建过程，如屏显 4-11 所示。

屏显 4-11 psem-named-open-demo 的输出

```
[lqm@localhost ~]$ gcc psem-named-open-demo.c -o psem-named-open-demo -lpthread
[lqm@localhost ~]$ psem-named-open-demo
please input a file name to act as the ID of the sem!
[lqm@localhost ~]$ psem-named-open-demo HelloWorld.c
[lqm@localhost ~]$
```

然后，我们来尝试执行 P/V 操作中的 V 操作（即对信号量进行减 1 操作，可能引发阻塞），程序如代码 4-7 所示。它通过 sem_wait()来执行 V 操作（减 1 操作），并且通过 sem_getvalue()来查看信号量的值。同样出于代码简洁的考虑，这里的代码也是没有检查 sem_open()是否成功获得了信号量。因此，如果输入错误的标识字符串，则无法成功获得所指定的信号量，sem_wait()引用无效的信号量而引发段错误。

代码 4-7 psem-named-wait-demo.c

```
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
        sem_t *sem;
        int val;

        if(argc!=2)
        {
                printf("please input a file name!\n");
```

```
        exit(1);
    }
    sem=sem_open(argv[1],0);                    获取信号量对象
    sem_wait(sem);                              执行 P 操作（-1 操作）
    sem_getvalue(sem,&val);                     获得出当前信号量的值
    printf("pid %ld has semaphore,value=%d\n",(long)getpid(),val);
    return 0;
}
```

编译并执行 psem-named-wait-demo，输入前面创建信号量时使用的文件名标识（屏显 4-11 中输入的 HelloWorld.c），此时打印出当前信号量值为 0（也就是说前面创建的时候初值是 1）。如果在运行一遍，此时信号量的值已经为 0，在进行 V 操作（减 1 操作）将阻塞该进程。这两次运行的情况如图 4-5 所示。

图 4-5 psem-named-wait-demo 的运行输出

图 4-5 显示 psem-named-wait-demo 第二次运行后并没有返回到 shell 提示符，如果此时用另一个中断执行 ps 命令可以看到该进程处于 S 状态。

屏显 4-12ps 查看 psem-named-wait-demo 的运行状态

```
[lqm@localhost ~]$ ps aux |grep psem-named-wait-demo
lqm       9333  0.0  0.0   6452   376 pts/2    S+   15:24   0:00 psem-named-wait-demo HelloWorld.c
lqm       9548  0.0  0.0 112664   976 pts/3    R+   15:36   0:00 grep --color=auto psem-named-wait-demo
[lqm@localhost ~]$
```

再接着来看看对该信量进行 P 操作（增 1 操作），使得前面的 psem-named-wait-demo 进程从原来的阻塞状态唤醒并执行结束。程序如代码 4-8 所示，这里也要注意代码并没有对 sem_open()的返回值进行判定，因此输入错误的文件标识时隐含出现段错误的可能。

代码 4-8 psem-named-post-demo.c

```
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
        sem_t *sem;
        int val;

        if(argc!=2)
        {
                printf("please input a file name!\n");
                exit(1);
        }
        sem=sem_open(argv[1],0);
        sem_post(sem);                                  //对信号量执行 P 操作（增 1）
        sem_getvalue(sem,&val);
        printf("value=%d\n", val);
```

```
        exit(0);
}
```

编译并执行 psem-named-post-demo（与前面 psem-named-wait-demo 不在同一个终端 shell 上），可以看到此时信号量的值增加到 1，并使得原来阻塞的 psem-named-post-demo 被唤醒并执行完毕，如所示。

图 4-6 psem-named-post-demo 的运行输出（并唤醒阻塞的 psem-named-wait-demo 进程）

最后，如果不希望使用这个信号量时，可以通过 sem_unlink()撤销该信号量，具体可以参考。

代码 4-9 psem-named-unlink-demo.c 代码

```
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
        if(argc!=2)
        {
                printf("please input a file name to act as the ID of the sem!\n");
                exit(1);
        }
        sem_unlink(argv[1]);                    //撤销指定的信号量
        exit(0);
}
```

无名信号量

POSIX 无名信号量适用于线程间通信，如果无名信号量要用于进程间同步，信号量要放在共享内存中（只要该共享内存区存在，该信号灯就可用）。无名信号量和有名信号量的区别主要在创建上，无名信号量使用 sem_init()创建，其函数原型如下：

```
#include<semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

第二个参数 pshared 表示该信号量是否由于进程间通步。当 pshared=0，那么表示该信号量只能用于进程内部的线程间的同步；当 pshared !=0，表示该信号量存放在共享内存区中（例如使用 shmget()），使得引用它的进程能够访问该共享内存区进行进程同步。

如果无名信号量是在单个进程内部的数据空间中，即信号量只能在进程内部的各个线程间共享，那么信号量是随进程的持续性，当进程终止时它也就消失了。如果无名信号量位于不同进程的共享内存区，因此只要该共享内存区仍然存在，该信号量就会一直存在。

这里需要注意的是，无名信号量不使用任何类似 O_CREAT 的标志，这意味着 sem_init()总是会初始化信号量的值，所以对于特定的一个信号量，我们必须保证只调用 sem_init()进行初始化一次，对于一个已初始化过的信号量调用 sem_init()的行为是未定义的。使用完一个无名

信号量后，调用 sem_destroy 进行撤销。这里要注意的是，撤销一个仍有线程阻塞在其上的信号量的行为是未定义的。

互斥量

互斥量是信号量的一个退化版本，仅用于并发任务间的互斥访问。下面先用一个代码来展示多线程并发且没有用互斥量保护共享变量的情形，如代码 4-10 所示，此时结果可能会出现错误。该程序对一个缓冲区（缓冲区内是数值为 3、4、3、4……交织的整数）内的每个整数进行检查，并对数值为 3 的整数进行计数统计，统计工作由 16 个线程并发完成（每个线程负责缓冲区的 1/16 的数据）。

代码 4-10 no-mutex-demo.c 代码

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <malloc.h>

#define  thread_num 16
#define  MB 1024 * 1024
int *array;
int length;     //array length
int count;
int t;          //number of thread
void *count3s_thread(void* id);

int main()
{
        int i;
        int tid[thread_num];
        pthread_t threads[thread_num];
        length = 64 * MB;
        array = malloc(length*4);                       //256MB
        for (i = 0; i < length; i++)                    //initial array
                array[i] = i % 2 ? 4 : 3;               //偶数 i 对应数值 3

        for (t = 0; t < thread_num; t++)                //循环创建 16 个线程
        {
                count = 0;
                tid[t]=t;
                int err = pthread_create(&(threads[t]), NULL, count3s_thread,&(tid[t]) );
                if (err)
                {
                        printf("create thread error!\n");
                        return 0;
                }
        }

        for (t = 1; t < thread_num; t++)
                        pthread_join(threads[t], NULL);    //等待前面创建的计数线程结束

        printf("Total count= %d \n",count);
        return 0;
}

void *count3s_thread(void* id)                          //计数线程执行的函数
```

```
{
        /*compute portion of the array that this thread should work on*/
        int length_per_thread = length / thread_num;    //length of every thread
        int start = *(int *)id * length_per_thread;     //every thread start position
        int i;

        for (i = start; i < start + length_per_thread; i++)
        {
                if (array[i] == 3)
                {
                        count++;                                //计数，为加入互斥保护
                }
        }
}
```

编译后运行 no-mutex-demo（注意编译时要有-lpthread 参数指出所需的线程库），得到屏显 4-13 所示的输出，每次运行结果并不唯一（共享变量未能得到互斥访问）。

屏显 4-13 mutex-demo 的输出

```
[lqm@localhost ~]$ gcc no-mutex-demo.c  -lpthread -o no-mutex-demo
[lqm@localhost ~]$ no-mutex-demo
Total count= 33554432
[lqm@localhost ~]$ no-mutex-demo
Total count= 31457280
[lqm@localhost ~]$ no-mutex-demo
Total count= 33554432
[lqm@localhost ~]$
```

如果对 count++这个临界区加以保护，就能避免出现这个错误。

代码 4-11 no-mutex-demo.c 代码

```
#include <pthread.h>
  #include <unistd.h>
  #include <stdio.h>
  #include <malloc.h>

  #define  thread_num 16
  #define  MB 1024 * 1024
  int *array;
  int length;     //array length
  int count;
  int t;          //number of thread
  void *count3s_thread(void* id);

  mutex m;                                  //增加一个互斥量

  int main()
  {
  pthread_mutex_init(&m,NULL);              //初始化互斥量


  ......


  }
  void *count3s_thread(void* id)
  {
```

```
        /*compute portion of the array that this thread should work on*/
        int length_per_thread = length / thread_num;    //length of every thread
        int start = *(int *)id * length_per_thread;     //every thread start position
        int i;

        for (i = start; i < start + length_per_thread; i++)
        {
                if (array[i] == 3)
                {
                        pthread_mutex_lock(&m);             //进入临界区
                        count++;
                        pthread_mutex_unlock(&m);           //推出临界区
                }
        }
}
```

运行 mutex_demo，每次运行都获得相同的结果，如屏显 4-14 所示。由于实现了共享变量的互斥访问，因此每次运行的结构都是确定的值。

屏显 4-14 mutex-demo 的输出

```
[lqm@localhost ~]$ gcc mutex-demo.c  -lpthread -o mutex-demo
[lqm@localhost ~]$ mutex-demo
Total count= 33554432
[lqm@localhost ~]$ mutex-demo
Total count= 33554432
[lqm@localhost ~]$ mutex-demo
Total count= 33554432
……
```

4.3. 小结

本章讨论了 Linux 操作系统上的进程间的通信和同步手段，并进行了最基本的编程实践。读者在实际应用中，需要能有选择地使用这些通信和同步手段，并能将这些基本技能综合应用以解决多进程/多线程并发程序的具体问题。编写并发的服务器程序是进程间通信和同步的典型应用之一，必须综合多种通信和同步手段。同时要注意到，Linux 的进程间通信中实际上已经实现了必要的同步，例如在读空的管道或消息队列时进程会阻塞等等。

4.4. 练习

1. 设计编写以下程序，着重考虑其通信和同步问题：

a）一个程序（进程）从命令行读入按键信息，一次将“一整行”按键信息保存到一个共享存储的缓冲区内并等待读取进程将数据读走，不断重复上面的操作；

b）另一个程序（进程）生成两个进程/线程用于显示缓冲区内的信息，这两个线程并发读取缓冲区信息后将缓冲区清空（一个线程的两次“读取+ 显示”操作之间可以加入适当的时延以便于观察）。

c）在两个独立的终端窗口上分别运行上述两个程序，展示其同步与通信功能，要求一次只有一个任务在操作缓冲区。

d） 运行程序，记录操作过程的截屏并给出文字说明。

要求使用 posix 信号量来完成这里的生产者和消费者的同步关系。

2. 将 2.6 的练习 3 进行扩展，使得你编写的 shell 程序具有支持管道的功能，也就是说你的 shell 中输入“dir |=| more”能够执行 dir 命令并通过管道将其输入传送给 more。
3. 2. 将 2.6 的练习 3 进行扩展（例如，实现输入输出重定向？自定义的？消息队列？）

第5章 内存管理

注意区分虚拟内存空间、物理内存以及连续内存管理方式、离散内存管理方式几个概念的关系。

5.1. 分页机制

5.2. 虚存空间管理

5.2.1. 分配与释放

观察 helloworld.c VMA 改变，内存统计数据改变，页表所占空间（/proc/.../ VmPTE）

5.2.2. 文件映射内存

创建文件映射内存，体验 VMA 上的变化

5.2.3. 查看进程页表

5.3. 物理页帧管理

5.3.1. 页表映射

http://lwn.net/Articles/267837/　　google 关键字 dump page table

读取其他进程的内存 http://blog.csdn.net/guojin08/article/details/9454467

缺页 perf

5.3.2. 物理内存

刚分配的虚存空间进行写操作前后，进程脏页（放到物理内存），观察 buddy 变化

刚分配的文件映射页进行改写前后的变化，脏页

5.3.3. 内存回收与交换