操作系统实验 2：进程和进程通信

王靖康
515030910059
网络空间安全学院
wangjksjtu_01@sjtu.edu.cn

1 实验题目

1. 设计一个程序，创建一个子进程，使父子进程合作，协调地完成某一功能。要求在该程序中还要使用进程的睡眠、进程图象改换、父进程等待子进程终止、信号的设置与传送（包括信号处理程序）、子进程的终止等有关进程的系统调用。

2. 分别利用 UNIX 的消息通信机制、共享内存机制（用信号灯实施进程间的同步和互斥）实现两个进程间的数据通信。具体的通信数据可从一个文件读出，接收方进程可将收到的数据写入一个新文件，以便能判断数据传送的正确性。

2 实验目的

- 理解进程和进程通信的原理和相关机制;
- 提高用 C 语言编制程序的能力，熟悉标准库函数 API 接口;
- 加深基本概念理解，掌握系统 V 的 IPC 机制。

3 进程通信

进程间通信是指在不同进程之间传播或交换信息。IPC 是一组编程接口，使得进程可以实现协作，包括数据传输、共享数据、通知事件、资源共享、进程控制等功能，从而完成用户多样化的要求。常见的进程间通信方法包括:1) 管道通信 (PIPE); 2) 消息通信; 3) 共享存储区; 4) 套接字 (Socket) 通信。

3.1 管道通信

管道实际是用于进程间通信的一段共享内存，创建管道的进程称为管道服务器，连接到一个管道的进程为管道客户机。一个进程在向管道写入数据后，另一进程就可以从管道的另一端将其读取出来。管道按照进程关系，可分为有名管道和无名管道。另外，管道只能在本地计算机中使用，而不可用于网络间的通信。

3.2 消息通信

消息队列是内核地址空间中的内部链表，通过 linux 内核在各个进程直接传递内容，消息顺序地发送到消息队列中，并以几种不同的方式从队列中获得，每个消息队列可以用 IPC 标识符唯一地进行识别。内核中的消息队列是通过 IPC 的标识符来区别，不同的消息队列直接是相互独立的。每个消息队列中的消息，又构成一个独立的链表。**消息队列克服了信号承载信息量少，管道只能承载无格式字符流的缺点**。

3.3 共享存储区

共享内存是在多个进程之间共享内存区域的一种进程间的通信方式，由 IPC 为进程创建的一个特殊地址范围，它将出现在该进程的地址空间中。其他进程可以将同一段共享内存连接到自己的地址空间中。所有进程都可以访问共享内存中的地址，就好像它们是 malloc 分配的一样。如果一个进程向共享内存中写入了数据，所做的改动将立刻被其他进程看到。

本次实验使用 C 语言在类 Unix 系统完成父子进程通信实验，消息通信机制和共享内存机制（使用信号灯实现进程的同步和互斥）实现的进程间数据通信。

4 父子进程通信

在 linux 中，在程序中创建子进程通过调用 fork() 函数来实现。fork() 函数通过系统调用创建一个与原来进程几乎完全相同的进程，这个新产生的进程称为子进程。

一个进程调用 fork() 函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。父进程和子进程共享代码段而数据空间、堆、栈等各自独立。

4.1 编程接口介绍

fork() 函数: pid_t fork(void); 在 C 程序中，调用 fork 函数，将重新克隆一个进程（少量数据不同，如 pid），每个进程均从该代码位置开始执行。若成功则对与子进程返回 0，对父进程返回子进程的进程标识符（pid）。若出错则返回-1。值得注意的是，若父进程先与子进程结束，则子进程归 init 进程管理。

getpid()/getppid() 函数: pid_t getpid(void); pid_t getppid(void); getpid 返回当前进程标识，getppid 返回父进程标识。

4.2 模块设计实现

根据实验题目的要求和参考代码，设计了父子进程合作通信，在程序中使用了进程的睡眠、进程图像改换、父进程等待子进程终止（设置等待时限，若规定时间内子进程未结束，则父进程给子进程发送进程终止信号）、系统程序调用，程序流程如图 1 所示。

父进程: 依次完成以下功能: 1) 打印父进程 pid; 2) 复制 test.in 文件到 test.out 文件; 3) 进程睡眠 3s; 4) 向子进程发送完成信号; 5) 向子进程发送自定义信号（打印当前日期）; 6) 进程睡眠 1s; 7) 等待子进程结束，若未结束，向子进程发送结束信号。6) 打印子进程的进程号和状态。

```
1 printf("[Child]:␣This␣is␣the␣child␣process␣(ID:␣%d).\n", getpid());
```

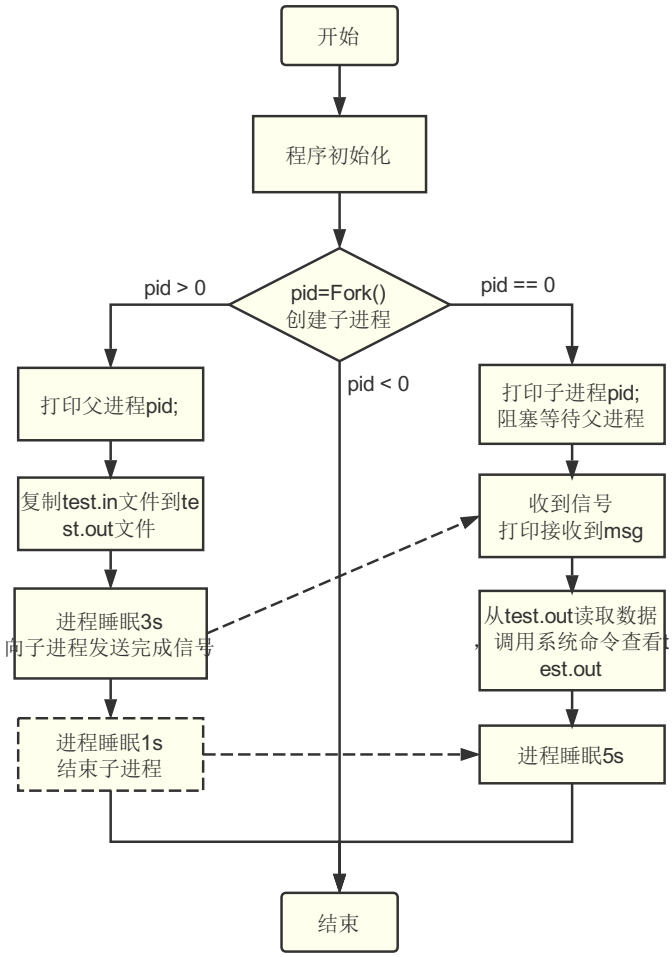

Figure 1: 父子进程通信流程图

```
pause();
printf("[Child]: Signal is received.\n");
char buffer[BUFFER_SIZE];
FILE *fp = fopen(outfile, "r");
if (fread(buffer, sizeof(char), 10, fp) > 0)
    printf("[Child]: Read from file `%s`: %s\n", outfile, buffer);
execl("/bin/ls", "ls", "-al", "test.out", (char*)0);
printf("execl error.\n");
printf("[Child]: Start sleeping.\n");
sleep(5);
exit(1);
```

子进程: 依次完成以下功能: 1）打印子进程 pid; 2）阻塞，等待父进程传送信号; 3) 打印接收到信号; 4) 从 test.out 文件读取一个 block 的数据; 5) 调用系统程序查看 test.out 文件的详细信息; 6) 进程睡眠 5s。

```
printf("[Child]: This is the child process (ID: %d).\n", getpid());
pause();
```

```
3  printf("[Child]:␣Signal␣is␣received.\n");
4  char buffer[BUFFER_SIZE];
5  FILE *fp = fopen(outfile, "r");
6  if (fread(buffer, sizeof(char), 10, fp) > 0)
7      printf("[Child]:␣Read␣from␣file␣`%s`:␣%s\n", outfile, buffer);
8  execl("/bin/ls", "ls", "-al", "test.out", (char*)0);
9  printf("execl␣error.\n");
10 printf("[Child]:␣Start␣sleeping.\n");
11 sleep(5);
12 exit(1);
```

4.3 功能测试

在该程序中完成了父子进程的多次信号通信，进程睡眠，进程图像改换等: 1) 子进程等待父进程写操作完毕再进行读操作; 2) 父进程等待子进程结束; 3) 父进程发送进程终止信号。

如图 2 所示，程序成功完成代码 3.2 的功能，进而完成了题目父子进程通信的要求。

```
wangjk@asus-wjk:~/programs/OS/lab2$ ./fork
[Father]: This is the parent process.(ID: 12799).
[Father]: Cope 'test.in' to 'test.out'.
[Child]: This is the child process (ID: 12800).
[Father]: Send the signal.
[Father]: Writing finished
Tue May 29 11:06:53 UTC 2018
[Child]: Signal is received.
[Child]: Read from file `test.out`: G9MluBHZuM
-rw-rw-r-- 1 wangjk wangjk 11473 May 29 19:06 test.out
[Father]: Child process -1, status=1
[Father]: Parent process will terminate.
```

Figure 2: 父子进程通信

5 消息队列机制

消息队列提供了一种从一个进程向另一个进程发送一个数据块的方法。每个数据块都被认为含有一个类型，接收进程可以独立地接收含有不同类型的数据结构。可以通过发送消息来避免命名管道的同步和阻塞问题。Linux 提供了一系列消息队列的函数接口来让我们方便地使用它来实现进程间的通信。它的用法与其他两个 System V PIC 信号量和共享内存相似。

5.1 编程接口介绍

msgget 函数: int msgget(key_t, key, int msgflg); 该函数用来创建和访问一个消息队列, 其中第一个参数为非负整数，用于命名消息队列。第二个参数 msgflg 是一个权限标志，表示消息队列的访问权限，它与文件的访问权限一样。它返回一个以 key 命名的消息队列的标识符（非零整数），失败时返回-1。

msgsnd 函数: int msgsend(int msgid, const void *msg_ptr, size_t msg_sz, int msgflg); 该函数用来把消息添加到消息队列，其中第一个参数 msgid 是由 msgget 函数返回的消息队列

标识符。第二个参数 msg_ptr 是一个指向准备发送消息的指针，但是消息的数据结构却有一定的要求，指针 msg_ptr 所指向的消息结构一定要是以一个长整型成员变量开始的结构体，接收函数将用这个成员来确定消息的类型。

本次实验的消息结构定义为:

```
struct msg_s {
    long msgtype;  // 消息类型
    int pid;       // 进程标识号
    int length;    // 消息长度
    char text[BUFFER_SIZE];  // 消息字符串
};
```

如果调用成功，消息数据的一分副本将被放到消息队列中，并返回 0，失败时返回-1。

msgrcv 函数: int msgrcv(int msgid, void *msg_ptr, size_t msg_st, long int msgtype, int msgflg); 该函数用来从一个消息队列获取消息，前三个参数和 msgsnd 相同。msgtype 可以实现一种简单的接收优先级。如果 msgtype 为 0，就获取队列中的第一个消息。如果它的值大于零，将获取具有相同消息类型的第一个信息。如果它小于零，就获取类型等于或小于 msgtype 的绝对值的第一个消息。最后一个参数 msgflg 用于控制当队列中没有相应类型的消息可以接收时将发生的事情。调用成功时，该函数返回放到接收缓存区中的字节数，消息被复制到由 msg_ptr 指向的用户分配的缓存区中，然后删除消息队列中的对应消息。失败时返回-1。

5.2 模块设计实现

服务端 (server.c) 服务端代码实现了消息队列中多次分块读入消息，并向客户端发送应答数据（通过 msgsnd 加入消息队列中实现）。

```
while (1) {
    msgrcv(qid, &buf, sizeof(buf.text), 1, MSG_NOERROR);
    printf("Server␣receive␣a␣request␣from␣process␣%d␣(", buf.pid);
    buf.msgtype = buf.pid;
    fwrite(buf.text, sizeof(char), buf.length, out);
    msgsnd(qid, &buf, sizeof(buf.text), 0);
    printf("%d)\n", buf.length);
    if (buf.length < BUFFER_SIZE) break;
}
```

客户端 (client.c) 客户端代码实现了从 test.in 文件中分块多次读取数据，并将数据加入消息队列，并从消息队列接受服务端的应答数据（服务端需要从消息队列分多块中读取数据，并写到本地某文件中）。

```
while ((nread = fread(buffer, sizeof(char), BUFFER_SIZE, in)) > 0) {
    qid = msgget(MSGKEY, IPC_CREAT|0666);
    buf.msgtype = 1;
    buf.pid = pid = getpid();
    strcpy(buf.text, buffer);
    buf.length = nread;
    msgsnd(qid, &buf, sizeof(buf.text), 0);
```

```
8       msgrcv(qid, &buf, 512, pid, MSG_NOERROR);
9       printf ("Request␣received␣a␣message␣from␣server.␣MSG_type␣is:␣%ld\n",
            buf.msgtype);
10  }
```

5.3 功能测试

代码 5.2 通过客户端和服务端配合，实现了客户端向服务端传送文件的功能。若客户端未传数据（未启动），服务端阻塞。当客户端运行程序像消息队列中加入数据时，服务端可与之完成通信。图 3 和图 4 中显示了客户端和服务端的交互过程，使用 diff 命令，检查 test.out 和 test.in 文件内容，完全一致。

```
wangjk@asus-wjk:~/programs/OS/lab2$ ./client
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
Request received a message from server. MSG_type is: 7721
```

Figure 3: 进程消息队列通信（客户端）

```
wangjk@asus-wjk:~/programs/OS/lab2$ ./server
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (1024)
Server receive a request from process 7721 (209)
```

Figure 4: 进程消息队列通信（服务端）

6 共享内存机制

共享内存就是允许两个不相关的进程访问同一个逻辑内存。

值得注意的是，共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取。所以我们通常需要用其他的机制来同步对共享内存的访问，如本实验采用的信号量。

6.1 编程接口介绍

shmget() 函数: int shmget(key_t key, size_t size, int shmflg); 其中, 第一个参数为非零整数 key，利用 key 为共享内存段命名。第二个参数 size 表示以字节为单位指定需要共享的内存容量。第三个参数 shmflg 是权限标志。例如: 0644 表示允许一个进程创建的共享内存被内存创建者所拥有的进程向共享内存读取和写入数据，同时其他用户创建的进程只能读取共享内存。shmget 函数成功时返回一个与 key 相关的共享内存标识符（非负整数），用于后续的共享内存函数。调用失败返回-1。

shmat() 函数: void *shmat(int shm_id, const void *shm_addr, int shmflg); 其中，第一个参数 shm_id 是由 shmget 函数返回的共享内存标识。第二个参数 shm_addr 指定共享内存连接到当前进程中的地址位置，通常为空，表示让系统来选择共享内存的地址。第三个参数 shm_flg 是一组标志位，通常为 0 调用该函数成功时返回一个指向共享内存第一个字节的指针，如果调用失败返回-1。

segmet() 函数: int semget(key_t key, int num_sems, int sem_flags); 其中，第一个参数 key 与 shmget 函数类似，用于创建信号量。第二个参数 num_sems 指定需要的信号量数目。第三个参数 sem_flags 是一组标志，当想要当信号量不存在时创建一个新的信号量，可以和值 IPC_CREAT 做按位或操作。semget 函数成功返回一个相应信号标识符（非零），失败返回-1。

semctl() 函数: int semctl(int sem_id, int sem_num, int command, ...); 其中，第一个参数 sem_id 是由 semget 返回的信号量标识符, 第二个 sem_num 参数除非使用一组信号量，否则为 0。第三个参数 command 通常是下面两值中的一个 SETVAL：用来把信号量初始化为一个已知的值。这个值通过 union semun 中的 val 成员设置，其作用是在信号量第一次使用前对它进行设置。

sembuf 和 semun 数据结构:

```
1  struct sembuf{
2      short sem_num;//除非使用一组信号量，否则它为0
3      short sem_op;//信号量在一次操作中需要改变的数据，通常是两个数，一个是
           -1，即P（等待）操作，
4                    //一个是+1，即V（发送信号）操作。
5      short sem_flg;//通常为SEM_UNDO,使操作系统跟踪信号，
6                    //并在进程没有释放该信号量而终止时，操作系统释放信号
                         量
7  };
8
9  union semun{
10     int val;
11     struct semid_ds *buf;
12     unsigned short *arry;
13 };
```

6.2 模块设计实现

根据实验题目的要求和参考代码，设计了共享内存通信机制。在程序中使用 fork() 创建子进程，父子进程使用信号量 (P/V 操作) 实现了进程的同步和互斥，完成了对于文件的读写功能。

```
    P(sid2);                /* 置信号灯2值为0，表示缓冲区空 */
    if(!(pid = fork())) {
        printf ("%d\n", pid);
        char* outfile = "test.out";

        FILE *out = fopen(outfile, "w");
        P(sid2);
        fwrite(segaddr, sizeof(char), SIZE, out);
        fclose(out);
        V(sid1);
    }
    else {
        printf ("%d\n", pid);
        char* infile = "test.in";
        int length = 0;
        char buffer [SIZE];
        FILE *in = fopen(infile, "r");
        if ((length = fread(buffer, sizeof(char), SIZE, in)) > 0) {
            P(sid1);
            strcpy(segaddr, buffer);
            printf("%d\n", length);
            V(sid2);
        }
    }
```

6.3 功能测试

代码 6.2 使用 fork() 创建了父子进程，创建共享内存，使用信号量实现了父子进程通信，完成文件读写功能。如图 5 所示，成功创建了共享内存，完成了父进程从文件中读出数据放入共享内存区，子进程从共享内存区读出文件，并保存到本地（只完成了一块的读写）。

7 总结与思考

7.1 重点分析

本次实验比较有趣，涉及了 Unix 系统进程通信的几种核心机制，通过实际代码编写可以发现，C 语言对这几种机制的接口提供非常一致，容易掌握。下面总结下本次实验的关键点（重难点）：

- fork() 函数的掌握，理解 fork() 函数的机理，学会区分父子进程。
- 使用信号进行通信 (kill)，理解信号在系统进程管理中的应用。

Figure 5: 共享内存机制

- 消息队列接口掌握（msgsnd，msgrcv，msgget,msgcrt 等）。
- 共享内存的创建 (权限问题) 和使用（shmget，shmat）。
- 信号量的创建和控制（P/V 操作，segmet 和 semctl 函数）。

7.2 个人总结

本次实验总体上说难度适中，但在实验进行过程中代码的调试花费了比较多的时间。经过本实验的练习，使得我门对于进程通信的 4 种手段有了较为全面的了解和认识。同时，使用 C 语言实现父子进程通信、消息队列机制、信号量、共享内存机制等使得我对进程通信的机制和原理更深入的理解，也进一步体会了 C 语言的高效性和一致性，提高了代码编写与调试技巧。最后，十分感谢刘老师上课的指导和帮助，使得我们对进程间通信几种方式有了宏观而整体的把握。

8 附录

8.1 程序源代码

实验 1–父子进程通信

```
1 // fork.c
2 #include <sys/types.h>
3 #include <sys/wait.h>
4 #include <signal.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <unistd.h>
8
9 #define BUFFER_SIZE 1024
```

```
10
11 void copy(char *infile, char *outfile) {
12     char buffer[BUFFER_SIZE];
13
14     FILE *in = fopen(infile, "r");
15     FILE *out = fopen(outfile, "w");
16
17     if (in == NULL || out == NULL) {
18         if (in == NULL) {
19             printf("[Error]: Can not open file %s\n", infile);
20         }
21         else {
22             printf("[Error]: Can not create file %s\n", outfile);
23         }
24         exit(1);
25     }
26     // printf("%s", buffer);
27     int length = 0;
28     while((length = fread(buffer, sizeof(char), BUFFER_SIZE, in)) > 0) {
29         fwrite(buffer, sizeof(char), length, out);
30     }
31
32     fclose(in);
33     fclose(out);
34 }
35
36 void sig_w() {
37     printf("[Father]: Writing finished\n");
38     system("date -u");
39 }
40
41 int main() {
42     int pid, ret, status = 1;
43     void sig_w();
44     char infile [] = "test.in";
45     char outfile [] = "test.out";
46     signal(SIGUSR1, sig_w);
47
48     while ((pid=fork())==-1);
49     if (pid) {
50         printf("[Father]: This is the parent process.(ID: %d).\n", getpid
               ());
51         printf("[Father]: Cope '%s' to '%s'.\n", infile, outfile);
52         copy(infile, outfile);
53         sleep(3);
54         printf("[Father]: Send the signal.\n");
55         kill(pid, SIGUSR1);
56         // pid = wait(&status);
```

```
57          // printf("[Father]: Child process %d, status=%d \n", pid, status
                );
58
59          sleep(1);
60          if ((waitpid(pid, NULL, WNOHANG)) == 0) {
61              if ((ret = kill(pid, SIGKILL)) == 0) {
62                   printf("[Father] Kill child process %d\n", pid) ;
63              }
64          }
65
66          pid = wait(&status);
67          printf("[Father]: Child process %d, status=%d \n", pid, status);
68      } else {
69          printf("[Child]: This is the child process (ID: %d).\n", getpid()
                );
70          pause();
71          printf("[Child]: Signal is received.\n");
72          char buffer[BUFFER_SIZE];
73          FILE *fp = fopen(outfile, "r");
74          if (fread(buffer, sizeof(char), 10, fp) > 0)
75              printf("[Child]: Read from file `%s`: %s\n", outfile, buffer)
                    ;
76          execl("/bin/ls", "ls", "-al", "test.out", (char*)0);
77          printf("execl error.\n");
78          printf("[Child]: Start sleeping.\n");
79          sleep(5);
80          exit(1);
81      }
82      printf("[Father]: Parent process will terminate.\n");
83  }
```

实验 2–消息通信机制

```
1   // msgcom.h
2   #include <errno.h>
3   #include <sys/types.h>
4   #include <sys/ipc.h>
5   #include <sys/msg.h>
6   #include <stdio.h>
7   #include <stdlib.h>
8   #include <unistd.h>
9   #include <string.h>
10
11  #define MSGKEY 5678
12  #define BUFFER_SIZE 1024
13
14  struct msg_s {
15      long msgtype;
16      int pid;
```

```
    int length;
    char text[BUFFER_SIZE];
};
```

```
// client.c
#include "msgcom.h"

int main() {
    struct msg_s buf;
    int qid, pid;
    char buffer [BUFFER_SIZE];
    char infile[] = "test.in";
    int nread;

    FILE *in = fopen(infile, "r");

    while ((nread = fread(buffer, sizeof(char), BUFFER_SIZE, in)) > 0) {
        // printf("%s", buffer);
        qid = msgget(MSGKEY, IPC_CREAT|0666);
        buf.msgtype = 1;
        buf.pid = pid = getpid();
        strcpy(buf.text, buffer);
        buf.length = nread;
        printf("%d\n", buf.length);
        msgsnd(qid, &buf, sizeof(buf.text), 0);
        msgrcv(qid, &buf, 512, pid, MSG_NOERROR);
        printf ("Request␣received␣a␣message␣from␣server.␣MSG_type␣is:␣%ld
            \n", buf.msgtype);
        // break;
    }
    return 0;
}
```

```
// server.c
#include "msgcom.h"

int main() {
    struct msg_s buf;
    int qid;
    qid = msgget(MSGKEY, IPC_CREAT | 0666);
    char outfile[] = "test.out";
    FILE *out = fopen(outfile, "w");

    if (qid == -1) {
        return (-1);
    }

    while (1) {
```

```
        msgrcv(qid, &buf, sizeof(buf.text), 1, MSG_NOERROR);
        printf("Server␣receive␣a␣request␣from␣process␣%d␣(", buf.pid);
        buf.msgtype = buf.pid;
        fwrite(buf.text, sizeof(char), buf.length, out);
        msgsnd(qid, &buf, sizeof(buf.text), 0);
        printf("%d)\n", buf.length);
        if (buf.length < BUFFER_SIZE) break;
    }
    fclose(out);

    return 0;
}
```

实验 3–共享内存机制

```
// memory.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define SHMKEY 1223
#define SIZE 1024
#define SEMKEY1 2378
#define SEMKEY2 2367

static void semcall(int sid, int op) {
    struct sembuf sb;
    sb.sem_num = 0;
    sb.sem_op = op;
    sb.sem_flg = 0;

    if (semop(sid, &sb, 1) == -1) {
        perror("semop");
    }
}

int createsem(key_t key) {
    int sid;
    union semun {
        int val;
        struct semid_ds *buf;
        ushort *array;
    } arg;

```

```
35      if ((sid = semget(key, 1, 0666|IPC_CREAT)) == -1) {
36          perror("semget");
37      }
38
39      arg.val = 1;
40      if (semctl(sid, 0, SETVAL, arg) == -1) {
41          perror("semctl");
42      }
43      return sid;
44  }
45
46  void P(int sid) {
47      semcall(sid, -1);
48  }
49
50  void V(int sid) {
51      semcall(sid, 1);
52  }
53
54  int main() {
55      char *segaddr;
56      int segid, sid1, sid2;
57      int pid;
58
59      // int shmid;
60      // int ret;
61      // void* mem;
62
63      segid = shmget(SHMKEY, SIZE, IPC_CREAT | 0666 );
64      printf("shmid␣is␣=␣%d,␣pid=%d\n", segid, getpid());
65      segaddr = shmat(segid, (const void*)0, 0);
66      if(segaddr == (void *) -1) {
67          perror("shmat");
68      }
69
70      sid1 = createsem(SEMKEY1); /* 创建两个信号灯, 初值为1 */
71      sid2 = createsem(SEMKEY2);
72      printf("%d␣%d\n", sid1, sid2);
73
74      /*
75      if ((segid = shmget(0x12367, SIZE, IPC_CREAT | 0666) == 1)) {
76          perror("shmget");
77      }
78      segaddr = shmat(segid, NULL, 0);
79      if (segaddr == (void*)-1) {
80          perror("shmat");
81          exit(2);
82      }*/
```

```
83
84      P(sid2);                /* 置信号灯2值为0，表示缓冲区空 */
85      if(!(pid = fork())) {
86          printf ("%d\n", pid);
87          char* outfile = "test.out";
88
89          FILE *out = fopen(outfile, "w");
90          P(sid2);
91          fwrite(segaddr, sizeof(char), SIZE, out);
92          fclose(out);
93          V(sid1);
94      }
95      else {
96          printf ("%d\n", pid);
97          char* infile = "test.in";
98          int length = 0;
99          char buffer [SIZE];
100         FILE *in = fopen(infile, "r");
101         fflush(stdout);
102
103         if ((length = fread(buffer, sizeof(char), SIZE, in)) > 0) {
104             P(sid1);
105             strcpy(segaddr, buffer);
106             printf("%d\n", length);
107             V(sid2);
108         }
109
110     }
111 }
```