实验二　　进程间通信实验报告

1　程序设计要求

1. 设计一个程序，在其中创建一个子进程，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、进程图像改换、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

1. 理解进程以及进程通信的原理和机制。

2. 加深理解基本概念，掌握 SYS V 的 IPC 机制。

3　进程通信简介

进程间通信指的是至少两个进程或线程间传送数据或信号的一些技术或方法。在计算机系统中，每一个进程都有自己的一部分独立的系统资源，它们彼此隔离。为了能使不同的进程相互访问资源并进行协调工作，进程间通信应运而生。
进程间通信的方法主要有：

1. 管道通信 (Pipe)。

2. 消息队列通信 (Message Queue)。

3. 信号量通信 (Semaphore)。

4. 共享内存通信 (Shared Memory)。

5. 套接字通信 (Socket)。

下面将分别对各种通信方法进行进一步介绍。

3.1 管道通信

管道是单向的、先进先出的、无结构的、固定大小的字节流，它是一种信息流缓冲机构，用于连接发送进程和接收进程，以实现它们之间的数据通信。发送进程能把信息以流的方式源源不断地写入管道中，接受进程能以与发送进程写入时的相同顺序读出管道中的信息。在发送进程和接收进程之间能传递任意大的信息，但是实现时所开辟的缓冲区的大小是有限的，故当管道写满时，发送进程就会被阻塞，反之亦然。

3.2 消息队列通信

消息队列提供了一种异步通信协议，即消息的发送发和接收方不必同时与消息队列进行通信，消息将会存储在队列中，直到接收方取回了它们，这也导致接收者必须轮询消息队列才能收到最近的消息。消息队列是信息的一个链表，与信号相比，可以传递更多的信息；与管道相比，它提供了有格式的数据，可以减少开发人员的工作量，但是消息队列能够发送的信息大小以及数目仍然存在限制。

3.3 信号量通信

信号量是一个同步对象，用于控制一个仅支持有限个用户的共享资源，它不需要使用忙碌等待方法。当一个线程（或进程）完成一次对该信号量对象的等待（wait）时，其数值减一；当一个线程（或进程）完成一次对该信号量对象的释放（release）时，其数值加一；当其数值为 0 时，线程（或进程）再对该信号量进行等待将会失败，进程将会被强迫停下来，直至该信号量数值大于 0。

3.4 共享内存通信

共享内存指的是可以被多个进程读写的内存，在通过 IPC 为进程创建一块特殊地址范围的内存后，其他进程可以将之链接到自己的地址空间中，通过权限设置可以使得所有进程都能访问这片内存，因此它可以被用于进程间的通信。与其他通信方式相比，共享内存是一种在进程间传递数据十分高效的方法。但是使用这种方法时，必须保证要通信的进程都运行在同一台机器上。

3.5 套接字通信

套接字是一种应用程序接口，使用 Internet 套接字的概念，使得主机间或者一台计算机上的进程间可以相互通信，在计算机网络通讯方面被广泛使用。具体流程为：使用 *socket*() 函数创建一个套接字，服务器端用 *bind*() 函数将之绑定在一个监听端口上，随后调用 *listen*() 函数对端口进行监听，并可以用 *accept*() 函数接受客户端的连接请求；而客户端可以使用 *connect*() 函数请求连接。连接成功之后，双方能够使用 *send*()、*recv*() 等函数进行通信，最后用 *close*() 函数关闭这个套接字。

4 实验设计

4.1 实验一

4.1.1 父子进程介绍

Linux 系统中，一个现有进程可以通过调用 $fork()$ 函数创建一个新进程，新进程也被称作子进程（child process）。子进程是父进程的副本，因此它将获得父进程的数据空间、堆、栈等资源的副本。父子进程中的这些值虽然基本相同，但是两者各自独立，并非共享同一份资源。

值得注意的是，$fork()$ 函数被调用后，父子进程都将会停留在此处，而函数成功执行后将会有两个不同的返回值：在父进程中，它的返回值为子进程的 ID；而在子进程中，它的返回值为 0。因此，在调用 $fork()$ 函数之后，可以通过返回值来确定当前进程属于父进程还是子进程。

4.1.2 实现方法

首先，为了实现父子进程协同完成某一项任务的要求，我设定了如下的场景：

- 父进程对 1 到 10000 进行求和，并将最终计算结果写入文件 result.txt 中。
- 子进程在父进程完成任务后对文件内容进行读取，并将之打印在屏幕上。

因为功能较为简单，因此我仅使用了信号来实现父子进程间的通信：

- 当父进程完成最后的写操作后，向子进程发送自定义信号。
- 子进程在收到信号之前一直处于阻塞状态，收到信号后对文件进行读操作并显示到屏幕上。

由于实验要求中还要求有信号处理程序，这里我定义了一个函数接收发送的信号，实现的功能为在屏幕上显示发出的信号对应的 ID 值及一串字符串。

除此以外，在父进程完成自己的任务后，使用 $sleep(5)$ 函数休眠 5 秒，之后查询子进程的运行状态：若子进程尚未结束，则利用 $kill()$ 函数强制杀死子进程，然后父进程退出；否则，父进程直接退出即可。为了显示出父进程杀死子进程的过程，我在子进程结束任务后又使用 $sleep(10)$ 使之睡眠 10 秒。程序的具体流程如图1所示：

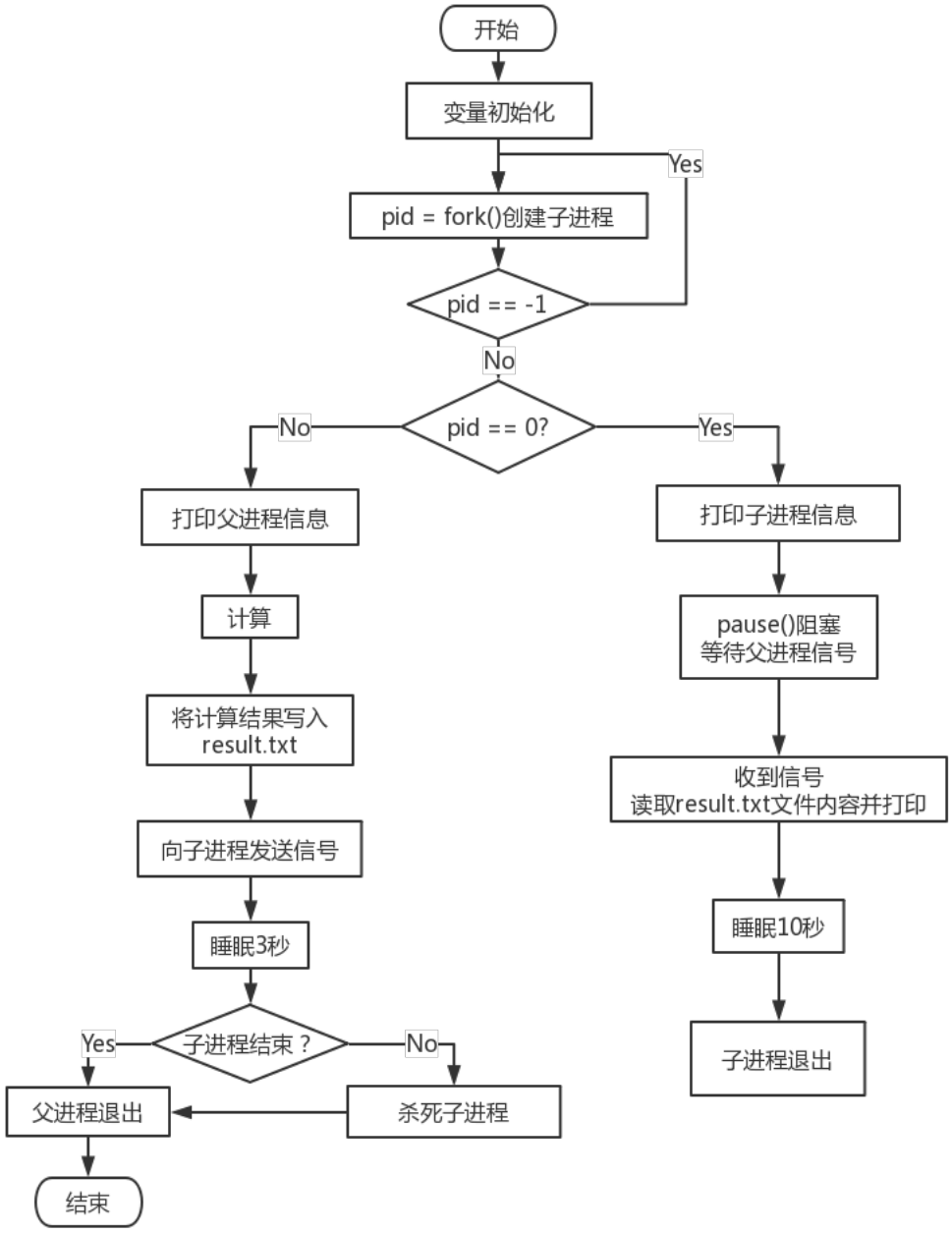

图 1: 父子进程通信流程图

4.1.3 运行结果

程序的具体运行结果如图2所示，可知其完成了实验要求。

```
zyr@ubuntu:~/os_homework/interprocess/comm_1$ ./comm
[Father] This is the father process. My process number is: 2812.
[Father] Start calculating.
[Child] This is the child process. My process number is: 2813; my father is: 2812.
[Father] Have got the sum.
[Father] What I will write is: The result calculated by father is: 49995000.
[Father] Have written result to the file.
[Father] Send the signal.
[Father] Start sleeping.
[System] Signal has been transmitted to the signal processing program.
        The number of signal is: 10.
[Child] Get the signal, ready to read the file.
[Child] I get the result!
[Child] The message in the file is:
        The result calculated by father is: 49995000
[Child] Child sleep.
[Father] Father have killed the child process: 2813.
```

图 2: 父子进程通信结果

4.2 实验二之消息队列

4.2.1 Linux 函数接口

在 Linux 系统中可以通过以下几个消息队列函数实现进程间的通信：

1. $int \quad msgget(key\_t\ key,\ int\ msgflg)$：用于创建一个新的或打开一个已存在的消息队列，此消息队列与 key 相对应。其中，第一个参数 key 用于命名消息队列；第二个参数 $msgflg$ 有多种选择：

   - IPC_CREAT：若消息队列不存在，则创建之；否则对其进行打开操作。
   - IPC_EXCL：与 IPC_CREAT 一起使用，若消息队列对象不存在，则创建之；否则产生一个错误并返回。

   除此以外，在 $msgflg$ 参数中还可以加入权限控制符，它与文件系统中的权限控制符类似，用于限定消息队列读写操作等的权限。

   函数最终返回一个消息队列标识值，若失败则返回-1。

2. $int \quad msgsnd(int\ msgid,\ const\ void\ *msgp,\ size\_t\ msg\_sz,\ int\ msgflg)$：用于把消息添加至消息队列中。其中，第一个参数 $msgid$ 为消息队列的识别码，即 $msgget()$ 函数的返回值；第二个参数 msg_ptr 是一个指向消息缓冲区的指针，它可由用户定义，但是其第一个变量必须是用于标识消息类型的长整形变量，接收函数将依据它来确定消息类型，还应包含需要发送的消息字符串。本次实验中使用的消息结构为：

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

第三个参数 msg_sz 表示发送的消息大小；第四个参数 $msgflg$ 同样是用于控制函数行为的标志位，其取值可以是：

- 0：忽略标志位。
- IPC_NOWAIT：若消息队列为空，则返回一个 ENOMSG，并将控制权交回调用函数的进程。
- 缺省：进程将被阻塞，直到消息可以被写入。

若函数执行成功，返回值为 0，否则为-1。

3. $int \quad msgrcv(int\ msgid,\ void\ *msgp,\ size_t\ msgsz,\ long\ msgtyp,\ int\ msgflg)$：用于从消息队列中取出信息，消息从队列中取出后，该消息就从队列中删除了。其中，前三个参数的含义与 $msgsnd$ 函数完全相同；第四个参数 $msgtyp$ 指定了函数从队列中取出消息的类型，函数将在队列中搜索与之匹配的消息并将其返回，若值为 0 则自动返回队列中最早的消息；第五个参数 $msgflg$ 用于控制函数行为，其取值可以是：

   - 0：忽略标志位。
   - IPC_NOWAIT：若消息队列为空，则返回一个 ENOMSG，并将控制权交回调用函数的进程。
   - MSG_NOERROR：若函数取得的消息长度大于 $msgsz$，将只返回 $msgsz$ 长度的信息，而丢弃余下部分。
   - 缺省：进程将被阻塞，直到函数可以从队列中得到符合条件的消息。

   若函数执行成功，返回值为取得消息的长度，否则为-1。

4.2.2 实现方法

为了使用消息队列实现进程通信，我实现了两个程序，分别命名为 server 和 client。

其中，client 端实现的功能为：读取 test_in.txt 文件中的内容，并分多次将它写入到消息队列中；server 端实现的功能为：分多次从消息队列中读取消息，并将之写入到新文件 test_out.txt 中。具体代码可见附录部分。

最后使用 $diff$ 命令对两个文件进行对比，以验证使用消息队列进行通信的可靠性。

4.2.3 运行结果

分别运行 server 和 client 程序实现消息队列通信，其结果如图3、4所示：

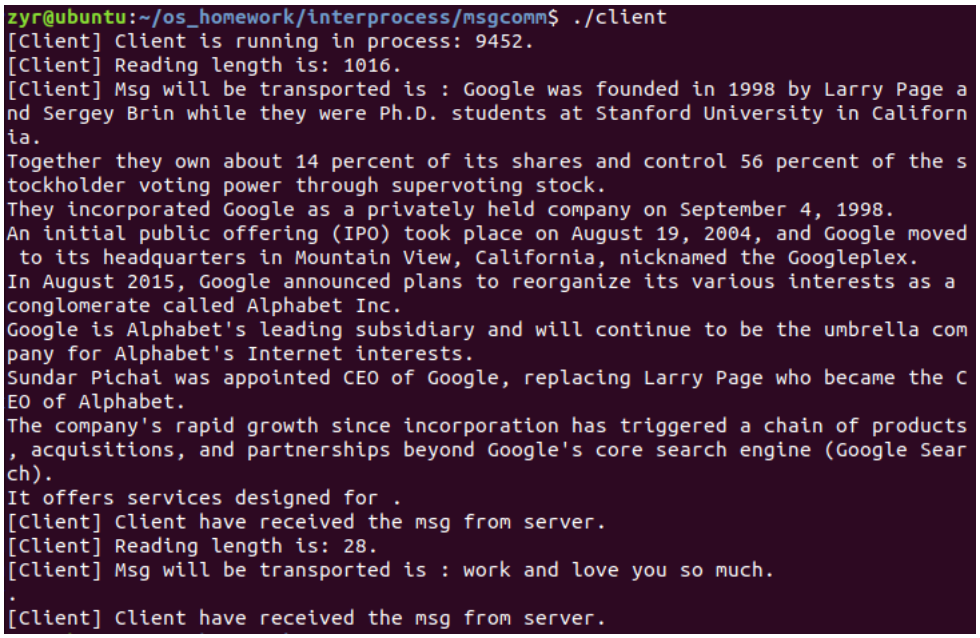

图 3: client 端运行结果

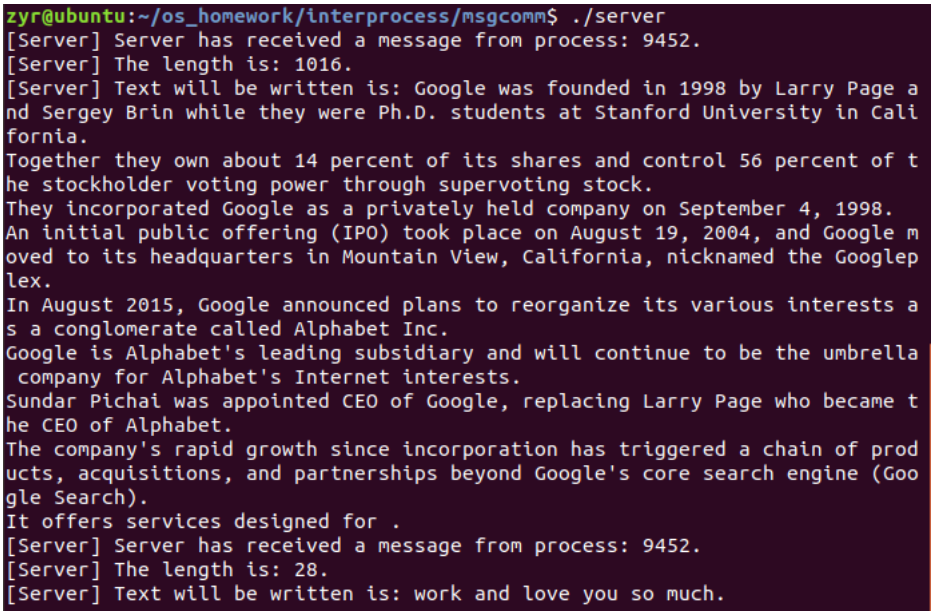

图 4: server 端运行结果

使用 $diff$ 操作对比程序生成的 test_out.txt 和原有的 test_in.txt 文件，结果如图5所示，说明两个文件完全相同，消息队列通信成功。

```
zyr@ubuntu:~/os_homework/interprocess/msgcomm$ ls
client  client.c  msgcomm.h  server  server.c  test_in.txt  test_out.txt
zyr@ubuntu:~/os_homework/interprocess/msgcomm$ diff test_in.txt test_out.txt -a
zyr@ubuntu:~/os_homework/interprocess/msgcomm$
```

图 5: 两进程通信结果

4.3 实验二之共享内存

共享内存可以使多个进程访问同一块内存空间。但是使用共享内存还需要提供同步机制，以避免一个进程对它进行写操作时，有其他进程对这块内存进行读取。在实验中，我采取了信号量的方法实现这一功能。

4.3.1 Linux 函数接口

Linux 系统中，可以使用以下函数接口来实现共享内存：

1. $int \quad shmget(key\_t\ key,\ size\_t\ size,\ int\ shmflg)$：用于得到一个共享内存标识符或创建一个共享内存对象。其中，第一个参数 key 用于为共享内存命名；第二个参数 $size$ 表示指定的需要共享的内存容量，以字节为单位；第三个参数 $shmflg$ 是用于控制函数行为的标志位，其取值可以是：

   - 0：取共享内存标识符，若不存在则函数会报错。
   - IPC_CREAT：如果内核中不存在键值与 key 相等的共享内存，则新建一个共享内存；如果存在这样的共享内存，返回此共享内存的标识符。
   - IPC_CREAT | IPC_EXCL：如果内核中不存在键值与 key 相等的共享内存，则新建一个共享内存；如果存在这样的共享内存则报错。

   若函数执行成功，则返回共享内存的标识符，否则返回-1。

2. $void\ *shmat(int\ shmid,\ const\ void\ *shmaddr,\ int\ shmflg)$：用于将参数指定的共享内存映射到调用进程的地址空间，之后可以像访问进程内存一样访问它。其中第一个参数 $shmid$ 为共享内存标识符；第二个参数 $shmaddr$ 用于指定共享内存出现在进程内存地址的什么位置，设置为 NULL 可以让内核决定；第三个参数 $shmflg$ 用于设定内存模式：SHM_RDONLY 为只读模式，其他为读写模式。

   若函数执行成功则返回附加好的共享内存地址，否则返回-1。

除了共享内存的使用，本实验还需要对信号量进行设置。在 Linux 系统中，可以通过如下函数接口来实现信号量：

1. $int \quad segmet(key\_t\ key,\ int\ num\_sems,\ int\ sem\_flags)$：用于创建一个信号量或获取一个已有信号量。其中第一个参数用于为信号量命名；第二个参数 num_sems 指定需要的信号量数目；第三个参数 sem_flags 是控制函数行为的标志位，其取值可以是：

- IPC_CREAT：如果内核中不存在键值与 key 相等的共享内存，则新建一个共享内存；如果存在这样的共享内存，返回此共享内存的标识符。
- IPC_CREAT | IPC_EXCL：如果内核中不存在键值与 key 相等的共享内存，则新建一个共享内存；如果存在这样的共享内存则报错。

若函数执行成功，则返回相应的信号标识符，否则返回-1。

2. *int semctl(int sem_id, int sem_num, int command, ···)*：用于直接控制信号量信息。第一个参数 *sem_id* 是由 *segmet()* 函数返回的信号量标识符；第二个参数 *sem_num* 除非使用一组信号量，否则为 0；第三个参数 *command* 有多个候选值，但通常最常被使用的是下列两值中的一个：

   - SETVAR：把信号量初始化为一个已知的值，其值通过可选的第四个参数进行定义。
   - IPC_RMID：删除一个已经无需继续使用的信号量标识符。

   若有第四个参数，通常是一个 union semun 结构变量，具体定义如下：

```
union semun{
        /* command为SETVAL时使用的值 */
        int var;

        /*
         * command为IPC_STAT, IPC_SET时
         * 使用的semid_ds结构
         */
        struct semid_ds *buf;

        /* command为SETALL, GETALL时使用的数组值 */
        ushort *array;
}
```

   若函数执行成功，则返回 0，否则返回-1。

3. *int semop(int sem_id, struct sembuf *sem_opa, size_t num_sem_ops)*：用于改变信号量的值。第一个参数 *sem_id* 是由 *segmet* 函数返回的信号量标识符；第二个参数 *sem_opa* 指向的结构体定义如下：

```
struct sembuf{
        /* 信号量序号 */
        short sem_num;

        /*
         * 标识对信号量进行的操作类型
         * 一般取值为:
```

```
8             *  -1: P(等待)操作
9             *  1: V(发送信号)操作
10            */
11           short sem_op;
12
13           /*
14            *  信号量操作的属性标识
15            *  0: 正常操作
16            *  IPC_WAIT: 对信号量操作时非阻塞
17            *            若不满足条件则直接返回-1
18            *  SEM_UNDO: 操作系统维护进程对信号量的调整值
19            *            以便进程结束时恢复信号量状态
20            */
21           short sem_flg;
22   }
```

第三个参数 *num_sem_ops* 为操作结构的数量，恒大于等于 1。

若函数执行成功，则返回 0，否则返回-1。

4.3.2 实现方法

为了利用共享内存来进行进程通信，首先需要创建一片内存作为共享内存，然后将之连接到多个进程之中。本实验中为了简单起见，我在程序中首先连接了共享内存，然后使用 $fork()$ 函数创建了一个子进程。这样一来，父子进程中均连接了这一共享内存。

程序完成的功能分为两大部分：

1. 进程的互斥。父子进程利用信号量 *semid_mutex* 互斥地访问共享内存，分别对之进行读写操作。

2. 进程的同步。父进程实现的功能为：从 test_in.txt 文件中读取内容，并将之写入到共享内存中；子进程实现的功能为：从共享内存中读取消息，并将之写入到 test_out.txt 文件中。在这个过程中使用 *semid_w* 和 *semid_r* 两个信号量，实现了全同步，即父进程需要等待子进程读取消息并清空共享内存以进行写入；子进程需要等待父进程向共享内存写入消息以进行读取。

4.3.3 运行结果

运行 shm_comm 程序实现共享内存通信，其结果如图6所示。

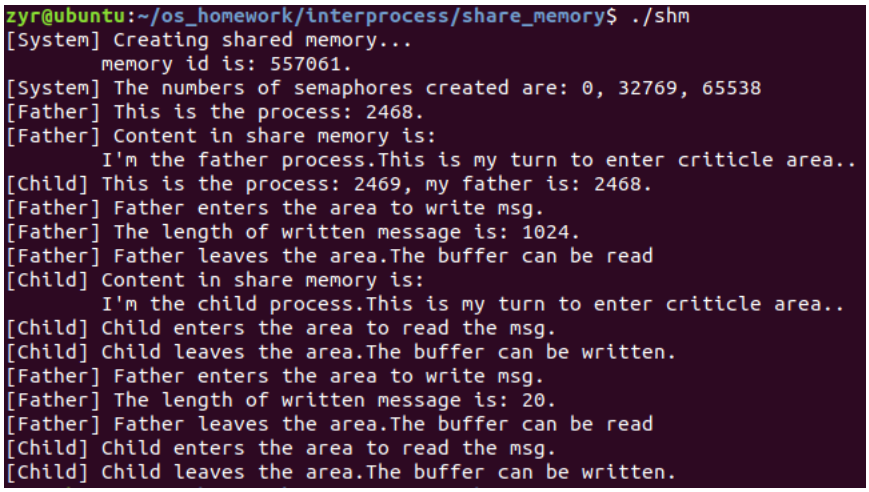

图 6: 共享内存实现进程通信

使用 $diff$ 操作对比程序生成的 test_out.txt 和原有的 test_in.txt 文件，结果如图7所示，说明两个文件完全相同，消息队列通信成功。

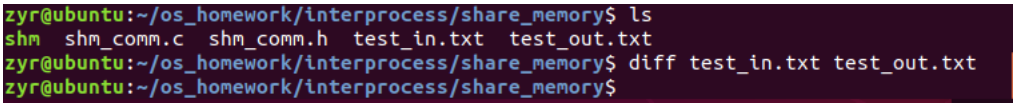

图 7: 父子进程利用共享内存进行通信结果

5 遇到的问题

在本次实验过程中，我也遇到了一些问题：

1. $fork()$ 函数的理解。在上课过程中，通过老师的讲解，我只是知晓在 Linux 系统中可以通过 $fork()$ 函数创建一个子进程，但是有关该子进程如何运行、它与父进程的关系具体如何等问题则并不太清楚。

   通过查询资料，了解到在程序执行过程中如果调用一次 $fork()$ 函数，它将会有两个返回值：对于父进程，函数的返回值为创建的子进程编号；对于子进程，函数的返回值为 0。因此可以借助对 $fork()$ 函数返回值的判断，划分当前位于父进程还是子进程中，进而执行不同的操作。且子进程创建时，它将获得父进程的数据空间、堆、栈等资源的副本，但是两者的资源仍然相互独立，因此仍需要进行进程间通信。

2. 打开文件后要执行关闭操作。实验一中我设定的步骤为：父进程打开文件向其写入计算结果后，子进程再打开文件读取结果。但是由于父进程打开文件后忘记执行关闭操作，导致最终读取结果为乱码。经过细心检查，发现了这一导致错误的原因，在以后的编程过程中也要引以为戒。

3. $msgsnd()$ 函数发送消息时有长度限制。在使用消息队列机制完成实验二的过程中，我开始时设定了消息字符串长度为 1024 字节，从文件中进行读取时也定义了最大读取长度为 1024 字节。但是使用 $msgsnd()$ 函数发送至消息队列后，再使用 $msgrcv()$ 函数从消息队列中读取消息，发现仅能读到 1016 字节数据。

   通过查询资料发现，$msgsnd()$ 函数能够发送的消息长度为 $sizeof(struct\ msg\_size) - sizeof(long)$，因此会出现上述少 8 个字节的情况。对代码进行修改之后，程序能够成功借助消息队列完成进程间通信。

4. 对字符串操作时注意结尾补'\0'。在实验过程中，对文件进行读操作时，我设定的最大读取长度与存储的字符串长度相同。但是这会导致字符串末尾缺少结束标志，将它通过消息队列抑或是共享内存进行传递后，再写入新文件时，可能会有溢出，产生异常结果。因此存储从文件中读取的字符串时，需要包含额外的空间添加结束标志，以保证输出结果的正确性。

5. 进程既有互斥又有同步时，$memset()$ 操作会引发异常。在使用共享内存机制实现实验二的过程中，我首先使父子进程对内存空间进行分别进行一次互斥的访问，然后再使用全同步借助共享内存进行读写操作。

   开始时，我在每次互斥访问后对与共享内存均使用 $memset()$ 函数进行一次清空操作，但是添加进程同步功能后，发现从共享内存中获得的消息为空。整理思路后发现：若父进程先执行，在执行完互斥访问、消息写入共享内存操作后，该进程将会阻塞，以等待子进程取走内存中的消息，但是此时子进程还未执行过，因此将先执行一次互斥访问，之后清除共享内存空间，这就导致子进程读入的消息为空。

   因此在互斥访问前，我使用一个局部变量存储当前共享内存中的数据，并在互斥访问将要结束时，对共享内存中的数据进行恢复，成功地解决了上述问题。

6 心得体会

通过本次实验，我对于进程通信中使用的多种机制有了更为全面的了解。通过动手设计程序实现进程的互斥、同步等操作，对于信号量的引入、全同步与半同步等都有了更为清晰的认识，巩固了课堂上所学的知识点。在 Linux 系统中进行程序的编写、调试，上网查阅函数定义、使用手册等步骤，很好地锻炼了我们的专业技能。在实验过程中虽然遇到了不少困难，但是通过与同学、老师进行交流、上网查阅资料等途径，较好地解决了它们，培养了协作精神。

总之，通过本次实验我收获颇丰，衷心感谢刘老师在实验过程中的指导与帮助！

7 附录

7.1 实验一代码

```
1 // comm.c
```

```c

#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


void sig_p(int dunno)
{
  printf("[System] Signal has been transmitted to the signal
      processing program.\n\tThe number of signal is: %d.\n",
      dunno);
}


int main()
{
  int pid, sum = 0, status = 0;

  char output[] = "result.txt";

  // bind the signal processing function and SIGUSR1
  signal(SIGUSR1, sig_p);

  // creating sub-process failed
  while((pid = fork()) == -1)
    ;

  // father process
  if(pid)
  {
    char buffer[20] = {'\0'};
    int ret = 0;

    printf("[Father] This is the father process. My process number
        is: %d.\n", getpid());

    printf("[Father] Start calculating.\n");
    // main task: calculating
    for(int i = 0; i < 10000; ++i)
    {
      sum += i;
    }
```

```
45
46      printf("[Father] Have got the sum.\n");
47
48      // transport int to char
49      sprintf(buffer, "%d", sum);
50
51      char str[] = "The result calculated by father is: ";
52
53      char *final = (char*) malloc(strlen(str) + strlen(buffer) + 1)
            ;
54
55      if (final == NULL)
56      {
57        // malloc fail
58        printf("malloc memory failed.\n");
59        exit(2);
60      }
61
62      strcpy(final, str);
63      strcat(final, buffer);
64
65      printf("[Father] What I will write is: %s.\n", final);
66
67      FILE *out = fopen(output, "w");
68      if(out == NULL)
69      {
70        printf("[Father] Open file %s fail.\n", output);
71        exit(1);
72      }
73
74      fwrite(final, sizeof(char), strlen(final), out);
75      // remember to close the file
76      fclose(out);
77
78      printf("[Father] Have written result to the file.\n");
79
80      // send signal to child
81      printf("[Father] Send the signal.\n");
82      kill(pid, SIGUSR1);
83
84      // father sleep
85      printf("[Father] Start sleeping.\n");
86      sleep(3);
87
88      // test child process finished or not
89      if( (waitpid(pid, NULL, WNOHANG)) == 0)
```

```
90      {
91        if( (ret = kill(pid, SIGKILL)) == 0)
92        printf("[Father] Father have killed the child process: %d.\n
              ", pid);
93      }
94
95    }
96    // children process
97    else
98    {
99      printf("[Child] This is the child process. My process number
            is: %d; my father is: %d.\n", getpid(), getppid());
100
101     // waiting for the signal
102     pause();
103
104     printf("[Child] Get the signal, ready to read the file.\n");
105
106     FILE *in = fopen(output, "r");
107     char readout[50] = {'\0'};
108
109     fread(readout, sizeof(char), 50, in);
110     fclose(in);
111
112     // after getting the signal
113     printf("[Child] I get the result!\n");
114     printf("[Child] The message in the file is:\n\t%s\n", readout)
            ;
115
116     printf("[Child] Child sleep.\n");
117     sleep(10);
118     exit(3);
119   }
120
121   return 0;
122 }
```

7.2 实验二之消息队列实现

```
1  // msgcomm.h
2
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <unistd.h>
```

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

#define BUFFER_SIZE 1024
#define MSGKEY 1234
#define C2S 123
#define S2C 456

struct msg_s {
  long msgtype;
  int length;
  char text[BUFFER_SIZE];
};
```

```c
// server.c

#include "msgcomm.h"

int main()
{
  struct msg_s rcv;

  memset(rcv.text, 0, sizeof(rcv.text));

  struct msg_s ret;

  // msgtype of server -> client is S2C: 456
  ret.msgtype = S2C;
  // initiate ret
  char retmsg[] = "Server have received the msg.";
  strcpy(ret.text, retmsg);
  ret.length = strlen(ret.text);

  // id of msg queue
  int qid;

  qid = msgget(MSGKEY, IPC_CREAT | 0666);

  char output[] = "test_out.txt";
  FILE *out = fopen(output, "w");

  // create msg fail
  if(qid == -1)
  {
```

```
31      printf("create message queue fail.\n");
32      return 2;
33    }
34
35    // listening
36    while(1)
37    {
38
39      // msgtpye from client is C2S: 123
40      msgrcv(qid, &rcv, sizeof(rcv.text), C2S, MSG_NOERROR);
41
42      printf("[Server] Server has received a message from the
            message queue.\n");
43      printf("[Server] The length is: %d.\n", rcv.length);
44
45      printf("[Server] Text will be written is: %s.\n", rcv.text);
46
47      fwrite(rcv.text, sizeof(char), rcv.length, out);
48
49      // tell client the msg have been received
50      msgsnd(qid, &ret, sizeof(ret.text), 0);
51
52      // nothing more will be sended
53      if(rcv.length < BUFFER_SIZE - sizeof(long))
54        break;
55      // clear the rcv.text
56      memset(rcv.text, 0, sizeof(rcv.text));
57    }
58
59    fclose(out);
60
61    return 0;
62  }
```

```
1   // client.c
2
3   #include "msgcomm.h"
4
5   int main()
6   {
7     int qid = 0, pid = 0;
8     struct msg_s snd;
9     char buffer[BUFFER_SIZE] = {'\0'};
10    char infile[] = "test_in.txt";
11    int nread = 0;
12
```

```
13    printf("[Client] Client is running in process: %d.\n", getpid())
          ;
14
15    FILE *in = fopen(infile, "r");
16
17    while( (nread = fread(buffer, sizeof(char), BUFFER_SIZE - sizeof
          (long), in)) > 0 )
18    {
19      printf("[Client] Reading length is: %d.\n", nread);
20
21      qid = msgget(MSGKEY, IPC_CREAT | 0666);
22
23      // msgtype of client -> server is C2S: 123
24      snd.msgtype = C2S;
25      pid = getpid();
26
27      // clear the snd.text
28      memset(snd.text, 0, sizeof(snd.text));
29
30      strcpy(snd.text, buffer);
31
32      snd.length = strlen(snd.text);
33
34      printf("[Client] Msg will be transported is : %s.\n", snd.text
            );
35
36      // the maximun can be sended is BUFFER_SIZE - sizeof(long)
37      msgsnd(qid, &snd, sizeof(snd.text), 0);
38
39      // msgtype of server -> client is S2C: 456
40      msgrcv(qid, &snd, 512, S2C, MSG_NOERROR);
41      printf("[Client] Client have received the msg from server.\n")
            ;
42
43      memset(buffer, 0, sizeof(buffer));
44      memset(snd.text, 0, sizeof(snd.text));
45    }
46
47    fclose(in);
48
49    return 0;
50  }
```

7.3 实验二之共享内存实现

```c
// shm_comm.h

#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 SHM_KEY 1234
#define SHM_SIZE 1024
#define SEMKEY1 2345
#define SEMKEY2 3456
#define SEMKEY3 4567

// create a new semaphore
int createsig(key_t key, int num)
{
  int semid = 0;
  union semun{
    int var;
    struct semid_ds *buf;
    ushort *array;
  } arg;

  // create a semaphore
  if ( (semid = semget(key, 1, IPC_CREAT | 0666)) == -1)

  {
    perror("semget error !");
    return -1;
  }

  arg.var = num;

  // use arg to set the value of the semaphore
  if (semctl(semid, 0, SETVAL, arg) == -1)
  {
    perror("semctl error !");
    return -1;
  }

  return semid;
}
```

```
47
48 // operate on a certain semaphore
49 void op_sem(int semid, int op)
50 {
51   struct sembuf buf;
52   buf.sem_num = 0;
53   buf.sem_op = op;
54   buf.sem_flg = 0;
55
56   if(semop(semid, &buf, 1) == -1)
57   {
58     perror("semop error !");
59   }
60 }
61
62 /*
63  *  P operation
64  *  enter the criticle area
65  */
66 void P(int semid)
67 {
68   op_sem(semid, -1);
69 }
70
71 /*
72  *  V operator
73  *  leave the criticle area
74  */
75 void V(int semid)
76 {
77   op_sem(semid, 1);
78 }
```

```
1 // shm_comm.c
2
3 #include "shm_comm.h"
4
5 int main()
6 {
7   int memid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
8   int semid_w = 0, semid_r = 0, semid_mutex = 0;
9   int pid = 0;
10
11   if (memid == -1)
12   {
13     printf("Shared memory creating fail !\n");
```

```
14      return 2;
15     }
16    else
17      printf("[System] Creating shared memory...\n\tmemory id is: %d
            .\n", memid);
18
19    char * memaddr = shmat(memid, NULL, 0);
20    if(memaddr == (void*)-1)
21      printf("map shared memory to the memory of process fail.\n");
22
23    do
24    {
25      semid_w = createsig(SEMKEY1, 1);
26    }while(semid_w == -1);
27
28    do
29    {
30      semid_r = createsig(SEMKEY2, 0);
31    }while(semid_r == -1);
32
33    do
34    {
35      semid_mutex = createsig(SEMKEY3, 1);
36    }while(semid_mutex == -1);
37
38    printf("[System] The numbers of semaphores created are: %d, %d,
        %d\n", semid_w, semid_r, semid_mutex);
39
40    /*
41     *  child process
42     *  child reads the msg from memory and writes to the file
43     */
44    if(!(pid = fork()))
45    {
46      printf("[Child] This is the process: %d, my father is: %d.\n",
            getpid(), getppid());
47
48      P(semid_mutex);
49
50      // write to the memory
51      char childtext[] = "I'm the child process.This is my turn to
            enter criticle area.";
52      char tmpbuffer[SHM_SIZE + 1] = {'\0'};
53      strcpy(tmpbuffer, memaddr);
54      tmpbuffer[SHM_SIZE] = '\0';
55      strcpy(memaddr, childtext);
```

```
56      printf("[Child] Content in share memory is:\n\t%s.\n", memaddr
            );
57      // memset(memaddr, 0, SHM_SIZE);
58      strcpy(memaddr, tmpbuffer);
59
60      V(semid_mutex);
61
62      char * output = "test_out.txt";
63      char buffer_c[SHM_SIZE + 1] = {'\0'};
64
65      FILE * out = fopen(output, "w");
66
67      while (1)
68      {
69        // wait for the buffer to read
70        P(semid_r);
71
72        printf("[Child] Child enters the area to read the msg.\n");
73        strcpy(buffer_c, memaddr);
74        buffer_c[SHM_SIZE] = '\0';
75        // printf("msg will be written to the file is: %s.\n",
              buffer_c);
76        fwrite(buffer_c, sizeof(char), strlen(buffer_c), out);//
              SHM_SIZE, out);
77        // fclose(out);
78
79        printf("[Child] Child will leave the area.The buffer can be
              written.\n");
80        // buffer can be written
81        V(semid_w);
82
83        if(strlen(buffer_c) < SHM_SIZE)
84          break;
85        memset(buffer_c, 0, SHM_SIZE);
86      }
87      fclose(out);
88    }
89    /*
90     * father process
91     * father writes the momery
92     */
93    else
94    {
95      printf("[Father] This is the process: %d.\n", getpid());
96
97      P(semid_mutex);
```

```
98
99      // write to the memory
100     char fathertext[] = "I'm the father process.This is my turn to
            enter criticle area.";
101     char tmpbuffer[SHM_SIZE + 1] = {'\0'};
102     strcpy(tmpbuffer, memaddr);
103     strcpy(memaddr, fathertext);
104     printf("[Father] Content in share memory is:\n\t%s.\n",
            memaddr);
105     // memset(memaddr, 0, SHM_SIZE);
106     strcpy(memaddr, tmpbuffer);
107
108     V(semid_mutex);
109
110     char *input = "test_in.txt";
111
112     FILE *in = fopen(input, "r");
113     char buffer_f [SHM_SIZE] = {'\0'};
114
115     // refresh the buffer of stdout
116     fflush(stdout);
117
118     int length = 0;
119
120     while ( (length = fread(buffer_f, sizeof(char), SHM_SIZE, in))
            > 0)
121     {
122       // wait for the buffer to write
123       P(semid_w);
124       printf("[Father] Father enters the area to write msg.\n");
125
126       strcpy(memaddr, buffer_f);
127       printf("[Father] The length of written message is: %d.\n",
              length);
128       memset(buffer_f, 0, SHM_SIZE);
129
130       printf("[Father] Father will leave the area.The buffer can
              be read.\n");
131       // buffer can be read
132       V(semid_r);
133     }
134     fclose(in);
135   }
136
137   return 0;
138 }
```