Operating System Lab 2 : 进程和进程通信

实 验 报 告

刘子涵　518021910690

【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**信号量**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. 理解进程和进程通信的原理和相关机制；
2. 提高用 C 语言编制程序的能力，熟悉标准库函数 API 接口；
3. 加深对进程基本概念的理解，掌握 Unix System V 的 IPC 机制。

【3】实验环境

- 操作系统：Ubuntu 20.04（Linux）
- Linux 内核版本：5.4.0
- GCC 版本：9.3.0

【4】实验原理

1. UNIX 进程控制的系统调用

UNIX 提供了大量从 C 程序中操作进程的系统调用，在第一个实验中实现相应的功能依赖这些系统调用，下面将介绍实验中用到的几个系统调用：

- `fork()`：父进程调用该函数创建一个新的运行的子进程。子进程得到与父进程用户级虚拟地址空间相同且独立的一份副本（代码、数据、堆、共享库、用户栈），还获得与父进程任何打开文件描述符相同的副本，两者有不同的 PID。该函数只在父进程调用一次，在父进程返回子进程 PID，在子进程返回 0，两者是并发执行的独立进程。

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

pid_t fork(void);
```

- `getpid()`：返回调用进程的 PID。返回类型 `pid_t` 被定义为 `int`。

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

pid_t getpid(void);
```

- `sleep()`：让调用进程挂起（睡眠）一段时间。返回还要休眠的秒数。

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
```

- `pause()`：让调用进程休眠，直到该进程收到一个信号。

```
#include <unistd.h>

int pause(void);
```

- `execl() / execv()`：在当前进程的上下文中加载并运行一个新程序，实现进程的图像改换。该函数加载并运行文件 `path`，两者区别在于 `execl` 是在调用时使用参数列表，`execv` 是事先构造一个指向各参数的指针数组 `argv[]`，然后将该数组的地址作为这些函数的参数。而 `execle()` 和 `execve()` 则多了环境变量指针数组 `envp[]`（使用新的环境变量代替调用进程的环境变量）。

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
```

- `waitpid() / wait()`：父进程调用该函数等待子进程终止。
  - 参数 `pid` 确定父进程等待集合的成员。如果 `pid>0`，则等待集合只包含进程 ID 等于 `pid` 的进程；如果 `pid=-1`，则等待集合包括父进程的所有子进程。
  - 参数 `statusp` 是指向 `status` 的指针。如果该参数非空，则该函数将导致子进程返回的状态写入 `status`。
  - 参数 `options` 描述函数的行为。如果 `options=0`，则默认挂起调用进程的执行，直到等待集合有一个子进程终止；如果 `options=WNOHANG`，则如果等待集合中任何子进程都没有终止，则立即返回，函数返回 0。
  - 该函数如果成功返回子进程 PID，如果是 `WNOHANG` 返回 0，其它错误返回 -1。
  - `wait(&status)` 相当于 `waitpid(-1, &status, 0)`。

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

pid_t waitpid(pid_t pid, int *statusp, int options);
pid_t wait(int *statusp);
```

- `signal()`：调用进程通过该函数设置与某个信号相关联的默认行为。

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

sighandler_t signal(int signum, sighandler_t handler);
e.g. signal(SIGUSR1, func);
```

- `kill()`：调用进程通过该函数发送信号给其它进程。如果 `pid>0`，则发送对象为进程 `pid`；如果 `pid=0`，则发送对象为调用进程所在进程组的每个进程（包括自己）；如果 `pid<0`，则发送对象为进程组 `|pid|` 的每个进程。

```c
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

- `exit()`：终止进程。以退出状态 `status` 来终止进程。

```c
#include <stdlib.h>

void exit(int status);
```

2. UNIX 消息通信机制及 API

消息通信机制的基本思想，是由系统的消息通信机构统一管理一组空闲的消息缓冲区。当一个进程想要给另一个进程发送消息时，先向系统申请一个空闲缓冲区，填写消息正文和一些控制信息，通过消息通信机构将该消息发送到接收进程的消息队列中。接收进程在一个适当时机从消息队列中移出一个消息，读取所有信息后，再释放消息缓冲区。

- `msgget()`：生成一个消息队列。
  - 参数 `key` 是通信双方约定的消息队列关键字（非负长整数）。如果与该关键字对应的消息队列不存在，系统为其创建一个消息队列，返回队列 ID `qid`；存在则直接返回。
  - 参数 `flags` 是 `o_flags` 和 `mode` 的组合。例如：`IPC_CREAT|0666` 表示建立一个新的消息队列（返回一个已存在的消息队列的 ID），并且设置其权限为 0666。

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

int msgget(key_t key, int flags);
```

- `msgsnd()`：向消息队列发送一个消息。参数 `qid` 是消息队列 ID，参数 `nbytes` 是消息正文长度，参数 `flags` 是发送标志，参数 `buf` 是用户定义的消息结构，本实验需要自己定义该数据结构，将在下文给出。

```c
int msgsnd(int qid, struct msgbuf *buf, int nbytes, int flags);
```

- `msgrcv()`：从消息队列接收一个消息。类似 `msgsnd()`，不再赘述。

```c
int msgrcv(int qid, struct msgbuf *buf, int nbytes, long mtype, int flags);
```

3. UNIX 共享内存机制及 API

共享内存机制的基本思想，是让相关进程直接共享某些内存区域，而不必移动数据本身。UNIX System V 支持任意数目进程对内存的共享，每一个共享内存区域称为一个共享段，一个进程可以访问多个共享段。

- `shmget()`：创建一个共享内存段。

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

int shmget(key_t key, int nbytes, int flags);
```

- `shmat()`：将共享内存段映射到进程的虚拟地址空间。

```c
char *shmat(int segid, char *addr, int flags);
```

- `shmdt()`：解除共享内存段的映射。

```c
int shmdt(char *addr);
```

4. UNIX 信号量机制及 API

进程间的互斥与同步利用 `semWait` 和 `semSignal` 操作来实现，但 UNIX 系统没有直接提供这两个操作，只提供了一组有关信号量的系统调用。

- `semget()`：创建一个信号量组。

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

int semget(key_t key, int nsems, int flags);
```

- `semop()`：对信号量组的操作。

```c
int semop(int sid, struct sembuf **ops, unsigned nops);

struct sembuf {
    short sem_num; // 信号量编号，从 0 开始
    short sem_op; // 信号量操作数（取正值或负值，对应信号量增加或减少该值）
    short sem_flg; // 操作标志
};
```

- `semctl()`：信号量控制。
  - `cmd` 取值为 `SETVAL`，表示设置信号量的值为 `arg.val`，用于对信号量初始化。

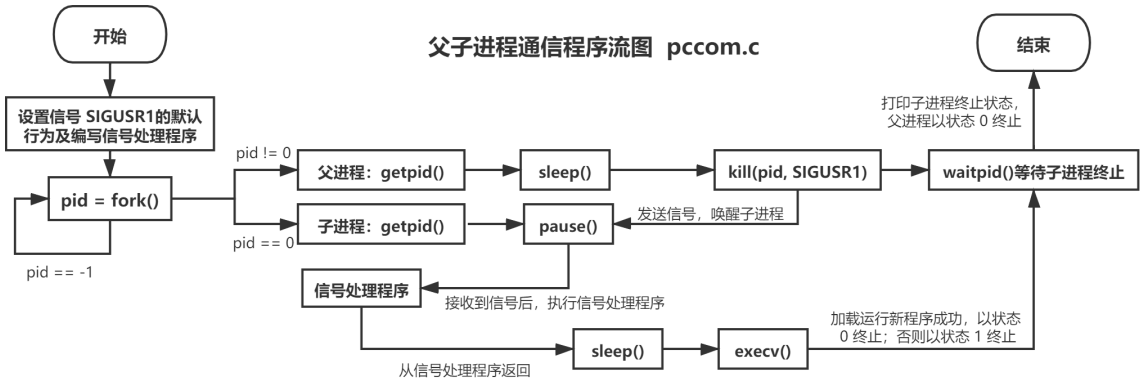
```c
int semctl(int sid, int snum, int cmd, union semun arg);

union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
}
```

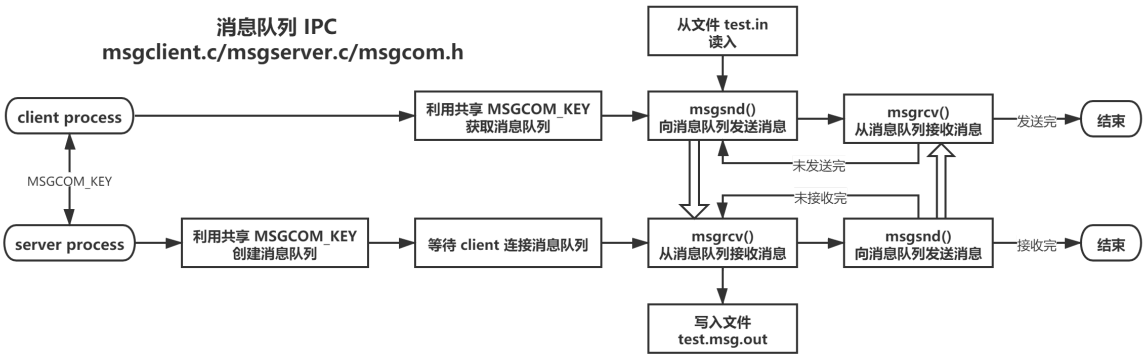
【5】实验设计与实现

1. 流程设计

- **父子进程通信**
  - 【父进程】① 调用 `signal()` 设置信号 `SIGUSR1` 的默认信号处理程序；② 调用 `fork()` 创建子进程；③ 调用 `getpid()` 获取父进程 ID；④ 调用 `sleep()` 睡眠 2 s；⑤ 调用 `kill()` 向子进程发送信号；⑥ 调用 `waitpid()` 等待子进程终止；⑦ 打印子进程终止状态并调用 `exit()` 终止自己。
  - 【子进程】① 调用 `getpid()` 获取子进程 ID；② 调用 `pause()` 睡眠直到接收到信号；③ 接收到信号后，自动调用信号处理程序；④ 返回源程序调用 `sleep()` 睡眠 2 s；⑤ 调用 `execv()` 加载并运行一个系统程序；⑥ 根据执行情况以相应状态退出。



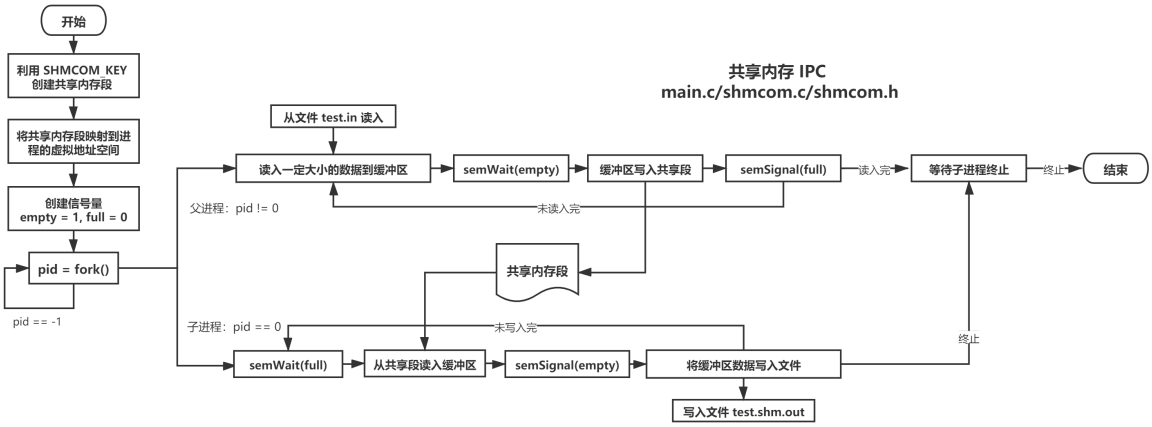
- **消息队列 IPC**
  - 【服务端进程】① 利用共享 MSGCOM_KEY 创建消息队列；② 在一个循环中，等待客户端向消息队列发送消息，接收一定大小数据并写入文件，并向消息队列发送回复消息，接收完则结束进程。
  - 【客户端进程】① 利用共享 MSGCOM_KEY 获取消息队列（后于服务端进程）；② 在一个循环中，从文件读入一定大小的数据，并向消息队列发送该消息，并等待服务端向消息队列发送回复消息，发送完则结束进程。



- **共享内存 IPC**
  - 【父进程】① 利用 SHMCOM_KEY 创建共享内存段，并将该内存段映射到进程的虚拟地址空间；② 创建两个信号量：`empty` 和 `full`，初值分别为 1 和 0；③ 调用 `fork()` 创建子进程，子进程复制了父进程的虚拟地址空间，从而可以访问共享内存段；④ 从文件读入一定大小的数据到缓冲区；⑤ 将缓冲区数据写入共享段；⑥ 如果从文件读入完数据，则等待子进程终止；⑦ 终止。
  - 【子进程】① 从共享段读入缓冲区；② 将缓冲区数据写入文件；③ 如果写入完，则终止。
  - 【同步】因为父子进程有相互制约关系，即设置共享段大小仅为 1，共享段满（`empty=0, full=1`）时，父进程不能向共享段写入；共享段空（`empty=1, full=0`）时，子进程不能

从共享段读出。所以需要设置两个信号量，记录当前共享段是否空、是否满。

- 【互斥】父子进程必须互斥访问共享段，但**不设置互斥量的原因在于，信号量 `empty` 和 `full` 两者必有一个为 0**，所以父子进程进入临界区前，检查信号量 `empty` 和 `full` 后并不会同时进入临界区，所以此处没有必要设置互斥量。



2. 数据结构

- **消息队列 IPC**：消息结构

```c
typedef struct {
    long msgtype;  // 消息类型
    int msglen;  // 消息长度
    pid_t sendpid;  // 发送进程 PID
    char msgtext[BUFFER_SIZE]; // 消息正文
} msgbuf;
```

- **共享内存 IPC**：信号量结构

```c
struct sembuf {
    short sem_num; // 信号量编号，从 0 开始
    short sem_op; // 信号量操作数（取正值或负值，对应信号量增加或减少该值）
    short sem_flg; // 操作标志
};
```

- **共享内存 IPC**：信号量参数

```c
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
}
```

3. 功能实现

- **父子进程通信** `pccom.c`

```c
/*
 *  pccom.c  --  parent-child communication with system calls
 *
 *  1. create child process with `fork()`
 *  2. make process sleep with `sleep()`
 *  3. change process image with `execv()`
```

```
 *  4. wait for child process to terminate with `wait()/waitpid()`
 *  5. set interrupt handler with `signal()`
 *  6. send signal with `kill()`
 *  7. terminate process with `exit()`
 */

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

void sighandler() {
    printf("[Signal] Start signal handler of SIGUSR1\n");
}

void main() {
    pid_t pid;
    int status;
    signal(SIGUSR1, sighandler);

    while ((pid = fork()) == -1);
    if (pid) {
        /* parent process */
        printf("[Parent] Start parent process: %d\n", getpid());
        sleep(2);  // sleeping
        printf("[Parent] Send signal...\n");
        kill(pid, SIGUSR1);  // send a SIGUSR1 signal to child process
        printf("[Parent] Wait for child to end...\n");
        if (waitpid(pid, &status, 0) < 0) {  // wait for child process to
terminate
            printf("[Parent] waitpid error\n");
        } else {
            if (WIFEXITED(status))
                printf("[Parent] Child process exited with status: %d\n",
WEXITSTATUS(status));
        }
        printf("[Parent] Parent process ended\n");
        exit(0);
    } else {
        /* child process */
        printf("[Child] Start child process: %d\n", getpid());
        pause();  // sleep until received a signal
        char *argv[] = {"uname", "-a", (char*)0};
        sleep(2);  // sleeping
        printf("[Child] Start executing `uname -a`...\n");
        execv("/bin/uname", argv);  // execute a built-in program
        printf("[Child] execv error\n");
        exit(1);  // exit abnormally
    }
}
```

- **消息队列 IPC** `msgcom.h`

```
/*
 *  msgcom.h  --  message communication header file
```

```
 */

#ifndef __MESSAGE_COMM__
#define __MESSAGE_COMM__

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSGCOM_KEY 123456
#define BUFFER_SIZE 1024
#define MSGTYPE 1

/* message buffer */
typedef struct {
    long msgtype;
    int msglen;
    pid_t sendpid;
    char msgtext[BUFFER_SIZE];
} msgbuf;

#endif
```

- **消息队列 IPC** `msgclient.c`

```
/*
 *  msgclient.c  --  message communication client end source file
 */

#include "msgcom.h"

void main() {
    msgbuf buf;
    int qid;  // msgqueue id
    pid_t pid;
    char buffer[BUFFER_SIZE];
    char infile[] = "../test/test.in";  // input message
    FILE *in = fopen(infile, "r");
    pid = getpid();
    // get existed msgqueue id after the server process has created the
message queue
    qid = msgget(MSGCOM_KEY, IPC_CREAT|0666);
    int length;

    while ((length = fread(buffer, sizeof(char), BUFFER_SIZE, in)) > 0) {
        // fill in the fields of message to be sent
        buf.msgtype = MSGTYPE;
        buf.msglen = length;
        buf.sendpid = pid;
        strcpy(buf.msgtext, buffer);
        // send the message to the server process
        msgsnd(qid, &buf, sizeof(buf.msgtext), 0);
        // receive the message from the server process
        msgrcv(qid, &buf, BUFFER_SIZE, pid, MSG_NOERROR);
```

```
        printf("Received message from server process: client pid %ld\n",
buf.msgtype);
    }

    fclose(in);
    exit(0);
}
```

- **消息队列 IPC** `msgserver.c`

```
/*
 *  msgserver.c  --  message communication server end source file
 */

#include "msgcom.h"

void main() {
    msgbuf buf;
    int qid;  // msgqueue id
    char outfile[] = "../test/test.msg.out";  // output message
    FILE *out = fopen(outfile, "w");
    // create message queue
    if ((qid = msgget(MSGCOM_KEY, IPC_CREAT|0666)) == -1) {
        printf("Message queue with specified key has already existed\n");
        exit(0);
    }

    while (1) {
        // receive the message from the client process
        msgrcv(qid, &buf, BUFFER_SIZE, MSGTYPE, MSG_NOERROR);
        printf("Received message from client process:\n%s\n", buf.msgtext);
        // write to the file
        fwrite(buf.msgtext, sizeof(char), buf.msglen, out);
        buf.msgtype = buf.sendpid;
        // send the message to the client process
        msgsnd(qid, &buf, sizeof(buf.msgtext), 0);
        if (buf.msglen < BUFFER_SIZE) break;
    }

    fclose(out);
    exit(0);
}
```

- **共享内存 IPC** `shmcom.h`

```
/*
 *  shmcom.h  --  shared memory communication header file
 */

#ifndef __SHARED_MEMORY_COMM__
#define __SHARED_MEMORY_COMM__

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
```

```c
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define SHMCOM_KEY 123456789
#define SHM_SIZE 1024
#define SEM_EMPTY_KEY 13579
#define SEM_FULL_KEY 24680
#define BUFFER_SIZE 1024

typedef int sem_t;

/* create semaphore */
sem_t createSem(key_t key);

/* semaphore wait function */
void semWait(sem_t sem);

/* semaphore signal function */
void semSignal(sem_t sem);

#endif
```

- **共享内存 IPC** `shmcom.c`

```c
/*
 *  shmcom.c  --  shared memory communication source file
 */

#include "shmcom.h"

/* create semaphore */
sem_t createSem(key_t key) {
    sem_t sem;
    union semun {
        int val;
        struct semid_ds *buf;
        __u_short *array;
    } arg;
    // get semaphore
    if ((sem = semget(key, 1, IPC_CREAT|0666)) == -1) {
        perror("semget error\n");
    }
    // control: `SETVAL` initialize semaphore value with `arg.val = 1`
    arg.val = 1;
    if (semctl(sem, 0, SETVAL, arg) == -1) {
        perror("semctl error\n");
    }
    return sem;
}

/* semaphore operation wrapper function */
static void semCall(sem_t sem, int op) {
```

```
    struct sembuf sb;
    sb.sem_num = 0;
    sb.sem_op = op;
    sb.sem_flg = 0;
    // operate on semaphore
    if (semop(sem, &sb, 1) == -1) {
        perror("semop error\n");
    }
}

/* semaphore wait function */
void semWait(sem_t sem) {
    semCall(sem, -1);
}

/* semaphore signal function */
void semSignal(sem_t sem) {
    semCall(sem, 1);
}
```

- **共享内存 IPC** `main.c`

```
/*
 *  main.c  --  shared memory communication main program
 */

#include "shmcom.h"

void main() {
    pid_t pid;
    int status;
    char *segaddr; // shared memory entry address
    int sid;  // shared memory id
    sem_t empty, full;

    // get shared memory segment
    if ((sid = shmget(SHMCOM_KEY, SHM_SIZE, IPC_CREAT|0666)) == -1) {
        perror("shmget error\n");
    }
    // attach shared memory segment
    segaddr = shmat(sid, 0, 0);
    // create two semaphores (init val = 1)
    empty = createSem(SEM_EMPTY_KEY);
    full = createSem(SEM_FULL_KEY);
    // set `full` = 0 (init empty buffer)
    semWait(full);

    // communication between parent and child process
    while ((pid = fork()) == -1);
    if (pid) {
        /* parent process: read and store */
        char buffer_p[BUFFER_SIZE];
        FILE *in = fopen("../test/test.in", "r");

        if (fread(buffer_p, sizeof(char), BUFFER_SIZE, in) > 0) {
            semWait(empty);
            strcpy(segaddr, buffer_p);
```

```
            semSignal(full);
        }

        if (wait(&status) == -1) {
            perror("wait error\n");
        }

        fclose(in);
    }
    else {
        /* child process: load and write */
        char buffer_c[BUFFER_SIZE];
        int length;
        FILE *out = fopen("../test/test.shm.out", "w");

        semWait(full);
        length = strlen(segaddr);
        strcpy(buffer_c, segaddr);
        semSignal(empty);

        fwrite(buffer_c, sizeof(char), length, out);

        fclose(out);
        exit(0);
    }

    exit(0);
}
```

【6】测试结果

- **父子进程通信**

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/pccom# ./pccom
[Parent] Start parent process: 411776
[Child] Start child process: 411777
[Parent] Send signal...
[Parent] Wait for child to end...
[Signal] Start signal handler of SIGUSR1
[Child] Start executing `uname -a`...
Linux iZuf63xs8u1971bor8zpc1Z 5.4.0-53-generic #59-Ubuntu SMP Wed Oct 21 09:38:44 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
[Parent] Child process exited with status: 0
[Parent] Parent process ended
```

- **消息通信 IPC**
  - 注：读入的文件 `test.in` 内容如下：

    ```
    hello world! This is a test file for inter-process communication.
    Copyright 2021
    Shanghai Jiao Tong University
    School of Cyber Science and Engineering
    Operating System
    Unix System V Programming
    Process Management and Communication
    ```

  - 在一个终端窗口运行 msgserver

    ```
    root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/msgcom# ./msgserver
    ```

  - 在另一个终端窗口运行 msgclient

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/msgcom# ./msgclient
Received message from server process: client pid 411789
```

- 回到运行 msgserver 的窗口

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/msgcom# ./msgserver
Received message from client process:
hello world! This is a test file for inter-process communication.
Copyright 2021
Shanghai Jiao Tong University
School of Cyber Science and Engineering
Operating System
Unix System V Programming
Process Management and Communication
```

- 比较输入和输出文件，说明通信成功

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/test# diff test.in test.msg.out
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/test#
```

- **共享内存 IPC**
  - 注：读入文件同上
  - 直接运行 main 进行测试，比较输入和输出文件，说明通信成功

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/shmcom# ./main
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/shmcom# diff ../test/test.in ../test/test.shm.out
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/shmcom# cat ../test/test.shm.out
hello world! This is a test file for inter-process communication.
Copyright 2021
Shanghai Jiao Tong University
School of Cyber Science and Engineering
Operating System
Unix System V Programming
Process Management and Communicationroot@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/shmcom#
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab2/shmcom#
```

【7】实验心得

本次实验总体上相对简单，在实践过程中对 UNIX 进程控制相关的系统调用，以及 UNIX 的消息通信机制、共享内存机制有了深入的理解和实际的应用，也在这一过程中提高了我查阅 API 文档的能力，提高了 C 语言系统编程的规范性，提高了调试技巧（此次实验多次使用 GDB 进行多进程调试）。非常感谢老师上课的指导和帮助，使得我们对进程控制和进程间通信有了宏观而整体的把握。

源代码 GitHub 地址：https://github.com/zhliuworks/OS-lab/tree/master/Lab2