实 验 报 告

课程名称：操作系统试验

实 验 一：进程管理与进程间通信

班　　级：

学生姓名：胡诚皓

学　　号：20201060330

专　　业：计算机科学与技术

指导教师：杨旭涛

学　　期：2022－2023 学年秋季学期

成　　绩：

云南大学信息学院

一、实验目的

1、熟悉 Linux 系统下 fork()函数的使用，并观察系统中进程创建和执行的并发执行情况

2、观察和了解软中断实现的进程通信；

3、观察和了解通过消息队列实现进程通信的过程；

4、观察和了解通过共享存储区实现进程通信的过程。

5、掌握进程对共享存储区或变量访问时进程互斥的实现方法。

二、知识要点

1、创建进程函数 fork()；

2、进程的并发执行；

3、中断调用，软中断与硬件中断；

4、消息队列通信，共享存储区通信；

5、进程的互斥访问。

三、实验预习（要求做实验前完成）

1、了解 Linux 系统中常用命令的使用方法；

2、掌握进程的并发执行、系统调用、中断、进程通信、进程互斥、进程同步的基本概念。

四、实验内容和试验结果

结合课程所讲授内容以及课件中的试验讲解，完成以下试验。请分别对试验过程和观察到的情况做描述和总结，并将试验结果截图附后。

1、观察进程的并发执行：用 fork 系统调用创建多个进程，各进程输出不同的内容。观察进程的行为。

① 进程的执行-1

这段程序中，父进程显示字符‘a’，两个子进程分别显示字符‘b’和‘c’。在下面的截图中可以发现程序的输出顺序基本为 abc，但其中有一次输出了 bac，说明三个进程被调度的顺序发生了改变。

```
abcgeorgehu@realDebian:~/OSexp$ ./exp1_1
abcgeorgehu@realDebian:~/OSexp$ ./exp1_1
abcgeorgehu@realDebian:~/OSexp$ ./exp1_1
abcgeorgehu@realDebian:~/OSexp$ ./exp1_1
bacgeorgehu@realDebian:~/OSexp$ ./exp1_1
abcgeorgehu@realDebian:~/OSexp$ ./exp1_1
abcgeorgehu@realDebian:~/OSexp$ ./exp1_1
abcgeorgehu@realDebian:~/OSexp$ ./exp1_1
abcgeorgehu@realDebian:~/OSexp$
```

② 进程的执行-2

父进程和子进程的功能就是输出自己的身份。由于输出过于庞大，将输出重定向到文件进行查看。下面是观察到的几个现象。

（1）输出的字符混起来了，下图中 pareson 事实上是 parent 未输出完成时，父进程被调度出 CPU，son 进程开始输出。后来，父进程重新被调度进 CPU 输出了未输出完的部分。这充分说明了每一条 C 语句其实包含了好多个 CPU 要执行的指令，很可能执行到一半时就被打断。

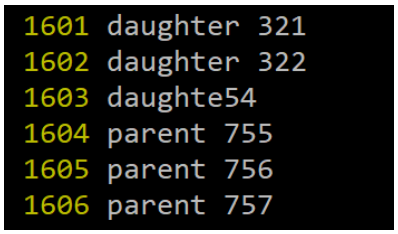
```
380 parent 379
381 parent 380
382 parent 381
383 pareson 0
384 son 1
385 son 2
386 son 3
```

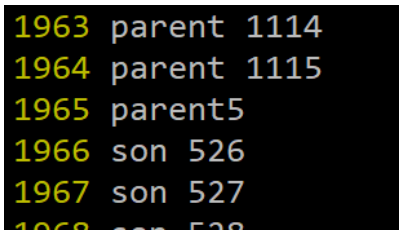
```
905 son 522
906 son 523
907 son 524
908 son 52nt 382
909 parent 383
910 parent 384
911 parent 385
```

（2）对时间片轮转的观察。尝试调整循环次数，观察输出，发现 for 语句中的 printf 循环的执行次数与是否被调度并没有非常直接的关系。这说明操作系统除了单纯的时间片轮转外，还应用有一些优化算法。

```
906 son 523
907 son 524
908 son 52nt 382
909 parent 383
910 parent 384
```

```
1277 parent 751
1278 parent 752
1279 parent 753
1280 parent 7daughter 0
1281 daughter 1
1282 daughter 2
```





对 Linux 中用于创建子进程的函数 fork()的理解

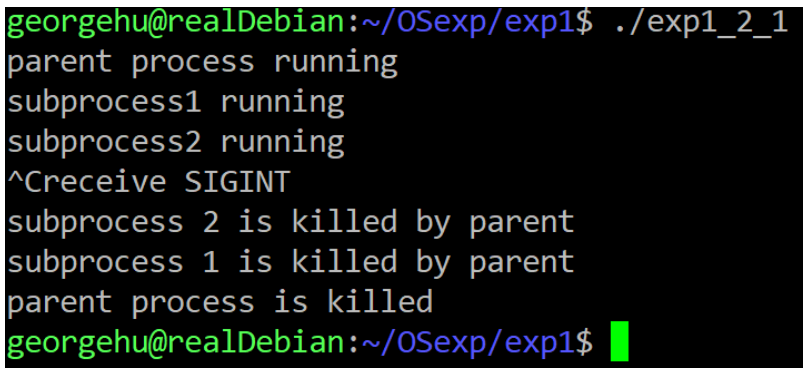
fork 函数用于创建子进程，是 Linux 内核提供给上层应用使用的一个系统调用，它会创建一个子进程作为父进程的副本，运行同样的代码。fork 本来是叉子的意思，父进程调用 fork 后，就像叉子前面的叉一样和子进程分开了，正常执行过程中各自独立运行，并且 PCB 中的代码都是一样的。

在 fork 函数内部，通过对子进程 PCB 中寄存器保存段的返回值寄存器赋值为 0 实现子进程的 fork 函数返回 0，父进程获得的返回值就是子进程的 PID。

2、软中断通信：用 fork 创建两个子进程，父进程响应键盘上来的中断信号（ctrl+c），调用 kill 系统调用向两个子进程发出信号（16、17 号软中断），子进程收到信号后，输出信息并结束。子进程结束后，父进程输出信息并结束。

① 软中断通信测试

在子进程 2 中，将 SIGINT 信号与自己写的 pt 函数进行绑定。pt 的功能就是输出“receive SIGINT”



从上图中可以得知，在终端中发起 ctrl+c 的软中断 SIGINT 后，不止父进程，子进程 2 也收到了这个信号。父进程在给两个子进程分别发送 16、17 信号后，使用 wait 函数等待子进程的结束。

在进一步测试和查找资料后，向某个处于前台的进程发起软中断信号，所有处于同一个组的进程（即 PGID 相同）都会收到此中断信号。换言之，若是父进程处于后台，手动向父进程发起软中断信号，处于同一组的进程并不会收到该中断信号。

```
georgehu@realDebian:~/OSexp/exp1$ ./exp1_2_1
parent process running
subprocess1 running
subprocess2 running
^Z
[1]+  Stopped                 ./exp1_2_1
georgehu@realDebian:~/OSexp/exp1$ jobs
[1]+  Stopped                 ./exp1_2_1
georgehu@realDebian:~/OSexp/exp1$ bg 1
[1]+ ./exp1_2_1 &
georgehu@realDebian:~/OSexp/exp1$ ps o pid,pgid,ppid,comm
    PID    PGID    PPID COMMAND
   1340    1340    1339 bash
   1415    1415    1340 exp1_2_1
   1416    1415    1415 exp1_2_1
   1417    1415    1415 exp1_2_1
   1419    1419    1340 ps
georgehu@realDebian:~/OSexp/exp1$ kill -2 1415
subprocess 1 is killed by parent
subprocess 2 is killed by parent
georgehu@realDebian:~/OSexp/exp1$ parent process is killed

[1]+  Done                    ./exp1_2_1
georgehu@realDebian:~/OSexp/exp1$
```

上图中，首先启动编译好的可执行程序以开启三个进程，接着按下 Ctrl+Z 挂起前台进程并将其转到后台，然后使用 bg 唤起进程，使其在后台运行。此时查看进程的情况，父进程的 PID 为 1415，两个子进程的 PID 为 1416 与 1417，三个进程都处于 1415 进程组中。使用 kill 命令向父进程发起 SIGINT（代码为 2）软中断，可以发现在两个进程结束后父进程也结束，并没有出现“receive SIGINT”，即子进程没有收到 SIGINT 信号。说明“向某个处于前台的进程发起软中断信号，所有处于同一个组的进程（即 PGID 相同）都会收到此中断信号”的说法是正确的。

② 进程的管道通信

代码执行结果如下，使用管道通信还是较为方便的。PPT 中给的源代码在写入管道后调用 sleep 进行等待，在反复思考过后，认为并没有必要。两个子进程在写入前进行加锁，写入后进行解锁，并不会相互干扰。另外父进程在读取管道中内容进行输出之前，使用了 wait 等待子进程执行完毕，不会出现还没写完就开始读取的情况。注释掉 sleep(1)后重新编译，反复执行测试后并没有发现有什么问题。

```
georgehu@realDebian:~/OSexp/exp1$ ./exp1_2_2
parent
p1
p2
[readpipe] child 1 process is sending a message!
[readpipe] child 2 process is sending a message!
georgehu@realDebian:~/OSexp/exp1$ ./exp1_2_2
```

3、消息队列通信：用 fork 创建两个子进程，第一个子进程（Server）创建消息队列，等待接收消息；第二个子进程（Client）打开消息队列，向消息队列中写长度为 1KB 的消息，循环 10 次。Server 从消息队列接收每一条消息。

为了进一步搞清楚 msgsnd 函数调用时，内核确实将发送的信息拷贝了一份到缓存队列中，将发送和接受使用的存储消息的临时变量分离。将 CLIENT 和 SERVER 函数改为下面的代码。

```
typedef struct msgform {
    long mtype;
    char mtrex[1024];
} msgform;

msgform msg;

void CLIENT() {
    msgqid = msgget(MSGKEY, 0777|IPC_CREAT);
    for (int i = 1; i <= 10; i++) {
        msg.mtype = i;
        sprintf(msg.mtrex, "This is message %d", i);
        msgsnd(msgqid, &msg, 1024, 0);
        printf("[client] message %d sent!\n", i);
    }
    exit(0);
}
void SERVER() {
    msgform tmp;
    msgqid = msgget(MSGKEY, 0777|IPC_CREAT);
    do {
        msgrcv(msgqid, &tmp, 1024, 0, 0);
        printf("[server] msg %d received: \"%s\"\n", tmp.mtype,
tmp.mtrex);
    } while(tmp.mtype != 10);
    msgctl(msgqid, IPC_RMID, 0);
    exit(0);
}
```

下图为运行结果。可以发现每个消息都被正常的接收到了，并没有混乱，说明在调用 msgsnd 的时候系统确实会把要发送的内容拷贝到消息队列中，并不是依赖源内存位置存储的。

另外，可以发现在接受消息队列中的信息时，并没有指定要接收的是第几个，但是可以看到在两个子进程 Client 和 Server 的并发执行过程中，确实是按先进先出的 FIFO 规则进行接收的，即消息“队列”。

```
georgehu@realDebian:~/OSexp/exp1$ ./exp1_3_1
[client] message 1 sent!
[client] message 2 sent!
[client] message 3 sent!
[client] message 4 sent!
[client] message 5 sent!
[client] message 6 sent!
[client] message 7 sent!
[client] message 8 sent!
[client] message 9 sent!
[client] message 10 sent!
[server] msg 1 received: "This is message 1"
[server] msg 2 received: "This is message 2"
[server] msg 3 received: "This is message 3"
[server] msg 4 received: "This is message 4"
[server] msg 5 received: "This is message 5"
[server] msg 6 received: "This is message 6"
[server] msg 7 received: "This is message 7"
[server] msg 8 received: "This is message 8"
[server] msg 9 received: "This is message 9"
[server] msg 10 received: "This is message 10"
georgehu@realDebian:~/OSexp/exp1$
```

若在 SERVER 函数中 do-while 循环开始前，先使用 msgrcv 函数接收一次消息队列中的信息，结果如下，msg1 没有被显示，即队列中被读取掉了一个消息。

```
georgehu@realDebian:~/OSexp/exp1$ ./exp1_3_1
[client] message 1 sent!
[client] message 2 sent!
[client] message 3 sent!
[client] message 4 sent!
[client] message 5 sent!
[client] message 6 sent!
[client] message 7 sent!
[client] message 8 sent!
[client] message 9 sent!
[client] message 10 sent!
[server] msg 2 received: "This is message 2"
[server] msg 3 received: "This is message 3"
[server] msg 4 received: "This is message 4"
[server] msg 5 received: "This is message 5"
[server] msg 6 received: "This is message 6"
[server] msg 7 received: "This is message 7"
[server] msg 8 received: "This is message 8"
[server] msg 9 received: "This is message 9"
[server] msg 10 received: "This is message 10"
georgehu@realDebian:~/OSexp/exp1$
```

4、共享存储区通信：用 fork 创建两个子进程，两个子进程之间使用共享存储区进行通信。

通过系统维护的一个共享存储区进行进程之间的通信。使用 shmget 来创建和获取共享存储区的 id，以便在后面进行写入和读取。PPT 中的代码使用共享存储区的第一个 int 长度空间作为类似于信号量的东西，使用 while“死等”对方的信号。

显然，这种使用“死等”实现进程同步的方式非常浪费 CPU 资源，此处尝试使用信号量相关的系统调用完成两个子进程的同步，实现交替执行的效果。

主要使用了 sys/sem.h 头文件中的一些函数，其中的函数都是 System V 规范下的，进行信号量的创建，修改以及控制。主要使用到的有下面三个函数。

① int semget(key_t key, int num_sems, int sem_flags)

用于创建或取得一个已有信号量的 id。其中 key 是个整数值，不相关进程可以通过其访问同一个信号量。num_sems 表示需要的信号量个数。sem_flags 用于规定信号量的访问权限以及在信号量不存在时的行为。

② int semop(int sem_id, struct sembuf *sem_opa, size_t num_sem_ops)

用于操作信号量的值，sem_id 为通过 semget 取得的 id 号，*sem_opa 指向定义了操作的结构体数组首地址，num_sem_ops 表示该数组的长度，即需要进行操作的数量。其中用于定义操作的结构体如下

```
struct sembuf{
```

```
    short sem_num; // 要操作的信号量是第几个（从 0 开始）
    short sem_op;  // 信号量需要加上的数值，-1 为 P 操作、1 为 V 操作
    short sem_flg; // 规定操作系统对信号的处理机制

};
```

③ int semctl(int sem_id, int sem_num, int command)

sem_id 为目标信号量的 id，sem_num 为目标信号量是第几个，command 规定了要进行的操作，常用的有用于删除信号量的 IPC_RMID、信号量赋值 SETVAL。

下图为运行的结果，确实为交替运行。在代码中，在两个子进程的函数中，特意在 V 操作之后使用 sleep 以观察两个进程的并发情况。运行过程中，可以观察的到每组 send 和 received 连续显示后会停顿一会儿才会显示下一组。说明两个 Client 和 Server 确实是并发执行的，系统并没有被某进程的 sleep 完全阻塞住。

```
georgehu@realDebian:~/OSexp/exp1$ ./exp1_4_1
[Client]send: 9
[Server]received: 9
[Client]send: 8
[Server]received: 8
[Client]send: 7
[Server]received: 7
[Client]send: 6
[Server]received: 6
[Client]send: 5
[Server]received: 5
[Client]send: 4
[Server]received: 4
[Client]send: 3
[Server]received: 3
[Client]send: 2
[Server]received: 2
[Client]send: 1
[Server]received: 1
[Client]send: 0
[Server]received: 0
georgehu@realDebian:~/OSexp/exp1$
```

下面是修改后的代码：

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

```
#define SHMKEY 75
#define SEM_KEY 70

int shmid, i;
int *addr;
int semid;


union semun {
   int val;
   struct semid_ds *buf;
   unsigned short *array;
   struct seminfo *__buf;
   void *__pad;
};

void P(int sem_id, int sem_num) {
    struct sembuf buf;
    buf.sem_flg = 0;
    buf.sem_op = -1;
    buf.sem_num = sem_num;

    int ret = semop(sem_id, &buf, 1);
    if (ret < 0) {
        printf("P failed on %d.%d\n", sem_id, sem_num);
        return;
    }
}

void V(int sem_id, int sem_num) {
    struct sembuf buf;
    buf.sem_flg = 0;
    buf.sem_op = 1;
    buf.sem_num = sem_num;

    int ret = semop(sem_id, &buf, 1);
    if (ret < 0) {
        printf("V failed on %d.%d\n", sem_id, sem_num);
        return;
    }
}

void CLIENT() {
    shmid = shmget(SHMKEY, 1024, 0777|IPC_CREAT);
```

```
    addr = shmat(shmid, 0, 0);
    semid = semget(SEM_KEY, 2, 0777|IPC_CREAT);
    for (int i = 9; i >= 0; i--) {
        P(semid, 0);
        printf("[Client]send: %d\n", i);
        fflush(stdout);
        *addr = i;
        V(semid, 1);
        sleep(1);
    }
    exit(0);
}

void SERVER() {
    shmid = shmget(SHMKEY, 1024, 0777|IPC_CREAT);
    addr = shmat(shmid, 0, 0);
    semid = semget(SEM_KEY, 2, 0777|IPC_CREAT);
    do {
        P(semid, 1);
        printf("[Server]received: %d\n", *addr);
        fflush(stdout);
        V(semid, 0);
        sleep(1);
    } while (*addr);
    shmctl(shmid, IPC_RMID, 0);
    exit(0);
}

void init_sem(int sem_id, int sem_num, int val) {
    union semun var;
    var.val = val;
    if (semctl(sem_id, sem_num, SETVAL, var) < 0) {
        printf("initialize failed\n");
        exit(-1);
    }
}

void rm_sem(int sem_id, int sem_num) {
    if (semctl(sem_id, sem_num, IPC_RMID) < 0) {
        printf("deleting failed\n");
    }
}

int main() {
```

```
    int p1, p2;
    while((p1 = fork()) == -1);
    if (p1 > 0) {
        while((p2 = fork()) == -1);
        if (p2 > 0) {
            // main process
            semid = semget(SEM_KEY, 2, 0666|IPC_CREAT);
            init_sem(semid, 0, 1);
            init_sem(semid, 1, 0);

            wait();
            wait();

        } else {
            // subprocess 2
            CLIENT();
        }
    } else {
        // subprocess 1
        SERVER();
    }
}
```