进程管理、信号、进程间通信

Q1

> 每个概念被创造都有其意义，请简述“进程”这个概念在Linux系统中有什么用途。

- 进程是描述程序执行过程和资源共享的基本单位
- 主要目的：控制和协调程序的执行

进程是一个二进制程序的执行过程。在Linux操作系统中，向命令行输入一条命令，按下回车键，便会有一个进程被启动。但进程并不是程序。进程存在于内存中，占用系统资源，是抽象的。当一次程序执行结束之后，进程随之消失，进程所用的资源被系统回收。

Q2

```
//代码段A
int i;
for(i=0;i<5;i++){
    pid=fork();
}
```

```
//代码段B
int i;
for(i=0;i<5;i++){
    if(pid=fork()==0)
        break;
}
```

> 阅读以上代码段，回答代码段A和代码段B的执行结果有什么区别？并解释为什么会有这样的区别。

- 代码段A测试：

```
#include<stdio.h>
#include<unistd.h>
int main(){
    int i,pid;
    for(i=0;i<5;i++){
        pid=fork();
        if(pid==0) printf("son\n");
        else if(pid>0) printf("father\n");
        else printf("error\n");
    }
}
```

输出：

```
father
son
father
father
```

```
father
father
father
father
father
father
son
son
son
son
father
father
son
son
father
father
son
father
son
father
son
son
father
father
son
son
son
son
son
father
father
father
son
father
father
father
son
son
father
son
father
son
father
son
son
son
father
father
son
father
son
son
son
father
father
son
son
son
```

生成了31个父进程、31个子进程

- 代码段B测试：

```
#include<stdio.h>
#include<unistd.h>
int main(){
    int i,pid;
    for(i=0;i<5;i++){
        pid=fork();
        if(pid==0) {printf("son\n"); break;}
        else if(pid>0) printf("father\n");
        else printf("error\n");
    }
}
```

输出：

```
father
father
father
father
father
son
son
son
son
son
```

生成了5个父进程，5个子进程

- 区别的原因：代码段B每当`fork()`生成一次子进程，便结束了循环，不继续进行，故产生的进程数即为循环的次数，即5。而代码段A中子进程创建后，还会继续当前循环，直至`i=5`跳出循环，故产生的进程数为$1+2+4+8+16=2^5-1=31$。

Q3

> 用自己的话阐述什么是僵尸进程，并描述进程通过调用`wait()`捕获僵尸态的子进程的过程。

僵尸进程：子进程已结束，但父进程未调用`wait()`等函数等待，即没有被正确清除，成为僵尸进程。

如当进程调用了`exit()`函数后，该进程并不是马上消失，而是留下一个僵尸进程的数据结构——它几乎放弃进程退出前占用的所有内存，既没有可执行代码也不能被调度，只在进程列表中保留一个位置，记载进程的退出状态等信息供父进程收集。若父进程中没有回收子进程的代码，子进程将会一直处于僵尸态。

调用`wait()`函数的进程会被挂起，进入阻塞状态，直到子进程变为僵尸态，`wait()`函数捕获到该子进程的退出信息时才会转为运行态，回收子进程资源并返回；若没有变为僵尸态的子进程，`wait()`函数会让进程一直阻塞。若当前进程有多个子进程，只要捕获到一个变为僵尸态的子进程的信息，`wait()`函数就会返回并使进程恢复执行。

Q4

> 请简述信号在Linux系统中的作用。

- 信号是发送给进程的特殊异步消息
- 当进程接收到信息时立即处理，此时并不需要完成当前函数调用甚至当前代码行
- 信号被应用于进程间通信

Q5

> 请简述信号什么时候处于未决状态，并简述信号存在未决状态的作用。

- 未决状态：发送的信号被阻塞，无法到达进程，内核就会将该信号的状态设置为未决。
- 有的时候信号可能因为当前的信号被阻塞，而不能及时地被处理，这时候系统会将该信号保存起来，如在调用`sigpending()`时，会有未决状态信号`SIGQUIT`。直到进程解除对此信号的阻塞，才执行递达的动作。
- 信号需要存在未决状态是因为这样能更完整地表示出信号在产生、递达之间的状态。

Q6

> 请设计一种通过信号量来实现共享内存读写操作同步的方式，文字阐述即可，不需要代码实现。（提示：在写进程操作未完成时，需要防止其他进程从共享内存中读取数据）

实现了第13周最后一题的要求：

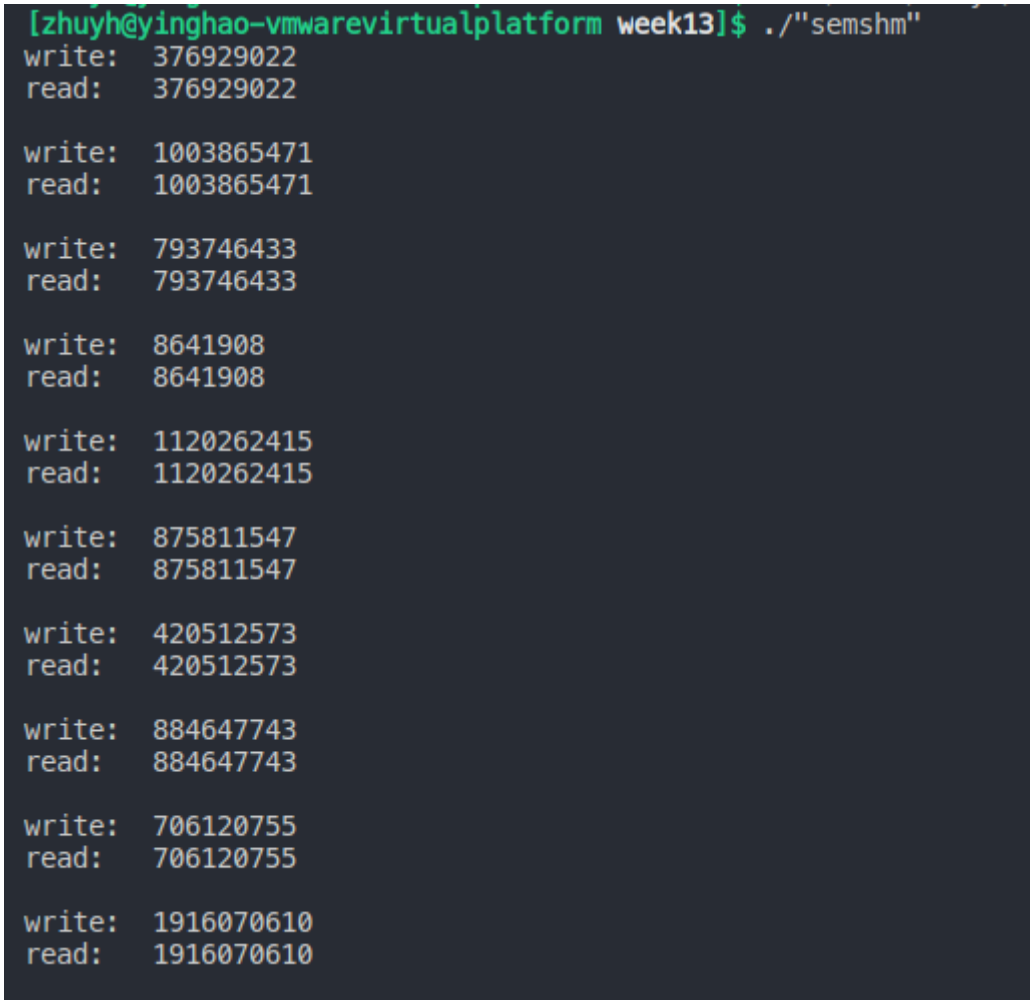
> 在《Linux编程基础》一书对共享内存的讲解中，其给出的例子是一个进程向共享内存写，然后终止，然后再启动一个进程从共享内存中读。你能不能**同时**使用**信号量**和**共享内存**实现一个这样的功能，同时运行两个进程，一个进程向共享内存中写入数据后阻塞，等待另一个进程读，再写，然后再读呢?

```c
#include<stdio.h>
#include<stdlib.h>
#include<sys/sem.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<unistd.h>
#include<string.h>
#define SEGSIZE 4096
typedef struct{
    int age;
}Stu;
//自定义共用体
union semu{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *_buf;
};
static int sem_id;
//设置信号量值
static int set_semvalue(){
    union semu sem_union;
    sem_union.val=1;
    if(semctl(sem_id,0,SETVAL,sem_union)==-1){
        return 0;
    }
    return 1;
}
//P操作，获取信号量
```

```c
static int semaphore_p(){
    struct sembuf sem_b;
    sem_b.sem_num=0;
    sem_b.sem_op=-1;
    sem_b.sem_flg=SEM_UNDO;
    if(semop(sem_id,&sem_b,1)==-1){
        perror("sem_p err");
        return 0;
    }
    return 1;
}
//V操作，释放信号量
static int semaphore_v(){
    struct sembuf sem_b;
    sem_b.sem_num=0;
    sem_b.sem_op=1;
    sem_b.sem_flg=SEM_UNDO;
    if(semop(sem_id,&sem_b,1)==-1){
        perror("sem_v err");
        return 0;
    }
    return 1;
}
//删除信号量
static void del_semvalue(){
    union semu sem_union;
    if(semctl(sem_id,0,IPC_RMID,sem_union)==-1){
        perror("del err");
    }
}
int main(){
    int i;
    int pid;
    int data;
    int shm_id;
    key_t key;
    Stu *smap;
    struct shmid_ds buf;
    srand((unsigned int)18373722);
    //sem
    sem_id=semget((key_t)1000,1,0664|IPC_CREAT);//创建信号量
    if(sem_id==-1){
        perror("sem_c err");
        exit(-1);
    }
    if(!set_semvalue()){
        perror("init err");
        exit(-1);
    }
    //shm
    key=ftok("/",0);//获取关键字
    if(key==-1){
        perror("ftok error");
        return -1;
    }
    //创建共享内存
    shm_id=shmget(key,SEGSIZE,IPC_CREAT|IPC_EXCL|0664);
    if(shm_id==-1){
```

```
        perror("create shared memory error\n");
        return -1;
    }
    //将进程与共享内存绑定
    smap=(Stu*)shmat(shm_id,NULL,0);
    //创建子进程
    pid=fork();
    for(i=0;i<10;i++){
        //若创建失败
        if(pid==-1){
            del_semvalue();
            exit(-1);
        }
        else if(pid!=0){
            //子进程中读(使用信号量)
            (smap+i)->age=rand();
            semaphore_p();//获取信号量
            printf("%s\t%d\n","write:",(smap+i)->age);
            fflush(stdout);
            sleep(1);
            semaphore_v();//释放信号量
        }
        else{
            //父进程中写(使用信号量)
            semaphore_p();
            printf("%s\t%d\n\n","read:",(*(smap+i)).age);
            fflush(stdout);
            sleep(1);
            semaphore_v();
        }
    }
    //删除信号量
    if(pid>0){
        wait(NULL);
        del_semvalue();
    }
    //解除绑定
    if(shmdt(smap)==-1){
        perror("detach error");
        return -1;
    }
    //删除共享内存
    shmctl(shm_id,IPC_RMID,&buf);
    return 0;
}
```

- 运行截图：


[zhuyh@yinghao-vmwarevirtualplatform week13]$ ./"semshm"
write: 376929022
read: 376929022
write: 1003865471
read: 1003865471
write: 793746433
read: 793746433
write: 8641908
read: 8641908
write: 1120262415
read: 1120262415
write: 875811547
read: 875811547
write: 420512573
read: 420512573
write: 884647743
read: 884647743
write: 706120755
read: 706120755
write: 1916070610
read: 1916070610