

# 实验一 创建进程

## 【实验目的】

学会通过基本的 Windows或者 Linux进程控制函数，由父进程创建子进程，并实现父子进程协同工作。

## 【实验软硬件环境】

Ubuntu 16.0.4 on VMware Station

## 【实验内容】

创建两个进程，让子进程读取一个文件，父进程等待子进程读取完文件后继续执行， 实现进程协同工作。

进程协同工作就是协调好两个进程，使之安排好先后次序并以此执行，可以用等待函数来实现这一点。当需要等待子进程运行结束时，可在父进程中调用等待函数。

## 【实验程序及分析】

### 1. 程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types>
#include <sys/wait.h>
int main()
{
    int pid;
    pid=fork();
    if(pid==-1){
        printf("ERROR!\n");
    }

    else if(pid==0){
```

```
        printf("Process of child!\n");
    }
    else {
        int y=waitpid(-1,NULL,0);
        printf("Process of father!\n");
    }
    return 0;
}
```

## 2. 代码分析

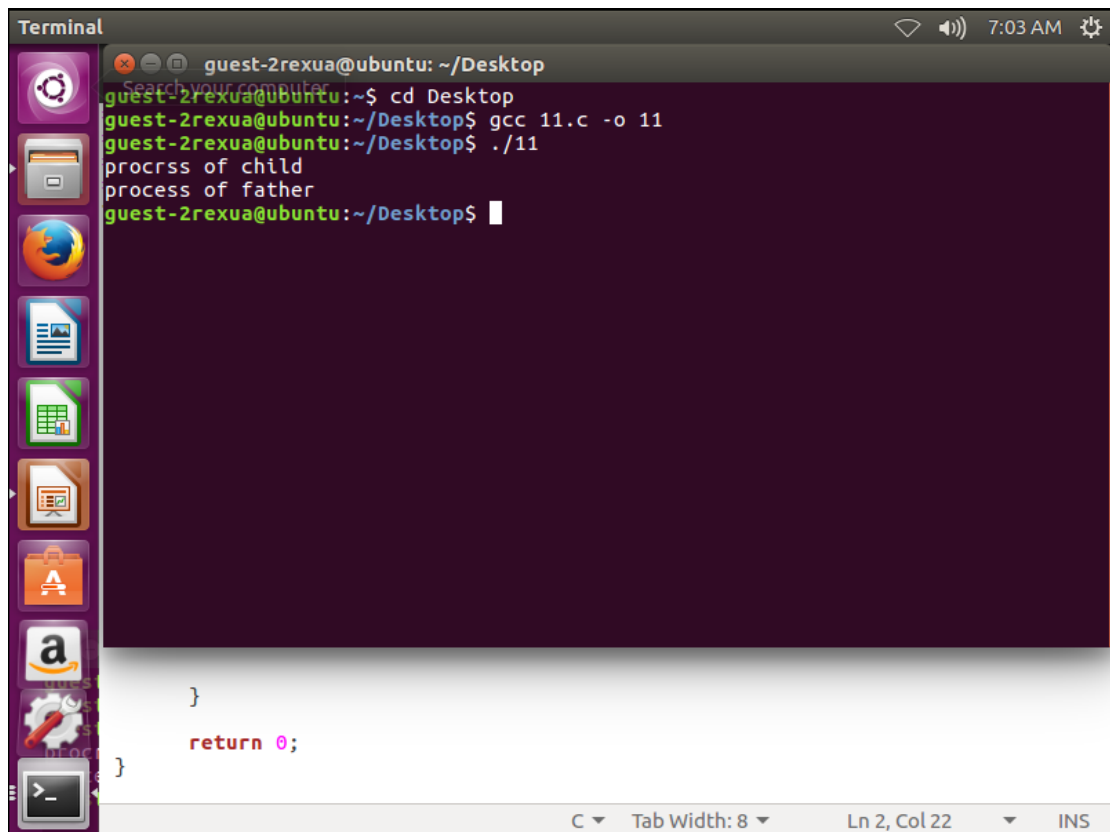
`pid=fork()`，创建进程但子进程共享父进程的地址空间。

创建进程的结果有两种返回值： 若出错，返回-1，打印输出相关的ERROR信息；若成功调用一次则返回两个值，子进程返回0，父进程返回子进程ID。

子进程返回0时，打印子进程的相关信息。

父进程返回子进程ID时，打印父进程的相关信息调用，`waitpid()`函数暂时停止目前进程的执行,直到有信号来到或子进程结束。如果在调用 `waitpid()`时子进程已经结束,则 `waitpid()`会立即返回子进程结束状态值。`y=waitpid(-1,NULL,0)` 第一个参数为欲等待的子进程识别码`pid`, `pid=-1` 等待任何子进程,相当于`wait()`；第二个参数为子进程的结束状态值参数`status`，不在意结束状态值,则参数 `status` 可以设成 `NULL`；第三个参数为参数`options`，提供了一些额外的选项来控制`waitpid`，我们不必使用它们，就可以把`options`设为0。

## 【实验结果截图】



```
Terminal
guest-2rexua@ubuntu: ~/Desktop
guest-2rexua@ubuntu:~$ cd Desktop
guest-2rexua@ubuntu:~/Desktop$ gcc 11.c -o 11
guest-2rexua@ubuntu:~/Desktop$ ./11
process of child
process of father
guest-2rexua@ubuntu:~/Desktop$
```

The screenshot shows a terminal window with a dark purple background. The prompt is 'guest-2rexua@ubuntu: ~/Desktop'. The user enters 'cd Desktop', then 'gcc 11.c -o 11', and finally './11'. The output of the program is 'process of child' followed by 'process of father' on the next line. The terminal window has a title bar with 'Terminal' and system icons. On the left, there is a sidebar with various application icons. At the bottom, there is a status bar showing 'C', 'Tab Width: 8', 'Ln 2, Col 22', and 'INS'.

## 【实验心得体会】

- 1、通过本次实验，我初步熟悉了在Linux系统上编写，运行C语言代码的基本操作，也熟悉了Linux操作系统的一些简单操作，比如Ctrl+Alt+T调出cmd命令窗口，感受到了Linux操作系统的一些优点。
- 2、学会了通过基本的Linux进程控制函数，掌握了fork(),waitpid()等函数的运用，对其中的一些参数了解更深刻了。懂得了由父进程创建子进程，并实现父子进程协同工作。

## 实验二 线程共享进程数据

### 【实验目的】

了解线程与进程之间的数据共享关系。创建一个线程，在线程中更改进程中的数据。

### 【实验软硬件环境】

ubuntu 16.0.4 on VMware Station

### 【实验内容】

在进程中定义全局共享数据，在线程中直接引用该数据进行更改并输出该数据。

### 【实验程序及分析】

#### 1.程序代码

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int a = 666;
void *create(void *arg)
{
    printf("new pthread...\n");
    printf("shared data: a = %d\n",a);
    return 0;
}
int main(int argc, char* argv[])
{
    pthread_t T;
    if(pthread_create(&T,NULL,create,argv[1]))
    {
        printf("failed to create thread\n");
        return -1;
    }
}
```

```
}

sleep(1);
printf("Create thread success.\n");
return 0;
}
```

## 2.代码分析

首先声明一个名为T的 pthread\_t 线程ID。

接着使用pthread\_create(&T,NULL,create,argv[1])创建线程。

第一个参数为指向线程标识符的指针。

第二个参数用来设置线程属性。

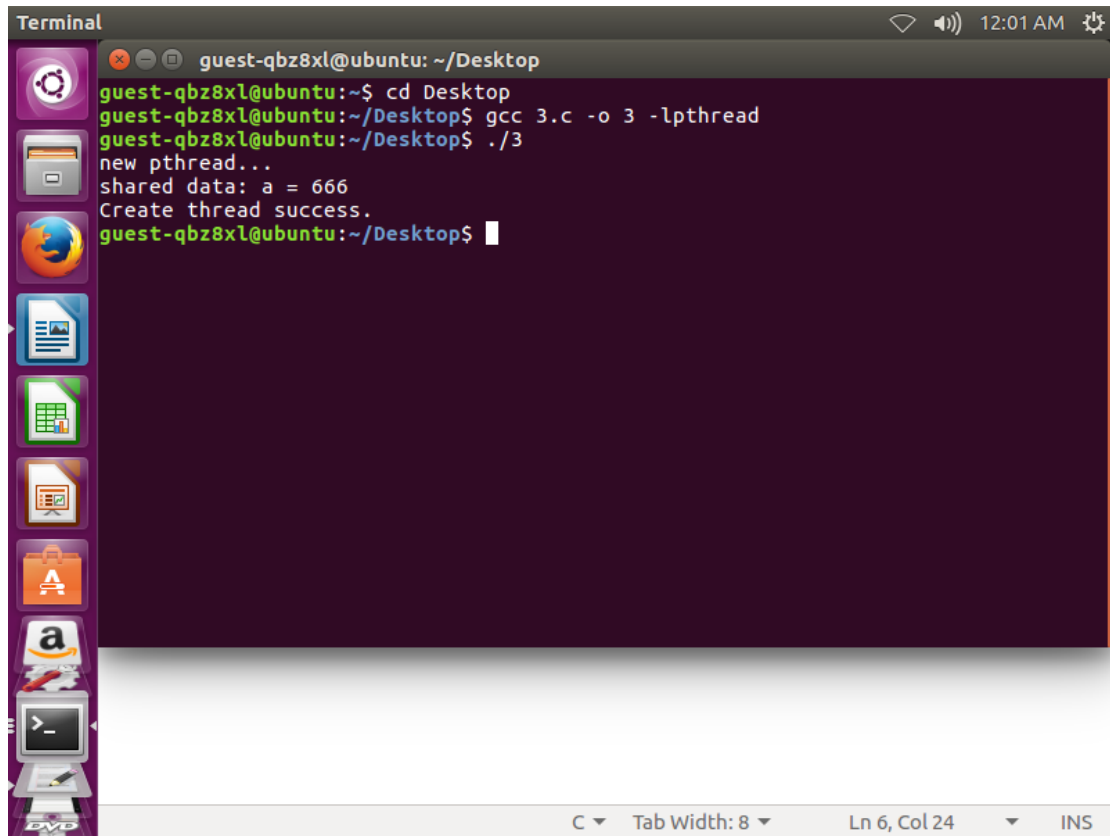
第三个参数是线程运行函数的起始地址。

最后一个参数是运行函数的参数。

其中create()函数是自己自定义的，里面包含提示产生新线程的输出以及打印线程和进程中共享的数据信息。

若创建线程失败，则打印failed to create thread的信息;若创建线程成功，则会调用create()函数，输出提示产生新线程的输出以及打印线程和进程中共享的数据信息。

### 【实验结果截图】



```
Terminal
guest-qbz8xl@ubuntu: ~/Desktop
guest-qbz8xl@ubuntu:~$ cd Desktop
guest-qbz8xl@ubuntu:~/Desktop$ gcc 3.c -o 3 -lpthread
guest-qbz8xl@ubuntu:~/Desktop$ ./3
new pthread...
shared data: a = 666
Create thread success.
guest-qbz8xl@ubuntu:~/Desktop$
```

The screenshot shows a terminal window titled "Terminal" with a dark background. The prompt is "guest-qbz8xl@ubuntu: ~/Desktop". The user enters "cd Desktop", then "gcc 3.c -o 3 -lpthread", and finally ". /3". The output shows "new pthread...", "shared data: a = 666", and "Create thread success.". The terminal window has a sidebar with various application icons and a status bar at the bottom showing "C", "Tab Width: 8", "Ln 6, Col 24", and "INS".

### 【实验心得体会】

1. 通过本次实验，我成功运用了pthread\_create 函数来创建线程，也知道了线程与进程中的数据是共享的，加深了对课本所学知识的理解。
2. 我还知道了在创建线程的代码中，编译时注意加上-lpthread参数，以调用链接库，因为pthread并非Linux系统的默认库，还学到了使用sleep()以更清楚实验的运行过程。

## 实验三 信号通信

### 【实验目的】

利用信号通信机制在父子进程及兄弟进程间进行通信

### 【实验软硬件环境】

ubuntu 16.0.4 on VMware Station

### 【实验内容】

父进程创建一个有名事件，由子进程发送事件信号，父进程获取事件信号后进行相应的处理。

### 【实验程序及分析】

#### 1.程序代码

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <wait.h>

void process()
{
    printf("child process---%d\n",getpid());
}

int main()
{
    signal(SIGUSR2,process);
    pid_t pid;
    pid=fork();
    if(pid==0){
        printf("...child process start...\n");
        printf("send a signal:%d\n",getpid());
        kill(getppid(),SIGUSR2);
    }
```

```

        printf("...child process finished...\n");
        exit(0);
    }
    else if(pid>0){
        printf("...father process start...\n");
        printf("receive a signal %d from child \n
father:%d\n",pid,getpid());
        waitpid(pid,NULL,0);
        printf("...father process finished...\n");
    }
    else{
        printf("ERROR\n");
    }

    return 0;
}

```

## 2.代码分析

首先用`signal(SIGUSR2,process)` , `SIGUSR2`为用户自定义`signal 2` , `process`函数为描述了与信号关联的动作(打印出子进程的进程号)。

再用`fork()`函数创建子进程, 在`fork`函数执行完毕后, 如果创建新进程成功, 则出现两个进程, 一个是子进程, 一个是父进程。

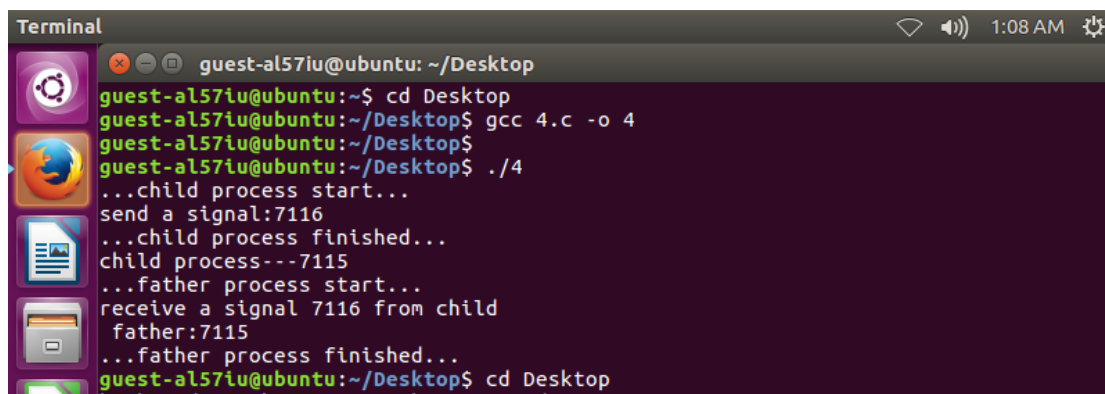
在子进程中, `fork` 函数返回0, 打印子进程开始的信息, 以及发送给父进程的信号值, `kill(getppid(),SIGUSR2)` 用于向父进程发送事件信号, 最后输出子进程结束提示。

在父进程中, `fork`函数返回新创建子进程的进程 ID 。打印父进程开始的信息, 以及输出子进程发送给父进程的信号值, 使用 `waitpid()` 函数暂时停止目前进程的执行,直到有信号来到或子进程结束, 最后输出父进程结束提示。



若创建进程失败，则打印创建失败提示信息。

### 【实验结果截图】



```
Terminal
guest-al57iu@ubuntu: ~/Desktop
guest-al57iu@ubuntu:~$ cd Desktop
guest-al57iu@ubuntu:~/Desktop$ gcc 4.c -o 4
guest-al57iu@ubuntu:~/Desktop$ ./4
...child process start...
send a signal:7116
...child process finished...
child process---7115
...father process start...
receive a signal 7116 from child
father:7115
...father process finished...
guest-al57iu@ubuntu:~/Desktop$ cd Desktop
```

### 【实验心得体会】

1. 通过本次实验，我不仅熟练使用了之前所学到的`fork`，`waitpid` 等函数，也学到了`signal`，`kill`等新函数的使用方法，了解到了这些新函数内参数的意思。
2. 我学会了利用信号通信机制在父子进程间进行通信的编程方法，由子进程发送事件信号，父进程获取事件信号后进行相应的处理，也懂得了打印相关进程信息来表明自己的程序是否正确，使得程序易于理解。

## 实验四 匿名管道通信

### 【实验目的】

学习使用匿名管道在两个进程间建立通信

### 【实验软硬件环境】

ubuntu 16.0.4 on VMware Station

### 【实验内容】

分别建立名为Parent 的单文档应用程序和 Child 的单文档应用程序作为父子进程，由父进程创建一个匿名管道，实现父子进程向匿名管道写入和读取数据。

### 【实验程序及分析】

#### 1. 程序代码

```
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 1000
int main(){
    int fd[2];
    char buf[MAX];
    pid_t pid;
    int len;
    if(pipe(fd)<0)
    {
        perror("failed to pipe\n");
        exit(1);
    }
    if((pid=fork())<0)
    {
        perror("failed to fork\n");
```

```

        exit(1);
    }
    else if(pid>0)
    {
        printf("father write sucess\n");
        write(fd[1],"success\n",MAX);
        sleep(1);
        read(fd[0],buf,MAX);
        printf("father read content: %s\n",buf);
        exit(0);
    }
    else
    {
        printf("child read from pipe\n");
        read(fd[0],buf,MAX);
        printf("child write reply to pipe\n");
        write(fd[1],"data\n",MAX);
    }
    return 0;
}

```

## 2.代码分析

首先分别创建匿名管道和进程。

若`pipe(fd)`创建的管道失败，则返回值为-1，即为小于0，打印创建管道失败的相关信息。

若创建进程失败，则打印进程创建失败的相关信息。

若创建成功，则在`fork`函数执行完毕后，则出现两个进程，一个是子进程，一个是父进程。

在父进程中，`fork` 函数返回新创建子进程的进程 ID，大于0。

将“ success” 写入`fd[1]`端，然后从管道读端读取数据并放入缓冲区，打印

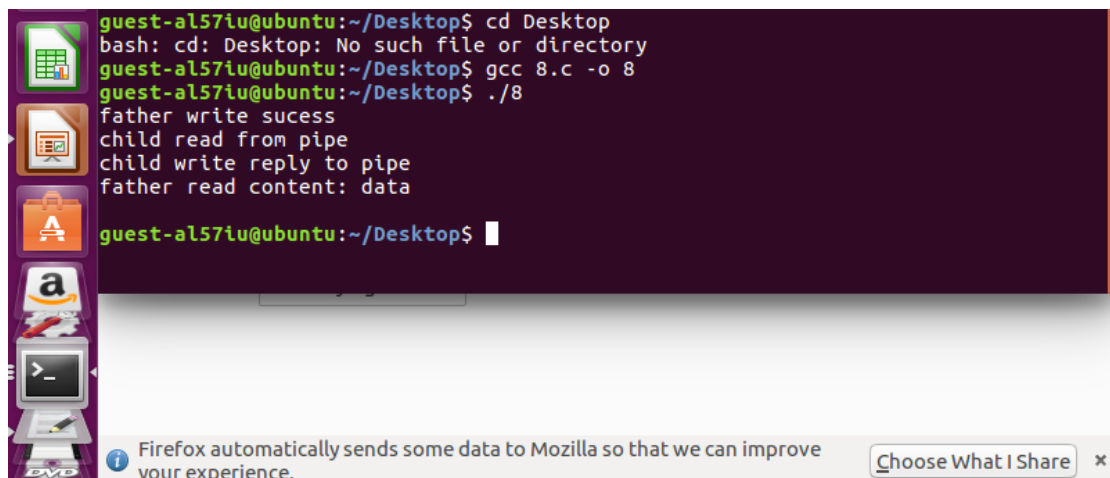
“父进程关闭写管道成功”提示信息。

在子进程中，**fork** 函数返回0。从管道读端读取数据并放入人缓冲区打印“子进程读取数据成功”提示信息，并输出缓冲区数据。

在 **pipe[2]** 中，管道两端可分别用描述字 **fd[0]** 以及 **fd[1]** 来描述，即一端只能用于读，由描述字 **fd[0]** 表示，称其为管道读端；另一端则只能用于写，由描述字 **fd[1]** 来表示，称其为管道写端。

**Read**和**write**函数第一个参数**fd**是有效文件描述符，第二个参数**buf**为指向缓冲区的指针，第三个**length**为缓冲区的大小（我使用了宏定义的**MAX**值）

### 【实验结果截图】



```
guest-al57iu@ubuntu:~/Desktop$ cd Desktop
bash: cd: Desktop: No such file or directory
guest-al57iu@ubuntu:~/Desktop$ gcc 8.c -o 8
guest-al57iu@ubuntu:~/Desktop$ ./8
father write sucess
child read from pipe
child write reply to pipe
father read content: data
guest-al57iu@ubuntu:~/Desktop$
```

### 【实验心得体会】

1. 通过本次实验，我掌握了建管道的基本方法，学会了**pipe**,**write**,**read**等函数的使用方法。
2. 学习了使用匿名管道在两个进程间建立通信，加深了对书本知识的理解。

## 实验五 信号量实现进程同步

### 【实验目的】

进程同步是操作系统多进程/多线程并发执行的关键之一，进程同步是并发进程为了完成共同任务采用某个条件来协调他们的活动，这是进程之间发生的一种直接制约关系，本次试验是利用信号量进行进程同步

### 【实验软硬件环境】

ubuntu 16.0.4 on VMware Station

### 【实验内容】

生产者进程生产产品，消费者进程消费产品。

当生产者进程生产产品时，如果没有空缓冲区可用，那么生产者进程必须等待消费者进程释放出一个缓冲区。

当消费者进程消费产品时，如果缓冲区中没有产品，那么消费者进程将被阻塞，直到新的产品被生产出来。

### 【实验程序及分析】

#### 1.程序代码

```
#include<stdio.h>
#include<unistd.h>
#include<semaphore.h>
#include<pthread.h>
#include<string.h>
#include<stdlib.h>
#include<sys/sem.h>

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

```
int full,empty,mutex;
int semaphore_p(int semId);
int semaphore_v(int semId);
int setValue(int semId,int value);
int i;

int main()
{
    full = semget((key_t)1234, 1, 0666 | IPC_CREAT);
    setValue(full, 0);
    empty = semget((key_t)1235, 1, 0666 | IPC_CREAT);
    setValue(empty, 6);
    mutex = semget((key_t)1236, 1, 0666 | IPC_CREAT);
    setValue(mutex, 1);
    pid_t customerOne, customerTwo;
    for(i = 0; i < 2; i++)
    {
        pid_t temp = fork();
        if(temp == 0)
        {
            if(i == 0) customerOne = getpid();
            if(i == 1) customerTwo = getpid();
            break;
        }
    }
    if(getpid() == customerOne)
    {
        while(1)
        {
            sleep(3);
            semaphore_p(full);
            semaphore_p(mutex);
```

```

        int value = semctl(full, 0, GETVAL, 0);
        printf("Customer %d: use 1 thing, now there are %d
things\n", getpid(), value);
        semaphore_v(mutex);
        semaphore_v(empty);
    }
}
else if(getpid() == customerTwo)
{
    while(1)
    {
        sleep(3);
        semaphore_p(full);
        semaphore_p(mutex);
        int value = semctl(full, 0, GETVAL, 0);
        printf("Customer %d: use 1 thing, now there are %d
things\n", getpid(), value);
        semaphore_v(mutex);
        semaphore_v(empty);
    }
}
else
{
    while(1)
    {
        sleep(1);
        semaphore_p(empty);
        semaphore_v(full);
        semaphore_p(mutex);
        int value = semctl(full, 0, GETVAL, 0);
        printf("Producer %d: produce 1 thing, now there
are %d things\n", getpid(), value);
        semaphore_v(mutex);

```

```

    }
}
return 0;
}
int semaphore_p(int semId)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = -1;
    sem_b.sem_flg = SEM_UNDO;
    if(semop(semId, &sem_b, 1) == -1)
    {
        printf("semaphore_p failed\n");
    }
    return 0;
}
int semaphore_v(int semId)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = 1;
    sem_b.sem_flg = SEM_UNDO;
    if(semop(semId, &sem_b, 1) == -1)
        printf("semaphore_v failed\n");
    return 0;
}
int setValue(int semId, int value)
{
    union semun sem_union;
    sem_union.val = value;
    if(semctl(semId, 0, SETVAL, sem_union) == -1)
        printf("Setting %d value is error\n", semId);
    return 0;
}

```



```
}
```

## 2.代码分析

I) 定义3个信号量，full，empty，mutex，分别表示产品个数，缓冲区空位个数，对缓冲区进行操作的互斥信号量，对应的初始化值分别为0，n，1。

生产者：

P（empty）---->P（mutex）----->V（mutex）----->V（full）

消费者：

P（full）----->P（mutex）----->V（mutex）----->V（empty）

II) int semget(key\_t key, int num\_sems, int sem\_flags);

semget函数成功返回一个相应信号标识符（非零），失败返回-1.

第一个参数key是整数值（唯一非零），不相关的进程可以通过它访问一个信号量，它代表程序可能要使用的某个资源，程序对所有信号量的访问都是间接的，程序先通过调用semget函数并提供一个键，再由系统生成一个相应的信号标识符（semget函数的返回值），只有semget函数才直接使用信号量键，所有其他的信号量函数使用由semget函数返回的信号量标识符。如果多个程序使用相同的key值，key将负责协调工作。

第二个参数num\_sems指定需要的信号量数目，它的值几乎总是1。

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。设置了IPC\_CREAT标志后，即使给出的键是一个已有信号量的键，也不会产生错误。而IPC\_CREAT | IPC\_EXCL则可以创建一个新的，唯一的信号量，如果信号量已存在，返回一个错误。

III) int semop(int sem\_id, struct sembuf \*sem\_opa, size\_t num\_sem\_ops);

sem\_id是由semget返回的信号量标识符，sembuf结构的定义如下：

```
struct sembuf{  
    short sem_num;//除非使用一组信号量，否则它为0  
    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，P（等待）操作，一个是+1，即V（发送信号）操作。

`short sem_flg;` //通常为SEM\_UNDO,使操作系统跟踪信号，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

`};`

IV) `int semctl(int sem_id, int sem_num, int command, ...);`

如果有第四个参数，它通常是一个union semun结构，定义如下：

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

前两个参数与前面一个函数中的一样，`command`通常是下面两个值中的其中一个

**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。

**IPC\_RMID:** 用于删除一个已经无需继续使用的信号量标识符。

V) 定义PV操作

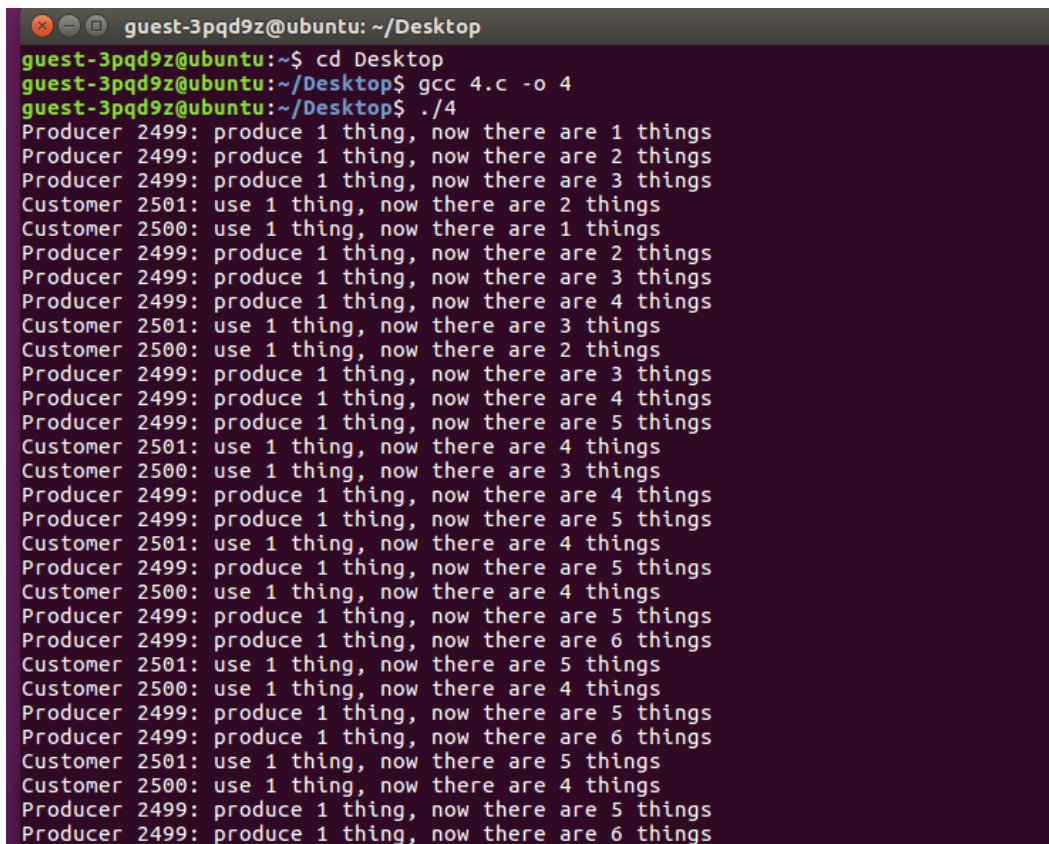
```
Int semaphore_p(int sem_id)
{
    //对信号量做减1操作，即等待P（sv）
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_b.sem_op = -1; //P()
    sem_b.sem_flg = SEM_UNDO;
    if(semop(sem_id, &sem_b, 1) == -1)
    {
        fprintf(stderr, "semaphore_p failed\n");
        return 0;
    }
}
```

```
    return 1;
}
```

## VI) 主函数

在我的主函数中，一共定义了两个消费者和一个生产者，他们消费和生产都有相应的睡眠时间。缓冲区中最大的限制为6，等于6时只能进行消费者的行为，等于0的时候只能进行生产者的操作，如此循环，基本完成了这个功能。

### 【实验结果截图】



```
guest-3pqd9z@ubuntu: ~/Desktop
guest-3pqd9z@ubuntu:~$ cd Desktop
guest-3pqd9z@ubuntu:~/Desktop$ gcc 4.c -o 4
guest-3pqd9z@ubuntu:~/Desktop$ ./4
Producer 2499: produce 1 thing, now there are 1 things
Producer 2499: produce 1 thing, now there are 2 things
Producer 2499: produce 1 thing, now there are 3 things
Customer 2501: use 1 thing, now there are 2 things
Customer 2500: use 1 thing, now there are 1 things
Producer 2499: produce 1 thing, now there are 2 things
Producer 2499: produce 1 thing, now there are 3 things
Producer 2499: produce 1 thing, now there are 4 things
Customer 2501: use 1 thing, now there are 3 things
Customer 2500: use 1 thing, now there are 2 things
Producer 2499: produce 1 thing, now there are 3 things
Producer 2499: produce 1 thing, now there are 4 things
Producer 2499: produce 1 thing, now there are 5 things
Customer 2501: use 1 thing, now there are 4 things
Customer 2500: use 1 thing, now there are 3 things
Producer 2499: produce 1 thing, now there are 4 things
Producer 2499: produce 1 thing, now there are 5 things
Customer 2501: use 1 thing, now there are 4 things
Producer 2499: produce 1 thing, now there are 5 things
Customer 2500: use 1 thing, now there are 4 things
Producer 2499: produce 1 thing, now there are 5 things
Producer 2499: produce 1 thing, now there are 6 things
Customer 2501: use 1 thing, now there are 5 things
Customer 2500: use 1 thing, now there are 4 things
Producer 2499: produce 1 thing, now there are 5 things
Producer 2499: produce 1 thing, now there are 6 things
Customer 2501: use 1 thing, now there are 5 things
Customer 2500: use 1 thing, now there are 4 things
Producer 2499: produce 1 thing, now there are 5 things
Producer 2499: produce 1 thing, now there are 6 things
```

### 【实验心得体会】

1. 通过本次实验，我更深入了了解了PV操作来实现进程的同步与互斥的基本思想，同时也掌握了写PV操作代码的方法，学会了这些函数的使用方法。
2. 让我感受到，实践才是检验真理的唯一标准，通过实验。虽然我自我感觉平时学的还可以，都是实验中还是会出现很多问题，对代码的而实现一开始我也没有非常好的展现，通过和室友的交流，以及咨询师兄师姐，还是了解了许多东西，很开心，非常的感谢补助我的人。

## 实验六 共享主存实现进程通信

### 【实验目的】

利用共享主存解决读写者问题。要求写者进程创建一个共享主存，并向其中写入数据，读者进程随后从该共享主存区中访问数据

### 【实验软硬件环境】

ubuntu 16.0.4 on VMware Station

### 【实验内容】

为基于共享主存解决读者-写着问题，需要由写进程首先创建一个共享主存，并将该共享主存区映射到虚拟地址空间，随后读进程打开共享主存，并将该共享主存区映射到自己的虚拟地址空间，从中获取数据，并进行处理，以此实现进程通信。

### 【实验程序及分析】

#### 1. 程序代码

```
/* This is reader.c */

#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct people{
    char name[6];
    int age;
};
int main(int argc,char **argv)
{
    struct people *p;
    int key=ftok(".",1);
    if(key==0){
```

```

        printf("Failed to create key value!\n");
    }
    else{
        int shm_id=shmget(key,4096,0666|IPC_CREAT);
        p=shmat(shm_id,NULL,7);
        printf("%s\n",p->name);
        printf("%d\n",p->age);
        shmdt(p);
    }
    return 0;
}

```

```

/* This is write.c */

#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct people {
    char name[6];
    int age;
};
int main(int argc,char **argv)
{
    struct people *p;
    int key=ftok(".",1);
    if(key==0){
        printf("Failed to create key value!\n");
    }
    else{
        int shm_id=shmget(key,4096,0666|IPC_CREAT);
        p=shmat(shm_id,NULL,7);
    }
}

```

```
        scanf("%s",p->name);
        scanf("%d",&p->age);
        shmdt(p);
    }
    return 0;
}
```

## 2.代码分析

定义一个结构体，其中包括姓名和年龄两种数据。

对于读者进程：首先调用 `ftok` 函数创建一个键值，`key = ftok(“.”,`

1)，这样就是将指定的文件名 `fname` 设为当前目录，`id` 是子序号。

若创建键值失败，即为`key`值为0，则打印“创建键值失败”提示信息。

若创建键值成功，调用 `shmget` 函数创建一块共享主存区，

```
shm_id=shmget(key,4096,0666|IPC_CREAT);
```

第一个参数 `key`为此值来源于`ftok`返回的IPC键值，

第二个参数`size`新建的共享内存大小为4096字节，

第三个参数 `shmflg` 为0666|IPC\_CREAT 取共享内存标识符。

接着连接共享内存标识符为 `shmid` 的共享内存，连接成功后把共享内存区对

象映射到调用进程的地址空间，随后可像本地空间一样访问。

```
p=shmat(shm_id,NULL,7);
```

第一个参数 `shmid` 为共享内存标识符

第二个参数 `shmaddr` 为NULL,指定共享内存出现在进程内存地址的什么位置，直接

指定为NULL让内核自己决定一个合适的地址位置

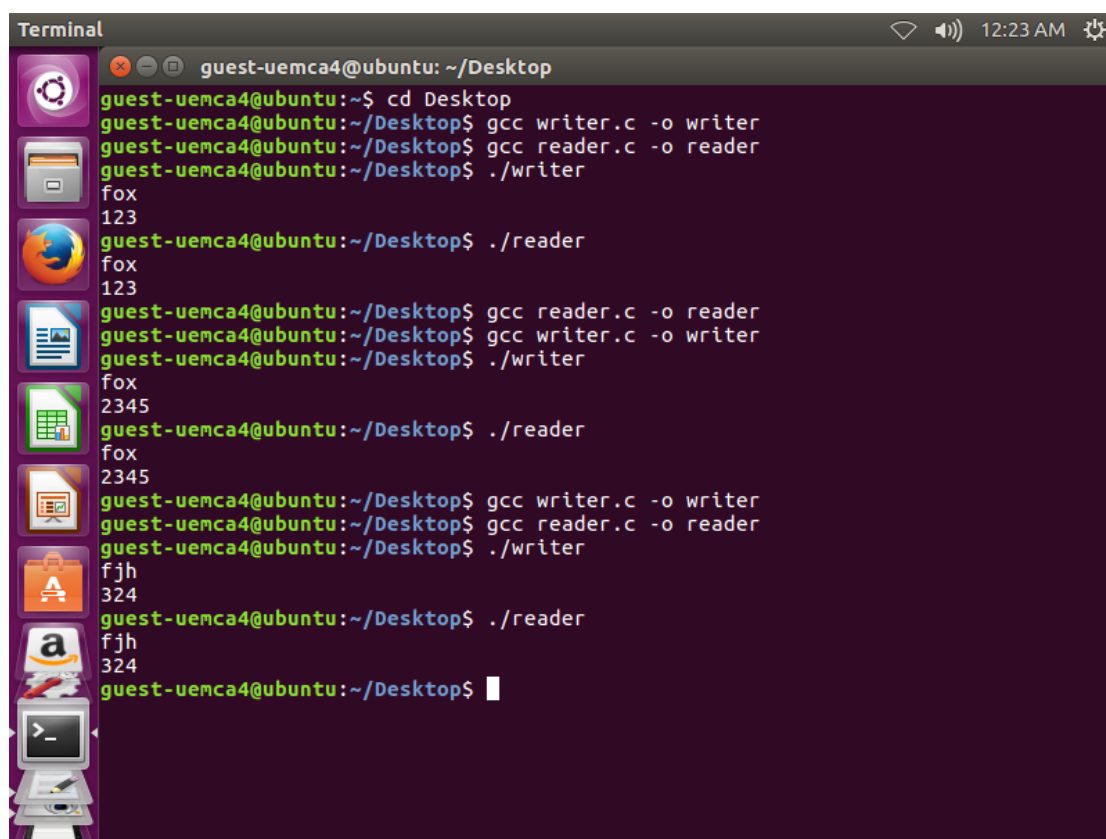
第三个参数 `Shmflg` 规定主存的读写权限，这里为7，表示可读也可写。

然后从主存中读取数据，打印相关信息。

最后利用 `shmdt` 函数将其从自己的贮存段中删除出去

对于对于写者进程：代码实现的方法与读者的进程相似，只有唯一一点的区别，就是要向主存中写入数据，实验中采用了在控制台端输入来实现，用 `scanf`函数即可，实验中先运行读者程序再运行读者程序。

### 【实验结果截图】

A terminal window titled 'Terminal' with a dark background and light text. The window shows a series of commands and their outputs. The user is in the directory ~/Desktop. The commands and outputs are as follows:

```
guest-uemca4@ubuntu: ~/Desktop
guest-uemca4@ubuntu:~$ cd Desktop
guest-uemca4@ubuntu:~/Desktop$ gcc writer.c -o writer
guest-uemca4@ubuntu:~/Desktop$ gcc reader.c -o reader
guest-uemca4@ubuntu:~/Desktop$ ./writer
fox
123
guest-uemca4@ubuntu:~/Desktop$ ./reader
fox
123
guest-uemca4@ubuntu:~/Desktop$ gcc reader.c -o reader
guest-uemca4@ubuntu:~/Desktop$ gcc writer.c -o writer
guest-uemca4@ubuntu:~/Desktop$ ./writer
fox
2345
guest-uemca4@ubuntu:~/Desktop$ ./reader
fox
2345
guest-uemca4@ubuntu:~/Desktop$ gcc writer.c -o writer
guest-uemca4@ubuntu:~/Desktop$ gcc reader.c -o reader
guest-uemca4@ubuntu:~/Desktop$ ./writer
fjh
324
guest-uemca4@ubuntu:~/Desktop$ ./reader
fjh
324
guest-uemca4@ubuntu:~/Desktop$
```

### 【实验心得体会】

1. 通过本次实验，我掌握了 `ftok`, `shmdt`, `shmget` 等函数的运用，基本实现了利用共享主存解决读写者问题。要求有写者进程创建一个共享主存，并向其中写入数据，读者进程从该共享主存中访问数据。同时，也掌握了一定的 Linux 编程知识，感觉自己学到了不少东西。
2. 也是最后一次实验，感觉自己在这个过程中学到了不少东西，这个课程确实

有开设的必要，纸上得来终觉浅，唯有通过实践，才能达到课程知识的真正意义上的运用。同时，通过一次次的编程，感觉自己对操作系统编程的思想有了进一步的理解，也为后续课程设计打下了坚实的基础。

3. 感谢帮助我的室友及大佬们，以及师兄师姐们的悉心指导，还有方敏老师课程的教诲，感到成长的路上有这么多人在我并肩同行就很开心。