操作系统上机

实 验 报 告

成绩

教师： 李晓

2018 年 6 月 22 日

班 级： 1603019班

学 号： 16030199025

姓 名： 张俊华

实验地点： E-Ⅲ-203

实验时间：2018年5月——2018年6月

**实验一、进程的建立**

1. **【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。

1. **【实验软硬件环境】**

软件环境：OS: Ubuntu 18.04 bionic [Ubuntu on Windows 10]；Kernel: x86\_64 Linux 4.4.0-17134-Microsoft；gcc version 7.3.0 (Ubuntu 7.3.0-16ubuntu3)

硬件环境：处理器：Intel（R）Core（TM）2 Duo CPU E7500@2.93GHz 2.94 GHz已安装的内存（RAM）：4.00GB

1. **【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。

进程框架：

#include <unistd.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

int main()

{

/\*创建子进程\*/

if(创建失败)

/\*打印“创建进程失败”提示信息\*/

else if(子进程){

/\*打印子进程相关信息\*/

/\*退出子进程\*/

}

else { /\*父进程\*/

/\*等待子进程信息\*/

/\*继续父进程的执行\*/

}

return 0;

}

1. **【实验程序及分析】**

**实验程序：**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main(){

int id = 0;

char string[100];

id = fork();

if(id == -1) puts("fork FAILED");

else if (id)

{

waitpid(-1,0,0);

FILE \* fp = fopen("file","r");

fscanf(fp,"%s",string);

puts(string);

fclose(fp);

}

else {

puts("this is child");

FILE \* fp = fopen("file","w");

fprintf(fp,"hello,hello\n");

puts("File Writted successfully!");

fclose(fp);

}

}

**实验分析：**

1. **Fork 是计算机程序设计中的分叉函数。返回值： 若成功调用一次则返回两个值，子进程返回0，父进程返回子进程标记；否则，出错返回-1。**
2. **Fork 函数原型：**

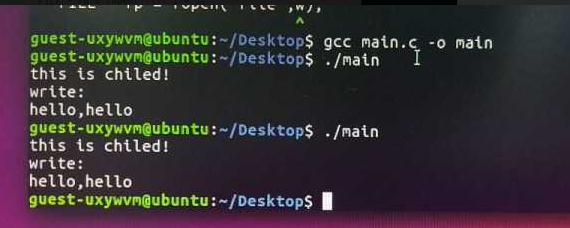
*pid\_t fork*(**void**);

1. **在本实验中，使用了fork函数创建了子进程，让子进程进行文件的写入操作，父进程在等待子进程结束后，进行文件的读取操作，并把读取结果显示在屏幕上**
2. **为了实现父进程等待子进程结束之后再读取文件内容，本实验使用了waitpid 函数，**

pid\_t waitpid(pid\_t pid,int \* status,int options);

子进程结束之后，父进程能正确读出子进程写入的文件内容，证明子进程创建成功

1. **【实验截图】**



1. **【实验心得体会】**

**通过本次实验，我学会了在 Linux 环境下，使用fork() 函数创建子进程的方法，成功创建了子进程并完成了文件的读写操作，与主进程进行了交互。除此之外，我还学习并熟练了在 Linux 环境下进行源代码编写，目录挂载，编译，运行的操作，对Linux 操作系统，和 fork() 函数创建进程的方法和效果有了更深的了解，使我对进程之间的关系有了更加明确的认识。**

**实验二、线程共享进程数据**

1. **【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。

1. **【实验软硬件环境】**

软件环境：OS: Ubuntu 18.04 bionic [Ubuntu on Windows 10]；Kernel: x86\_64 Linux 4.4.0-17134-Microsoft；gcc version 7.3.0 (Ubuntu 7.3.0-16ubuntu3)

硬件环境：处理器：Intel（R）Core（TM）2 Duo CPU E7500@2.93GHz 2.94 GHz已安装的内存（RAM）：4.00GB

1. **【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。程序框架

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <stdlib.h>

Static int shdata=4;

Void \* create(void \*arg)

{

printf(“new pthread ….\n”);

printf(“shared data = %d \n”,shdata);

return (void \*)0;

}

Int main(int argc, char \*argv[])

{ /\*用pthread\_create()创建新线程\*/

if(创建失败){

/\*打印“创建线程失败”提示信息\*/

/\*返回-1\*/

}

/\*休眠一段时间\*/

/\*打印“创建线程成功”提示信息\*/

/\*返回0\*/

}

**【实验程序及分析】**

**实验程序：**

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

static int a=4;

void \*create(void \*arg)

{

printf("new thread created successfully!!\n");

printf("thread: a==%d \n",a);

printf("thread: Change a to 6!\n");

a=6;

return (void \*)0;

}

int main(int argc,char \*argv[])

{

pthread\_t tidp;

int error;

a=5;

error=pthread\_create(&tidp,NULL,create,NULL);

if(error!=0)

{

printf("new thread is not create ... ");

return -1;

}

sleep(1);

printf("the new thread exit \n");

printf("main: now a is :%d\n\n",a);

return 0;

}

**实验分析：**

1. 本次实验进行多线程编程设计，通过pthread\_create()函数创建线程，thread\_create是类Unix操作系统（Unix、Linux、Mac OS X等）的创建线程的函数。它的功能是创建线程（实际上就是确定调用该线程函数的入口点），在线程创建以后，就开始运行相关的线程函数。

函数模型如下：

int pthread\_create(pthread\_t \*restrict tidp, const pthread\_attr\_t \*restrict attr, void \*(\*start\_rtn)(void), void \*restrict arg)

创建线程，若成功则返回0，否则返回出错编号

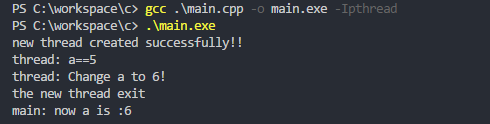
第一个参数为指向线程标识符的指针。

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。

1. 线程创建完成后，线程通过对全局变量的修改，来明确线程和进程之间数据共享的关系
2. 主进程为了等待线程的结束，使用 sleep() 函数进行等待
3. 通过实验，我们观察到，线程修改了全局变量之后，进程读取该变量时，其值也发生了变化，这证明了：进程和线程之间可以共享「全局变量」等公共资源
4. **【实验截图】**



1. **【实验心得体会】**

**本次实验，我们主要学习并掌握了线程的创建操作，线程创建通过 pthread\_create() 函数进行实现，其中第三个参数为线程所要执行的函数，而函数之间的参数传递则可以通过 void\* arg 参数实现，创建好的线程可与进程并发运行，提高程序的工作效率。通过本次实验，我还进一步明确了线程和进程间的数据共享关系，并实际尝试了对全局变量的重新赋值操作。使我对操作系统课程中学习到的知识有了亲身的体会和更深的印象。**

**实验三、信号通信**

1. **【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】**

软件环境：OS: Ubuntu 18.04 bionic [Ubuntu on Windows 10]；Kernel: x86\_64 Linux 4.4.0-17134-Microsoft；gcc version 7.3.0 (Ubuntu 7.3.0-16ubuntu3)

硬件环境：处理器：Intel（R）Core（TM）2 Duo CPU E7500@2.93GHz 2.94 GHz已安装的内存（RAM）：4.00GB

1. **【实验内容】**

父进程创建一个匿名事件，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。

程序模板：

#include <stdio.h>

#include<stdlib.h>

#include <unistd.h>

#include <signal.h>

#include <wait.h>

void process()

{

// do something;

// getpid();

} /\*返回0\*/

int main(void)

{

//创建信号（signal(SIGUSR2, process)）

//创建子进程

if (pid==0)

{

//子进程向父进程发送信号

kill( , SIGUSR2);

}

else if (pid>0)

{

//父进程等待子进程发送过来的消息并执行相应的操作

}

else

{

错误信息；

}

1. **【实验程序及分析】**

**实验程序：**

#include <signal.h>

#include <wait.h>

void process(){

printf("this process pid = %d\n",getpid());

printf("get message success\n");

}

int main(){

signal(SIGUSR2,process);

int pid = fork();

if (pid == 0){

printf("ppid = %d\n",getppid());

printf("Child process send message successfully\n");

kill(getppid(),SIGUSR2);

}

else if (pid>0)

{

sleep(1);

printf("emd\n");

}

else

{

printf("error");

}

}

1. **实验分析：**
2. **本次实验是利用信号通信机制在父子进程之间进行通信，首先，使用 signal() 函数来创建信号量，signal()函数原型：**

**void (\*signal(int signum, void(\* handler)(int)))(int)**

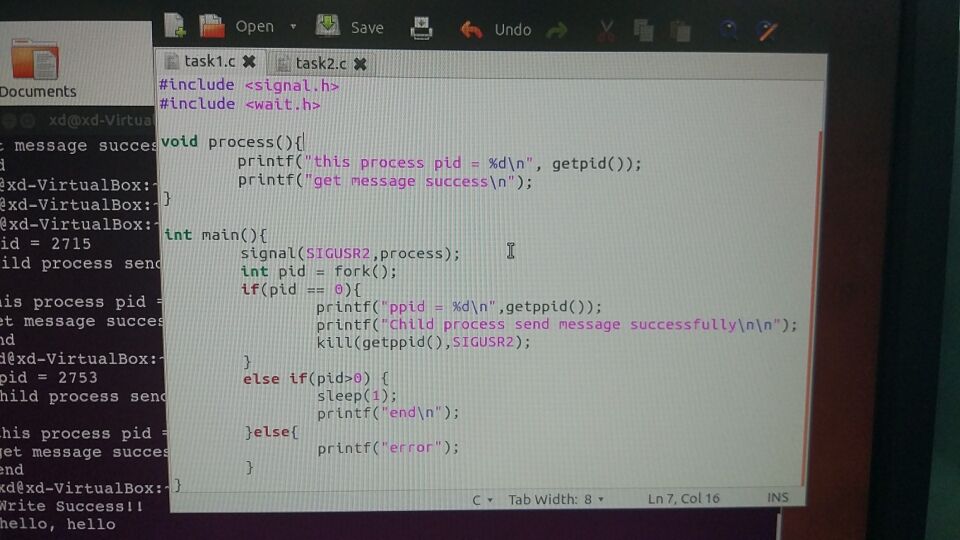
**第一个参数为要处理的信号，第二个参数为所要执行的动作，当收到信号之后，就执行 handler 所指定的函数**

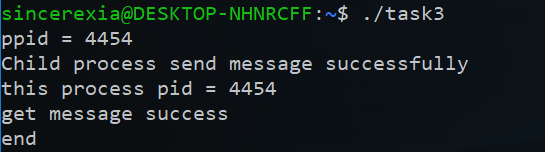
1. **为了发送信号，使用了fork() 函数创建子进程，创建了子进程后，在子进程中，使用 kill() 函数向父进程发送信号，kill() 函数原型：**

**int kill (pid\_t pid, int sig)**

**这里，我们使用kill(getppid(),SIGUSR2);这条语句，向父进程发送 SIGUSR2 这个信号**

1. **父进程收到信号之后，去执行 process 函数**
2. **【实验截图】**





1. **【实验心得体会】**

**本实验实现了进程之间进行通信的一种办法——信号通信，我掌握了使用 signal 函数，进行信号创建的方法，并利用 kill 函数，向指定的进程，发送了指定信号，并成功得到了信号收到之后主进程的执行结果。通过这次实验，我对进程间信号通信的实现有了切身的了解，掌握了 Linux 环境下，信号的创建和发送方法。**

**实验四、匿名管道通信**

1. **【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】**

软件环境：OS: Ubuntu 18.04 bionic [Ubuntu on Windows 10]；Kernel: x86\_64 Linux 4.4.0-17134-Microsoft；gcc version 7.3.0 (Ubuntu 7.3.0-16ubuntu3)

硬件环境：处理器：Intel（R）Core（TM）2 Duo CPU E7500@2.93GHz 2.94 GHz已安装的内存（RAM）：4.00GB

1. **【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。程序模板：

#include <wait>

#include <unistd.h>

#include <string.h>

#define MAX\_LINE 80

Void main()

{ /\*创建一个匿名管道\*/

if(管道创建成功){

/\*创建子进程成功\*/

if(进程创建成功){

/\*关闭写端\*/

/\*休眠一段时间\*/

/\*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\*/

/\*关闭读端\*/

/\*退出\*/

}else if(进程创建成功){

/\*关闭读端\*/

/\*向管道写端写入数据\*/

/\*打印“父进程写管道成功”\*/

/\*关闭写管道\*/

/\*打印“父进程关闭写管道成功”提示信息\*/

/\*休眠一段时间\*/

}

}

return 0；

}

1. **【实验程序及分析】**

**实验程序：**

#include <unistd.h>

#include <stdio.h>

int main(){

int fd[2];

int status = pipe(fd);

if (status == 0){

int pid = fork();

if (pid){

close(fd[1]);

sleep(1);

char s2[100];

read(fd[0],s2,100);

printf(s2);

printf("\nREAD Success!!\n");

close(fd[0]);

exit(0);

}

else

{

close(fd[0]);

char s[] = "hello,hello\n";

printf("Write Success!!\n");

write(fd[1],s,100);

close(fd[1]);

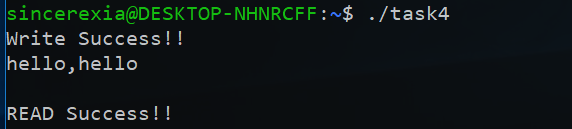
exit(0);

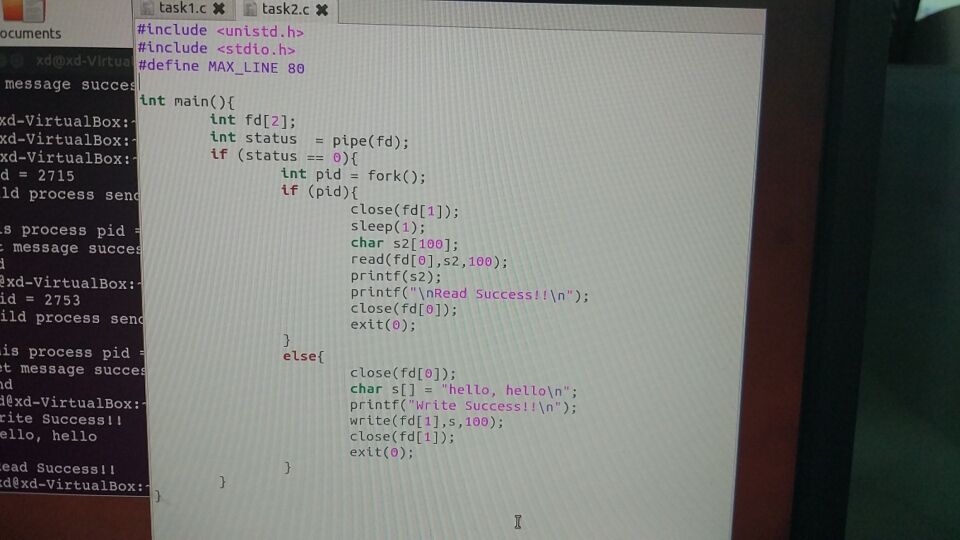
}

}

}

1. **实验分析：**
2. **本次实验使用进程间的另一种通信方式——管道通信，来在进程间传递数据**
3. **父进程创建管道，通过 pip() 函数，得到两个文件描述符指向管道的两端**
4. **父进程 fork 出子进程，子进程也有两个文件描述符指向统一管道**
5. **父进程关闭fd[0],子进程关闭fd[1]，即⽗进程关闭管道读端,⼦进程关闭管道写端（因为管道只支持单向通信）。⽗进程可以往管道⾥写,⼦进程可以从管道⾥读,管道是⽤环形队列实现的,数据从写端流⼊从读端流出,这样就实现了进程间通信。**
6. **【实验截图】**





1. **【实验心得体会】**

**本实验实现了进程之间进行通信的另一种办法——管道通信，管道是调用 pipe() 函数来创建的。我了解并掌握了管道的创建方法和通信过程，成功使用管道在父子进程之间进行了通信，**

## 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

**软件环境**：OS: Ubuntu 18.04 bionic [Ubuntu on Windows 10]；Kernel: x86\_64 Linux 4.4.0-17134-Microsoft；gcc version 7.3.0 (Ubuntu 7.3.0-16ubuntu3)

**硬件环境**：处理器：Intel（R）Core（TM）2 Duo CPU E7500@2.93GHz 2.94 GHz 已安装的内存（RAM）：4.00GB

### 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（仓库）可用，那么生产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，如果缓冲区产品，那么消费者进程将被阻塞，直到新的产品被生产出来。模拟一个生产者两个消费者的情况。

程序模板：

#include<unistd.h>   
#include<sys/types.h>   
#include<sys/stat.h>   
#include<fcntl.h>   
#include<stdlib.h>   
#include<stdio.h>   
#include<string.h>   
#include<sys/sem.h>   
union semun{   
 int val;   
 struct semid\_ds \*buf;   
 unsigned short \*arry;   
};   
//基本量定义（三种信号量、P操作、V操作、初始化/删除信号量）   
int main(intargc, char \*argv[])   
{   
//创建信号量   
//程序第一次被调用，初始化信号量   
//set\_semvalue三种信号量的判断   
 pid\_t p1,p2;   
//pv操作   
}   
staticint set\_semvalue(intsem\_id,intvalue)   
{   
//用于初始化信号量，在使用信号量前必须这样做   
}   
staticvoid del\_semvalue(intsem\_id)   
{   
//删除信号量   
}   
staticint semaphore\_p(intsem\_id)   
{   
//对信号量做减1操作，即等待P（sv）   
}   
staticint semaphore\_v(intsem\_id)   
{   
//一个释放操作，它使信号量变为可用，即发送信号V（sv）   
}

### 实验程序及分析

**实验程序：**

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/sem.h>

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*arry;

};

//基本量定义（三种信号量、P操作、V操作、初始化/删除信号量）

static int full;

static int empty;

static int mutex;

static int set\_semvalue();

static void del\_semvalue();

static int semaphore\_p();

static int semaphore\_v();

int main(int argc, char \*argv[])

{

int num = 0;

//创建信号量,利用semget函数

full = semget((key\_t)666, 1, 0666 | IPC\_CREAT);

empty = semget((key\_t)777, 1, 0666 | IPC\_CREAT);

mutex = semget((key\_t)888, 1, 0666 | IPC\_CREAT);

set\_semvalue(full, 0);

set\_semvalue(empty, 5);

set\_semvalue(mutex, 1);

//初始化信号量，利用semctl函数

pid\_t p1, p2;

if (p1 = fork())

{

while (1)

{

semaphore\_p(empty);

semaphore\_p(mutex);

printf("Producer:%d produced a thing, total:%d\n", getpid(), semctl(full, 0, GETVAL, 0) + 1);

semaphore\_v(mutex);

semaphore\_v(full);

sleep(2);

}

}

else

{

if ((p2 = fork()) == 0)

{

while (1)

{

sleep(2);

semaphore\_p(full);

semaphore\_p(mutex);

num -= 1;

printf("Customer:%d used a thing and there is %d things\n", getpid(), semctl(full, 0, GETVAL, 0) + 1);

semaphore\_v(mutex);

semaphore\_v(empty);

sleep(5);

}

}

else

{

while (1)

{

sleep(2);

semaphore\_p(full);

semaphore\_p(mutex);

num -= 1;

printf("Customer:%d used a thing and there is %d things\n", getpid(), semctl(full, 0, GETVAL, 0) + 1);

semaphore\_v(mutex);

semaphore\_v(empty);

sleep(5);

}

}

}

}

static int set\_semvalue(int sem\_id, int n)

{

//用于初始化信号量，在使用信号量前必须这样做

union semun sem\_union;

sem\_union.val = n;

if (semctl(sem\_id, 0, SETVAL, sem\_union) == -1){

printf("error!\n");

return 0;

}

return 1;

}

static int semaphore\_p(int sem\_id)

{

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;

}

static int semaphore\_v(int sem\_id)

{

//这是一个释放操作，它使信号量变为可用，即发送信号V（sv）

struct sembuf sem\_b;

sem\_b.sem\_num = 0;

sem\_b.sem\_op = 1; //V()

sem\_b.sem\_flg = SEM\_UNDO;

if (semop(sem\_id, &sem\_b, 1) == -1)

{

fprintf(stderr, "semaphore\_v failed\n");

return 0;

}

return 1;

}

**实验分析：**

1. 本次实验通过封装 C 语言内置函数实现信号量以及 p、v 操作
2. 信号量的创建使用 semget() 函数进行实现，函数原型为：

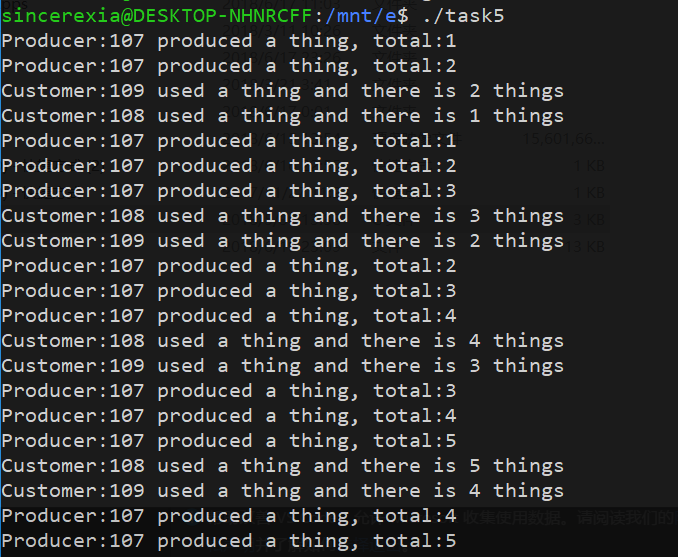
* int semget(key\_t  key, int num\_sems, int sem\_flags);
* 本次实验，我们需要创建 full、empty、mutex 三个信号量，分别表示产品个数，缓冲区空位个数，对缓冲区进行操作的互斥信号量

1. 信号量的初始化操作由 sectl 函数进行，实验中对该函数进行了二次封装，实现对三个信号量初值的设定操作，在本实验中， full、empty、mutex 三个信号量的初值分别是 0、n、1
2. 信号量的 P、V 操作由 semop 函数实现，其原型为：

* int semop(int sem\_id, struct sembuf \*sem\_opa, size\_tum\_sem\_ops);
* 通过修改 sembuf结构体中 sem\_op 的值，实现对应的 P、V 操作

1. 本次实验模拟生产者和消费者之间的同步关系，故需要执行两次 fork 操作，得到三个进程：一个生产者，两个消费者，使用封装好的 semaphore\_p(int sem\_id) 和 semaphore\_v(int sem\_id) 操作来进行同步关系的模拟

### 实验截图



### 实验心得及体会

本次实验较前几次实验来说，感觉难度明显增大，难度提高的原因：一是代码长度大大增长，本次实验所用到的很多功能都需要再次封装，实验的代码逻辑也较上几次复杂；二是实验中引入的函数数量明显增加，并且这些函数的参数结构复杂，很多函数使用了结构体进行传参，并且需要进行权限的设置。通过本次实验，我了解了 Linux 系统中信号量实现进程同步的一种方式，掌握了信号量的初始化以及 P、V 操作的实现方法。并通过调用自己封装的信号量操作函数，实现了生产者消费者同步关系的模拟。通过本次实验，提高了我编写复杂代码解决问题的能力，使我对信号量的实现方式以及操作流程有了更为深入的了解。

## 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

**软件环境**：OS: Ubuntu 18.04 bionic [Ubuntu on Windows 10]；Kernel: x86\_64 Linux 4.4.0-17134-Microsoft；gcc version 7.3.0 (Ubuntu 7.3.0-16ubuntu3)

**硬件环境**：处理器：Intel（R）Core（TM）2 Duo CPU E7500@2.93GHz 2.94 GHz 已安装的内存（RAM）：4.00GB

### 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。

程序模板：

**写者：**

#include <sys/ipc>   
#include <sys/shm.h>   
Typedef struct {   
 char name[4];   
 int age;   
 }people;   
Main(int argc, char \*\*argv)   
{   
 /\*调用ftok函数创建一个键值\*/   
 if(创建键值失败)   
 /\*打印“创建键值失败”提示信息\*/   
 /\*调用shmget创建一块共享主存区\*/   
 /\*将共享主存区附加到自己的主存段\*/   
 /\*向共享主存中写入数据\*/   
 /\*将其从自己的贮存段中删除出去\*/   
}

**读者：**

#include <sys/ipc.h>   
#include <sys/shm.h>   
Typedef struct{   
 char name[4];   
 int age;   
}people;   
Main(int argc, char \*\*argv)   
{   
 /\*调用ftok函数创建一个键值\*/   
 if(创建键值失败)   
 /\*打印“创建键值失败”提示信息\*/   
 /\*调用shmget创建一块共享主存区\*/   
 /\*将共享主存区附加到自己的主存段\*/   
 /\*从共享主存中读取数据\*/   
 /\*将其从自己的贮存段中删除出去\*/   
}

### 实验程序及分析

**实验程序：**

/\*\*

\* 共享内存 ———— 写者

\*

\*/

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

typedef struct {

    char name[20];

    int age;

}people;

int main(){

    key\_t key ;

    key = ftok(",",1);

    int shem\_id;

    shem\_id=shmget(key,getpagesize(),IPC\_CREAT|0666);

    printf("%d",shem\_id);

    people \* p;

    p = (people \*)shmat(shem\_id,NULL,0);

    char str[] = "zhangJunhua";

    strcpy(p->name, str);

    p->age = 19;

    printf("WRITE SUCCESS\n");

    shmdt(p);

}

/\*\*

\* 共享内存 ———— 读者

\*

\*/

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

typedef struct {

    char name[20];

    int age;

}people;

int main(){

    key\_t key ;

    key = ftok(",",1);

    int shem\_id;

    shem\_id=shmget(key,getpagesize(),IPC\_CREAT|0666);

    people \* p;

    p = (people \*)shmat(shem\_id,NULL,0);

    printf("name: %s\n\n", p->name);

    printf("age :%d\n\n",p->age);

    shmdt(p);

}

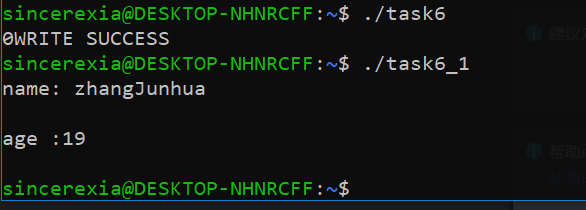
**实验分析：**

1. 本次实验中，两个进程之间通过共享主存的方式进行通信，实现共享主存，就要先创建一个共享内存区，共享内存区通过 shmget 函数进行实现，其函数原型为：

* int shmget(key\_t  key, size\_t  size, int shmflg)
* 该函数能在主存中开辟一段大小为 size 的空间，用于进程间的资源共享

1. 共享内存建立之后，需要将内存区映射到当前进程的地址空间，之后才能进行访问操作，映射共享内存的函数操作为：shmat 其返回值为一个当前进程的地址空间
2. 在本实验中，两个进程之间共享的数据为一结构体数据，其有两个子元素，一是一个字符类型数组，用来保存一个姓名，二为一个 int 类型数据，用于保存年龄。在写者进程中，对共享主存区的该结构体进行赋值操作，在读者进程中，进行访问和读取，并把获取到的数据打印在屏幕上，实现进程之间的数据共享和通信

### 实验截图



### 实验心得及体会

通过这次实验，我掌握了使用共享主存的方式进行进程通信的方法。掌握了共享主存区的分配和挂载操作，对共享主存的实现方式有了进一步的了解，并成功运用共享主存实现了读者和写者进程之间的数据通信，在实验的过程中，出现了共享主存区分配失败和权限不正确的问题，仔细检查发现是传入参数有误的原因。通过这几次的操作系统上机实验，使我对所学的相关知识有了更深的体验和了解，感受到了操作系统功能实现的巧妙，培养了我动手实践和代码编写，调试的能力，增强了我独立实践解决问题的能力，使我受益匪浅。