

为了实现生产者和消费者的同步与互斥，使用了信号量机制

为了实现先生产的商品先被消费的原则，定义用于追踪生产次序的变量。

所有程序所对应的生产者消费者的流程伪代码如下：

```
mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

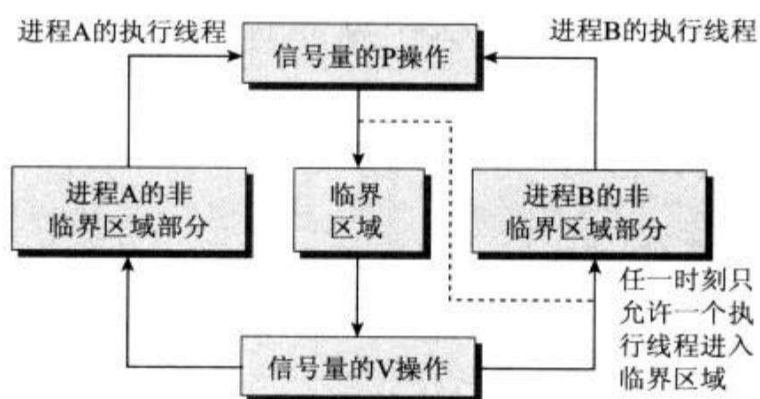
procedure producer() {
    while (true) {
        item = produceItem();
        P(emptyCount);
        P(mutex);
        putItemIntoBuffer(item);    //生产者将字母放入缓冲区
        V(mutex);
        V(fillCount);
    }
}

procedure consumer() {
    while (true) {
        P(fillCount);
        P(mutex);
        item = removeItemFromBuffer();    //消费者从缓冲区取出产品
        V(mutex);
        V(emptyCount);
    }
}
```

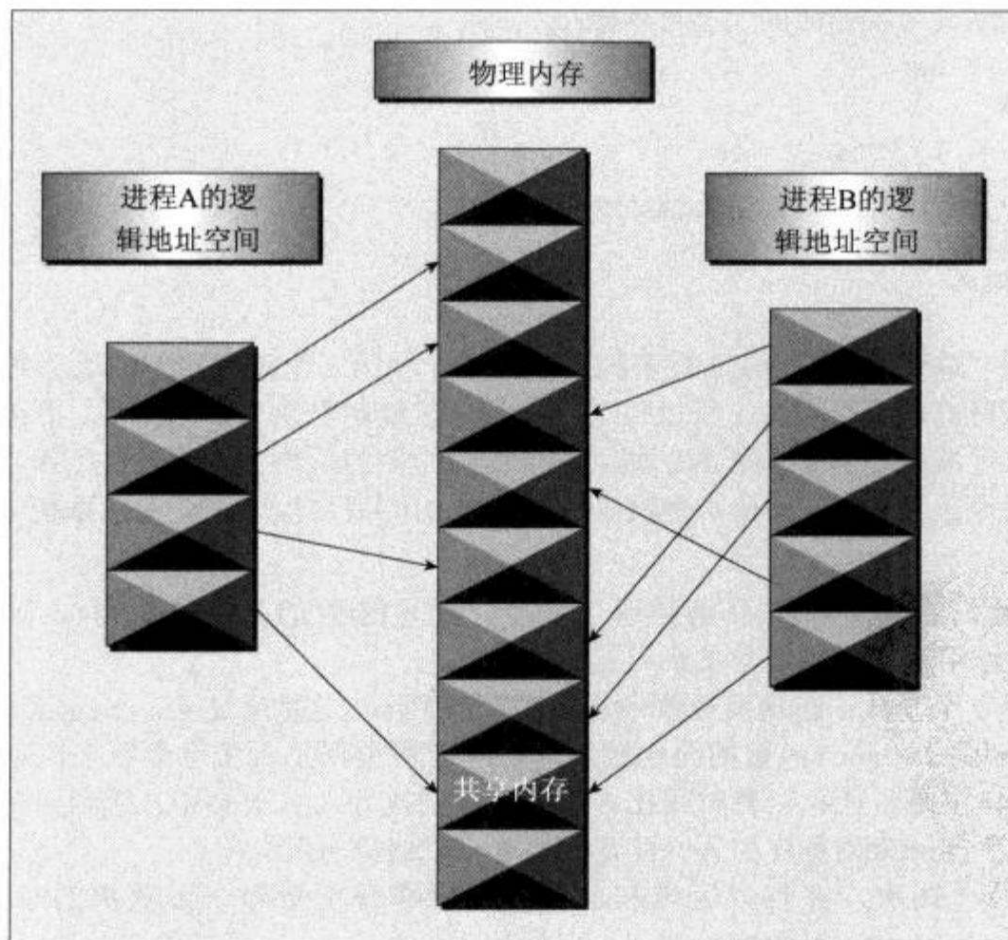
其中 emptycount 记录空闲的缓冲区个数，fillcount 记录已有产品的缓冲区个数。

Linux 系统中的实现

在 Linux 中，信号量的使用流程为：semget() 返回信号量标识符，之后对标识符调用 semctl() 函数，用 SETVAL 命令对信号量进行赋值，而 P、V 操作分别对应着向 semop() 函数中传入的 sembuf 结构所指定的 semop 成员的值，一般指定为 1 为 V 操作，指定为 -1 为 P 操作。在使用完后，调用 semctl()，使用 IPCRMID 命令删除标识符。下图演示了信号量对于临界区，即本次实验中缓冲区的同步作用。



而对于共享内存的使用，对于创建者，先由 `shmget` 函数返回共享区信号量的标识符，之后使用 `shmat` 调用返回一个指向当前进程中地址的指针，表示共享内存的起始区域，之后可以进行类型转换将自己所定义的结构体的指针指向该位置。而其他进程如果向使用该共享区域，则同样需要先使用 `shmget` 调用建立于共享内存的连接，之后调用 `shmat`。当所需要的操作完成后，要使用 `shmdt` 进行于共享内存区的分离。下图为共享内存区的原理。



在 Linux 中本实验只编写了一个代码，并使用 `fork()` 来创建新进程，由于 `fork` 本身复制进程映像的特点，配合使用全局变量可以省去对一些操作的重复定义。

代码解析：

```
static void setSemValue(int sem_id,int val);    //定义需要用到的对信号量的赋值操作与PV操作
static int P(int sem_id);
static int V(int sem_id);
static int delSem(int sem_id);

static int mutex;    //互斥访问信号量以及写满，空余信号量
static int fillcount;
static int emptycount;

union semun    //用在 semctl 赋值时的联合结构
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
```

对于信号量使用的全局定义如上，本次实验的一个特点为，对信号量的赋值操作，PV 操作以及删除操作都进行了封装，在编程时可以直接按照伪码的方式进行编写，提高了可读性。同时使用全局变量保证了数据在不同进程中的可用性。联合结构 `semun` 是一个需要自己定义的用于信号量赋值时所传入的参数。

对于共享内存的使用，预先定义了空指针以及存储共享区域 `id` 的变量：

```
static int shmid;    //共享内存区的id
void *shared_memory = (void *)0;
```

对于此次的缓冲区的构造，定义了如下结构：

```
struct buffet{
    char for_name[BUFFER_SIZE][2];
    int written_by_me;
};

static struct buffet *hereBuffer;    //指向缓冲区的指针
```

操作系统课程设计实验报告

```
static int traceP=0;
static int traceS=0; //用于跟踪上次存放的缓冲区位置，以实现先生产先消费
```

结构 buffet 中字符型二维数组 forname 用来存储每次放入的字母，forname[1][0] 即代表着第二个缓冲区的字符，而 for_name[1][1] 代表着在第二个缓冲区中当前字符被生产的顺序，而这也是本实验为了实现先生产的产品先消费的创新之处。即设置全局变量 traceS 和 traceP，分别对应着每次生产者将产品放入缓冲区值时的赋值以及当前消费者消费的产品个数，消费者每次消费时都要对在缓冲区中找到的产品的 traceS 值进行检查，只有其与当前的 traceP 值相等，消费者才对该产品进行消费。对消费者和生产者的定义如下：

```
void buffer_set(void); //初始化缓冲区
void putItemIntoBuffer(char word); //生产者将字符写入
char produceItem(void); //生产物品
char removeItemFromBuffer(void); //消费者将字符从缓冲区取出
void showProducerStatus(char item); //打印当前缓冲区信息
void showConsumerStatus(char item);
//定义生产者进程
void producer()
{
    for(int i=0;i<4;i++) //重复指定的次数
    {
        srand((unsigned int)getgid()); // 初始化随机数
        sleep(rand()%3); //等待一段三秒以内的时间
        char item;
        item=produceItem();
        P(emptycount);
        P(mutex);
        putItemIntoBuffer(item);
        showProducerStatus(item);
        V(mutex);
        //printf("release mutex.\n");
        V(fillcount);
        //printf("release fillcount.\n");
    }
}
//定义消费者进程
void consumer()
{
    for(int i=0;i<3;i++) //重复指定的次数
    {
        srand((unsigned int)getpid()); // 初始化随机数
        sleep(rand()%4);
        char item;
        P(fillcount);
        P(mutex);
```

```

        item=removeItemFromBuffer();
        showConsumerStatus(item);
        V(mutex);
        //printf("release mutex.\n");
        V(emptycount);
        //printf("release emptycount.\n");
    }
}

```

本实验中对缓冲区的初始化将每个字符都赋值为#，生产者在将生产的产品放入缓冲区时，检查如果当前值为#便认定此区域为空。

在主函数中的操作如下：

//获取信号量标识符

```

mutex=semget((key_t)6666,1,0666|IPC_CREAT);
emptycount=semget((key_t)6789,1,0666|IPC_CREAT);
fillcount=semget((key_t)6999,1,0666|IPC_CREAT);
//先初始化信号量
setSemValue(mutex,1);
setSemValue(fillcount,0);
setSemValue(emptycount,BUFFER_SIZE);

```

```
pid_t processID;
```

```
processID=fork(); //创建子进程
```

```
printf("Now begin the simulation of Producer and Consumer...\n");
```

即先获取信号量标识符并进行初始化，之后创建子进程。其他进程如果想要使用信号量，只需要在 semget 调用中传入相同的 key_t 值进行获取即可。

子进程运行消费者代码的流程如下：

//由消费者进程创建共享内存区

```
shmid=shmget((key_t)1234,sizeof(struct buffet),0666|IPC_CREAT); //
```

获取共享区信号量标识符

```

if(shmid==-1){
    fprintf(stderr,"shmget failed.\n");
    exit(EXIT_FAILURE);
}

```

shared_memory=shmat(shmid,(void*)0,0); //返回一个指向共享内存第一个字节的指针

```

if(shared_memory==(void*)-1)
{
    fprintf(stderr,"shmat failed.\n");
    exit(EXIT_FAILURE);
}

```

```
//printf("memory attached at %X\n",(int)shared_memory);
```

```

    hereBuffer=(struct buffet*)shared_memory;    //将 sharedmemory 分配给
hereBuffer
    buffer_set();
    mutex=semget((key_t)6666,1,0666|IPC_CREAT);
    emptycount=semget((key_t)6789,1,0666|IPC_CREAT);
    fillcount=semget((key_t)6999,1,0666|IPC_CREAT);
    for(int i=0;i<CONSUMER_AMOUNT;i++)
        consumer();    //父进程运行消费者进程
    //分离共享内存并删除
    if(shmdt(shared_memory)==-1)
    {
        fprintf(stderr,"shmdt failed.\n");
        exit(EXIT_FAILURE);
    }
    if(shmctl(shmid,IPC_RMID,0)==-1)
    {
        fprintf(stderr,"shmctl(IPC_RMID ) failed.\n");
        exit(EXIT_FAILURE);
    }
    break;

```

为了简便模拟多个消费者，直接在进程中使用了循环运行消费者进程的方法，这样对结果并无太大影响。

父进程运行生产者进程的代码如下：

/先进行共享内存的连接

```

    shmid=shmget((key_t)1234,sizeof(struct buffet),0666|IPC_CREAT);
    if(shmid==-1)
    {
        fprintf(stderr,"failed to shmget.\n");
        exit(EXIT_FAILURE);
    }
    shared_memory=shmat(shmid,(void*)0,0);
    if(shared_memory==(void*)-1)
    {
        fprintf(stderr,"failed to shmat.\n");
        exit(EXIT_FAILURE);
    }
    hereBuffer=(struct buffet *)shared_memory;

    mutex=semget((key_t)6666,1,0666|IPC_CREAT);    //获取信号量标识符
    emptycount=semget((key_t)6789,1,0666|IPC_CREAT);
    fillcount=semget((key_t)6999,1,0666|IPC_CREAT);

    for(int i=0;i<PRODUCER_AMOUNT;i++)

```

操作系统课程设计实验报告

```
producer();    //子进程运行生产者进程
```

```
if(shmdt(shared_memory)==-1)    //与共享内存区分离
{
    fprintf(stderr,"failed to shmdt.\n");
    exit(EXIT_FAILURE);
}
break;
```

最后父进程调用 wait 函数等待子进程的退出：

```
if(processID>0)
{
    pid_t child_pid;
    int stat_val;    //存储状态信息

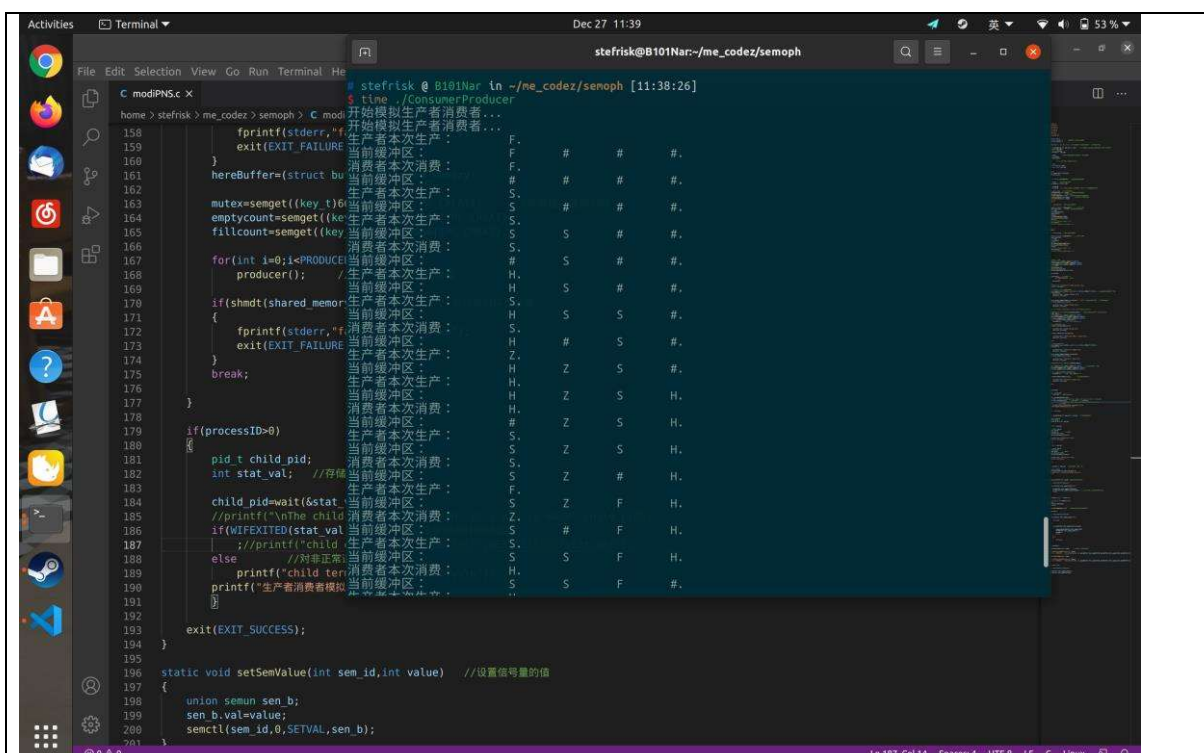
    child_pid=wait(&stat_val);
    printf("\nThe child process has ended, it's pid is %d\n",child_pid);

    if(WIFEXITED(stat_val))    //如果正常退出打印状态码
        printf("child exited with code %d\n",WEXITSTATUS(stat_val));
    else    //对非正常退出
        printf("child terminated abnormally.\n");
    printf("the simulation ends.\n");
}
```

以上为程序的主要流程。 具体的函数实现参见源代码。

程序运行如下：

操作系统课程设计实验报告



Windows 系统中的实现

在 Windows 中的实验实现与在 Linux 中大同小异，区别主要在于对信号量和共享内存区的具体操作不同，以及创建进程的使用方法不同。

信号量的声明为句柄类型，即 HANDLE，对于信号量的操作，同样在程序中将其封装成 PV 函数以及删除函数。P 操作使用 WaitForSingleObject 函数，传入要等待的信号量句柄，V 操作使用 ReleaseSemaphore() 函数，传入信号量的句柄以及需要增加的值。删除信号量则使用 CloseHandle() 关闭相应句柄即可。在 Windows 中信号量的声明使用 CreateSemaphore 函数，要传入最大使用量，当前可使用量以及信号量的名称，其中信号量的名称即用于其他进程对创建的信号量进行连接。

具体创建如下：

```
HANDLE mutex; //声明需要用到的三个信号量的句柄
```

```
HANDLE fillcount;
```

```
HANDLE emptycount;
```

```
mutex = CreateSemaphore(NULL, 1, 1, "mutexx");  
fillcount = CreateSemaphore(NULL, 0, BUFFER_SIZE, "fillcountt");  
emptycount = CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, "emptycountt");
```

而对信号量的连接使用的操作为 OpenSemaphore()，需要传入信号量名称：

```
mutex = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "mutexx");  
emptycount = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "emptycountt");
```


操作系统课程设计实验报告

```
fillcount = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "fillcountt");
```

共享内存的使用同样分为创建和连接,创建时使用 CreateFileMapping 创建共享内存区域的映射,之后使用 MapViewOfFile 返回对应的指针。连接时使用 OpenFileMapping 调用后,使用 MapViewOfFile 返回对应的指针。

创建时代码如下:

//创建共享内存区域的映射

```
shared_memory = CreateFileMapping(
    INVALID_HANDLE_VALUE,
    NULL,
    PAGE_READWRITE,
    0,
    sizeof(struct buffet),
    (LPCTSTR)L"share_memory"
);

if (shared_memory == NULL)
{
    printf("Could not create file mapping object (%d).\n", GetLastError());
    exit(EXIT_FAILURE);
}
myBuf = (LPTSTR)MapViewOfFile(shared_memory,
    FILE_MAP_ALL_ACCESS,
    0,
    0,
    sizeof(struct buffet));

if (myBuf == NULL)
{
    printf("Could not map view of file (%d).\n", GetLastError());

    CloseHandle(shared_memory);
    exit(EXIT_FAILURE);
}
hereBuffer = (struct buffet*)myBuf; //将指向共享内存的指针修改为指向所需结构的指针
buffer_set(); //初始化缓冲区
```

本次实验中有两个程序分别运行消费者和生产者进程,创建进程时使用了实验二中创建进程的思路和代码。

程序运行如下:

```
C:\Users\Stefanny\Desktop>consumer producer
```

```
开始模拟生产者消费者...
```

```
生产者本次生产: Z
当前缓冲区为: Z # # #
本次消费者消费: Z
当前缓冲区为: # # # #
生产者本次生产: F
当前缓冲区为: F # # #
生产者本次生产: F
当前缓冲区为: F F # #
生产者本次生产: F
当前缓冲区为: F F F #
生产者本次生产: F
当前缓冲区为: F F F F
本次消费者消费: F
当前缓冲区为: # F F F
生产者本次生产: F
当前缓冲区为: F F F F
本次消费者消费: F
当前缓冲区为: # F F F
生产者本次生产: H
当前缓冲区为: F H F F
本次消费者消费: F
当前缓冲区为: F H # F
生产者本次生产: S
当前缓冲区为: F H S F
本次消费者消费: F
当前缓冲区为: F H S #
生产者本次生产: F
当前缓冲区为: F H S F
本次消费者消费: F
当前缓冲区为: # H S F
生产者本次生产: F
当前缓冲区为: H S F F
本次消费者消费: H
当前缓冲区为: # S F F
生产者本次生产: Z
当前缓冲区为: Z S F F
本次消费者消费: S
当前缓冲区为: Z # F F
生产者本次生产: S
当前缓冲区为: Z S F F
本次消费者消费: F
当前缓冲区为: Z S #
生产者本次生产: F
当前缓冲区为: # Z S #
本次消费者消费: Z
当前缓冲区为: # # S #
生产者本次生产: S
当前缓冲区为: # # # #
模拟结束。
```

五、实验结果和分析

经过多次运行实验代码后，发现每次都可以顺利模拟完成，不会发生死锁的情况。但是通过循环多次运行进程的方法来实现多个生产者和消费者并不是完全符合标准，可以考虑在进程中定义并行的生产者和消费者线程或者通过 `fork` 和 `createprocess` 函数多次创建进程来实现真正的并发运行。

此外，在编写 Windows 程序完成初期一直有生产者生产完但是消费者毫无反应的情形造成死锁，在多次调试后发现原因存在于在创建和打开信号量时使用的类型转换造成了名字读取错误从而无法打开信号量的情况，即把 `char *` 类型转换为 `LPCWSTR` 型，后来发现这个转换并无必要。所以以后编程时一定要注意类型转换可能引发的错误。

最后对于完成本次实验，在 Linux 系统中编写了 303 行代码，Windows 中两个进程一共 325 行代码之后，更能意识到将一个大问题拆解，分析其中要用到的原理并逐个实现并整合的巧妙性及合理性。愿在之后的编程解决问题的过程中更加注意对问题的分析拆解。