_zach L10 2017-03-11 06:43:14 地址映射与共享

“操作系统原理与实践”实验报告

地址映射与共享

一.追踪Linux 0.11的地址翻译过程

0.正如老师说的,只要照着实验指导书上的步骤来做,这个子实验"基本不消耗脑细胞".

1.用sreg命令得到ldtr,gdtr中的内容:

```
<bochs:6> sreg
cs:s=0x000f, dl=0x00000002, dh=0x10c0fa00, valid=1
ds:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=3
ss:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
es:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
fs:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
gs:s=0x0017, dl=0x00003fff, dh=0x10c0f300, valid=1
ldtr:s=0x0068, dl=0x42d40068, dh=0x000082fd, valid=1
tr:s=0x0020, dl=0xb58c0068, dh=0x00008b01, valid=1
gdtr:base=0x00005cb8, limit=0x7ff
idtr:base=0x000054b8, limit=0x7ff
```

2.用命令“xp /2w gdtr中的值 + 13*8"来得到LDT表的起始地址(虚拟地址)的段表表项(段描述符),再根据段描述符的格式用一种开挂般的方法直接得到LDT表的物理地址0x00fd42d4(按理说这是"段"描述符,所以得到的不应该是虚拟地址吗?指导书上没有明说.我的理解是,LDT表是放在内核态空间中的,在0.11中既是放在最底端的1MB空间内,这1MB的物理内存是不进行分页处理的,于是上述得到的虚拟地址直接就是物理地址)

```
<bochs:8> xp /2w 0x00005cb8+13*8
[bochs]:
0x00005d20 <bogus+       0>:    0x42d40068      0x000082fd
```

3.用命令"xp /8w LDT表的物理地址0x00fd42d4"得到LDT表的前四项.因为ds==0x17,所以段选择子为0x10,所以看LDT表的第三个段描述符,为0x00003fff 0x10c0f300,根据2中所述的"开挂"法得到ds段的起始地址(虚拟地址)0x10000000,用其与i的段内偏移0x3004相加,即可得到i的虚拟地址为0x10003004.

```
<bochs:10> xp /8w 0x00fd42d4
[bochs]:
0x00fd42d4 <bogus+       0>:    0x00000000      0x00000000      0x00000002      0
x10c0fa00
0x00fd42e4 <bogus+      16>:    0x00003fff      0x10c0f300      0x00000000      0
x0001c2a0
```

4.根据3中得到的i的虚拟地址,得到i的物理地址的页目录号为64,页号为3,页内偏移是4.用creg命令查看CR3寄存器中的值,为0,即页目录表的起始(物理)地址为0,用"xp /w 0+64*4"命令查看页目录号为*64*的页目录项,得知页表所在物理页框号为*0x00fa8*,所以页表的物理地址为*0x00fa8000*,用命令*"xp /w 0x00fa8000+34"*得到3号页表项.所以线性地址0x10003004对应的物理页框号为0x00fa5,加上页内偏移,即可得到i的物理地址0x00fa5004.验证:用命令"xp /w 0x00fa5004"查看物理地址0x00fa5004中的内容,发现即为i的值0x12345678.

```
<bochs:14> xp /w 0+64*4
[bochs]:
0x00000100 <bogus+       0>:    0x00fa8027
<bochs:15> xp /w 0x00fa8000+3*4
[bochs]:
0x00fa800c <bogus+       0>:    0x00fa5067
<bochs:16> xp /w 0x00fa5004
[bochs]:
0x00fa5004 <bogus+       0>:    0x12345678
```

二.在ubuntu上用共享内存+信号量解决生产者-消费者问题

1.关于这部分,我又一次倒在了open()相关的权限flag上.在ubuntu上编译时,不要忘了加上-pthread这个编译选项.详见注释.

实验数据

学习时间 351分钟

操作时间 51分钟

按键次数 111次

实验次数 3次

报告字数 17317字

是否完成 完成

评分

未评分

下一篇

篇

相关报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 基于内核栈切换的进程切换 实验报告

操作系统原理与实践: 熟悉实验环境 实验报告

操作系统原理与实践: 信号量的实现和应用 实验报告

2.producer.c

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

#define BUFFER_SIZE 10
#define PRODUCT_RANGE 500

int main(void)
{
    sem_t * Mutex;
    sem_t * FreeBuf;
    sem_t * AvaiPro;
    int shmID;
    int Product;
    int * Buffer;
    key_t Key;

    /**
        用ftok()生成一个IPC Key,关于这个我也不懂,
        反正POSIX中规定,两个进程要想打开同一段共享内存,
        提供的Key必须相等
    **/
    if ((Key=ftok("/home/worstbrick/tmp",0x03))==-1)
    {
        perror("ftok error");
        return (-1);
    }
    /*printf("The IPC key is %d\n",Key);*/

    /**
        (1)关于调用shmget()所要提供的权限flag,若我只用IPC_CREAT,
        到consumer调用shmget()时就会出现EACCES错误,即consumer
        的权限不够.
        (2)对于sem_open()的权限flag,和open()的是一样的,因为
        没怎么学过POSIX中的相关知识,所以之前折磨过我的权限flag的
        选择问题又来了.我试了很多,S_IRWXU,S_IRWXG,S_IRWXO都上了
        一遍,可是全部都会使consumer出现EACCES错误.尤其是那个S_IRWXO
        ,号称"其他用户具有可读、可写及可执行的权限",也不知道这个
        "其他用户"是指什么?是指不同进程组的进程?还是说什么乱七八糟的
        进程都可以?结果根本就都"不可以".
        (3)最后只能选择0777这个"公交车"权限.以后要好好补补这方面的知识了.
    **/
    if ((shmID=shmget(Key,BUFFER_SIZE*sizeof(int),0666|IPC_CREAT))==-1)
    {
        perror("shmget error");
        return (-1);
    }
    if ((int)(Buffer=(int *)shmat(shmID,NULL,SHM_RND))==-1)
    {
        perror("shmat error");
        return (-1);
    }
    if ((Mutex=sem_open("Mutex",O_CREAT,0777,1))==SEM_FAILED)
    {
        perror("sem_open error");
        return (-1);
    }
    if ((FreeBuf=sem_open("FreeBuf",O_CREAT,0777,BUFFER_SIZE))==SEM_FAILED)
    {
        perror("sem_open error");
        return (-1);
    }
    if ((AvaiPro=sem_open("AvaiPro",O_CREAT,0777,0))==SEM_FAILED)
    {
        perror("sem_open error");
        return (-1);
    }

    /**
        Product为生产者生产的产品,产品序列为0~PRODUCT_RANGE
    **/
    for (Product=0;Product<=PRODUCT_RANGE;Product++)
    {
        sem_wait(FreeBuf);
        sem_wait(Mutex);
        Buffer[Product%BUFFER_SIZE]=Product;
        sem_post(Mutex);
        sem_post(AvaiPro);
    }
```

```
    shmdt(Buffer);

    return 0;
}
```

3.consumer.c

```c
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 10
#define CONSUMER_QUANT  2
#define PRODUCT_RANGE 500
#define PUBLIC_INDEX 10

int main(void)
{
    sem_t * Mutex;
    sem_t * FreeBuf;
    sem_t * AvaiPro;
    int shmID;
    int Output;
    int i,Length,DeadLockers;
    int * Buffer;
    pid_t CurrentID;
    key_t Key;
    char * Tmp=(char *)malloc(20*sizeof(char));

    if ((Key=ftok("/home/worstbrick/tmp",0x03))==-1)
    {
        perror("ftok error");
        return -1;
    }
    printf("The IPC key is %d\n",Key);
    if ((shmID=shmget(Key,BUFFER_SIZE*sizeof(int),0666|IPC_CREAT))==-1)
    {
        perror("shmget error");
        return -1;
    }
    if ((int)(Buffer=(int *)shmat(shmID,NULL,SHM_RND))==-1)
    {
        perror("shmat error");
        return -1;
    }
    if ((Mutex=sem_open("Mutex",O_CREAT,0777,1))==SEM_FAILED)
    {
        perror("sem_open error");
        return -1;
    }
    if ((FreeBuf=sem_open("FreeBuf",O_CREAT,0777,BUFFER_SIZE))==SEM_FAILED)
    {
        perror("sem_open error");
        return -1;
    }
    if ((AvaiPro=sem_open("AvaiPro",O_CREAT,0777,0))==SEM_FAILED)
    {
        perror("sem_open error");
        return -1;
    }

    if
((Output=open("/home/worstbrick/Output.log",O_RDWR|O_TRUNC|O_CREAT))==-1)
    {
        perror("open error");
        return -1;
    }

    /**
        因为有复数个消费者,那么在某个时刻,在临界区中消费者该取出Buffer中的哪个元素呢?
        如果每个消费者各自私有一个index来指作为Buffer的下标肯定是不行的.
        所以用一个全局变量来完成这个工作:因为根据POSIX标准,共享内存的大小至少为1页,
        容纳BUFFER_SIZE==10个Product绰绰有余,所以我就用Buffer[PUBLIC_INDEX]来存放
        上面说到的全局的index,在这里初始化为0.
    **/
    Buffer[PUBLIC_INDEX]=0;

    /**
        fork()出CONSUMER_QUANT个子进程,所以一共有CONSUMER_QUANT+1个
        消费者.
    **/
    for (i=0;i<CONSUMER_QUANT;i++)
    {
```

```
        if ((CurrentID=fork())==0)
            break;
    }

    while (1)
    {
        sem_wait(AvaiPro);
        sem_wait(Mutex);
        sprintf(Tmp,"%d:
%d\n",getpid(),Buffer[Buffer[PUBLIC_INDEX]%BUFFER_SIZE]);
        Buffer[PUBLIC_INDEX]++;
        Length=strlen(Tmp);
        write(Output,(void *)Tmp,(Length+1)*sizeof(char));
        sem_post(Mutex);
        sem_post(FreeBuf);
        if (Buffer[PUBLIC_INDEX]>=PRODUCT_RANGE)
            break;
    }

    /**
        (1)由作为父进程的消费者来做收尾工作.这里
        用了wait()来回收子进程消费者,只要wait()
        没有返回-1,说明还有子进程还没有被回收,
        而此时,父进程已经结束了上面的while循环了
        即已经完成了所有的消费活动了,然而却还有
        子进程阻塞在信号量AvaiPro上,所以就让父进程
        把他们唤醒.
        (2)这里有个问题:万一是子进程先从上面的while循环中出来,
        而父进程却"死"在里面怎么办?所以我觉得应该把下面
        用//注释掉的部分加进来
    **/

    //while (getsemval(AvaiPro)<0)
            //sem_post(AvaiPro);
    if (CurrentID!=0)
    {
        while (wait(NULL)!=-1)
                sem_post(AvaiPro);
        sem_unlink("Mutex");
        sem_unlink("FreeBuf");
        sem_unlink("AvaiPro");
        shmdt(Buffer);
        close(Output);
        return 0;
    }

    return 0;
}
```

三.在Linux 0.11上实现共享内存

0.在这个子实验中,遇到的最大的问题就是无法处理"写时拷贝"机制.因为这是个山寨玩具版的shmget()和shmat(),用起来很不灵活,对于一些特殊的情况(其实也算不上特殊)就无法处理了:如果我把在ubuntu上实现的consumer.c完全移植到0.11上的话,就会输出错误的结果.罪魁祸首就是(1)我写的shmget()和shmat()无法提供相应的辨识共享内存的机制;(2)写时拷贝机制.原因如下:

- 在ubuntu上实现的consumer.c,是在父进程fork出子进程之前调用shmat(),也就是说,子进程会继承由父进程申请或打开的共享内存.在ubuntu上没有问题.
- 但我写的shmat()只是简单的把一页物理内存映射到某进程虚拟地址空间上,设一块共享内存的物理地址为pAddr,也就是说,只是让多个进程的某个虚拟地址区域都与pAddr建立映射——一很熟悉吧,0.11为了方便,父进程在fork的时候,对于子进程data段的处理也是这样的———就是简单的让父进程和子进程先共享父进程的data段(把该段设为只读),等发生写操作时,再真正的申请一块新的空闲内存,把父进程的data段复制过去,也就是大名鼎鼎的"写时拷贝".
- 那么问题来了,如果先由父进程调用shmat(),再fork出子进程继承,os才不管是不是共享内存呢(因为我也没有提供相应的辨识共享内存的机制),直接一股脑的把父进程的data段全部设为只读,来和子进程共享,这就杯具了———本来是可读/可写的共享内存,变成一段只读的共享区域,一进行写操作,就会触发"写时拷贝",这样,父进程和子进程这些消费者们,读写的都不会是同一块内存区域,输出结果自然就错了.

1.fork()是通过调用memory.c中的copy_page_tables()来实现写时拷贝的,该函数把父进程的页表条目复制到子进程的页表中,来实现共享,但会进行特殊处理:把这些页表条目的R/W位全部设为只读.所以我觉得当务之急就是提供一套可以辨识出共享内存的机制,使copy_page_tables()不会对共享内存的页表进行处理(即设为只读).尝试了如下的方法,但都行不通:

- 可以从页表项中取出物理页框的基址,能否根据这个基址来判断该物理页是否是共享内存?已知基址存放在共享内存的哈希表中,但想根据这个基址来索引哈希表好想不太可行.关于这个方法,没去尝试.

- 根据0.11页表项的格式,我发现第3位好像是没有用的,就想着能不能把这一位作为"共享位",为1时说明该条目是共享内存,所以copy_page_tables()不去处理R/W位.这样,对于那些"非共享内存"的页表条目,其"共享位"都要设为0.看起来不错,但尝试了发现这么做会导致死机.个人认为可能的原因为(1)页表条目的第3位是有别的用处的,只是我不知道而已;(2)必须要保证"共享内存"的页表条目的"共享位"为1,"非共享内存"为0,如果这个"共享位"本来是没有用的,那么很有可能,在(对我来说)如此庞杂的内核代码的某处就把这个共享位修改了.
- 在PCB中添加一个域 int shmid,专门存放该进程当前打开的共享内存的shmid,初始化为一个负数.因为可以用shmid直接索引共享内存的哈希表,而哈希表的元素中存有共享内存的物理地址,可以让这个物理地址与页表条目中的物理地址进行比对来判断这个页表条目指向的页框是否是共享内存————看似可行,效率不会低,但也不太高,因为要把页表条目转成物理地址(当然这是很快的过程).而且这务必要修改跟PCB的初始化有关的那些函数(比如copy_process),想想就觉得麻烦.当然最致命的是,万一该进程申请或打开了多块共享内存怎么办?把PCB中的这个域设为一个数组吗?那查找效率必定大打折扣.所以,这个方法可以用于本实验(因为只有一块共享内存),但并不具有通用性.

2.终于我受不了了,就去网上找答案,发现哈工大的标准答案根本就不care我刚刚考虑的那些问题一一一答案中的consumer.c没有调用fork(),即只有一个消费者,自然不用关心写时拷贝引发的问题(详情请戳这里).而他的shmget()和shmat()也跟我的差不多(详情请戳这里).也就是说,只是我自己想多了.真是呵呵哒.

3.所以怎么办?先把ubuntu上的consumer.c改一改再移植过去呗.即改成先fork(),再调用shmat(),详见代码.但又有一个问题:无论生产者还是消费者,当有一个exit时,os会自动调用memory.c中的free_page()来回收该进程占用的内存页,**包括共享内存**,该函数会把该页的对应的mem_map项减1.而当mem_map项为0了还调用free_page()来回收该页的话,即"trying to free free page",就会宕机,如代码所示.结果就是,比如生产者第一个exit,把该共享内存free了,该页的mem_map项减为0.然后消费者A接着exit,又对该共享页调用free_page(),完美宕机,一颗赛艇.

```
void free_page(unsigned long addr)
{
    if (addr < LOW_MEM) return;
    if (addr >= HIGH_MEMORY)
        panic("trying to free nonexistent page");
    addr -= LOW_MEM;
    addr >>= 12;
    if (mem_map[addr]--) return;
    mem_map[addr]=0;
    panic("trying to free free page");
}
```

4.针对3中的问题,我就在shmat()中动了点手脚:每当一个进程调用shmat()时,就会使该共享内存的mem_map项加1.该工作由一个放在memory.c中的函数incre_mem_map()完成.不过这样做会有个后遗症:当调用put_page(unsigned long page,unsigned long address)时,若参数page(即一页内存的物理地址)对应的mem_map项不为1,就会发出警告"mem_map disagrees with 参数page at 参数address".但起码不会宕机了吧.如果嫌警告讨厌,可以把那句printk注释掉.

```
void incre_mem_map(unsigned long PhyAddr)
{
    PhyAddr-=LOW_MEM;
    PhyAddr>>=12;
    mem_map[PhyAddr]++;
}
```

5.shm.h(内含管理共享内存的数据结构的定义)

```
#ifndef SHM_H_INCLUDED
#define SHM_H_INCLUDED

#define SHM_QUANT 23

#ifndef NULL
#define NULL ((void *) 0)
#endif

typedef int key_t;

/**
    (1)struct SHMNode是和某段共享内存有关的
    数据结构,Key这个field由shmget()的第一个参数设置,
    详见shmget().
    (2)ShmPhyAddress即是这段共享内存的起始(物理)地址.
**/
struct SHMNode{
    key_t Key;
    unsigned long ShmPhyAddress;
};
//把指向struct SHMNode的指针typedef为EachShm
typedef struct SHMNode * EachShm;

/**
    一样的套路:用哈希表.
    (1)shmget()的第一个参数key作为查找这个哈希表的键值.
    哈希表的元素即为上面的EachShm.
    (2)struct SHashNode即为这个哈希表的结构,Quant这个field
    用来标示当前哈希表中有几个元素,即当前有几块共享内存.
    Table域是个EachShm的数组一一一即真正的那个哈希表.
    大小限定为SHM_QUANT.所以最多允许同时出现SHM_QUANT
    块共享内存.一一一至于为什么这样做,就是觉得实现起来方便.
**/
struct SHashNode{
    int Quant;
    EachShm Table[SHM_QUANT];
};

#define INIT_SHMHASH \
{0,\
{NULL,NULL,NULL,NULL,NULL,NULL,\
NULL,NULL,NULL,NULL,NULL,NULL,\
NULL,NULL,NULL,NULL,NULL,NULL,\
NULL,NULL,NULL,NULL,NULL}}


#endif // SHM_H_INCLUDED
```

6.shm.c

```
#include <linux/mm.h>
#include <sys/shm.h>
#include <errno.h>
#include <linux/kernel.h>
#include <stddef.h>

/**
    专门处理共享内存的全局哈希表SHashTable,用一个宏INIT_SHMHASH
    初始化.域Quant初始化为0,域Table作为EachShm的数组,所有元素初始化为NULL.
    为了方便,ShmTable来存放SHashTable.Table.
**/
struct SHashNode SHashTable=INIT_SHMHASH;
EachShm * ShmTable=SHashTable.Table;

unsigned long OperateBrk(unsigned long PhyAddress);

//哈希函数就是最弱智的取模
int ShmHash(key_t key)
{
    return key%SHM_QUANT;
}

/**
    最弱智的线性探测法.
    关于shmid,并不知道POSIX是怎么定义的.
    但实验指导书上说这是"某块共享内存在OS内部的ID",
    也就是说,shmid是每块共享内存独占的,不能重复.
    所以自然就想到用其在哈希表内部的地址来作为这个
    shmid.
**/
int FindShmid(key_t key)
{
    int shmid=ShmHash(key);

    while (ShmTable[shmid]!=NULL && ShmTable[shmid]->Key!=key)
        shmid=(shmid+1)%SHM_QUANT;

    return shmid;
}

/**
    sys_shmget()的作用,就是根据参数key找到某块(可能已经申请过的)共享内存(的物理地址).
    如果key对应的共享内存还不存在,就申请一页物理内存,并构造相关的数据结构
**/
int sys_shmget(key_t key,size_t size)
{
    if (size>PAGE_SIZE)
        return -(EINVAL);
    if (SHashTable.Quant>=SHM_QUANT)
        return -(ENOMEM);

    unsigned long PhyAddress;
    int shmid;

    /**
        因为没有要求实现shmdt(),所以无需考虑对哈希表的删除问题
    **/
    shmid=FindShmid(key);
    if (ShmTable[shmid]==NULL)
    {
        if ((PhyAddress=get_free_page())==0)
            return -(ENOMEM);
        ShmTable[shmid]=(EachShm)malloc(sizeof(struct SHMNode));
        ShmTable[shmid]->Key=key;
        ShmTable[shmid]->ShmPhyAddress=PhyAddress;
        (SHashTable.Quant)++;
    }

    return shmid;
}

/**
    sys_shmat()的关键解释见/kernel/sched.c中的OperateBrk()
**/
void * sys_shmat(int shmid)
{
    if (shmid>=SHM_QUANT || shmid<0 || ShmTable[shmid]==NULL)
        return -(EINVAL);

    unsigned long LinearAddress;

/**
    这个OperateBrk()很关键.因为其涉及对PCB的某些域修改,
```

```
    而我又懒得把sched.h文件include进来,所以把OperateBrk()
    的实现放到了sched.c中.详见OperateBrk().
**/
    if ((LinearAddress=OperateBrk(ShmTable[shmid]->ShmPhyAddress))==0)
        return -(ENOMEM);
    return (void *)LinearAddress;
}
```

7.放在sched.c中的函数OperateBrk().详见注释.

```
/**
    OperateBrk()由mm/shm.c中的sys_shmat()调用.

    (1)sys_shmat()的作用是把某块共享内存的(起始)物理地址开始的
    一页内存映射到某进程一页大小的虚拟地址上.所谓"映射"就是
    建立相关的页表表项,交给/mm/memory.c中的put_page()去
    处理就行了.

    (2)怎么在一个进程的虚拟地址空间中划出一页空闲的虚拟地址?
    根据<<注释>>P640的图13-6,PCB的域brk指向该进程当前data段的
    末尾,一般库函数中的malloc()动态分配内存时,划出的空闲线性空间都是
    以这个brk作为起始地址的.以此为启发,NewBrk为current->brk增加
    一页大小后值,若其大于等于(start_stack-16384),说明划出这一页
    空闲线性地址会覆盖掉当前进程的栈段的虚拟空间,就返回0.
    否则,就进行(3)中的处理.

    (3)有一点千万要注意:PCB中的start_code,end_code,end_data,brk,start_stack
    这些域,存放的都是"段内偏移",所以必须先用get_base()获得data段的基址.
    将这个基址和current->brk相加后的结果才能交给put_page()处理.
    同时,对于shmat()的返回值,是"共享内存映射到某进程的虚拟地址空间
    中的某块区域的首地址",这个"首地址"是拿来用的(就像malloc()的返回值一样),
    所以它必须是"段内偏移".

    (4)Linux 0.11中有个全局的"内存映射字节图"mem_map(1个字节代表1页内存),
    每个页面对应的字节用于标记该页面当前被"占用"的次数.每当用shmat()把一页
    共享(物理)内存映射到某进程的虚拟空间上时,都应该令该页对应的mem_map[]项
    加1,否则,我写的运行在Linux 0.11中的consumer.c就会死机.详见实验报告.
**/
unsigned long OperateBrk(unsigned long PhyAddress)
{
    unsigned long NewBrk=current->brk+PAGE_SIZE;
    unsigned long LinearAddress;

    if (NewBrk<current->start_stack-16384)
    {
        LinearAddress=get_base(current->ldt[2]);
        put_page(PhyAddress,LinearAddress+current->brk);
        incre_mem_map(PhyAddress);
        LinearAddress=current->brk;
        current->brk=NewBrk;
        return LinearAddress;
    }
    else
        return 0;
}
```

8.在Linux 0.11下运行的producer.c

```c
#define __LIBRARY__
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

#define BUFFER_SIZE 10
#define PRODUCT_RANGE 500
#define PUBLIC_INDEX 10

int main(void)
{
    struct sem_t * Mutex;
    struct sem_t * FreeBuf;
    struct sem_t * AvaiPro;
    int shmID;
    int Product;
    int * Buffer;

    if ((shmID=shmget((key_t)1,BUFFER_SIZE*sizeof(int)))==-1)
    {
        perror("shmget error");
        return (-1);
    }
    if ((int)(Buffer=(int *)shmat(shmID))==-1)
    {
        perror("shmat error");
        return (-1);
    }
    if ((Mutex=sem_open("Mutex",1))==NULL)
    {
        perror("sem_open error");
        return (-1);
    }
    if ((FreeBuf=sem_open("FreeBuf",BUFFER_SIZE))==NULL)
    {
        perror("sem_open error");
        return (-1);
    }
    if ((AvaiPro=sem_open("AvaiPro",0))==NULL)
    {
        perror("sem_open error");
        return (-1);
    }

    Buffer[PUBLIC_INDEX]=0;

    for (Product=0;Product<=PRODUCT_RANGE;Product++)
    {
        sem_wait(FreeBuf);
        sem_wait(Mutex);
        Buffer[Product%BUFFER_SIZE]=Product;
        sem_post(Mutex);
        sem_post(AvaiPro);
    }

    return 0;
}
```

9.在Linux 0.11下运行的consumer.c

```c
#define __LIBRARY__
#include <unistd.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 10
#define CONSUMER_QUANT  2
#define PRODUCT_RANGE 500
#define PUBLIC_INDEX 10

int main(void)
{
    struct sem_t * Mutex;
    struct sem_t * FreeBuf;
    struct sem_t * AvaiPro;
    int shmID;
    int Output;
    int i,Length;
    int * Buffer;
    pid_t CurrentID;
    char * Tmp=(char *)malloc(20*sizeof(char));

   if ((shmID=shmget((key_t)1,BUFFER_SIZE*sizeof(int)))==-1)
    {
        perror("shmget error");
        return (-1);
    }
    if ((Mutex=sem_open("Mutex",1))==NULL)
    {
        perror("sem_open error");
        return (-1);
    }
    if ((FreeBuf=sem_open("FreeBuf",BUFFER_SIZE))==NULL)
    {
        perror("sem_open error");
        return (-1);
    }
    if ((AvaiPro=sem_open("AvaiPro",0))==NULL)
    {
        perror("sem_open error");
        return (-1);
    }
    if ((Output=open("/usr/root/Output.log",O_RDWR|O_TRUNC|O_CREAT))==-1)
    {
        perror("open error");
        return -1;
    }

    for (i=0;i<CONSUMER_QUANT;i++)
    {
        if ((CurrentID=fork())==0)
            break;
    }

    if ((int)(Buffer=(int *)shmat(shmID))==-1)
    {
        perror("shmat error");
        return (-1);
    }

    while (1)
    {
        sem_wait(AvaiPro);
        sem_wait(Mutex);
        sprintf(Tmp,"%d:
%d\n",getpid(),Buffer[Buffer[PUBLIC_INDEX]%BUFFER_SIZE]);
        Buffer[PUBLIC_INDEX]++;
        Length=strlen(Tmp);
        write(Output,(void *)Tmp,(Length+1)*sizeof(char));
        sem_post(Mutex);
        sem_post(FreeBuf);
        if (Buffer[PUBLIC_INDEX]>=PRODUCT_RANGE)
            break;
    }

    while (getsemval(AvaiPro)<0)
        sem_post(AvaiPro);
```

```
    if (CurrentID!=0)
    {
        while (wait(NULL)!=-1)
                sem_post(AvaiPro);
        sem_unlink("Mutex");
        sem_unlink("FreeBuf");
        sem_unlink("AvaiPro");
        close(Output);
        return 0;
    }

    return 0;
}
```

四.实验报告

1.详见子实验一中的说明.不想再讨论这方面的问题了.

2.得到的i的物理地址可能会不同.从0.11的源码来看,因为有虚拟内存和段页结合的内存管理机制,get_free_page()在物理页框中找出空闲页是很随意的————只要是空闲的页框就直接拿来用.而test.exe重启后,其各个段在上一次执行时使用的物理页框很可能已经被其他进程占用了,所以这一次其data段有可能会被分配到别的物理页框中去,所以得到的i的地址可能会不同.