mqmelon L55 2015-07-21 04:12:06 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

地址映射与共享——实验报告#

实验目的##

深入理解操作系统的段、页式内存管理，深入理解段表、页表、逻辑地址、线性地址、物理地址等概念； 实践段、页式内存管理的地址映射过程； 编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。 实验内容

本次实验的基本内容是：

用Bochs调试工具跟踪Linux 0.11的地址翻译（地址映射）过程，了解IA-32和Linux 0.11的内存管理机制； 在Ubuntu上编写多进程的生产者—消费者程序，用共享内存做缓冲区； 在信号量实验的基础上，为Linux 0.11增加共享内存功能，并将生产者—消费者程序移植到Linux 0.11。 跟踪地址翻译过程

首先以汇编级调试的方式启动bochs，引导Linux 0.11，在0.11下编译和运行test.c。它是一个无限循环的程序，永远不会主动退出。然后在调试器中通过查看各项系统参数，从逻辑地址、LDT表、GDT表、线性地址到页表，计算出变量i的物理地址。最后通过直接修改物理内存的方式让test.c退出运行。test.c的代码如下：

```
#include <stdio.h>

int i = 0x12345678;

int main(void)
{
    printf("The logical/virtual address of i is 0x%08x", &i);
    fflush(stdout);

    while (i)
        ;

    return 0;
}
```

基于共享内存的生产者—消费者程序

本项实验在Ubuntu下完成，与信号量实验中的pc.c的功能要求基本一致，仅有两点不同：

不用文件做缓冲区，而是使用共享内存； 生产者和消费者分别是不同的程序。生产者是producer.c，消费者是consumer.c。两个程序都是单进程的，通过信号量和缓冲区进行通信。Linux下，可以通过shmget()和shmat()两个系统调用使用共享内存。

共享内存的实现###

本部分实验内容是在Linux 0.11上实现上述页面共享，并将上一部分实现的producer.c和consumer.c移植过来，验证页面共享的有效性。

具体要求在mm/shm.c中实现shmget()和shmat()两个系统调用。它们能支持producer.c和consumer.c的运行即可，不需要完整地实现POSIX所规定的功能。

实验数据

学习时间	546分钟
操作时间	204分钟
按键次数	0次
实验次数	10次
报告字数	18681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

```
shmget() int shmget(key_t key, size_t size, int shmflg);
/*shmget()会新建/打开一页内存，并返回该页共享内存的shmid（该块共享内存在操作系统内部的id）。所有使用同一块共享内存的进程都要使用相同的key参数。如果key所对应的共享内存已经建立，则直接返回shmid。如果size超过一页内存的大小，返回-1，并置errno为EINVAL。如果系统无空闲内存，返回-1，并置errno为ENOMEM。shmflg参数可忽略。*/

shmat() void shmat(int shmid, const void shmaddr, int shmflg);
/*shmat()会将shmid指定的共享页面映射到当前进程的虚拟地址空间中，并将其首地址返回。如果shmid非法，返回-1，并置errno为EINVAL。shmaddr和shmflg参数可忽略。*/
```

实验报告##

一、线性地址到物理地址###

1、从线性地址到物理地址

线性地址也叫逻辑地址，是进程中使用的相对地址。一个线性地址由32位共8个字来表示，前10位表示页目录位置，第2个10位表示页面位置，低位的12位表示页面内的偏移，在32系统中每页大小均为4K，这个可以在段描述符中的G位找到对应的表示，当G位为1时表示页大小为4K，若为0则表示页大小用字节表示。 那进程中的段偏移地址则是数据或代码在进程中的位置，如实验中用到的&x的值为0x00003004，即表示该值存储位置为从数据段的开始处算共有3004个字节，那么这3004个字节对应线性地址中的哪一部分呢？起初我一直以为是最后的偏移位置，但经过实验发现不正确，应该是低22位，即包括页位置及页内偏移两部分。如果查看线性地址的设计以及偏移值的二进制代码就明白了。32位系统下，每页大小为4K，即每页共有1024个项目，每个项目为4字节。即每个项目的索引为0-1023，展开为二进制代码，如0x00003004，二进制代码为：0B 0011 0000 0000 0100，可以看到从第13位开始计算则刚好位于第3页，而页内偏移则为4。比对一下1023的二进制码：0B 0011 1111 1111 ，4095的二进制码：0B 1111 1111 1111，即第0页的最后一个偏移位置的后12位均为1，即一个地址在数据段内的偏移表示成22位后刚好从12位处分开，高位处为页码，低12位为偏移值。所以计算一个地址的线性地址完全不需要进行额外的运算，只需要将当前的段内偏移值加上段基址就可以了，所以实验中的&x=0x0000 3004 ，转换成线性地址就是 code_base+offset=0x1000 0000 + 0x0000 3004 = 0x1000 3004。 而一个段的基址则可以从段描述符中找到，而段描述符则保存在LDT表中，而LDT表则可以查看ldtr寄存器找到其在GDT表中的位置。当然这些值是完全可以在进程中获取的。因为task struct数据结构中保存有一个进程的这些值。

2、从线性地址寻找对应的物理地址

得到了一个变量的线性地址，要寻找其对应的物理地址，则要查找页目录表以及页表。在linux0.11中，页目录表只有一个其保存在物理内存0处，即在系统初始化时被初始化的pgr变量地址。而当前运行的进程的LDT表、GDT表以及全局页目录表的位置均保存在相应的寄存器中，其中全局页目录表的位置保存在cr3寄存器中，所在在本次实验中，通过调试的方法来寻找，可以就可以通过查看这些寄存器值得到。通过creg命令得到。这样就可以得到数据段、LDT表的描述符，以及GDT表的基址和全局页目录表的地址。进一步就可以得到相应线性地址的物理地址。

实验步骤

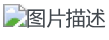
实验用代码

```
#include <stdio.h>
int i=0x12345678;
int main(void)
{
    printf("The logical/Virtual address of i is 0x%08x\n",&i);
    fflush(stdout);
    while(i)
    ;

    return 0;
}
```

1. 通过sreg寄存器找到数据段的选择符为0x0017，LDT的选择符为0x0068，而gdt的基址为0x0000 5cb8。
2. 0x0017的二进制码0B 0000 0000 0001 0111 可知，该段选择符保存在LDT表中，其为用户态权限，保存位置为LDT表中的第2项。 所以要查看LDT表在全局表中的位置索引， 由0x0068的二进制码0B 0000 0000 0110 1000可知，该LDT的描述符保存在GDT表中，其位置为第13项。所以要查看GDT表中第13项内容，而DGT表保存的物理地址为0x0000 5cb8，通过命令：xp /12w 0x0000 5cb8 + 13 * 8(每个描述符占8字节)，可以得到LDT的描述符为：0x52d8 0068 0x0000 82fd，即LDT的基址为：0x00fd 52d8。 通过命令： xp /12w 0x00fd 52d8+2 * 8，可以得到当前数据段的描述符为：0x00003fff 0x10c0 f300，其基址为：0x1000 0000。故变量i的线性地址为：&i=0x1000 0000 + 0x0000 3004 = 0x1000 3004。展开其二进制码为：0B 0001 0000 00 00 0000 0011 0000 0000 0100，得到变量i的线性地址为第64个目录项所指的页表中第3项，其页内偏移为4。

3. 通过creg寄存器找到全局页目录物理地址为：0x0000 0000(在linux 0.11 中全局页目录总是保存在内存0处)。故通过命令：xp /12w 0x0000 0000 +64*4(页表中每项占4字节)，可以得知其值为：0x00fa7027，而低3位表示项的属性，因为所有内存页框地址均4K对齐，所以低12位均为0。故得知其页面所在物理地址为：0xfa7000。
4. 通过命令：xp /12w 0x00fa 7000 +2*4，得到0x00fa 6027，故数据段对应页框地址为：0x00fa 6000。通过命令：xp /4w 0x00fa 6000+4，看到其内容正是0x12345678，即变量i的值。到此，找到了变量i对应的物理地址为：0x00fa6004。
5. 通过命令：setpmem 0x00fa 6004 4 0，将其值修改为0。通过命令c让系统继续，看到进程已经正常退出。

实验过程截图



完成实验后，在实验报告中回答如下问题：###

对于地址映射实验部分，列出你认为最重要的那几步（不超过4步），并给出你获得的实验数据。

答：第一步为找到需要的各个选择符，DS:S=0X0017，LDT:s=0x0068，GDT:BASE=0X0000 5CB8。第二步为找到数据段的基址。实验中进程的NR为4，即第5个进程，所以按照LINUX 0.11的设计，其基址应为 4*64M，转换为16进制即为0x1000 0000。实验结果也验证了这一点。第三步根据找到的线性地址，通过查看页目录表就可以找到对应页表的物理地地址。实验中页表的物理地址为0x00fa 70000。 第四步根据找到的页表所对就应的物理地址就可以找到变量所在页的物理地址。实验中变量所在页的物理地址为0x00fa6000。这里有个很有趣的现象，页表的物理地址比具体一页的物理地址高。而页表一定是先分配的内存，这也说明linux 0.11中物理内存的分配是从高到低进行的。这一点通过查看get_free_page函数得到印证。

test.c退出后，如果马上再运行一次，并再进行地址跟踪，你发现有哪些异同？为什么？

答：若马上再运行一次，则可以看到得到的线性地址为0x1400 3004，即进程的数据段基地多了0x0400 0000。就是多了一个64M。这也正是linux 0.11中对进程线性地址分配的方法即：nr*64M。因为马上运行此时进程4并没有被删除，而只是状态个性为3，所以再运行的进程正好是5。所以其分配的线性地址基址为0x1400 0000。

共享内存实现消费者程序###

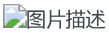
在UBUNTU下运行共享内存的实验代码

```
//producer.c

#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include    <linux/kernel.h>
#include    <fcntl.h>
#include    <linux/types.h>
#include    <sys/stat.h>
#include    <sys/shm.h>
#include     <sys/types.h>

#define BUFFERSIZE    10
struct int_buffer
{
    int int_count;
    int head;
    int tail;
    int data[BUFFERSIZE];
};

sem_t *mutex; // = sem_open("mutex",O_CREAT | O_EXCL,0644,BUFFERSIZE);

int main(void)
{
    char err_desc[255];
    char mutex_sem[]= "mutex";
    int itemValue = -1;

    key_t key;
    struct int_buffer * logbuf;
    int shmid_main;

    key = ftok("./tmp.txt",0x03);
    shmid_main=shmget(key,sizeof(struct int_buffer),IPC_CREAT|0666);
    if(shmid_main == -1)
    {
        printf("shmget(1234,..) error!\n");
        perror(err_desc);
        return -1;
    }

    mutex = sem_open(mutex_sem,O_CREAT,0644,1);
    if(mutex == SEM_FAILED)
    {
        printf("create semaphore mutex error!\n");
        return 1;
    }



    logbuf=(struct int_buffer *)shmat(shmid_main,NULL,0);
    if((long)logbuf==-1)
    {
        printf("in producer shmat(shmid_main,NULL,0) error!\n");
        perror(err_desc);
        exit(-1);
    }

    while(itemValue<499)
    {
        itemValue++;

        //printf("producer(%u) int_count=>%d  itemValue=>
%d\n",getpid(),logbuf->int_count,itemValue);

        //
        while(logbuf->int_count==10)
            ;
        if(sem_wait(mutex)!=0)
        {
            printf("in producer sem_wait(mutex) error!\n");
            perror(err_desc);
            break;
        }
        //
        logbuf->int_count+=1;
        logbuf->data[logbuf->head]=itemValue;
        (logbuf->head)++;
        if(logbuf->head>=BUFFERSIZE)
        {
```

```
            logbuf->head=0;
        }


        if(sem_post(mutex)!=0)
        {
            printf("in producer sem_post(mutex) error!\n");
            perror(err_desc);
            break;
        }


    }

    //
    if(shmdt(logbuf)!=0)
    {
        printf("in producer shmdt(logbuf) error!\n");
        perror(err_desc);
    }

    return 0;
}


//consumer.c

#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include    <linux/kernel.h>
#include    <fcntl.h>
#include    <linux/types.h>
#include    <sys/stat.h>
#include    <sys/shm.h>

#define BUFFERSIZE    10
struct int_buffer
{
    int int_count;
    int head;
    int tail;
    int data[BUFFERSIZE];
};

sem_t *mutex; // = sem_open("mutex",O_CREAT | O_EXCL,0644,BUFFERSIZE);

int main(void)
{
    int get_count=0;

    char err_desc[255];
    char mutex_sem[]= "mutex";
    int itemValue = -1;

    int shmid_main;
    key_t key;
    struct int_buffer * logbuf;
    key = ftok("./tmp.txt",0x03);

    shmid_main=shmget(key,sizeof(struct int_buffer),IPC_CREAT|0666);
    if(shmid_main == -1)
    {
        printf("shmget(1234,..) error!\n");
        perror(err_desc);
        return -1;
    }


    mutex = sem_open(mutex_sem,O_CREAT,0644,1);
    if(mutex == SEM_FAILED)
    {
        printf("create semaphore mutex error!\n");
        return 1;
    }


    //
    logbuf=(struct int_buffer *)shmat(shmid_main,NULL,0);
    if((long)logbuf==-1)
    {
        printf("in producer shmat(shmid_main,NULL,0) error!\n");
```

```
        perror(err_desc);
        exit(-1);
    }

    while(get_count<500)
    {
        //
        while(logbuf->int_count<=0)
            ;


        if(sem_wait(mutex)!=0)
        {
            printf("in customer %u,sem_post(empty) error!",getpid());
            perror(err_desc);
            break;
        }


        itemValue=logbuf->data[logbuf->tail];
        logbuf->int_count--;
        (logbuf->tail)++;
        if(logbuf->tail>=BUFFERSIZE)
        {
            logbuf->tail=0;
        }
        printf("%u:%d\n",getpid(),itemValue);
        get_count++;


        if(sem_post(mutex)!=0)
        {
            printf("in customer %u,sem_post(empty) error!\n",getpid());
            perror(err_desc);
            break;
        }

    }

    //detach the shared memory
    if(shmdt(logbuf)!=0)
    {
        printf("in customer shmdt(logbuf) error!\n");
        perror(err_desc);
    }
    //delete the shared memory
    if(shmctl(shmid_main,IPC_RMID,0)==-1)
    {
        printf("in customer shmctl(shmid, IPC_RMID, 0) error!\n");
        perror(err_desc);
    }

    sem_unlink("mutex");

    return 0;
}
```

运行结果截图



实验的补充说明

由于此次实验的进程仅有两个，所以在实现中不使用信号量进行同步也完全没有问题。去掉进程同步代码结果也完全相同。

在linux 0.11下实现共享内存

此次在LINUX 0.11下实现共享内存并没有太多的困难。当然如果要实现符合POSXI标准的代码就要困难得多。查看现在的LINUX 2.6 内核中相关代码时发现其使用的是文件与内存相关联的方法来实现的。此次仅是使用最简单的办法来实现的。本次实验共实现了4个函数，shmget(),shmat(),shmdt()以及shmctl()其中shmctl函数仅实现了删除共享内存的代码。具体代码见下：

```
//unistd.h 中修改的部分

/*melon*/
#define    NR_SHM    64

struct shmid_ds
{
    int key;
    int size;
    unsigned long page;
    int attached;
};
/*melon 2015-7-16*/


//shm.c

#define __LIBRARY__
#include <unistd.h>
#include <stdarg.h>
#include <errno.h>
#include <linux/sched.h>
#include <asm/segment.h>
#include <string.h>
#include <linux/kernel.h>

#define LOW_MEM 0x100000

static struct shmid_ds shm_list[NR_SHM]={{0,0,0,0}};
extern void add_mem_count(long addr);
extern void remove_page(long addr);

int sys_shmget(int key,int size,int shmflag)
{

    int i;
    int return_key=-1;
    unsigned long page;


    if(size>PAGE_SIZE)
    {
        return -EINVAL;
    }


    for(i=0;i<NR_SHM;i++)
    {
        if(shm_list[i].key==key)
        {
            return i;
        }
    }


    for(i=0;i<NR_SHM;i++)
    {
        if(shm_list[i].key==0)
        {

            return_key=i;
            break;
        }
    }


    page=get_free_page();
    printk("shmget get memory's address is 0x%08x\n",page);
    if(page==0)
    {
        return -ENOMEM;
    }


    shm_list[return_key].key=key;
    shm_list[return_key].size=size;
    shm_list[return_key].page=page;
    shm_list[return_key].attached=0;


    return return_key;
}
```

```
void * sys_shmat(int shmid,const void *shmaddr, int shmflag)
{
    unsigned long data_base;
    unsigned long brk;
    unsigned long page;


    if(shm_list[shmid].key<=0 || shm_list[shmid].page==0)
    {
        return (void*)-EINVAL;
    }


    data_base=get_base(current->ldt[2]);
    printk("current's data_base = 0x%08x,new page =
0x%08x\n",data_base,shm_list[shmid].page);

    brk=current->brk+data_base;
    current->brk +=PAGE_SIZE;

    page=put_page(shm_list[shmid].page,brk);
    if(page==0)
    {

        return (void*)-ENOMEM;
    }

    //add_mem_count(page);

    shm_list[shmid].attached++;
    //printk("current->brk=0x%08x,shmat return 0x%08x,put_page return
0x%08x\n",current-brk,brk,page);

    return (void *)(brk - data_base);

}


int sys_shmdt(int shmid)
{
    unsigned long data_base;

    if(shm_list[shmid].key<=0 && shm_list[shmid].page==0)
    {
        return -EINVAL;
    }
    shm_list[shmid].attached--;

    data_base=get_base(current->ldt[2]); //取数据段基址
    current->brk-=PAGE_SIZE;

    remove_page(data_base+current->brk); //将线性地址从页面中去除

    return 0;
}


int sys_shmctl(int shmid,int shmcmd, struct shmid_ds * buf)
{
    int ret=0;
    switch(shmcmd)
    {
        case 0:
            if(shm_list[shmid].attached>=0 && shm_list[shmid].page!=0)
            {
                free_page(shm_list[shmid].page);
            }
            else
            {
                ret=-EINVAL;
            }
            break;
    }

    return ret;

}
```

在LINUX 0.11下使用的生产者及消费者程序代码

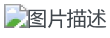
```
//producer11.c

#define __LIBRARY__
#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>
#include    <linux/kernel.h>
#include    <fcntl.h>
#include    <sys/types.h>
#define BUFFERSIZE    10

static _syscall2(sem_t *,sem_open,const char *,name,int,value);
static _syscall1(int,sem_post,sem_t *,sem);
static _syscall1(int,sem_wait,sem_t *,sem);
static _syscall1(int,sem_getvalue,sem_t *,sem);
static _syscall1(int,sem_unlink,const char*,name);

static _syscall3(int,shmget,int,key,int,size,int,shmflag);
static _syscall3(void *,shmat,int,shmid,const void *,shmaddr,int,shmflag);
static _syscall1(int,shmdt,int,shmid);
static _syscall3(int,shmctl,int,shmid,int,shmcmd,struct shmid_ds *,buf);

struct int_buffer
{
    int int_count;
    int head;
    int tail;
    int data[BUFFERSIZE];
};

sem_t *mutex;

int main(void)
{
    char err_desc[255];
    int itemValue = -1;
    char mutex_sem[]= "mutex";
    struct int_buffer * logbuf;
    int shmid_main;
    int mutex_value;

    mutex = sem_open(mutex_sem,1);
    if(mutex == NULL)
    {
        printf("create semaphore mutex error!\n");
        return 1;
    }

    mutex_value=sem_getvalue(mutex);
    if(mutex_value==-1)
    {
        printf("sem_getvalue(mutex) error\n");
        perror(err_desc);
        return 1;
    }
    else
    {
        printf("producer:mutext's value=%d\n",mutex_value);
    }

    shmid_main=shmget(1234,sizeof(struct int_buffer),0);
    if(shmid_main == -1)
    {
        printf("shmget(1234,..) error!\n");
        perror(err_desc);
        return -1;
    }

    logbuf=(struct int_buffer *)shmat(shmid_main,NULL,0);

    if((long)logbuf==-1)
    {
        printf("in producer shmat(shmid_main,NULL,0) error!\n");
        perror(err_desc);
        exit(-1);
    }

    while(itemValue<499)
    {
        itemValue++;
```

```
        while(logbuf->int_count==10)
            ;

        if(sem_wait(mutex)!=0)
        {
            printf("in customer %u,sem_post(empty) error!",getpid());
            perror(err_desc);
            break;
        }

        logbuf->int_count++;

        if(sem_post(mutex)!=0)
        {
            printf("in customer %u,sem_post(empty) error!\n",getpid());
            perror(err_desc);
            break;
        }

        logbuf->data[logbuf->head]=itemValue;
        (logbuf->head)++;
        if(logbuf->head>=BUFFERSIZE)
        {
            logbuf->head=0;
        }

    }


    if(shmdt(shmid_main)!=0)
    {
        printf("in producer shmdt(logbuf) error!\n");
        perror(err_desc);
    }

    return 0;
}


//consumer11.c

#define __LIBRARY__
#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>
#include    <linux/kernel.h>
#include    <fcntl.h>
#include    <sys/types.h>
#define BUFFERSIZE    10

static _syscall2(sem_t *,sem_open,const char *,name,int,value);
static _syscall1(int,sem_post,sem_t *,sem);
static _syscall1(int,sem_wait,sem_t *,sem);
static _syscall1(int,sem_getvalue,sem_t *,sem);
static _syscall1(int,sem_unlink,const char*,name);

static _syscall3(int,shmget,int,key,int,size,int,shmflag);
static _syscall3(void *,shmat,int,shmid,const void *,shmaddr,int,shmflag);
static _syscall1(int,shmdt,int,shmid);
static _syscall3(int,shmctl,int,shmid,int,shmcmd,struct shmid_ds *,buf);

struct int_buffer
{
    int int_count;
    int head;
    int tail;
    int data[BUFFERSIZE];
};

sem_t *mutex;

int main(void)
{
    int get_count=0;

    char err_desc[255];
    char mutex_sem[]= "mutex";
    int itemValue = -1;
    int shmid_main;
    struct int_buffer * logbuf;
    int log = open("pclog.log", O_CREAT|O_TRUNC|O_RDWR, 0666);
    char buflog[255];
```

```
int mutex_value;

shmid_main=shmget(1234,sizeof(struct int_buffer),0);
if(shmid_main == -1)
{
    printf("shmget(1234,..) error!\n");
    perror(err_desc);
    return -1;
}


mutex = sem_open(mutex_sem,1);
if(mutex == NULL)
{
    printf("create semaphore mutex error!\n");
    return 1;
}

mutex_value=sem_getvalue(mutex);
if(mutex_value==-1)
{
    printf("sem_getvalue(mutex) error\n");
    perror(err_desc);
    return 1;
}
else
{
    printf("producer:mutext's value=%d\n",mutex_value);
}


logbuf=(struct int_buffer *)shmat(shmid_main,NULL,0);
if((long)logbuf==-1)
{
    printf("in producer shmat(shmid_main,NULL,0) error!\n");
    perror(err_desc);
    exit(-1);
}

while(get_count<500)
{


    while(logbuf->int_count<=0)
        ;


    itemValue=logbuf->data[logbuf->tail];

    if(sem_wait(mutex)!=0)
    {
        printf("in customer %u,sem_post(empty) error!",getpid());
        perror(err_desc);
        break;
    }

    logbuf->int_count--;

    if(sem_post(mutex)!=0)
    {
        printf("in customer %u,sem_post(empty) error!\n",getpid());
        perror(err_desc);
        break;
    }

    (logbuf->tail)++;
    if(logbuf->tail>=BUFFERSIZE)
    {
        logbuf->tail=0;
    }

    lseek(log,0,SEEK_END);
    sprintf(buflog,"%u:%d\n",getpid(),itemValue);
    write(log,&buflog,sizeof(char)*strlen(buflog));

    get_count++;



}

close(log);
```

```
    if(shmdt(shmid_main)!=0)
    {
        printf("in customer shmdt(logbuf) error!\n");
        perror(err_desc);
    }

    if(shmctl(shmid_main,0,0)==-1)
    {
        printf("in customer shmctl(shmid, IPC_RMID, 0) error!\n");
        perror(err_desc);
    }

    sem_unlink("mutex");

    return 0;
}
```

运行结果截图



关于shmdt及shmctl函数的补充###

由于实验指导中没有要求实现shmdt以入shmctl函数，所以关于这两个函数问题没有打算写入报告中，但考虑到代码中有关于这两个函数的内容，而且本人以为这两个函数的实现在难度上要稍高于另外两个，所以还是补充在这里。

开始时shmdt函数的实现只是简单地从共享内存的结构中将计数器减1，以及从将进程的数据段长度修改回最初，但在运行中却发现，两个进程使用同一段共享内存时其中数据却是随机的。而且有时还会提示释放一段已释放的内存的信息（说明：在最初测试时使用两进程都没有对共享内存进行删除）。可以想象当时是如何意外，也很蒙的。随后仔细想了想，主要是考虑到系统提示说释放一段已释放的内存。那一定是对共享内存进行了删除操作的。而测试进程是一定没有释放的，那也就是说系统自动在释放这段内存。会在哪里呢？后来想到第二个进程运行时第一个进程已经结束，那一定是系统在删除结束进程时对进程所持有的物理内存进行了释放。那就得想办法让系统在结束一个僵尸进程时不释放指定内存，但看了看内存管理的全局结构，似乎不可行。后来就决定在shmat同时将mem_map对应数据增1，因为记得内存的低1M内存在mem_map中其值都是USED(=100)，所以在其中加入了一个增加mem_map[]计数的函数，只是简单在将其增１。以为这样在进程结束被系统回收时其值只会减１，而不会真正被释放掉。但一运行才发现根本不可行，因为在用户态的put_page中要求mem_map只能为1，不能为其他值。也就是说LINUX 0.11不允许用户态内存共享。当然这里也可以简单地将其限制修改掉，但不能保证其他部分会不会有涉及此值的验证或相关代码，所以不能在这里修改。也就是说共享的这一页内存在mem_map中其值只能为1。那就只能在其他部分想办法。最的决定在shmdt时按照put_page的反操作将对应页表中的值去除。本以为有现在函数可用，找了好久也没找到，最后只能在memory.c中手动增加一个remove_page的函数来实现这个目的。具体代码见下面。

```
void remove_page(long addr)
{
    unsigned long * page_table;

    page_table=(unsigned long*)((addr>>20)&0xffc); //取目录地址
    if((*page_table) & 1) //如果对应页面在内存中存在，则取出页地址
        page_table=(unsigned long *)(0xfffff000 & *page_table);

    page_table[(addr>>12)&0x3ff]=0; //去掉页中存在的内存地址

}
```

请输入想说的话

0 / 2000

发表评论

最新评论

暴龙兽 L73

牛X

2017-10-15 15:30:39 回复