哈尔滨工业大学计算机科学与技术学院

2016年秋季学期《操作系统》

Lab5:地址映射与共享

|  |  |  |
| --- | --- | --- |
| 姓名 | 学号 | 联系方式 |
| 匡盟盟 | 1143220116 | [kuangmeng@msn.com](mailto:kuangmeng@msn.com) |
|  |  |  |

目 录

一、实验目的 1

二、实验内容 1

1、跟踪地址翻译过程 1

2、基于共享内存的生产者—消费者程序 1

3、共享内存的实现 1

3.1 shmget() 2

3.2 shmat() 2

三、实验过程 2

准备 2

进行test.c实验 2

进行共享内存实验 6

四、回答问题 9

附录 10

1. shm.c 10
2. sem.c 11
3. 用于linux-0.11使用的producer.c 14
4. 用于linux-0.11使用的consumer.c 15
5. 用于Ubuntu使用的producer.c 16
6. 用于Ubuntu使用的consumer.c 16

一、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。

二、实验内容

实验基本内容：

* 1. 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；
  2. 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；
  3. 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。

1、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;  
 }

2、基于共享内存的生产者—消费者程序

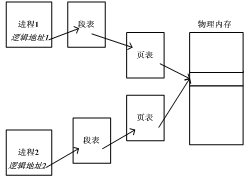
本次实验在Ubuntu下完成，与信号量实验中的pc.c的功能要求基本一致，仅有三点不同：

1. 不用文件做缓冲区，而是使用共享内存；
2. 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；
3. 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。

3、共享内存的实现

进程之间可以通过页共享进行通信，被共享的页叫做共享内存，结构如下图所示：



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。

*3.1 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。

*3.2 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。

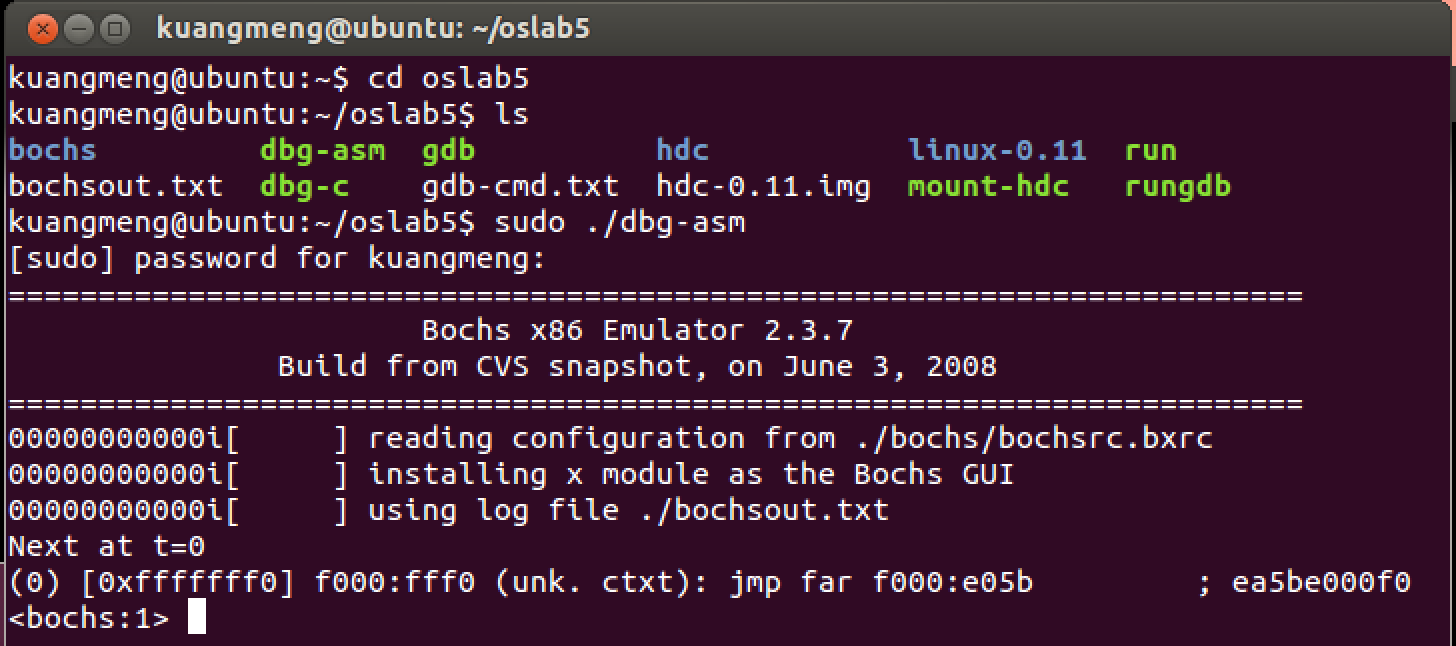
三、实验过程

准备

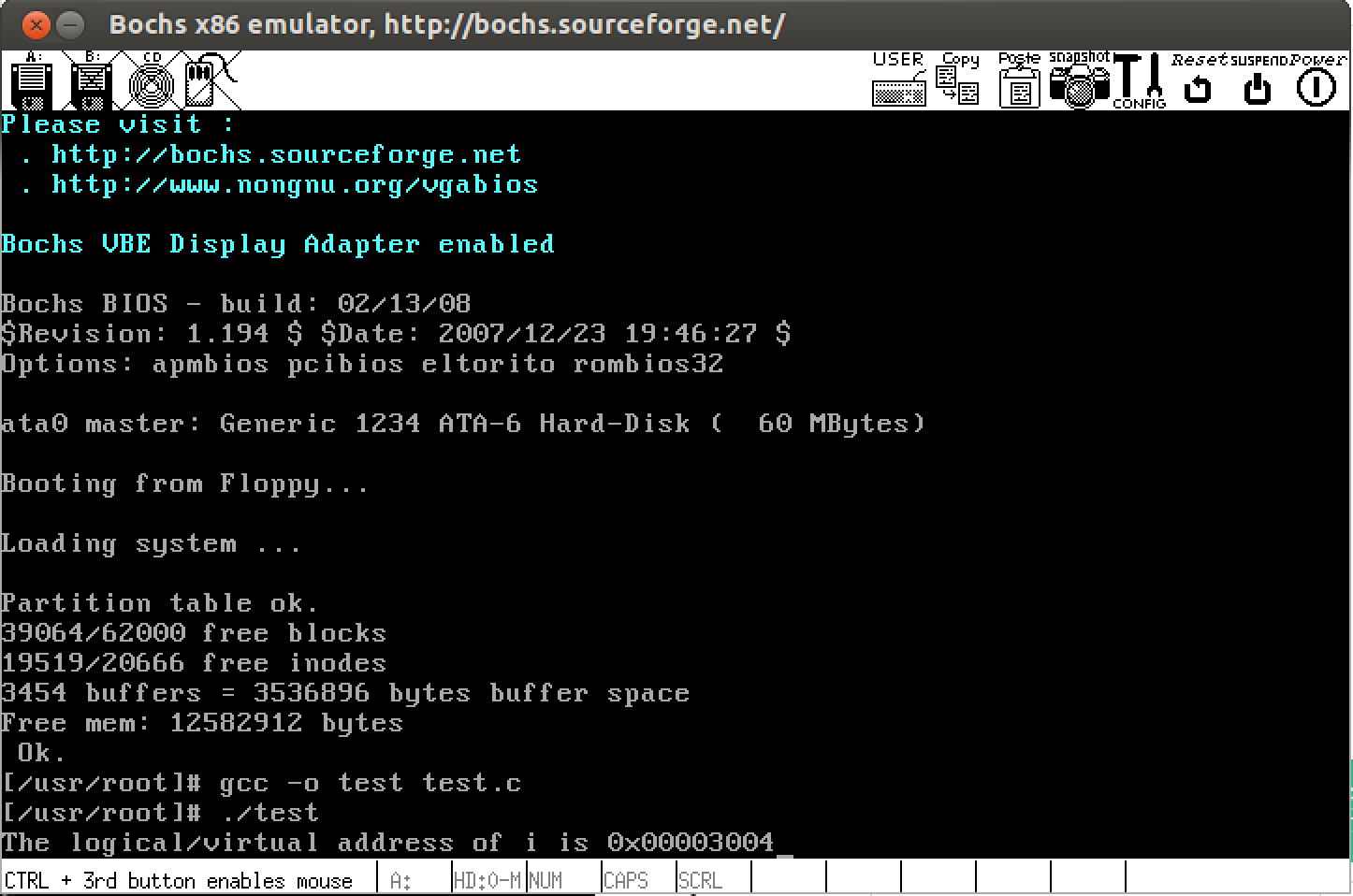
* + 编写test.c文件，放入oslab5/linux-0.11/目录下；
  + 编译linux-0.11（进入linux-0.11目录下，直接“make all”即可）；
  + 修改linux-0.11/mm/memory.c文件：
  + 在Makefile中加入：“shm.s shm.o: shm.c ../include/linux/kernel.h ../include/unistd.h”；
  + 编写linux-0.11/mm/shm.c文件；
  + 修改linux-0.11/kernel/system\_call.s中一行：
  + 修改linux-0.11/include/unistd.h文件，添加两处：
  + 修改linux-0.11/include/linux/sys.h文件：
  + 修改linux-0.11/kernel/sem.c文件；
  + 编写producer.c及consumer.c文件，并放在linux-0.11/目录下。

进行test.c实验

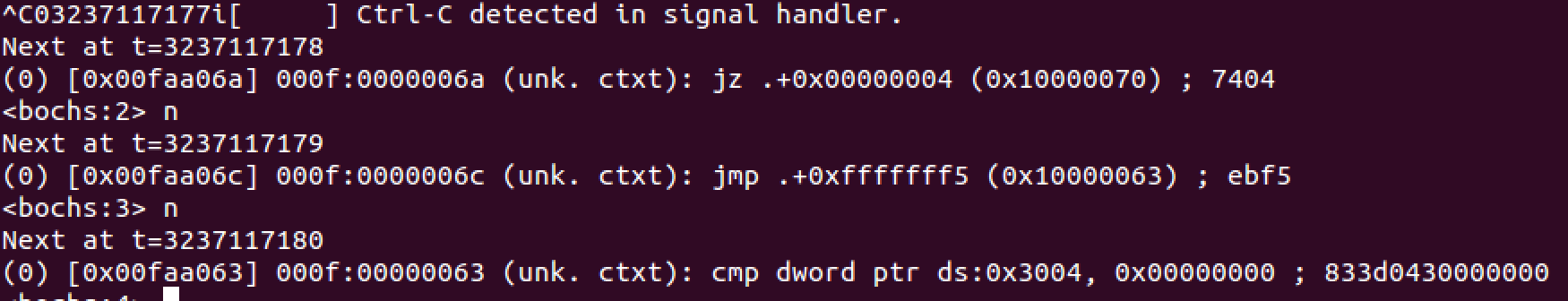
1. 通过“sudo ./dbg-asm”命令进入debug模式，此时控制台显示如下（Bochs显示黑屏）：



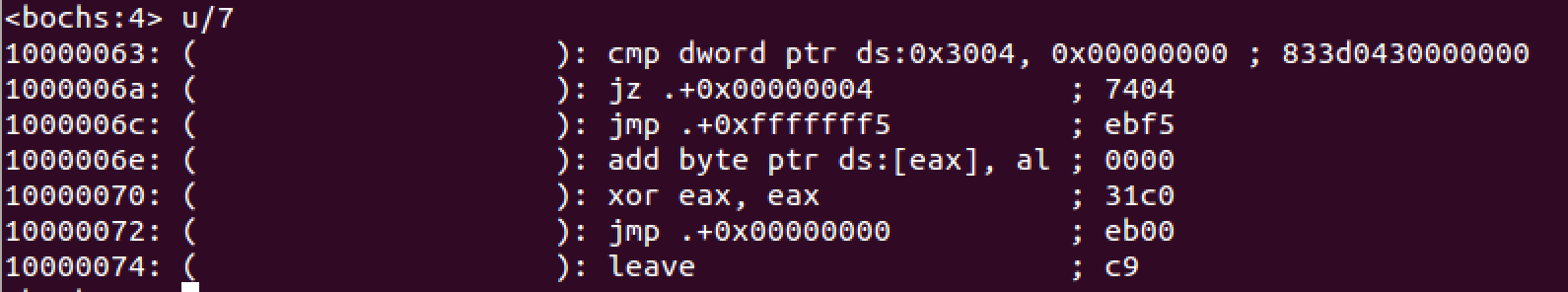
1. 输入指令“c”，继续程序的运行，Bochs此时开始加载系统，运行test,截图如下:



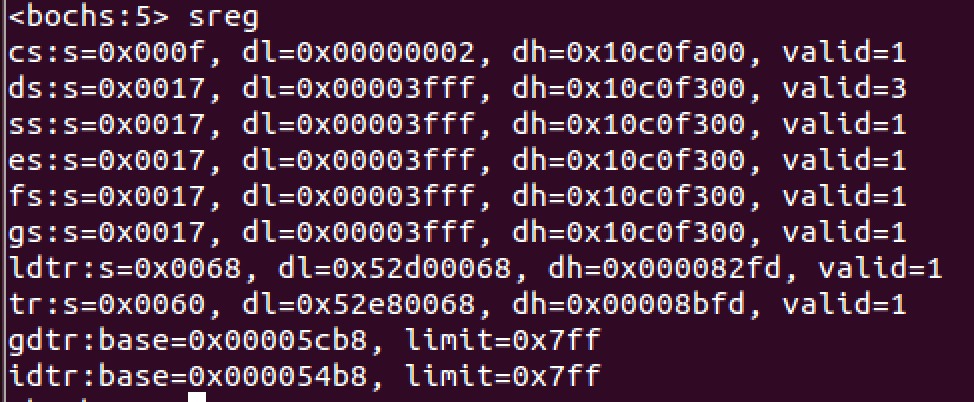
1. 在控制台运行“CTRL/COMMAND+C”，Bochs会暂停运行，进入调试状态，虽然我的程序停在了test内，但是下一条指令不是“cmp……”，用“n”指令单步运行两次，进入指定位置：



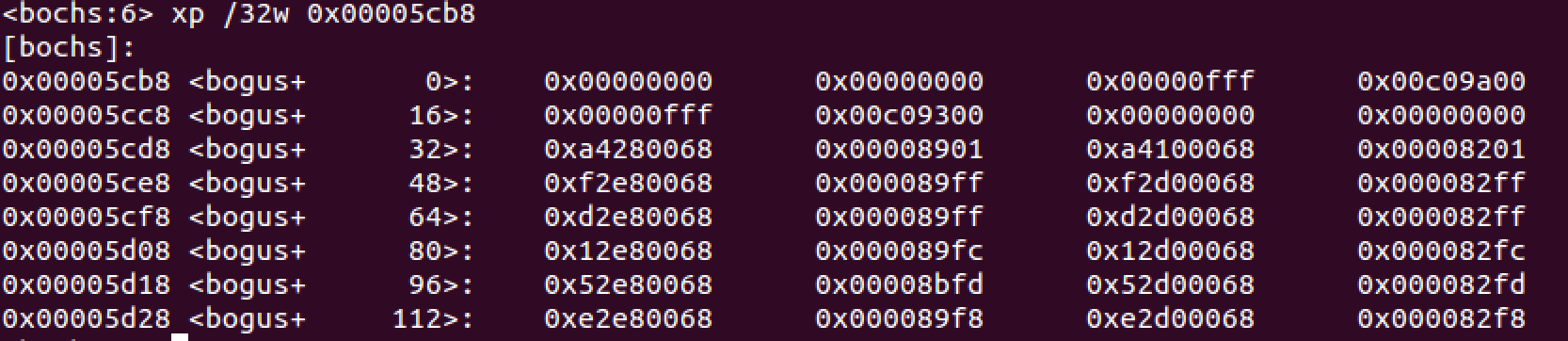
1. 使用指令“u/7”，显示从当前位置开始的7条指令的反汇编代码，如下图：



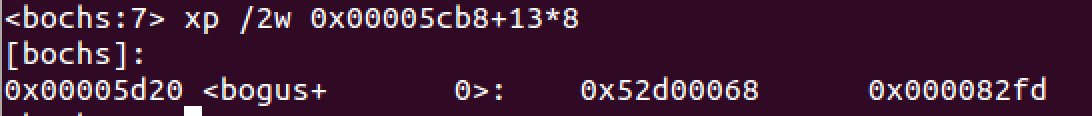
1. ds:0x3004是虚拟地址，ds表明这个地址属于ds段。我们需要首先找到段表，然后通过ds的值在段表中找到ds段的具体信息，才能继续进行地址翻译。每个在IA-32上运行的程序都有一个段表，叫LDT，段的信息叫段描述符。通过“sreg”指令，我们可以得到如下信息：



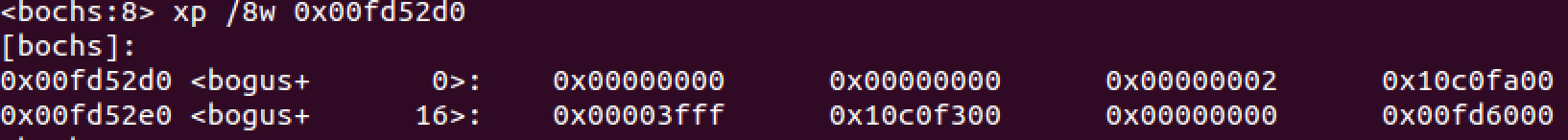
1. 可以看到ldtr的值是0x0068=0000000001101000（二进制），表示LDT表存放在GDT表的1101(二进制)=13（十进制）号位置（每位数据的意义参考后文叙述的段选择子）。而GDT的位置已经由gdtr明确给出，在物理地址的0x00005cb8。用“xp /32w 0x00005cb8”查看从该地址开始，32个字的内容，及GDT表的前16项：



1. GDT表中的每一项占64位（8个字节），所以我们要查找的项的地址是“0x00005cb8 + 13 \* 8”。对应指令“xp /2w 0x00005cb8 + 13 \* 8”，得到：



1. “0x**52d0**0068 0x**00**0082**fd**”将其中的加粗数字组合为“0x00fd52d0”，这就是LDT表的物理地址。通过指令“xp /8w 0x00fd52d0”，得到：



1. 可以看到，ds的值是0x0017。段选择子是一个16位寄存器，它各位的含义如下图：

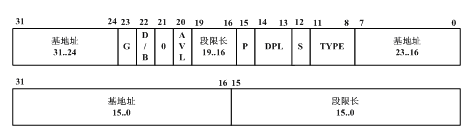
段选择子

其中RPL是请求特权级，当访问一个段时，处理器要检查RPL和CPL（放在cs的位0和位1中，用来表示当前代码的特权级），即使程序有足够的特权级（CPL）来访问一个段，但如果RPL（如放在ds中，表示请求数据段）的特权级不足，则仍然不能访问，即如果RPL的数值大于CPL（数值越大，权限越小），则用RPL的值覆盖CPL的值。而段选择子中的TI是表指示标记，如果TI=0，则表示段描述符（段的详细信息）在GDT（全局描述符表）中，即去GDT中去查；而TI=1，则去LDT（局部描述符表）中去查。

看看上面的ds，0x0017=0000000000010111（二进制），所以RPL=11，可见是在最低的特权级（因为在应用程序中执行），TI=1，表示查找LDT表，索引值为10（二进制）= 2（十进制），表示找LDT表中的第3个段描述符（从0开始编号）。

LDT和GDT的结构一样，每项占8个字节。所以第3项“0x00003fff 0x10c0f300”就是搜寻好久的ds的段描述符了。用“sreg”输出中ds所在行的dl和dh值可以验证找到的描述符是否正确。

接下来看看段描述符里面放置的是什么内容：

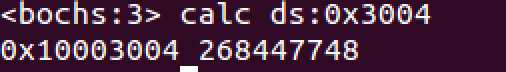


可以看到，段描述符是一个64位二进制的数，存放了段基址和段限长等重要的数据。其中位P（Present）是段是否存在的标记；位S用来表示是系统段描述符（S=0）还是代码或数据段描述符（S=1）；四位TYPE用来表示段的类型，如数据段、代码段、可读、可写等；DPL是段的权限，和CPL、RPL对应使用；位G是粒度，G=0表示段限长以位为单位，G=1表示段限长以4KB为单位；其他内容就不详细解释了。

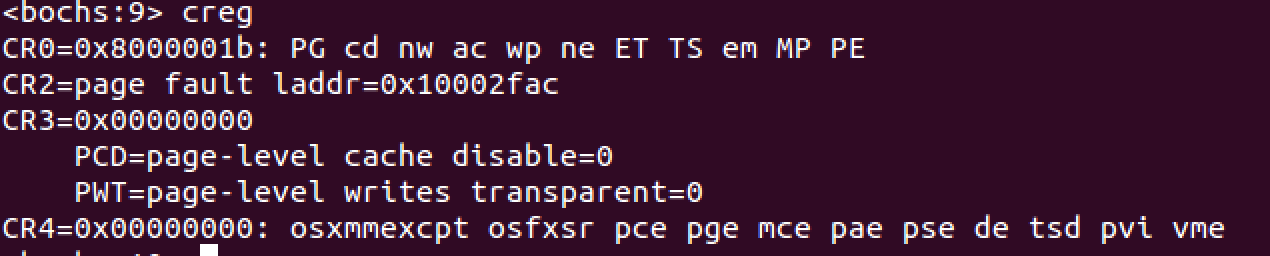
实际上我们需要的只有段基址一项数据，即段描述符“0x00003fff 0x10c0f300”中加粗部分组合成的“0x10000000”。这就是ds段在线性地址空间中的起始地址。用同样的方法也可以算算其它段的基址，都是这个数。段基址+段内偏移，就是线性地址了。所以ds:0x3004的线性地址就是：

0x10000000 + 0x3004 = 0x10003004

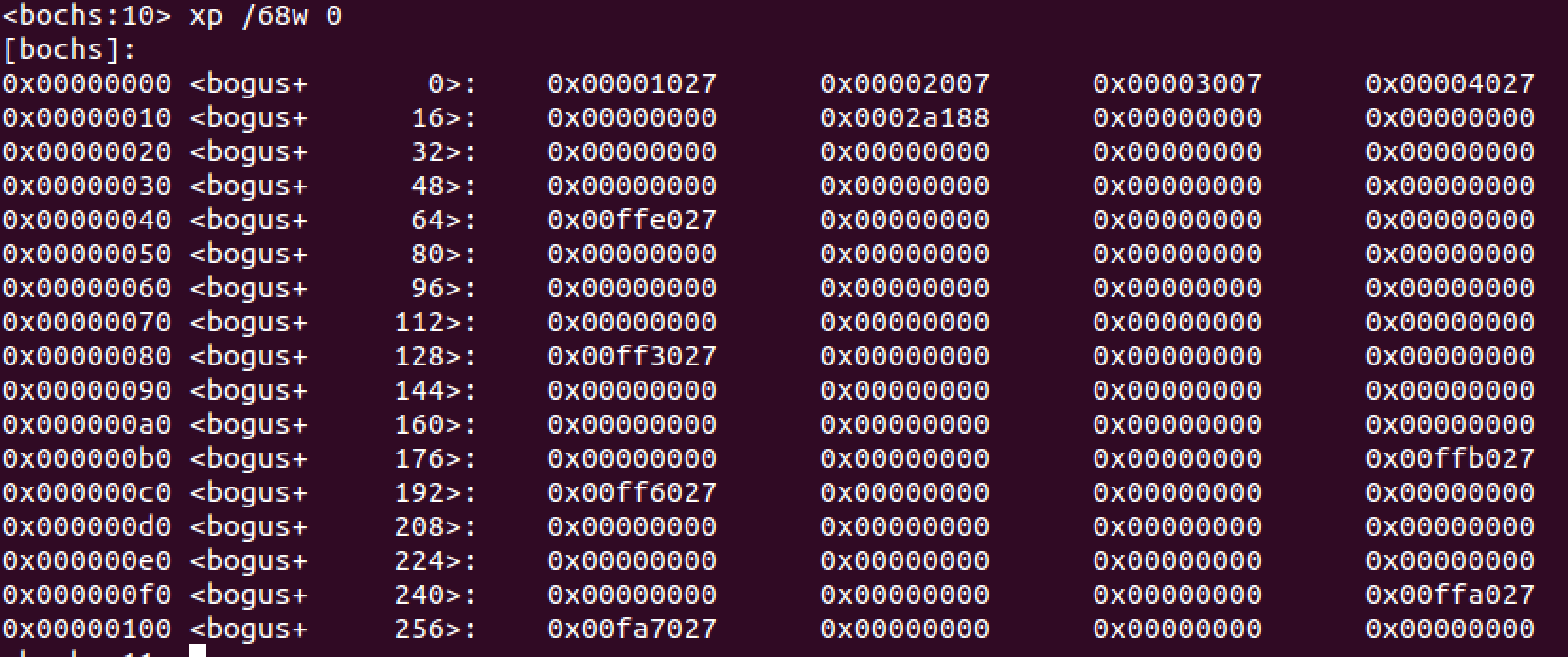
用“calc ds:0x3004”命令可以验证这个结果：



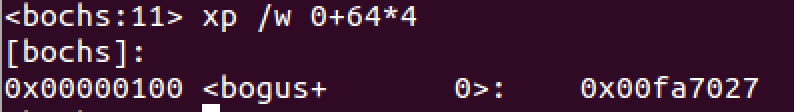
1. 算出线性地址中的页目录号、页表号和页内偏移，它们分别对应了32位线性地址的10位+10位+12位，所以0x10003004的页目录号是64，页号3，页内偏移是4。IA-32下，页目录表的位置由CR3寄存器指引。“creg”命令可以看到：



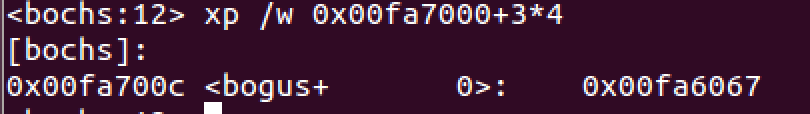
1. 页目录表的基址为0。看看其内容，指令“xp /68w 0”：



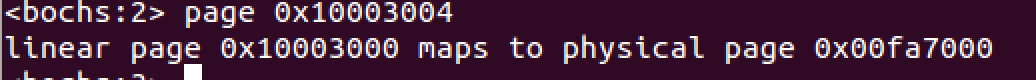
1. 页目录表和页表中的内容很简单，是1024个32位（正好是4K）数。这32位中前20位是物理页框号，后面是一些属性信息（其中最重要的是最后一位P）。其中第65个页目录项就是我们要找的内容，用“xp /w 0+64\*4”查看：



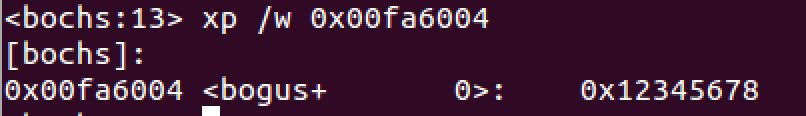
1. 其中的027是属性，显然P=1，页表所在物理页框号为0x00fa7，即页表在物理内存的0x00fa7000位置。从该位置开始查找3号页表项，指令“xp /w 0x00fa7000+3\*4”：



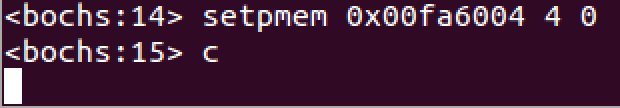
1. 线性地址0x10003004对应的物理页框号为0x00fa6，和页内偏移0x004接到一起，得到0x00fa6004，这就是变量i的物理地址。可以通过两种方法验证：
   * 1. 用命令“page 0x10003004”，可以得到信息：



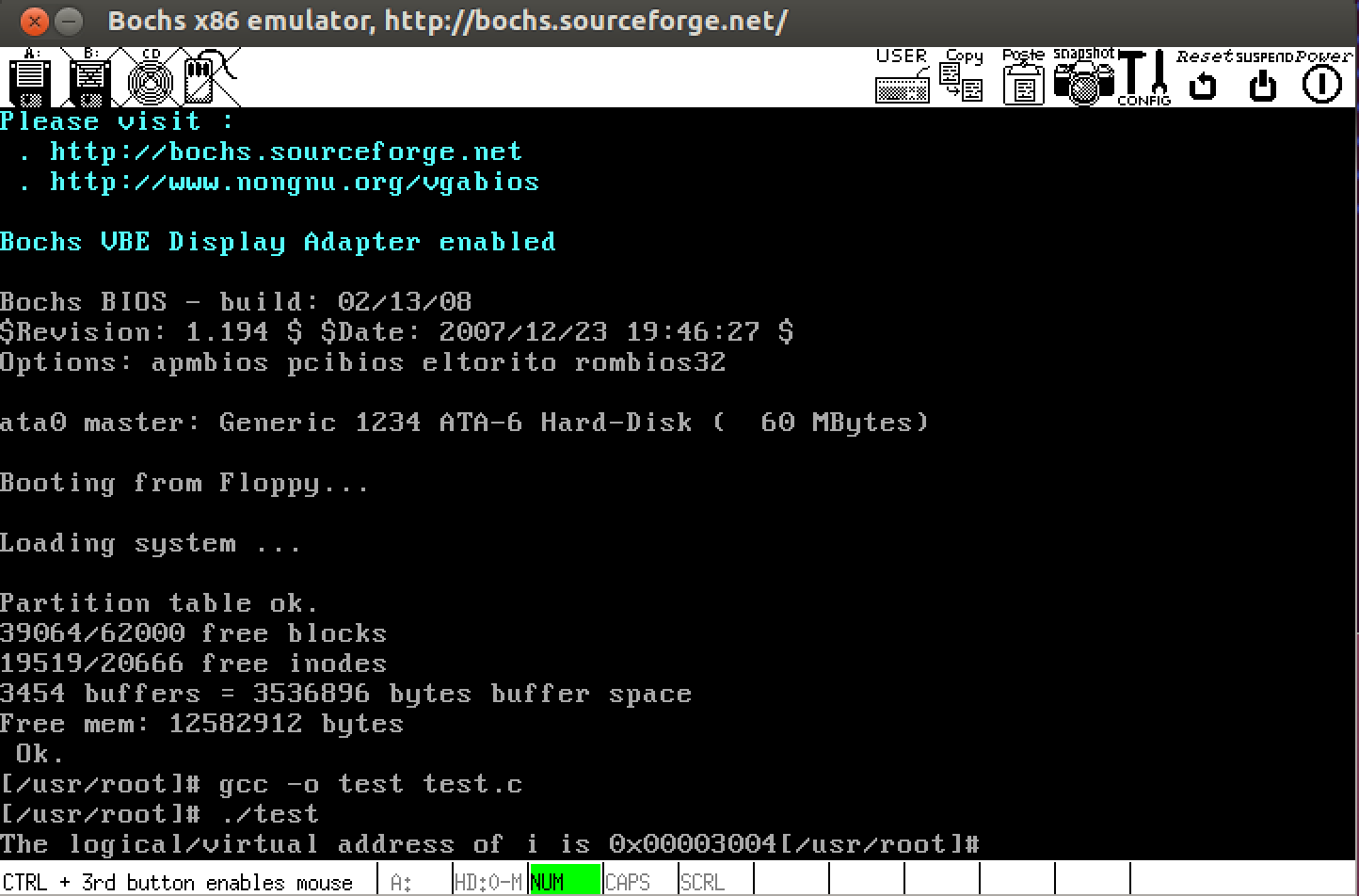
* + 1. 用命令“xp /w 0x00fa6004”，可以看到（即为i的初值）：



* 1. 通过直接修改内存来改变i的值为0，命令是： setpmem 0x00fa6004 4 0，表示从0x00fa6004地址开始的4个字节都设为0，然后再用“c”命令继续Bochs的运行，可以看到test退出了，说明i的修改成功了：



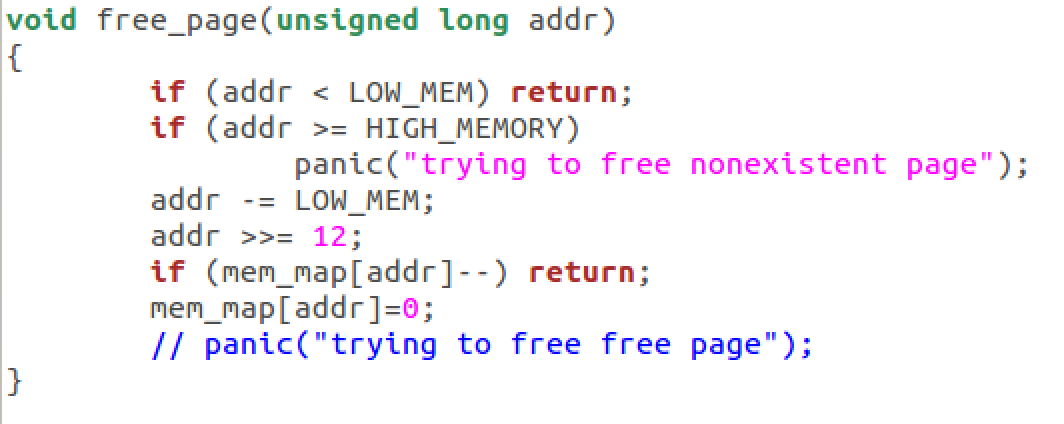
Bochs停止运行截图：



至此，该实验结束！

进行共享内存实验

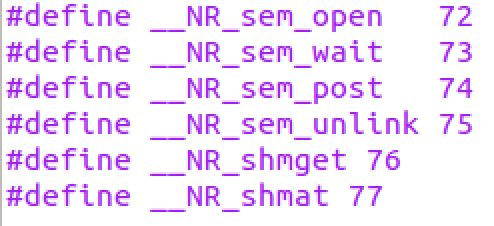
* 1. 修改linux-0.11/mm/目录下的memory.c文件，注释一行（防止报错）：

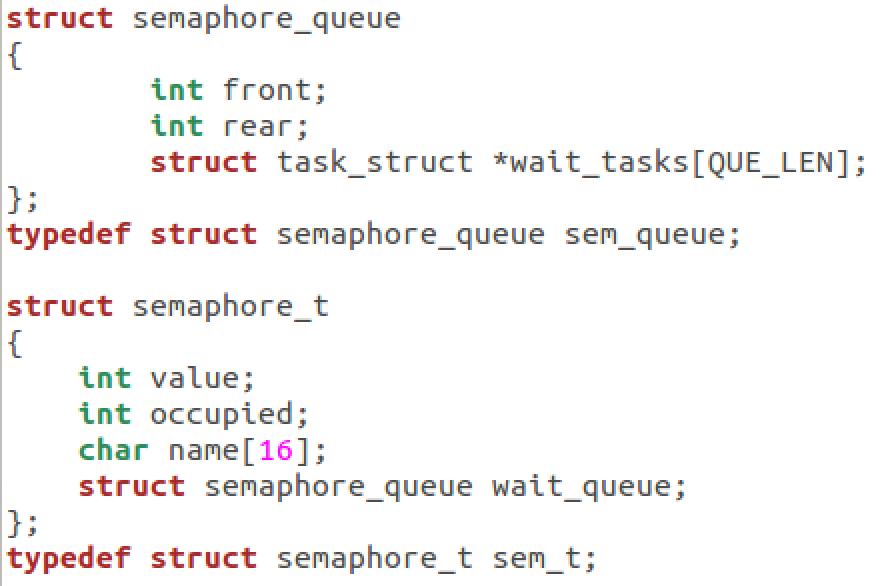


* 1. 在linux-0.11/mm/目录下编写shm.c文件，并在Makefile中做一些添加：

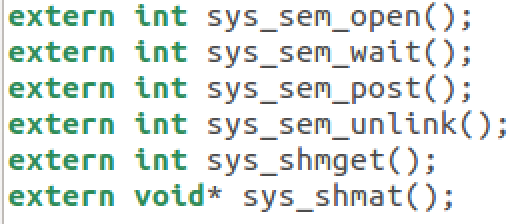


* 1. 修改linux-0.11/include/目录下的unistd.h文件，添加系统调用及声明结构体：

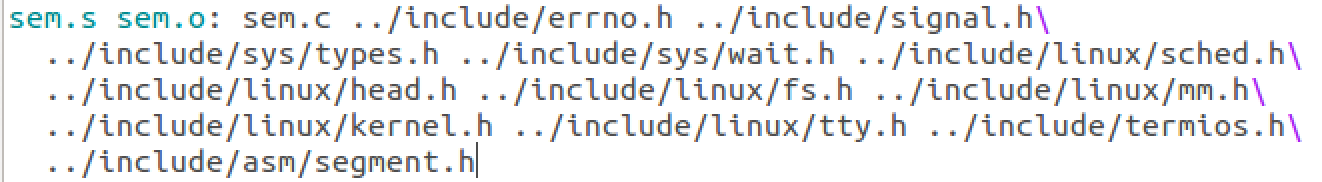




* 1. 在linux-0.11/include/linux/目录下的sys.h文件中，添加如下系统函数信息（sys\_call\_table[]中也添加，没截图）：



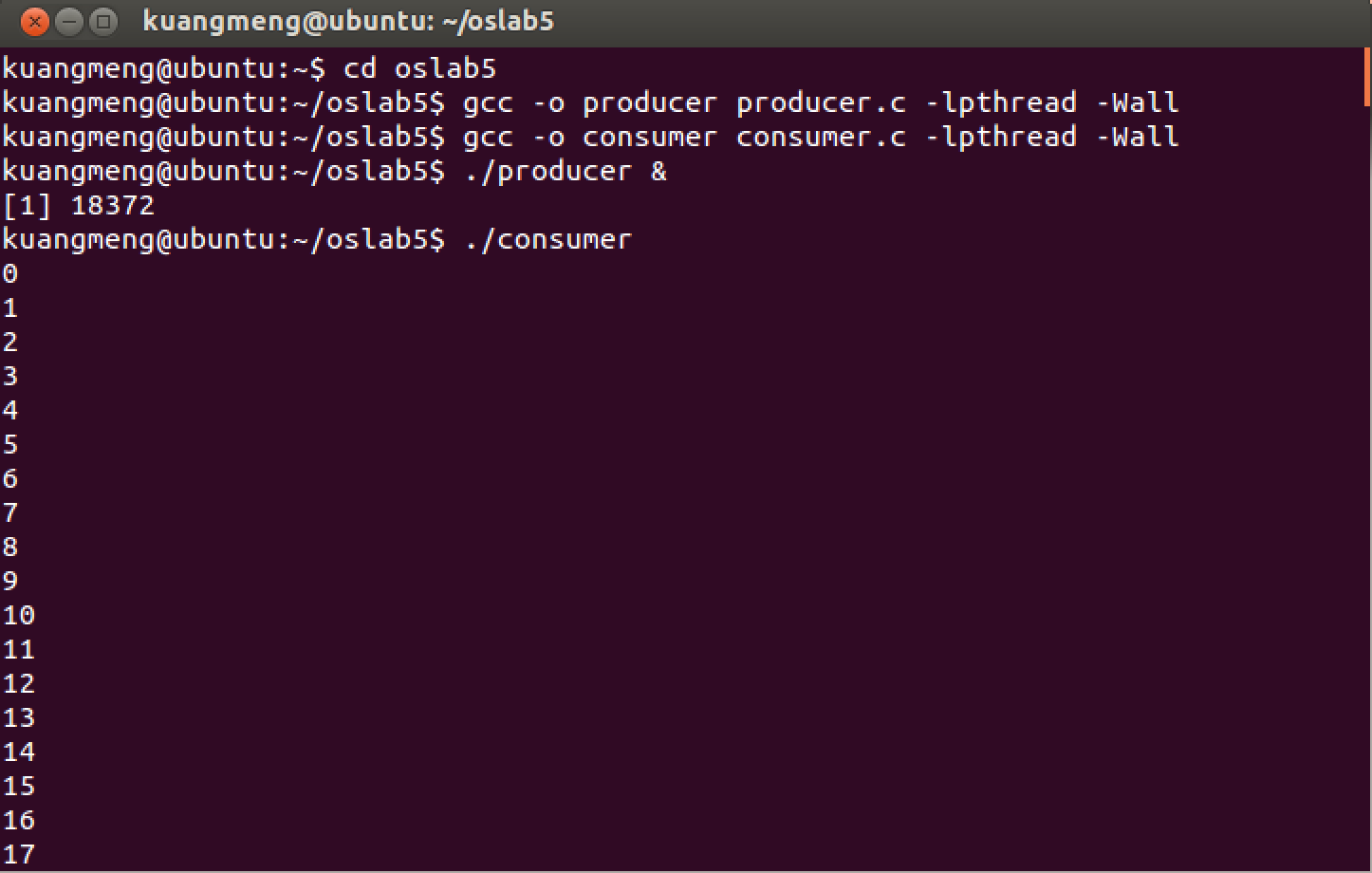
* 1. 在linux-0.11/kernel/目录下编写sem.c文件，并添加以下内容到Makefile中：

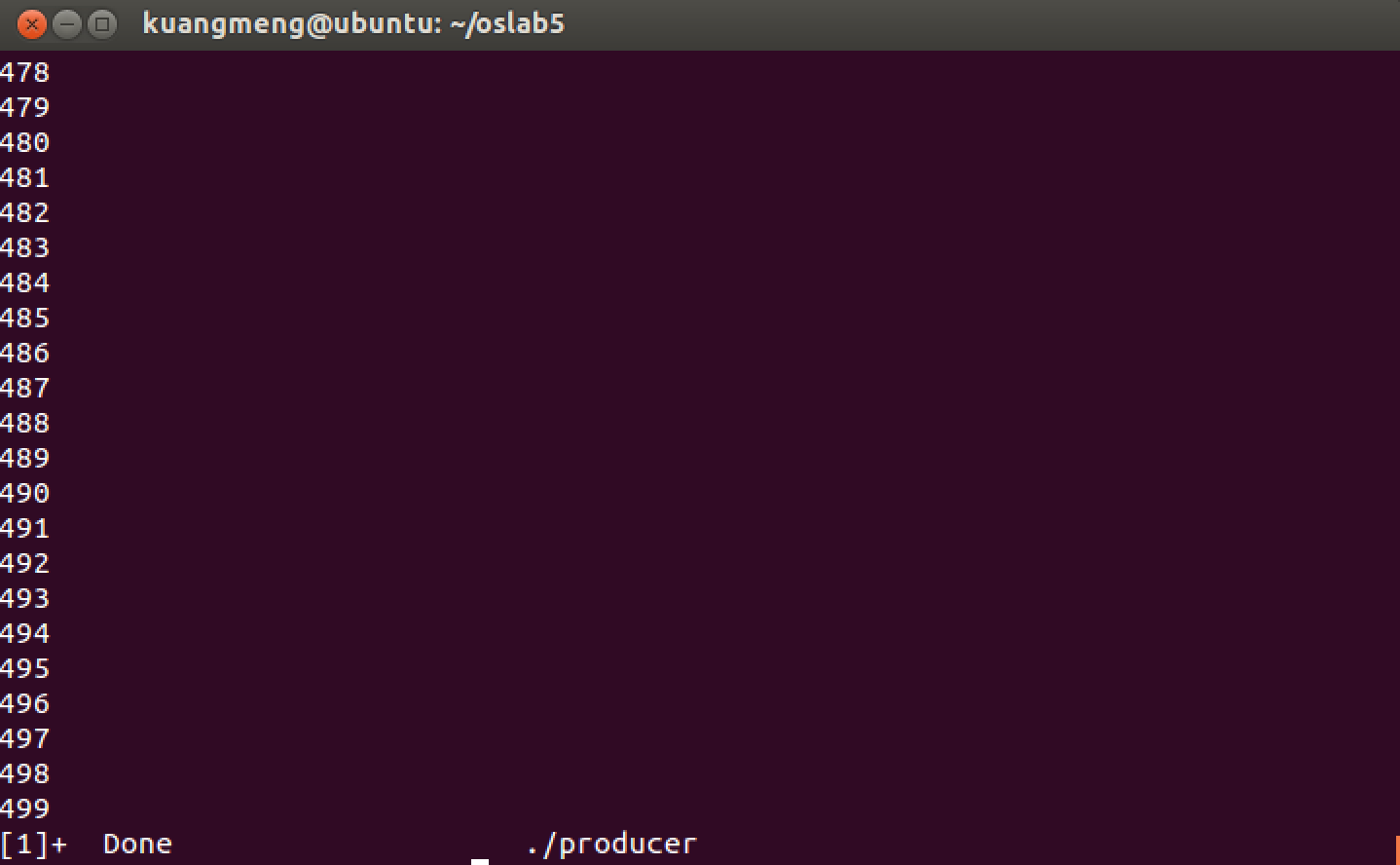


* 1. 修改linux-0.11/kernel/目录下system\_call.s文件，修改如下值：

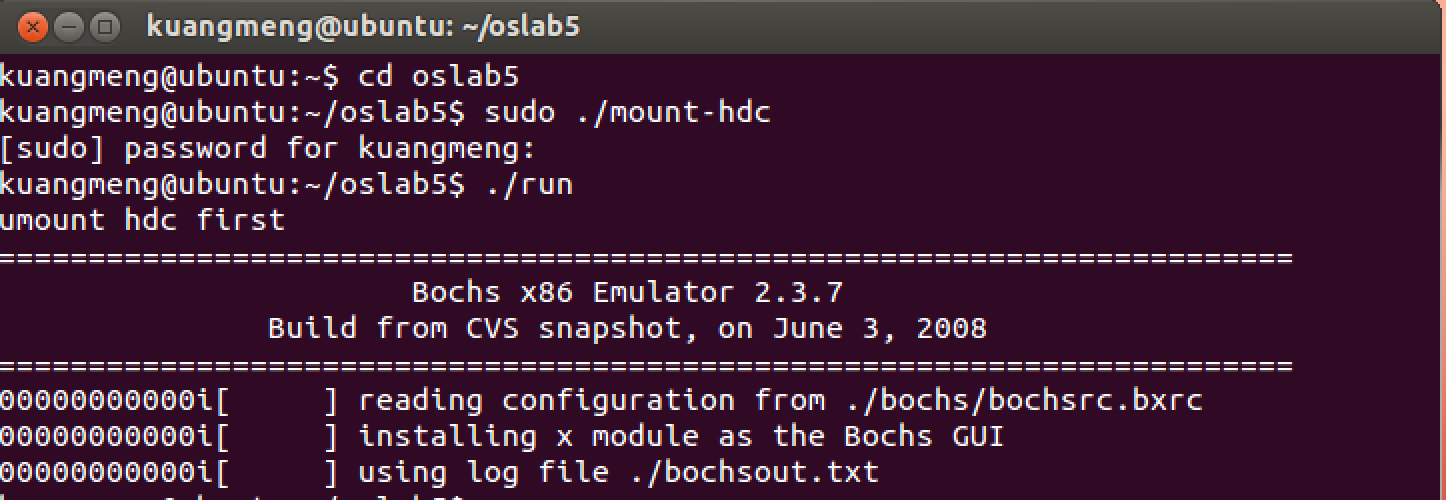


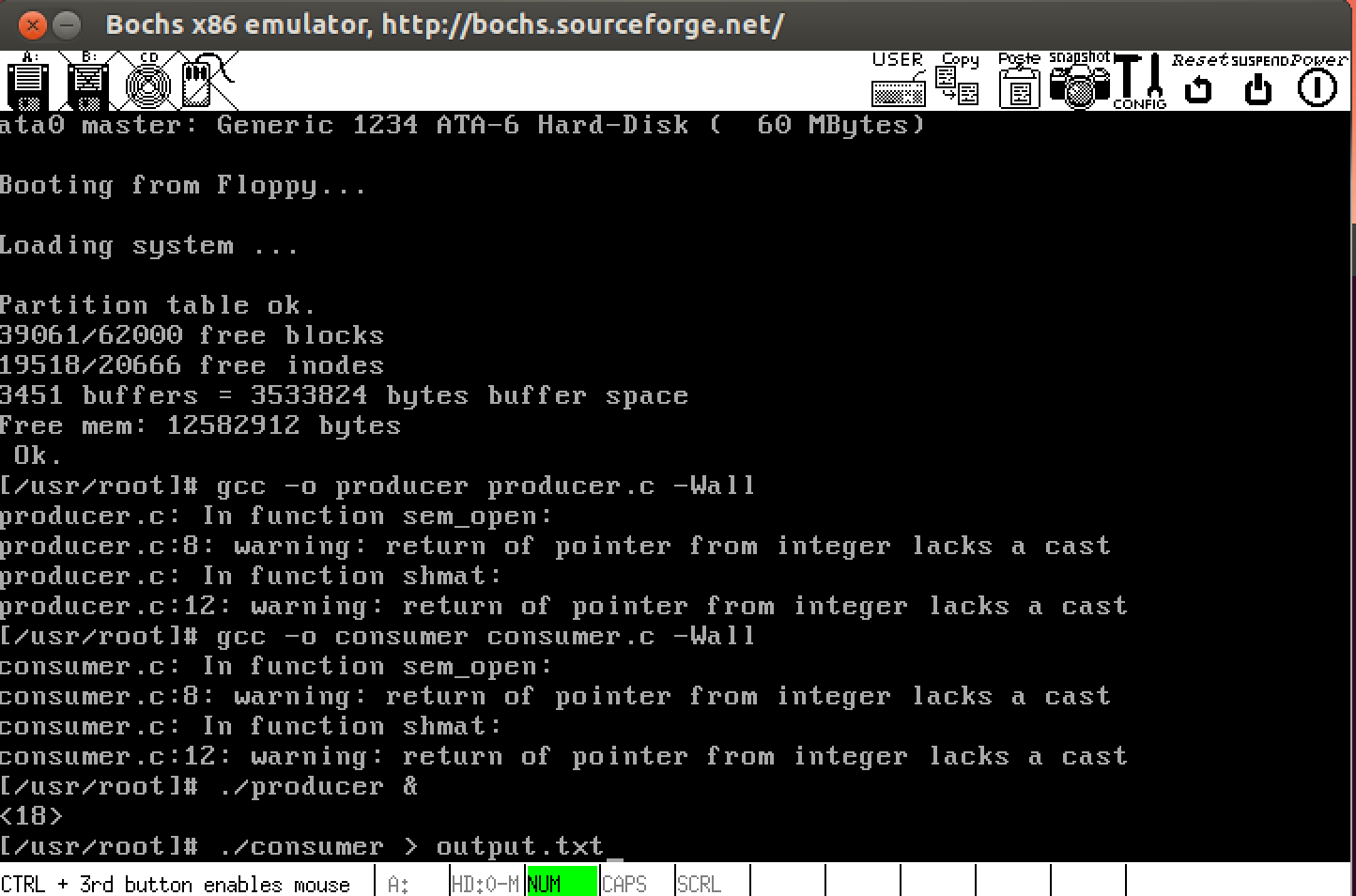
* 1. 编写用于Ubuntu下运行的consumer.c及producer.c，直接放在oslab5/目录下，在控制台中编译运行，截图如下：





* 1. 编写linux-0.11系统下使用的consumer.c及producer.c文件，挂载linux-0.11，修改hdc/usr/include/目录下的unistd.h，保持与之前的一致，将上述两文件放到hdc/usr/root/目录下，在bochs中编译运行，并输出到output.txt中，通过vim打开如下内容：







四、回答问题

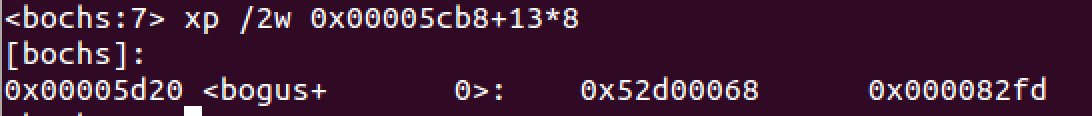
1. 对于地址映射实验部分，列出你认为最重要的那几步（不超过**4**步），并给出你获得的实验数据。

答： 前提：知道段选择子（16位），段描述符（64位），页目录表及页表项（32位）的意义

第一步：首先需要获得逻辑地址，***LDT***的地址

逻辑地址 0x00003004

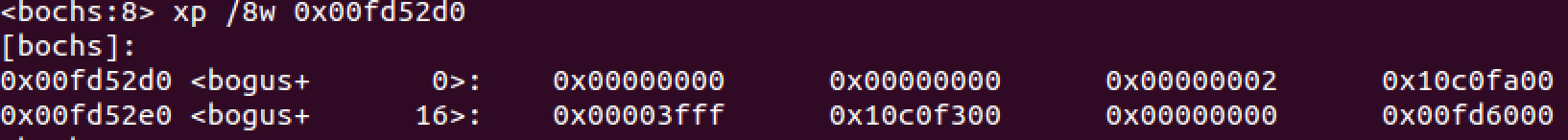
全局描述符表物理地址： gdtr:base=0x00005cb8, limit=0x7ff

而局部描述符选择子：ldtr:s=0x0068 二进制0000,0000,0110,1000b 1101即GDT表第14项

即LDT的物理地址： 0x00fd52d40

第二步：需要获得线性地址

ds段选择子 ds:s=0x0017 即LDT表中第3项



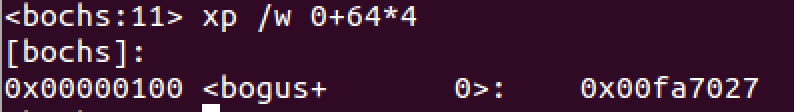
即ds段基地址为：0x10000000

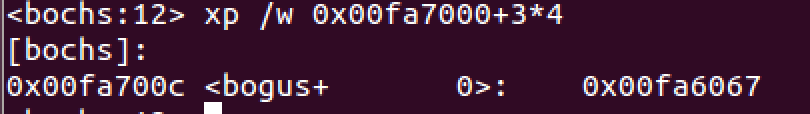
所以线性地址为：0x10003004

第三步：获得页目录及页表地址计算得物理地址

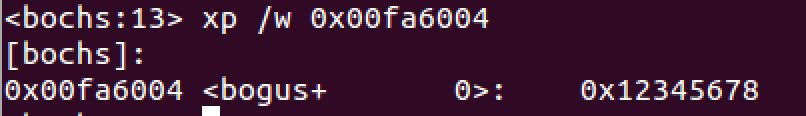
CR3内容: CR3=0x00000000

虚拟地址对应页目录第65项，页表第4项，页表内偏移0x004

查询页目录第65项：

页表物理地址为： 0x00fa7000, 查询第4项

页框物理地址：0x00fa6000 加上偏移0x004

计算得到物理地址： 0x00fa6004（此处为i的初值）

1. **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：逻辑地址和虚拟地址不变，页目录地址是操作系统放置的， 物理分页变了，所以物理地址也变了。原因是每次进程加载后都有64M的虚拟地址空间，而且，逻辑地址没有变化。操作系统加载程序时，由于虚拟地址是按nr分配64M，两次运行nr一致，所以虚拟地址没变。

附录

1. shm.c

#include <asm/segment.h>

#include <linux/kernel.h>

#include <unistd.h>

#include <string.h>

#include <linux/sched.h>

#define SHM\_COUNT 20

#define SHM\_NAME\_SIZE 20

struct struct\_shm\_tables{

int occupied;

char name[SHM\_NAME\_SIZE];

long addr;

} shm\_tables[SHM\_COUNT];

int find\_shm\_location(char \*name){

int i;

for(i=0; i<SHM\_COUNT; i++){

if(!strcmp(name,shm\_tables[i].name) && shm\_tables[i].occupied ==1){

return i;

}

}

return -1;

}

int sys\_shmget(char \* name){

int i,shmid;

char tmp[SHM\_NAME\_SIZE];

for(i=0; i<SHM\_NAME\_SIZE; i++){

tmp[i] = get\_fs\_byte(name+i);

if(tmp[i] == '\0') break;

}

shmid = find\_shm\_location(tmp);

if( shmid != -1) {

return shmid;

}

for(i=0; i<SHM\_COUNT; i++){

if(shm\_tables[i].occupied == 0){

strcpy(shm\_tables[i].name,tmp);

shm\_tables[i].occupied = 1;

shm\_tables[i].addr = get\_free\_page();

return i;

}

}

printk("SHM Number limited!\n");

return -1;

}

void \* sys\_shmat(int shmid){

if(shm\_tables[shmid].occupied != 1){

printk("SHM not exists!\n");

return -1;

}

put\_page(shm\_tables[shmid].addr,current->brk + current->start\_code);

return (void\*)current->brk;

}

1. sem.c

#define \_\_LIBRARY\_\_

#include <unistd.h>

#include <linux/sched.h>

#include <linux/kernel.h>

#include <asm/segment.h>

#include <asm/system.h>

#define SEM\_COUNT 32

sem\_t semaphores[SEM\_COUNT];

/\*队列相关操作，rear始终是下一个待写入的位置，front始终是队列第一个元素\*/

void init\_queue(sem\_queue\* q) {

q->front = q->rear = 0;

}

int is\_empty(sem\_queue\* q){

return q->front == q->rear?1:0;

}

/\*留下标QUE\_LEN-1不用，判断是否慢\*/

int is\_full(sem\_queue\* q){

return (q->rear+1)%QUE\_LEN == q->front?1:0;

}

/\*获得队列头第一个任务\*/

struct task\_struct \* get\_task(sem\_queue\* q){

if(is\_empty(q)){

printk("Queue is empty!\n");

return NULL;

}

struct task\_struct \*tmp = q->wait\_tasks[q->front];

q->front = (q->front+1)%QUE\_LEN;

return tmp;

}

/\*任务插入队列尾\*/

int insert\_task(struct task\_struct \*p,sem\_queue\* q){

// printk("Insert %d",p->pid);

if(is\_full(q)){

printk("Queue is full!\n");

return -1;

}

q->wait\_tasks[q->rear] = p;

q->rear = (q->rear+1)%QUE\_LEN;

return 1;

}

/\*信号量是否已打开，是返回位置\*/

int sem\_location(const char\* name){

int i;

for(i = 0;i < SEM\_COUNT; i++) {

if(strcmp(name,semaphores[i].name) == 0 && semaphores[i].occupied == 1){

return i;

}

}

return -1;

}

/\*打开信号量\*/

sem\_t\* sys\_sem\_open(const char\* name,unsigned int value){

char tmp[16];

char c;

int i;

for( i = 0; i<16; i++){

c = get\_fs\_byte(name+i);

tmp[i] = c;

if(c =='\0') break;

}

if(c >= 16){

printk("Semaphore name is too long!");

return NULL;

}

if((i = sem\_location(tmp)) != -1){

return &semaphores[i];

}

for(i = 0;i< SEM\_COUNT; i++){

if(!semaphores[i].occupied){

strcpy(semaphores[i].name,tmp);

semaphores[i].occupied = 1;

semaphores[i].value = value;

init\_queue(&(semaphores[i].wait\_queue));

// printk("%d %d %d %s\n",semaphores[i].occupied,i,semaphores[i].value,semaphores[i].name);

// printk("%p\n",&semaphores[i]);

return &semaphores[i];

}

}

printk("Numbers of semaphores are limited!\n");

return NULL;

}

/\*P原子操作\*/

int sys\_sem\_wait(sem\_t\* sem){

cli();

sem->value--;

if(sem->value < 0){

/\*参见sleep\_on\*/

current->state = TASK\_UNINTERRUPTIBLE;

insert\_task(current,&(sem->wait\_queue));

schedule();

}

sti();

return 0;

}

/\*V原子操作\*/

int sys\_sem\_post(sem\_t\* sem){

cli();

struct task\_struct \*p;

sem->value++;

if(sem->value <= 0){

p = get\_task(&(sem->wait\_queue));

if(p != NULL){

(\*p).state = TASK\_RUNNING;

}

}

sti();

return 0;

}

/\*释放信号量\*/

int sys\_sem\_unlink(const char \*name){

char tmp[16];

char c;

int i;

for( i = 0; i<16; i++){

c = get\_fs\_byte(name+i);

tmp[i] = c;

if(c =='\0') break;

}

if(c >= 16){

printk("Semphore name is too long!");

return -1;

}

int ret = sem\_location(tmp);

if(ret != -1){

semaphores[ret].value = 0;

strcpy(semaphores[ret].name,"\0");

semaphores[ret].occupied = 0;

return 0;

}

return -1;

}

1. 用于linux-0.11使用的producer.c

#define \_\_LIBRARY\_\_

#include <unistd.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

\_syscall2(sem\_t\*,sem\_open,const char \*,name,unsigned int,value);

\_syscall1(int,sem\_wait,sem\_t\*,sem);

\_syscall1(int,sem\_post,sem\_t\*,sem);

\_syscall1(int,sem\_unlink,const char \*,name);

\_syscall1(void\*,shmat,int,shmid);

\_syscall1(int,shmget,char\*,name);

#define NUMBER 520 /\*打出数字总数\*/

#define BUFSIZE 10 /\*缓冲区大小\*/

sem\_t \*empty, \*full, \*mutex;

int main(){

int i,shmid;

int \*p;

int buf\_in = 0; /\*写入缓冲区位置\*/

/\*打开信号量\*/

if((mutex = sem\_open("carpelamutex",1)) == SEM\_FAILED){

perror("sem\_open() error!\n");

return -1;

}

if((empty = sem\_open("carpelaempty",10)) == SEM\_FAILED){

perror("sem\_open() error!\n");

return -1;

}

if((full = sem\_open("carpelafull",0)) == SEM\_FAILED){

perror("sem\_open() error!\n");

return -1;

}

shmid = shmget("buffer");

if(shmid == -1){

return -1;

}

p = (int\*) shmat(shmid);

/\*生产者进程\*/

for( i = 0 ; i < NUMBER; i++){

sem\_wait(empty);

sem\_wait(mutex);

p[buf\_in] = i;

buf\_in = ( buf\_in + 1)% BUFSIZE;

sem\_post(mutex);

sem\_post(full);

}

/\*释放信号量\*/

sem\_unlink("carpelafull");

sem\_unlink("carpelaempty");

sem\_unlink("carpelamutex");

return 0;

}

1. 用于linux-0.11使用的consumer.c

#define \_\_LIBRARY\_\_

#include <unistd.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

\_syscall2(sem\_t\*,sem\_open,const char \*,name,unsigned int,value);

\_syscall1(int,sem\_wait,sem\_t\*,sem);

\_syscall1(int,sem\_post,sem\_t\*,sem);

\_syscall1(int,sem\_unlink,const char \*,name);

\_syscall1(void\*,shmat,int,shmid);

\_syscall1(int,shmget,char\*,name);

#define NUMBER 520 /\*打出数字总数\*/

#define BUFSIZE 10 /\*缓冲区大小\*/

sem\_t \*empty, \*full, \*mutex;

int main(){

int i,shmid,data;

int \*p;

int buf\_out = 0; /\*从缓冲区读取位置\*/

/\*打开信号量\*/

if((mutex = sem\_open("carpelamutex",1)) == SEM\_FAILED){

perror("sem\_open() error!\n");

return -1;

}

if((empty = sem\_open("carpelaempty",10)) == SEM\_FAILED){

perror("sem\_open() error!\n");

return -1;

}

if((full = sem\_open("carpelafull",0)) == SEM\_FAILED){

perror("sem\_open() error!\n");

return -1;

}

shmid = shmget("buffer");

if(shmid == -1){

return -1;

}

p = (int \*)shmat(shmid);

for( i = 0; i < NUMBER; i++ ){

sem\_wait(full);

sem\_wait(mutex);

data = p[buf\_out];

buf\_out = (buf\_out + 1) % BUFSIZE;

sem\_post(mutex);

sem\_post(empty);

/\*消费资源\*/

printf("%d: %d\n",getpid(),data);

fflush(stdout);

}

/\*释放信号量\*/

sem\_unlink("carpelafull");

sem\_unlink("carpelaempty");

sem\_unlink("carpelamutex");

return 0;

}

1. 用于Ubuntu使用的producer.c

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <semaphore.h>

#include <sys/shm.h>

int main(){

sem\_t \*empty,\*full,\*mutex;

empty=(sem\_t \*)sem\_open("empty",O\_CREAT,0777,10);

full=(sem\_t \*)sem\_open("full",O\_CREAT,0777,0);

mutex=(sem\_t \*)sem\_open("mutex",O\_CREAT,0777,1);

int i,\*buff,buffsig;

buffsig=shmget(666666,sizeof(int)\*10,IPC\_CREAT|0666);

if(buffsig==-1){

printf("Share Memory Error!\n");

return 0;

}

buff=(int \*)shmat(buffsig,NULL,0);

for(i=0;i<500;i++){

sem\_wait(empty);

sem\_wait(mutex);

buff[i%10]=i;

sem\_post(mutex);

sem\_post(full);

}

sem\_unlink("empty");

sem\_unlink("full");

sem\_unlink("mutex");

return 0;

}

1. 用于Ubuntu使用的consumer.c

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <semaphore.h>

#include <sys/shm.h>

int main(){

sem\_t \*empty,\*full,\*mutex;

empty=(sem\_t \*)sem\_open("empty",O\_CREAT,0777,10);

full=(sem\_t \*)sem\_open("full",O\_CREAT,0777,0);

mutex=(sem\_t \*)sem\_open("mutex",O\_CREAT,0777,1);

int i,\*buff,buffsig;

buffsig=shmget(666666,sizeof(int)\*10,IPC\_CREAT|0666);

if(buffsig==-1){

printf("Share Memory Error!\n");

return 0;

}

buff=(int \*)shmat(buffsig,NULL,0);

for(i=0;i<500;i++){

sem\_wait(full);

sem\_wait(mutex);

printf("%d\n",buff[i%10]);

fflush(stdout);

sem\_post(mutex);

sem\_post(empty);

}

sem\_unlink("empty");

sem\_unlink("full");

sem\_unlink("mutex");

return 0;

}