

# 操作系统

## 实 验 报 告

实验名称： 地址映射与共享

学 员： 侯华玮 学 号： 202102001015

培养类型： 无军籍学员 年 级： 21 级

专业： 计算机科学与技术（天河拔尖班） 所属学院： 计算机学院

指导教员： 文艳军 职 称： 教授

实 验 室：                      实验日期： 2023. 04. 12

国防科技大学训练部制

## 《实验报告》填写说明

1. 学员完成人才培养方案和课程标准要所要求的每个实验后，均须提交实验报告。

2. 实验报告封面必须打印，报告内容可以手写或打印。

3. 实验报告内容编排及打印应符合以下要求：

(1) 采用 A4 (21cm×29.7cm) 白色复印纸，单面黑字打印。上下左右各侧的页边距均为 3cm；缺省文档网格：字号为小 4 号，中文为宋体，英文和阿拉伯数字为 Times New Roman，每页 30 行，每行 36 字；页脚距边界为 2.5cm，页码置于页脚、居中，采用小 5 号阿拉伯数字从 1 开始连续编排，封面不编页码。

(2) 报告正文最多可设四级标题，字体均为黑体，第一级标题字号为 4 号，其余各级标题为小 4 号；标题序号第一级用“一、”、“二、”……，第二级用“（一）”、“（二）”……，第三级用“1.”、“2.”……，第四级用“（1）”、“（2）”……，分别按序连续编排。

(3) 正文插图、表格中的文字字号均为 5 号。

## 一、实验目的和内容

### 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：

- 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；
- 为 Linux 0.11 增加共享内存功能，编写并测试生产者—消费者程序。

## 二、操作方法与实验步骤

### （一）地址映：跟踪 Liunx 的地址映射过程

#### 1. 主要操作方法：

1. 以汇编级调试的方式启动 Bochs。
2. 在 Linux 0.11 下编译运行测试程序 test.c, 该程序会进入死循环, 不自动退出。
3. 跟踪运行中的测试程序中的变量 i 的地址映射过程：  
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。
4. 查看物理地址处变量 i 的值，直接修改物理内存，使程序退出。

#### 2. 实验步骤与过程：

- 编译 0 号内核源码，用 bochsdbg 开始调试内核。
- 编写并运行测试程序 test.c：

```
#include <stdio.h>

int i = 0x12345678;
int main(void)
{
    printf("The logical/virtual address of i is 0x%08x", &i);
    fflush(stdout);
    while (1)
        ;
    return 0;
}
```

- 在调试器中中断程序执行，并反汇编查看程序指令：

```
Bochs x86 emulator, http://bochs.sourceforge.net/
Bochs VBE Display Adapter enabled
Bochs BIOS - build: 02/16/17
Revision: 13073 $ Date: 2017-02-16 22:43:52 +0100 (Do, 16. Feb 2017) $
Options: apmbios pcibios pnpbios eltorito rombios32
ata0 master: Generic 1234 ATA-6 Hard-Disk ( 10 MBytes)
Press F12 for boot menu.

nudu@ubuntu: ~/os/linux-0.11-lab
cs:0x000f, dh=0x10c0fb00, dl=0x00000002, valid=1
Code segment, base=0x10000000, limit=0x00002fff, Execute/Read, Non-Conforming, Accessed, 32-bit
ss:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ds:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=3
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
fs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
gs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ldtr:0x0068, dh=0x000082fa, dl=0x12d00068, valid=1
tr:0x0060, dh=0x00008bfa, dl=0x12e80068, valid=1
gdtr:base=0x00005cc0, limit=0x7fff
ldtr:base=0x000054c0, limit=0x7fff
<bochs:15> u /7
10000057: (      ): cmp dword ptr ds:0x0003004, 0x00000000 ; 833d0430000000
10000058: (      ): jz .+4 ; 7404
10000059: (      ): jmp -11 ; eb15
10000062: (      ): add byte ptr ds:[eax], al ; 0000
10000064: (      ): xor eax, eax ; 31c0
10000066: (      ): jmp .+0 ; eb00
10000068: (      ): leave ; c9
<bochs:16>
```

发现当前指令将 ds:0x3004 处的值与 0 比较，并由跳转构成的循环可知，变量 i 就保存在地址 ds:0x3004 中。下面通过段翻译和页翻译找到变量 i 的物理地址。

### • 段翻译：逻辑地址→线性地址

使用 sreg 命令查看段寄存器

```
Bochs x86 emulator, http://bochs.sourceforge.net/
Bochs VBE Display Adapter enabled
Bochs BIOS - build: 02/16/17
Revision: 13073 $ Date: 2017-02-16 22:43:52 +0100 (Do, 16. Feb 2017) $
Options: apmbios pcibios pnpbios eltorito rombios32
ata0 master: Generic 1234 ATA-6 Hard-Disk ( 10 MBytes)
Press F12 for boot menu.

nudu@ubuntu: ~/os/linux-0.11-lab
Next at t=584710736
(0) [0x000000fc7057] 000f:00000057 (unk. ctxt): cmp dword ptr ds:0x0003004, 0x00000000 ;
833d0430000000
<bochs:13> sregs
<bochs:13> syntax error at 'sregs'
<bochs:14> sreg
es:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
cs:0x000f, dh=0x10c0fb00, dl=0x00000002, valid=1
Code segment, base=0x10000000, limit=0x00002fff, Execute/Read, Non-Conforming, Accessed, 32-bit
ss:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ds:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=3
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
fs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
gs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ldtr:0x0068, dh=0x000082fa, dl=0x12d00068, valid=1
tr:0x0060, dh=0x00008bfa, dl=0x12e80068, valid=1
gdtr:base=0x00005cc0, limit=0x7fff
ldtr:base=0x000054c0, limit=0x7fff
<bochs:15>
```

得到对应数据段的选择符 ds = 0x17，则逻辑地址为：0x17: 0x3004

接下来要通过分析段选择符 ds，找到对应的段描述符，获得变量 i 所在段的基址。

ds = 0x17 = b(0001 0111)，其中 TI = 1，INDEX = b(10) = 2，即所寻段的描述符位于局部描述符表 LDT 的 2 号索引处。

如何查局部描述符表呢？根据 386 寻址机制，局部描述符表也被视作“段”，其段描述符位于全局描述符表中，由内存管理寄存器 LDTR 为其选择符。全局描述符表 GDT 起始地址由全局描述符表寄存器 GDTR 给出。

在本示例中， $gdtr = 0x54c0$ ， $ldtr = 0x68 = b(0110\ 1000)$ ， $INDEX = b(01101) = 13$  则当前进程的 LDT 的段描述符位于以  $0x5cc0$  为起始地址的 GDT 的 13 号索引处。由于一个描述符长度为 8 个字节，则应查看  $0x54c0 + 13 * 8 =$  处两个字长的内容。

使用命令 `xp /2w 0x5d28`，结果如下：

```
0x00005d00 <bogus+ 64>: 0x02e80068 0x000089ff 0x02
0x00005d10 <bogus+ 80>: 0x12e80068 0x000089fc 0x12
0x00005d20 <bogus+ 96>: 0x12e80068 0x00008bfa 0x12
<bochs:20> xp /2w 0x5d28
[bochs]:
0x00005d28 <bogus+ 0>: 0x12d00068 0x000082fa
<bochs:21> |
```

根据段描述符的组成结构可得，当前 LDT 的基地址为： $0x00fa12d0$ ，则该 LDT 的第 2 号索引处  $0xfa12d0 + 2 * 8 = 0xfa12e0$  就是 ds 对应段的描述符了。

使用命令 `xp /2w 0xfa12e0`，结果如下：

```
0x00fa12f0 <bogus+ 32>: 0x00000010 0x00000000
<bochs:23> xp /2w 0xfa12e0
[bochs]:
0x00fa12e0 <bogus+ 0>: 0x00003fff 0x10c0f300
<bochs:24> |
```

有  $dl = 0x3fff$ ， $dh = 0x10c0f300$ ，则可得 ds 段在虚拟内存中的基地址为  $0x10000000$ 。

结合  $offset = 0x3004$ ，可得变量 i 的线性（虚拟）地址为  $0x1000\ 3004$ 。

## • 第二步：线性地址→物理地址

根据线性地址的组成结构，可获得：

- 页目录号 DIR =  $b(0001\ 0000\ 00) = 64$
- 页表号 PAGE =  $b(00\ 0000\ 0011) = 3$
- 页内偏移量 OFFSET =  $b(0000\ 0000\ 0100) = 4$

由  $CR3 = 0x00$  获得页目录表的基地址  $0x00$ 。

查看页目录表的第 64 号索引项： $0x00 + 64 * 4 = 0x100$ 。

`xp /w 0x100`

```
<bochs:26> xp /2w 0x100
[bochs]:
0x00000100 <bogus+ 0>: 0x00fb0027 0x00000000
<bochs:27> xp /w 0x100
[bochs]:
0x00000100 <bogus+ 0>: 0x00fb0027
<bochs:28> |
```

得到页目录项为 0x00fb 0027, 其中 P=1, R/W = 1 则该页目录项有效。则二级页表所在物理页框号为 0xfb0, 即起始地址为 0xfb 0000, 从该处查看第三号索引项: 0xfb 0000 + 3 \* 4 = 0xfb 000c。

xp /w 0xfb 000c

```
[bochs]:  
0x00fb000c <bogus+ 0>: 0x00fad067  
<bochs:29> |
```

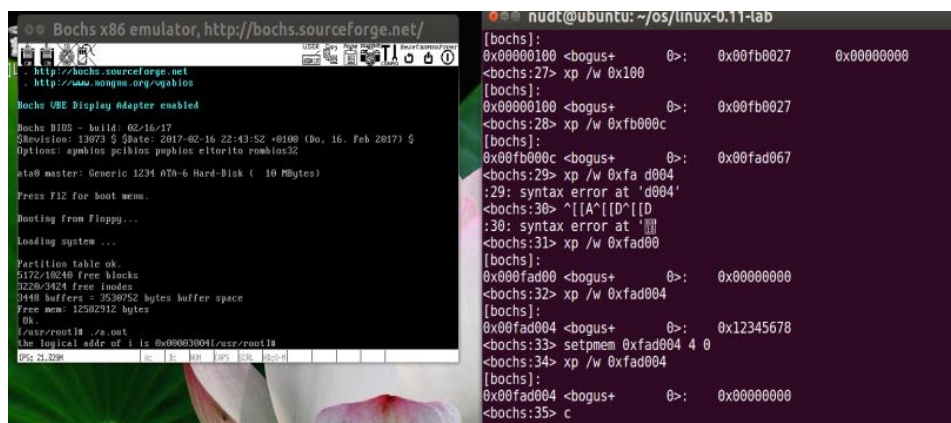
页表项内容为 0xfad067, 则该页表项有效, 可得物理页的页框号为 0xfad, 起始地址为 0xfa d000, 与页内偏移量 0x4 相加, 可得变量 i 所在的物理地址 0xfa d004 查看该地址处内容, 发现确为 i 的值 0x12345678。

```
<bochs:32> xp /w 0xfad004  
[bochs]:  
0x00fad004 <bogus+ 0>: 0x12345678  
<bochs:33> |
```

直接修改 i 的值:

```
0x00fad004 <bogus+ 0>: 0x12345678  
<bochs:33> setpmem 0xfad004 4 0  
<bochs:34> xp /w 0xfad004  
[bochs]:  
0x00fad004 <bogus+ 0>: 0x00000000  
<bochs:35> |
```

在调试器中输入命令 c 继续运行, 发现程序退出, 符合预期。



## (二) 内存共享: 添加内存共享系统调用并编写消费者-生产者测试程序

### 1. 主要操作方法:

1. 修改版本 0 的 Linux 0.11 内核源码, 添加 shmget 与 shmat 两个与内存共享相关的系统调用。
2. 在 bochs 虚拟机中, 编写并测试多进程的生产者消费者程序。

## 2. 实验步骤与过程:

- 添加系统调用

```
int shmget(key_t key, size_t size, int shmflg);
```

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

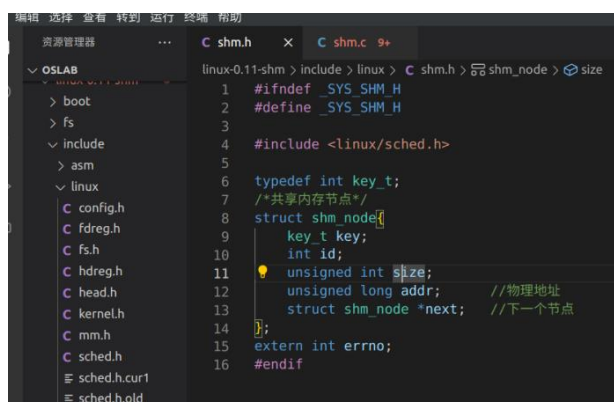
shmget 系统调用

该系统调用会新建/打开一页物理内存作为共享内存，返回该页共享内存的 shmid，即该页共享内存存在操作系统中的标识 id。如果多个进程使用相同的 key 调用 shmget，则这些进程就会获得相同的 shmid，即得到了同一块共享内存的 id。在 shmget 实现时，如果 key 所对应的共享内存已经建立，则直接返回 shmid，否则新建。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 参数可以忽略。

shmat 系统调用

该系统调用会将 shmid 指定的共享页面映射到当前进程的虚拟地址空间中，并返回一个逻辑地址 p，调用进程可以读写逻辑地址 p 来读写这一页共享内存。两个进程都调用 shmat 可以关联到同一页内存上，从而形成进程共享内存页结构，此时两个进程读写 p 指针就是在读写同一页内存，从而实现了基于共享内存的进程间通信。如果 shmid 非法，返回-1，并置 errno 为 EINVAL，shmaddr 和 shmflg 这两个参数都可忽略。

在/linux/include/linux/目录下，添加头文件 shm.h，声明共享内存的结点结构体 shm\_node，即每块分配的共享内存，都会将其对应的共享内存结点添加到一个链表中，从而进行管理维护。

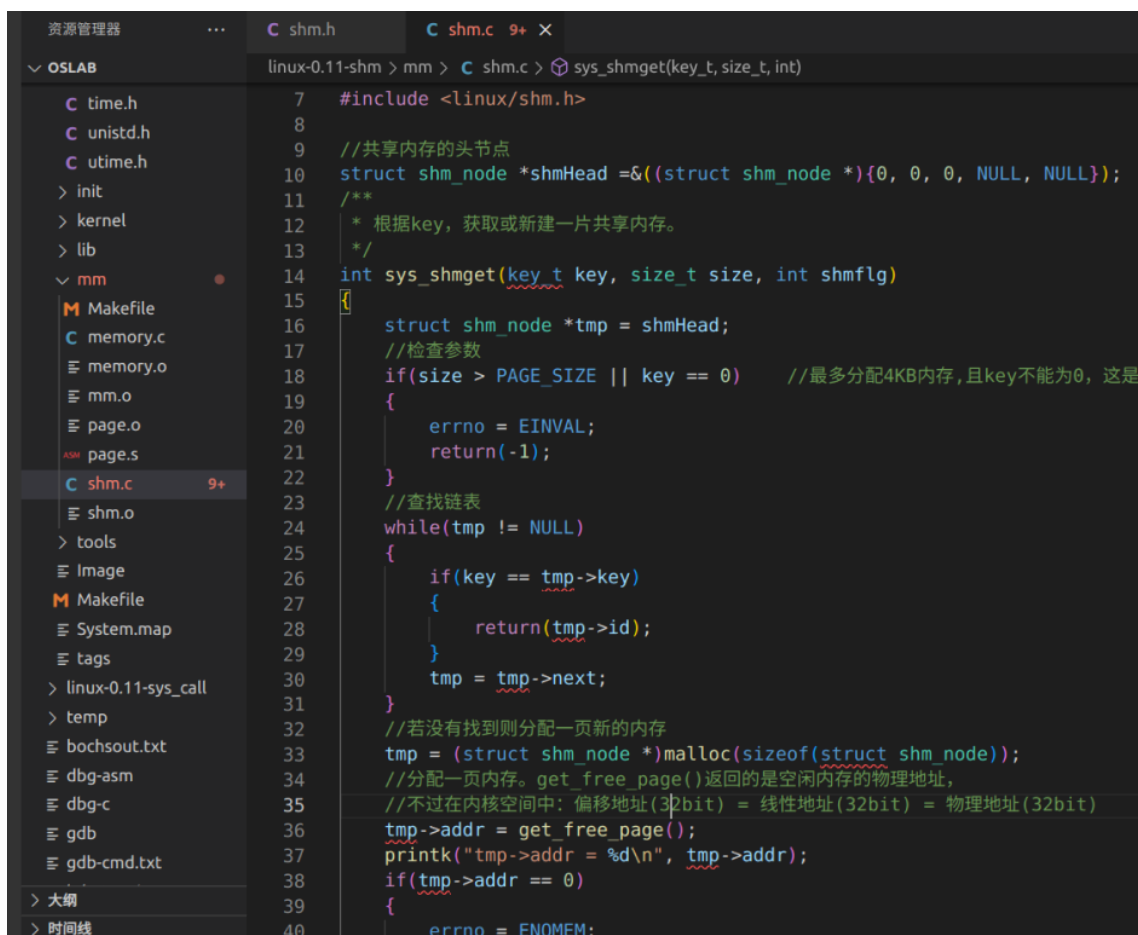


由于这两个系统调用都是对内存进行操作的函数，故在/linux/mm/目录下实现两个系统调用函数：



sys\_shmget 的实现逻辑:

- 先对传入的参数进行检验。
- 根据传入的 key, 查找共享内存结点链表, 若 key 存在, 则直接返回其 shmid。
- 若未找到 key, 说明尚未分配, 则调用 malloc 分配一页新的虚拟内存。
- 将用 get\_free\_page 获得的物理页帧与该块虚拟内存建立映射。



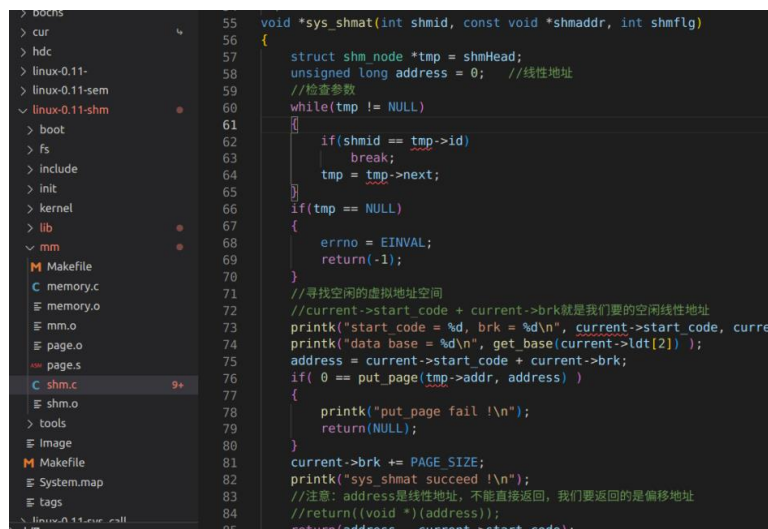
```
7 #include <linux/shm.h>
8
9 //共享内存的头节点
10 struct shm_node *shmHead = &((struct shm_node *){0, 0, 0, NULL, NULL});
11 /**
12  * 根据key, 获取或新建一片共享内存。
13  */
14 int sys_shmget(key_t key, size_t size, int shmflg)
15 {
16     struct shm_node *tmp = shmHead;
17     //检查参数
18     if(size > PAGE_SIZE || key == 0) //最多分配4KB内存, 且key不能为0, 这是
19     {
20         errno = EINVAL;
21         return(-1);
22     }
23     //查找链表
24     while(tmp != NULL)
25     {
26         if(key == tmp->key)
27         {
28             return(tmp->id);
29         }
30         tmp = tmp->next;
31     }
32     //若没有找到则分配一页新的内存
33     tmp = (struct shm_node *)malloc(sizeof(struct shm_node));
34     //分配一页内存。get_free_page()返回的是空闲内存的物理地址,
35     //不过在内核空间中: 偏移地址(32bit) = 线性地址(32bit) = 物理地址(32bit)
36     tmp->addr = get_free_page();
37     printk("tmp->addr = %d\n", tmp->addr);
38     if(tmp->addr == 0)
39     {
40         errno = ENOMEM;
```

sys\_shmat 的实现逻辑:

- 首先检查传入的参数。
- 遍历共享内存链表, 查找与给定 shmid 相匹配的共享内存块。
- 如果找不到相应的共享内存块, 则设置 errno 为 EINVAL 并返回-1。
- 在进程地址空间中寻找空闲的虚拟地址空间, 以便将共享内存映射到该地址空间中。
- 如果找不到可用的地址空间, 则返回 NULL。
- 如果找到可用的地址空间, 则将共享内存块的物理地址映射到该地址空间中。
- 更新进程的 brk 指针, 以便下一次映射能够使用不同的地址空间。



- 返回偏移地址。



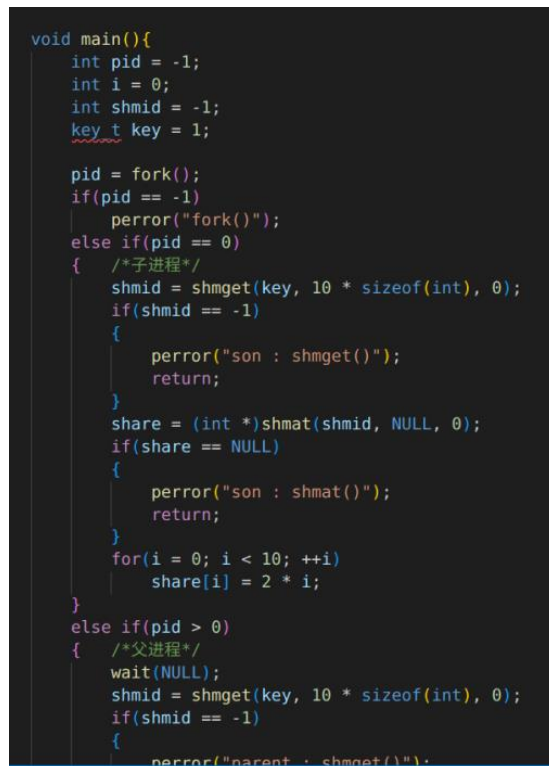
```

55 void *sys_shmat(int shmid, const void *shmaddr, int shmflg)
56 {
57     struct shm_node *tmp = shmHead;
58     unsigned long address = 0; //线性地址
59     //检查参数
60     while(tmp != NULL)
61     {
62         if(shmid == tmp->id)
63             break;
64         tmp = tmp->next;
65     }
66     if(tmp == NULL)
67     {
68         errno = EINVAL;
69         return(-1);
70     }
71     //寻找空闲的虚拟地址空间
72     //current->start_code + current->brk就是我们需要的空闲线性地址
73     printk("start code = %d, brk = %d\n", current->start_code, current->brk);
74     printk("data base = %d\n", get_base(current->ldt[2]));
75     address = current->start_code + current->brk;
76     if(0 == put_page(tmp->addr, address))
77     {
78         printk("put_page fail !\n");
79         return(NULL);
80     }
81     current->brk += PAGE_SIZE;
82     printk("sys_shmat succeed !\n");
83     //注意: address是线性地址, 不能直接返回, 我们要返回的是偏移地址
84     //return((void *) (address));
85     return(address - current->start_code);

```

## 共享内存测试程序

首先, 定义一个 `int` 类型的指针变量 `share`, 用于指向共享内存区域。然后, 它定义了一个 `key_t` 类型的变量 `key`, 并将其设置为 1。这个 `key` 将用于创建共享内存区域。使用 `fork()` 函数创建一个子进程。如果 `fork()` 返回值为 -1, 则输出错误信息并退出程序。如果 `fork()` 返回值为 0, 则进入子进程代码块。



```

void main(){
    int pid = -1;
    int i = 0;
    int shmid = -1;
    key_t key = 1;

    pid = fork();
    if(pid == -1)
        perror("fork()");
    else if(pid == 0)
    { /*子进程*/
        shmid = shmget(key, 10 * sizeof(int), 0);
        if(shmid == -1)
        {
            perror("son : shmget()");
            return;
        }
        share = (int *)shmat(shmid, NULL, 0);
        if(share == NULL)
        {
            perror("son : shmat()");
            return;
        }
        for(i = 0; i < 10; ++i)
            share[i] = 2 * i;
    }
    else if(pid > 0)
    { /*父进程*/
        wait(NULL);
        shmid = shmget(key, 10 * sizeof(int), 0);
        if(shmid == -1)
        {
            perror("parent : shmget()");

```

在子进程代码块中, 代码使用 `shmget()` 函数创建一个共享内存区域, 并将共享内存区域附加到子进程的地址空间中。若分配成功, 则通过 `shmat()` 获得逻辑地址

偏移量，在共享内存区域中写入一些数据。

在父进程代码块中，代码等待子进程结束。然后，它使用 `shmget()` 函数创建一个共享内存区域，并将共享内存区域附加到父进程的地址空间中。如果获取到正确的偏移量，则通过 `shmat()` 获得逻辑地址偏移量，从共享内存区域中读取数据，并在屏幕上输出这些数据。

其中，子进程是生产者进程，父进程是消费者进程，实现了消费者-生产者之间的内存共享。

测试结果如下：

```

a.out      hello      linux-0.00  pc.c      shoelace.tar.Z
buffer_file hello.c      linux0.tgz  sem_output test_shm.c
[/usr/root]# gcc test_shm.c
test_shm.c:10: linux/shm.h: ENOENT
[/usr/root]#
```

```

Partition table ok.
39006/62000 free blocks
19509/20666 free inodes
3449 buffers = 3531776 bytes buffer space
Free mem: 12582912 bytes
Ok.
[/usr/root]# ./a.out
tmp->addr = 16306176
sys_shmget succeed !
start_code = 335544320, brk = 20480
data base = 335544320
sys_shmat succeed !
start_code = 268435456, brk = 20480
data base = 268435456
mem_map disagrees with 00F8D000 at 10005000
sys_shmat succeed !
0, 2, 4, 6, 8, 10, 12, 14, 16, 18,
Kernel panic: trying to free free page
```

### 三、实验结果与分析

#### (一) 地址映射：

成功通过地址映射过程找到了变量的物理地址。

逻辑地址->线性地址->物理地址

在 i386 架构下的 Linux 0.11 中，地址映射是通过段式存储管理实现的。段式存储管理是将进程的虚拟地址空间划分为多个段，每个段都有自己的段基址和长度。当进程访问虚拟地址时，系统将根据虚拟地址所属的段来计算出对应的物理地址。

具体实现时，可以使用描述符表（即 GDT 和 LDT）来存储每个段的信息，并使用段选择子来指定当前进程使用哪个段描述符。

## （二）内存共享：

通过实现 `shmget` 和 `shmat` 系统调用，编写生产者-消费者程序，实现了内存共享。

在 i386 架构下的 Linux 0.11 中内存共享是通过共享内存机制实现的。共享内存是一种进程间通信的方式，它允许多个进程共享同一个物理内存区域。实现共享内存需要使用系统调用 `shmget` 和 `shmat`。`shmget` 用于创建或获取一个共享内存区域的标识符，而 `shmat` 用于将共享内存区域映射到进程的地址空间中。具体实现时，需要使用内存管理功能来实现共享内存区域的分配和管理，但本实验没有使用同步机制（如信号量或互斥锁）来保证数据的一致性和完整性，以避免生产者和消费者之间的竞争和冲突，这是需要改善的地方。

## 四、问题与建议

### 问题：

1. 编译错误或警告：需要仔细阅读编译器的错误或警告信息，并根据信息对代码进行修改。
2. 文件系统损坏：在进行文件系统实验时，可能会出现文件系统损坏的情况。建议在操作之前备份重要文件，以防止文件丢失。
3. 硬件或虚拟机配置问题：实验需要在特定的硬件或虚拟机配置下运行，如果配置不正确，则可能导致实验无法正常进行。

### 建议：

1. 仔细阅读实验指导书，理解实验要求和操作步骤。
2. 详细记录实验过程和结果，便于后期分析和调试。
3. 遇到问题时，先仔细查看和分析错误信息，然后针对性地进行调试和修改。
4. 在实验完成后，可以通过查看系统日志等方式对实验结果进行验证，以确保实验的正确性。
5. 对实验过程中遇到的问题汇总归纳，获得一般性的的解决方法和经验。