4 进程同步与互斥—生产者⁄消费者问题

**一．实验目的**

1. 理解 POSIX 线程机制；

2. 解决生产者-消费者问题。

**二．实验内容**

1. 建立 POSIX 线程并使用线程执行耗时操作

2. 使用 POSIX 条件变量解决解决生产者-消费者问题

1. **实验过程**

实验代码附在附录中。

1. 建立 POSIX 线程并使用线程执行耗时操作

调用 pthread\_create() 函数实现线程的创建，函数原型被定义在 pthread.h 中。

程序建立一个线程后通过线程同步使主线程等待子线程的执行完成，子线程执行完成后两者一起退出。其流程图如下：

运行结果：

$ ./mythread\_posix1 10000000000

I am main function, I am creating the threads!

Thread 1 has been created!

I am main function , I am waiting the threads finished!

I'm thread 1

thread1 :Is Main function waiting for me ?

Theread 1 has exited!

counter = 1410065408

2. 使用 POSIX 条件变量解决解决生产者-消费者问题

**实验中使用了 Mutex 互斥锁实现生产者和消费者线程之间的同步和互斥。由于线程之间是共享一个资源区域，所以他们可以共同维护同一个锁，使用 pthread\_mutex\_lock() 和 pthread\_mutex\_unlock() 来加锁和解锁。**

**程序的运行流程图如下：**

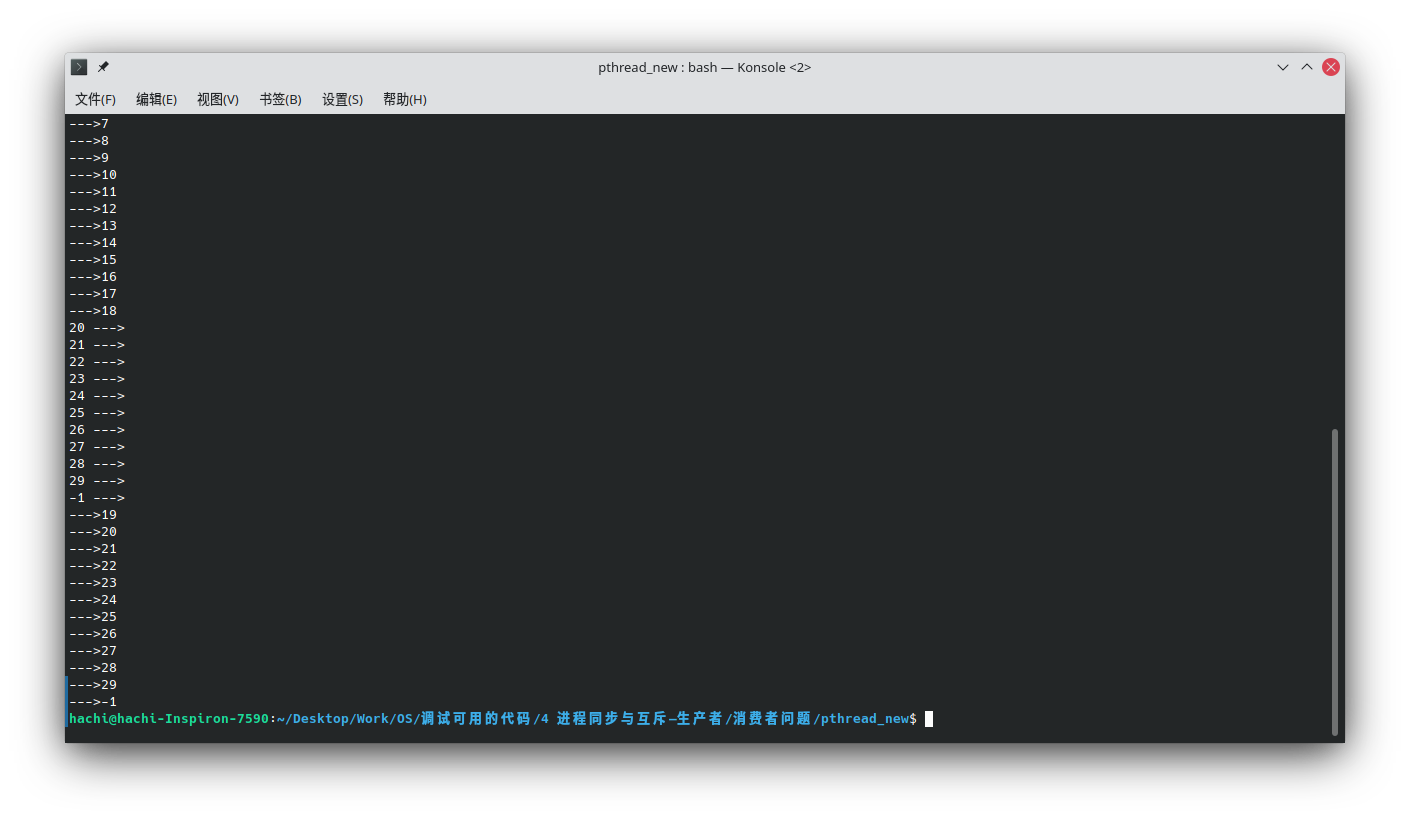
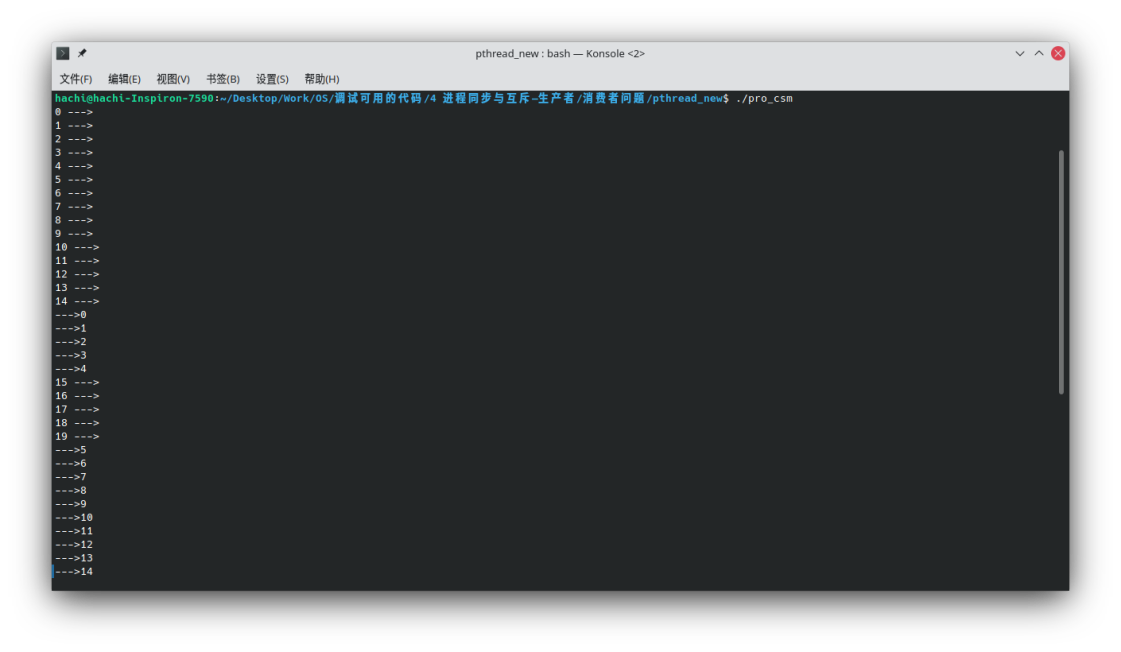
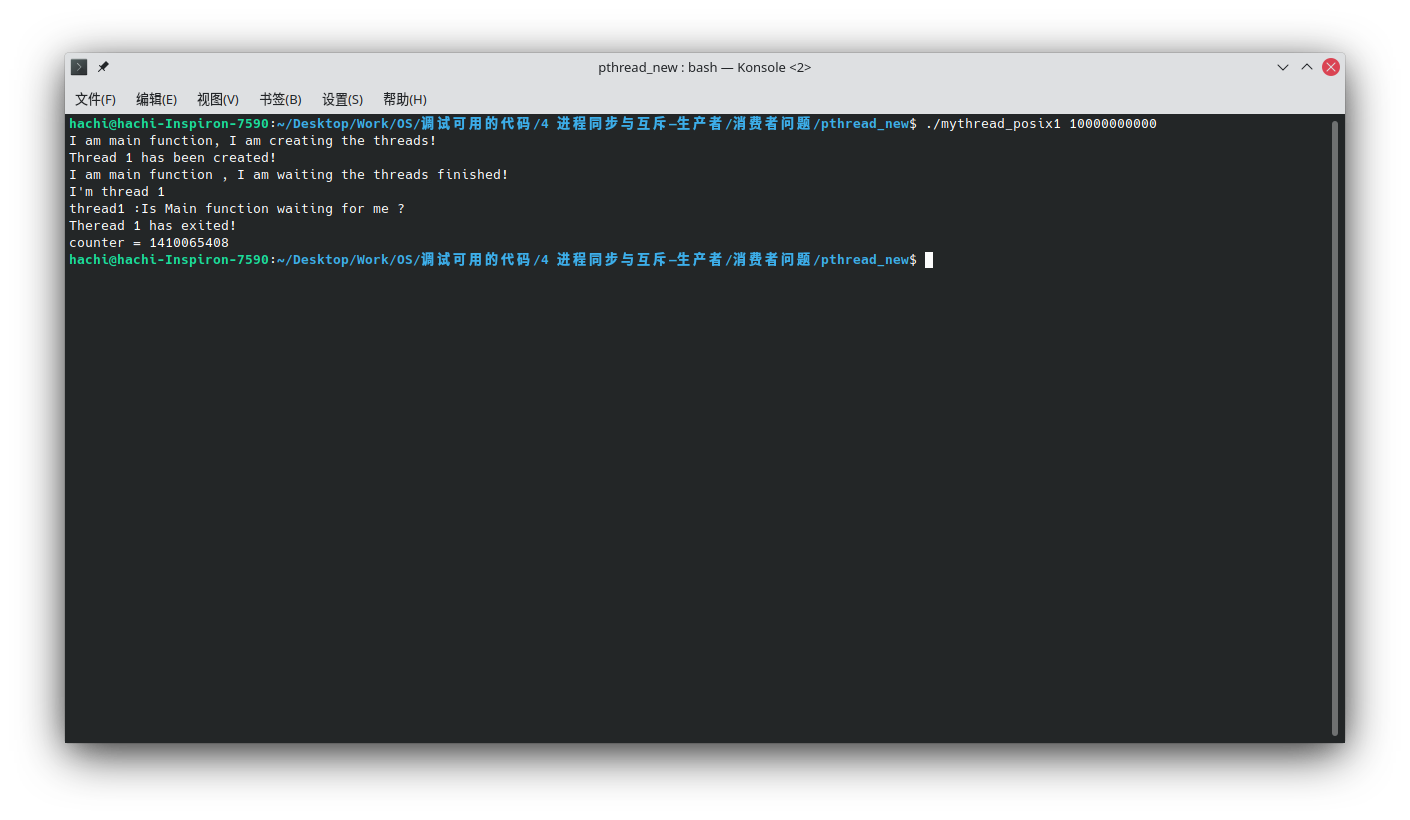
**四．实验结果及问题分析**

**1. 遇到的问题**

**在解决生产者和消费者问题时，在消费者线程中，最初我使用了类似于 if(buffer.writepos == buffer.readpos) 的语句，且成功完成了实验。但是仔细阅读了实验指导书后发现这样的写法是错误的，会产生虚假唤醒的问题，可能导致程序运行结果不可预测，改成使用 while 才是正确的。**

1. **实验结果**

**两个实验内容的运行结果截图如下：**



可以看到他们所呈现的实验效果符合预期。

**五．实验小结**

**在本次实验中，我们学习了 POSIX 线程的创建和使用，并使用 Mutex 互斥锁实现线程之间的同步和互斥，最终解决了生产者-消费者问题和此类问题中经常发生的虚假唤醒问题。在写出可能发生虚假唤醒的代码后，该段代码并没有出现运行结果异常的情况，可见在多线程并发的编程中，问题往往是难以发现的。**

1. **附录**
2. 建立 POSIX 线程并使用线程执行耗时操作

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/time.h>

#include <string.h>

pthread\_t thread;

int counter=0;

int upper;

// 线程函数，功能是对counter数值进行累加；

void \*my\_thread()

{

int i, temp;

printf ("I'm thread 1\n");

for (i = 0; i < upper; i++)

{

temp = counter;

temp += 1;

counter = temp;

}

printf("thread1 :Is Main function waiting for me ?\n");

pthread\_exit(NULL);

}

void thread\_create()

{

int temp;

memset(&thread, 0, sizeof(thread));

if((temp = pthread\_create(&thread, NULL, my\_thread, NULL)) != 0) //创建线程

printf("Creating thread 1 has failed!\n");

else

printf("Thread 1 has been created! \n");

}

void thread\_wait(void)

{

if(thread !=0)

{

pthread\_join(thread,NULL);//等待线程退出

printf("Theread 1 has exited! \n");

}

}

int main(int argc, char \*argv[])

{

if (argc != 2)

{

fprintf(stderr,"usage: mythrd-posix1 integer\_value\n");

return -1;

}

if (atoi(argv[1]) < 0)

{

fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));

return -1;

}

upper = atoi(argv[1]);

printf("I am main function, I am creating the threads! \n");

thread\_create();

printf("I am main function , I am waiting the threads finished! \n");

thread\_wait();

printf("counter = %d\n",counter);

return 0;

}

2. 使用 POSIX 条件变量解决解决生产者-消费者问题

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define BUFFER\_SIZE 16 // 缓冲区数量

#define PRO\_NO 30 // PRODUCING NO

#define OVER ( - 1) //生产结束标志

#define PSLEEP 10000 // 生产者随机睡眠时间

#define CSLEEP 10000 // 消费者随机睡眠时间

#define PPNO 2 // 生产者数量

#define CPNO 2 // 消费者数量

pthread\_mutex\_t lock; /\* 互斥体lock 用于对缓冲区的互斥操作 \*/

pthread\_cond\_t notempty; /\* 缓冲区非空的条件变量 \*/

pthread\_cond\_t notfull; /\* 缓冲区未满的条件变量 \*/

struct prodcons

{// 缓冲区相关数据结构

int buf[BUFFER\_SIZE]; /\* 实际数据存放的数组\*/

int readpos, writepos; /\* 读写指针\*/

};

struct prodcons buffer;

/\* 初始化缓冲区结构 \*/

void init(struct prodcons \*b)

{

b->readpos = 0;

b->writepos = 0;

}

/\* 测试:生产者线程将0 到 PRO\_NO的整数送入缓冲区,消费者线

程从缓冲区中获取整数,两者都打印信息\*/

void \*producer(void \*data)

{

int n;

for (n = 0; n <= PRO\_NO; n++)

{

pthread\_mutex\_lock(&lock);

/\* 等待缓冲区未满,应该用while判断，因为有可能发送虚假唤醒：期待的条件尚不成立的唤醒。\*/

while ((buffer.writepos + 1) % BUFFER\_SIZE == buffer.readpos)

//if ((buffer.writepos + 1) % BUFFER\_SIZE == buffer.readpos) //会产生虚假唤醒

{

pthread\_cond\_wait(&notfull, &lock);

}

/\* 写数据,并移动指针 \*/

if (n < PRO\_NO)

{

buffer.buf[buffer.writepos] = n;

printf("%d --->\n", n);

usleep(PSLEEP);

}

else

{

buffer.buf[buffer.writepos] = OVER;

printf("%d --->\n", OVER);

}

buffer.writepos++;

if (buffer.writepos >= BUFFER\_SIZE)

buffer.writepos = 0;

/\* 设置缓冲区非空的条件变量\*/

pthread\_cond\_signal(&notempty);

pthread\_mutex\_unlock(&lock);

}

return NULL;

}

void \*consumer(void \*data)

{

int d;

while (1)

{

pthread\_mutex\_lock(&lock);

/\* 等待缓冲区非空,应该用while判断，因为有可能发送虚假唤醒：期待的条件尚不成立的唤醒。\*/

while(buffer.writepos == buffer.readpos)

//if(buffer.writepos == buffer.readpos) //会产生虚假唤醒

{

pthread\_cond\_wait(&notempty, &lock);

}

/\* 读数据,移动读指针\*/

d = buffer.buf[buffer.readpos];

//usleep(CSLEEP);

buffer.readpos++;

if (buffer.readpos >= BUFFER\_SIZE)

buffer.readpos = 0;

/\* 设置缓冲区未满的条件变量\*/

pthread\_cond\_signal(&notfull);

pthread\_mutex\_unlock(&lock);

printf("--->%d \n", d);

if (d == OVER)

break;

}

return NULL;

}

int main(void)

{

pthread\_t th\_c, th\_p;

void \*retval;

int i;

init(&buffer);

pthread\_mutex\_init(&lock, NULL);

pthread\_cond\_init(&notempty, NULL);

pthread\_cond\_init(&notfull, NULL);

/\* 创建生产者和消费者线程\*/

pthread\_create(&th\_c, NULL, producer, 0);

pthread\_create(&th\_p, NULL, consumer, 0);

/\* 等待两个线程结束\*/

pthread\_join(th\_c, &retval);

pthread\_join(th\_p, &retval);

pthread\_mutex\_destroy(&lock);

pthread\_cond\_destroy(&notempty);

pthread\_cond\_destroy(&notfull);

return 0;

}

5 Linux内存管理

**一．实验目的**

1. 了解 Linux 物理内存管理方法；

2. 了解 IA32 架构中的逻辑地址映射机制，包括分段和分页；

3. 了解 Linux 的分段机制及其特色；

4. 理解并掌握 Linux 从线程地址到物理地址的映射机制；

5. 理解 Linux 进程虚拟地址空间映射机制；

6. 理解 Linux 虚拟内存区域 vma 的管理方法。

**二．实验内容**

1. 获取 IA32 段寄存器中的描述符

2. 获取 IA32 中的全局段表信息

3. 显示进程的虚拟地址空间分布信息

4. 获取一个进程的虚拟存储区域信息

1. **实验过程**

每个实验内容对应的代码附在附录中。

1. 获取 IA32 段寄存器中的描述符

编译运行 segmentselector 下的代码，并查看 /proc/segselector 的内容，可以看到其中的内容。由于在 5.6+ 的内核中， proc 的文件操作符已经不由 file\_operations 来定义，所以需要改用 proc\_ops 来定义。

程序定义内核模块的入口函数是 showgdtmod\_init() 出口函数为 showgdtmod\_exit() 。在 showgdtmod\_init() 中打开指定的 proc 文件并分别将 cs 寄存器、 es 寄存器、 gs 寄存器中的内容通过内嵌汇编分别转存到对应的变量，然后 seg\_printf 打印到 proc 文件。

1. 获取 IA32 中的全局段表信息

程序首先需要获得 gdtr 寄存器中的信息，给出全局段表的地址。然后由 gdtr 的低 16 位得到 GDT 的限长，高 48 位得到全局段表的基地址和物理地址。最后通过 kmap 和 kunmap 的合作循环打印全局段表的信息。

1. 显示进程的虚拟地址空间分布信息

程序首先获得当前进程的 task\_struct 指针，也就是当前进程的进程控制块。在 pcb 中存储了进程的虚拟内存地址空间分布信息，该内容在 task\_struct 中为 mm\_struct 结构体类型的 mm 成员。获取到 mm 后向 proc 文件打印其内容即可。

mm\_test 的功能是打开 /proc/mm ，读取并打印其中的信息。附录中只给出 mm.c 即内核模块的内容。

4. 获取一个进程的虚拟存储区域信息

程序首先获取当前进程 pcb 中存储的虚拟内存地址信息并打印到 proc 文件中，然后循环打印所有内存段的起始和结束地址及其权限信息。

附录只给出打印内容的 my\_seq\_show() 函数的代码。

**四．实验结果及问题分析**

**1. 遇到的问题**

由于实验代码本身是根据 IA32 设计的，运行在 3.+ Linux 内核上，内嵌了在 32bit Linux 系统下可以使用的代码；但是实际实验中我们使用的 amd64 架构的操作系统，使用的 Linux 内核版本也已经是 5.+ ，所以会出现代码无法编译通过的问题。我在实验过程中修改了部分汇编代码，使其在 amd64 下也可以正常运行；并通过宏使其在 5.6+ 的平台上以 proc\_ops 替代过时的 file\_operations ，最终成功完成了实验。

**2．实验结果**

各个实验所得的结果如下

获取 IA32 段寄存器中的描述符

$ cat /proc/segselector  
CS=0010  DS=0000    
ES=0000  FS=0000    
GS=0000  SS=0018

$ sudo dmesg

[ 4537.633218] <1>

Installing 'segselector' module

[ 5269.672425] <1>Removing 'segselector' module

[ 5404.328561] <1>

获取 IA32 中的全局段表信息

$ cat /proc/myshowgdt    
GDT segment descriptors' number is 16   
gdt\_virt\_address=FFFFFE00001A9000 gdt\_phys\_address=FFFFFDFF401A9000

$ sudo dmesg

[ 5404.328561] <1>

Installing 'myshowgdt' module

[ 5404.328578] <1>ram\_size is 18446739765521809408

[ 6099.485500] <1>Removing 'myshowgdt' module

显示进程的虚拟地址空间分布信息

$ ./mm\_test

This is parent pid 7489

Info from the Memory Management structure for task 'mm\_test' (pid=7489)

pgd=FFFF8DE3C8F34000 mmap=FFFF8DE3C4519790 map\_count=27 mm\_users=1 mm\_count=1

task\_size=0x7FFFFFFFF000

start\_code=56222A3D7000 end\_code=56222A3D74C5 len\_code=000004C5

start\_data=56222A3D9D60 end\_data=56222A3DA010 len\_data=000002B0

start\_brk=56222AE9D000 brk=56222AEBE000

arg\_start=7FFFC0A8DF77 arg\_end=7FFFC0A8DF81

env\_start=7FFFC0A8DF81 env\_end=7FFFC0A8EFEE

start\_stack=7FFFC0A8CCC0 down\_to=7FFFC0A6ACC0 <--- stack grows downward

This is child pid 7490

Info from the Memory Management structure for task 'mm\_test' (pid=7490)

pgd=FFFF8DE3C8E4C000 mmap=FFFF8DE3C4518680 map\_count=27 mm\_users=1 mm\_count=1

task\_size=0x7FFFFFFFF000

start\_code=56222A3D7000 end\_code=56222A3D74C5 len\_code=000004C5

start\_data=56222A3D9D60 end\_data=56222A3DA010 len\_data=000002B0

start\_brk=56222AE9D000 brk=56222AEBE000

arg\_start=7FFFC0A8DF77 arg\_end=7FFFC0A8DF81

env\_start=7FFFC0A8DF81 env\_end=7FFFC0A8EFEE

start\_stack=7FFFC0A8CCC0 down\_to=7FFFC0A6ACC0 <--- stack grows downward

$ sudo dmesg

[ 6363.733121] <1>

Installing 'mm' module

[ 6367.468354] seq\_file: buggy .next function my\_seq\_next [mm] did not update position index

[ 6367.468529] seq\_file: buggy .next function my\_seq\_next [mm] did not update position index

[ 6371.923753] <1>Removing 'mm' module

从实验结果可以看到进程和其线程共用同一段内存空间。

获取一个进程的虚拟存储区域信息

$ ./vma\_test

This is parent pid 8347

Info from the Memory Management structure for task 'vma' (pid=8347)

pgd=FFFF8DE33A6B2000 mmap=FFFF8DE2D661E5B0 map\_count=27 mm\_users=1 mm\_count=1

start\_code=55D24A378000 end\_code=55D24A378545 len\_code=00000545

start\_data=55D24A37AD50 end\_data=55D24A37B010 len\_data=000002C0

start\_brk=55D24A928000 brk=55D24A949000

arg\_start=7FFE54F8DF92 arg\_end=7FFE54F8DF98

env\_start=7FFE54F8DF98 env\_end=7FFE54F8EFF2

start\_stack=7FFE54F8D040 down\_to=7FFE54F6B040 <--- stack grows downward

List of the Virtual Memory Areas for task 'vma' (pid=8347)

1 vm\_start=55D24A377000 vm\_end=55D24A378000 vm\_len=00001000 r--p

2 vm\_start=55D24A378000 vm\_end=55D24A379000 vm\_len=00001000 r-xp

3 vm\_start=55D24A379000 vm\_end=55D24A37A000 vm\_len=00001000 r--p

4 vm\_start=55D24A37A000 vm\_end=55D24A37B000 vm\_len=00001000 r--p

5 vm\_start=55D24A37B000 vm\_end=55D24A37C000 vm\_len=00001000 rw-p

6 vm\_start=55D24A928000 vm\_end=55D24A949000 vm\_len=00021000 rw-p

7 vm\_start=7F80428F8000 vm\_end=7F80428FB000 vm\_len=00003000 rw-p

8 vm\_start=7F80428FB000 vm\_end=7F8042921000 vm\_len=00026000 r--p

9 vm\_start=7F8042921000 vm\_end=7F8042A8C000 vm\_len=0016B000 r-xp

10 vm\_start=7F8042A8C000 vm\_end=7F8042AD8000 vm\_len=0004C000 r--p

11 vm\_start=7F8042AD8000 vm\_end=7F8042ADB000 vm\_len=00003000 r--p

12 vm\_start=7F8042ADB000 vm\_end=7F8042ADE000 vm\_len=00003000 rw-p

13 vm\_start=7F8042ADE000 vm\_end=7F8042AE7000 vm\_len=00009000 rw-p

14 vm\_start=7F8042AE7000 vm\_end=7F8042AEE000 vm\_len=00007000 r--p

15 vm\_start=7F8042AEE000 vm\_end=7F8042AFE000 vm\_len=00010000 r-xp

16 vm\_start=7F8042AFE000 vm\_end=7F8042B03000 vm\_len=00005000 r--p

17 vm\_start=7F8042B03000 vm\_end=7F8042B04000 vm\_len=00001000 r--p

18 vm\_start=7F8042B04000 vm\_end=7F8042B05000 vm\_len=00001000 rw-p

19 vm\_start=7F8042B05000 vm\_end=7F8042B0B000 vm\_len=00006000 rw-p

20 vm\_start=7F8042B26000 vm\_end=7F8042B27000 vm\_len=00001000 r--p

21 vm\_start=7F8042B27000 vm\_end=7F8042B4E000 vm\_len=00027000 r-xp

22 vm\_start=7F8042B4E000 vm\_end=7F8042B58000 vm\_len=0000A000 r--p

23 vm\_start=7F8042B58000 vm\_end=7F8042B5A000 vm\_len=00002000 r--p

24 vm\_start=7F8042B5A000 vm\_end=7F8042B5C000 vm\_len=00002000 rw-p

25 vm\_start=7FFE54F6D000 vm\_end=7FFE54F8F000 vm\_len=00022000 rw-p

26 vm\_start=7FFE54FB4000 vm\_end=7FFE54FB8000 vm\_len=00004000 r--p

This is child pid 8348

27 vm\_start=7FFE54FB8000 vm\_end=7FFE54FBA000 vm\_len=00002000 r-xp

CR3=17A6B2001 mm->pgd=000000004d01454f mm->map\_count=27

Info from the Memory Management structure for task 'vma' (pid=8348)

pgd=FFFF8DE3B5D50000 mmap=FFFF8DE3C452A340 map\_count=27 mm\_users=1 mm\_count=1

start\_code=55D24A378000 end\_code=55D24A378545 len\_code=00000545

start\_data=55D24A37AD50 end\_data=55D24A37B010 len\_data=000002C0

start\_brk=55D24A928000 brk=55D24A949000

arg\_start=7FFE54F8DF92 arg\_end=7FFE54F8DF98

env\_start=7FFE54F8DF98 env\_end=7FFE54F8EFF2

start\_stack=7FFE54F8D040 down\_to=7FFE54F6B040 <--- stack grows downward

List of the Virtual Memory Areas for task 'vma' (pid=8348)

1 vm\_start=55D24A377000 vm\_end=55D24A378000 vm\_len=00001000 r--p

2 vm\_start=55D24A378000 vm\_end=55D24A379000 vm\_len=00001000 r-xp

3 vm\_start=55D24A379000 vm\_end=55D24A37A000 vm\_len=00001000 r--p

4 vm\_start=55D24A37A000 vm\_end=55D24A37B000 vm\_len=00001000 r--p

5 vm\_start=55D24A37B000 vm\_end=55D24A37C000 vm\_len=00001000 rw-p

6 vm\_start=55D24A928000 vm\_end=55D24A949000 vm\_len=00021000 rw-p

7 vm\_start=7F80428F8000 vm\_end=7F80428FB000 vm\_len=00003000 rw-p

8 vm\_start=7F80428FB000 vm\_end=7F8042921000 vm\_len=00026000 r--p

9 vm\_start=7F8042921000 vm\_end=7F8042A8C000 vm\_len=0016B000 r-xp

10 vm\_start=7F8042A8C000 vm\_end=7F8042AD8000 vm\_len=0004C000 r--p

11 vm\_start=7F8042AD8000 vm\_end=7F8042ADB000 vm\_len=00003000 r--p

12 vm\_start=7F8042ADB000 vm\_end=7F8042ADE000 vm\_len=00003000 rw-p

13 vm\_start=7F8042ADE000 vm\_end=7F8042AE7000 vm\_len=00009000 rw-p

14 vm\_start=7F8042AE7000 vm\_end=7F8042AEE000 vm\_len=00007000 r--p

15 vm\_start=7F8042AEE000 vm\_end=7F8042AFE000 vm\_len=00010000 r-xp

16 vm\_start=7F8042AFE000 vm\_end=7F8042B03000 vm\_len=00005000 r--p

17 vm\_start=7F8042B03000 vm\_end=7F8042B04000 vm\_len=00001000 r--p

18 vm\_start=7F8042B04000 vm\_end=7F8042B05000 vm\_len=00001000 rw-p

19 vm\_start=7F8042B05000 vm\_end=7F8042B0B000 vm\_len=00006000 rw-p

20 vm\_start=7F8042B26000 vm\_end=7F8042B27000 vm\_len=00001000 r--p

21 vm\_start=7F8042B27000 vm\_end=7F8042B4E000 vm\_len=00027000 r-xp

22 vm\_start=7F8042B4E000 vm\_end=7F8042B58000 vm\_len=0000A000 r--p

23 vm\_start=7F8042B58000 vm\_end=7F8042B5A000 vm\_len=00002000 r--p

24 vm\_start=7F8042B5A000 vm\_end=7F8042B5C000 vm\_len=00002000 rw-p

25 vm\_start=7FFE54F6D000 vm\_end=7FFE54F8F000 vm\_len=00022000 rw-p

26 vm\_start=7FFE54FB4000 vm\_end=7FFE54FB8000 vm\_len=00004000 r--p

27 vm\_start=7FFE54FB8000 vm\_end=7FFE54FBA000 vm\_len=00002000 r-xp

CR3=1F5D50003 mm->pgd=0000000060a6ffe3 mm->map\_count=27

$ sudo dmesg

[ 7047.637694] <1>

Installing 'vma' module

[ 7076.785271] seq\_file: buggy .next function my\_seq\_next [vma] did not update position index

[ 7076.785465] seq\_file: buggy .next function my\_seq\_next [vma] did not update position index

[ 7136.415209] <1>Removing 'vma' module

**五．实验小结**

在本节实验中，我们通过在内核态读取和查看 task\_struct 和 mm\_struct 的内容，获取进程的段表信息，实际了解了 IA32 的段表以及虚拟地址转换成物理地址的过程。实验中部分代码已经过时，但只要了解基础的 Linux 内核编程知识以及 GNU C 知识，就可以实现对程序的修改并完成实验。经过本次实验，我们对操作系统内部的地址转换有了更深的理解。

**六．附录**

1. 获取 IA32 段寄存器中的描述符

#include <linux/module.h> // for init\_module()

#include <linux/proc\_fs.h> // for create\_proc\_read\_entry()

#include <linux/seq\_file.h>

char modname[] = "segselector";

// using ‘global variables’ simplifies the inline assembly language

short \_cs, \_ds, \_es, \_fs, \_gs, \_ss; // global variables

static int my\_seqshow(struct seq\_file \*m, void \*v)

{

//asm( " mov %%cr3, %%eax \n mov %%eax, %0 " : "=m" (\_cr3) :: "ax" );

asm(" mov %cs, \_cs \n mov %ds, \_ds ");

asm(" mov %es, \_es \n mov %fs, \_fs ");

asm(" mov %gs, \_gs \n mov %ss, \_ss ");

seq\_printf(m, "CS=%04X DS=%04X \n", \_cs, \_ds );

seq\_printf(m, "ES=%04X FS=%04X \n", \_es, \_fs );

seq\_printf(m, "GS=%04X SS=%04X \n", \_gs, \_ss );

return 0; //!! must be 0, or will show nothing T.T

}

static int my\_open(struct inode \*inode, struct file \*file)

{

return single\_open(file, my\_seqshow, NULL);

}

// replace struct file\_operations with struct proc\_ops for kernel version 5.6 or later

#include <linux/version.h>

#if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5,6,0)

static const struct proc\_ops my\_proc = {

.proc\_open = my\_open,

.proc\_read = seq\_read,

.proc\_lseek = seq\_lseek,

.proc\_release = seq\_release

};

#else

static const struct file\_operations my\_proc =

{

.owner = THIS\_MODULE,

.open = my\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = single\_release,

};

#endif

static int \_\_init showgdtmod\_init( void )

{

struct proc\_dir\_entry\* my\_proc\_entry;

printk( "<1>\nInstalling \'%s\' module\n", modname );

my\_proc\_entry = proc\_create(modname, 0x644, NULL, &my\_proc);

return 0; //SUCCESS

}

static void \_\_exit showgdtmod\_exit(void )

{

remove\_proc\_entry( modname, NULL );

printk( "<1>Removing \'%s\' module\n", modname );

}

module\_init( showgdtmod\_init );

module\_exit( showgdtmod\_exit );

MODULE\_LICENSE("GPL");

1. 获取 IA32 中的全局段表信息

#include <linux/module.h> // for init\_module()

#include <linux/proc\_fs.h> // for create\_proc\_info\_entry()

#include <linux/seq\_file.h>

#include <linux/highmem.h> // for kmap(), kunmap()

#include <asm/uaccess.h> // for copy\_to\_user()

#define START\_KERNEL\_map 0xC0000000

char modname[] = "myshowgdt";

loff\_t ram\_size;

unsigned short gdtr[3];

unsigned long gdt\_virt\_address;

unsigned long gdt\_phys\_address;

// seq\_operations -> show

static int my\_seqshow(struct seq\_file \*m, void \*v)

{

int n\_elts, i;

int frame\_number, frame\_indent;

struct page \*pp;

unsigned long long \*from;

unsigned long long descriptor;

// use inline assembly language to get GDTR register-image

asm(" sgdt gdtr ");

// extract GDT virtual-address from GDTR register-image

gdt\_virt\_address = \*(unsigned long\*)(gdtr+1);

// compute GDT physical-address using subtraction

gdt\_phys\_address = gdt\_virt\_address - START\_KERNEL\_map;

// extract GDT segment-limit and compute descriptor count

n\_elts = (1 + gdtr[0])/8;

seq\_printf(m, "GDT segment descriptors' number is %d\n", n\_elts );

// report the GDT virtual and physical memory-addresses

seq\_printf(m, "gdt\_virt\_address=%08lX ", gdt\_virt\_address );

seq\_printf(m, "gdt\_phys\_address=%08lX ", gdt\_phys\_address );

seq\_printf(m, "\n" );

if ( gdt\_phys\_address >= ram\_size ) return 0;

// determine which physical page to temporarily map

// and how far into that page to begin reading from

frame\_number = gdt\_phys\_address / PAGE\_SIZE;

frame\_indent = gdt\_phys\_address % PAGE\_SIZE;

pp = pfn\_to\_page( frame\_number);

from = (unsigned long long\*)(kmap( pp ) + frame\_indent);

for (i = 0; i < n\_elts; i++)

{

if ( ( i % 4 ) == 0 ) seq\_printf(m, "\n %04X: ", i\*8 );

descriptor = \*from;

seq\_printf(m, "%016llX ", descriptor );

from++;

}

seq\_printf(m, "\n" );

kunmap( pp );

return 0; //!! must be 0, or will show nothing T.T

}

static int my\_open(struct inode \*inode, struct file \*file)

{

return single\_open(file, my\_seqshow, NULL);

}

// replace struct file\_operations with struct proc\_ops for kernel version 5.6 or later

#include <linux/version.h>

#if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5,6,0)

static const struct proc\_ops my\_proc = {

.proc\_open = my\_open,

.proc\_read = seq\_read,

.proc\_lseek = seq\_lseek,

.proc\_release = seq\_release

};

#else

static const struct file\_operations my\_proc =

{

.owner = THIS\_MODULE,

.open = my\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = single\_release,

};

#endif

static int \_\_init my\_init( void )

{

struct proc\_dir\_entry\* my\_proc\_entry;

printk( "<1>\nInstalling \'%s\' module\n", modname );

my\_proc\_entry = proc\_create(modname, 0x644, NULL, &my\_proc);

ram\_size = (loff\_t)totalram\_pages << PAGE\_SHIFT;

printk( "<1>ram\_size is %llu\n",ram\_size );

return 0; //SUCCESS

}

static void \_\_exit my\_exit(void )

{

remove\_proc\_entry( modname, NULL );

printk( "<1>Removing \'%s\' module\n", modname );

}

module\_init( my\_init );

module\_exit( my\_exit );

MODULE\_LICENSE("GPL");

1. 显示进程的虚拟地址空间分布信息

#include <linux/module.h> // for init\_module()

#include <linux/proc\_fs.h> // for create\_proc\_info\_entry()

#include <linux/sched.h> // for 'struct task\_struct'

#include <linux/seq\_file.h> // for sequence files

#include <linux/mm.h> // for 'struct mm\_struct'

char modname[] = "mm";

static void \* my\_seq\_start(struct seq\_file \*m, loff\_t \*pos)

{

if (0 == \*pos)

{

++\*pos;

return (void \*)1;

}

return NULL;

}

static void \* my\_seq\_next(struct seq\_file \*m, void \*p, loff\_t \*pos)

{

// do nothing

return NULL;

}

static void my\_seq\_stop(struct seq\_file \*m, void \*p)

{

//// do nothing

}

static int my\_seq\_show(struct seq\_file \*m, void \*p)

{

struct task\_struct \*tsk = current;

struct mm\_struct \*mm = tsk->mm;

unsigned long stack\_size = (mm->stack\_vm << PAGE\_SHIFT);

unsigned long down\_to = mm->start\_stack - stack\_size;

seq\_printf( m, "Info from the Memory Management structure " );

seq\_printf( m, "for task \'%s\' ", tsk->comm );

seq\_printf( m, "(pid=%d) \n", tsk->pid );

seq\_printf( m, " pgd=%08lX ", (unsigned long)mm->pgd );

seq\_printf( m, "mmap=%08lX ", (unsigned long)mm->mmap );

seq\_printf( m, "map\_count=%d ", mm->map\_count );

seq\_printf( m, "mm\_users=%d ", mm->mm\_users.counter );

seq\_printf( m, "mm\_count=%d ", mm->mm\_count.counter );

seq\_printf( m, "\n" );

seq\_printf( m, " task\_size=0x%08lX ", mm->task\_size );

seq\_printf( m, "\n" );

seq\_printf( m, " start\_code=%08lX ", mm->start\_code );

seq\_printf( m, " end\_code=%08lX\n", mm->end\_code );

seq\_printf( m, " start\_data=%08lX ", mm->start\_data );

seq\_printf( m, " end\_data=%08lX\n", mm->end\_data );

seq\_printf( m, " start\_brk=%08lX ", mm->start\_brk );

seq\_printf( m, " brk=%08lX\n", mm->brk );

seq\_printf( m, " arg\_start=%08lX ", mm->arg\_start );

seq\_printf( m, " arg\_end=%08lX\n", mm->arg\_end );

seq\_printf( m, " env\_start=%08lX ", mm->env\_start );

seq\_printf( m, " env\_end=%08lX\n", mm->env\_end );

seq\_printf( m, " start\_stack=%08lX ", mm->start\_stack );

seq\_printf( m, " down\_to=%08lX ", down\_to );

seq\_printf( m, " <--- stack grows downward \n" );

//seq\_printf( m, "\n" );

return 0;

}

static struct seq\_operations my\_seq\_fops = {

.start = my\_seq\_start,

.next = my\_seq\_next,

.stop = my\_seq\_stop,

.show = my\_seq\_show

};

static int my\_open(struct inode \*inode, struct file \*file)

{

return seq\_open(file, &my\_seq\_fops);

}

// replace struct file\_operations with struct proc\_ops for kernel version 5.6 or later

#include <linux/version.h>

#if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5,6,0)

static const struct proc\_ops my\_proc = {

.proc\_open = my\_open,

.proc\_read = seq\_read,

.proc\_lseek = seq\_lseek,

.proc\_release = seq\_release

};

#else

static const struct file\_operations my\_proc =

{

.owner = THIS\_MODULE,

.open = my\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = seq\_release

};

#endif

static int \_\_init my\_init( void )

{

struct proc\_dir\_entry\* my\_proc\_entry;

printk( "<1>\nInstalling \'%s\' module\n", modname );

my\_proc\_entry = proc\_create(modname, 0, NULL, &my\_proc);

if (NULL == my\_proc\_entry)

{

return -ENOMEM;

}

return 0; //SUCCESS

}

static void \_\_exit my\_exit(void )

{

remove\_proc\_entry( modname, NULL );

printk( "<1>Removing \'%s\' module\n", modname );

}

module\_init( my\_init );

module\_exit( my\_exit );

MODULE\_LICENSE("GPL");

4. 获取一个进程的虚拟存储区域信息

static int my\_seq\_show(struct seq\_file \*m, void \*p)

{

struct task\_struct \*tsk = current;

struct vm\_area\_struct \*vma;

unsigned long ptdb;

int i=0;

struct mm\_struct \*mm = tsk->mm;

unsigned long stack\_size = (mm->stack\_vm << PAGE\_SHIFT);

unsigned long down\_to = mm->start\_stack - stack\_size;

seq\_printf( m, "\nInfo from the Memory Management structure " );

seq\_printf( m, "for task \'%s\' ", tsk->comm );

seq\_printf( m, "(pid=%d) \n", tsk->pid );

seq\_printf( m, " pgd=%08lX ", (unsigned long)mm->pgd );

seq\_printf( m, "mmap=%08lX ", (unsigned long)mm->mmap );

seq\_printf( m, "map\_count=%d ", mm->map\_count );

seq\_printf( m, "mm\_users=%d ", mm->mm\_users.counter );

seq\_printf( m, "mm\_count=%d ", mm->mm\_count.counter );

seq\_printf( m, "\n" );

seq\_printf( m, " start\_code=%08lX ", mm->start\_code );

seq\_printf( m, " end\_code=%08lX\n", mm->end\_code );

seq\_printf( m, " start\_data=%08lX ", mm->start\_data );

seq\_printf( m, " end\_data=%08lX\n", mm->end\_data );

seq\_printf( m, " start\_brk=%08lX ", mm->start\_brk );

seq\_printf( m, " brk=%08lX\n", mm->brk );

seq\_printf( m, " arg\_start=%08lX ", mm->arg\_start );

seq\_printf( m, " arg\_end=%08lX\n", mm->arg\_end );

seq\_printf( m, " env\_start=%08lX ", mm->env\_start );

seq\_printf( m, " env\_end=%08lX\n", mm->env\_end );

seq\_printf( m, " start\_stack=%08lX ", mm->start\_stack );

seq\_printf( m, " down\_to=%08lX ", down\_to );

seq\_printf( m, " <--- stack grows downward \n" );

seq\_printf( m, "List of the Virtual Memory Areas " );

seq\_printf( m, "for task \'%s\' ", tsk->comm );

seq\_printf( m, "(pid=%d)", tsk->pid );

// loop to traverse the list of the task's vm\_area\_structs

vma = tsk->mm->mmap;

while ( vma ) {

char ch;

seq\_printf( m, "\n%3d ", ++i );

seq\_printf( m, " vm\_start=%08lX ", vma->vm\_start );

seq\_printf( m, " vm\_end=%08lX ", vma->vm\_end );

ch = ( vma->vm\_flags & VM\_READ ) ? 'r' : '-';

seq\_printf( m, "%c", ch );

ch = ( vma->vm\_flags & VM\_WRITE ) ? 'w' : '-';

seq\_printf( m, "%c", ch );

ch = ( vma->vm\_flags & VM\_EXEC ) ? 'x' : '-';

seq\_printf( m, "%c", ch );

ch = ( vma->vm\_flags & VM\_SHARED ) ? 's' : 'p';

seq\_printf( m, "%c", ch );

vma = vma->vm\_next;

}

//seq\_printf( m, "\n" );

// display additional information about tsk->mm

//asm(" movl %%cr3, %%ecx \n movl %%ecx, %0 " : "=m" (ptdb) );

// now in 64bit machine

asm(" movq %%cr3, %%rcx \n movq %%rcx, %0 " : "=m" (ptdb) );

seq\_printf( m, "\nCR3=%08lX ", ptdb );

seq\_printf( m, " mm->pgd=%p ", tsk->mm->pgd );

seq\_printf( m, " mm->map\_count=%d ", tsk->mm->map\_count );

seq\_printf( m, "\n" );

return 0;

}

6 proc文件系统及查看进程信息

**一．实验目的**

1. 理解 proc 伪文件系统的基本概念和功能，掌握常见操作命令；

2. 了解 Linux 进程控制块 task\_struct ，并理解其重要成员变量的意义；

3. 理解基于 seq\_file 机制的 proc 伪文件操作机制；

4. 熟练掌握生成 proc 伪文件的 Linux 内核模块代码实现方法。

**二．实验内容**

1. 创建显示系统进程信息的 proc 模块

2. 显示 Linux 进程控制块中的更多信息

1. **实验过程**

代码附在附录中。

1. 创建显示系统进程信息的 proc 模块

编译 vexp2 下的 tasklist.c 文件，产生内核模块并插入到系统内核中，可以看到 /proc 目录中创建了一个 tasklist 文件，查看文件内容可以看到程序在里面写入了当前系统中所有进程的信息和进程对应的命令行。由于系统进程较多，实验结果中只展示了文件内容的前 10 行。

程序通过 proc\_ops 建立一个 proc 文件（file\_operations 在 Linux5.6 被取消）。在 proc\_ops 结构体中，指定了 proc\_open 为 my\_open 函数，而 my\_open 函数指定了获取 proc 文件内容的 my\_seq\_start 、 my\_seq\_next 、 my\_seq\_stop 和 my\_seq\_show 函数，通过三者配合遍历系统所有进程并获取其进程命令行。

2. 显示 Linux 进程控制块中的更多信息

在进程控制块中，除了进程对应的命令行，还有其他进程信息如进程号、调度策略、优先级、运行时间等。我们只需要修改 my\_seq\_show 函数即可实现实验所需的功能。所以附录中只放上该函数的代码，实验结果同样只放上前 10 个进程的信息。另外 real\_start\_time 成员在 Linux5.6+ 被取消，取而代之的是 boottime 。

**四．实验结果及问题分析**

**1. 遇到的问题**

在我的实验环境中，由于 file\_operation 和 task\_struct 中的 real\_start\_time 成员已经被取消，取而代之的是 proc\_ops 和 boottime ，我无法成功编译源代码，但是经过少量的修改依然可以完成实验。

**2. 实验结果**

创建显示系统进程信息的 proc 模块

$ cat /proc/tasklist

#0 0 0 swapper/0

#1 1 1 systemd

#2 2 1 kthreadd

#3 3 1026 rcu\_gp

#4 4 1026 rcu\_par\_gp

#5 6 1026 kworker/0:0H

#6 9 1026 mm\_percpu\_wq

#7 10 1 rcu\_tasks\_rude\_

#8 11 1 rcu\_tasks\_trace

#9 12 1 ksoftirqd/0

#10 13 1026 rcu\_sched

显示 Linux 进程控制块中的更多信息

$ cat /proc/tasklist

#0 0 0 swapper/0 120 120 120 0 0 0 4577741 0 560000000 0

#1 1 1 systemd 120 120 120 0 0 4285 282 788000000 980000000 75524089

#2 2 1 kthreadd 120 120 120 0 0 534 1 0 32000000 75524089

#3 3 1026 rcu\_gp 100 100 100 0 0 2 0 0 0 75524089

#4 4 1026 rcu\_par\_gp 100 100 100 0 0 2 0 0 0 75524089

#5 6 1026 kworker/0:0H 100 100 100 0 0 4 0 0 0 75524089

#6 9 1026 mm\_percpu\_wq 100 100 100 0 0 2 0 0 0 75524089

#7 10 1 rcu\_tasks\_rude\_ 120 120 120 0 0 2 0 0 0 75524089

#8 11 1 rcu\_tasks\_trace 120 120 120 0 0 2 0 0 0 75524089

#9 12 1 ksoftirqd/0 120 120 120 0 0 16485 5 0 0 75524089

#10 13 1026 rcu\_sched 120 120 120 0 0 1196167 55 0 88000000 75524089

**五．实验小结**

在本次实验中，我们第一次认识了 Linux 中 /proc 这一虚拟文件系统。并实际建立了 proc 文件并在内核态下获取了系统中全部的进程信息，即进程控制块 pcb 。尽管内核版本的不同导致需要对代码进行少量的修改，但我依然成功完成了实验。通过实际查看 pcb 中的信息，我们更深入地了解了进程控制块对进程调度的重要作用。

**六．附录**

1. 创建显示系统进程信息的 proc 模块

#include <linux/module.h> // for init\_module()

#include <linux/proc\_fs.h> // for create\_proc\_info\_entry()

#include <linux/sched.h> // for init\_task

#include <linux/seq\_file.h> // for sequence files

#include <linux/slab.h> // for kzalloc, kfree

char modname[] = "tasklist";

struct task\_struct \*task;

int taskcounts=0; // 'global' so value will be retained

static void \* my\_seq\_start(struct seq\_file \*m, loff\_t \*pos)

{

///printk(KERN\_INFO"Invoke start\n"); //可以输出调试信息

if ( \*pos == 0 ) // 表示遍历开始

{

task = &init\_task; //遍历开始的记录地址

return &task; //返回一个非零值表示开始遍历

}

else //遍历过程中

{

if (task == &init\_task ) //重新回到初始地址，退出

return NULL;

return (void\*)pos ;//否则返回一个非零值

}

}

static int my\_seq\_show(struct seq\_file \*m, void \*v)

{//获取进程的相关信息

//printk(KERN\_INFO"Invoke show\n");

seq\_printf( m, "#%-3d ", taskcounts++ );

seq\_printf( m, "%5d ", task->pid );

seq\_printf( m, "%lu ", task->state );

seq\_printf( m, "%-15s ", task->comm );

seq\_puts( m, "\n" );

return 0;

}

static void \* my\_seq\_next(struct seq\_file \*m, void \*v, loff\_t \*pos)

{

//printk(KERN\_INFO"Invoke next\n");

(\*pos)++;

task= next\_task(task); //指向下一个进程

return NULL;

}

static void my\_seq\_stop(struct seq\_file \*m, void \*v)

{

//printk(KERN\_INFO"Invoke stop\n");

// do nothing

}

static struct seq\_operations my\_seq\_fops = {//序列文件记录操作函数集合

.start = my\_seq\_start,

.next = my\_seq\_next,

.stop = my\_seq\_stop,

.show = my\_seq\_show

};

static int my\_open(struct inode \*inode, struct file \*file)

{

return seq\_open(file, &my\_seq\_fops); //打开序列文件并关联my\_seq\_fops

}

// replace struct file\_operations with struct proc\_ops for kernel version 5.6 or later

#include <linux/version.h>

#if LINUX\_VERSION\_CODE >= KERNEL\_VERSION(5,6,0)

static const struct proc\_ops my\_proc = {

.proc\_open = my\_open,

.proc\_read = seq\_read,

.proc\_lseek = seq\_lseek,

.proc\_release = seq\_release

};

#else

static const struct file\_operations my\_proc =

{ //proc文件操作函数集合

.owner = THIS\_MODULE,

.open = my\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = seq\_release

};

#endif

int \_\_init my\_init( void )

{

struct proc\_dir\_entry\* my\_proc\_entry;

printk( "<1>\nInstalling \'%s\' module\n", modname );

my\_proc\_entry = proc\_create(modname, 0x644, NULL, &my\_proc);//生成proc文件

if (NULL == my\_proc\_entry)

{

return -ENOMEM;

}

return 0; //SUCCESS

}

void \_\_exit my\_exit( void )

{

remove\_proc\_entry( modname, NULL );//删除proc文件

printk( "<1>Removing \'%s\' module\n", modname );

}

module\_init(my\_init);

module\_exit(my\_exit);

MODULE\_LICENSE("GPL");

2. 显示 Linux 进程控制块中的更多信息

static int my\_seq\_show(struct seq\_file \*m, void \*v)

{//获取进程的相关信息

//printk(KERN\_INFO"Invoke show\n");

seq\_printf( m, "#%-3d ", taskcounts++ );

seq\_printf( m, "%5d ", task->pid );

seq\_printf( m, "%lu ", task->state );

seq\_printf( m, "%-15s ", task->comm );

seq\_printf( m, "%-5d ", task->prio); //优先级

seq\_printf( m, "%-5d ", task->static\_prio);

seq\_printf( m, "%-5d ", task->normal\_prio);

seq\_printf( m, "%-5d ", task->rt\_priority);

seq\_printf( m, "%u ", task->policy);//调度策略

seq\_printf( m, "%lu ", task->nvcsw);//上下文切换

seq\_printf( m, "%lu ", task->nivcsw);

seq\_printf( m, "%llu ", task->utime);//运行时间

seq\_printf( m, "%llu ", task->stime);

// real\_start\_time (represented as struct timespec until 3.17, replaced with u64 nanosecond timestamps) now start\_boottime in kernel 5.6+

seq\_printf( m, "%llu ", task->start\_boottime);

seq\_puts( m, "\n" );

return 0;

}

7 Linux驱动程序

**一．实验目的**

1. 掌握 Linux 设备驱动程序基本原理；

2. 掌握 Linux 字符设备驱动程序的创建流程；

3. 掌握 Linux 字符设备驱动的常见函数；

4. 理解 Linux 设备驱动的内存映射机制。

**二．实验内容**

1. 实现读取 CMOS 实时时钟信息的驱动程序

2. 实现一个基于内存映射的设备驱动程序

**三．实验过程**

代码附在附录中。

1. 实现读取 CMOS 实时时钟信息的驱动程序

实验中我们使用了手动创建节点的方式，通过 mknod 实现对节点的创建，并改变其节点权限为所有人可读可写。在该实验中，我们依然可以使用 linux/fs.h 中定义的 file\_operations 结构体，这个结构体并没有被弃用。实验完成后需要手动删除 /dev/cmos 节点。

由于 Linux 的特性， cmos 时间是国际标准时，最后查看 cmos\_test 的执行结果时，需要 +8 得到当前的中国上海时间即 CST 。

2. 实现一个基于内存映射的设备驱动程序

Linux 将各种无法分类的设备称为杂项设备，其主设备号均为 10 ，次设备号不同。定义杂项设备的结构体 struct miscdevice 定义在 linux/miscdevice.h 中，注册后 Linux 会调用其中的 file\_operations 结构体中注册的文件操作函数接口进行操作。

程序在 file\_operations 中定义了内存映射通过 my\_map 函数实现，在这个函数中，首先开辟了一段虚拟内存空间，然后将一段自定义的信息打印其中。当 miscdev\_maptest 调用这个设备时 Linux 内核就会将其返回给调用的程序。

附录中只呈现了本实验中 my\_map() 函数部分。

**四．实验结果及问题分析**

**1. 遇到的问题**

在进行实验的第一个内容时，并没有注意到设备节点的权限问题，导致测试程序一直无法正常打开设备，导致实验无法继续。之后偶然使用 sudo 提权运行测试程序成功，才注意到了设备节点默认并不允许非超级用户操作， chmod 改变其权限后程序运行正常。

**2．实验结果**

实现读取 CMOS 实时时钟信息的驱动程序

$ sudo insmod cmos.ko

$ sudo mknod /dev/cmos

$ sudo mknod /dev/cmos c 70 0

$ sudo chmod a+rw /dev/cmos

$ ./cmos\_test

CMOS Real-Time Clock/Calendar: 07:07:11 on SAT, 01 JAN 2022

CMOS Real-Time Clock/Calendar: 07:07:11 on SAT, 01 JAN 2006

CMOS Real-Time Clock/Calendar: 07:07:11 on SAT, 01 JAN 200F

$ sudo rm /dev/cmos

实现一个基于内存映射的设备驱动程序

$ sudo insmod miscdev\_map.ko

$ sudo insmod miscdev\_map.ko

$ sudo chmod a+rw /dev/mymap

$ gcc miscdev\_maptest.c -o miscdev\_maptest

$ ./miscdev\_maptest

I am the devive mymap, this is the test output by weilinfox

$ sudo rmmod miscdev\_map

**五．实验小结**

本此实验我们认识了两种 Linux 驱动，了解了 Linux 驱动的原理和实现方法，并实际实现了对字符设备 CMOS 的读取和自定义了一个基于内存映射的杂项设备驱动，向虚拟内存区域写入了信息并成功从设备节点中读取。通过这次实验，我们对 Linux 设备驱动具有了一些基本的认识。尽管驱动程序的实现非常复杂，但是 Linux 驱动程序的结构是非常简洁的，通过将其抽象成文件并注册其操作函数，在内核态下我们可以实现一些平时用户态不可能实现的功能。

**六．附录**

1. 实现读取 CMOS 实时时钟信息的驱动程序

#include <linux/module.h> // for printk()

#include <linux/fs.h> // for register\_chrdev()

#include <linux/cdev.h>

#include <asm/uaccess.h> // for put\_user(), get\_user()

#include <asm/io.h> // for inb(), outb()

char devname[] = "cmos"; // name for the device's file

int my\_major = 70; // major ID-number for driver

int cmos\_size = 128; // total bytes of cmos memory

int write\_max = 9; // largest 'writable' address

ssize\_t my\_read( struct file \*file, char \*buf, size\_t len, loff\_t \*pos )

{

unsigned char data;

if ( \*pos >= cmos\_size ) return 0;

outb( \*pos, 0x70 ); data = inb( 0x71 );

if ( put\_user( data, buf ) ) return -EFAULT;

\*pos += 1;

return 1;

}

ssize\_t my\_write( struct file \*file, const char \*buf, size\_t len, loff\_t \*pos )

{

unsigned char data;

if ( \*pos >= cmos\_size ) return 0;

if ( \*pos > write\_max ) return -EPERM;

if ( get\_user( data, buf ) ) return -EFAULT;

outb( \*pos, 0x70 ); outb( data, 0x71 );

\*pos += 1;

return 1;

}

loff\_t my\_llseek( struct file \*file, loff\_t pos, int whence )

{

loff\_t newpos = -1;

switch ( whence )

{

case 0: newpos = pos; break; // SEEK\_SET

case 1: newpos = file->f\_pos + pos; break; // SEEK\_CUR

case 2: newpos = cmos\_size + pos; break; // SEEK\_END

}

if (( newpos < 0 )||( newpos > cmos\_size )) return -EINVAL;

file->f\_pos = newpos;

return newpos;

}

struct file\_operations my\_fops = {

owner: THIS\_MODULE,

llseek: my\_llseek,

write: my\_write,

read: my\_read,

};

struct cdev cmos\_dev;

static void cmos\_setup\_cdev(void)

{

int err, devno;

devno = MKDEV(my\_major, 0);

cdev\_init(&cmos\_dev, &my\_fops);

cmos\_dev.owner = THIS\_MODULE;

//cmos\_dev.ops = &my\_fops;

err = cdev\_add(&cmos\_dev, devno, 1);

if (err)

printk("Error %d\n", err);

else

printk("have finish add\n");

}

static int \_\_init my\_init( void )

{

int ret; dev\_t devno;

printk( "<1>\nInstalling \'%s\' module ", devname );

printk( "(major=%d) \n", my\_major );

//return register\_chrdev( my\_major, devname, &my\_fops );

devno = MKDEV(my\_major, 0);

ret = register\_chrdev\_region(devno, 1, devname );

cmos\_setup\_cdev();

printk ("<1>\n \'%s\'has been Initialized \n",devname);

return 0;

}

static void \_\_exit my\_exit(void )

{

//unregister\_chrdev( my\_major, devname );

cdev\_del(&cmos\_dev);

unregister\_chrdev\_region(MKDEV(my\_major, 0), 1);

printk( "<1>Removing \'%s\' module\n", devname );

}

module\_init( my\_init );

module\_exit( my\_exit );

MODULE\_LICENSE("GPL");

2. 实现一个基于内存映射的设备驱动程序

#define DEVICE\_INFO "I am the devive mymap, this is the test output by weilinfox"

static int my\_map(struct file \*filp, struct vm\_area\_struct \*vma)

{

unsigned long page;

unsigned long start = (unsigned long)vma->vm\_start;

unsigned long size = (unsigned long)(vma->vm\_end - vma->vm\_start);

vma->vm\_flags |= VM\_IO;

vma->vm\_flags |= (VM\_DONTEXPAND | VM\_DONTDUMP); //VM\_RESERVED

//得到物理地址

page = virt\_to\_phys(buffer);

//将用户空间的一个vma虚拟内存区映射到以page开始的一段连续物理页面上

if(remap\_pfn\_range(vma,start,page>>PAGE\_SHIFT,size,PAGE\_SHARED))//第三个参数是页帧号，由物理地址右移PAGE\_SHIFT得到

return -1;

//往该内存写数据

sprintf(buffer, "%s", DEVICE\_INFO);

return 0;

}