# 操作系统实验报告

题 目 基于30天的验证性和设计实现性实验

课程名称 操作系统

学 院 网络空间安全学院

成 员 张政 曾扬 姚敏 沈如晖 闵庚云 孙伟 孙默 唐天晓

指导老师 杜晓峰

日期：2016.12.28

1. **实验要求**

（1）验证性实验：

必做：《30天自制操作系统》0-31天所有内容。

（2）设计实现性实验：

必做：

* 依据30天实验中使用LDT结构的用户程序版本，设计、实现并展示用户程序中变量的逻辑地址到实际物理地址的转换（只有分段，没有分页）。
* 在操作系统内核设计并实现进程访问共享变量时的竞争条件。
* 在操作系统内核态设计并实现竞争条件的解决方案（两进程、多进程），并进一步实现某个同步场景（生产者消费者、读者作者等），至少一个解决方案，至少一个同步场景。
* 设计并实现从用户态启动内核态竞争条件、解决方案、同步场景（练习系统调用），至少实现一个。

选做：

* 设计并实现用户态的竞争条件、解决方案和同步场景。
* 设计并实现新的内存分配算法，并在系统中验证。

1. **实验环境**

实验环境主要使用《30天自制操作系统》中前期已开发的操作系统，在这个基础上进行实验功能的相应开发。不同的模块由不同的人进行开发，所以使用了QEMU和BOCHS两个模拟器。

1. **设计思路**

**我们将实验分为了三个部分：****①地址转换 ②竞争条件的实现与解决方案 ③同步场景的实现** **④新型内存分配算法**

**②和③我们设计并实现了基于内核态、以及从用户态启动内核态、直接基于用户态。**

**①地址转换**

依据30天实验中使用LDT结构的用户程序版本，我们设计并实现了用户程序中变量的逻辑地址到实际物理地址的分段转换。

**物理地址**：加载到内存地址寄存器中的地址，内存单元的真正地址。在前端总线上传输的内存地址都是物理内存地址，编号从0开始一直到可用物理内存的最高端。这些数字被北桥(Nortbridge chip)映射到实际的内存条上。物理地址是明确的、最终用在总线上的编号，不必转换，不必分页，也没有特权级检查(no translation, no paging, no privilege checks)。

**逻辑地址**：CPU所生成的地址。逻辑地址是内部和编程使用的、并不唯一。在进行[C语言](http://lib.csdn.net/base/c" \o "C语言知识库" \t "http://blog.csdn.net/u013063153/article/details/_blank)指针编程中，可以读取指针变量本身值(&操作)，实际上这个值就是逻辑地址。是相对于当前进程数据段的地址（偏移地址），不和绝对物理地址相干。

分段的内存管理方式可以支持这种思路。一段main函数代码，里面包含Sqrt函数的调用。这段代码运行时，操作系统应该分配内存给：符号表(编译时使用)，栈(存放局部变量与函数参数值)，Sqrt代码段，主函数代码段等。这样，我们就可以方便地指出："函数sqrt内存模块的第五条指令"，来定位一个元素。逻辑地址空间由一组段组成。每个段都有名字和长度。地址指定了段名称和段内偏移。因此用户通过两个量来指定地址：段名称和偏移。段是编号的，通过段号而非段名称来引用。因此逻辑地址由有序对构成：

<segment-number,offset>(<段号s, 段内偏移d>)

段偏移d因该在0和段界限之间，如果合法，那么就与基地址相加而得到所需字节在物理内存中的地址。因此段表是一组基地址和界限寄存器对。

**②竞争条件的实现与解决方案**

在操作系统内核中，我们提供一个函数供用户态调用，该函数创建两个或多个进程，并共享同一个变量，当进程操作同步时，就产生了竞争条件的可能。设计思路如下：

竞争条件的定义是两个或多个进程同时访问同一个变量，且结果与其执行的顺序有关，那么就有可能产生竞争条件。

总结来看一共有三个条件：A.多个进程 B.同一个变量 C.结果与执行顺序有关

针对三个问题逐一解决：

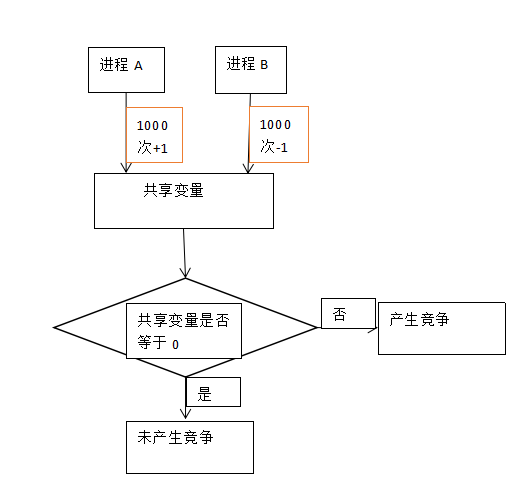
A. 解决进程：系统内核态提供了进程管理的功能，能够很方便的创建进程，回收进程。进程的创建需要对一些寄存器进行初始化，这些代码在第16天的基础上稍加改进即可。

B. 解决共享变量：变量的本质也是内存中的某一个地址上所存放的数据。因此只需要让不同的进程之间能够知道他们共享的变量的地址是多少就能解决问题。为简化问题，我们直接在内存中规定一个地址为进程所共享地址。本次实验中规定为：0x0000f000。

C.能产生冲突。由于对内存数据的访问可以是将其读入到寄存器中，对其进行修改，写回内存中的。如果数据在进行修改之后还未写回内存中，这时被另一个进程所抢占，那么另一个进程对数据的操作很有可能没有效果，最终导致竞争条件的产生。不光是在寄存器数据没有写入内存之间可以产生竞争条件，当数据刚读入寄存器时被另一个进程抢占也可能会发生竞争。

关于如何判断是否发生了竞争条件这一问题，我们对共享变量进行能够预知结果的操作，并将最终的结果打印出来，通过判断打印出来的值与正常处理应得到的值是否相同判断竞争条件是否发生。具体做法为：对共享变量0x0000f000处的值进行固定次数的加1操作，如加1000次。同时另一个进程对共享变量进行1000次的减1操作。将最终的结果打印出来，结果是否为0可以作为竞争条件的判断条件。

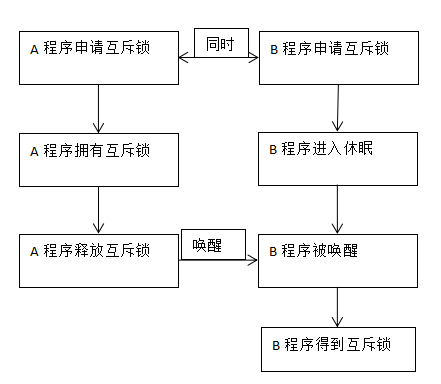
可以由以下图示说明：



当竞争条件产生时，我们计划在内核态通过临界区或同步硬件等的方式来解决竞争条件，并将其和前面的函数合在一起构成一个新的函数，并能让用户程序直接调用——即**一个能应用于多进程的“互斥锁”来解决。**互斥锁的设计思路如下：

按照互斥锁的原理，程序在得到互斥锁的时候才可以执行，并且在A程序拥有互斥锁的时候，需要使用同一个互斥锁的B程序必须等待，直至A程序释放互斥锁后B程序才可进行下一步操作。我们初步的想法是当A程序拥有互斥锁时，令B程序循环去询问需要的互斥锁是否为空。但这样会浪费资源，所以最后采用了A释放互斥锁的时候去唤醒B的形式，而B需要等待互斥锁的时候会休眠。最后，为方便不同的进程去声明同一个互斥锁。我们设定名字相同的互斥锁为同一个互斥锁。

为了方便理解，图示如下。



当有多个程序的时候，我们设计了一个队列存放申请互斥锁的程序ABCDE...先申请的先得到互斥锁，其余在等待的时候都进入休眠状态。

**③同步场景的实现**

在操作系统内核态设计并实现基于多进程竞争条件的解决方案后，我们进一步实现了读者写者的同步场景，将其开放给用户程序调用。

根据读者写者的要求：

一个读者读的时候其他读者都可以读，所有写者都不可以写。

一个写者写的时候其他写者不可以写，所有读者都不可以读。

本次实现了两个读者任务，两个写者任务。

写者A循环十次，每次先申请互斥锁A和互斥锁B，之后对共享变量加一，之后释放互斥锁B和互斥锁A。

写者B循环五次，每次先申请互斥锁A和互斥锁B，之后对共享变量减一，之后释放互斥锁B和互斥锁A。

读者A无限循环，每次先申请互斥锁A，之后读取共享变量显示出来，之后释放互斥锁A。

读者B无限循环，每次先申请互斥锁B，之后读取共享变量显示出来，之后释放互斥锁B。

值得注意的是，在实现的过程中，只有显示问题比较麻烦。因为程序运行太快，不进行处理的话，屏幕上的数字一闪而过，就无法看清读者写者的状态改变。所以此处使用了定时器，每次读者读取数据并显示在屏幕上之后就使用定时器等待1s，每次写者对数据进行修改之后也使用定时器等待1s，来展示数据变化。

**而从用户态启动内核态竞争条件、解决方案、同步场景的设计和实现，我们将内核态的函数封装成了API，能够直接从用户态通过系统调用访问共享变量。**

而关于最终如何判断是否发生了竞争条件并解决这一问题，由于我们在内核态完成的时候直接就进行了封装，所以下文中的实验中我们是通过用户态直接输入命令a来演示竞争条件产生和解决的。

另外，我们设计并实现用户态的竞争条件、解决方案和同步场景。前面已经在内核态实现了多个进程共享同一个变量，为了能让用户态访问到共享的变量，在内核态分配共享变量的基础上把分配函数开放成API，这样就能通过API调用的方式来让不同的进程实现共享变量。

**④新型内存分配算法**

基于30天中第9天已有的内存分配算法，我们实现了新的内存分配算法，分别为最佳分配算法Best-fit和最差分配算法Worst-fit，并在系统中进行了验证。

**最佳适应算法（Best Fit）**：该算法总是把既能满足要求，又是最小的空闲分区分配给进程。为了加速查找，该算法要求将所有的空闲区按其大小排序后，以递增顺序形成一个空白链。这样每次找到的第一个满足要求的空闲区，必然是最优的。孤立地看，该算法似乎是最优的，但事实上并不一定。因为每次分配后剩余的空间一定是最小的，在存储器中将留下许多难以利用的小空闲区。同时每次分配后必须重新排序，这也带来了一定的开销。

特点：每次分配给文件的都是最合适该文件大小的分区。

缺点：内存中留下许多难以利用的小的空闲区。

**最坏适应算法(Worst Fit)**：该算法按大小递减的顺序形成空闲区链，分配时直接从空闲区链的第一个空闲区中分配（不能满足需要则不分配）。很显然，如果第一个空闲分区不能满足，那么再没有空闲分区能满足需要。这种分配方法初看起来不太合理，但它也有很强的直观吸引力：在大空闲区中放入程序后，剩下的空闲区常常也很大，于是还能装下一个较大的新程序。

最坏适应算法与最佳适应算法的排序正好相反，它的队列指针总是指向最大的空闲区，在进行分配时，总是从最大的空闲区开始查寻。

该算法克服了最佳适应算法留下的许多小的碎片的不足，但保留大的空闲区的可能性减小了，而且空闲区回收也和最佳适应算法一样复杂。

特点：给文件分配分区后剩下的空闲区不至于太小，产生碎片的几率最小，对中小型文件分配分区操作有利。

 缺点：使存储器中缺乏大的空闲区，对大型文件的分区分配不利。

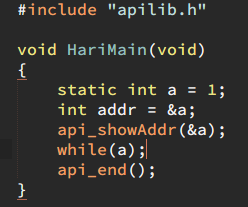
1. **具体实现**

**①地址转换**

## 将逻辑地址转换为实际物理地址

由于这部分的重点不在于编程，所以将数据结构等略过，主要描述具体的实现细节和测试过程。

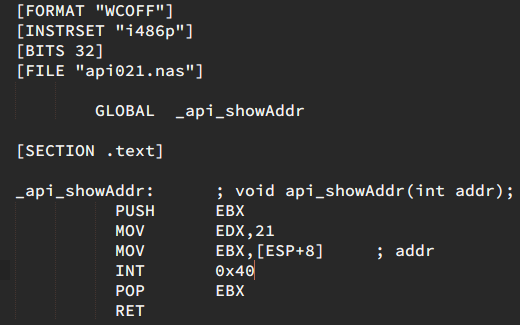
编写如下的测试函数./showAddr/showAddr.c：



其中api\_showAddr的具体实现为(./haribote/bootpack.c)：

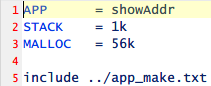


添加api021.nas文件：

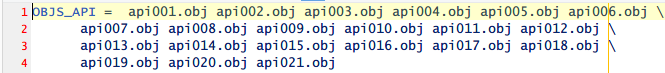


然后修改各Makefile文件:

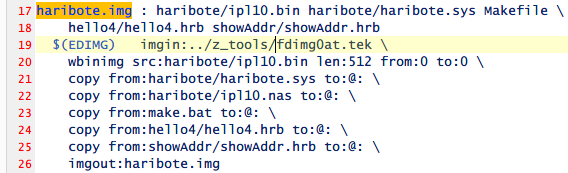
./showAddr/Makefile:

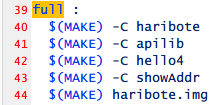


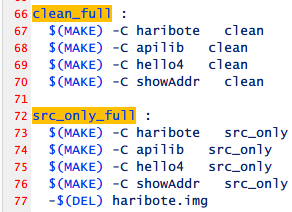
./apilib/Makefile（添加api021.obj）:



./Makefile:

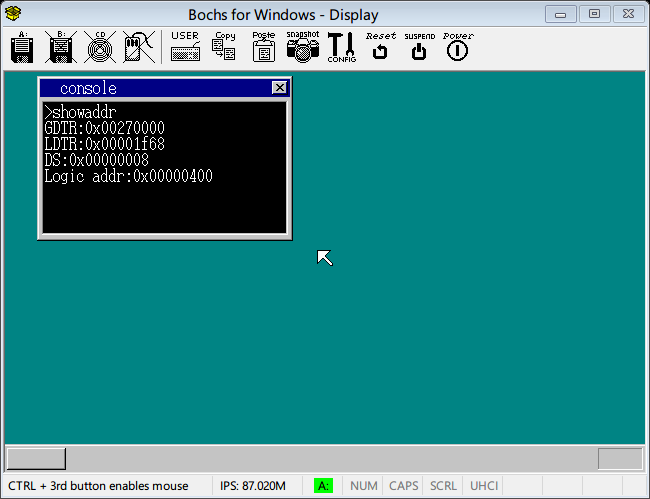




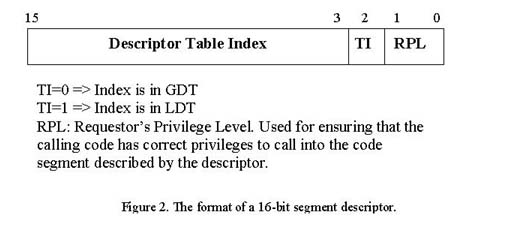


然后运行./ !cons\_nt.bat，命令 “make full”在./目录下生成系统镜像haribote.img。

将haribote.img加载到调试模式下的Bochs，输入“c”继续运行，并在命令行输入“showaddr”来启动测试程序：



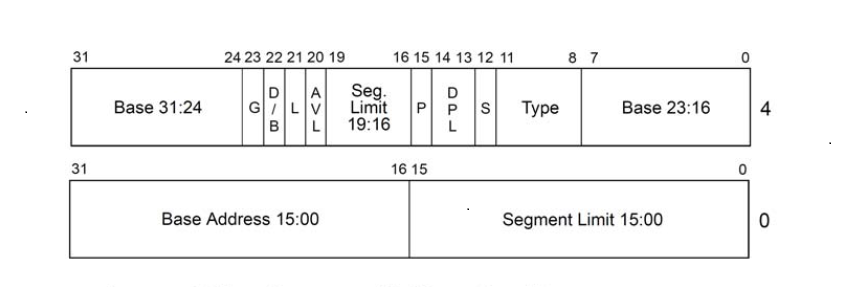
其中GDTR=0x00001f68=0001 1111 1100 1000，则根据如下图的结构可知此时在GDT中的索引为0011 1111 1001 = 1005。



GDT中每一项大小为8个字节，则LDT的地址为LDTR+index\*8 = 0x002700+1005\*8 = 0x00271f68。在Bochs Console窗口点击“Ctrl + c”断下程序，输入“xp /2wx 0x00271f68”查看此处的内容：



此处为SEGMENT\_DESCRIPTOR结构，其内容分布如下：



将0x41ec000f 0x00008204分解，可知LDT的基地址为：0x000441ec。此时ds=0x8（应该为0xC），按照结构1分解，可知其在LDT中的索引为1。由于LDT中每项的大小也为8字节，则保存数据段基地址的SEGMENT\_DESCRIPTOR结构位于0x000441ec + 1\*8 =0x000441f4。查看该处的内容（被保存在task->ldt[1]）：



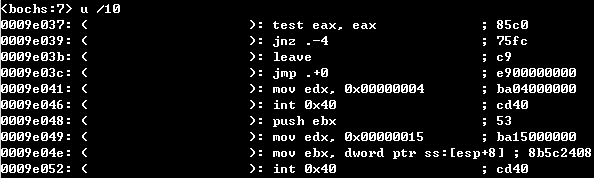
同样将其按照SEGMENT\_DESCRIPTOR结构分解，可知数据段的基地址为0x0043f000（该值同样被保存在task->ds\_base中）。由程序的输出可知，变量的逻辑地址为0x400，则其物理地址为：0x0043f000+0x400=0x0043f400。

查看该处的值：

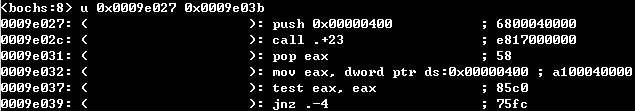


可知此处的值为1。

反汇编此处的代码：



获取该处之前的反汇编代码：



可见程序将a的值保存eax中，故修改0x0043f400不能跳出while循环。作为验证，我们多次修改源程序中a的值，如果0x0043f400处的值一直保持与a一致，可知此处即为a的物理地址。以下为在程序中设置a为不同值时0x0043f400的内容：

a=2：



a=50：



a=100：



可见a的物理地址为0x0043f400。

**②竞争条件的实现与解决**

**由于在内核态实现之后，我们直接把函数包装成了API使之能在用户态进行系统调用。所以以下的测试都是直接基于从用户态调用内核态的。**

竞争条件产生具体代码如下：

/\*两个进程相差的仅仅是对共享变量进行不同的操作\*/

void task\_b\_main(struct SHEET \*sht\_win\_b)

{

char s[12];

int \*position=0x0000f000;//共享变量的地址

int j=0;

for (;;) {

if(j<MAX){

myadd(position);//对共享变量进行操作

++j;

}

if(j==MAX){

sprintf(s,"%11d",\*(position));

putfonts8\_asc\_sht(sht\_win\_b, 24, 28, COL8\_000000, COL8\_C6C6C6, s, 11);//将结果展示出来

//++j;

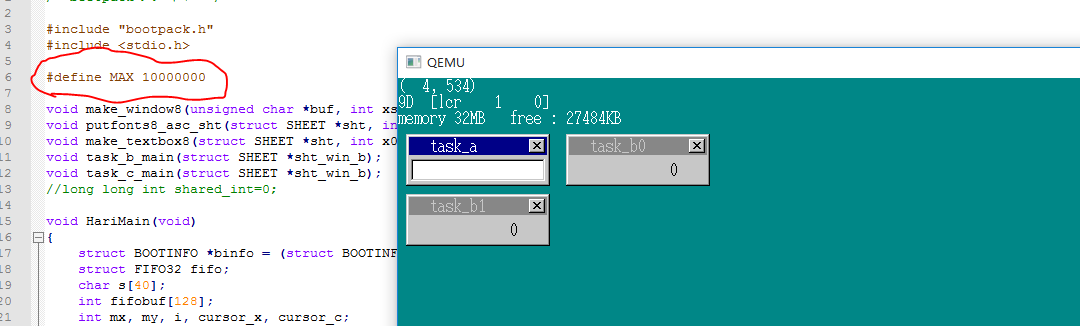
}

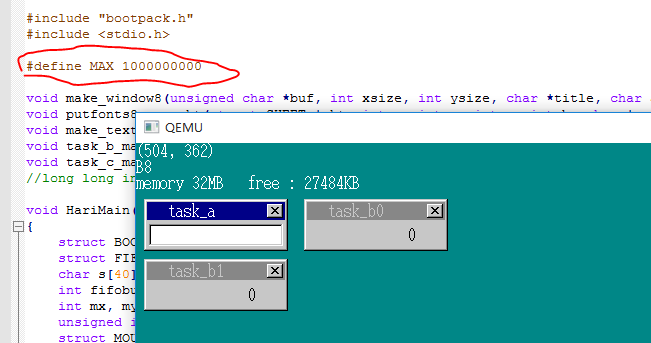
}

}

在HairMain中添加对进程的初始化操作。如果竞争条件产生了，那么执行的结果应该不是0。

然而在测试的时候遇到的问题是:尽管将进行加减法操作的次数设置的非常大，但是仍然非常难产生竞争条件。





分析出现问题的原因：操作系统进行任务切换的时间为0.02s切换依次，每个0.02s之间可以执行上百万条指令。对于每次执行的那么对指令，要在寄存器还没有写入内存时发生抢占，产生的概率很小很小，几乎不会产生。

所以为了产生竞争条件应该增加数据在没有写回内存时被另一个进程抢走的概率。每一条指令执行的的间都很短，不能直接的改变其执行时间，于是将加一或减一操作写成一个函数。函数的内容是将数据读入寄存器、操作寄存器中的数、做无用的循环延时、写回内存。这样程序很容易的在无用循环处时间片被用完，导致数据还没有写回内存，发生竞争条件。

具体的算法如下：

\_myadd: ;void myadd(int \* pointer)

MOV EAX,[ESP+4]

MOV ECX,[EAX]

INC ECX ;数据还未写入寄存器

MOV EBX,0

add\_loop: ;循环达到延时的效果

INC EBX

CMP EBX,0xffff

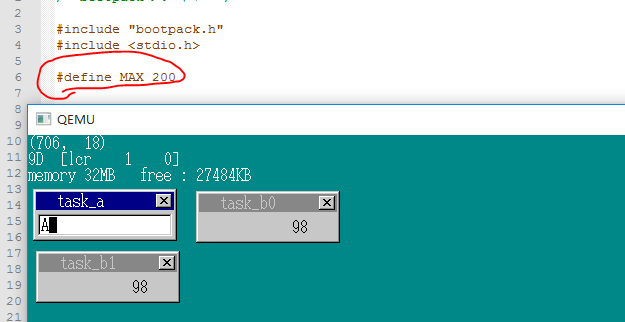
JBE add\_loop

MOV [EAX],ECX ;将数据写入寄存器

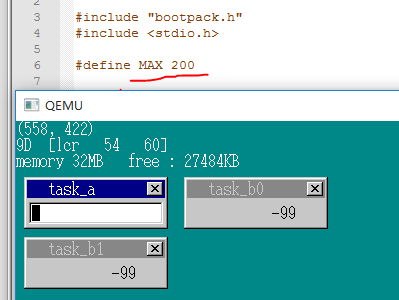
RET

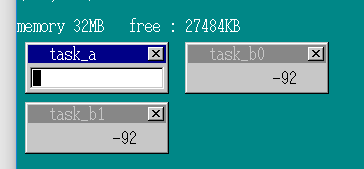
减法的操作与之类似。

即使加减的次数很少也能产生竞争条件：



而且出现的值是随机的：





**（1）在内核态实现竞争条件的解决方案。**

竞争条件已经能很好的产生了，怎样解决竞争条件？采用的方案是：n个进程临界区的算法，也即面包店算法。该算法通过两个数组来解决n个进程的同步问题。而这两个数组是若干进程需要共享的，为此，若还是在内存中找一个地址让进程共享已经不太合适了。于是将分配共享变量的机制进行了规整，采用结构体的形式来完成共享变量的分配。结构体具体如下设计：

struct SHAR{

int flag;//表示变量是否处于共享状态

char name[MAX\_NAME];//共享变量的名称

int shared;//共享的变量

int share\_num; //该变量被多少个进程共享

unsigned char choosing[MAX\_PROCESS];//解决竞争条件的数组

int number[MAX\_PROCESS];

};

//管理共享变量的结构体

struct SHARCTL{

struct SHAR shares0[MAX\_SHARE];

struct SHAR \* shares[MAX\_SHARE];//指向shares0的指?

int using;//共享了多少了个变量

};

在不同的进程中利用shar\_alloc(name)可以获得一个共享变量，不同进程之间通过变量的名字来区分。如果两个进程需要共享一个变量，那么只要将相同的名字作为shar\_alloc的参数，就能完成变量的共享，同时利用struct SHAR中的数组能完成竞争条件的解决。

进程的代码如下：

void task\_b\_main(struct SHEET \*sht\_win\_b,struct TASK \* task)

{

char s[12];

int j=0,k;

struct SHAR \* shar\_b = shar\_alloc("sharedint");//声明共享变量

shar\_b->shared=0;

int \* position=&(shar\_b->shared);//保存共享变量的地址

for (;;) {

if(j<MAX){

//进入区

shar\_b->choosing[0]=1;

shar\_b->number[0]=max(shar\_b->number,2)+1;

shar\_b->choosing[0]=0;

for(k=0;k<2;++k){

while(shar\_b->choosing[k]);

while(shar\_b->number[k]!=0&&(shar\_b->number[k] < shar\_b->number[0]||(shar\_b->number[k]==shar\_b->number[0]&&k<0)));

}

myadd(position);//临界区

shar\_b->number[0]=0;//退出区

++j;

}

if(j==MAX){

sprintf(s,"%11d",(shar\_b->shared));

putfonts8\_asc\_sht(sht\_win\_b, 24, 28, COL8\_000000, COL8\_C6C6C6, s, 11);

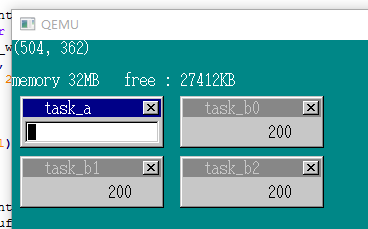
}

}

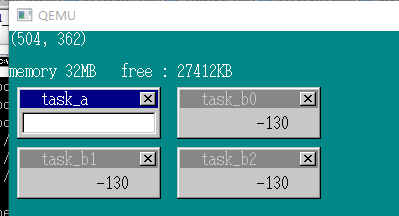
}

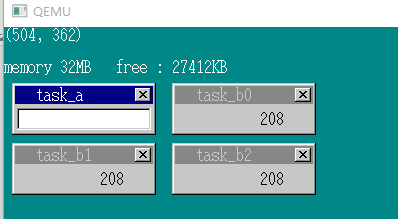
其他几个进程的代码与之类似。

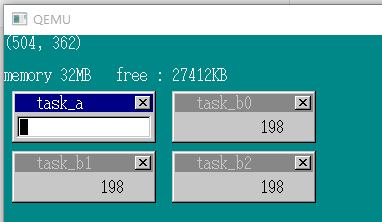
一共有三个进程共享变量sharedint，其中两个进程对其做加法，另外一个做减法，如果没有竞争条件则最终结果应为200.



如果不添加竞争条件解决办法的情况下，结果如下：





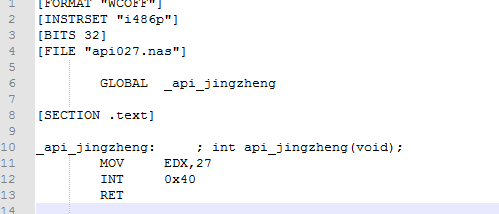


可以看到结果不为200，为一个随机值。

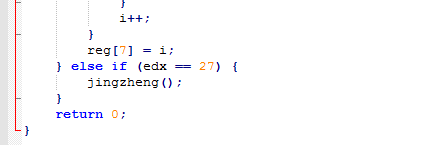
**（2）将产生并解决竞争条件整个过程开放成一个API供用户态调用。**

API调用的原理是，用户态里调用API函数，而函数的处理流程是，将该API的编号放入寄存器中，并产生一个中断，系统判断是多少号中断，如果发现是系统调用的中断，那么就调用相应的中断处理程序，中断处理程序根据寄存器中的值来进行不同的处理。

将api\_jingzheng()规定为第27号API

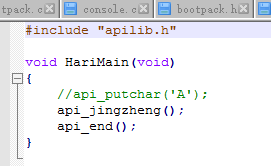


对第27号API的处理如下：



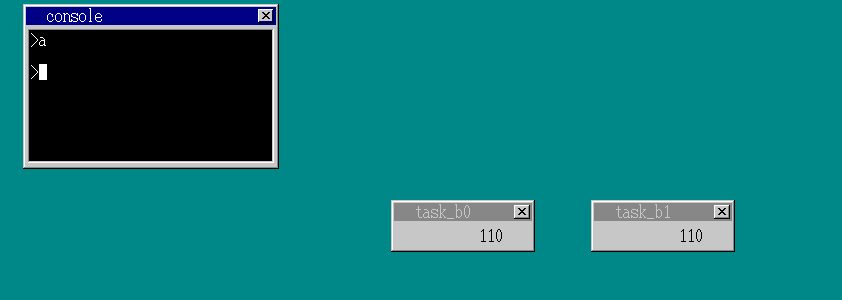
调用jingzheng()函数，而该函数的功能就是创建多个进程，这些进程共享同一个变量，并用面包店算法解决竞争。

调用api\_jingzheng()函数的应用程序a。



未解决竞争条件情况下：





结果不为0，为一个随机值。

解决竞争条件后结果如下：



结果为0.

**（3）竞争条件的解决——互斥锁**

定义的常量：

MAX\_MUTEX 可使用的最多的互斥锁的数量

MAX\_KEEP\_TASK 等待当前互斥锁的最大任务数量

MAX\_NAME互斥锁名字的最大长度

结构体 MUTEX （互斥锁）

|  |  |  |
| --- | --- | --- |
| 成员 | 类型 | 作用 |
| Now | struct \*TASK | 记录拥有当前互斥锁的任务，不被任何任务所有时，设为0 |
| Flag | int | 为1时当前互斥锁以被某个任务声明，为0是未被声明 |
| Head，tail | int | 等待队列头和尾，head队列头，tail-1为队列最后一个元素 |
| Tasks0 | struct | 决定所允许的最大进程等待数量 |

结构体MUTEXCTL

|  |  |  |
| --- | --- | --- |
| 成员 | 类型 | 作用 |
| Num | int | 记录当前被申请的互斥锁量 |
| Mutex | MUTEX | 存放所有可使用的互斥锁 |

变量mutexctl：

类型：struct MUTEXCTL

作用：存放所有可使用的互斥锁

函数mutex\_alloc(const char \*s)：

类型：struct MUTEX\*

参数s：申请使用的互斥锁名

作用：申请使用互斥锁

函数signal(struct MUTEX \*mutex,struct SHEET \*sht\_win\_b, char s[20])

类型void

参数：锁、图层的结构体和要显示的字符串

作用：释放锁，并显示释放信息

函数wait(struct MUTEX \*mutex,struct SHEET \*sht\_win\_b, char s[20])

类型void

参数：锁、图层的结构体和要显示的字符串

作用：等待加锁，并显示等待信息

函数mutex\_release(struct MUTEX \*mutex):

类型：int

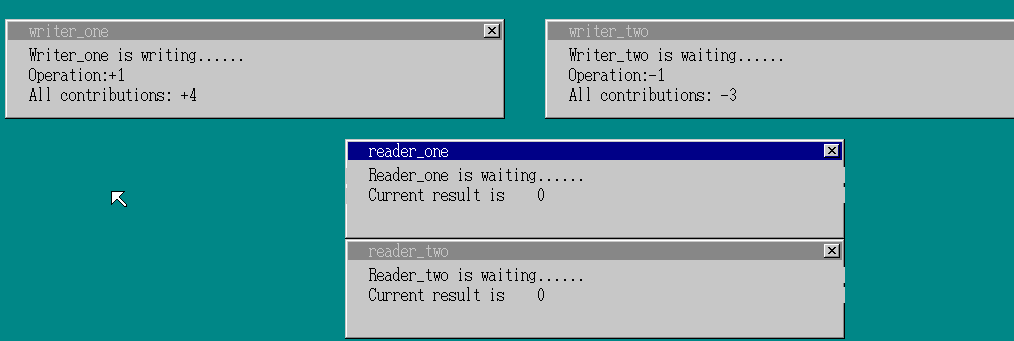
参数mutex：准备释放的互斥锁

作用：释放指定的互斥锁

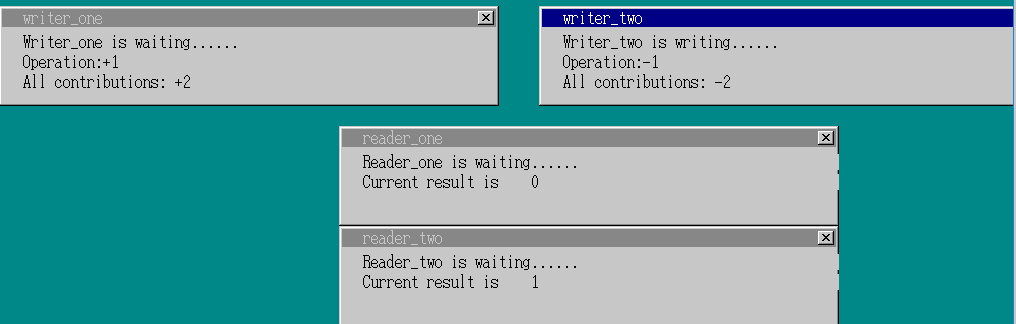
返回值：1代表互斥锁释放成功过，0代表互斥锁释放失败

**（4）同步场景**

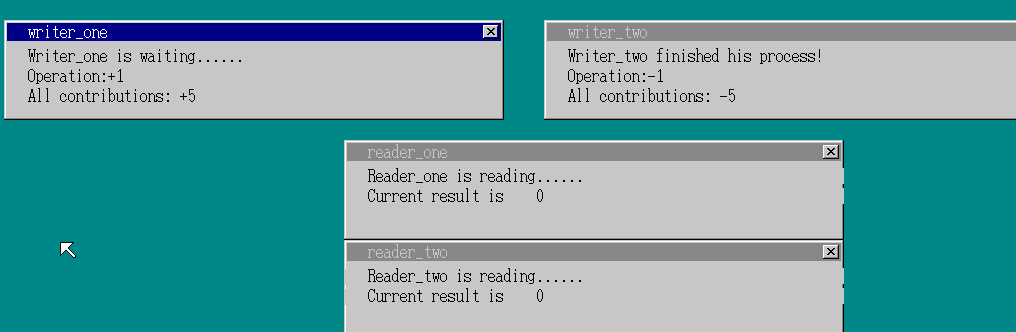
写者一写的时候，写者二和读者一、二都在等待。



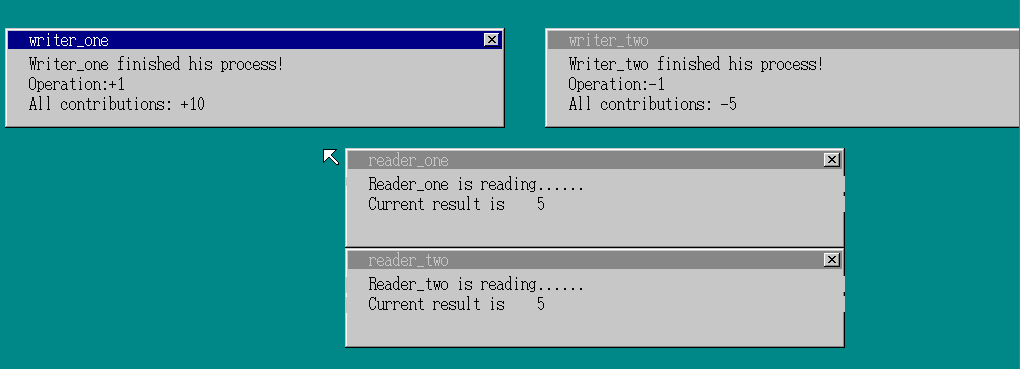
写者二写的时候，写着一和读者一、二都在等待。



读者一读的时候，写着一、二都在等待，读者二可以和读者一一起并发读。



最后，写者一加了10次，写者二减了5次，最终结果为5。



以上就是同步场景的演示，可以看到实现了读的时候不能写、写的时候不能读、可以同时读但不能同时写的设计，并且最终数据的确同步了。

函数set\_time\_waiting(int time)

类型：void

参数：等待的时间

作用：让进程等待来展示变量

函数task\_writer\_one(struct SHEET \*sht\_win\_b,struct TASK \* task)

类型：void

参数：图层和任务

作用：写者任务，负责加解锁和修改共享变量

函数task\_writer\_two(struct SHEET \*sht\_win\_b,struct TASK \* task)

类型：void

参数：图层和任务

作用：写者任务，负责加解锁和修改共享变量

函数task\_reader\_one(struct SHEET \*sht\_win\_b,struct TASK \* task)

类型：void

参数：图层和任务

作用：读者任务，负责加解锁和读取共享变量

函数task\_reader\_two(struct SHEET \*sht\_win\_b,struct TASK \* task)

类型：void

参数：图层和任务

作用：读者任务，负责加解锁和读取共享变量

1. **直接从用户态实现同步场景**

为了实现读者写者问题，分配的共享变量可以为数组，并需要只是都为之和写位置的指针。为了操作内核态的数据，需提供一些访问内核态数据的API如下所示：

struct SHAR \* api\_sharealloc(char \*name,int mode);//提供共享的变量

char api\_get(struct SHAR \* shar,int position); //获取指定位置的元素

void api\_change(struct SHAR \* shar,int position,int ch);//改变某个下标的值

int api\_getin(struct SHAR \*shar); //获取下标in的值

int api\_getout(struct SHAR \*shar);//获取下标out的值

void api\_changein(struct SHAR \*shar,int in);//利用in的值覆盖shar里面的in

void api\_changeout(struct SHAR \*shar,int out);//将out值覆盖掉原out值

共享的结构体如下所示：

struct SHAR{

int flag;//表示变量是否处于共享状态

int mode;//为0表示使用的为shared，大于0表示需要为数组分配的内存大小。

int inicialed;

int in,out;//标识读者写者位置的下标

char name[MAX\_NAME];//共享变量的名称

int shared;//共享的变量

char \* shared\_ch;//指向char数组的指针，可以指向任意大小

//解决竞争条件需要的数据结构

int share\_num;

unsigned char choosing[MAX\_PROCESS];

int number[MAX\_PROCESS];

};

通过基本的操作能够基本实现对数组的任意访问，为用户态实现同步场景提供了基础。

//分配共享变量，利用变量名称来区分，mode表示是否需要分配数组，如果大于0则分配mode大小的内存空间。

struct SHAR \* shar\_alloc(char \* name,int mode)；

//回收分配的共享变量

void share\_release(struct SHAR \* share)；

在这个生产者消费者模型中，我们使用一个5字节的char型数组作为仓库。之后设置了两个指针in和out

对这个模型的同步采用了书中的模型

生产者

while(1){

while((in + 1) % buffer\_size == out)；

buffer[in] == products[in];

}

in = (in + 1) % buffer\_size;

}

消费者

while(1){

while(in == out)；

nextproducts = buffer[out];

}

out = (out + 1) % buffer\_size;

}

具体代码中使用api\_putstr0和api\_putsstr1进行字符串打印

使用for循环进行空等。

之后同步场景是打开三个窗口，一个是生产者，负责向仓库中增加内容，分别向其中写入a,b,c,d,e五个元素，消费者则是取出数据，并把数据改写为0表示已经消费。最后一个窗口用来实时展示仓库的剩余库存。

## C:\Users\45799\AppData\Roaming\Tencent\Users\457997235\QQ\WinTemp\RichOle\8D76TK~DD4@F998GD%ZJB(S.png

**④新型内存分配算法**

修改./haribote/memory.c中的memman\_alloc函数使之支持最佳分配和最差分配（系统原只支持最先分配）

unsigned int memman\_alloc(struct MEMMAN \*man, unsigned int size)

/\* 0：最先分配

1：最佳分配

2：最差分配\*/

{

int choice = 2;

unsigned int i, a, addr, tar, maxOff;

int mark = 0;

switch(choice)

{

case 0://First-fit

{

for (i = 0; i < man->frees; i++) {

if (man->free[i].size >= size)

{

a = man->free[i].addr;

man->free[i].addr += size;

man->free[i].size -= size;

if (man->free[i].size == 0)

{

man->frees--;

for (; i < man->frees; i++)

{

man->free[i] = man->free[i + 1];

}

}

return a;

}

}

break;

}

case 1://Best-fit

{

for (i = 0; i < man->frees; ++i)

{

if (man->free[i].size == size)

{

addr = man->free[i].addr;

man->frees--;

for (; i < man->frees && (i + 1 < MEMMAN\_FREES); ++i)

{

man->free[i] = man->free[i + 1];

}

return addr;

}

else if (man->free[i].size > size && (mark == 0))

{

a = i;

mark = 1;

}

}

if (mark == 1)

{

addr = man->free[a].addr;

man->free[a].addr += size;

man->free[a].size -= size;

return addr;

}

break;

}

case 2://Worst-fit

{

tar = maxOff = 0;

for (i = 0; i < man->frees; i++)

{

if (man->free[i].size >= size)

{

if (man->free[i].size - size >= maxOff)

{

maxOff = man->free[i].size - size;

tar = i;

mark = 1;

}

}

}

if (mark == 1)

{

a = man->free[tar].addr;

man->free[tar].addr += size;

man->free[tar].size -= size;

if (man->free[tar].size == 0)

{

man->frees--;

for (; tar < man->frees && tar + 1 < MEMMAN\_FREES; tar++)

{

man->free[tar] = man->free[tar + 1];

}

}

return a;

}

break;

}

default:

break;

}

return 0;

}

**5.实验分工和总结**

|  |  |
| --- | --- |
| 姓名 | 分工 |
| 姚敏 | 解决竞争条件并开放成API和直接提供共享变量的API |
| 孙默 | 将同步场景开放成API和用户态同步场景 |
| 唐天晓 | 实现竞争条件解决方案互斥锁 |
| 闵庚云 | 实现竞争条件 |
| 张政 | 地址转换和新的内存算法 |
| 沈如晖 | 部分地址转换和文档 |
| 曾扬 | PPT及展示 |
| 孙伟 | 部分文档 |

姚敏：

通过这次操作系统的实验，对操作系统的工作机制有了较深的了解。为了实现竞争条件，特意的放大了停在数据写回内存中的时间。利用汇编语言编写程序。将函数封装成API共用户程序调用，对中断处理的流程理解更加深刻。

闵庚云：

通过本次实验，我深入了解到了操作系统中竞争条件的原理，认识到了操作系统的内部结构，在实验过程中加深了对操作系统的理解，通过团队合作，我还学习到了内存分配、竞争条件的同步场景及解决方案的知识。虽然经过了小组的讨论，但是我们还是没能想出为进程间分配共享内存的办法。总而言之，经过这次实验，我收获到了许多操作系统的知识。

张政：

在本次实验里，我主要学到了如何将逻辑地址转换为物理地址以及如何实现内存分配算法中的最优分配和最差分配。在实验过程中，我遇到了在bochs中无法正确查看寄存器值的问题，即使用info cpu出现错误信息，然后我编写程序动态获取到所需寄存器的值。然而使用sreg命令可以正确显示结果。第二个问题在于设计内存分配算法的演示场景，我最后模拟了一串空闲块，然后动态分配内存并将分配前后的空闲块状态显示出来，最后可以看到在不同分配算法下新申请的内存被分配到不同的空闲块上。总之这次实验让我收获良多，特别是让我明白了团队的力量真的大于个体之和，这对我们今后的学习工作都有很大的启示。

沈如晖：

通过本次实验，我更深入地了解到了地址转换在操作系统中的实现原理和意义，并且跟队友一起将其实现了。因为生病在家休息了一个月，所以课程知识多少有些跟不上。非常感谢队友不嫌弃，耐心地跟我讲解原理让我能够更好地参与实验。通过写文档的工作我也对其他成员的工作有个一定的了解，比如竞争条件的产生以及解决、还有同步方案等，也对用户态、内核态有了更加具体的认识。在实验中我们一起合作，一起进步，不仅收获了知识，更收获了友谊。

孙默：

通过本次实验，我学习到了同步场景实现的一些具体细节。虽然同步场景的原理都较为简单，但是在具体实现起来还是有各种各样的问题。比如在试验中sprintf这个函数，只是编译，都不调用的情况下就会对目标数据进行修改。而且因为对这个函数的实现机理不了解，所以也无法进行修改，只能暂时避免。

另一方面，自己定义并实现自己的函数，真的能够让代码的整洁性更好。在这个程序里面，我把定时器相关函数进行了封装，这样就可以一行代码实现等待，方便展示。

唐天晓：

这次了解多任务在内核态具体的实现，在编写互斥锁的过程中，学习了互斥锁的原理，在实现互斥锁的过程，更加深刻理解了系统多任务的实现过程，从而对互斥锁进行了优化。总的说，这次的实验加深了我对操作系统的理解和认识。

曾扬：

通过本次实验，实际的理解了操作系统中竞争条件、同步场景、逻辑地址与实际物理地址的转换关系、内存分配等实现方法的原理和实际操作。通过团队分工协作锻炼了分析问题、处理问题的能力，对今后的学习生活大有裨益。