1. 对于一个有p个处理器的共享存储系统：

1）写出一段程序，用load/store指令、运算指令和转移指令实现一个自旋锁。

int p;

int level[100];

int victim[100];

void **initial**()

{

for(int i = 0; i < p; ++i)

{

level[i] = 0;

}

}

void **lock**(int core\_id)

{

for (int i=1; i<p; i++)

{

level[core\_id] = i; //当前cpu在第i层

victim[i] = core\_id; //最新进入第i层的cpu是当前cpu

for (int k=0; k<p; k++)

while ((k != core\_id) && (level[k] >= i && victim[i] == core\_id)){} //如果自己是最后一个进入该层，并且已经有cpu在更高层，就自旋

}

}

void **unlock**(int core\_id)

{

level[core\_id] = 0;

}

上述程序中，没有使用testandset原子指令，以及对总线的特殊操作，因此可用用load/store指令、运算指令和转移指令实现。

该算法构建了p-1层障碍，每次通过障碍时，阻挡最后一个到达的cpu，最终只有一个cpu能到达临界区。对于一个cpu，当它处于比其他所有cpu都更深层的位置时，不会被阻拦。

汇编代码如下（精简不必要的标识符）：

@function

initial:

.LFB0:

endbr64

pushq %rbp

movq %rsp, %rbp

movl $0, -4(%rbp)

.L3:

movl p(%rip), %eax

cmpl %eax, -4(%rbp)

jge .L4

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq level(%rip), %rax

movl $0, (%rdx,%rax)

addl $1, -4(%rbp)

jmp .L3

.L4:

nop

popq %rbp

.cfi\_def\_cfa 7, 8

ret

@function

lock:

.LFB1:

endbr64

pushq %rbp

movq %rsp, %rbp

movl %edi, -20(%rbp)

movl $1, -8(%rbp)

.L11:

movl p(%rip), %eax

cmpl %eax, -8(%rbp)

jge .L12

movl -20(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq level(%rip), %rax

movl $1, (%rdx,%rax)

movl -8(%rbp), %eax

cltq

leaq 0(,%rax,4), %rcx

leaq victim(%rip), %rdx

movl -20(%rbp), %eax

movl %eax, (%rcx,%rdx)

movl $0, -4(%rbp)

.L10:

movl p(%rip), %eax

cmpl %eax, -4(%rbp)

jge .L7

.L9:

movl -4(%rbp), %eax

cmpl -20(%rbp), %eax

je .L8

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq level(%rip), %rax

movl (%rdx,%rax), %eax

cmpl %eax, -8(%rbp)

jg .L8

movl -8(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq victim(%rip), %rax

movl (%rdx,%rax), %eax

cmpl %eax, -20(%rbp)

jne .L8

jmp .L9

.L8:

addl $1, -4(%rbp)

jmp .L10

.L7:

addl $1, -8(%rbp)

jmp .L11

.L12:

nop

popq %rbp

.cfi\_def\_cfa 7, 8

ret

@function

unlock:

.LFB2:

endbr64

pushq %rbp

movq %rsp, %rbp

movl %edi, -4(%rbp)

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq level(%rip), %rax

movl $0, (%rdx,%rax)

nop

popq %rbp

ret

2）写出一段程序，用load/store指令、运算指令和转移指令实现一个公平的自旋锁。

int p;

int number[100];

bool entering[100];

void **initial**()

{

for(int i = 0; i < p; ++i)

{

number[i] = 0;

entering[i] = false;

}

}

int **max**(int number[100])

{

int temp = 0;

for(int i = 0; i < p; i++)

temp = temp < number[i]?number[i]:temp;

return temp;

}

void **lock**(int core\_id)

{

entering[core\_id] = true; //拿号

number[core\_id] = 1 + **max**(number); *//从number中找出最大的号码，加1后分配*

entering[core\_id] = false; //拿号结束

for (int i=0; i<p; i++) //遍历所有处理器

{

if (i != core\_id)

{

while (entering[i]); //等i号处理器取完号

while ((number[i]!=0) && (number[core\_id]>number[i] ||

(number[core\_id]==number[i] && core\_id>i)));

//如果i号已经取号，并且号码小于自己，或者号码与自己相同，但id号高于自己，就自旋；否则获取锁。

}

}

}

void **unlock**(int core\_id)

{

number[core\_id] = 0;

}

上述程序中，没有使用testandset原子指令，以及对总线的特殊操作，因此可用用load/store指令、运算指令和转移指令实现。

该算法遵循先来后到的原则，先取号的cpu会拿到较小的号码，而拥有较小号码和相同号码时，较小id的cpu会先进入临界区，因此是公平的。

汇编代码如下（精简不必要的标识符）：

@function

Initial：

.LFB0:

endbr64

pushq %rbp

movq %rsp, %rbp

movl $0, -4(%rbp)

.L3:

movl p(%rip), %eax

cmpl %eax, -4(%rbp)

jge .L4

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq number(%rip), %rax

movl $0, (%rdx,%rax)

movl -4(%rbp), %eax

cltq

leaq entering(%rip), %rdx

movb $0, (%rax,%rdx)

addl $1, -4(%rbp)

jmp .L3

.L4:

nop

popq %rbp

.cfi\_def\_cfa 7, 8

ret

@function

max：

endbr64

pushq %rbp

movq %rsp, %rbp

movq %rdi, -24(%rbp)

movl $0, -8(%rbp)

movl $0, -4(%rbp)

.L9:

movl p(%rip), %eax

cmpl %eax, -4(%rbp)

jge .L6

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

movq -24(%rbp), %rax

addq %rdx, %rax

movl (%rax), %eax

cmpl %eax, -8(%rbp)

jge .L7

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

movq -24(%rbp), %rax

addq %rdx, %rax

movl (%rax), %eax

jmp .L8

.L7:

movl -8(%rbp), %eax

.L8:

movl %eax, -8(%rbp)

addl $1, -4(%rbp)

jmp .L9

.L6:

movl -8(%rbp), %eax

popq %rbp

.cfi\_def\_cfa 7, 8

ret

@function

lock:

.LFB2:

endbr64

pushq %rbp

movq %rsp, %rbp

subq $24, %rsp

movl %edi, -20(%rbp)

movl -20(%rbp), %eax

cltq

leaq entering(%rip), %rdx

movb $1, (%rax,%rdx)

leaq number(%rip), %rdi

call max

leal 1(%rax), %ecx

movl -20(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq number(%rip), %rax

movl %ecx, (%rdx,%rax)

movl -20(%rbp), %eax

cltq

leaq entering(%rip), %rdx

movb $0, (%rax,%rdx)

movl $0, -4(%rbp)

.L17:

movl p(%rip), %eax

cmpl %eax, -4(%rbp)

jge .L18

movl -4(%rbp), %eax

cmpl -20(%rbp), %eax

je .L13

.L15:

movl -4(%rbp), %eax

cltq

leaq entering(%rip), %rdx

movzbl (%rax,%rdx), %eax

testb %al, %al

je .L14

jmp .L15

.L14:

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq number(%rip), %rax

movl (%rdx,%rax), %eax

testl %eax, %eax

je .L13

movl -20(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq number(%rip), %rax

movl (%rdx,%rax), %edx

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rcx

leaq number(%rip), %rax

movl (%rcx,%rax), %eax

cmpl %eax, %edx

jg .L14

movl -20(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq number(%rip), %rax

movl (%rdx,%rax), %edx

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rcx

leaq number(%rip), %rax

movl (%rcx,%rax), %eax

cmpl %eax, %edx

jne .L13

movl -20(%rbp), %eax

cmpl -4(%rbp), %eax

jle .L13

jmp .L14

.L13:

addl $1, -4(%rbp)

jmp .L17

.L18:

nop

leave

.cfi\_def\_cfa 7, 8

ret

@function

unlock:

.LFB3:

.cfi\_startproc

endbr64

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

movq %rsp, %rbp

.cfi\_def\_cfa\_register 6

movl %edi, -4(%rbp)

movl -4(%rbp), %eax

cltq

leaq 0(,%rax,4), %rdx

leaq number(%rip), %rax

movl $0, (%rdx,%rax)

nop

popq %rbp

.cfi\_def\_cfa 7, 8

ret

3）用load/store指令、运算指令和转移指令实现一个公平的自旋锁，分别最少需要访问多少个内存地址？

最少需要访问p个。每个cpu想要访问时，至少需要向1个地址写入信息，表明自己想要进入临界区，因此最少需要P个地址。否则，当P个cpu同时都想写入时，会出现覆盖写入，从而导致先写入的cpu信息被忽略，导致不公平。

1. 硬件同步原语

1）列出两种硬件同步原语，并给出他们在处理器上的实现

Load-Linked and Store-Conditional，LL/SC指令。LL指令从内存中读取一个字，并记录内存地址，将CPU中的一个特殊寄存器置1；SC指令根据该特殊寄存器的值检查是否为原子操作，若是则更新新值到内存中，并返回1（置源寄存器为1），否则不对内存进行操作，返回0（置源寄存器为0）。MIPS、ARM、PowerPC 实现了 LL/SC 指令。

CompareAndSwap，CAS指令。CAS指令将内存位置的内容与给定值进行比较，只有相同时才会将内存位置的内容修改为新的给定值。X86架构的指令集，IA64、以及Sparc也实现了CAS指令。通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新 值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

TestAndSet，TAS。一个TAS指令包括两个子步骤：把给定的内存地址设置为1，然后返回之前的旧值。在指令执行期间不会被其他处理器打断。TAS指令的底层可通过CAS的逻辑实现。

2）分析各种同步原语的优劣

LL/SC的关键步在SC的写。如果在LL与SC之间发生中断等例外，SC就强制失败，这导致SC的执行成功率较低，LL/SC实现的锁执行开销很大。

相比于LL/SC，CAS的硬件设计更加复杂。以x86为例，指令集中通过cmpxchg指令来实现对内存位置的读取、比较和改写。然而，该指令并不是原子指令。于是，存在A-B-A问题：cmpxchg在取回内存值A后，可能会被挂起；另一个处理器获取内存值A，通过cmpxchg指令将其修改为B；再有一个处理器获取内存值B，通过cmpxchg指令将其修改为A；随后被挂起的处理器重新执行cmpxchg，发现内存值还是A，便会误以为挂起期间没有其他处理器访问该内存地址，继续执行修改。这种问题在仅仅依赖值的结果上，cas不会出现问题，但是如果问题严格依赖值每一次的变化，就有可能出现故障。

举个不太恰当的例子：

ATM机取钱，账户A余额100，取出50。此时如果为保障通信，并发多个线程使用cas指令进行cmpxchg 100，50。理论上，应该只有一个线程成功，其他线程判定失败退出。但有可能在线程1执行结束前被挂起，其后的线程2成功执行，余额已经被修改为50。若恰好有人向A账户转账50，余额被成功修改为100。此时线程1继续执行，cmpxchg也会判定成功执行修改，于是余额又被修改为50，从而出错。

为避免ABA问题，一个好用的方法是将cmpxchg指令增加为128位，并用多出来的空间记录时间戳。

另外，由于cmpxchg指令并非原子指令，在访问并修改内存值的时候，可能有其他处理器也对该地址进行操作。为避免这种情况，通常使用lock cmpxchg来锁死总线。当冲突很频繁时，每一次cas操作都会发出广播通知其他处理器，并锁定总线，会引发缓冲一致性流量风暴，影响程序性能。

TAS的关键在于无论是否能获取锁，都会对共享内存进行一次写。当使用TAS实现TASLock锁，它的特点是自旋时，每次尝试获取锁时，底层还是使用CAS操作。会遇到与CAS相似的问题。

3）用各种同步原语实现公平的自旋锁

LL/SC的公平自旋锁:

@function

\_llsclock:

la r1, sem //加载锁地址

la r8, number //加载数字地址，r8寄存器用来传参

call acqire\_number //调用面包师算法，给当前core分配一个比当前所有数字都大的数字用来排号

sw x0, 4\*core\_id(number) //存放数字地址，n个处理器共有n个地址

TryAgain:

ll r2, 0x0(r1)

bnez r2, TryAgain

call compare //调用另外的比较函数，当number数组中最小的非0值等于4\*core\_id(number)中的值时，就继续执行，否则跳转到TryAgain

li r2, 1

sc r2, 0x0(r1)

beqz r2, TryAgain

nop

@function

\_llscunlock:

la r1, sem

sw r0, 4\*core\_id(number)

sw r0, 0x0(r1)

CAS的公平自旋锁：//为方便起见，定义指令CAS reg1，reg2，Mem reg1为旧值，reg2为新值，Mem为内存地址，若成功执行就置reg1为1，否则为0。初始(sem)=(number)=0

@function

\_caslock:

la r1, sem

Acquire\_number: //原子读取number，自己的号码为number记录在r4中,然后number+=1

la r3, number

add r4, r3, 1

cas r3, r4, 0x0(number)

beqz r3, Acquire\_number

add r5, r4, 1 //自增1

TryAgain:

cas r4, r4, 0x0(r1) //如果锁的值等于自己的number，说明轮到自己

beqz r4, TryAgain //若失败，说明锁的值不等于自己的number，未轮到自己，就自旋

sub r4, r5, 1

@function

\_casunlock:

la r0, sem

sw r5, 0x0(r1) //释放时，将锁的值+1