第12章作业.

3. 3. 加速比的定义是：使用增强措施时完成整个任务的性能/不使用增强措施时完成整个任务的性能。

(1) 请给出一个例子，使用多线程能获得超线性加速比(即采用多个线程比单个线程运行时间减少的倍数超过了线程数)。

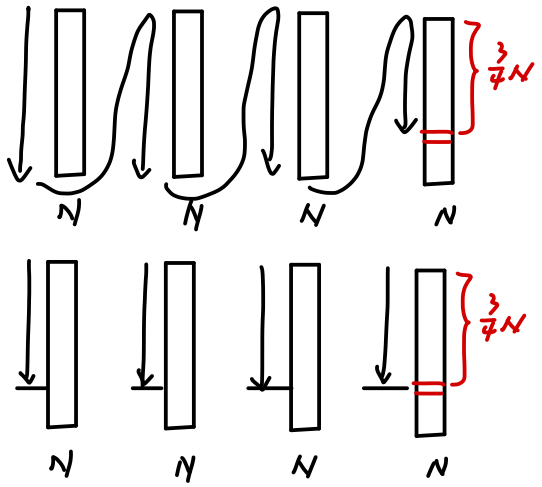
(2) 上述例子是否违背了 Amdahl 定律。

(1) 若能获得超线性加速比，则加速主要来源于：

① 对于多处理器并行，要访问的数据都位于各自的 Cache 中，而对于单处理器，由于 Cache 容量限制，平均数据访问速度慢于并行，从而导致超加速比。

② 对于多线程，顺序搜索所用时间可能高于多线程搜索时间之和。因为当搜索以并行方式进行时，能以不同的次序进行，这意味着总的被搜索数量少于顺序搜索。

如：



顺序搜索用时 $3N+\frac{3}{4}N=\frac{15}{4}N$

并行搜索用时 $\frac{3}{4}N$

加速比为 $\frac{15}{3}=5>4$.

(2) Amdahl 定律：加速比 $=\dfrac{1}{1-F_{\text{加速部分}}+\dfrac{F_{\text{加速部分}}}{S}}$

其中 $F_{\text{加速部分}}$ 为 可加速部分的占比

S 为 局部 加速比 . 当 $F=1$ 时，加速比 $=S$.

由(1)中分析可知，S 可以超过 n. 此时整个程序拥有超线性加速比.

但并不常见.

4. 4. 在共享存储多处理器系统中假共享会带来不可忽视的性能损失，为了尽量减少假共享的发生，程序员在写程序时应注意些什么？

假共享指不同处理器线程修改位于同一 Cache 行上的非相同内存单元。

这会导致导致不同线程并非真正共享某些内存位置的访问权，但却需要频繁的交换 Cache 行的权限。

因此，写程序时要避免多个处理器同时写的不同变量处于同一 Cache 行上。

5.

5. 在基于目录的 cache 一致性系统中，目录记载了 P1 处理器已经有数据块 A 的备份。

(1) 哪些情况下目录会又收到了一个 P1 对 A 块访问的请求。

(2) 如何正确处理上述情况？

(1) ①若 A 处于共享状态，而 P1 要写块 A，此时 P1 会向目录发出 write 请求以独占该块。

②P1 已经把 A 替换出去，而消息还未到达目录。此时 P1 又要访问块 A，便向目录发送请求。而后面的请求先于前面的消息到达目录

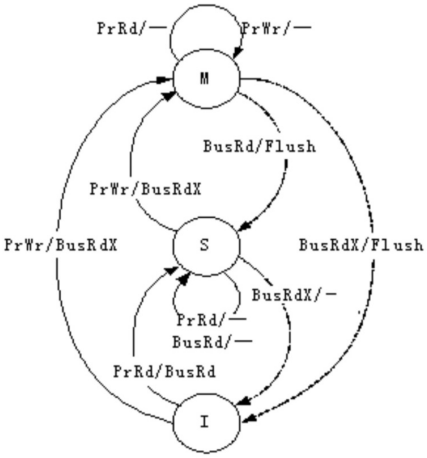
(2) ①是 ESI 协议中合法规定的情况，按 ESI 进行即可。

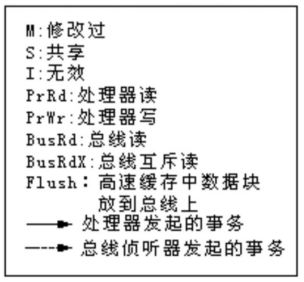
②可要求总线遵循 FIFO 的要求进行传递，即强制先发送先到达。

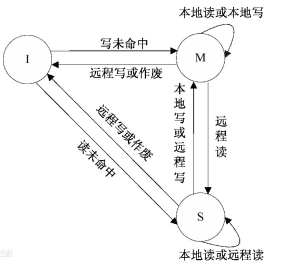
6.

6. 假设在一个双 CPU 多处理器系统中，两个 CPU 用单总线连接，并且采用监听一致性协议(MSI)，cache 的初始状态均为无效，然后两个 CPU 对内存中同一数据块进行如下操作：CPU A 读、CPU A 写、CPU B 写、CPU A 读，写出每次访问后两个 CPU 各自的 cache 的状态变化。

	CPU A	CPU B
初始	I	I
CPU A 读	S	I
CPU A 写	M	I
CPU B 写	I	M
CPU A 读	S	S







7.

7. 在下面程序段中，A、B、C、D 为进程 P1、P2、P3 的共享变量，且初始值均为 0。该程序的正确运行解结果是 D=2000。

(1) 在一个用可伸缩网络连接、采用基于目录的 cache 一致性协议的分布式共享存储系统中运行上述程序得到结果是 D=0，请解释产生上述结果的原因。

(2) 在实现顺序一致性的分布式共享存储系统中，对上述程序施加什么限制可以保证执行结果的正确性。

(3) 在实现弱一致性的分布式存储系统中，采用 barrier 进行同步，请问上述程序如何插入 barrier 操作才能保证执行的正确性？

```
P1                P2                    P3
A=2000 ①          while(B!=1) {;} ③     while(C!=1) {;} ⑤
B=1    ②          C=1;            ④     D=A             ⑥
```

(1) 逻辑上，P1 执行完后，P2 才会执行；P2 执行完后，P3 才会继续执行，应是由 B=1 和 C=1 的修改决定的。

程序能运行结束，说明 D=A 得以执行，即有 A=0

由此，原因为 P1 对 A 的修改未能传播至 P2 和 P3.

(2) 顺序一致性下，P1，P2，P3 内部均按顺序执行。则值指令执行顺序有唯一的 ①②④⑥，结果 D=2000 正确。故顺序一致性下无需施加限制。

(3)

P1	P2	P3
A=2000	barrier 1	barrier 1
B=1	while(B!=1) {;}	barrier 2
barrier 1	C=1	while (C!=1) {;}
barrier 2	barrier 2	D=A

8.

8. 有以下两个并行执行的进程，在顺序一致性和弱一致性下，它各有几种正确的执行顺序，给出执行次序和最后的正确结果(假设 a 和 b 的初始值为 0)。

```
P1                P2
a=1;  ①           b=1;  ③
print b; ②        print a; ④
```

顺序一致性：①一定在②之前，③一定在④之前，有 $\frac{C_4^4}{2\times 2}=6$ 种

1) ①②③④：a=1 b=0 打出：01

2) ①③②④：a=1 b=1 打出：11

3) ①③④②：a=1 b=1 打出：11

4) ③①②④：a=1 b=1 打出：11

5) ③①④②：a=1 b=1 打出：11

6) ③④①②：a=0 b=1 打出：01

弱一致性：任意执行顺序均可，有 $C_4^4=24$ 种执行顺序。

打印 a,b 时的值有：a=1 b=1

a=1 b=0

a=0 b=1

a=0 b=0

打印可以有：00

01

10

11

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
(%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。

2. 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
```