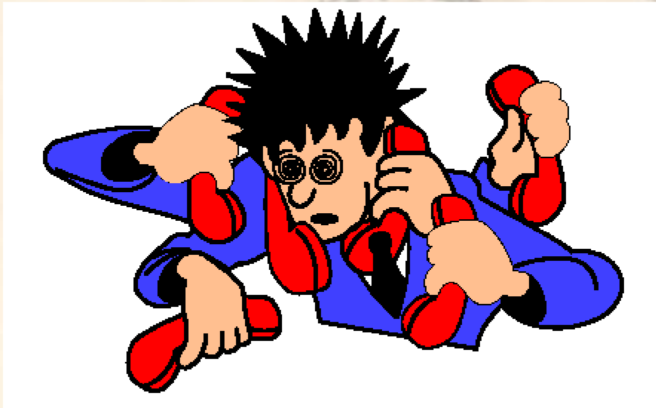
Chapter four

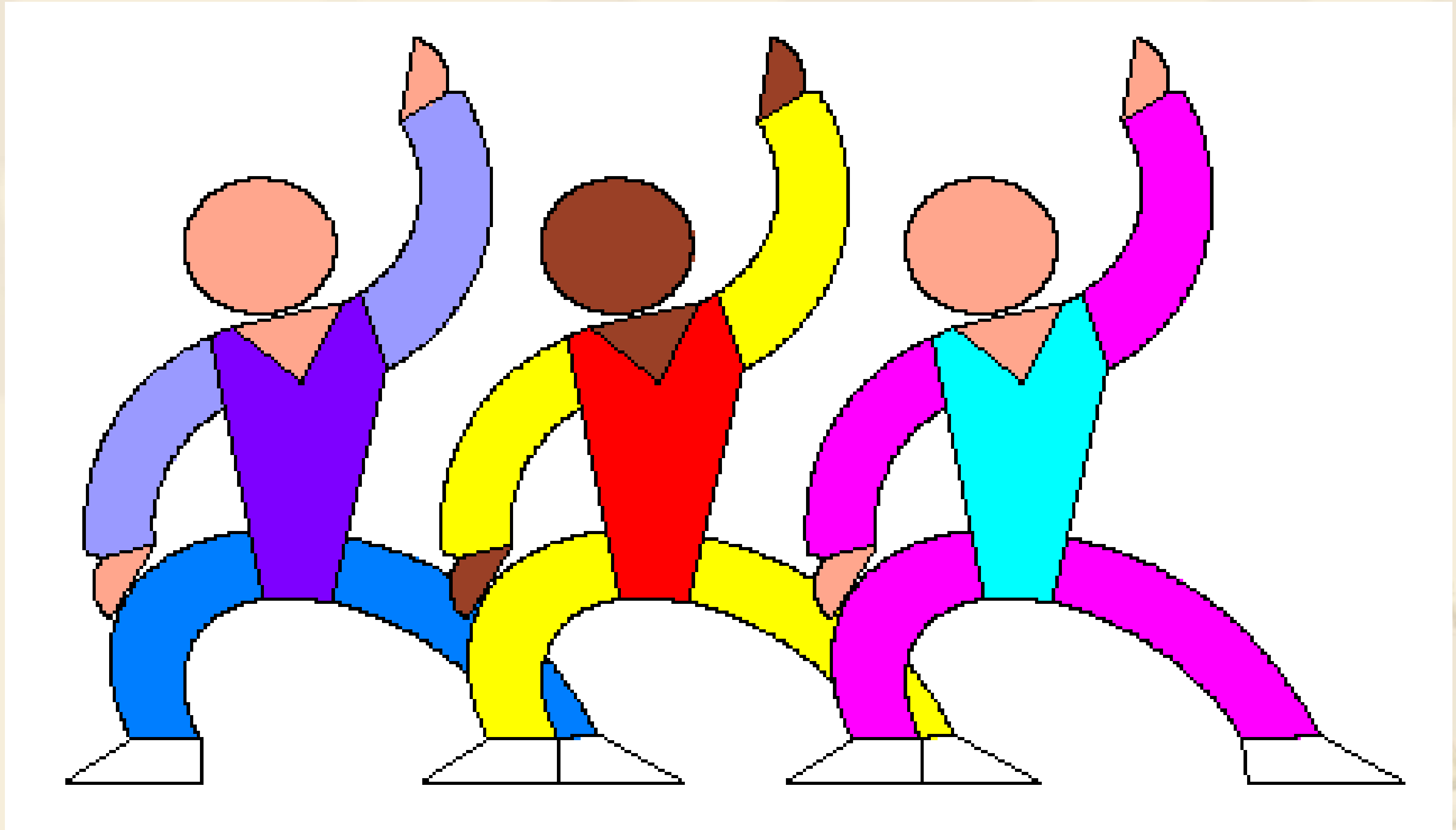
Multiprocessors and Thread-Level Parallelism

（续2）



陈文智

4.4 同步



概念

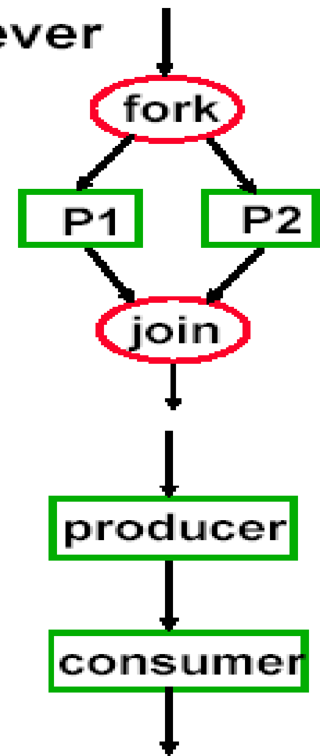
Synchronization

The need for synchronization arises whenever there are parallel processes in a system *(even in a uniprocessor system)*

Forks and Joins: In parallel programming a parallel process may want to wait until several events have occurred

Producer-Consumer: A consumer process must wait until the producer process has produced data

Exclusive use of a resource: Operating system has to ensure that only one process uses a resource at a given time



- 运行在不同处理器上的进程之间需要通信以协调地完成一个任务。进程间的通信可以通过使用共享变量来实现信息交换。但对共享变量的访问要保证互斥访问。即：保证每次只有一个进程访问共享变量。
- 同步机制的实现
  - 硬件提供同步原语；
  - 用户层软件实现。
- 在小规模或竞争较少的情况下
  - 硬件的关键功能：提供不可中断的指令；或实现原子地读和更新一个值的指令。
  - 软件机制在硬件基础上建立：如自旋锁。

- 在规模较大或竞争较多的情况下
  - 同步成为性能瓶颈。（延时增加）
  - 需研究更好的硬件机制来支持同步。
- 介绍顺序：
  - 硬件基本原语→构造基本同步例程→同步操作过程中竞争如何产生→更强的硬件原语

6.4.1 基本硬件原语

- 硬件原语的功能
  - 支持原子地读和修改存储单元；
  - 以某种方式告知是否进行了原子读或写操作(执行反馈)。

  是构造同步操作和同步库的基本构造模块。
- 几种典型的硬件原语
  - 原子交换（atomic exchange)
  - 测试和设置（test-and-set）
  - 取值和增值（fetch-and-increment）
  - LL/SC指令对：链接Load指令/条件Store指令（Load linked / store conditional ）

一、原子交换

- exch：原子地交换一寄存器和一个存储单元的值。
- 实现上锁：

```
        DADDUI  R2, R0, #1
lockit: EXCH    R2, 0(R1)
        BNEZ    R2, lockit
```

- 若有多个处理器竞争同一个锁，即试图同时执行交换操作，则由写操作的串行化保证，只有其中的一个处理器能首先执行交换操作。

- 二、TEST-AND-SET
  - 例子：(test 0) and (set 1)
- 三、FETCH-AND-INCREMENT
  - 例子：

四、链接Load指令/条件Store指令

- 指令对顺序执行；
- 如果某个被LL指令指定的存储单元的内容在SC指令对其操作之前改变了，那么SC指令就失败。如果处理器在两条指令之间作了一次进程切换，那么条件Store指令也失败；
- LL指令返回存储单元的初始值；
- SC指令返回的值表明Store操作是否成功。如果成功，返回1(??)，否则返回0。

链接Load指令/条件Store指令的优点:

- 可以用来构建其它同步原语。读写操作是分离的。
- 例：实现原子交换

```
try: OR      R3, R4,R0     ;传送交换值
     LL      R2, 0(R1)     ;链接load
     SC      R3, 0(R1)     ;条件Store
     BEQZ R3, try          ;条件Store失败，跳转
     MOV  R4, R2           ;取回的值送R4
```

- 例2：实现取值并增值

```
try: LL          R2, 0(R1)     ;链接Load
     DADDUI R3, R2, #1         ;增值
     SC          R3, 0(R1)     ;条件Store
     BEQZ      R3, try         ;条件Store失败，跳转
```

链接Load指令/条件Store的实现：

- 具体实现：
  - 设置链接寄存器。将LL指令中的访存地址保留在链接寄存器中。
  - 跟踪链接寄存器中的存储器地址，若发生中断或与该地址匹配的Cache块无效后，链接寄存器就被清除。
  - SC指令只是简单地检查操作地址与链接寄存器中的存储器地址是否匹配，若匹配成功，则成功，否则失败。
- 使用注意：
  - 一般只有r-r指令能插在指令对之间，否则可能发生死锁；
  - 插入两指令间的指令应尽可能少。

4.4.2 利用一致性实现锁同步

一、原子交换

- 自旋锁：

```
        DADDUI    R2,R0，#1
lockit: EXCH      R2, 0(R1)
        BNEZ      R2, lockit
```

- 无Cache一致性时，锁变量存放在内存中。
- 有Cache一致性时，锁变量可存放在本地Cache中。
  - 获得锁的自旋过程在本地Cache中进行，不必作全局访问。
  - 因访问局部性，锁值常驻Cache,减少了获得锁的时间。

自旋锁性能分析（缺点）

- 每次交换都尝试作一次写操作，此时，若多个处理器都试图获得锁，则每个处理器都会产生一个写失配。
- 改进：不断对本地锁copy作读操作，直到看到锁可用，再试图通过原子交换来获得锁。
- 改进后代码：

```
Lockit : LD          R2, 0(R1)      ;取锁值
      BNEZ    R2, lockit      ;锁不可用，自旋等待
      DADDUI      R2, R0，#1   ;
            EXCH       R2，0（R1）;交换
      BNEZ    R2, lockit       ;若锁值非0，重新开始竞争
```

Cache-coherence steps and bus traffic for P0,P1,P2

	P0	P1	P2	Coherence state of lock	Bus/directory activity
1	Has lock	Spins, test if lock==0	Spins, test if lock==0	Shared	None
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock Variable from P0
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write back fromP0
4		(Waits while bus/directory busy)	Lock=0	Shared	Cache miss for P2 Satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap,returns 0 and set Lock=1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate
7		Swap completes and returns 1,and set Lock =1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; generates write back
8		Spins, test if Lock ==0			None

前例说明

- 采用写无效一致性协议。P0退出并开锁（step2）后，P1和P2竞争，看哪个先读到所值等于0。P2胜出并进入临界区（step6,7），P1失败开始自旋等待。（step7,8）。
- 获得总线和对失配事件的反应要花很多时间。图中省略。

二、用链接Load/条件Store指令实现自旋锁

```
Lockit:LL       R2, 0(R1)   ;Load linked
        BNEZ    R2, lockit  ;not available-spin
        DADDUI  R2,R0，#1   ;locked value
        SC      R2, 0(R1)   ;store
        BEQZ    R2, lockit  ;branch if store fails
```

- 第一条转移指令是自旋循环，（读锁值，等待可用。）
- 第二条转移指令：是当两个处理器同时看到可用锁后竞争获得锁，失败者重新进入自旋等待。

同步操作性能分析

- 自旋锁的扩展性不好。由于由目录或总线来完成处理器同步操作的串行化，当处理器数目增大时，处理器间同步会使锁的竞争迅速加剧并带来大量的总线数据传输。造成同步性能下降。

例：自旋锁的同步性能

- 假设：共享总线的10个processor同时企图对一共享变量上锁（竞争锁）。设每次总线事务（一次read miss或write miss）需要花100个时钟周期。忽略在Cache中读写锁的时间。设开始的时候所有锁均释放，所有处理器都在自旋读锁值。假设总线是完全对称的，并且所有处理器一样快，要等当前一轮所有请求服务完以后才会响应新一轮到来的请求。
- 问：
  - 10个处理器都得到一次锁，共需完成多少个总线事务?
  - 完成10个处理器的上锁任务，需要多长时间。

例题解答：

- *i*个处理器从上一次释放锁到下次释放锁的过程
  - *i*次读锁值访问总线；
  - *i*次锁锁值访问总线
  - **1次释放锁，锁的所有者写→ 写失配**
    - **10个处理器作无效化操作。**
  - **共有2 *i* +1个总线事务；**
- **对于n 个处理器，共有总线事务是**
  - $\Sigma(2i+1)=n(n+1)+n=n^2+2n$
- *10*个处理器有120个总线事务，共12000个时钟，平均1个锁120时钟

三、Barrier Synchronization

- A barrier forces all processes to wait until all the processes reach the barrier and then release all the processes.
- 实现：两个自旋锁
  - 一个用于锁住共享变量：已到达进程计数器；
  - 一个迫使到达的进程自旋等待最后一个进程；

1. 实现Barrier的程序:

```
lock ( counterlock );                //ensure update atomic
if ( count == 0 ) release = 0;  // first-->reset release
count = count +1;                   // count arrivals
unlock ( counterlock );            // release lock
if ( count == total ){              // all arrived
        count = 0;                   // reset counter
        release =1;                 // release processes
}
else {                                  // more to come
        spin(release ==1) ;   // wait for arrivals
}
```

前面barrier实现代码分析：

- 有可能出现某个进程永远被关在Barrier内的情况：
  - **barrier常常用在循环体内，所以离开barrier的进程可能再次到达barrier;**
  - **跑得快的进程有可能在最后一个进程离开barrier前又已经再次到达barrier 。**
  - **跑得快的进程又重置release标志，使得所有进程会无限制地等在barrier那里。因为前一轮的最后那个进程不可能第二次到达barrier，所以计数器就永远达不到进程总数。**

可能的改进方法：

- 进程离开barrier时再次计数， 当离开barrier的进程数小于进程总数时，不允许其它进程再次进入barrier。
  - 缺点是进一步增大同步延时和共享变量访问竞争。
- 利用私有变量的sense-reversing barrier
  - 优点：解决了前述进程陷于barrier无限等待的问题，安全性更好；
  - 缺点：同步性能仍然不太好。

Sense-reversing barrier

```
初始值: local_sense = release
}
Local_sense = ! Local_sense;  //toggle local_sence
lock ( counterlock );                //ensure update atomic
count = count +1;                    // count arrivals
if ( count == total ){               // all arrived
        count = 0;                   // reset counter
        release = local_sense;       // release processes
}
unlock ( counterlock );              // release lock
spin(release == local_sense) ;  // wait for signal
}
```

2. Sense-reversing barrier性能分析

- 假设：共享总线的10个processor同时企图执行barrier同步操作。每次总线事务需要花100个时钟周期。忽略在Cache中读写锁的时间以及barrier操作中其它非同步操作的时间。设开始的时候所有10个处理器都在自旋等待对计数器上锁。假设总线是完全对称的，请求按顺序响应,并且所有处理器一样快。
- 问：10个处理器到达Barrier，然后释放离开barrier，共需完成多少个总线事务？整个过程需要多长时间。

分析

Event	Number of times for process *i*	Corresponding source line	Comment
LL counter lock	*i*	Lock(counterlock)	All processes try for lock
Store conditional	*i*	Lock(counterlock)	All processes try for lock
LD count	1	??? t+1	Successful process
Load linked	*i* -1	Lock(counterlock)	Unsuccessful process;try again
SD count	1	Count=count+1	Miss to get exclusive access
SD counterlock	1	??? lock)	Miss to get the lock
LD release	2	Spin(release==local_sense)	Read release: misses initially and when finally written

- 第*i*个处理器共有3*i*+4个总线事务
- 最后一个处理器到达栅栏需要少1个总线事务
- 10个处理器总共有204个总线事务:

$$\Sigma（3i+4）-1 = \frac{3n^2+11n}{2} - 1$$

同步操作性能分析

- 当多处理器间对共享变量的访问竞争严重时，同步性能会成为整个系统性能的瓶颈。
- 如果竞争不厉害，同步操作的频度又不高时，我们更关心同步操作的延时：即单个进程完成一次同步操作需要多长时间。
  - 一个基本的自旋锁操作需要花2个总线周期：一个周期用于读失配，一个用于写失配。
  - 减小同步延时至一个总线周期的改进方法是：自旋执行原子交换指令。缺点是：如果锁不常是可用的话，会使总线传输量急剧增大。

同步操作性能分析（2）

- 实际上自旋锁的性能不象例子中那么差。因为写失配是按upgrade处理的，因此比读失配要快。
- 严重的问题是：同步操作的串行化。
  - 当存在同步变量的访问竞争时，同步操作的串行化会大大增加完成同步操作的时间。
  - 例如：如果完成10次上锁和开锁操作，只和非竞争情况下的同步操作延时相关的话，那么总的只需要10×1×100= 1000时钟周期，而不是15000多个时钟周期。
  - 总线会加剧同步操作串行化带来的性能下降问题，同样在基于目录的多处理机中串行化带来的问题也很严重，因为同步操作的时延更大了。

4.4.3 大规模多处理机上的同步机制

- 目标：
  - 在非竞争的情况下，减小同步操作延时；
  - 在竞争严重的情况下，使串行化操作最小化。
- 软件实现方法
  - 竞争导致延时增大的原因：当获得锁的尝试失败后就自旋等待，但多个处理器的自旋引起严重的无谓的竞争，增加了无谓的总线数据传输量。
  - 解决方法：当获得锁的尝试失败后，人为推迟再次获取锁的尝试。通过这种方法来减少竞争。减少需串行化的同步操作次数。

一、软件实现

1. 带指数后退的自旋锁

```
        DADDUI  R3,R0,# 1       ;R3=initial delay
lockit: LL      R2, 0(R2)       ;load linked
        BNEZ    R2, lockit      ;not available-spin
        DADDUI  R2, R2, #1      ;get locked value
        SC      R2, 0(R1)       ;store conditional
        BNEZ    R2, gotit       ;branch if store
                                 succeeds
        DSLL    R3, R3, #1      ;increase delay by 2
        PAUSE   R3              ;delays by value in R3
        J       lockit
gotit:          use data protected by lock
```

- Exponential back-off是一种常用的减少共享资源访问竞争的方法，如用于访问共享网络和总线。
- 实现的目的：希望减小同步操作延时，即使在竞争不严重的情况下，也能有较好的性能。
  - 不推迟初次自旋等待获取锁的尝试。以此保证在竞争不严重情况下的性能。代价是当有很多处理器竞争时，不能减轻处理器初次尝试获取锁时的竞争。
  - 也可以推迟初次自旋等待的延时，这可以解决初次尝试导致的严重竞争问题，但当仅有两个处理器时，性能会很差，特别是当第一个进程第一次获取锁时发现锁是不可用的时候。

2. Combining tree barrier

- Barrier竞争：
  - gather stage: 原子地更新计数器；竞争严重，因为要互斥访问同步变量，所以会产生更多的串行化同步操作。
  - release stage:读取释放标志。
- 通过合并树减少竞争：将多个请求以树的方式局部地合并，以减少同一共享变量的竞争。
- 合并树可同时用于**gather stage**和**release stage**。

Combining tree barrier实现代码

```
struct node{/* a node in the combining tree */
        int counterlock; /* lock for this node */
        int count; /* counter for this node */
        int parent; /* parent in the tree = 0..P-1cep except for root
};
struct node tree [0..P-1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release; /* global release flag */

/* function to implement barrier */
barrier (int mynode) {
        lock (tree[mynode].counterlock); /* protect count */
        tree[mynode].count=tree[mynode].count+1;
                /* increment count */
        unlock (tree[mynode].counterlock); /* unlock */
        if (tree[mynode].count==k) {/* all arrived at mynode */
                if (tree[mynode].parent >=0) {
                        barrier(tree[mynode].parent);
                } else{
                        release = local_sense;
                };
                tree[mynode].count = 0; /* reset for the next time */
        } else{
                spin (release==local_sense); /* wait */
        };
};
/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);
```

说明

- 合并树使用了预先建好的n元树结构。变量K表示扇入数。
- 每一个合并节点有一个单独的计数器和锁；因此在一个合并节点，最多只有K个进程竞争一个锁。当最后这第K个进程到达时，就将该节点的计数器清零，置释放标志。
- 当一个合并节点的所有结点到达时，才继续向上前进到其父节点，这样就减少了在父节点合并的进程数。
- 同时使用了反向感知技术，避免进程陷入一个栅栏无限等待的情况。

二、硬件原语的实现方法

- 分析两个硬件同步原语
  - 针对锁操作
  - 针对barrier，以及需计数或提供特别索引的用户级同步操作。
- 与前面介绍的硬件原语的不同之处：
  - 延时相同，但串行化操作少，当存在竞争时，有更好的可扩性。
- 前面锁实现方法的缺点：
  - 产生大量无谓的竞争。当锁被释放时，尽管最多只有一个处理器可以获得锁，但所有的处理器都会产生一个读失配请求和一个写失配请求。

改进：

- 显式地在需要获取锁的处理器间传递锁。以避免无谓的竞争。用一张表记录想获得锁的处理器，锁被释放时，将锁显式地传递给轮到的处理器。----queuing lock

1. 队列锁的实现
   - 用硬件实现，针对基于目录的机器，且每个处理器的Cache都是可寻址的。
   - 软件实现：用一数组跟踪等待锁的处理器。更适合于基于总线的机器。每个处理器各自使用不同的锁地址，可以将锁从一个进程显式地传递给另一进程。

队列锁工作原理

- 同步控制器----实现锁传递；
  - 在基于总线的系统中，可集成在存储器控制器中；
  - 在基于目录的系统中，可集成在目录控制器中。
- 当第一次访问共享变量失配时，失配事件被送到同步控制器。
- 若锁可用，则立即返回给请求的处理器。
- 若锁不可用，则由同步控制器生成一个节点请求记录，（比如将一个向量的某一对应位置位），然后返回给请求处理器一个锁定值。请求处理器则自旋等待该锁定值可用。
- 当锁可用时，由同步控制器从请求队列中选取一个处理器让它获得锁。它或是更新该处理器Cache中的锁值，或是无效化该处理器Cache中的锁值，使该处理器访问失配以获取一个新的有效副本。

队列锁的性能

- 问：10台处理器使用队列锁对变量进行上锁和开锁操作需要多少次总线操作？需多长时间？设队列锁在失配时更新锁值，其它假设与前面例子相同。
- 解：
  - 对n个处理器，
    - 每个处理器初始都想访问锁，均产生1个总线事务
    - 其中只有一个成功并需要释放锁，共产生n+1个总线事务
  - 余下n-1个处理器，每个只要2个总线事务
    - 一个接受锁；一个释放锁
  - 总共需要29个总线事务
    - n+1+2（n-1）=3n-1 = 29
  - 结论：比用一致性自旋锁的性能要好得多。

实现队列锁的关键技术

- 分辨锁的初次访问，因为只有初次锁访问请求进入队列排队。
- 实现锁的释放，以便将锁交给其它处理器。
- 实现等待进程队列：
  - 基于目录的机器上，队列类似于共享集合；可用类似的硬件实现目录和队列锁操作。
- 困难：硬件要能够回收锁。一个获得锁的进程可能因为进程切换，而不再被同一处理器所调度。如果没有回收锁的功能，可能传递出去的锁就永远不会被释放。

2. 取值并增值原语（fetch-and-increment）

- 用队列锁可以改进barrier操作的性能；
- 用fetch-and-increment能减少串行化操作
  - 可用于反向感知barrier。减少计数器增值时写失配的串行化操作；
  - 也可用于合并树barrier。减少树中每个结点上的串行化操作。

用取并增值原语实现反向感知barrier

```
Local_sense = ! Local_sense;  //toggle local_sence
fetch_and_increment(count);    //atomic update
if ( count == total ){          // all arrived
        count = 0;                // reset counter
        release = local_sense;     // release processes
}
else {     //  more to come
     spin(release == local_sense) ; // wait for signal
}
```

改进后判断-回旋（反向感知）Barrier的性能

- 问：设一个取并增值原语需100个时钟周期，其它假设与前相同，求10台处理器通过barrier的时间及总线操作次数？
- 解答：
  - 对n个处理器需
    - n个 fetch-and-increment操作
    - n次访问count操作cache失配
    - n次访问release操作cache失配
  - 总共3n次总线事务
    - 3×10×100=3000时钟

Why not (2n-1) ?

小结：

- 如何高效利用大规模并行计算机是极具挑战性的一个课题
  - 同步问题在大规模并行计算机中很尖锐；
  - 大的存储器访问延时
  - 计算负载不平衡问题

6.5 存储器连贯性模型

- 连惯性定义
  - 由Cache一致性出发：（1）一个处理器什么时候能看到被另一个处理器更新的数据。
  - 由利用共享数据进行通信出发：（2）一个处理器以什么次序观察另一处理器写回的数据。
  - 由利用读操作观察另一处理器写的结果：（3）不同处理器向不同存储单元进行读和写操作，必须按什么规则进行。

例子 （Cp418, Ep605）

```
P1:   A=0;
        …
       A=1;
L1:  if ( B==0)…
```

```
P2:   B=0;
        …
       B=1;
L2:  if ( A==0)...
```

P1、P2进程在不同的处理器上运行，**A**和**B**均为两个处理器的**Cache**中的两个单元，其初值都是**0**。

由不同假设导出不同结果

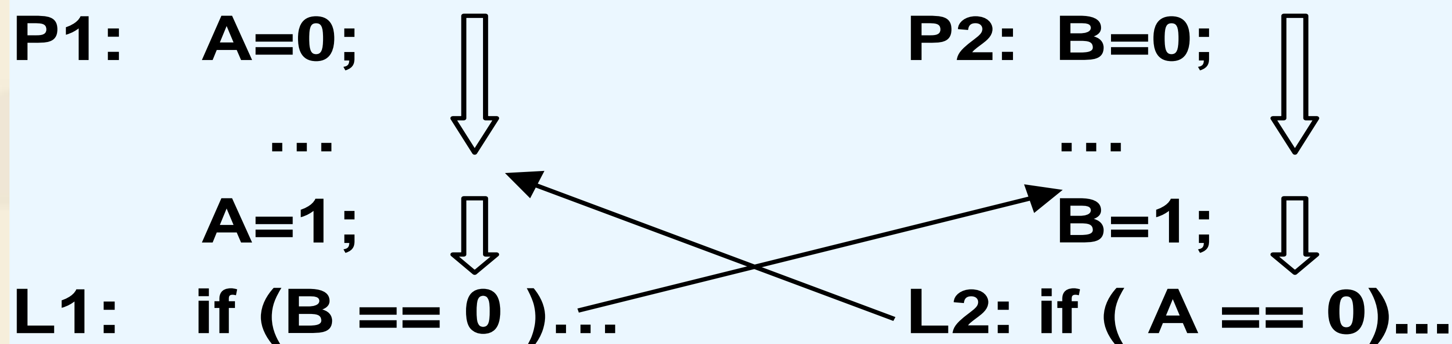
- 设写立即生效，且可立即被另一个处理器知道结果，则两个if语句为真的条件不可能都成立。
- 设写无效滞后生效，且处理器在滞后期中可继续执行，则两个if语句为真的条件可同时成立。

顺序连贯性（sequential consistency）

- 若处理器内访问是按顺序执行，处理器之间访问是交叉进行（即有先后次序，不能同时发生），则程序任何一次执行的结果都是相同的。（即可得到同一结果）。

重新考虑上述例子

- 一个进程内对存储器的访问必须按序进行，对**A**和**B**的读（即**P1**读**B, P2**读**A**）必须交叉进行，从而必定有一个先完成，这样两个if均为真不可能发生。
- 两个if均为真的条件是必须按下图细线次序进行。这一次序必定与程序的顺序相矛盾，因此不可能成立，或违背顺序连贯性定义。



实现顺序连贯性的最简单方法：

- 一个处理器让每一次存储器访问推迟到所有该次访问引起的无效处理完成后才结束。这意味着，下一次存储器访问推迟到前一次访问完成后才开始。
- 顺序连贯性是针对不同变量之间的操作而言。即要求按序完成执行的两次访问实际上针对的是不同存储单元。
- 顺序连贯性提供了一种简单的编程范例，但降低了机器的性能。
  - 比如，不能将一个写操作送入写缓冲之后就继续后续的读操作。

例:顺序连贯性对性能的影响 (Cp419, Ep606)

- 设：
  - 写失配需**40CC**建立拥有权；
  - 发出每个无效化信息需**10CC;**
  - 完成无效化处理和得到应答需**50CC;**
  - 设**4**个处理器共享一个**Cache**块。
- 问：在顺序连贯性条件下发生一次写失配，处理器将停顿几个时钟周期？
- 答：
  - 40 +（10+10+10+10）+ 50 = 130 （CC）

从程序员的角度考察顺序连贯性模型

- 顺序连贯性模型尽管会降低性能，但有编程简单的特点。
- 目的：找到一种编程模型，既简单明了，又有较高性能。
- 假设：程序是同步的。即所有对共享数据的访问都由同步操作排序。即在任何可能的执行顺序下，一个处理器的写变量操作与另一个处理器对这个变量的访问（写或读）被一对同步操作所分离。一个同步操作在写处理器的写操作后执行，另一个同步操作在第二个处理器的访问操作前执行。

相关概念

- 在没有同步操作排序的情况下发生的变量更新操作叫做数据赛跑（data races）。因为执行的结果依赖于处理器间的相对速度。
- 同步程序又称为免数据赛跑程序**(free-data-race**）。
- 例：两个处理器对同一变量的读和写操作。为保证访问的互斥性和连贯性，两个操作之间被一对同步操作所分隔：
  - 写操作之后的开锁（释放）
  - 读操作之前的上锁（请求）
- 两个处理器对同一变量的写操作，即使中间没有其他处理器的读操作，也必须被一对同步操作所分隔。

用请求和释放重新定义程序同步

- 如果程序中每个执行序列，该执行序列包含一个处理器的写操作以及紧跟其后的另一个处理器对同一数据的访问（**access**）操作，具有下列事件序列，则称程序是同步的：

  write(X)
  …
  release(S)
  …
  acquire(S)
  …
  access(X)

说明

- 如果程序是同步的，则程序不可能出现**data race**现象；因为对共享数据的访问都由同步操作安排。
- 可认为大多数程序都是同步的。否则难以确定程序的行为。
- 当然，程序员自己编写同步机制来确保顺序性是很难的。原因有：
  - 容易有**bug;**
  - 得不到体系结构的支持；
  - 不具有向后兼容性。
- 非同步访问主要用于想避免同步开销并且可以接受内存不一致性等场合。

存储器操作的顺序（1）

- 隔板----计算流程中固定的点，它保证读或写操作不能从隔板的一侧移到另一侧。
- 写隔板（write fences）
  - 所有在**P**执行写隔板操作前发生的**P**的写操作已经结束，在**P**的隔板后发生的写操作不能提前在隔板前启动。
  - 一般标记写操作最迟必须完成的执行点
- 读隔板（read fences）
  - 所有在**P**执行读隔板操作前发生的**P**的读操作已经结束，在**P**的隔板后发生的读操作不能提前在隔板前启动。
  - 一般标记读操作可能的最早执行点

存储器操作的顺序（2）

- 在顺序连贯性模型中，所有的读操作都是读隔板操作，所有的写操作都是写隔板操作。
- 存储器隔板操作（memory fences）：
  - 相当于读、写隔板操作结合的操作。它强制地对不同进程的存储器访问进行排序。在单一进程中，要求程序顺序不可改变，所以对同一存储单元的读、写操作不可互换。
- 松弛连贯性模型
  - 定义较少的读写隔板操作，使一些读写操作尽可能有一些重叠，从而实现隐藏读写延时。

松弛连贯性模型

- 目的：高性能的实现和简单的编程。
- 性质：根据这些模型编制的同步程序的执行语义与顺序连贯性模型下的执行语义相同。
- 区别：
  - 限制可能的执行序列的严格程度
  - 对实现的限制多少

- 四种顺序：
  - R→R：一个读操作紧跟一个读操作；
  - R →W: 一个读操作后跟着一个写操作，若对同一地址进行操作，则称反相关；
  - W →W: 一个写操作后跟着一个写操作；若对同一地址进行操作，则称输出相关；
  - W →R: 写操作后跟一个读操作，若对同一地址进行操作，则称真相关。
- 如果读写操作之间存在**相关性**，则单处理器程序的语义要求操作**按顺序**进行。如果**不存在相关性**，则由存储器**连贯性模型**决定哪些顺序必须被保持。

- 顺序连贯性模型：
  - 所有四种顺序都必须保持。这相当于假设存在一个集中共享存储器，所有的存储器操作都被串行化了。也等价于把所有的读写操作都看成是存储器隔板操作。
  - 尽管顺序连贯性模型要求四种顺序都保持，但在实际实现时有一定的灵活性，可以对部分事件重新排序，只要重新排序不被看到就可以了。
- 放松一种顺序：意味着处理器后执行的操作可以在一个先执行的操作完成之前先完成。
- 例：放松W →R:
  - **R**在**W**的写失配开始处理后开始执行；
  - **R**可以在**W**完成之前，即所有的无效化操作完成前完成。

- 松弛连贯性模型的实质：
  - 找出那些可被看到的顺序，
  - 只保持那些可见的顺序不变。从而减少必须保持的顺序的数目。
  - 例如两个写操作，可以允许后面的写操作在前一个写操作完成前开始执行，只要保证后一次写的结果在前一次写完成前不被看到就可以了。

连贯性模型

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	R→R, R→W, W→R,W→W	S→W, S→R, R→S, W→S, S→S
Total store order	IBMS/370, DEC,VAX, SPARC	R→R, R→W, W→W	S→W, S→R, R→S, W→S, S→S
Partial store order	SPARC	R→R, R→W,	S→W W→S
Weak ordering	PowerPC		S→W, R→S, W→S, S
Release consistency	Alpha, MIPS		S_A→W, S_A→R, R→S_R, W→S_R, S_A→S_A, S_A→S_R, S_R→S_A, S_R→S_R

W→S_A, R→S_A,
S_R→W,S_R→R

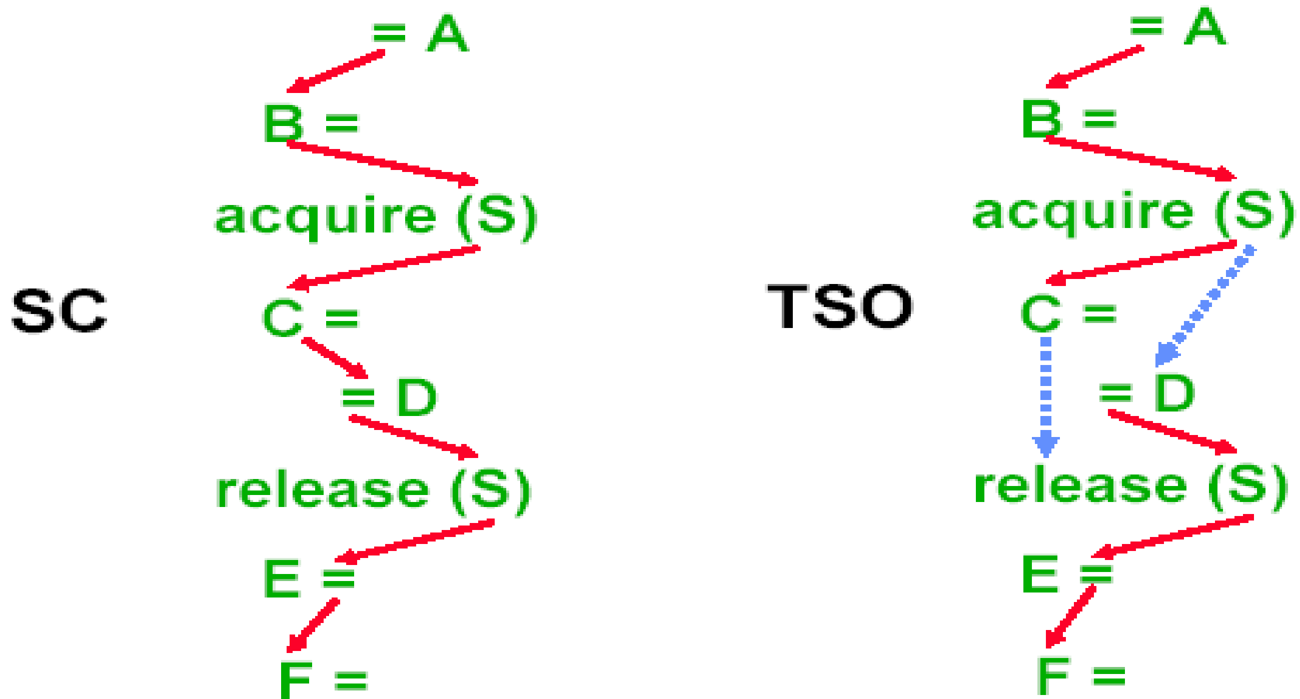
说明（1）---- TSO

- 放松了对**W→R**的要求。
- 允许设置写缓冲，使后续的读操作可以绕过写继续向前执行。即处理器在保证前一个写操作被所有处理器看到之前，就可以开始后续的读操作。*隐藏了部分写延迟*。
- 在此模型下，即使是非同步程序也可以正确执行，尽管需要一个同步操作来保证写操作在读操作完成前结束。即**W…S…R** 情况下，可以由**W →S, S→R**来保证**W →R**顺序。
- 此模型等价于让写操作成为写隔板操作。

Total Store Order (TSO)

IBM370, DECVAX

- Eliminates the order W(a) → R(b) $a \neq b$
- ***Advantage?***



TSO vs. SC

Initially x = old, y = old

Processor P_1

x = [text obscured by overlay]

y_copy = y;

Processor P_2

x_copy = x;

TSO both can get old values

SC at least one has to get the value of new

Under SC what values can x_copy *and* y_copy *get ?*

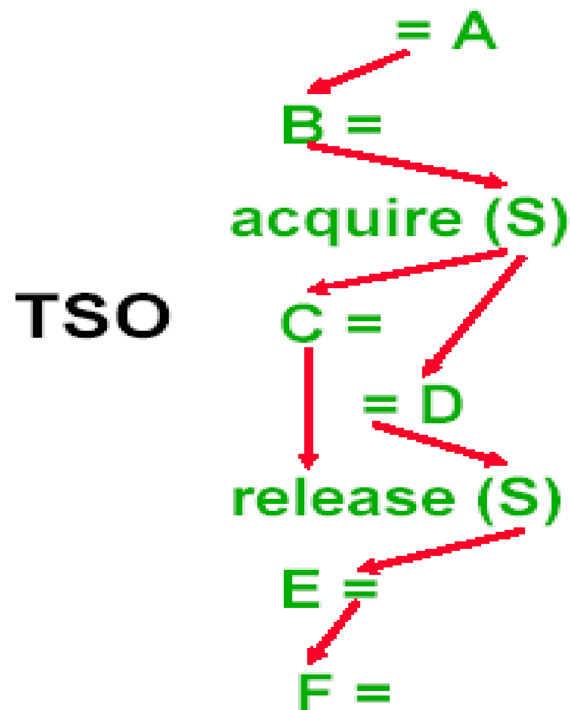
Under TSO what values can x_copy *and* y_copy *get ?*

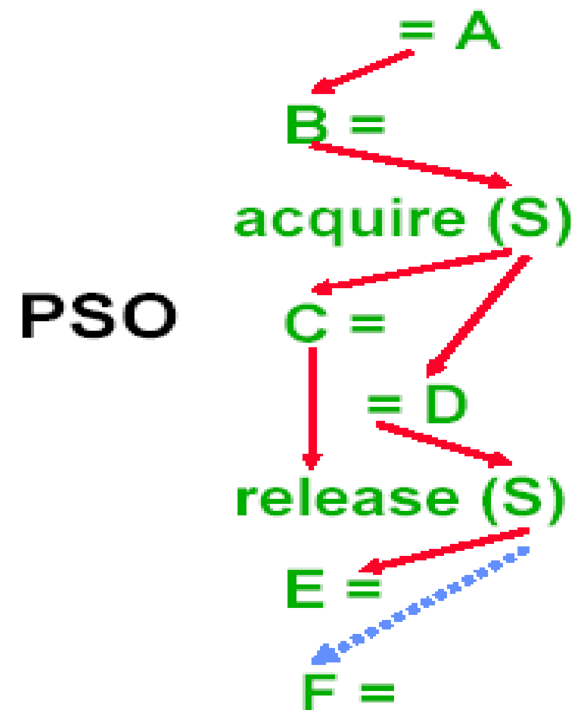
说明（2）---- PSO

- 进一步放松W →**W**顺序。即允许不相关的写操作可以乱序完成。
- 从实现的角度看，该模型允许流水化实现写操作，即允许重迭执行写操作，写操作仅当碰到同步操作（写隔板）时才会造成**Stall**。

Partial Store Ordering (PSO)
SPARC

- Also eliminates the order W(a) → W(b)   a ≠ b
- *Advantage?*





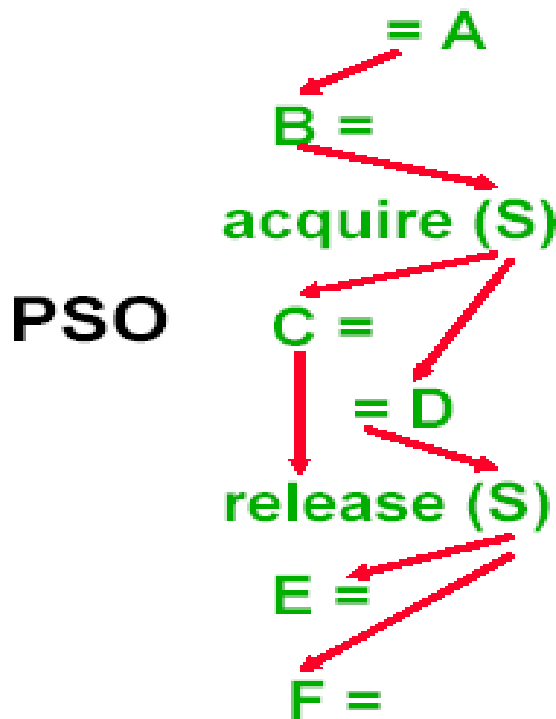
说明（3）---- Weak ordering

- 进一步放松**R →R, R → W**的顺序要求。
- 只要求保证*读写操作与同步操作*之间的顺序：
  - 程序顺序上在读写操作之后的同步操作要在读写操作完成之后才开始执行。
  - 程序顺序上在读写操作之前的同步操作必须在读写操作开始执行前完成。
- 仅当处理器支持*非阻塞读*功能时，去掉**R →R, R → W**才有好处。否则阻塞读操作隐含地保持**R →R, R → W**顺序。即使这样好处有限，即当**CPU**出现**Read miss**时，还是要等待。（非阻塞**Cache**可以更好地利用）
- 该模型的主要好处还是隐藏写延迟。

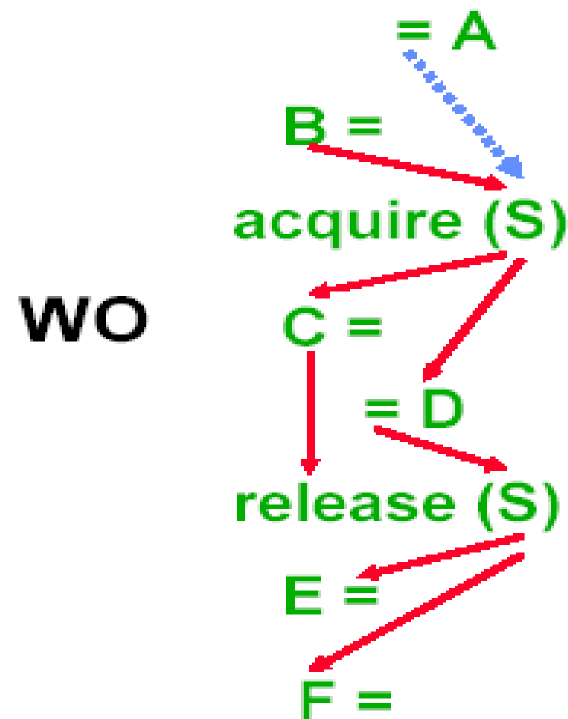
Weak Ordering
POWERPC

- Also eliminates the orders R(a) → R(b) $a \neq b$ and R(a) → W(b) $a \neq b$
- Need non-blocking reads to exploit relaxation

PSO



WO



说明（4）----Release consistency

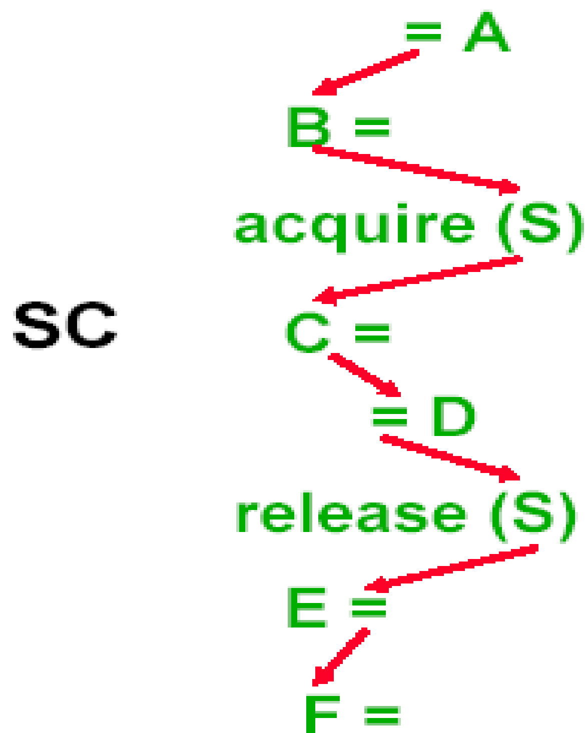
- 同步操作往往由acquire和release组成。
  - 先执行acquire, 然后才使用共享变量；
  - 在更新共享变量之后，必须执行一次release操作；
  - release操作之后，才允许下一个同步操作开始执行。
- 所以：
  - 在acquire之前的读、写操作不一定要在acquire操作之前完成；
  - 在release之后的读、写操作也不一定非要等到看到release结果之后才开始。

Release Consistency

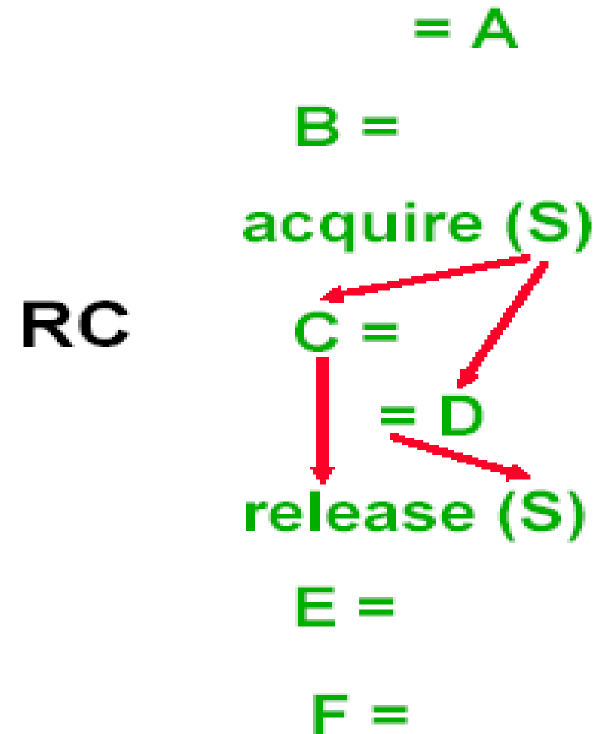
Alpha, MIPS

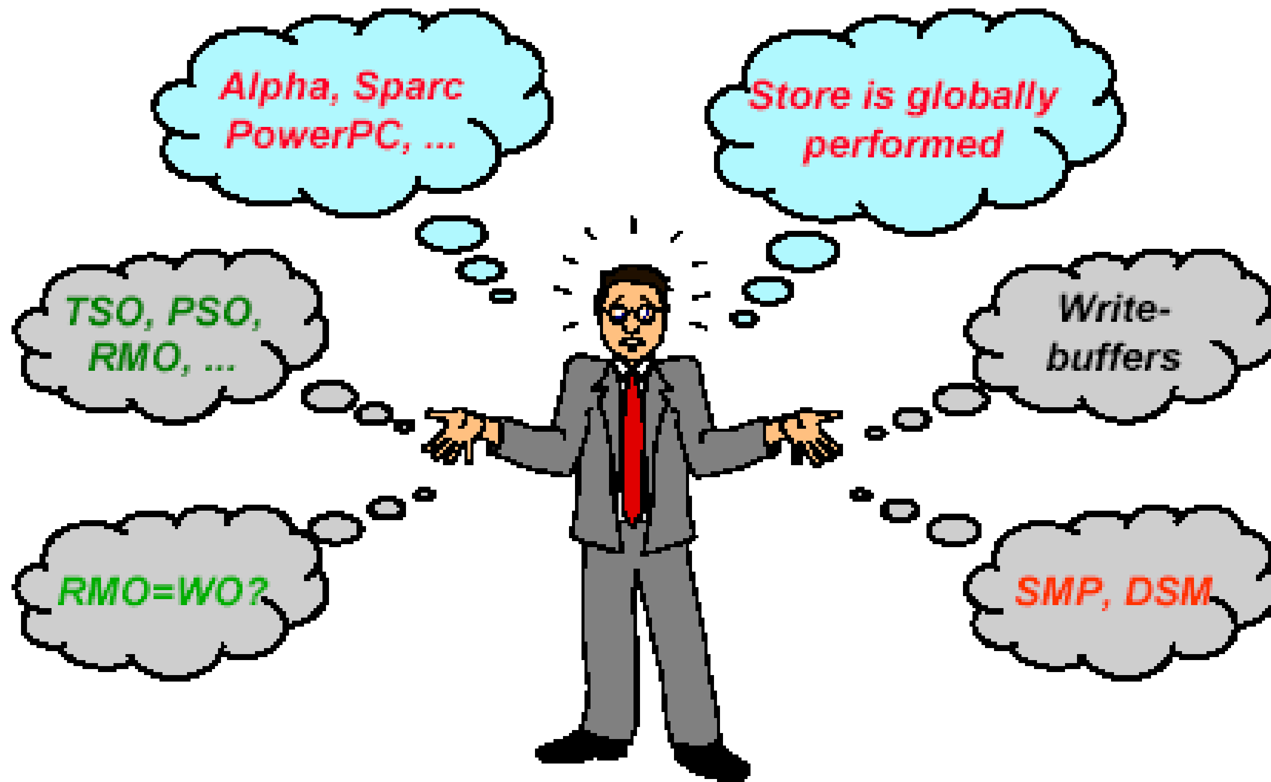
- Read/write that precedes acquire need not complete before acquire, and read/write that follows a release need not wait for the release

SC



RC





- Hard to understand and remember
- Unstable - *Modèle de l'année*