Computer Architecture, Spring 2017

清華大學
Tsinghua University

Synchronization

（同步机制）

王海霞
hx-wang@tsinghua.edu.cn
清华大学

并行程序

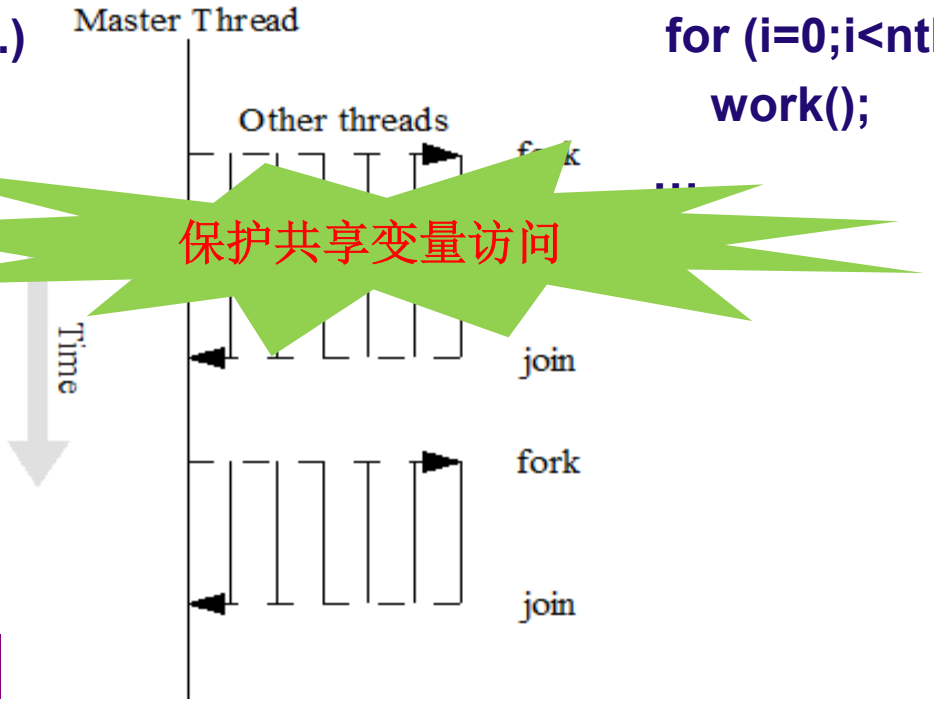
- 基于Pthread、OpenMP库的并行程序

Pthread

```
void main()
{
  …
  for (i=0;i<nthreads;i++)
     pthread_create(….)
  …
}
```

OpenMP

```
void main()
{
  …
  #pragma OMP parallel for
  for (i=0;i<nthreads;i++)
     work();
  …
```



共享变量访问冲突示例

- 银行账户管理问题:

```
int balance;

void deposit(int amount) {
        balance += amount;
}

int withdraw(int amount){
        balance -= amount;
        return balance;
}
```

- 如果2个人同时在ATM机上存钱，结果怎么样？

共享变量访问冲突示例

- 存钱实际上分为3条指令

```
void deposit(int amount) {
        balance += amount;
}
```

```
Load  R1, balance
 Add   R1, amount
Store R1, balance
```

- 如果两个人同时存会发生什么？

Process 1

```
Load  R1, balance
 Add   R1, amount
Store R1, balance
```

Process 2

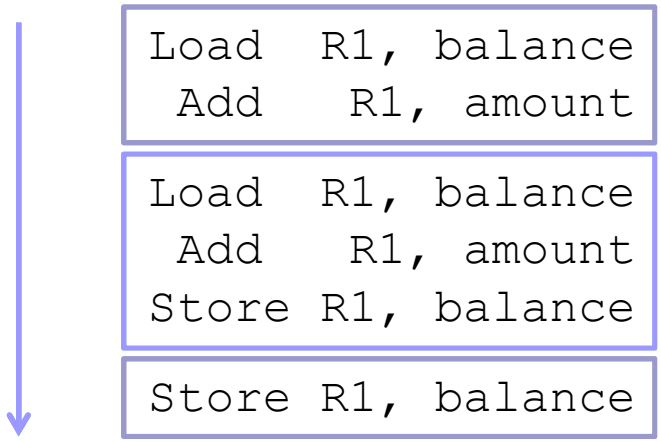
```
Load  R1, balance
 Add   R1, amount
Store R1, balance
```

- 竞争: 结果依赖于指令交叉顺序

共享变量访问冲突示例

- 如果指令交叉执行，银行账户结果是什么？

后台数据库看到的指令序列



竞争

- 两个并行执行的进程或线程访问共享资源，如果没有同步机制，会产生竞争（race）
  - 输出结果具有非确定性
  - 输出结果无法反应所有进程或线程对共享资源的修改
- 需要一种机制控制对共享资源的访问
  - 能够控制程序的结果

同步

- 同步机制用于保证对共享资源互斥（Mutual Exclusion）访问，这些只能互斥访问的代码集合称为临界区（Critical Section）
  - 任何时刻只有一个进程或者线程可以访问临界区
  - 所有进程或线程必须等待，直到其他进程或线程离开才能进入
- 对于临界区，需要获取锁（lock）才能访问
  - 锁对应一个存储单元的值：
    - 通常0表示未上锁（可用）；1表示已上锁（不可用）
  - 常用锁包括自旋锁（spin-lock）、互斥锁（mutex）等

Spin Lock和Mutex

- Spin lock:处理器不停请求获得使用权的锁
  - 属于busy-waiting类型，在等待spin lock的过程处理器不停的循环检查，并不执行其他指令
  - 适合临界区小、锁占用时间比较短的情况
- Mutex：是高级抽象锁信号量（Semaphore）的一种
  - 属于sleep-waiting类型，如果得不到锁进程或线程会被挂起
  - 适合临界区大、锁占用时间比较长的情况

同步机制

- 无论Spin lock、Mutex还是信号量，都需要在硬件提供的同步指令基础上，通过用户级同步库函数来建立的。
- 在多处理机中实现同步，所需的主要功能是：
  - 硬件原语：能以原子操作的方式读出并修改存储单元的同步指令。它们都能以原子操作的方式读/修改存储单元，并指出所进行的操作是否以原子的方式进行。
  - 同步库函数：通常情况下，用户不直接使用基本的硬件原语，原语主要供系统程序员用来编制同步库函数。

Pthread库提供的同步函数

- **Spin lock相关的函数**
  - pthread_spin_lock (pthread_spinlock_t *lock);
    pthread_spin_trylock (pthread_spinlock_t *lock);
    pthread_spin_unlock (pthread_spinlock_t *lock);

- **Mutex相关的函数**
  - pthread_mutex_lock (pthread_mutex_t *mutex);
    pthread_mutex_trylock (pthread_mutex_t *mutex);
    pthread_mutex_unlock (pthread_mutex_t *mutex);

硬件原语和锁

清華大學 Tsinghua University

- 1）原子交换（swap or xchg）
  - 功能：将一个存储单元的值和一个寄存器的值交换

swap r1, 0(&lock) ⟷

```
mov r1->r2
ld r1, 0(&lock)
st r2, 0(&lock)
```

  - 利用原子交换实现锁
    - 获取锁

```
A0: swap r1, 0(&lock)        (value of r1 is 1)
bnez r1, #A0
```

    - 释放锁

```
st 0(&lock) , r1             (value of r1 is 0)
```

  - 实现同步的关键：操作的原子性

硬件原语和锁

清華大學
Tsinghua University

- swap功能描述如下

```
void swap(int *lock, int *new) {
        int old = *lock;
        *lock = *new;
        *new = old;
}
```

- 锁实现示例

```
typedef struct __lock_t {
        int flag;
}
void init(lock_t *lock) {
        lock->flag = 0;
}
void lock(lock_t *lock) {
        int  new=1;
        while (new == 1) {
                swap(&lock->flag, &new);
        } // spin-wait (do nothing)
}
void unlock(lock_t *lock) {
        lock->flag = 0;
}
```

硬件原语和锁

清華大學
Tsinghua University

- 2）测试并置定（test_and_set）
  - 读回一个存储单元的值，并赋给新值
  - 功能描述如下

```
int TestAndSet(int *lock, int new) {
        int old = *lock;
        *lock = new;
        return old;
}
```

  - 锁实现示例

```
typedef struct __lock_t {
        int flag;
}
void init(lock_t *lock) {
        lock->flag = 0;
}
void lock(lock_t *lock) {
        while (TestAndSet(&lock->flag, 1) == 1); // spin-wait (do nothing)
}
void unlock(lock_t *lock) {
        lock->flag = 0;
}
```

硬件原语和锁

清華大學 Tsinghua University

3）比较并交换（compare_and_swap, cas）

- 将一个存储单元的值与一个给定的期望值进行比较，如果相等，则更新存储单元
- 功能描述如下

```
int CompareAndSwap(int *ptr, int expected,
int new) {
        int actual = *ptr;
        if (actual == expected)
                *ptr = new;
        return actual;
}
```

- 锁实现示例

```
typedef struct __lock_t {
        int flag;
}
void init(lock_t *lock) {
        lock->flag = 0;
}
void lock(lock_t *lock){
   while(CompareAndSwap(&lock-
>flag, 0, 1) == 1); // spin
}
void unlock(lock_t *lock) {
        lock->flag = 0;
}
```

硬件原语和锁

清華大學
Tsinghua University

- 4）读取并加1（fetch_and_add）
  - 返回存储单元的值并自动增加该值
  - 功能描述如下

```
int FetchAndAdd(int *ptr) {
        int old = *ptr;
        *ptr = old + 1;
        return old;
}
```

  - 锁实现示例

```
typedef struct __lock_t {
        int ticket;
        int turn;
} lock_t;
void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn = 0;
}
void lock(lock_t *lock) {
        int myturn = fetchandadd(&lock->ticket);
        while (lock->turn != myturn); // spin
}
void unlock(lock_t *lock) {
        fetchandadd(&lock->turn);
}
```

硬件原语和锁

清華大學
Tsinghua University

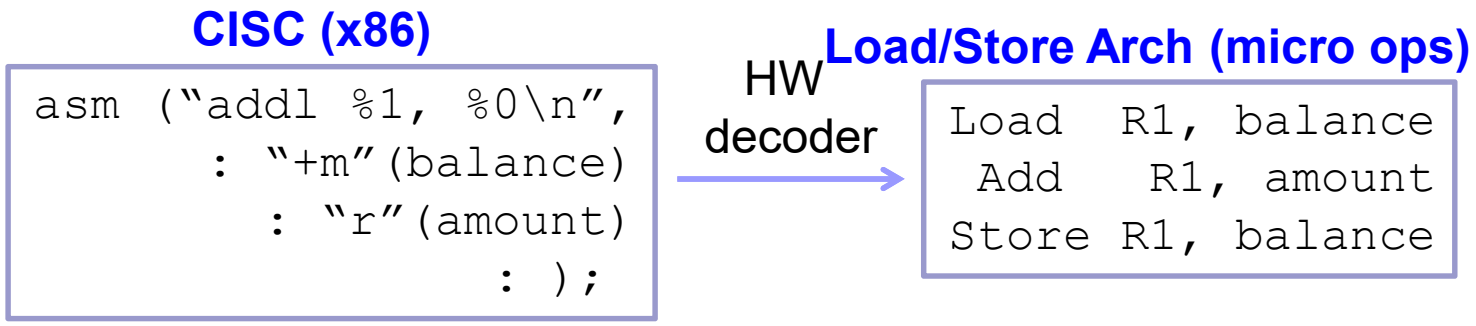
- 5)Load Linked/Store Conditional (LL/SC)
  - 功能描述如下

```
int  LL(int *ptr) {
        return *ptr;
}
int SC(int *ptr, int value) {
        if (no one has updated *ptr since
the load_linked to this address) {
                *ptr = value;
                return 1; // success!
        } else {
                return 0; // failed to update
        }
}
```

  - 锁实现示例

```
void lock(lock_t *lock) {
        while (1) {
                while (LL(&lock->flag) == 1);
                if (SC(lock->flag, 1) == 1)
                        return;
        }
}
void unlock(lock_t *lock) {
        lock->flag = 0;
}
```

硬件实现——x86 架构



- X86在指令前加Lock 前缀，可强制一条指令的所有微操作原子执行
  - 内存总线增加lock信号
  - Lock总线后，不允许其他处理器访问内存
- asm ("lock <instr>" :::)

硬件实现—LL/SC

- Load Linked/Store Conditional (LL/SC)
  - LL指令将要读取的存储单元地址保存到连接寄存器中
  - 如果在LL和SC指令间其他处理器修改了该存储单元的内容，或者发生中断，则连接寄存器被清空
  - SC指令执行时，检查它的存数地址是否与连接寄存器内容匹配，如果不匹配，则执行失败；
- SC返回一个值来指出该指令操作是否成功
  - “1”：成功；“0”：不成功
- Used by ARM, PowerPC, MIPS, Itanium

LL/SC实现其他原子指令

- 基于LL/SC可实现其他所有原子指令
- 实现原子交换swap

对由R1指出的存储单元与R4寄存器进行原子交换操作

```
try：OR      R3，R4，R0      // R4中为交换值。把该值送入R3
     LL      R2，0（R1）     // 把单元0（R1）中的值取到R2
     SC      R3，0（R1）     // 若0（R1）中的值没有被修改过，则
将R3值写入0（R1），置R3的值为1，否则置为0
     BEQZ    R3，try         // 存失败（R3的值为0）则转移
     MOV     R4，R2          // 将取的值送往R4
```

最终R4和由R1指向的单元值进行原子交换，在LL和SC之间如有别的处理器插入并修改了存储单元的值，SC将返回0并存入R3中，从而使这段程序再次执行。

LL/SC实现其他原子指令

- 实现fetch_and_add指令

```
try:    LL      R2, 0(R1)
        DADDIU  R2, R2, #1
        SC      R2, 0(R1)
        BEQZ    R2, try
```

无Cache一致性时的自旋锁

清華大學 Tsinghua University

- 无Cache一致性机制时，实现自旋锁只需要在存储器中保存锁变量，处理器可以不断地通过一个原子操作请求使用权即可

  比如：利用原子交换操作，并通过测试返回值知道锁的使用情况，并进行加锁。释放锁的时候，处理器只需简单地将锁置为0。

  例：用原子交换操作对自旋锁进行加锁，R1中存放的是该自旋锁的地址。

```
          DADDIU    R2, R0, #1
lockit:   SWAP      R2, 0(R1)
          BNEZ      R2, lockit
```

支持Cache一致性时的自旋锁

- 在系统支持Cache一致性时，原有自旋锁的实现方式存在什么问题？

```
            DADDIU      R2, R0, #1
lockit:     SWAP        R2, 0(R1)
            BNEZ        R2, lockit
```

- 主要问题：每次执行交换操作时产生一个写操作，该操作写作废所有有效副本，产生大量总线消息
- 如何解决？

支持Cache一致性时的自旋锁

- 开始重复读取锁变量的值; 只有当值发生变化时（锁被释放），才尝试执行交换操作（加锁）
- 优点：
  - 通过将锁调入Cache，使“环绕”的进程只对本地Cache中的锁（副本）进行操作，避免了每次请求占用锁时都进行一次全局的存储器访问；
  - 通过一致性机制使锁值保持一致，使得任何释放锁的操作对其他处理器可见

```
lockit:  LD      R3, 0(R1)     ;load var
         BNEZ    R3, lockit    ;not free=>spin
         DADDIU R2, R0, #1
         SWAP    R2, 0(R1)     ;atomic exchange
         BNEZ    R2, lockit    ;already locked?
```

例：自旋锁争用过程

清華大學 Tsinghua University

步骤	处理器P0	处理器P1	处理器P2	锁的状态	总线/目录操作
1	占有锁	环绕测试lock=0?	环绕测试lock=0?	共享	无
2	锁置为0	（收到作废命令）	（收到作废命令）	专有（P0）	P0发出对锁变量的作废消息
3		Cache不命中	Cache不命中	共享	总线/目录收到P2 Cache不命中；锁从P0写回
4		（因总线/目录忙而等待）	lock=0	共享	P2 Cache不命中被处理
5		Lock=0	执行交换，导致Cache不命中	共享	P1 Cache不命中被处理
6		执行交换，导致Cache不命中	交换完毕：返回0 并置lock=1	专有（P2）	总线/目录收到P2 Cache不命中；发作废消息
7		交换完毕：返回1	进入关键程序段	专有（P1）	总线/目录处理P1 Cache不命中；写回
8		环绕测试lock=0?			无

支持Cache一致性时的自旋锁

- LL / SC原语实现
  - 读写操作明显分开，LL不产生总线数据传送，这使下面代码与使用经过优化交换的代码具有相同的特点：

```
lockit:  LL      R2, 0(R1)
         BNEZ    R2, lockit
         DADDIU  R2, R0, #1
         SC      R2, 0(R1)
         BEQZ    R2, lockit
```

自旋锁的性能问题

- 大规模多处理机中，若所有的处理器都同时争用同一锁，则会导致大量的争用和通信开销
- 例题：
  - 假设某条总线上有10个处理器同时准备对同一变量加锁。如果每个总线事务处理（读不命中或写不命中）的时间是100个时钟周期，忽略对已调入Cache中的锁进行读写的时间以及占用该锁的时间。
  - 假设锁基于LL/SC实现，锁在时间为0时被释放，并且所有处理器都在旋转等待该锁。
  - 问：所有10个处理器都获得该锁所需的总线事务数目是多少？处理10个请求大概需要多少时间？

自旋锁的性能问题

- 解：当i个处理器争用锁的时候，它们都各自完成以下操作序列，每一个操作产生一个总线事务：
  - 访问该锁的i个LL指令操作
  - 试图占用该锁（并上锁）的i个SC指令操作
  - 1个释放锁的存操作指令

  对n个处理器，总的总线事务个数为：

$$\sum_{i=1}^{n}(2i+1) = n(n+1)+n = n^2+2n$$

对于10个处理器来说，其总线事务数为120个，需要12000个时钟周期。

栅栏（Barrier）

- 栅栏强制所有到达该栅栏的进程等待，直到全部的进程到达栅栏，然后释放全部的进程，形成同步
- 栅栏的典型实现

用两个自旋锁：

- 一个用来保护一个计数器，它记录已到达该栅栏的进程数；
- 一个用来封锁进程直至最后一个进程到达该栅栏。

□ 一种典型的实现

其中：

- lock和unlock提供基本的自旋锁
- 变量count记录已到达栅栏的进程数
- total规定了要到达栅栏的进程总数

栅栏

```
lock（counterlock）;                    //<确保更新的原子性<
if（count==0）release=0;                //<第一个进程则重置release
count=count+1;                          //<到达进程数加1<
unlock（counterlock）;                  //<释放锁<
if（count==total）{                     //<进程全部到达<
    count=0;                            //<重置计数器<
    release=1;                          //<释放进程<
}
else {                                  //<还有进程未到达<
    spin（release==1）;                 //<等待别的进程到达<
}
```

栅栏

- 实际情况中会出现的问题
  - 问题：如果最后一个进程到达，有的进程离开栅栏，进入下一次循环的栅栏并初始化release=0，但是有的进程还没有离开，那么有可能出现进程永远离不开栅栏的状况
  - 一种解决方法：当进程离开栅栏时进行计数（和到达时一样），在上次栅栏使用中的所有进程离开之前，不允许任何进程重用并初始化本栅栏。但这会明显增加栅栏的延迟和竞争。
  - 另一种解决办法：采用sense_reversing栅栏，每个进程均使用一个私有变量local_sense，该变量初始化为1。

栅栏

```
local_sense=! local_sense;          //<local-sense取反<
lock (counterlock) ;                //<确保更新的原子性<
count++;                            //<到达进程数加1<
unlock (counterlock) ;              //<释放锁<
if (count==total) {                 //<进程全部到达<
    count=0;                        //<重置计数器<
    release=local_sense;            //<释放进程<
    }
else {                              //<还有进程未到达<
    spin (release==local_sense) ; //<等待信号<
    }
```

同步问题

- 当竞争不激烈且同步操作较少时，主要关心的是一个同步原语操作的延迟。
  - 即单个进程要花多长时间才完成一个同步操作。
  - 基本的自旋锁操作可在两个总线周期内完成：
    - 一个读锁
    - 一个写锁

- 同步操作的主要问题：进程执行同步操作时的串行化问题，它大幅度地增加了完成同步操作所需要的时间。

MEMORY CONSISTENCY
（存储一致性）

Memory Consistency

- Cache Coherence
  - Creates globally uniform (consistent) view…
  - Of a single memory location (in other words: cache blocks)
  - Not enough
    - Cache blocks A and B can be individually consistent…
    - But inconsistent with respect to each other
- Memory Consistency
  - Creates globally uniform (consistent) view…
  - Of all memory locations relative to each other
- Who cares? Programmers
  - Globally inconsistent memory creates mystifying behavior

Shared Memory Example

- Initially: all variables zero

A=0, flag=0

Processor 0	Processor 1
A=1;	while (!flag); // spin
flag=1;	print A;

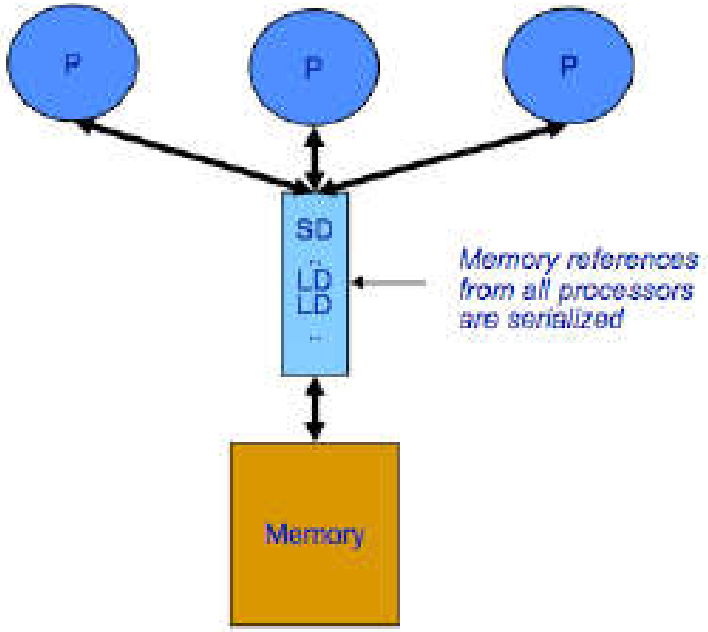
- Intuition says: P1 prints A=1
- In real system with cache coherence, P1 may prints A=0 (non-deterministic)
  - P1 can see P0's write of flag before write of A!!! How?
    - P0 has a coalescing store buffer that reorders writes
    - Or out-of-order load execution
    - Or compiler reorders instructions

Memory Consistency

- Reordering hiding store miss latency, but make system complicated
- Memory Consistency is needed to define when the value written by one processor (e.g. address A) must be seen by other processors

Sequential Consistency(SC)

- *Sequential Consistency (Lamport)* "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program."



Sequential Consistency(SC)

- **Sequential consistency (SC)**
  - Processors see their own loads and stores in program order
  - Processors see others' loads and stores in program order
  - All processors see same global load/store ordering
- **SC constrains all memory operations:**
  - Write → Read (Read After Write)
  - Write → Write (WAW)
  - Read → Read, Write (RAR, WAR)
- **Simple model for reasoning about parallel programs**

Relaxed Consistency Models

清華大學
Tsinghua University

- Sequential Consistency
  - + simplicity
  - -performance disadvantage
- How to develop a programming model that is simple and allows a high-performance implementation?
  - Based on synchronized programs
- Most programs are synchronized
  - A write of a variable by one processor and an access of that variable by another processor are separated by a pair of synchronization operations
  - Cases where variables may be updated without ordering by synchronization are called data race

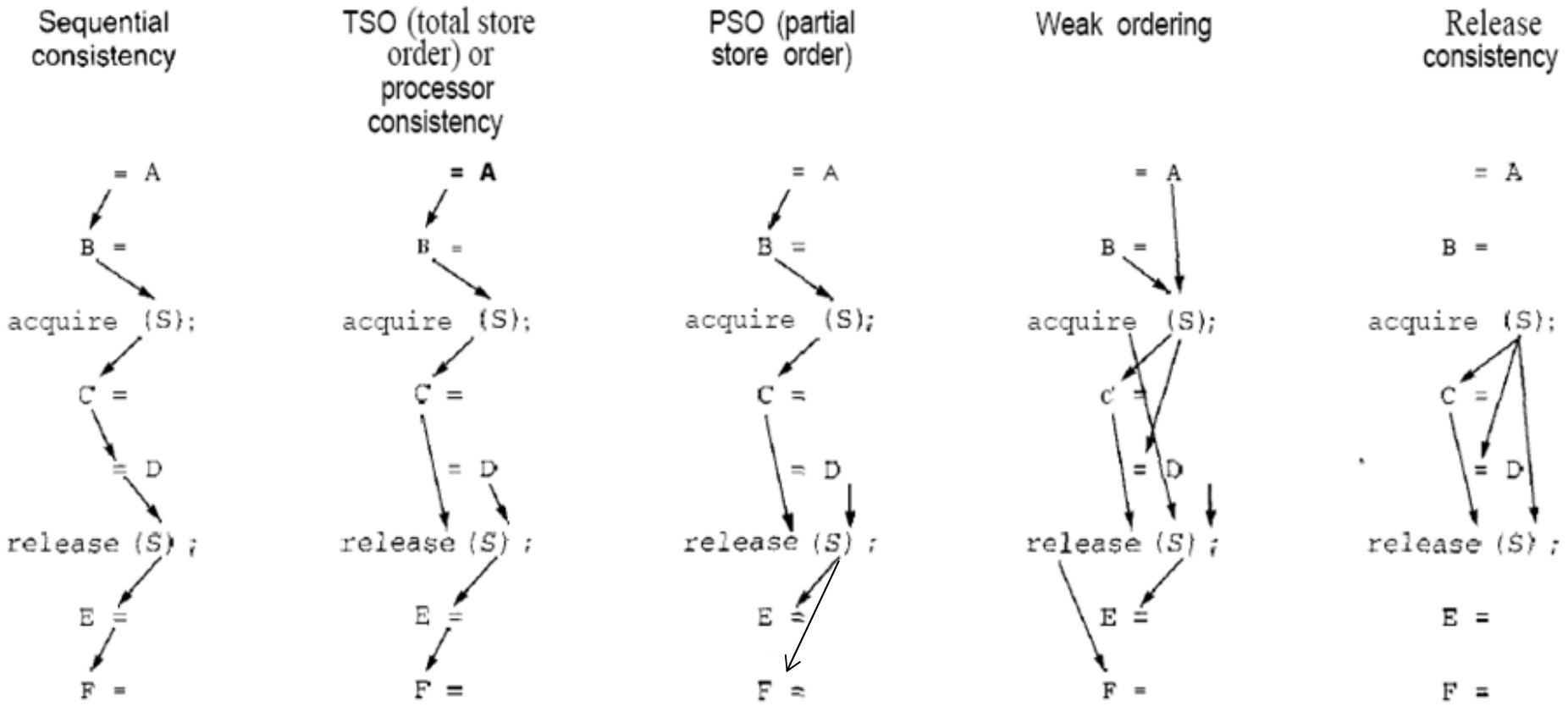
Relaxed Consistency Models

清華大學
Tsinghua University

- **Relaxed Consistency Models: allow reads & writes to complete out of order, but to use synchronization operations to enforce ordering**
  - **Relax W → R, W → W, R → R, R → W Ordering**

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 or processor consistency	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, S→R, R→S, W→S, S→S
Weak ordering	PowerPC		S→W, S→R, R→S, W→S, S→S
Release consistency	Alpha, MIPS		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_R$, $W \rightarrow S_R$ $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$

Relaxed Consistency Models



Weak Ordering

清華大學
Tsinghua University

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence
  - All data operations before synch in program order must complete before synch is executed
  - All data operations after synch in program order must wait for synch to complete
  - Synchs are performed in program order

Release Consistency

清華大學
Tsinghua University

- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - Acquires: operations like lock
  - Release: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - all memory operations before release are complete
  - but accesses after release in program order do not have to wait for release
  - operations which follow release and which need to wait must be protected by an acquire

清華大學
Tsinghua University

谢 谢！