

无锁编程简介

An Intro to Lock-free Programming

吕慧伟

ASG 小组讨论班

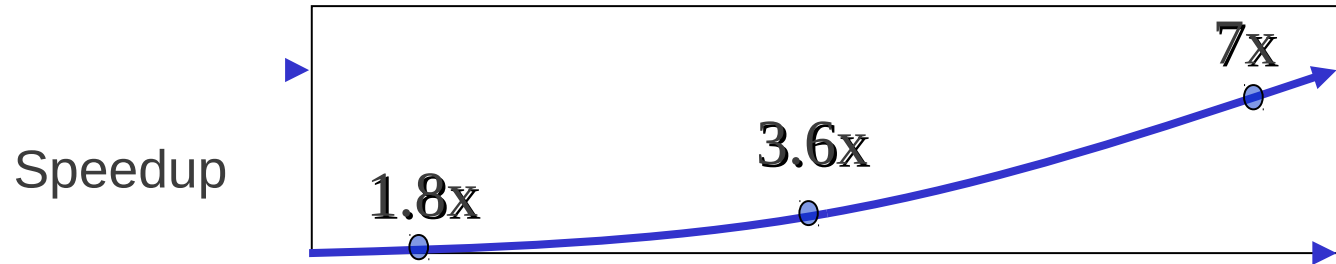
<http://asg.ict.ac.cn/lhw>

2011.7

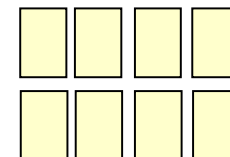
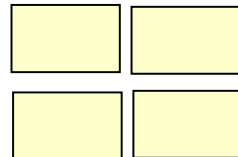
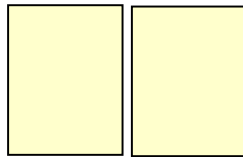
提纲

- 无锁编程概述
 - 动机：锁开销影响并行程序扩展性
 - What/Why/How
- 无锁编程实例
 - 无锁队列
- 无锁编程研究问题
 - 事务内存
 - 无锁数据结构

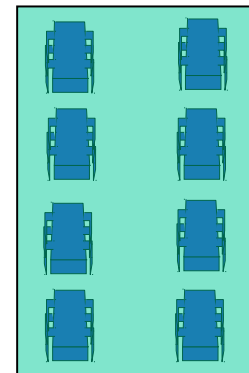
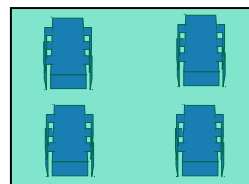
Multicore Scaling Process



User code

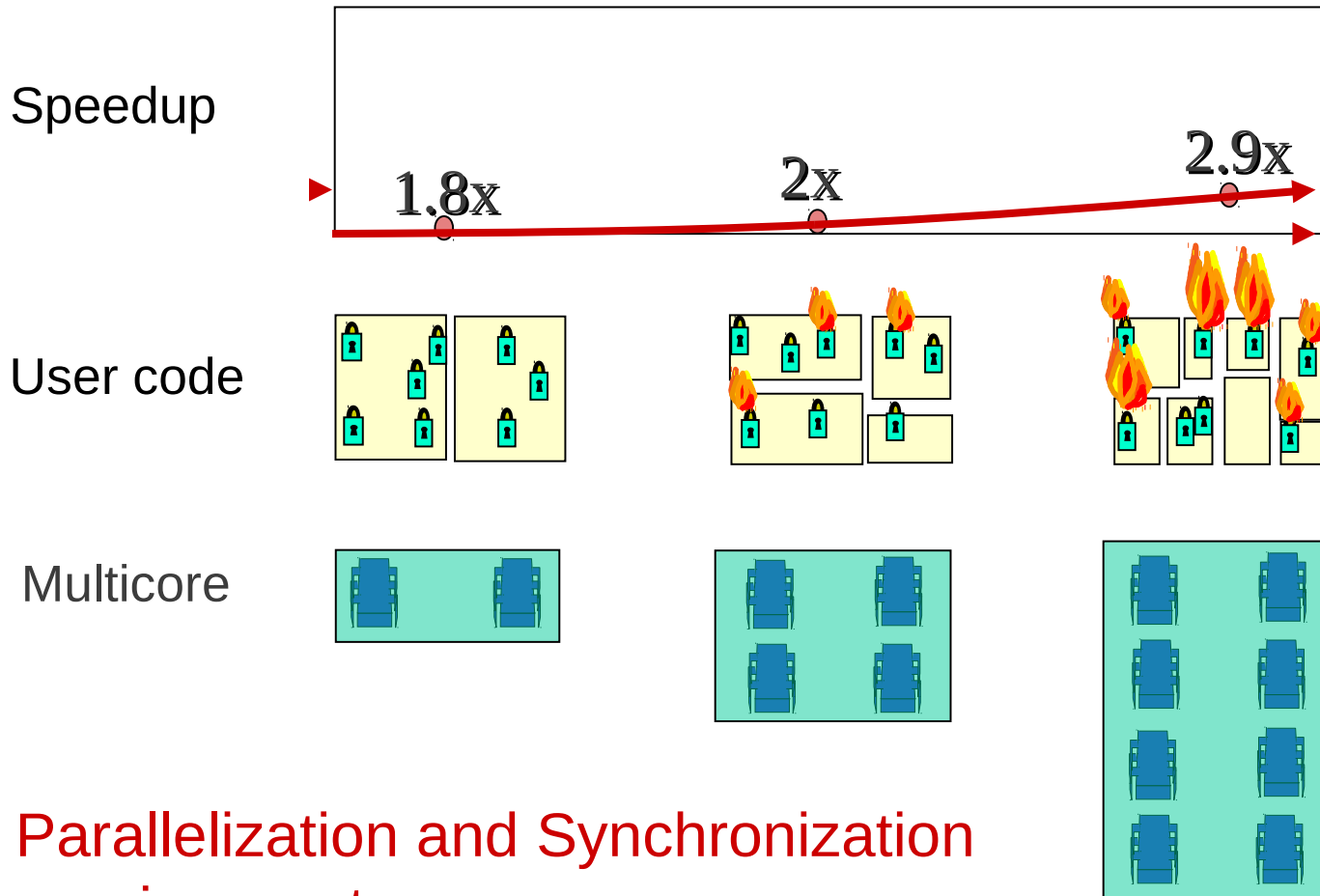


Multicore

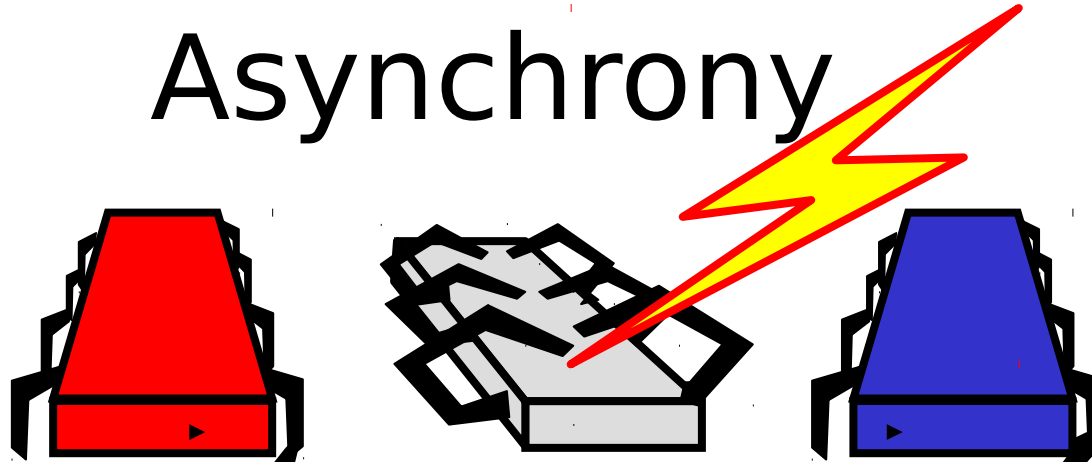


Unfortunately, not so simple...

Real-World Scaling Process



Asynchrony



- Sudden unpredictable delays
 - Cache misses (*short*)
 - Page faults (*long*)
 - Scheduling quantum used up (*really long*)

What: 什么是无锁编程？

- 如果一个共享数据结构的操作不需要互斥，那么它是无锁的。如果一个进程在操作中间被中断，其他进程的操作不受影响。 [Herlihy 1991]
- 并行算法同步的分类
 - 阻塞同步 (mutex, semaphore, ...)
 - 无阻塞同步 [LLF10]
 - 无等待 \in 无锁 \in 无阻碍
 - wait-free \in lock-free \in obstruction-free

Why: 为什么要无锁？

- 性能考虑
 - 对一些应用性能更好
- 避免锁的使用引起的错误和问题
 - 死锁：两个以上进程互等结束
 - Convoy：多个进程反复竞争同一个锁，抢占锁失败后强制上下文切换。引起性能下降。
 - 优先级反转：低优先级进程拥有锁时被抢占

How: 如何无锁？

- 方法：像事务一样操作 [DRD08]，知道谁拥有数据
 - 不同进程对私有数据更新互相隔离
 - 提交操作是原子的：或者成功，或者丢弃
 - 一致性：确保从一个状态变到另一状态
- 工具：原子指令
- 使用注意点
 - 无锁算法要求从硬件并行的角度考虑算法
 - 设计通用无锁算法很困难，一般只设计无锁的数据结构

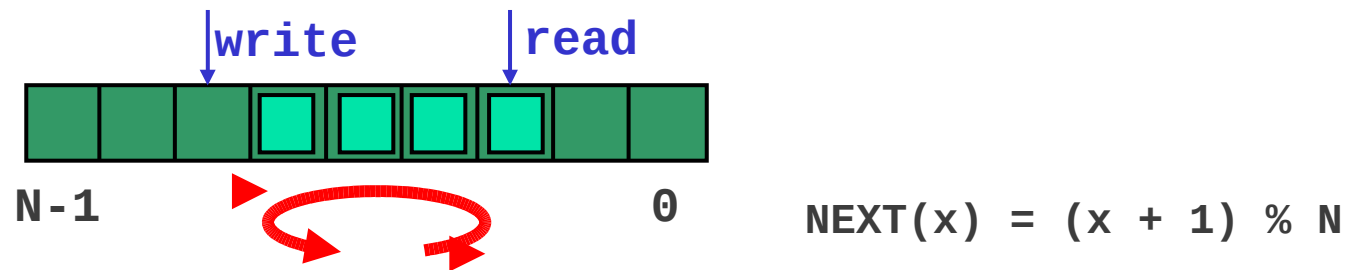
提纲

- 无锁编程概述
- 无锁编程实例
 - 单生产者单消费者 FIFO：不需要原子指令
 - 多生产者多消费者 FIFO：原子指令
 - 无锁堆栈、ABA 问题
- 无锁编程研究问题

Lamport's Lock-Free Ring Buffer

[Lamport, Comm. of ACM, 1977]

- Operate on control variables: **read** and **write**, which resp. point to next read and write slots



Insert(T element)

```
1: wait until  $\text{NEXT}(\text{write}) \neq \text{read}$   
2:  $\text{buffer}[\text{write}] = \text{element}$   
3:  $\text{write} = \text{NEXT}(\text{write})$ 
```

Extract(T* element)

```
1: wait until  $\text{read} \neq \text{write}$   
2:  $*\text{element} = \text{buffer}[\text{read}]$   
3:  $\text{read} = \text{NEXT}(\text{read})$ 
```

Compare-and-Swap

atomically

```
val CAS( val* addr, val old, val new)
{
    val prev = *addr;
    if (prev == old) { *addr = new; }
    return prev;
}
```

- *CMPXCHG* (with “lock”) – Intel x86
- *Load Linked / Store Conditional* – MIPS, PowerPC

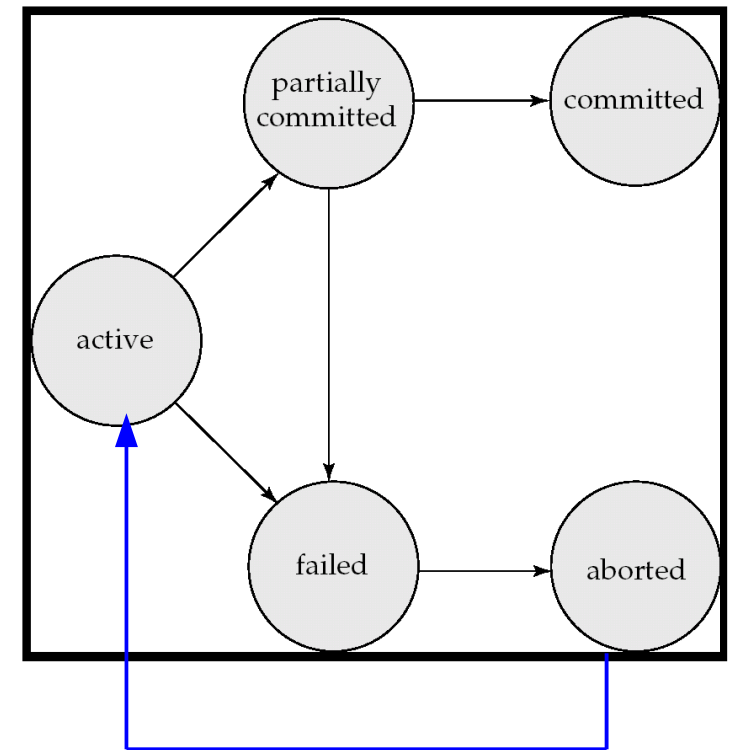
多生产者多消费者

FIFO

```
void enQ(request, queue)
{
    do{
        local_head = queue->head;
        request->next = queue->head;
        val = cmpxchg(&queue->head,
local_head, request);
    }while(val != local_head)
}
```

Atomic & Consistency: **cmpxchg**
Isolation: 私有数据

事务操作状态

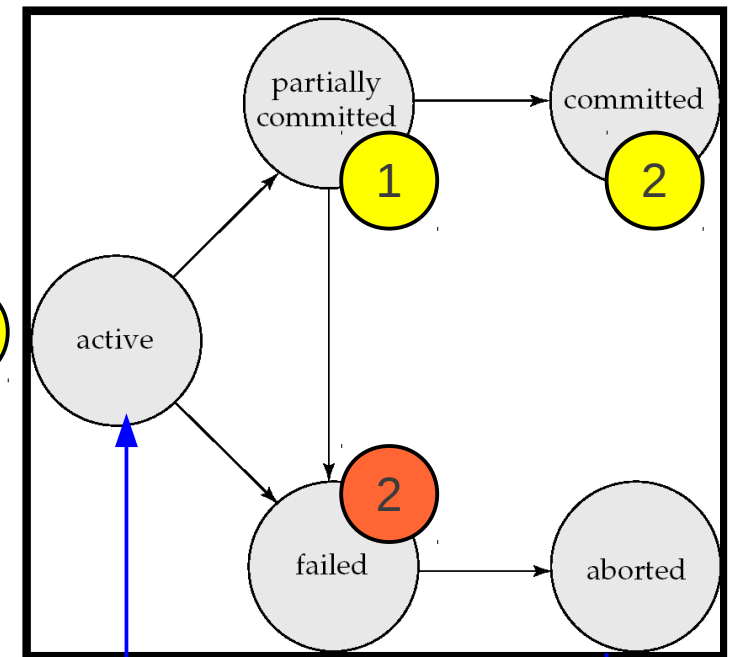


多生产者多消费者

FIFO

```
void enQ(request, queue)
{
  do{
    local_head = queue->head;
    request->next = queue->head;
    val = cmpxchg(&queue->head,
local_head, request);
  }while(val != local_head)
}
```

事务操作状态



如果在执行 `cmpxchg` 时，`queue->head == local_head`。即在 1, 2 之间没有进程将 `queue->head` 更改了。则 `request` 赋给了 `queue->head` 否则，上述过程将重新进行

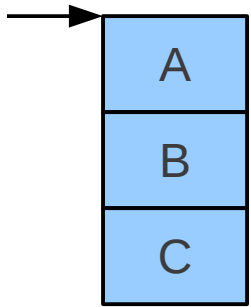
无锁堆栈

```
struct elem {  
    elem *link  
    any data;  
}  
  
elem *qhead;
```

```
Push (elem *x)  
do  
    old = qhead;  
    x->link = old;  
    new = x;  
    cc = CAS(qhead, old, new);  
until (cc == old;)
```

```
Pop ()  
do  
    old = qhead;  
    new = old->link;  
    cc = CAS(qhead, old, new);  
until (cc == old;)  
return old;
```

ABA 问题



thread 1:

pop()

thread 2:

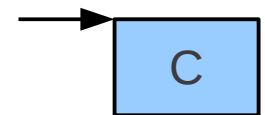
pop()

pop()

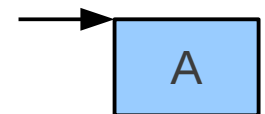
push(A)

正确执行:

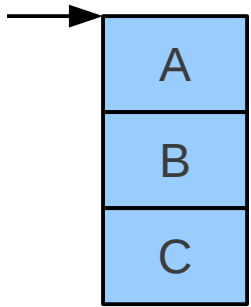
pop(); pop(); push(A); pop();



pop(); pop(); pop(); push(A);



ABA 问题



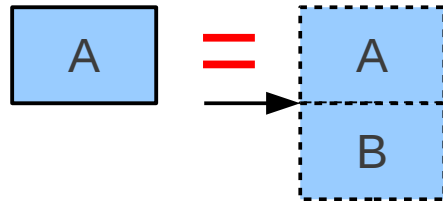
pop()

ret = A;

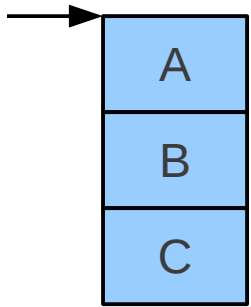
next = B;

// interrupted

CAS(A,A,B)



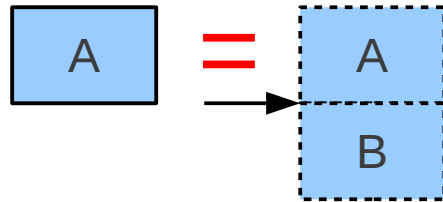
ABA 问题



pop()

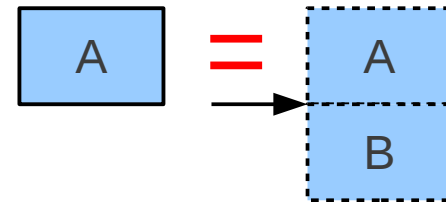
```
ret = A;  
next = B;
```

// interrupted
CAS(A,A,B)

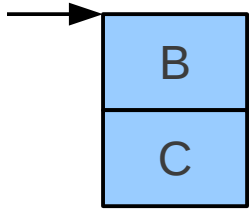


pop()

```
ret = A;  
next = B;  
CAS(A,A,B)
```



ABA 问题



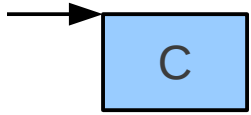
// interrupted
CAS(A,A,B)

pop()
ret = B;
next = C;
CAS(B,B,C)
return B;

ABA 问题

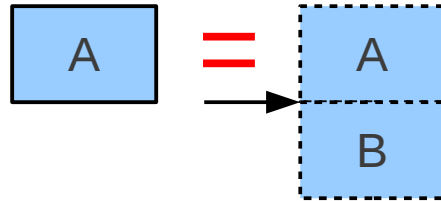
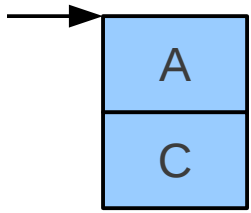
// interrupted
CAS(A,A,B)

push(A)
A->next = C;
CAS(C,C,A)



ABA 问题

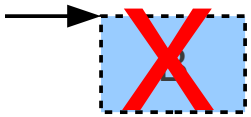
// resumes
CAS(A,A,B)



ABA 问题

// resumes

CAS(A,A,B)



Stack with DWCAS

```
struct elem {  
    elem *link;  
    any data;  
}
```

```
struct qhead {  
    elem *link;  
    int seq; ← version number  
} qhead;
```

```
Push(elem *x)  
do  
    old = qhead.link;  
    oldseq = qhead.seq;  
    x->link = old;  
    cc = DWCAS(qhead,  
                <old, oldseq>,  
                <x, oldseq+1>);  
until (cc);
```

```
Pop()  
do  
    old = qhead.link;  
    oldseq = qhead.seq;  
    new = old->link;  
    cc = DWCAS(qhead,  
                <old, oldseq>,  
                <new, oldseq+1>);  
until (cc);  
return old;
```

提纲

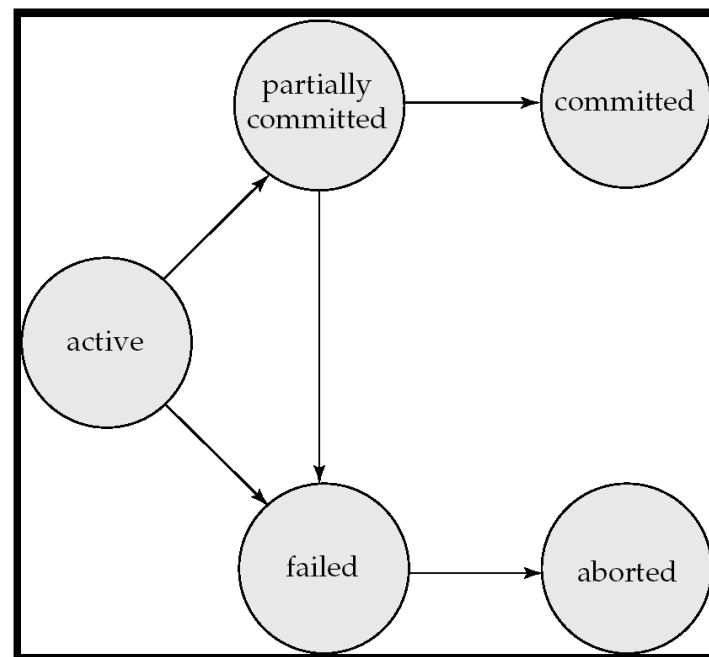
- 无锁编程概述
- 无锁编程实例
- 无锁编程研究问题
 - 无锁数据结构
 - Transactional Memory

事务内存

Transactional Memory

为什么叫 Transactional Memory

- 从 Database 的核心概念 Transaction 借鉴
- Database 一次 Transaction 过程
 - 1. Begin the transaction
 - 2. Execute several data manipulations and queries
 - 3. If no errors occur then **commit** the transaction and end it
 - 4. **If errors** occur then **rollback** the transaction and end it
- ACI 特征 (Atomicity, Consistency, Isolation)



Transaction State

Transactional Memory [TM M.K.]

- 1977 Lomet
 - (发现一种) 保证共享数据的抽象机制
- 1993 Herlihy and Moss, ISCA
 - Transactional Memory - architectural support for **lock-free** data structures
 - 支持无锁数据结构的机制
- ISCA, HPCA, PPOPP, PODC...
 - Challenge: to build an **efficient** TM infrastructure

Herlihy and Moss, ISCA 1993

[TM M.K.]

- **coined the term** *Transactional Memory*
- 增加 6 种新指令供程序员构建无锁数据结构
 - *load-transactional*: 读数据到私有寄存器
 - *load-transactional-exclusive*: 读数据到私有寄存器, 如果 cache-miss, 请求数据所有权
 - *store-transactional*: 写到内存 (cache) 但其他进程在 commit 之前不可见
 - *commit*: 提交内存更改
 - *abort*: 丢弃写更改
 - *validate*: 查询当前事务状态

Herlihy and Moss, ISCA 1993

[TM M.K.]

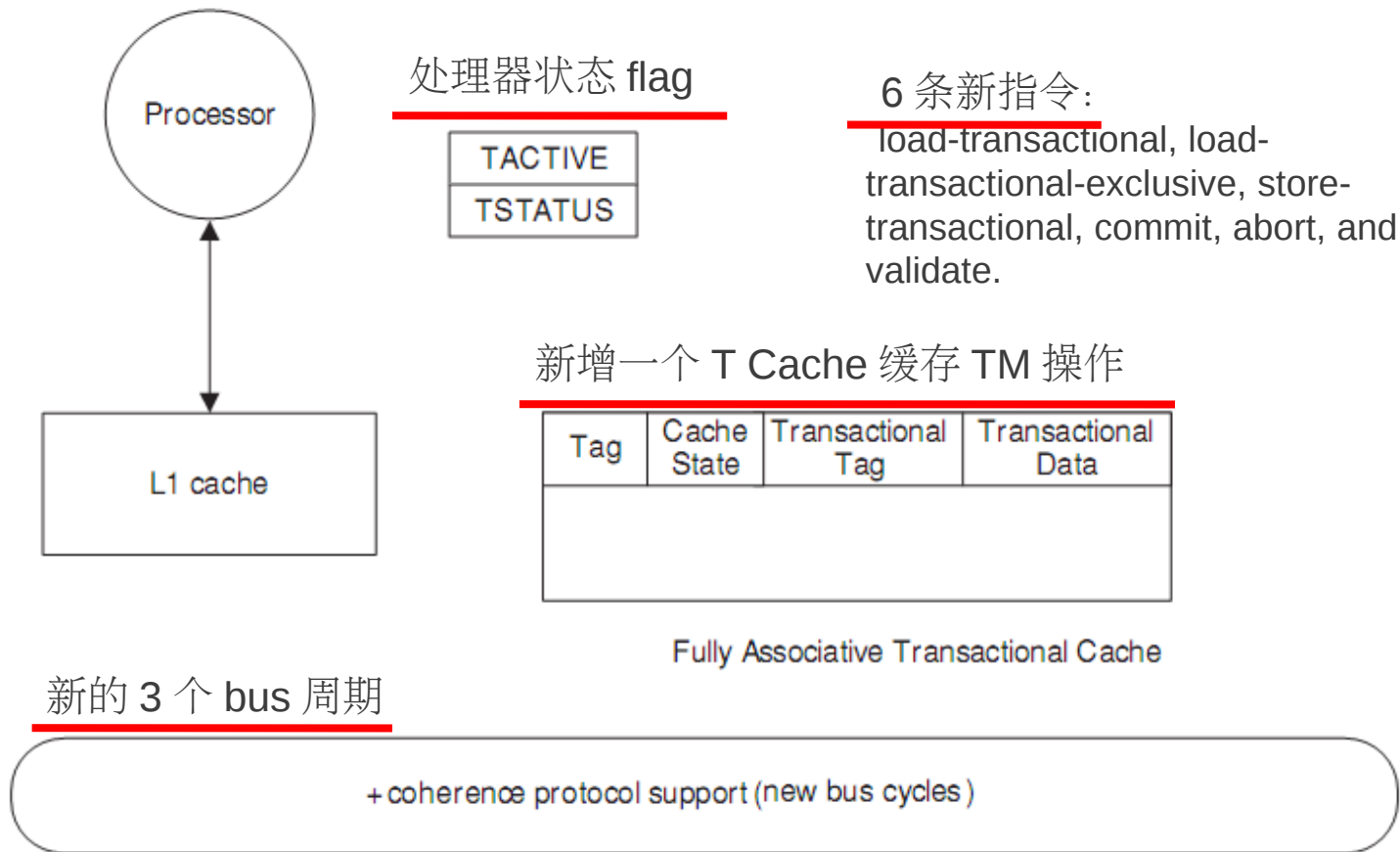


FIGURE 4.7: Herlihy and Moss Ttransactional Mmemory support

Transactional Memory

[TM M.K.]

- **TM 如何解决线程级并行?**
 - 程序员把计算 **wrap** 成 **transaction**
 - 不是万能药，但是把同步操作从程序员转移到编译器、运行环境和硬件
 - 目前主要的信心来自 **transaction** 解决了数据库的问题。尚没有用 **TM** 来解决一般的并行编程问题。
 - 目前还只是**语义级别**的 **TM** 定义（尚没有编译器、运行环境、库的支持）
 - **TM** 会和非 **TM** 的代码**并存**很长一段时间。 **TM** 的成功取决于旧代码和新代码的无缝结合。

小结

- 无锁编程可以避免使用锁的一些常见错误，并有可能带来性能提高
- 设计通用的无锁算法很难，目前常见的有一些无锁的数据结构
- 无锁的数据结构要求按照事务处理的方式编程
- 事务内存是对无锁的支持，目前研究很热，难点在于有效的实现

参考资料

- [DRD08] Dr Dobbs, Writing Lock-Free Code: A Corrected Queue, <http://drdobbs.com/high-performance-computing/210604448>
- [LLF10] 杨小华, 透过 Linux 内核看无锁编程, <http://www.ibm.com/developerworks/cn/linux/l-cn-lockfree/>
- [Maurice 08] Maurice Herlihy and Nir Shavit, The Art of Multiprocessor Programming
- [TM M. K.] James R. Larus and Ravi Rajwar, Transactional Memory, published by Morgan Kaufman

参考资料

- [Herlihy 1991] Maurice Herlihy, Wait-free Synchronization, 1991
- [NSYNC] Nonblocking synchronization bibliography, <http://www.cs.wisc.edu/trans-memory/biblio/swnbs.html>
- [Langdale05] Geoff Langdale, Lock-Free Programming, www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf
- [Lee10] Patrick P.C. Lee, A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring

参考资料

- [CS380D]Wait-Free Synchronization Lecture, CS380D—Distributed Computing, The University of Texas at Austin