CPU Cache Coherence & Memory Consistency

何登成

个人简介

- 何登成

- 联系方式
  - 新浪微博：　　　　何_登成 (http://weibo.com/u/2216172320)

内容简介

- 关注CPU的两个核心功能
  - Cache Coherence
  - Memory Consistency
- Definition

  Many modern computer systems and most **multicore chips** support **shared memory** in hardware. In a shared memory system, each of the processor cores may read and write to a single shared address space. For a shared memory machine, the memory consistency model defines the architecturally visible behavior of its memory system. Consistency definitions provide rules about loads and stores (or memory reads and writes) and how they act upon memory. As part of supporting a memory consistency model, many machines also provide cache coherence protocols that ensure that multiple cached copies of data are kept up-to-date.

  - *A primer on Memory Consistency and Cache Coherence (Synthesis Lectures on Computer Architecture)*

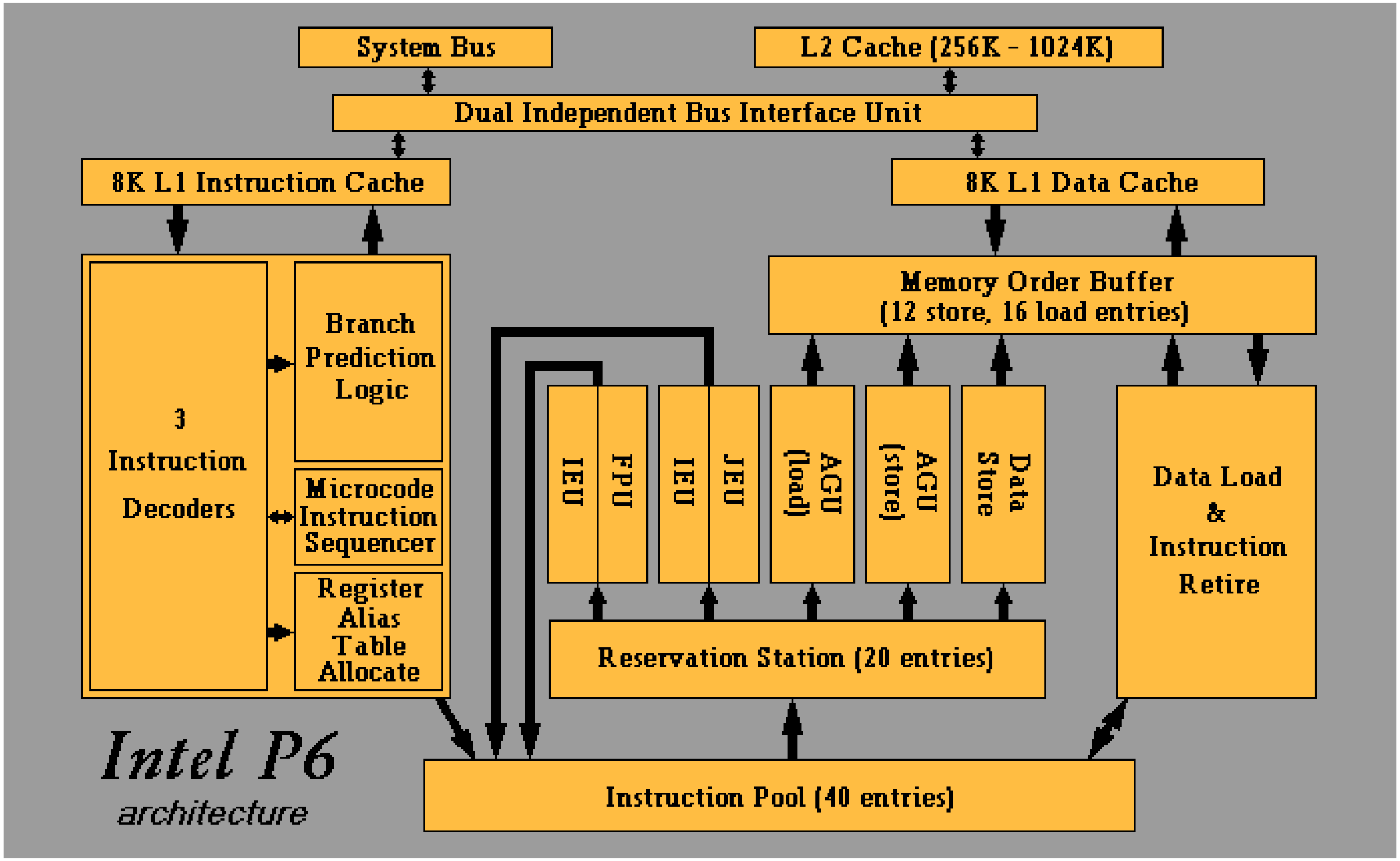
Outline

- CPU Cache
  - Cache Structure
  - Cache Coherence

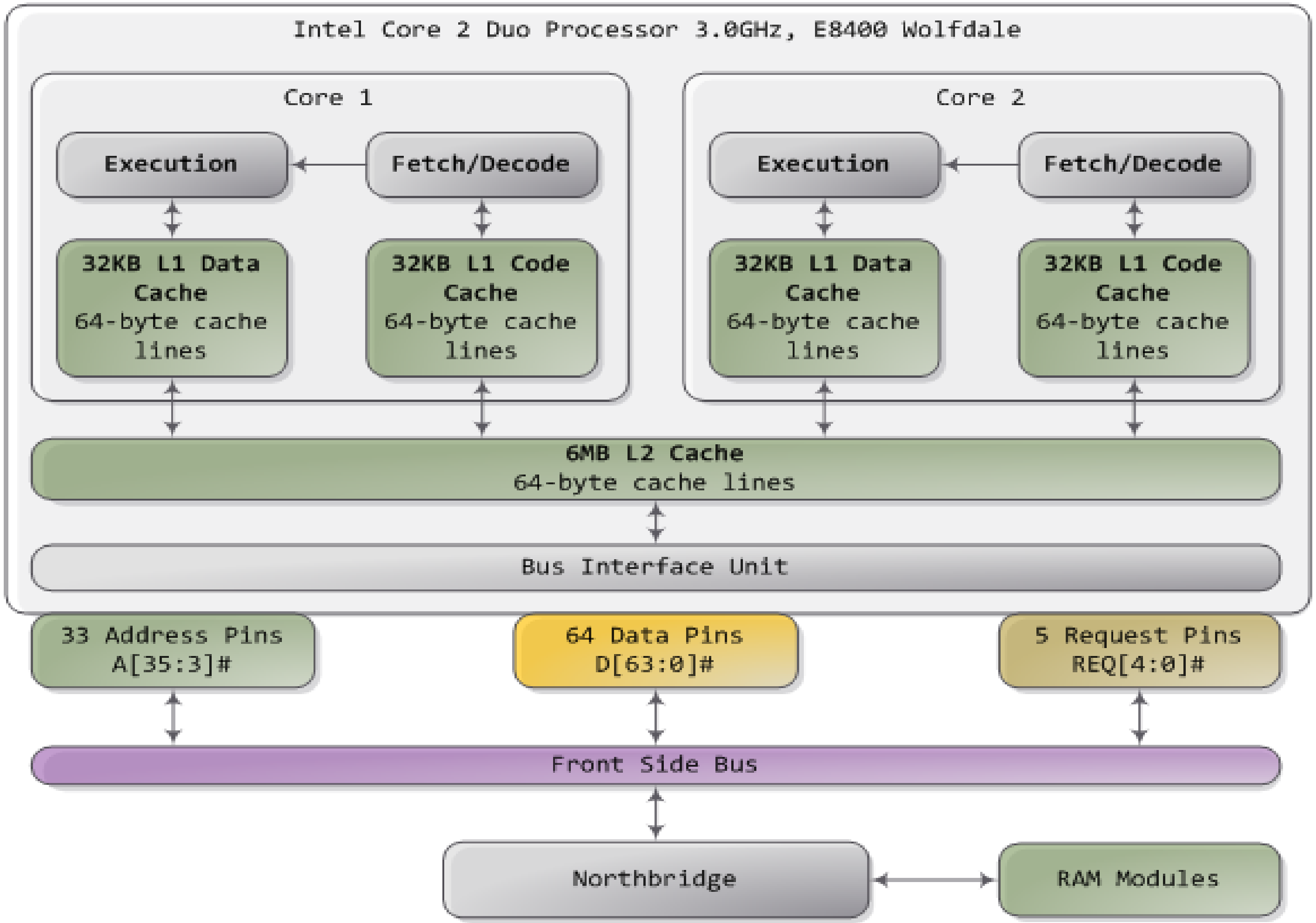
- Memory Consistency
  - Atomic vs Reorder
  - Memory Barrier
    - Compiler Memory Barrier
    - CPU Memory Barrier
  - Load Acquire vs Store Release

- 并发程序设计
  - Implement a spin lock
  - 一个真实案例
  - Others

CPU Architecture(复杂版)



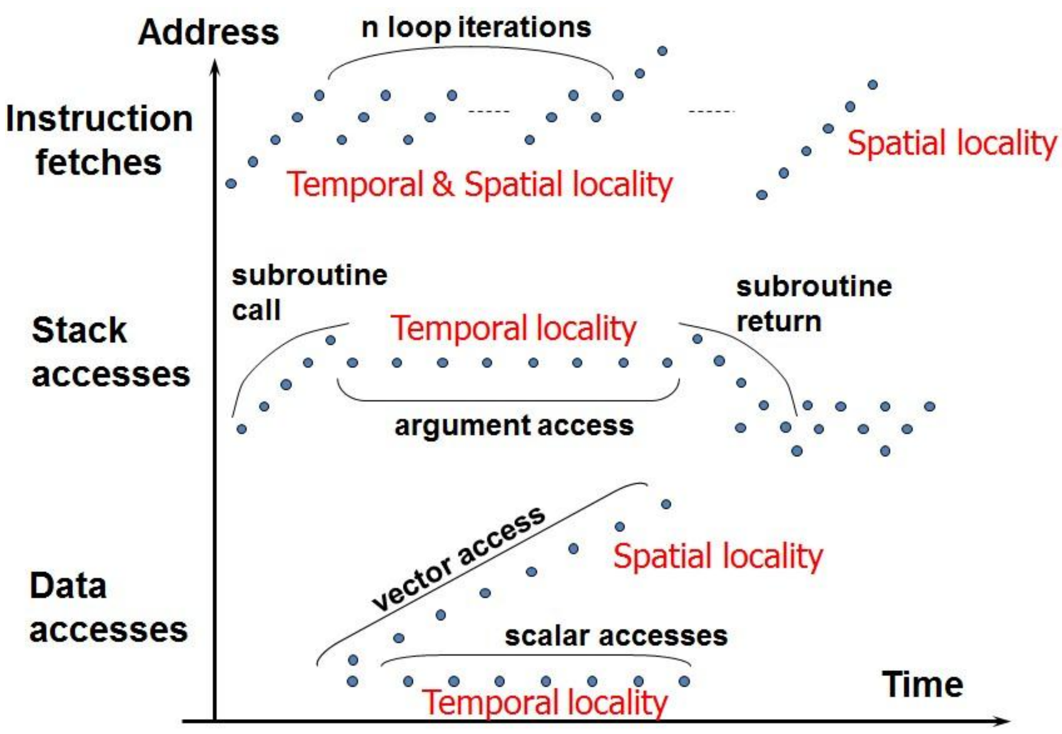
CPU Architecture(简化版)



http://duartes.org/gustavo/blog

CPU Cache

- What is a cache?
  - Small, fast storage used to improve average access time to slow memory.
- Cache原理 (Memory Access Pattern)
  - Spatial Locality
  - Temporal Locality

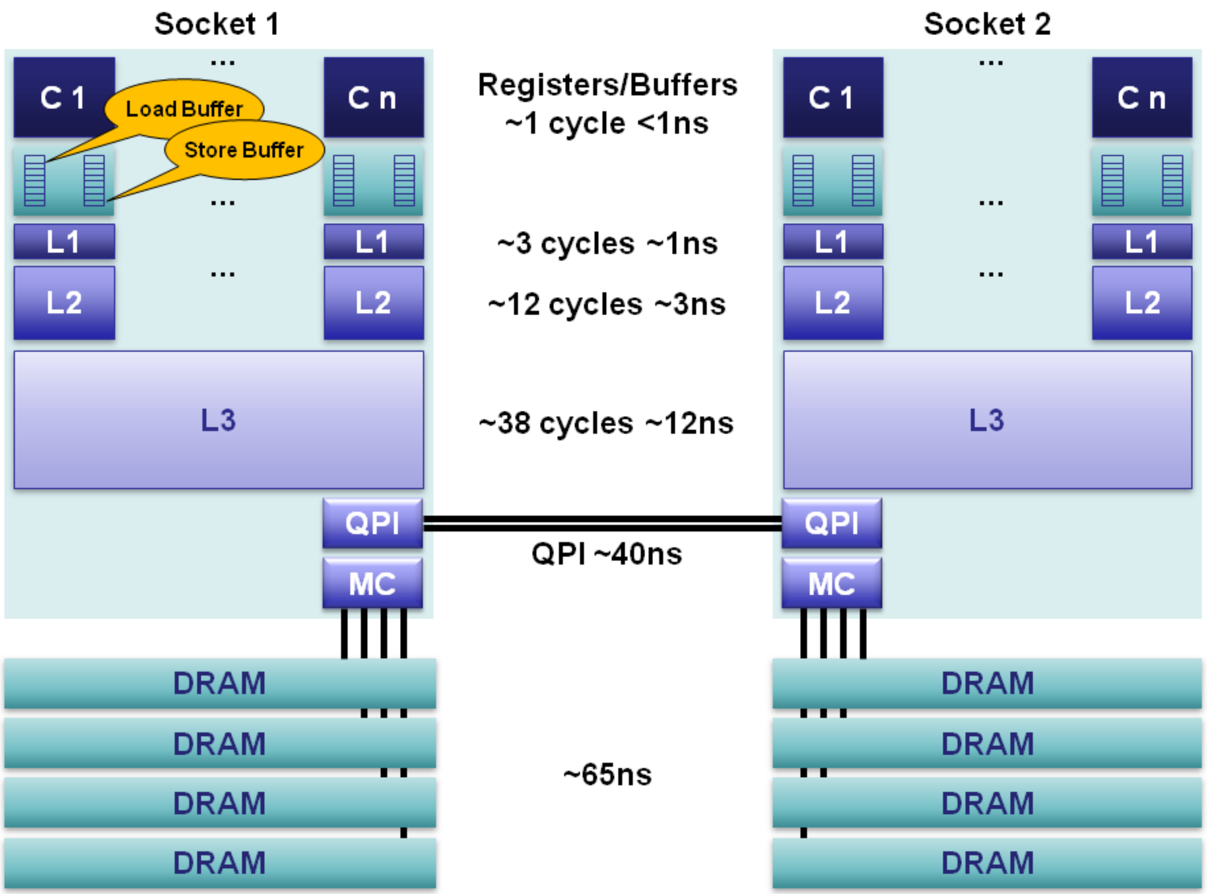


Cache Hierarchy

- Multi-Level of Cache
  - Nehalem (Three-Level)
    - L1(Per-Core)：32 KB D-Cache；32 KB I-Cache；
    - L2(Per-Core)：256 KB；
    - L3(Shared)： 8M；
    - How to Test Cache Size？
      - Igor’s Blog (Example 3)

  - Pentium(R) Dual-Core CPU E5800(Two-Level)
    - 本人PC机
    - 32 KB L1 Data Cache；32 KB L1 Instruction Cache；
    - 2 MB L2 Cache；(Unified Cache)

Cache Performance



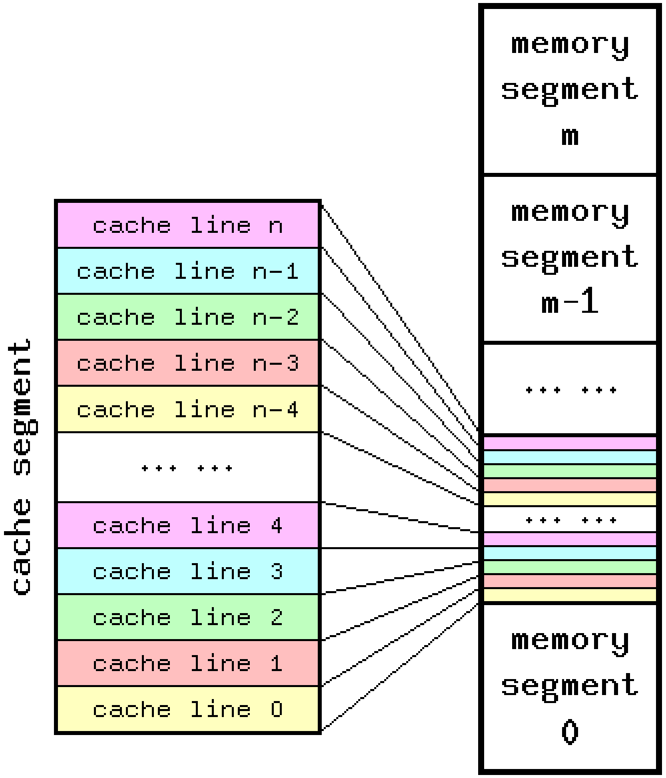
Cache Line

- **The minimum amount of cache which can be loaded or stored to memory**

- X86 CPUs
  - 64 Bytes

- ARM CPUs
  - 32 Bytes

- Cache Line Size Testing
  - [Igor's Blog]: Example 2



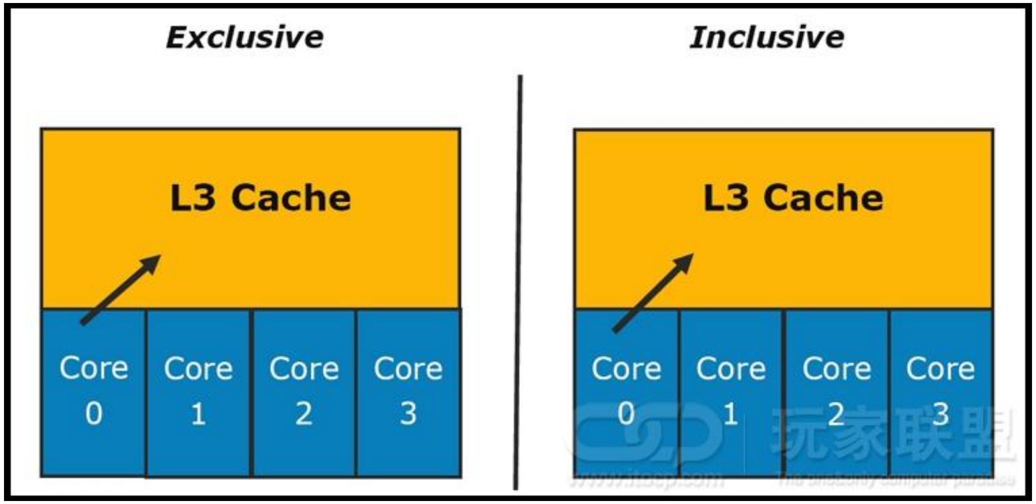
Cache Policy

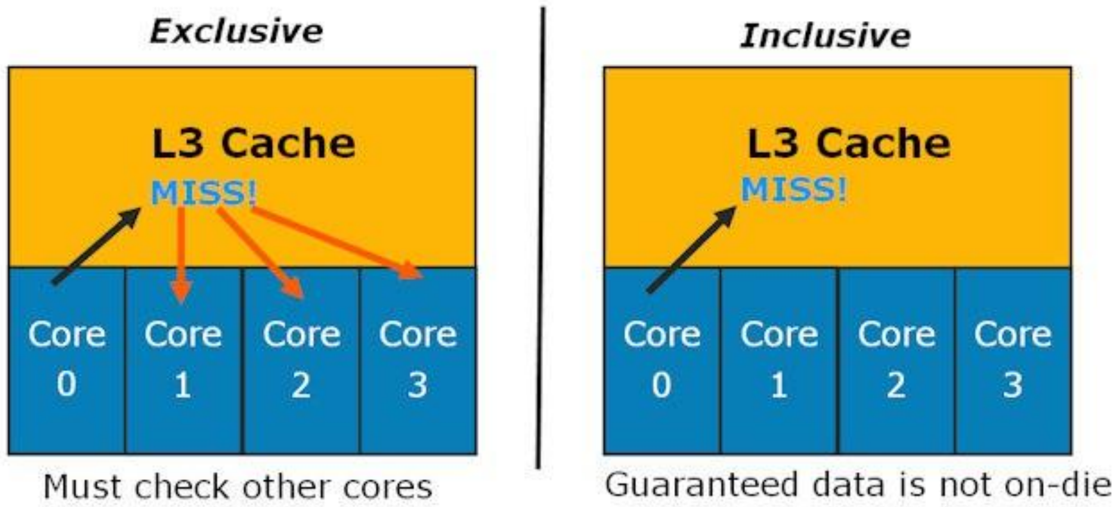
- Inclusive Multilevel Cache
  - 外层Cache包含内层Cache的所有数据；
  - 外部访问，只需访问最外层的Cache(L3)；
  - 最常见形式

- Exclusive Multilevel Cache
  - 外层Cache可能不包含内存Cache的数据；
  - 外部访问，需要遍历所有层级Cache(L1/L2/L3)，寻找记录；

- 选择原则
  - 空间与效率之间的平衡；
  - Inclusive：浪费空间；效率高；
  - Exclusive： 节约空间；效率低；

Cache Policy(续)





Cache Structure

- **Large caches are implemented as hardware hash tables with fixed-size hash buckets (or “sets”) and no chaining.**

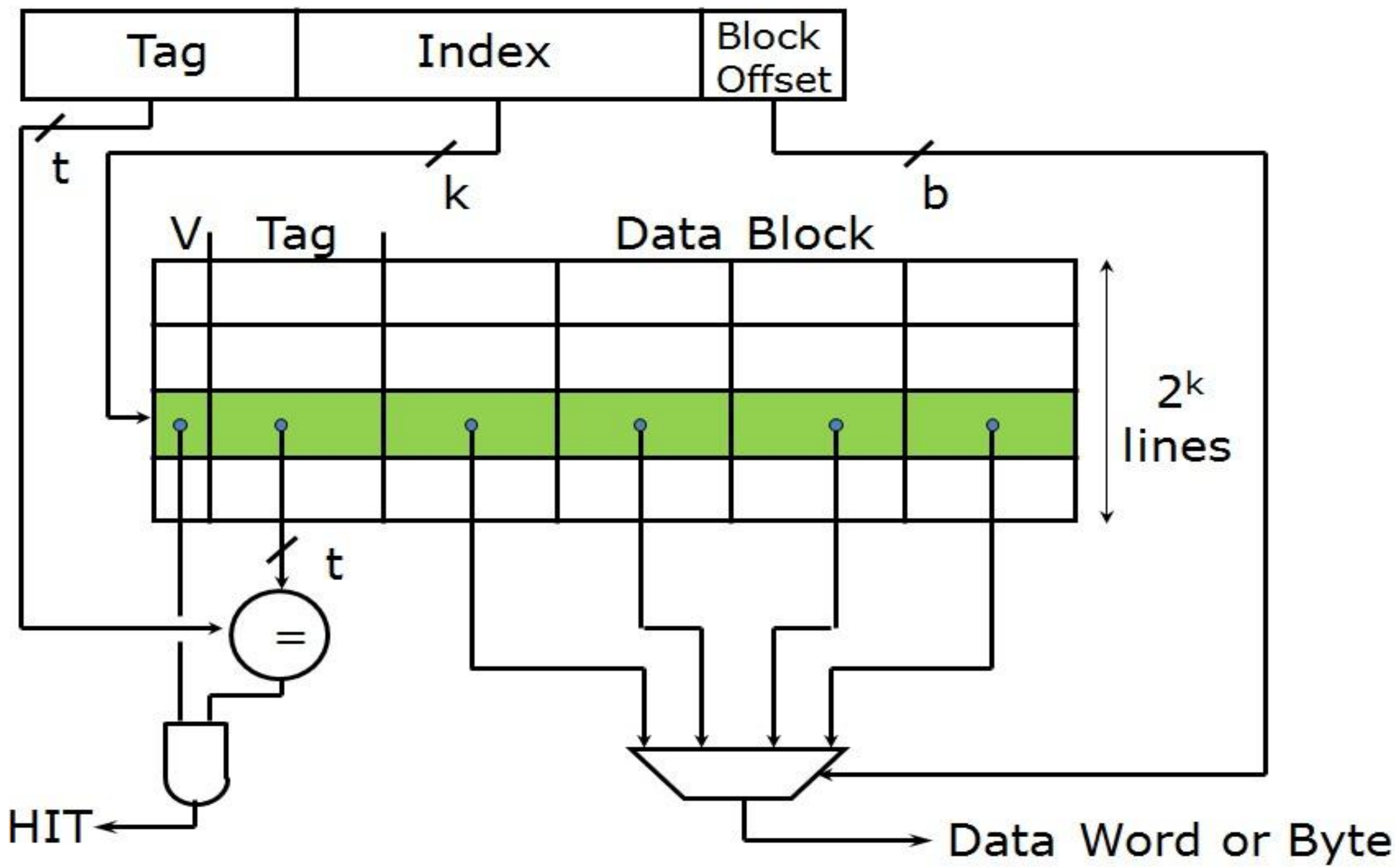
- sets
  - hardware hash tables中的hash入口数量；

- ways
  - 每个hash入口能够存储的项数量；

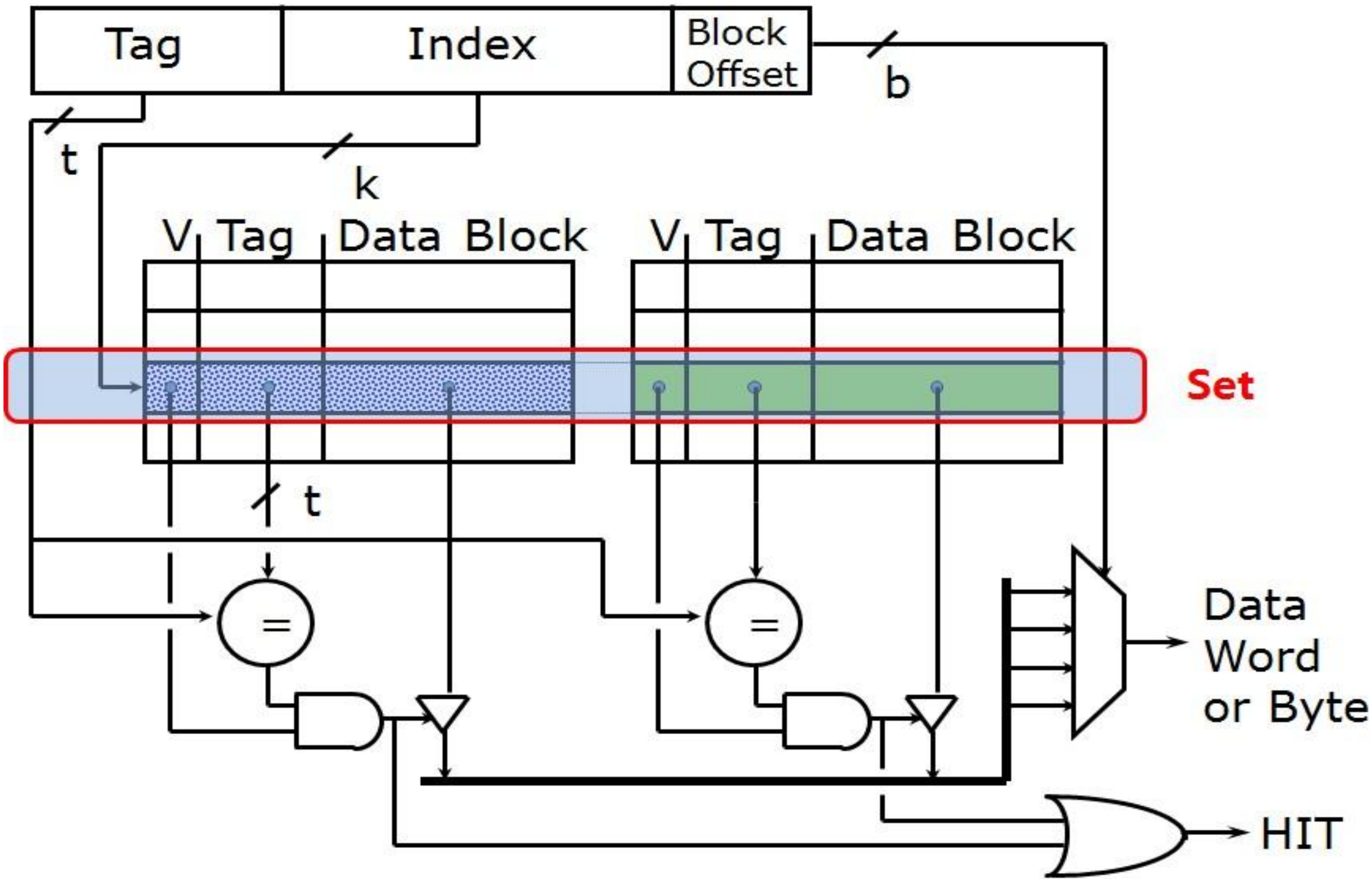
- N-way set associative cache
  - N = 1
    - Direct-Mapped Cache
  - N = 8
    - 8-way set associative cache
  - N = cache size / cache line size
    - full associative cache

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

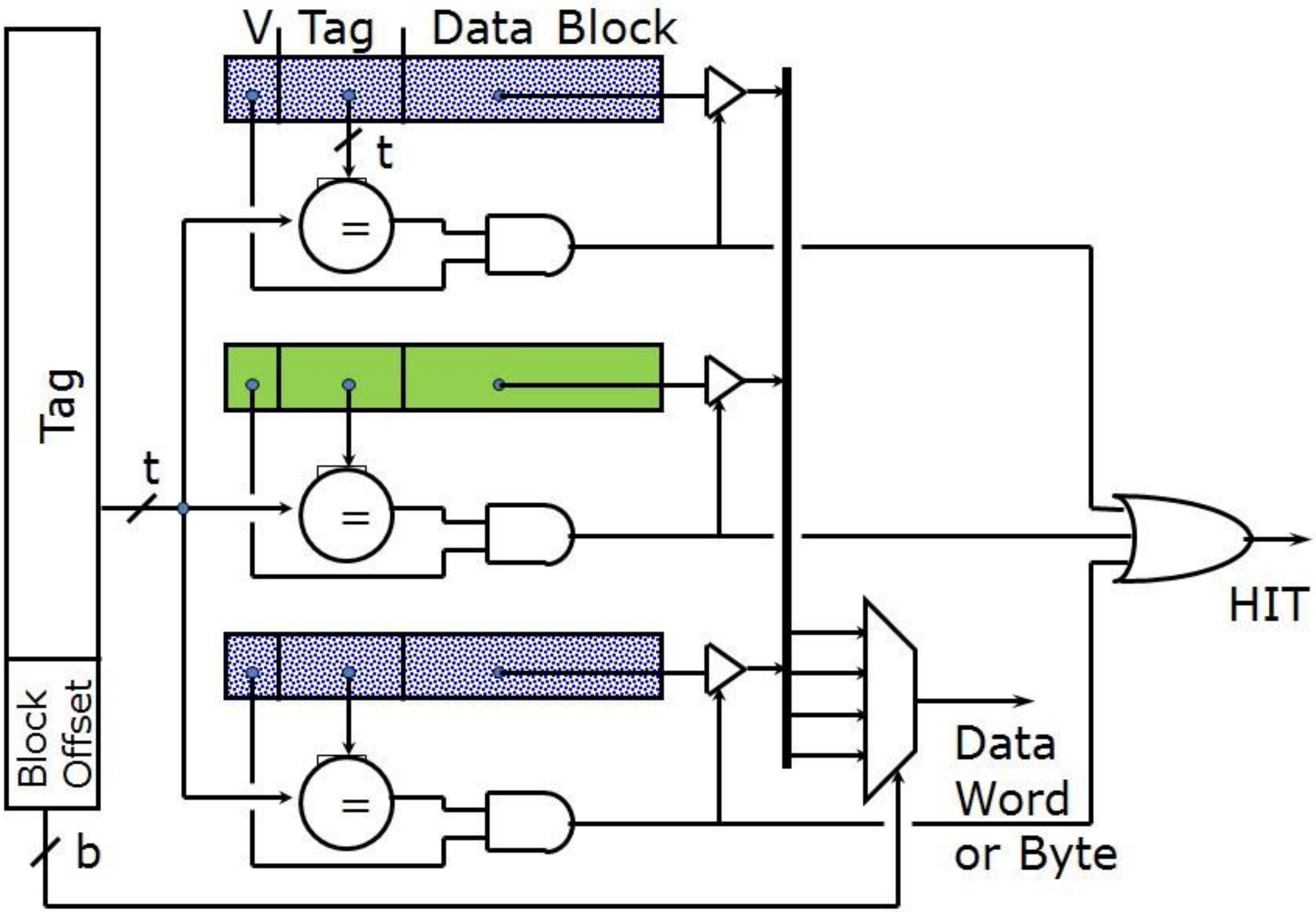
Direct-Mapped Cache



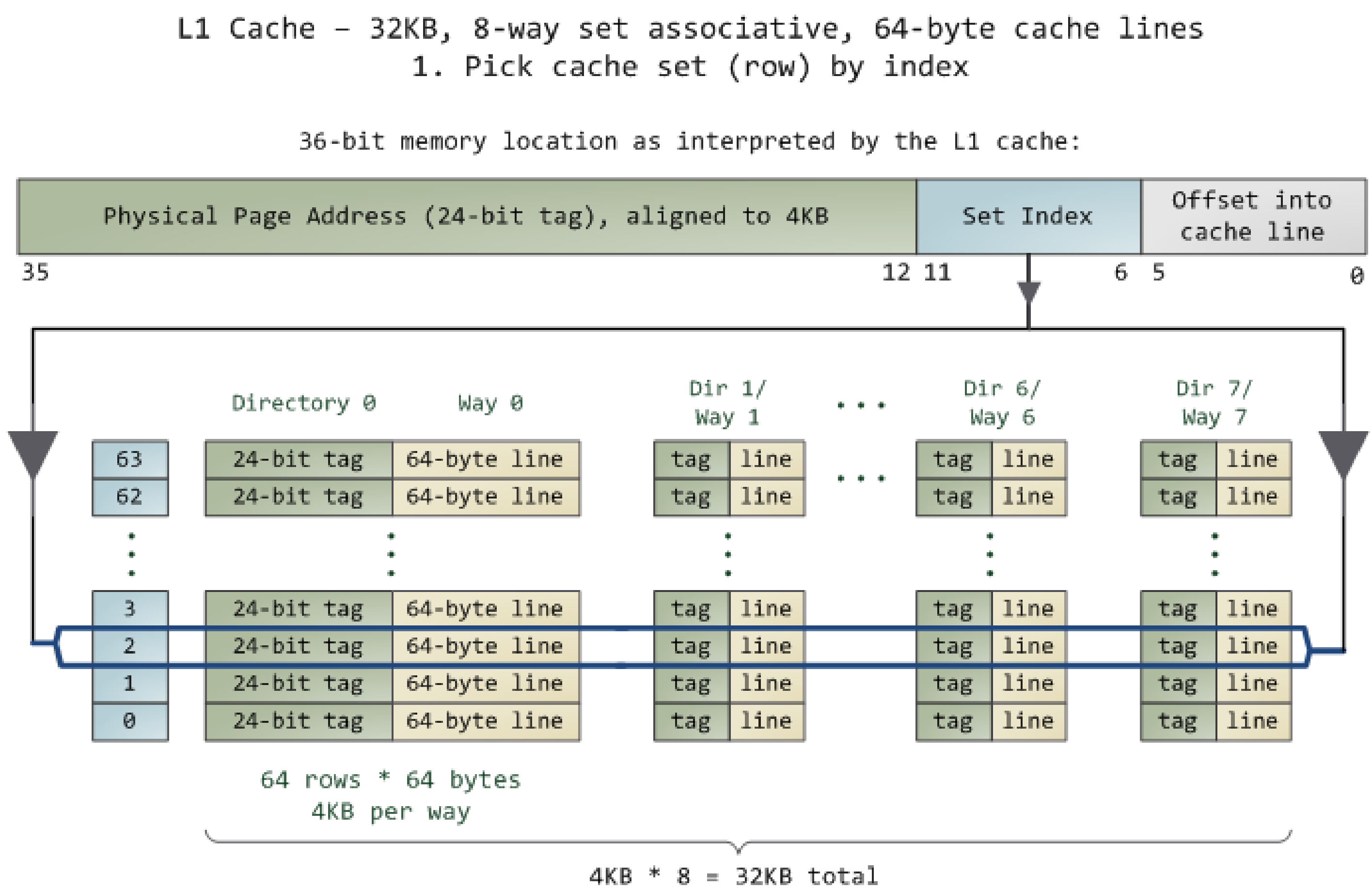
2-Way Associative Cache



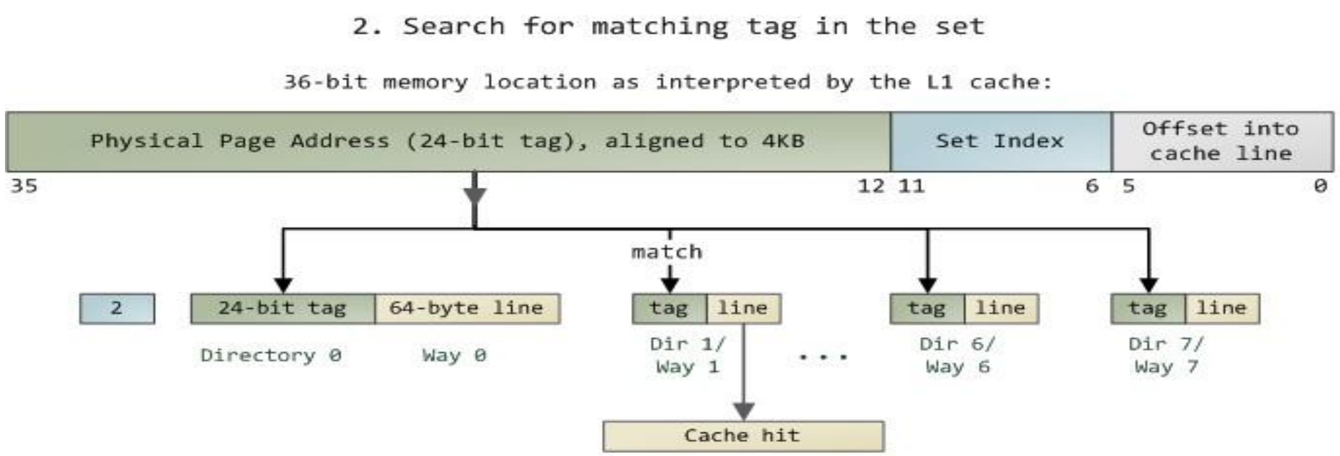
Full Associative Cache



Cache Line Locate (1)



Cache Line Locate (2)



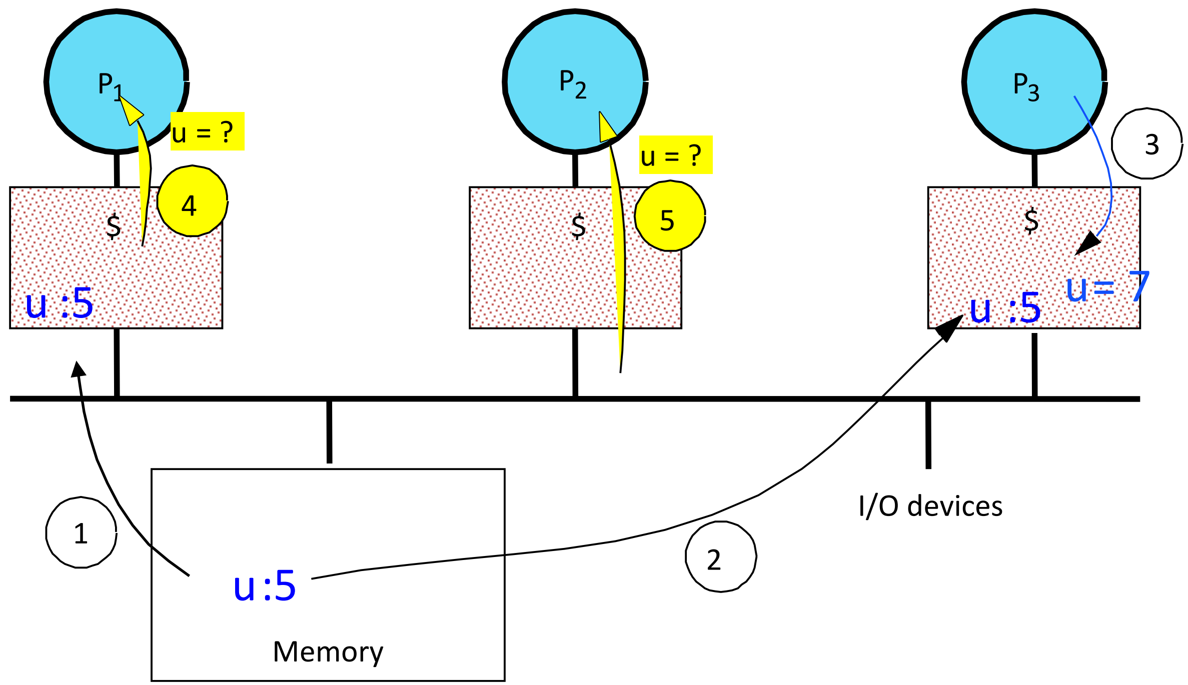
- Virtual Address vs Physical Address
  - A memory access usually starts with a linear (virtual) address
- tag vs index
  - index： used to locate set entry;
  - tag： used to find the correct cache line in the set;
- PIPT vs VIPT
  - PIPT： **Physically indexed， physically tagged**
  - VIPT： **Virtually indexed， physically tagged (L1 Cache)**

Cache Associative分析

- Direct-Mapped Cache
  - 定位最快，冲突最严重；
- 2/4/8-Way Associative Cache
  - N值越大，冲突越低，定位越慢；
- Full Associative Cache
  - 冲突最低，定位最慢；
- 我的个人电脑
  - 8-Way Associative L2 (2MB)
  - 每个set大小
    - 64 * 8 = 512 Bytes；
  - Sets = 2 MB/512 Bytes = 4096;
    - 每隔4096 * 64B = 256KB的地址的Cache Line，就会在同一个Set中；

```
Logical Processor to Cache Map:
*-  Data Cache          0, Level 1,   32 KB, Assoc   8, LineSize  64
*-  Instruction Cache   0, Level 1,   32 KB, Assoc   8, LineSize  64
-*  Data Cache          1, Level 1,   32 KB, Assoc   8, LineSize  64
-*  Instruction Cache   1, Level 1,   32 KB, Assoc   8, LineSize  64
**  Unified Cache       0, Level 2,    2 MB, Assoc   8, LineSize  64
```

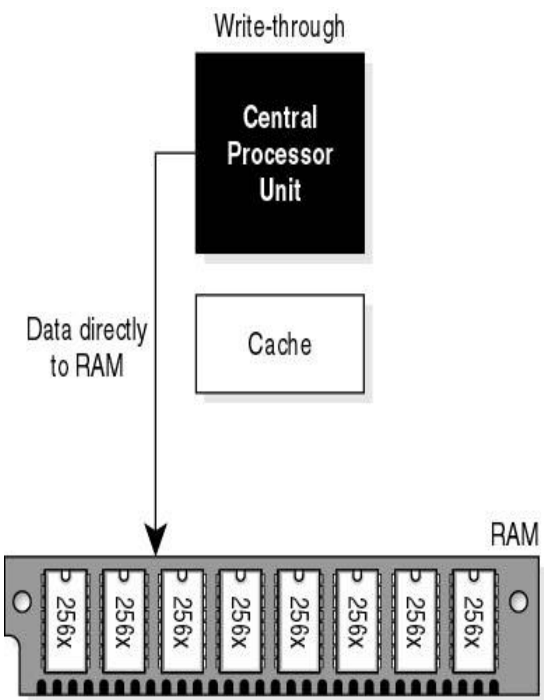
Cache Coherence Problem

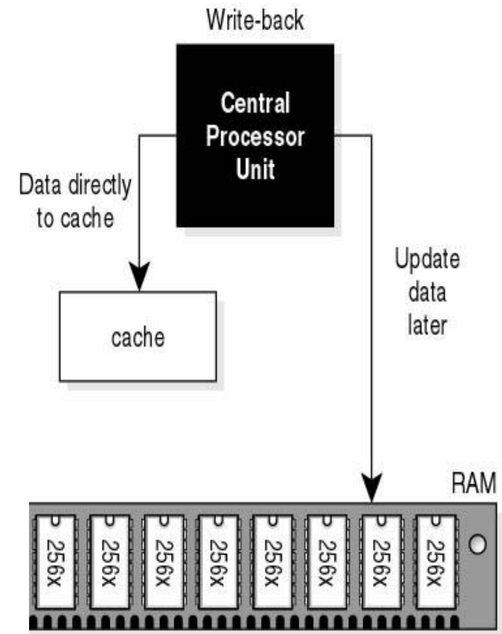


- Assumption: Write back scheme
- Problem:
  - Processors see different values for u after event 3

Cache Write Policy

- Write Back vs Write Through
  - Write Back
    - 脏数据，写出到Cache；
    - Write Miss
      - Read Cache Line；
      - Write Allocate；
  - Write Through
    - 脏数据，写穿到Memory;
    - Write Hit
      - 更新Cache；
    - Write Miss
      - 绕过Cache，直接写memory;



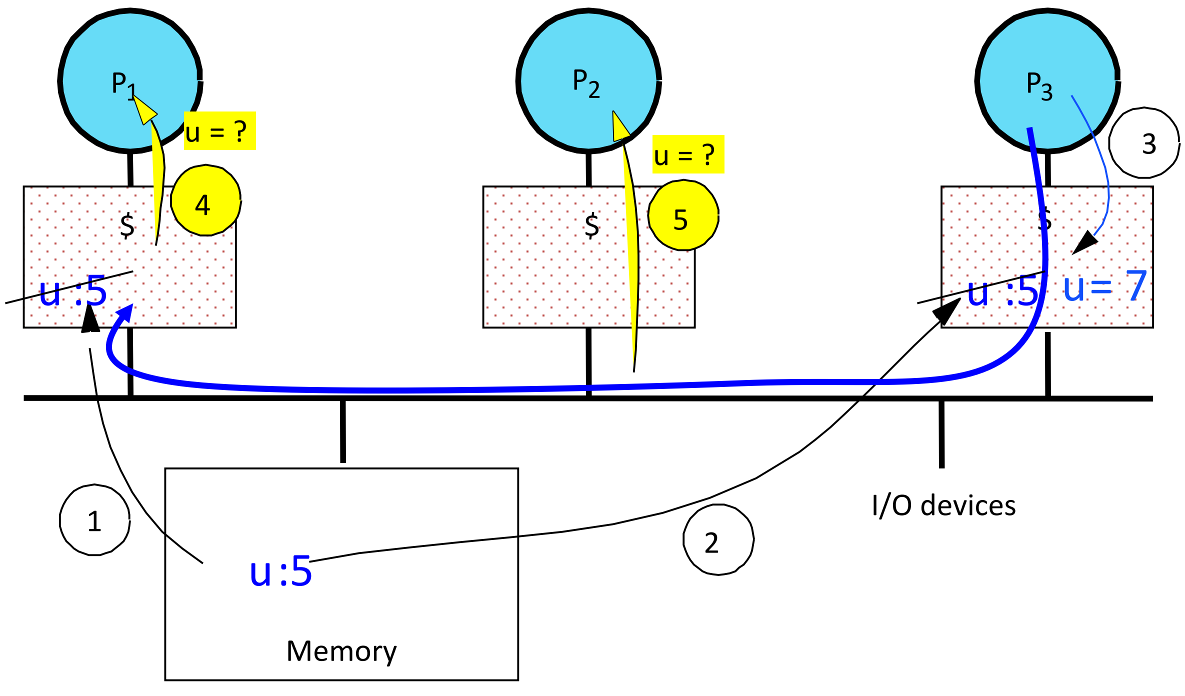


Cache Write Policy(续)

- Write Invalidate vs Write Update
  - Write Invalidate
    - Write时，同时发出Invalidate消息，使得所有其他CPU L1/L2 Cache中同一Cache Line失效；
    - 优势：实现简单；
    - 不足：会导致其他CPU再次读取时出现Cache Miss；
  - Write Update
    - Write时，同时更新其他CPU L1/L2 Cache中同一Cache Line；
    - 优势：对应write Invalidate的不足；
    - 不足：对应write invalidate的优势；
  - 选择
    - 目前，基本都是Write Invalidate方式

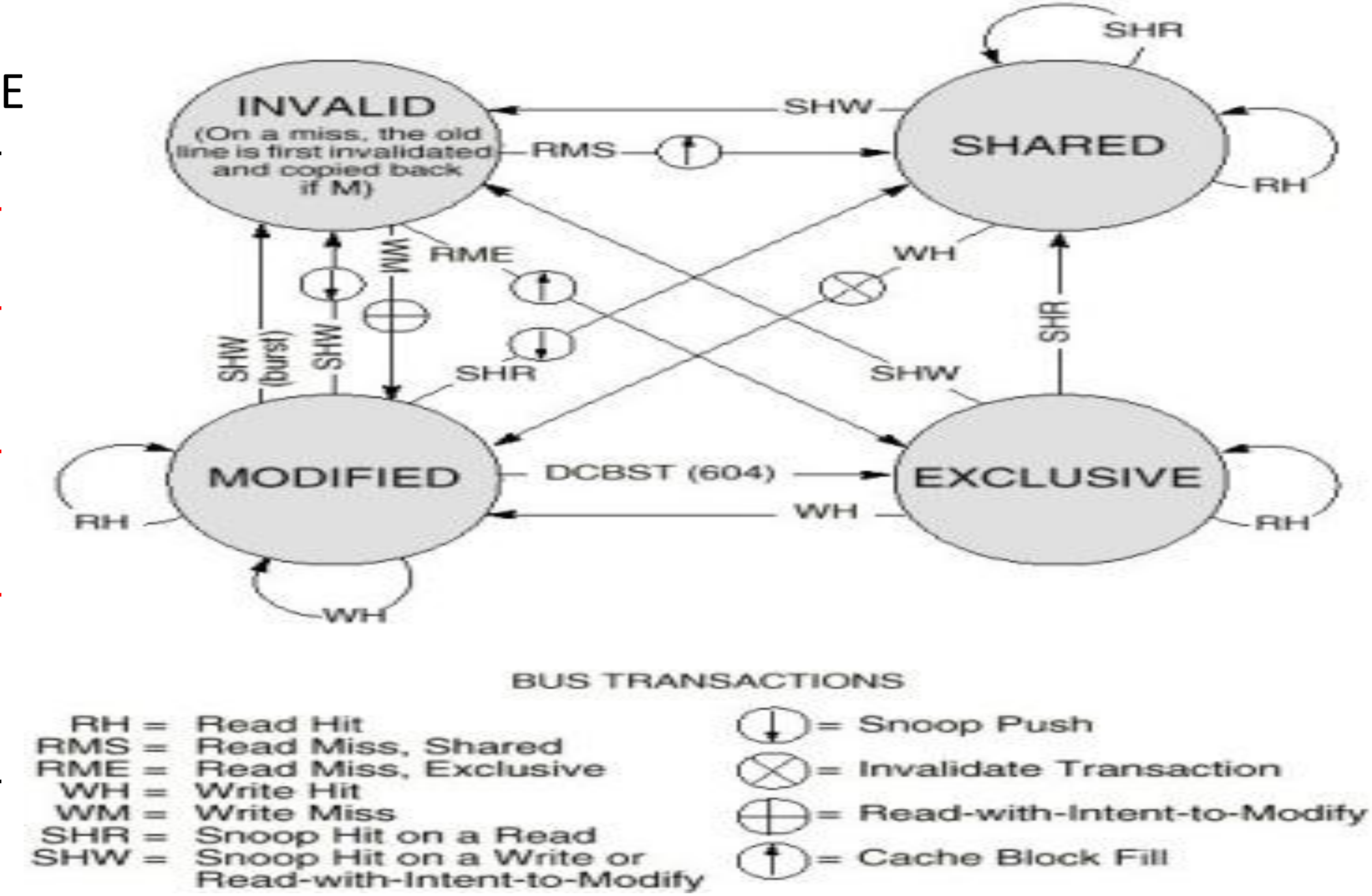
Solve Cache Coherence Problem(1)

- Write-Through Invalidate



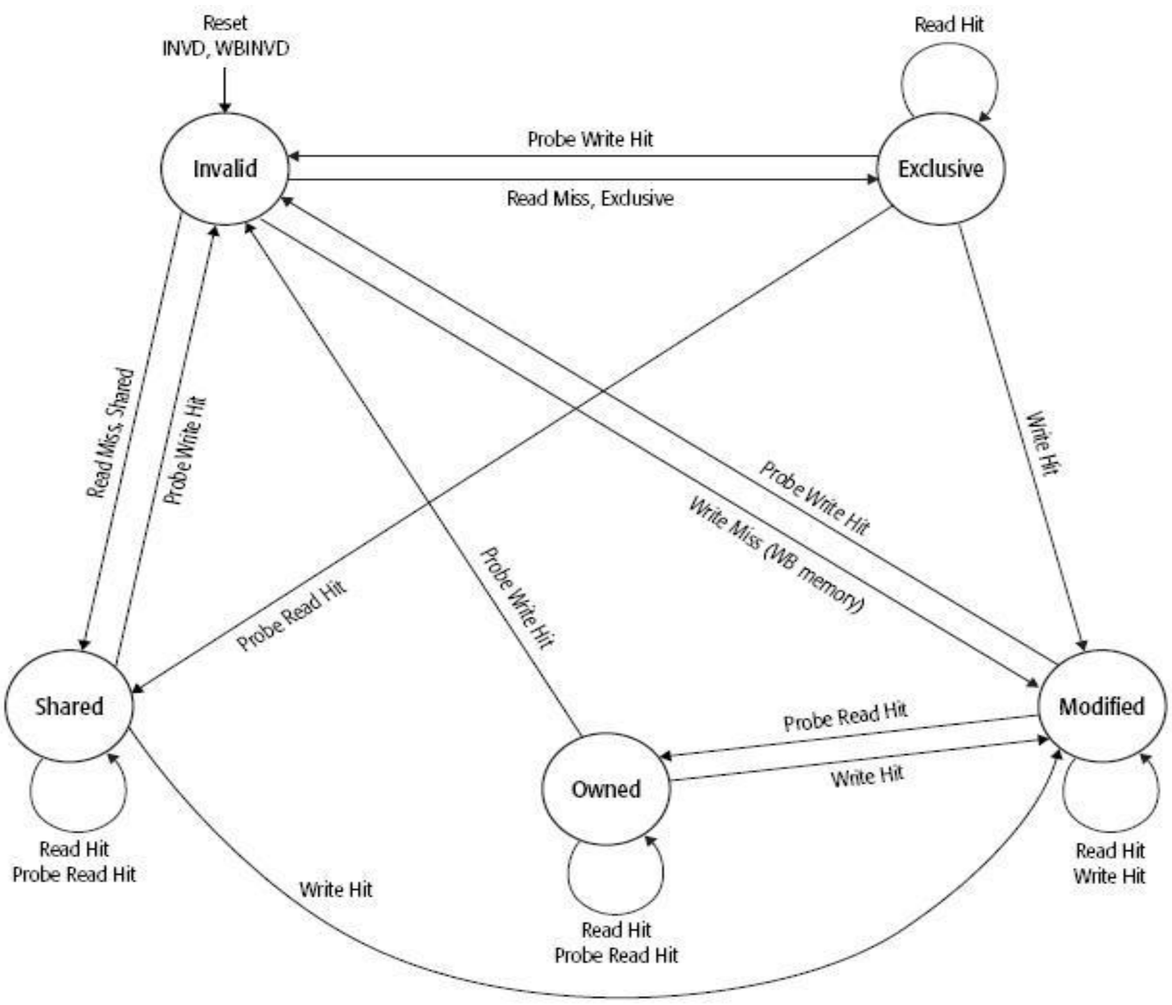
Solve Cache Coherence Problem(2)

- ME



Solve Cache Coherence Problem(3)

- MO
  - –
  - –
  - –
  - –
  - –
  - –
  - –



Cache Coherence

- 注意
  - 作用域

    Cache Coherence Protocal (MESI, MOESI)， 作用于CPU Cache与Memory层面，若操作的数据在Register， 或者是Register与L1 Cache之间(后续提到的Store Buffer， Load Buffer)， 则这些数据不会参与Cache Coherence协议；

  - Message

    Cache Coherence协议中的Message，是由汇编指令触发的。一条高级语言(C/C++)，可能会被编译为多条汇编指令 (例如：a++ 至少被编译为3条汇编指令)；

    一条汇编指令，可能会发出多条Messages。例如：一个Write操作，如果Cache Miss，会发出多条Messages：Read + Invalidate + ...

Any Question about CPU Cache?

Memory Consistency

- Memory Consistency
  - ***Memory Consistency模型，是整个并发程序设计的基础***。
  - 并发程序设计分为4个阶段：
    - 阶段一：知道什么是Spinlock，什么是Mutex，也知道访问共享资源需要进行保护；
    - 阶段二：知道如何实现一个高性能的Spinlock，Mutex，以面对不同的需求；
    - 阶段三：知道Spinlock，Mutex实现的内部原理是什么？为什么可以用来保护共享资源；
    - 阶段四：在熟练使用锁的基础上，追求高性能，尝试Lock-Free编程；
  - 而为了从阶段一，晋级到阶段二，三，甚至是阶段四，离不开对于Memory Consistency模型的深入理解。本PPT关于Memory Consistency的内容，按照如下方式组织：
  - Atomic vs Reorder
    - 讨论什么是Atomic Operation？讨论程序有哪些乱序行为？
  - Memory Barrier
    - 何谓Memory Barrier？Memory Barrier有哪些种类？Memory Barrier如何使用？
  - Load Acquire vs Store Release
    - Load Acquire与Store Release，是什么意思？

Atomic Operation

- An operation acting on shared memory is **atomic** if it completes in a single step relative to other threads. When an atomic store is performed on a shared variable, no other thread can observe the modification half-complete. When an atomic load is performed on a shared variable, it reads the entire value as it appeared at a single moment in time.

- Atomic Operation in CPU
  - Intel CPU

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

  - AMD CPU

Atomicity of accesses. Single load or store operations (from instructions that do just a single load or store) are naturally atomic on any AMD64 processor as long as they do not cross an aligned 8-byte

高级语言与汇编指令的映射

- 高级语言(如：C/C++)，被编译为汇编语言，才能够被执行。因此，高级语言与汇编语言之间，存在着几种简单的映射关系。

- Simple Write
  - Write to Memory
  - Atomic

```
int a = 1;
0034146E  mov         dword ptr [a],1
```

- Simple Read
  - Read from Memory
  - Atomic

```
int b;

b = a;
00341475  mov         eax,dword ptr [a]
00341478  mov         dword ptr [b],eax
```

- Read-Modify-Write(RMW)
  - Read from Memory
  - Modify
  - Write to Memory

  - Non-Atomic

```
a++;
0034147B  mov         eax,dword ptr [a]
0034147E  add         eax,1
00341481  mov         dword ptr [a],eax
```

Non-Atomic Operations

- Examples
  - Read/Write 64 Bits on 32 Bits Systems
    - Write： Non-Atomic
    - Read： Non-Atomic

```
    29:      uint64_t c;
    30:      c = 0x100000002;
00E33684  mov          dword ptr [c],2
00E3368B  mov          dword ptr [ebp-2Ch],1
    31:
    32:      uint64_t d = c;
00E33692  mov          eax,dword ptr [c]
00E33695  mov          dword ptr [d],eax
00E33698  mov          ecx,dword ptr [ebp-2Ch]
00E3369B  mov          dword ptr [ebp-3Ch],ecx
```

  - RMW Operations
    - Non-Atomic

```
    a++;
0034147B  mov          eax,dword ptr [a]
0034147E  add          eax,1
00341481  mov          dword ptr [a],eax
```

Non-Atomic Operations(续)

- Questions?
  - 32位系统，是否4 Bytes的Simple Read/Write一定是Atomic？
  - 64位系统，是否8 Bytes的Simple Read/Write一定是Atomic？
- Exceptions
  - Intel486 and newer [参考Igor文章的Example 3，查询CPU类型工具]
    - Unaligned 16-, 32-bit access;
  - Pentium and newer
    - Unaligned 16-, 32-, 64-bit access;
  - P6 and newer
    - Cross cache line access; (注意：P6 and newer CPU，允许Atomic Unaligned access)
  - AMD
    - Unaligned 16-, 32-, 64-bit access;
  - ARM
    - strd instruction...

Non-Atomic的危害

- Half Write
  - mov dword ptr [c], 2执行后，会短暂出现c的half write现象；
- Half Read
  - 若c出现half write，则读取c会出现half read现象；

```
    29:        uint64_t c;
    30:        c = 0x100000002;
00E33684  mov           dword ptr [c],2
00E3368B  mov           dword ptr [ebp-2Ch],1
    31:
    32:        uint64_t d = c;
00E33692  mov           eax,dword ptr [c]
00E33695  mov           dword ptr [d],eax
00E33698  mov           ecx,dword ptr [ebp-2Ch]
00E3369B  mov           dword ptr [ebp-3Ch],ecx
```

- Composite Write
  - 两个线程同时write c，一个完成，一个half write，则c的值，来自线程1，2两个write操作的组合；
- 危害
  - 出现Half Read，会导致程序判断逻辑出错；
  - 出现Composite Write，会导致数据出错；

Atomic Instructions and Lock(1)

- Atomic Instructions
  - 常见指令：CMPXCHG，XCHG，XADD，...
  - CMPXCHG (compare-and-exchange)

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA

- 将Operand 1(Reg/Mem)中的内容与EAX比较，若相等，则拷贝Operand 2(Reg)中的内容至Operand 1；若不等，则将Operand 2中的数据写入EAX；
- 一个Atomic RMW操作，若Operand 1为Memory，则CMPXCHG指令还需要Lock指令配合 (Lock prefix)；

Atomic Instructions and Lock(2)

- Lock Instruction

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F0	LOCK	NP	Valid	Valid	Asserts LOCK# signal for duration of the accompanying instruction.

  - Lock指令是一个前缀，可以用在很多指令之前，代表当前指令所操作的内存(Memory)，在指令执行期间，只能被当前CPU所用；
  - **Question**: 若指令没有操作内存，那么Lock前缀还有意义吗？
  - Intel’s Description about Lock Instruction

Causes the processor’s LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

  - Lock with CMPXCHG

```
case __X86_CASE_L:
{
        volatile u32 *__ptr = (volatile u32 *)(ptr);
        asm volatile(lock "cmpxchgl %2,%1"
                     : "=a" (__ret), "+m" (*__ptr)
                     : "r" (__new), "" (__old)
                     : "memory");
        break;
}
```

Non-Atomic消除

- 如何消除Non-Atomic Read/Write？
  - 平台方面 (参考各CPU白皮书)
    - Intel/AMD CPU
      - Aligned 2-，4-Byte Simple Read/Write → Atomic
      - Aligned 8-Byte，CPU型号 → 一般为Atomic
      - Unaligned 2-, 4-, 8-Byte，CPU型号判断 → 尽量少用
    - 其他
  - RMW Operation
    - 尽量使用系统自带的，或者是提供的原子操作函数；这些函数，对不同CPU类型，做了较好的封装，更加易用；
    - Windows Synchronization Functions
    - Linux Built-in Functions for Atomic Memory Access
    - C++ 11 Atomic Operations Library

Any Question about Atomic?

Memory Ordering(Reordering)

- Reordering
  - Reads and writes **do not always** happen in the order that you have written them in your code.

- Why Reordering
  - **Performance**

- Reordering Principle
  - In single threaded programs from the programmer's point of view, all operations appear to have been executed in the order specified, with all inconsistencies hidden by hardware.

  - 一段程序，在Reordering前，与Reordering后，拥有相同的执行效果(**Single Thread)**

Reordering

- Examples
  - Example 1

```
int A, B;

void foo()
{
    A = B + 1;
    B = 0;
}
```

```
gcc -S -masm=intel cordering.c
mov     eax, DWORD PTR B[rip]
add     eax, 1
mov     DWORD PTR A[rip], eax
mov     DWORD PTR B[rip], 0
```

```
gcc -O2 -S -masm=intel cordering.c
mov     eax, DWORD PTR B[rip]
mov     DWORD PTR B[rip], 0
add     eax, 1
mov     DWORD PTR A[rip], eax
```

    - A, B 赋值操作被Reorder；

  - Example 2

    - 假设X，Y初始化为0；
    - **Question**：那么Load X，Y，会得到X，Y均为0吗？
    - Test Code & Test Result

```
ntse@db-21:~/hdc/ordering$ ./ordering 1
1 reorders detected after 1568 iterations
2 reorders detected after 2473 iterations
3 reorders detected after 3624 iterations
4 reorders detected after 3656 iterations
5 reorders detected after 5207 iterations
```

Reordering-Type

- Compiler Reordering
  - Example 1，出现在编译期间的Reordering，称之为Compiler Reordering；
- CPU Memory Ordering
  - Example 2，出现在执行期间的Reordering，称之为CPU Memory Ordering；
- 用户程序，无论是在编译期间，还是在执行期间，都会产生Reordering；

Compiler Reordering & Compiler Memory Barrier

- Compiler Reordering能够提高程序的运行效率。但有时候 (尤其是针对Parallel Programming)，我们并不想让Compiler将我们的程序进行Reordering。此时，就需要有一种机制，能够告诉Compiler，不要进行Reordering，这个机制，就是Compiler Memory Barrier。

- Memory Barrier
  - A memory barrier, is a type of **barrier instruction** which causes a **central processing unit (CPU)** or **compiler** to **enforce an ordering constraint** on memory operations issued before and after the barrier instruction. This typically means that certain operations are guaranteed to be performed before the barrier, and others after.

- **Compiler Memory Barrier**
  - 顾名思义，Complier Memory Barrier就是阻止Compiler进行Reordering的Barrier Instruction；

Compiler Memory Barrier

- Compiler Memory Barrier Instructions
  - GNU
    - asm volatile("" ::: "memory");
    - **__asm__ __volatile__ ("" ::: "memory");**
  - Intel ECC Compiler
    - **__memory_barrier();**
  - Microsoft Visual C++
    - **_ReadWriteBarrier();**

- 使用Compiler Memory Barrier后的效果

```
int A, B;
void foo()
{
    A = B + 1;
    __asm__ __volatile__ ("" ::: "memory");
    B = 0;
}
```

```
gcc -O2 -S -masm=intel cordering.c
 mov      eax, DWORD PTR B[rip]
 add      eax, 1
 mov      DWORD PTR A[rip], eax
 mov      DWORD PTR B[rip], 0
```

  - **乱序消失；**

Compiler Memory Barrier

- 注意：
  - Compiler Memory Barrier只是一个通知的标识，告诉Compiler在看到此指令时，不要对此指令的上下部分做Reordering。
  - 在编译后的汇编中，Compiler Memory Barrier消失，CPU不能感知到Compiler Memory Barrier的存在，这点与后面提到的CPU Memory Barrier有所不同；

CPU Memory Ordering

- Definition
  - The term memory ordering refers to the **order** in which the processor issues **reads(loads)** and **writes(stores)** through the **system bus** to **system memory**. (From Intel System Programming Guide 8.2)

- Some Questions
  - 为什么需要reordering？
    - 1: L1 Latency 4 clks; L2 10 clks; L3 20 clks; **Memory 200 clks → Huge Latency**
    - 2: 考虑指令执行时，read与write的优先级；(CPU设计时，重点考虑)
  - 有哪些Reordering情况？不同的CPU，支持哪些Reordering？

CPU Reordering Types

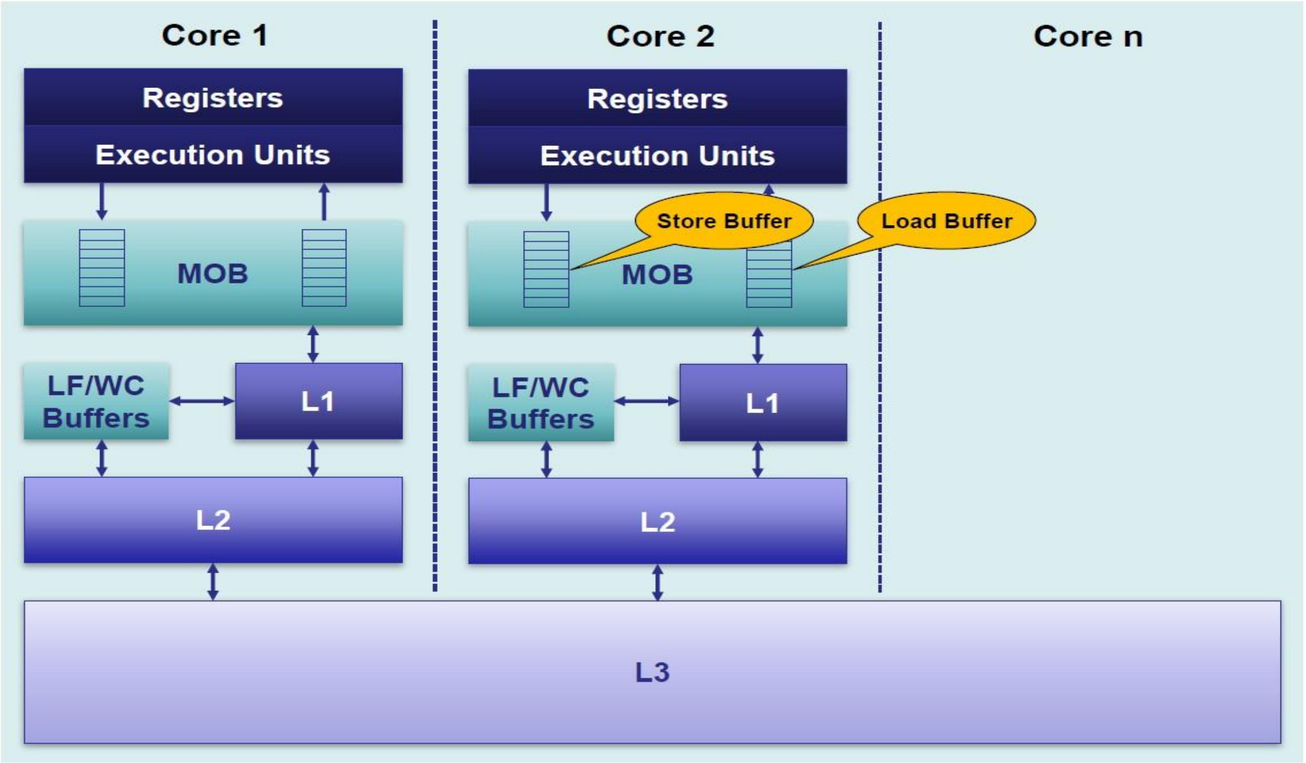
- 2 Instructions
- 2 operation types： read(load) and write(store)

- 4 CPU Reordering Types
  - LoadLoad
    - 读读乱序
  - LoadStore
    - 读写乱序
  - StoreLoad
    - 写读乱序
  - StoreStore
    - 写写乱序

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

扩展知识：CPU如何实现Memory Reordering

- Buffer and Queue
  - Load/Store Buffer；Line Fill Buffer/Write Combining Buffer；Invalidate Message Queue；...

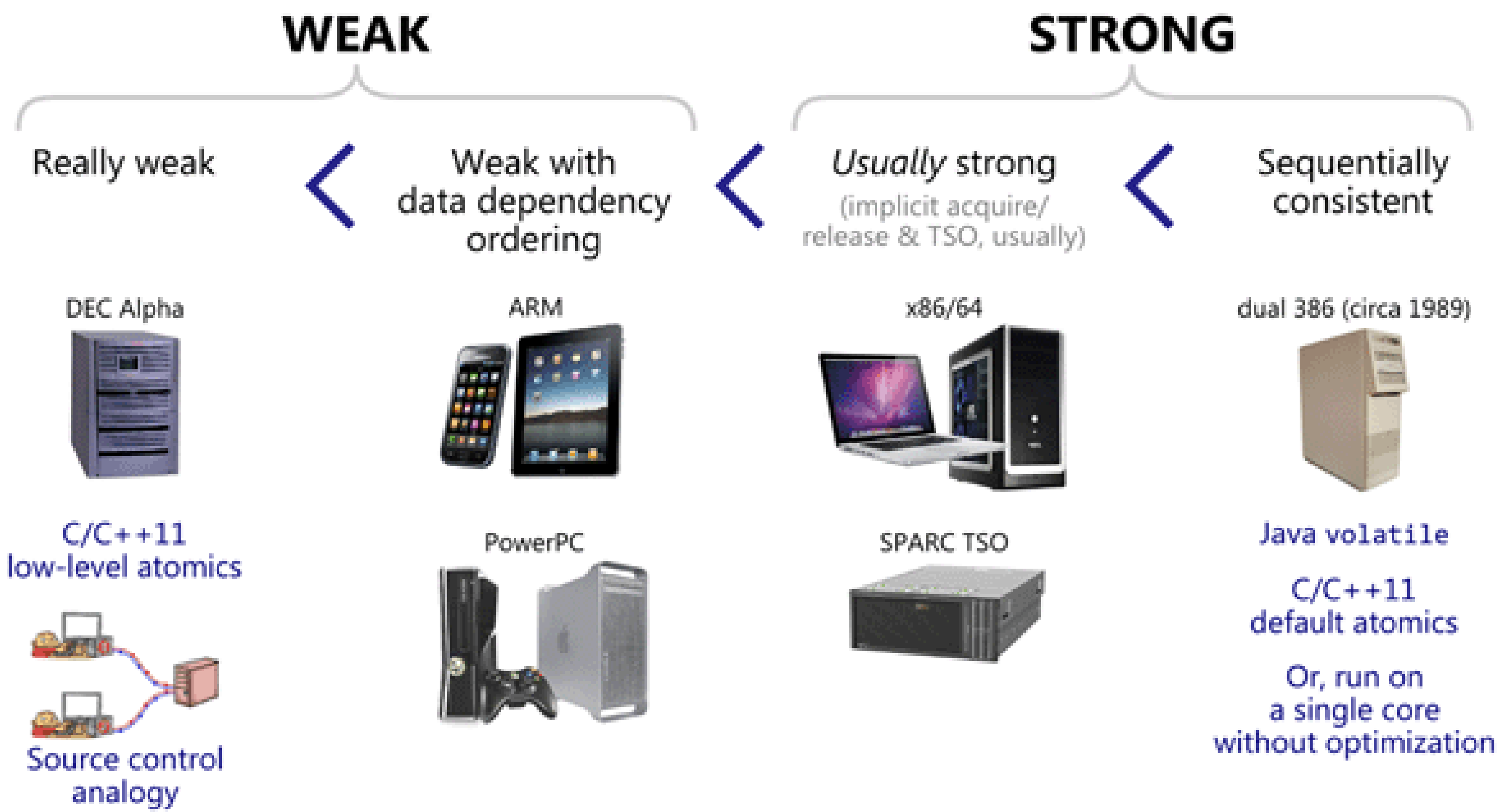


  - 深入了解，见下面列出的参考资料

CPU Memory Models

- Definitions
  - Memory consistency models describe how threads may **interact through shared memory** consistently.
  - There are **many types of memory reordering**, and not all types of reordering occur equally often. It all **depends on processor** you're targeting and/or the **tool chain you're using for development**.
- 主要的CPU Memory Models (Memory Consistency)
  - Sequential Consistency (SC) Strong
  - Total Store Order Model (TSO)
  - Data Dependency Order Weak
  - ...

CPU Memory Models



Intel X86/64 Memory Model(1)

- **In a single-processor system for memory regions defined as write-back cacheable.**
  - Reads are **not reordered** with **other** reads.
  - Writes are **not reordered** with **older** reads.
  - Writes to memory are **not reordered** with **other** writes.

  - **Reads may be reordered** with **older writes** to **different locations** but not with older writes to the same location.

  - 注：以下部分，稍后分析
  - ~~Reads or writes cannot be reordered with I/O instructions, locked instructions, or serializing instructions.~~
  - ~~Reads cannot pass earlier LFENCE and MFENCE instructions.~~
  - ~~Writes cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.~~
  - ~~LFENCE instructions cannot pass earlier reads.~~
  - ~~SFENCE instructions cannot pass earlier writes.~~
  - ~~MFENCE instructions cannot pass earlier reads or writes.~~

Intel X86/64 Memory Model(2)

- **In a multiple-processor system**
  - Individual processors use the same ordering principles as in a single-processor system.
  - **Writes by a single processor** are observed **in the same order** by all processors.
  - Writes from an individual processor are NOT ordered with respect to the writes from other processors.
  - Memory ordering obeys causality (memory ordering respects transitive visibility).
  - **Any two stores** are seen in a **consistent order** by **processors other** than those performing the stores.

  - 注：以下部分，稍后分析
  - ~~Locked instructions have a total order.~~

Intel X86/64 Memory Model(3)

- 解读
  - **普通内存操作**，只可能存在**StoreLoad** Reordering；
  - LoadLoad、LoadStore、StoreStore均不可能Reordering；

#LoadLoad	#LoadStore
#StoreLoad	#StoreStore

  - 一个Processor的Writes操作，其他Processor看到的顺序是一致的；(TSO)
  - 不同Processors的Writes操作，是没有顺序保证的；

- StoreLoad Reordering Problem
  - [Failure of Dekker’s algorithm]()
  - [Test Code]()

	Processor 1	Processor 2	
store to X	mov [X], 1	mov [Y], 1	store to Y
load from Y	mov r1, [Y]	mov r2, [X]	load from X

StoreLoad Reordering Problem

-

```
class Peterson
{
private:
    // indexed by thread ID, 0 or 1
    bool _interested[2];
    // who's yielding priority?
    int _victim;
public:
    Peterson()
    {
        _victim = 0;
        _interested[0] = false;
        _interested[1] = false;
    }
    void lock()
    {
        // threadID is thread local,
        // initialized either to zero or one
        int me = threadID;
        int he = 1 - me;
        _interested[me] = true;
        _victim = me;
        while (_interested[he] && _victim == me)
            continue;
    }
    void unlock()
    {
        int me = threadID;
        _interested[me] = false;
    }
}
```

```
zeroWants = false;
oneWants = false;
victim = 0;
```

Thread 0	Thread 1

zeroWants = true;
victim = 0;
while (oneWants && victim == 0)
continue;
// critical code
zeroWants = false;
	oneWants = true;
	victim = 1;
	while (zeroWants && victim == 1)
	continue;
	// critical code
	oneWants = false;

Thread 0	Thread 1
store(zeroWants, 1) store(victim, 0) r0 = load(oneWants) r1 = load(victim)	store(oneWants, 1) store(victim, 1) r0 = load(zeroWants) r1 = load(victim)

Thread 0	Thread 1
r0 = load(oneWants) store(zeroWants, 1) store(victim, 0) r1 = load(victim)	**r0 = load(zeroWants)** store(oneWants, 1) store(victim, 1) r1 = load(victim)

What About Other CPUs?

Memory ordering in some architectures[2][3]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

- So you know why we call X86, AMD64 as Strong-Ordered (Total Store Order, TSO).

How to Prevent CPU Memory Reordering

- Think about Compiler Memory Barrier
  - asm volatile("" ::: "memory");
  - **__asm__ __volatile__ ("" ::: "memory");**

- Memory Barrier Definition
  - A memory barrier, is a type of **barrier instruction** which causes a **central processing unit (CPU)** or **compiler** to **enforce an ordering constraint** on **memory operations** issued before and after the barrier instruction. This typically means that certain operations are guaranteed to be performed before the barrier, and others after.

- CPU Memory Barrier
  - 顾名思义，Compiler Memory Barrier既然是用来告诉Compiler在**编译阶段**不要进行指令乱排，那么CPU Memory Barrier就是用来告诉CPU，在**执行阶段**不要交互两条操作内存的指令的顺序；
  - **注意**：由于CPU在执行时，必须感知到CPU Memory Barrier的存在，因此**CPU Memory Barrier是一条真正的指令**，存在于编译后的汇编代码中；

CPU Memory Types(theoretical)

- 面临的问题
  - 4种CPU Memory Reordering
    - LoadLoad, LoadStore, StoreLoad, StoreStore
- 4种基本的CPU Memory Barriers
  - LoadLoad Barrier
  - LoadStore Barrier
  - StoreLoad Barrier
  - StoreStore Barrier
- 更为复杂的CPU Memory Barriers
  - **Store Barrier (Write Barrier)**
    - 所有在Store Barrier前的Store操作，必须在Store Barrier指令前执行完毕；而所有Store Barrier指令后的Store操作，必须在Store指令执行结束后才能开始；
    - Store Barrier只针对Store(Write)操作，对Load无任何影响；
  - **Load Barrier (Read Barrier)**
    - 将Store Barrier的功能，全部换为针对Load操作即可；
  - **Full Barrier**
    - Load + Store Barrier，Full Barrier两边的任何操作，均不可交换顺序；

Memory Barrier Instructions in CPU

- **x86, x86-64, amd64**
  - **lfence**：Load Barrier
  - **sfence**：Store Barrier
  - **mfence**：Full Barrier
- PowerPC
  - sync：Full Barrier
- MIPS
  - sync：Full Barrier
- Itanium
  - mf：Full Barrier
- ARMv7
  - dmb
  - dsb
  - isb

Use CPU Memory Barrier Instructions(x86)

- Only CPU Memory Barrier
  - asm volatile(“**mfence**”);

- CPU + Compiler Memory Barrier
  - asm volatile(“**mfence**” ::: ”**memory**”);

- Use Memory Barrier in C/C++

```
// ----- THE TRANSACTION! -----
X = 1;
asm volatile("mfence" ::: "memory");  // Prevent memory reordering
r1 = Y;
```

```
...
mov     DWORD PTR _X, 1
mfence
mov     eax, DWORD PTR _Y
mov     DWORD PTR _r1, eax
...
```

Yes！ We Need **Lock Instruction**’s Help！

- Question？
  - 除了CPU本身提供的Memory Barrier指令之外，是否有其他的方式实现Memory Barrier？

- Yes! We Need **Lock Instruction**’s Help!
  - **Reads or writes cannot** be reordered with I/O instructions, **locked instructions**, or serializing instructions.

  - 解读
    - 既然read/write不能穿越locked instructions进行reordering，那么所有带有lock prefix的指令，都构成了一个天然的**Full Memory Barrier**；

Use Lock Instruction to Implement a MB

- lock addl
  - **asm volatile("lock; addl $0,0(%%esp)" ::: "memory")**
  - addl $0,0(%%esp) → do nothing
  - lock; → to be a cpu memory barrier
  - “memory” → to be a compiler memory barrier

- xchg
  - **asm volatile(“xchgl (%0),%0” ::: “memory”)**
  - **Question**: why xchg don’t need lock prefix?
  - Answer: **The LOCK prefix is automatically assumed for XCHG instruction**.

- lock cmpxchg
  - Do it yourself

Memory Barriers in Compiler & OS

- Linux(x86, x86-64)
  - smp_rmb()
  - smp_wmb()
  - smp_mb()

```
#ifdef CONFIG_X86_32
/*
 * Some non-Intel clones support out of order store. wmb() ceases to be a
 * nop for these.
 */
#define mb() alternative("lock; addl $0,0(%%esp)", "mfence", X86_FEATURE_XMM2)
#define rmb() alternative("lock; addl $0,0(%%esp)", "lfence", X86_FEATURE_XMM2)
#define wmb() alternative("lock; addl $0,0(%%esp)", "sfence", X86_FEATURE_XMM)
#else
#define mb()    asm volatile("mfence":::"memory")
#define rmb()   asm volatile("lfence":::"memory")
#define wmb()   asm volatile("sfence" ::: "memory")
```

```
void foo(void)
{
  a = 1;
  smp_mb();
  b = 1;
}

void bar(void)
{
  while (b == 0) continue;
  smp_mb();
  assert(a == 1);
}
```

- Windows(x86, x86-64)
  - MemoryBarrier()

```
FORCEINLINE
VOID
MemoryBarrier (
    VOID
    )
{
    LONG Barrier;
    __asm {
        xchg Barrier, eax
    }
}
```

X86 Memory Ordering with Memory Barrier(1)

- **In a single-processor system for memory regions defined as write-back cacheable.**
  - Reads are not reordered with other reads.
  - Writes are not reordered with older reads.
  - Writes to memory are not reordered with other writes.
  - Reads may be reordered with older writes to different locations but not with older writes to the same location.

  - 注：新增部分
  - **Reads or writes cannot be reordered** with **I/O instructions**, **locked instructions**, or serializing instructions.
  - **Reads cannot** pass earlier **LFENCE** and **MFENCE** instructions.
  - **Writes** cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
  - LFENCE instructions cannot pass earlier reads.
  - SFENCE instructions cannot pass earlier writes.
  - MFENCE instructions cannot pass earlier reads or writes.

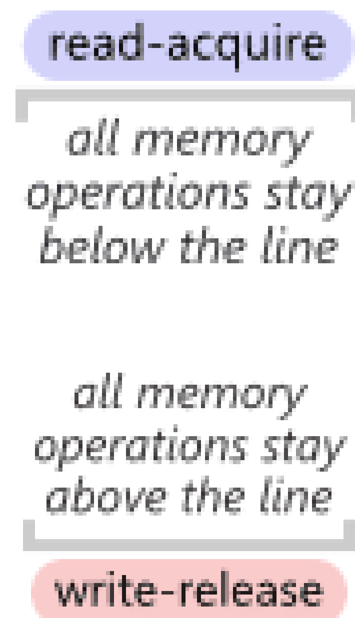
X86 Memory Ordering with Memory Barrier(2)

- **In a multiple-processor system**
  - Individual processors use the same ordering principles as in a single-processor system.
  - Writes by a single processor are observed in the same order by all processors.
  - Writes from an individual processor are NOT ordered with respect to the writes from other processors.
  - Memory ordering obeys causality (memory ordering respects transitive visibility).
  - Any two stores are seen in a consistent order by processors other than those performing the stores.

  - 注：新增部分
  - **Locked instructions have a total order.**

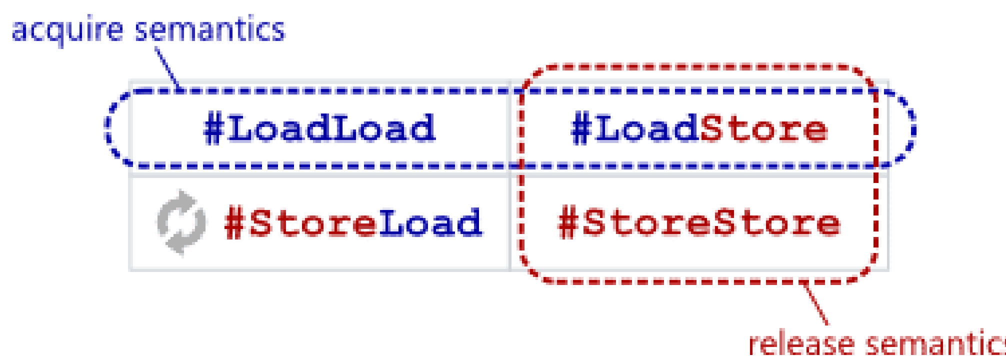
Read Acquire vs Write Release(1)

- Read Acquire and Write Release
  - Two Special Memory Barriers.
  - Definition
    - A **read-acquire** executes **before all reads and writes** by the same thread that **follow** it in program order.
    - A **write-release** executes **after all reads and writes** by the same thread that **precede** it in program order.
- Question
  - Read Acquire and Write Release 有何作用？

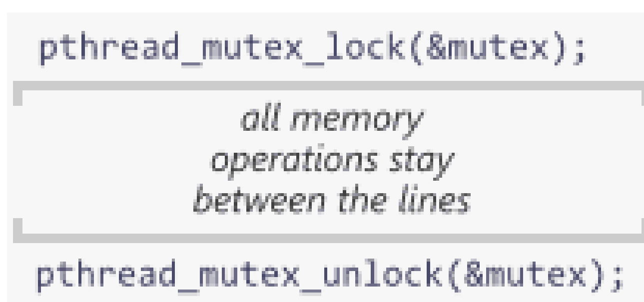


Read Acquire vs Write Release(2)

- Read Acquire and Write Release Barriers
  - Read Acquire
    - LoadLoad + LoadStore Barrier
  - Write Release
    - LoadStore + StoreStore Barrier



- 解读
  - Read Acquire + Write Release语义，是所有锁实现的基础(Spinlock, Mutex, RWLock, ...)，所有被[Read Acquire, Write Release]包含的区域，即构成了一个临界区，临界区内的指令，确保不会在临界区外运行。因此，Read Acquire又称为Lock Acquire，Write Release又称为Unlock Release；



How to Implement Read Acquire/Write Release?

- Intel X86, X86-64
  - Full Memory Barrier
    - mfence
    - locked instruction

- Compiler and OS
  - Linux
    - smp_mb()

  - Windows
    - Functions with Acquire/Release Semantics
    - InterlockedIncrementAcquire ()...

Extension： StoreLoad Reorder

- Question
  - 为什么Intel CPU在LoadLoad， LoadStore， StoreLoad， StoreStore乱序中，仅仅保持了StoreLoad乱序？
  - 为什么， LoadLoad/LoadStore/StoreStore Barrier乱序被称之为lightweight Barrier? 而StoreLoad Barrier则为Expensive Barrier？
    - on PowerPC, the lwsync (short for “lightweight sync”) instruction acts as all three #LoadLoad, #LoadStore and #StoreStore barriers at the same time, yet is less expensive than the sync instruction, which includes a #StoreLoad barrier.
- Answer
  - Store Buffer;
  - Store异步不影响指令执行；
  - Load只能同步；
- 注意
  - Intel CPU， Load自带Acquire Semantics； Store自带Release Semantics；

Any Question about Memory Ordering?

并发程序设计

- 在充分理解了CPU Cache架构，以及Memory Ordering之后，开始进行并发程序设计与实现，就显得水到渠成；

- 本部分的内容
  - 实现一个自己的Spinlock；
  - 一个真实的案例；
  - volatile：C++ vs Java；
  - 探讨并发程序设计中的一些优化建议；

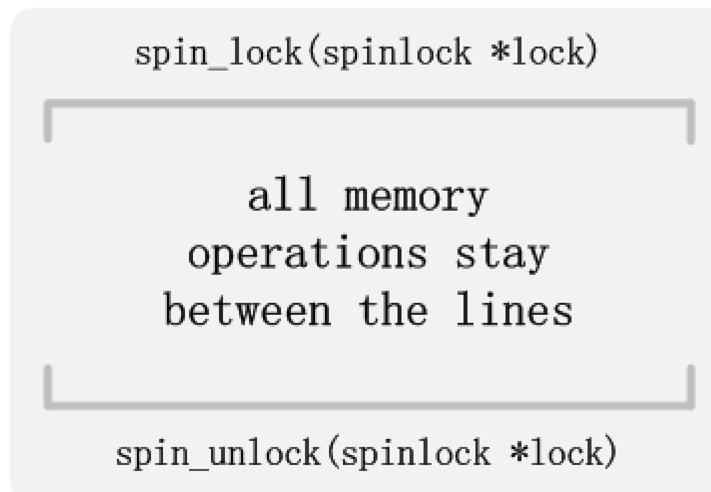
Implement a Spinlock

- Simplest Spinlock
  - [From Lockless]()
- 解读
  - 功能
    - 给定一个unsigned值，0代表未加锁，1代表加锁；只有一个能加锁成功；
  - **spin_lock**
    - xchg
    - implicit lock instruction
  - **spin_unlock**
    - asm volatile(“”: : :”memory”)
  - **Load Acquire**
    - locked instruction = full barrier
  - **Write Release**
    - compile barrier；
    - X86； No LoadStore, StoreStore Reorder;

```c
/* Compile read-write barrier */
#define barrier() asm volatile("": : :"memory")

static inline unsigned xchg_32(void *ptr, unsigned x)
{
    __asm__ __volatile__("xchgl %0,%1"
                :"=r" ((unsigned) x)
                :"m" (*(volatile unsigned *)ptr), "0" (x)
                :"memory");

    return x;
}

#define EBUSY 1
typedef unsigned spinlock;

static void spin_lock(spinlock *lock)
{
    while (xchg_32(lock, EBUSY));
}

static void spin_unlock(spinlock *lock)
{
    barrier();
    *lock = 0;
}

static int spin_trylock(spinlock *lock)
{
    return xchg_32(lock, EBUSY);
}
```

Simplest Spinlock分析(1)

- **功能上**
  - 保证同一时间，只有一个线程能够spin_lock成功，其余线程全部堵在while循环；
  - spin_lock实现了Load Acquire Semantics；
  - spin_unlock实现了Write Release Semantics；
  - 功能上：Success



- 成功应用了前面的多个知识点
  - Intel Memory Ordering Model；
  - CPU Memory Barrier
  - Atomic Instruction
  - Locked Instruction
  - Compile Memory Barrier
  - Load Acquire
  - Store Release

Simplest Spinlock分析(2)

- 性能上
  - 此Simplest Spinlock有很多问题，可以进行优化，集中在spin_lock()函数；
- 问题分析(参考)
  - 1. 根据predict，CPU发现xchg_32函数极少会返回0(Success)，因此将会采用**speculative execution**，CPU流水线中充满xchg指令，消耗CPU；
  - 2. 由于流水线中充斥着**speculative xchg**指令，因此当xchg返回0(Success)，投机失败惩罚较大，尤其是针对长流水线；
  - 3. 若其他CPU长时间持有spin_lock，则当前CPU无法释放资源给其他程序运行；
  - 4. xchg指令，用在此处效率不高；

Simplest Spinlock改进

- **针对问题1，2**
  - 引入pause指令；
    - asm volatile ("pause");
    - 部分平台，不支持pause，可用rep; nop替代；
  - pause指令功能
    - 通知CPU，当前处于spinlock函数调用之中，消除**speculative**，降低CPU消耗，加锁成功后，无需处理失败的**speculative**指令，性能更高；
    - 在超线程下，空闲出来的CPU流水线，可以交给另一个线程使用，提高CPU利用率；
- **针对问题3**
  - 若处理临界区的时间较长，spinlock可从Active模式逐渐退化为Passive模式；
  - Active
    - 不释放CPU资源，反复尝试；
  - Passive
    - 释放CPU资源，进入Sleep，甚至等待唤醒；
- **针对问题4**
  - xchg指令每次都会修改内存，使用更为高效的cmpxchg替代；

Spinlock： Active vs Passive

- Spinning.

- Active
  - Only pause, not release CPU
    - pause(); _mm_pause();

- Passive
  - Release CPU to System, but not Sleep
    - pthread_yield(); SwitchToThread(); Sleep(0);

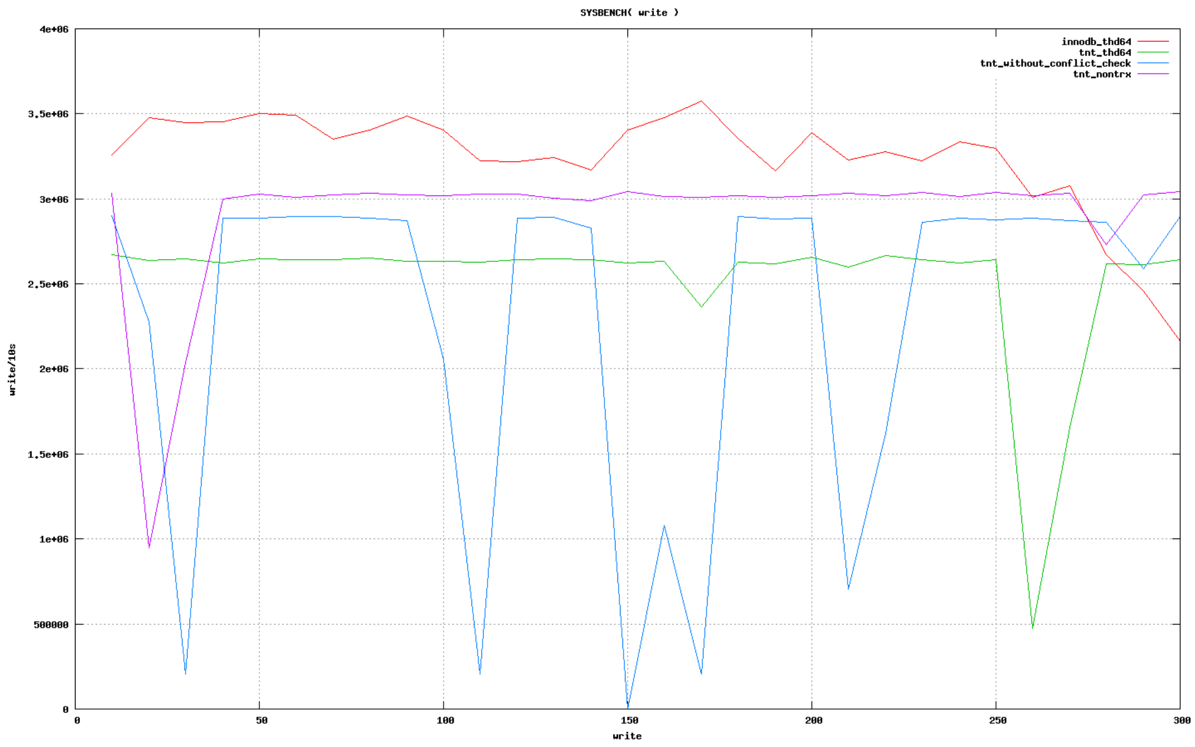
  - Release CPU to System, and Sleep
    - Sleep(n);

- Hybrid
  - Active + Passive
  - 主流实现方式

```
void do_backoff(int& backoff) // backoff is initialized
{
    if (backoff < 10)
        _mm_pause();
    else if (backoff < 20)
        for (int i = 0; i != 50; i += 1) _mm_pause();
    else if (backoff < 22)
        SwitchToThread();
    else if (backoff < 24)
        Sleep(0);
    else if (backoff < 26)
        Sleep(1);
    else
        Sleep(10);
     backoff += 1;
}
```

一个真实案例

- 案例来源
  - TNT/NTSE引擎为了高效率，实现了自己的Mutex；
  - 进行Sysbench Insert测试的效果：TPS



一个真实案例(续)

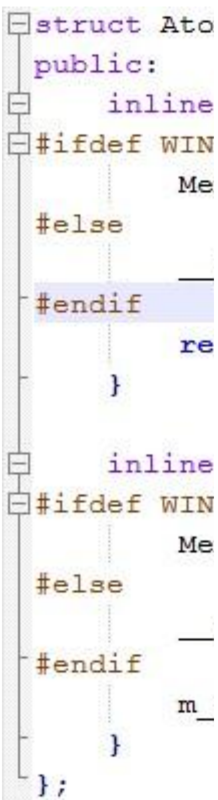
- 案例分析(Mutex部分实现)

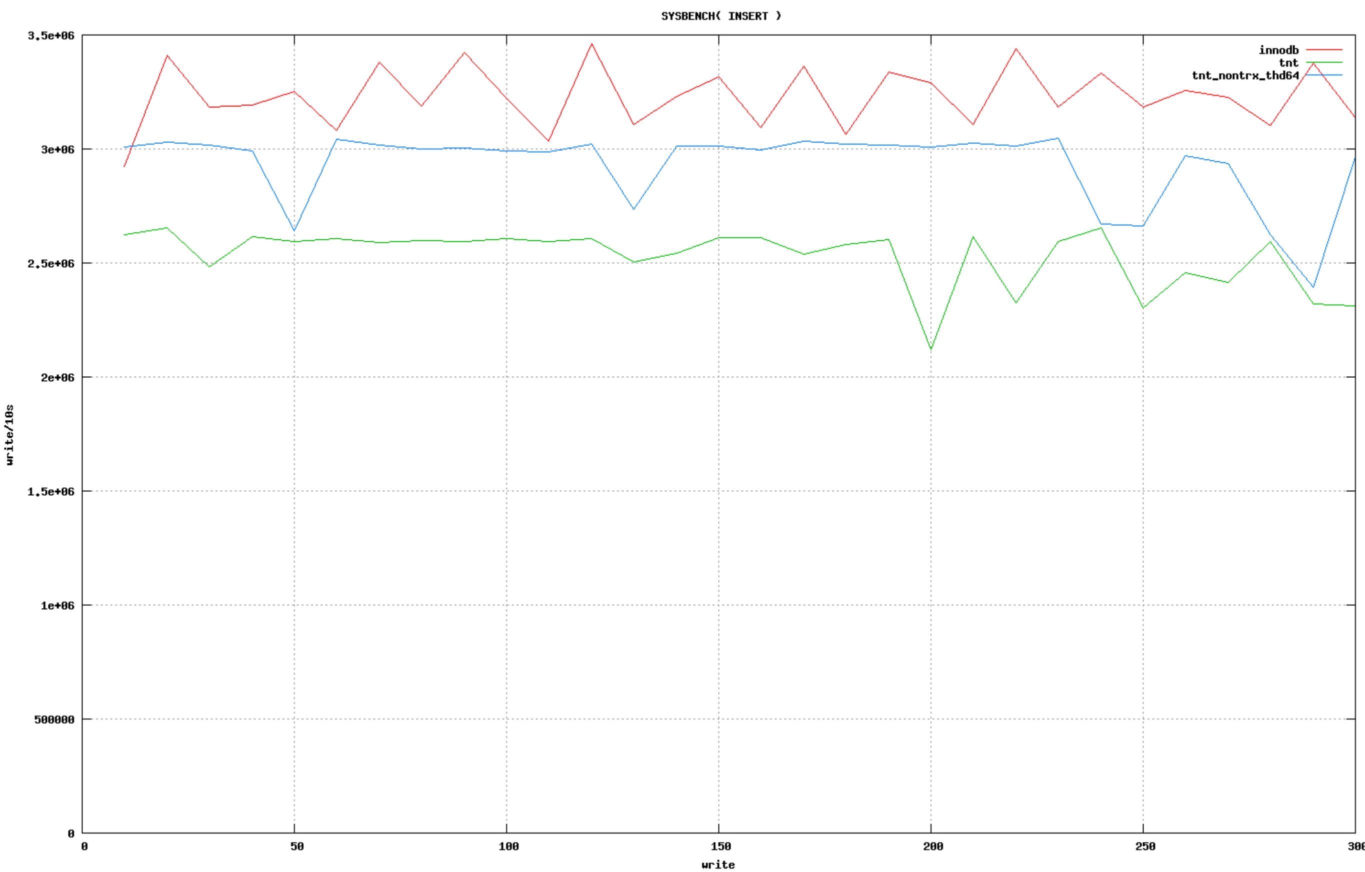
```
struct Atomic {
public:
    inline T get() const {
        return m_v;
    }

    inline void set(T v) {
#ifdef WIN32
        MemoryBarrier();
#else
        __sync_synchronize();
#endif
        m_v = v;
    }
};
```

```
struct Mutex: public Lock {
public:
    inline void lock(const char *file, uint line) {
        if (!m_lockWord.compareAndSwap(0, 1))
            lockConflict(-1);
    }

    inline bool isLocked() const {
        return m_lockWord.get() > 0;
    }

    inline void unlock() {
        assert(m_lockWord.get() == 1);
        m_lockWord.set(0);
        if (m_waiting.get() > 0)
            m_event.signal(true);
    }

private:
    Atomic<int>     m_lockWord; /** 0表示未被锁定，1表示被锁定 */
    Atomic<int>     m_waiting;  /** 正在等待加锁的线程数 */
    Event           m_event;    /** 用于唤醒等待线程的事件 */
};
```

一个真实案例(续)

- 解决方案/效果



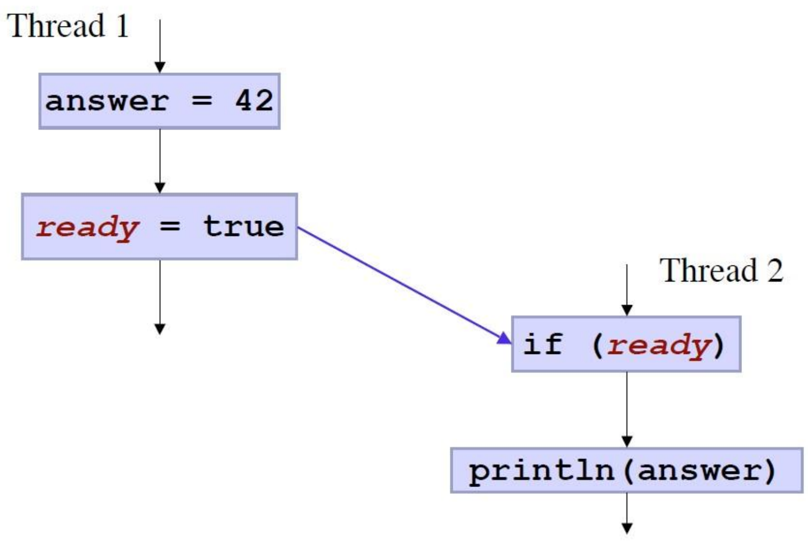


Volatile： C/C++ vs Java (1)

- **Volatile**
  - 易失的，不稳定的…
- **Volatile in C/C++**
  - The **volatile** keyword is used on variables that may be modified simultaneously by other threads. This warns the compiler to fetch them fresh each time, rather than caching them in registers. (**read/write actually happens**)
  - **No reordering** occurs between **different volatile reads/writes**. (**only volatile variables guarantee no reordering**)
- **Volatile in Java**
  - (*The Same as C/C++*)， Plus
  - Volatile reads and writes establish a happens-before relationship, much like acquiring and releasing a mutex. (**no reordering takes place**)
    - **Read Volatile**： Acquire Semantics； **Write Volatile**： Release Semantics；
  - Writes and reads of volatile **long and double** values are always **atomic**.
    - The Java Language Specification Java SE 7 Edition： Chapter 17.7

Volatile： C/C++ vs Java (2)

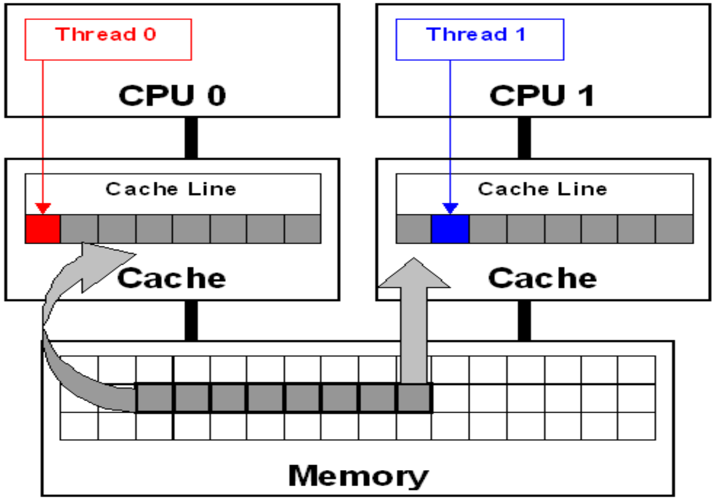
- Examples
  - **int answer = 0;**
  - **bool volatile ready;**



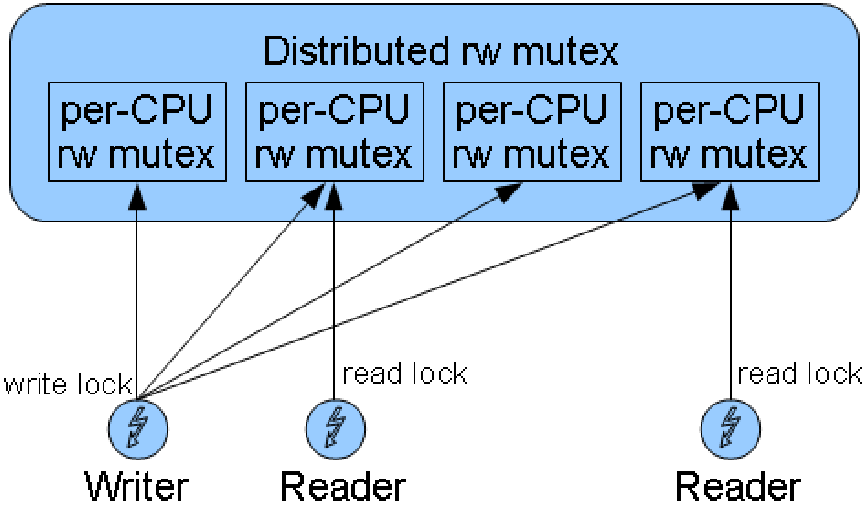
- **Question**
  - Thread 2的answer会输出什么结果？
  - C/C++
    - answer = 42 or 0，均有可能；
  - Java
    - answer = 42，只有唯一的结果；
  - Why？
    - Java’s Volatile：Write Release Semantics → ready(true)一定在answer(42)之后执行；

Others

- False Sharing



- Distributed Read-Write Mutex



- Per-Processor Data

Parallel Programming，未完待续...

Reference-综合

- A Primer on Memory Consistency and Cache Coherence. (Synthesis Lectures on Computer Architecture)
- Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C
- AMD64 Architecture Programmers Manual Volume 1 System Programming
- AMD64 Architecture Programmers Manual Volume 2 System Programming
- MYTHBUSTING MODERN HARDWARE TO GAIN "MECHANICAL SYMPATHY"
- Performance Tuning for CPU(Marat Dukhan)
- Understanding The Linux Kernel 3rd Edition
- Working Draft, Standard for Programming Language C++
- The Art of Multiprocessor Programming
- Nehalem - Everything You Need to Know about Intel's New Architecture
- Intel Core i7 (Nehalem): Architecture By AMD?

Reference-CPU Cache

- Cache Coherence Protocols
- 高速缓存（Cache Memory）
- Cache(268 Pages)
- Cache: a place for concealment and safekeeping
- Gallery of Processor Cache Effects
- Getting Physical With Memory
- Intel’s Haswell CPU Microarchitecture
- Introduction of Cache Memory
- CPU Cache Flushing Fallacy
- Multiprocessor Cache Coherence
- Understanding the CPU Cache
- What Every Programmer Should Know About Memory - Akkadia.org
- What Programmer Should Know about Memory Consistence

Reference-Atomic

- An attempt to illustrate differences between memory ordering and atomic access
- Anatomy of Linux synchronization methods
- Atomic Builtins - Using the GNU Compiler Collection (GCC)
- Atomic vs. Non-Atomic Operations
- Understanding Atomic Operations
- Validating Memory Barriers and Atomic Instructions

Reference-Memory Ordering(1)

- Acquire and Release Semantics
- An attempt to illustrate differences between memory ordering and atomic access
- what is a store buffer?
- Which is a better write barrier on x86: lock+addl or xchgl?
- Relative performance of swap vs compare-and-swap locks on x86
- difference in mfence and asm volatile (“” : : : “memory”)
- Inline Assembly
- Intel memory ordering, fence instructions, and atomic operations.
- Intel’s ‘cmpxchg’ instruction
- Lockless Programming Considerations for Xbox 360 and Microsoft Windows
- Write Combining
- Memory barriers: a hardware view for software hackers
- Memory Barriers Are Like Source Control Operations
- Memory Ordering at Compile Time
- Memory Reordering Caught in the Act
- Memory barriers - The Linux Kernel Archives
- Understanding Memory Ordering
- Weak vs. Strong Memory Models
- Who ordered memory fences on an x86?

Reference-Memory Ordering(2)

- Cambridge Relaxed-Memory Concurrency Group
- Who ordered sequential consistency?
- The Java Memory Model

Reference-Programming(1)

- An Introduction to Lock-Free Programming
- Distributed Reader-Writer Mutex
- Effective Concurrency: Eliminate False Sharing
- False-sharing
- False Sharing
- x86 spinlock using cmpxchg
- SetThreadAffinityMask for unix systems
- Lock Free Algorithms - QCon London
- pause instruction in x86
- Per-processor Data
- Pointer Packing
- Spinlocks and Read-Write Locks
- Spinning
- What does “rep; nop;” mean in x86 assembly?
- Volatile variable
- What are we going to do about volatile?
- What Volatile Means in Java

Reference-Programming(2)

- [Why the “volatile” type class should not be used]
- [C++ and the Perils of Double-Checked Locking]
- [volatile : Java Glossary]
- [Volatile fields]
- [volatile (C++)]
- [volatile vs. volatile]
- [Why is volatile not considered useful in multithreaded C or C++ programming?]

Thanks！