为什么Java程序员也需要了解CPU

作者：周忱 | 淘宝综合业务
微博：@MinZhou
邮箱：zhouchen.zm@taobao.com

关于我

- 花名:周忱(chén)
- 真名:周敏
- 微博: @MinZhou
- Twitter: @minzhou
- 2010年6月加入淘宝
- 曾经淘宝Hadoop&Hive研发组Leader
- 目前专注分布式实时计算
- Hive Contributor
- 自由、开源软件热爱者

关于我

- 花名:周忱(chén)
- 真名:周敏
- 微博: @MinZhou
- Twitter: @minzhou
- 2010年6月加入淘宝
- 曾经淘宝Hadoop&Hive研发组Leader
- 目前专注分布式实时计算
- Hive Contributor
- 自由、开源软件热爱者

什么是volatile?

Puzzle 1

下面哪个程序跑得更快?

```
private static long value;

public static void increment() {
   while (value < 100000000L) {
      value++;
   }
}
```

```
private static volatile long value;

public static void increment() {
   while (value < 100000000L) {
      value++;
   }
}
```

如果换成AtomicLong呢?

Puzzle 2

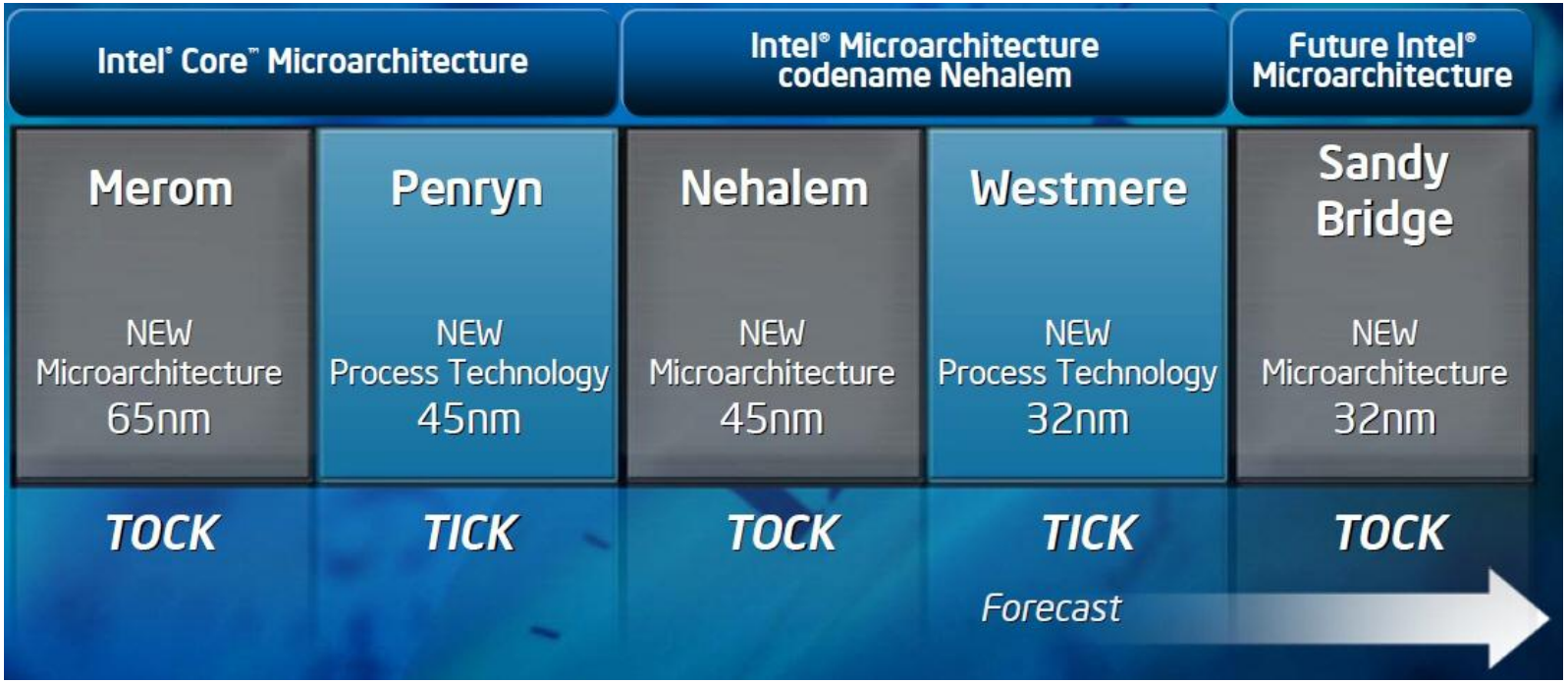
下面哪个程序跑得更快?

```
AtomicIntegerFieldUpdater  state = …

public  void lock() {
   while(state.getAndSet(true)) {} ; //spin
}
```

```
AtomicIntegerFieldUpdater  state= …

public  void lock() {
   while (true) {
      while(state.get()) {}; //spin
      if(!state.getAndSet(true))
         return;
   }
}
```

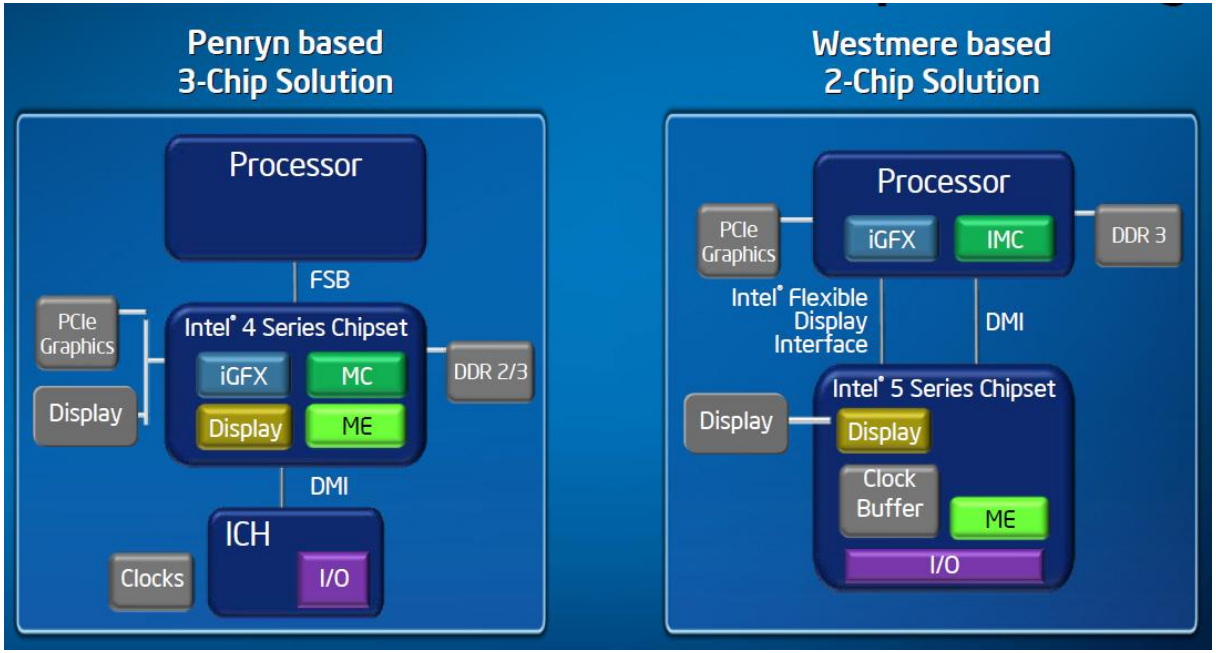
如果换成多线程呢?

Intel 十年计划:Tick-Tock

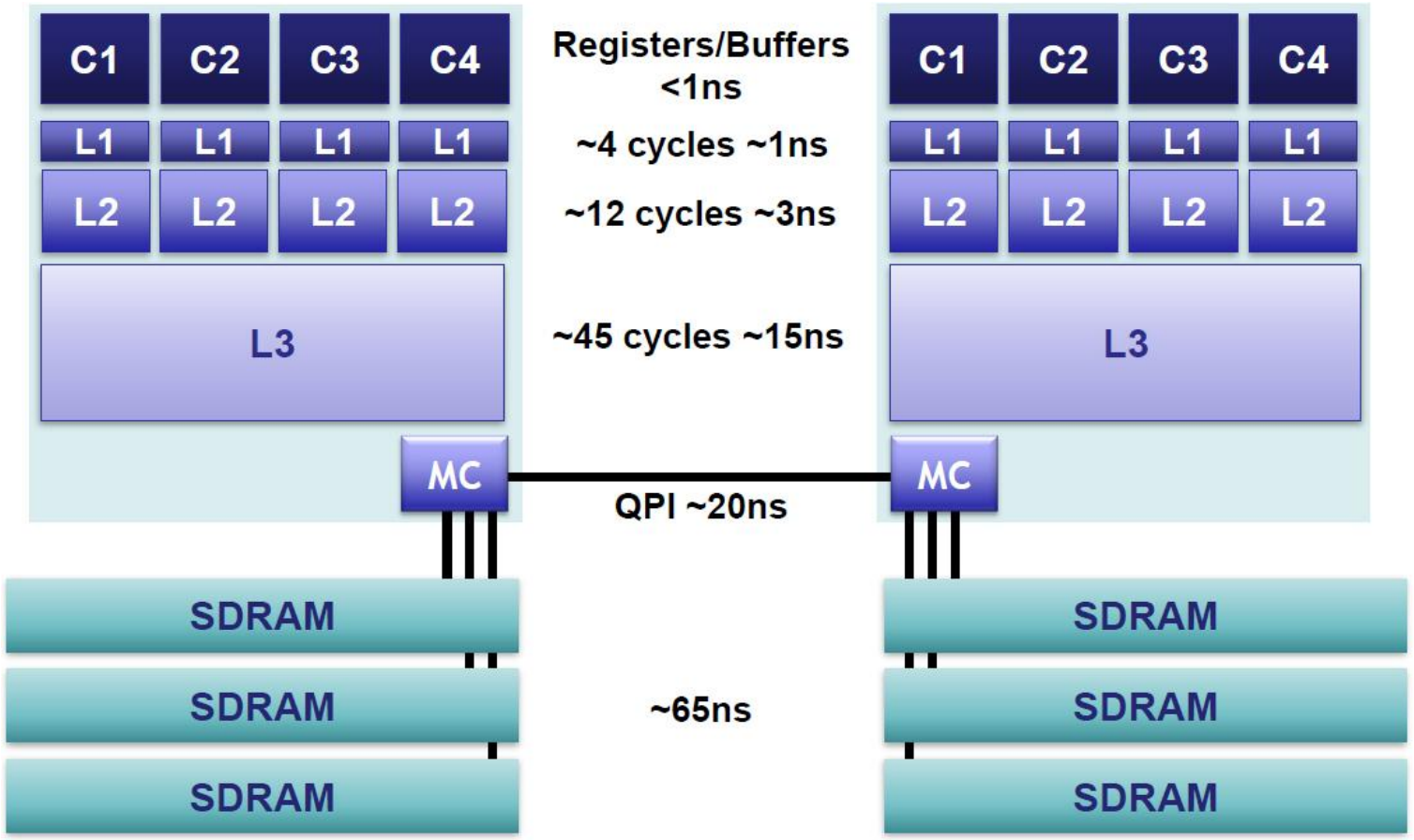


CPU微架构变革

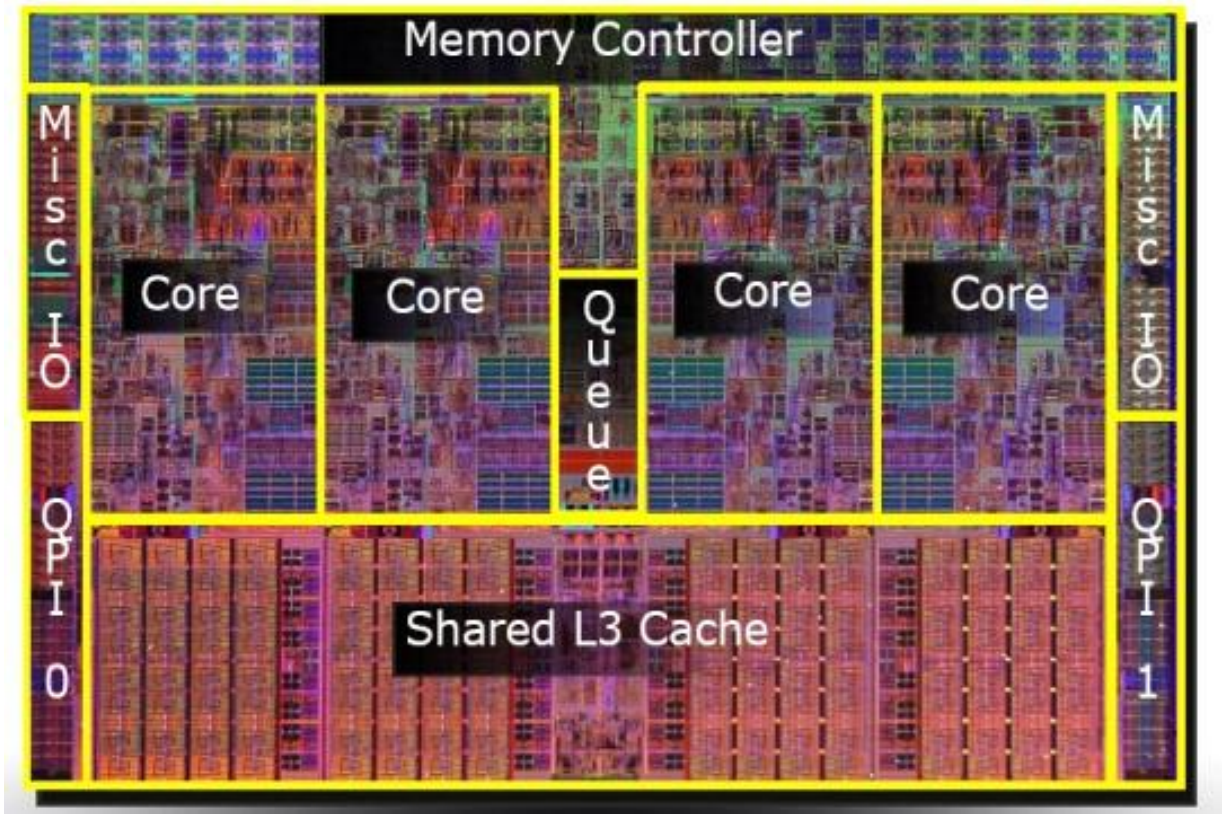
- Hyper-threading
- 32KB L1d 与32KB L1i
- 256KB L2 Cache
- 4–12 MB L3 cache
- QPI 代替FSB
- PCI E和DMI整合入处理器, 北桥消失
- 支持 2/3通道DDR3



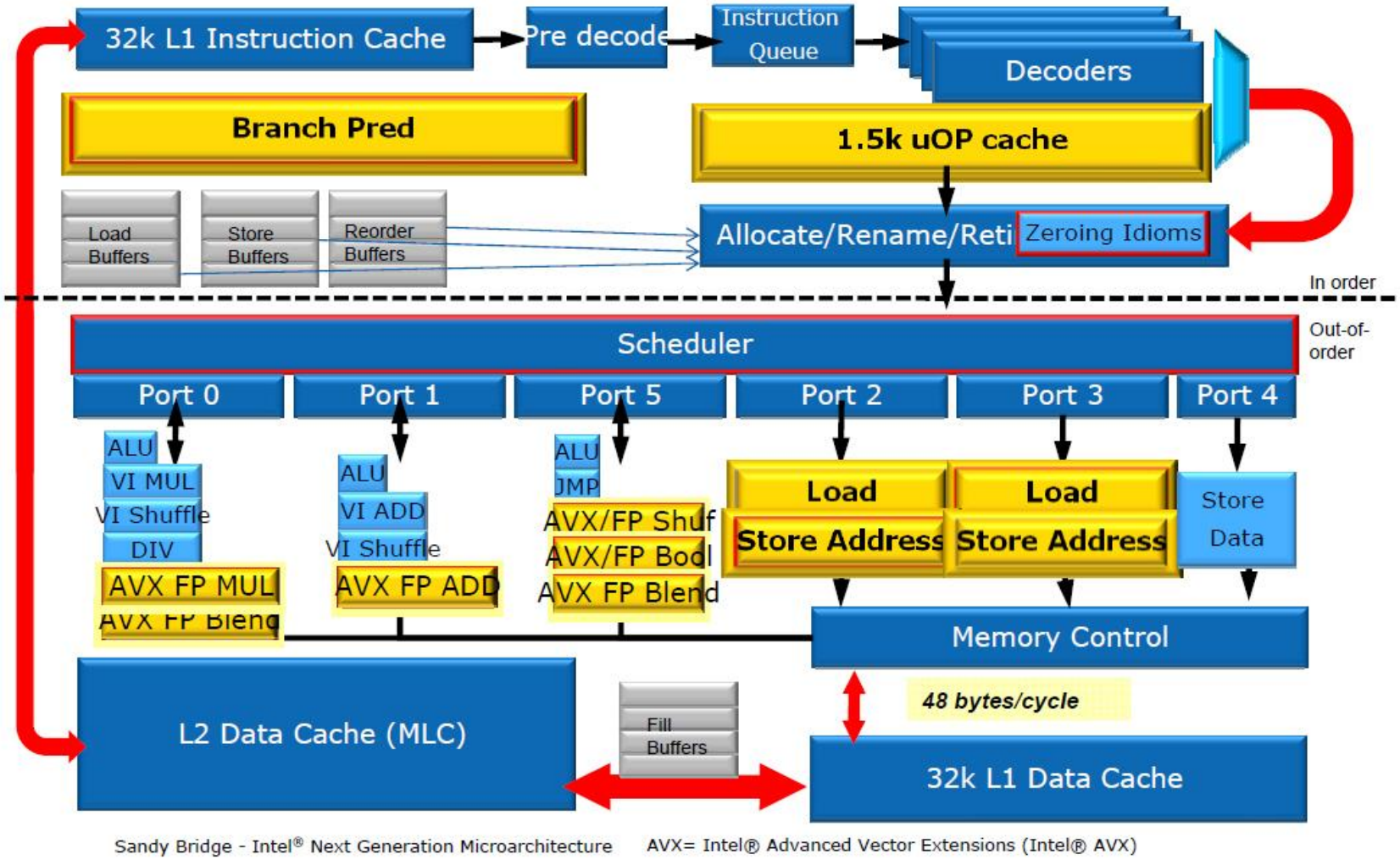
现代CPU微架构


C1
C2
C3
C4
L1
L2
L3
Registers/Buffers
<1ns
~4 cycles ~1ns
~12 cycles ~3ns
~45 cycles ~15ns
MC
QPI ~20ns
SDRAM
~65ns

Intel Nehalem 排布



Intel Sandy Bridge



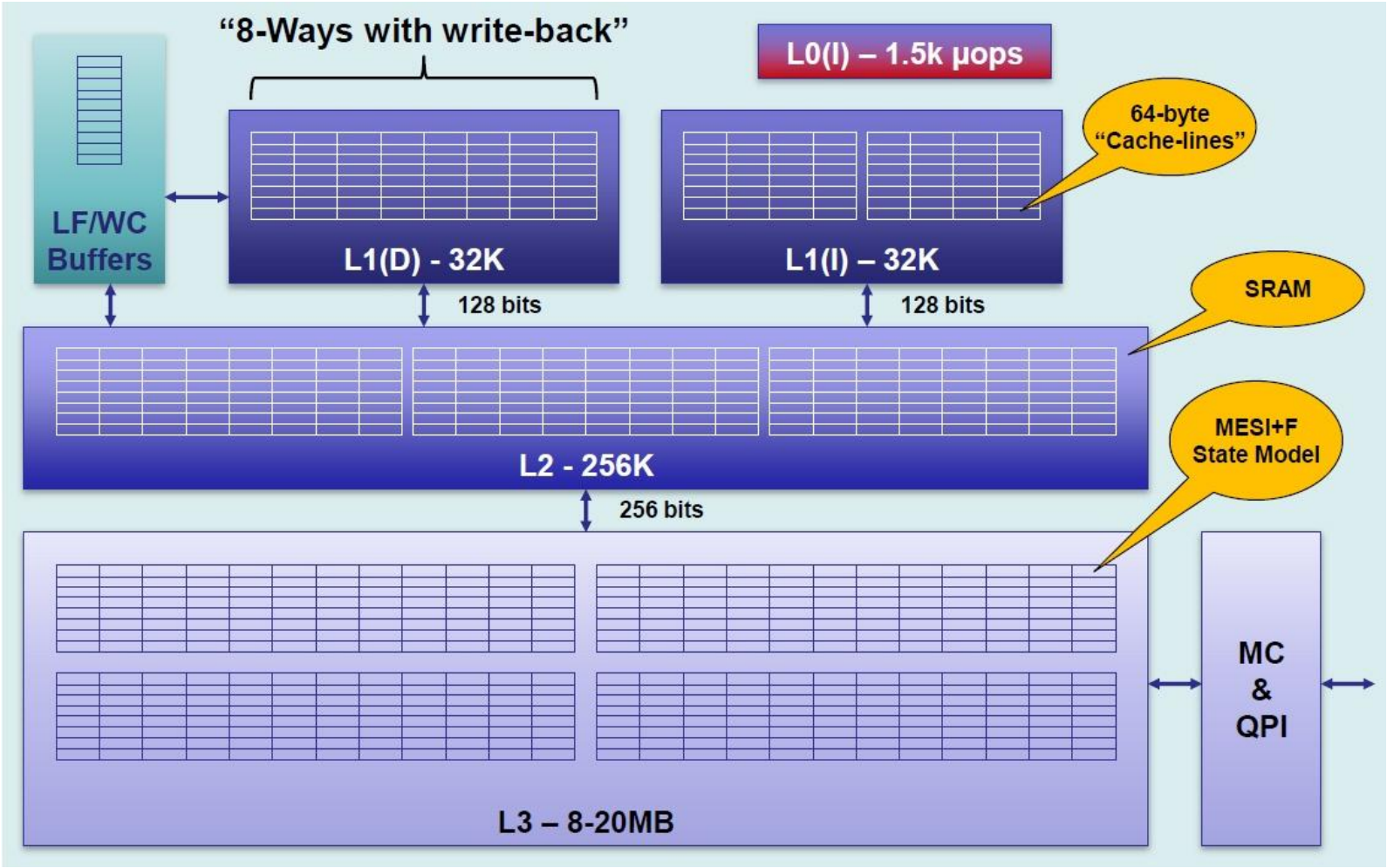
Puzzle 3

下面哪个程序跑得更快?

```
private static long[][] longs;

for (int i = 0; i < DIMENSION_1; i++) {
   for (int j = 0; j < DIMENSION_2; j++) {
      sum += longs[i][j];
    }
}
```

```
private static long[][] longs;

for (int j = 0; j < DIMENSION_2; j++) {
   for (int  i = 0; i < DIMENSION_1; i++){
      sum += longs[i][j];
    }
}
```

CPU Cache

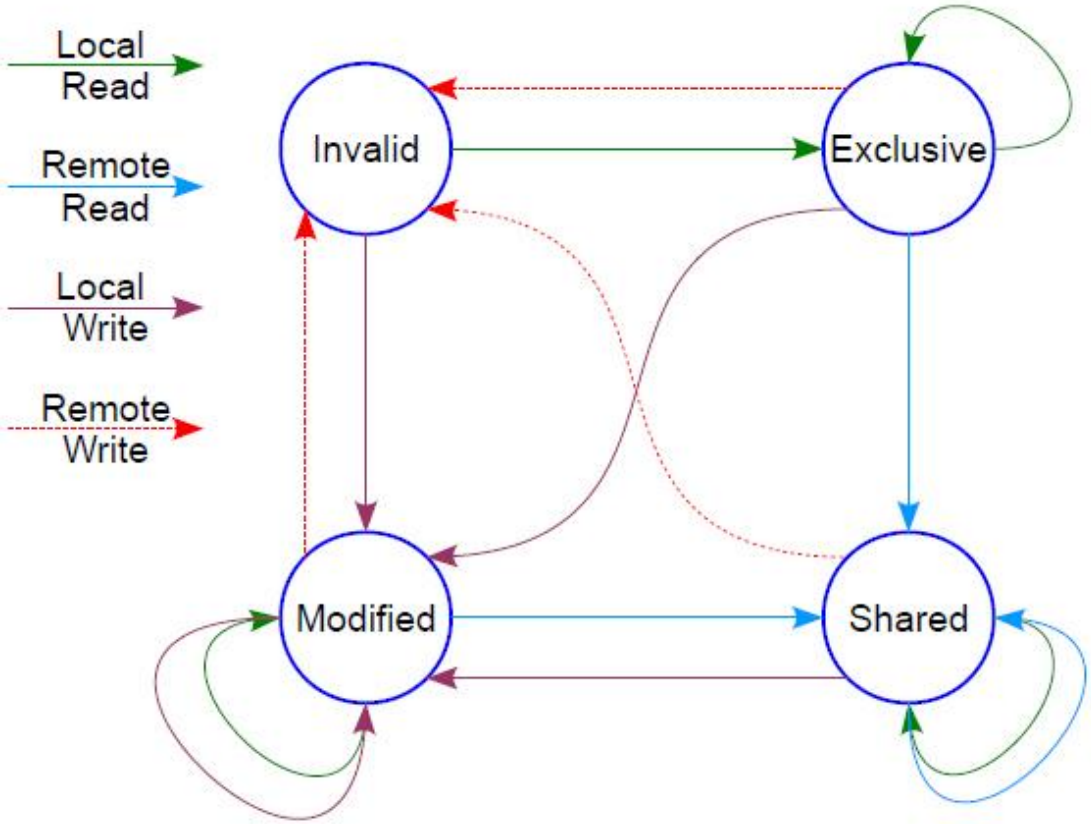


cat /sys/devices/system/cpu/cpu0/cache/index0/*

Cache Coherence & MESI协议

- M: 修改
- E：独占
- S：共享
- I：失效

发音: messy



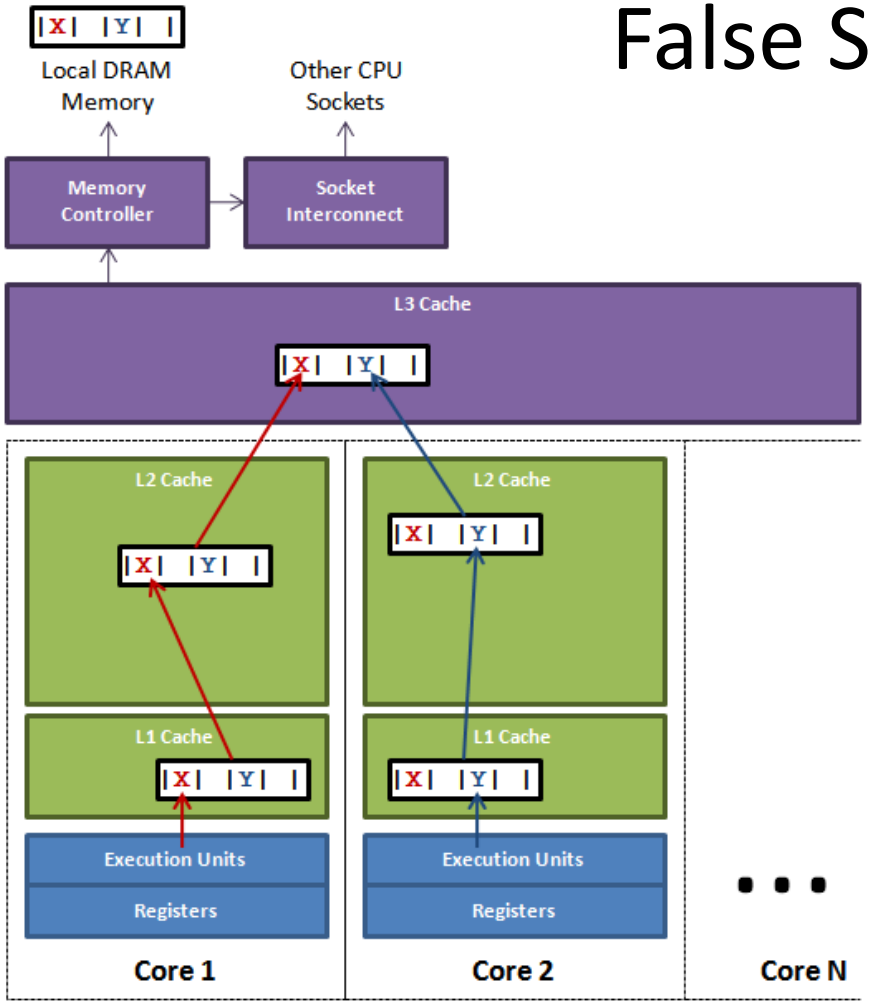
Puzzle 4

下面程序有什么问题?

```
public final static class VolatileLong {
    public volatile long value = 0L;
}

public  class FS implements Runnable {
    public void run() {
        long i = ITERATIONS + 1;
        while (0 != --i) {
            longs[arrayIndex].value = i;
        }
    }
}
```

```
longs = new VolatileLong[NUM_THREADS];
for (int i = 0; i < longs.length; i++) {
    longs[i] = new VolatileLong();
}
// 每个线程更新独立的变量
for (int i = 0; i < NUM_THREADS; i++) {
    threads[i] = new Thread(new FS(i)));
}
for (Thread t : threads) {
    t.start();
}
for (Thread t : threads) {
    t.join();
}
```

False Sharing

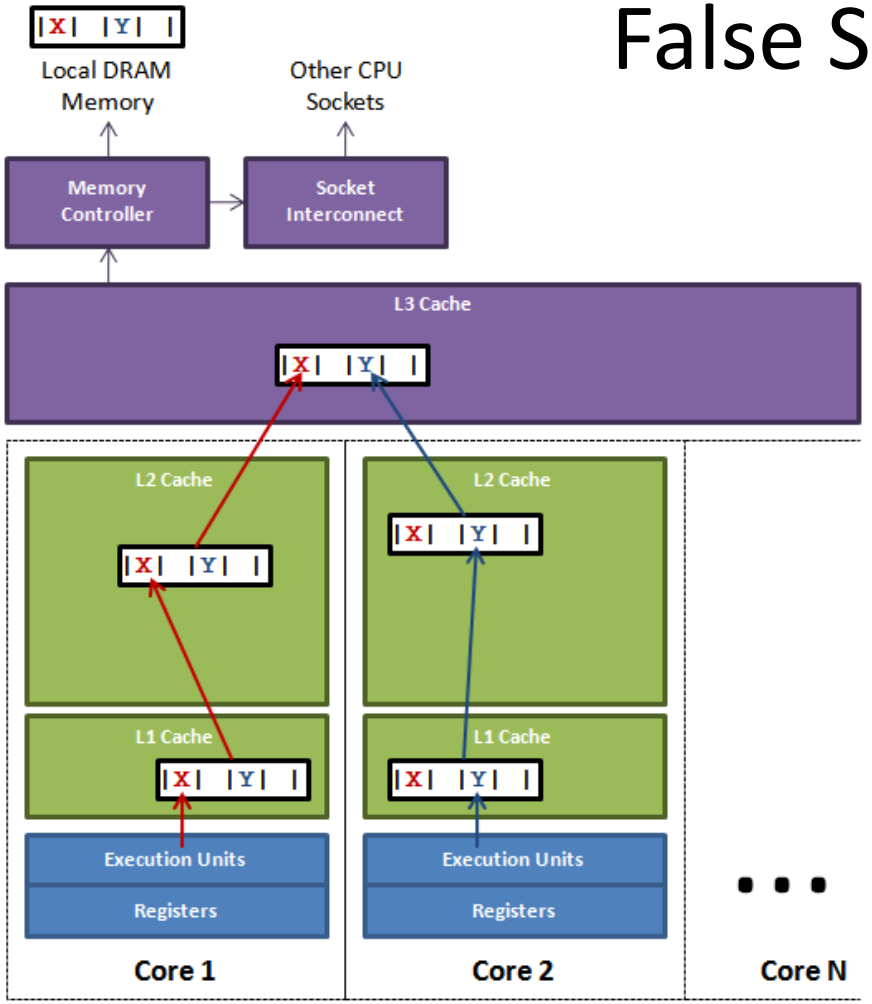



```
public final static class VolatileLong {
    public volatile long value = 0L;
}
```

=>

```
public final static class VolatileLong {
    public volatile long value = 0L;
    public long p1, p2, p3, p4, p5, p6;
}
```

False Sharing

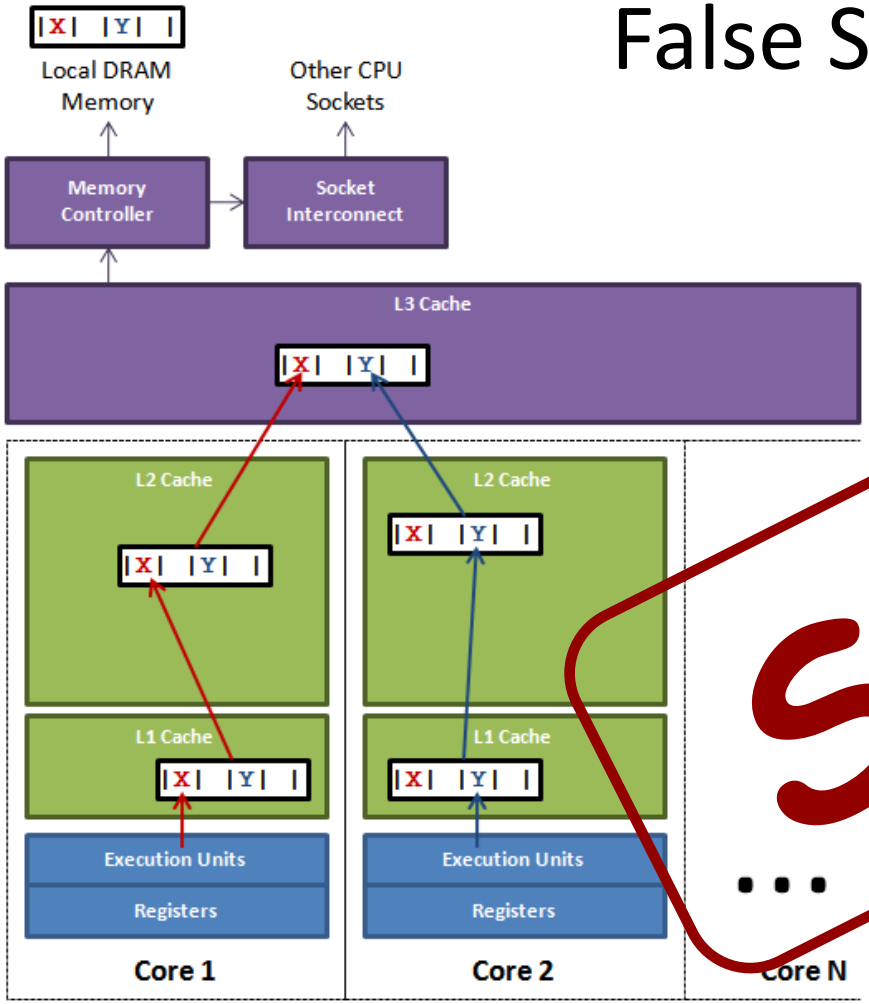



```
public final static class VolatileLong {
    public volatile long value = 0L;
}
```

=>

```
public final static class VolatileLong {
    public volatile long value = 0L;
    public long p1, p2, p3, p4, p5, p6;
}
```

False Sharing

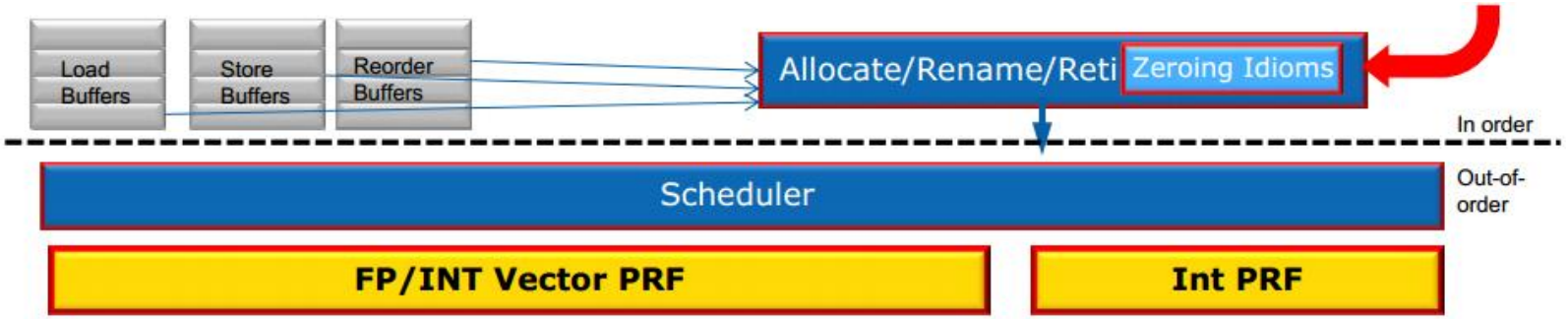



```
public final static class VolatileLong {
    public volatile long value = 0L;
}

=>

public final static class VolatileLong {
    public volatile long value = 0L;
    public long p1, p2, p3, p4, p5, p6;
}
```

Solved

Cache Consistency & 乱序执行

- CPU指令重排序



- 编译重排序

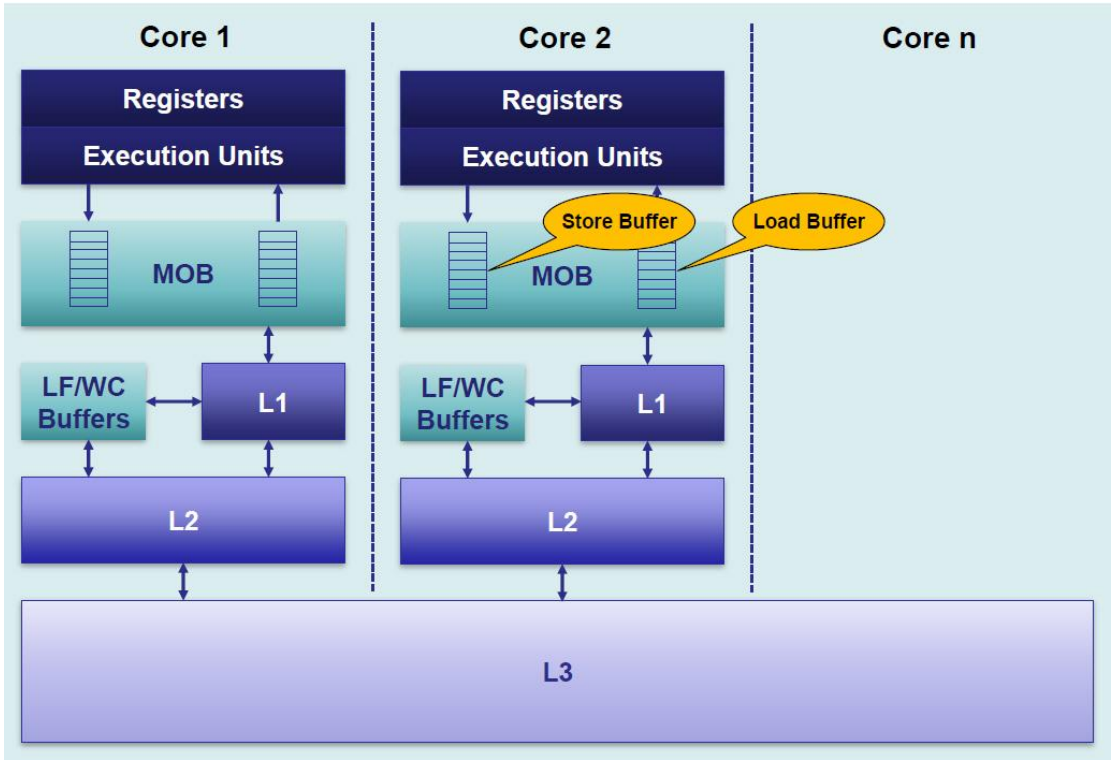
Memory Ordering

- 程序顺序(Program ordering): 读写和指令编制一致, 也称强顺序.
- 处理器顺序(Processor ordering): 加快指令执行速度, 例如读可跨越它之前的buffered写

- Intel386及之前内存顺序模型: 程序顺序
- Pentium和Intel486内存顺序模型:大多数情况下遵循程序顺序
- P6及以后内存顺序模型： 处理器顺序, 同时提供其它加强或弱化内存顺序模型的特殊指令

内存屏障

- Load Buffer
- Store Buffer
- CPU串行化指令
  - CPUID
  - SFENCE
  - LFENCE
  - MFENCE
- Lock系指令



Java内存模型

- Happens-Before
- StoreStore
- LoadStore
- StoreLoad
- LoadLoad
- Full Fence
- 锁和synchronized块
- volatile 关键字
- final 关键字

```
// Lock
pthread_mutex_lock(&lock);
sequence = i;
pthread_cond_signal(&condition);
pthread_mutex_unlock(&lock);

// Soft Barrier
asm volatile("" ::: "memory");
sequence = i;

// Fence
asm volatile("" ::: "memory");
sequence = i;
asm volatile("lock addl $0x0,(%rsp)");
```

Java内存模型

重排序	次操作		
首操作	Normal Load Normal Store	Volatile Load Monitor Enter	Volatile Store Monitor Exit
Normal Load Normal Store			No
Volatile Load Monitor Enter	No	No	No
Volatile store Monitor Exit		No	No

需要内存屏障	次操作			
首操作	Normal Load	Normal Store	Volatile Load MonitorEnter	Volatile Store
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load MonitorEnter	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store MonitorExit			StoreLoad	StoreStore

Example

```
1. class X {
2.   int a, b;
3.   volatile int v, u;
4.   void f() {
5.     int i, j;
6.
7.     i = a; // load a
8.     j = b; // load b
9.     i = v; // load v
10.            // LoadLoad
11.    j = u; // load u
12.            // LoadStore
13.    a = i; // store a
14.    b = j; // store b
15.            // StoreStore
16.    v = i; // store v
17.            // StoreStore
18.    u = j; // store u
19.            // StoreLoad
20.    i = u; // load u
21.            // LoadLoad
22.            // LoadStore
23.    j = b; // load b
24.    a = i; // store a
25.  }
26.}
```

```
$ java -XX:+UnlockDiagnosticVMOptions -XX:PrintAssemblyOptions=hsdis-print-bytes -
XX:CompileCommand=print,X.f X

...

[Verified Entry Point]
  0xb3a1e28c: push %ebp ;...55
  0xb3a1e28d: sub $0x8,%esp ;...81ec0800 0000
  0xb3a1e293: mov 0x10(%ecx),%ebx ;...8b5910   ; i = v
  0xb3a1e296: mov 0x14(%ecx),%edi ;...8b7914   ; j = u
  0xb3a1e299: mov %ebx,0x8(%ecx) ;...895908    ; a = i
  0xb3a1e29c: mov %edi,0xc(%ecx) ;...89790c    ; b = j
  0xb3a1e29f: mov %ebx,0x10(%ecx) ;...895910   ; v = i
  0xb3a1e2a2: mov %edi,0x14(%ecx) ;...897914   ; u = j
  0xb3a1e2a5: lock addl $0x0,(%esp) ;...f0830424  ; memory barrier
  0xb3a1e2aa: mov 0x14(%ecx),%ebp ;...8b6914   ; i = u
  0xb3a1e2ad: mov %ebp,0x8(%ecx) ;...896908    ; a = i
  0xb3a1e2b0: add $0x8,%esp ;...83c408
  0xb3a1e2b3: pop %ebp ;...5d
  0xb3a1e2b4: test %eax,0xb78ab000 ;...850500b0 8ab7

...
```

参考: jdk/hotspot/src/os_cpu/linux_x86/vm/orderAccess_linux_x86.inline.hpp

Puzzle 5

为什么有volatile还需要AtomicXX及Unsafe.compareAndSwapXX?

原子指令

- CompareAndSwap: LOCK XCHG
- LOCK XADD
- ABA问题

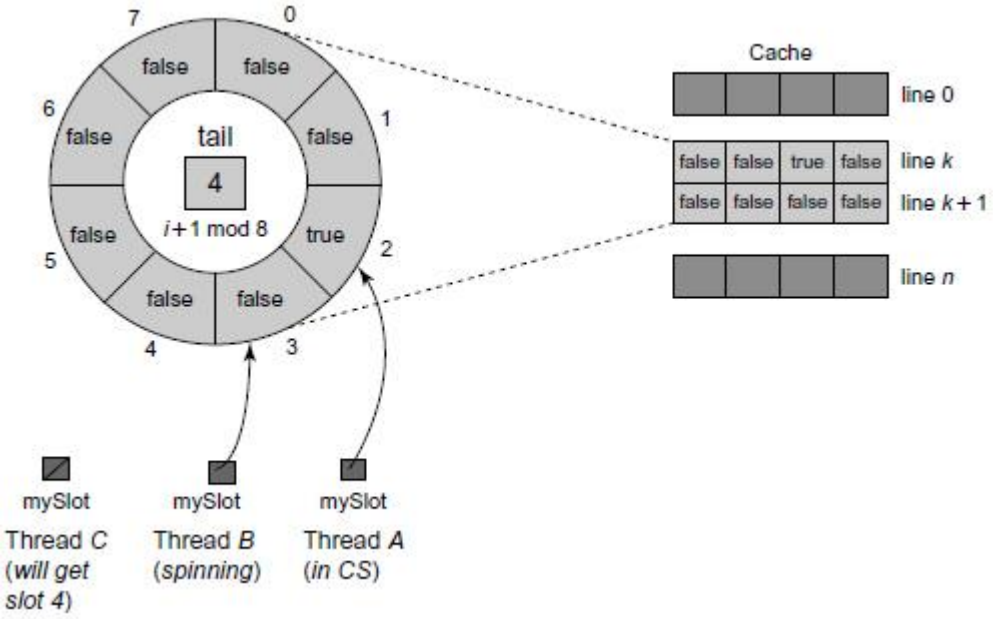
让步锁

```
public class BackoffLock {
  private AtomicBoolean state = new
AtomicBoolean(false);
  private static final int MIN_DELAY = ...;
  private static final int MAX_DELAY = ...;

  public void lock() {
    Backoff backoff = new Backoff(MIN_DELAY,
MAX_DELAY);
    while (true) {
      while (state.get()) {};
```

```
if (!state.getAndSet(true)) {
        return;
      } else {
        backoff.backoff();
      }
    }
  }

  public void unlock() {
    state.set(false);
  }

}
```

队列锁

- BackOff锁的问题
  - Cache-coherence Traffic
  - 临界区利用率
  - 公平性问题
  - 饥饿
- 基于数组的队列
- CLH队列
- MCS队列
- NUMA-Aware变种



https://blogs.oracle.com/dave/entry/flat_combining_numa_locks

Puzzle 6

为什么j.u.c或者jvm里面在加锁的时候, 喜欢用先CAS spin尝试N次, 如果失败则yield多次(可选), 如果最后还是失败则park, 同时把线程加到队列里?

工具

- top
- vmstat
- lscpu
- perf
- Valgrind tools suite
- OProfile
- SystemTap
- numactl
- Intel Vtune
- MAT

推荐读物

- What every programmer should know about memory
- Intel® 64 and IA-32 Architectures Software Developer Manuals
- The Art of Multiprocessor Programming
- The JSR-133 Cookbook for Compiler Writers (Java Memory Model)
- 本人博客: http://coderplay.javaeye.com

Q & A

作者：周忱 | 淘宝综合业务
微博：@MinZhou
邮箱：zhouchen.zm@taobao.com

淘宝网 Taobao.com