JMM

马士兵

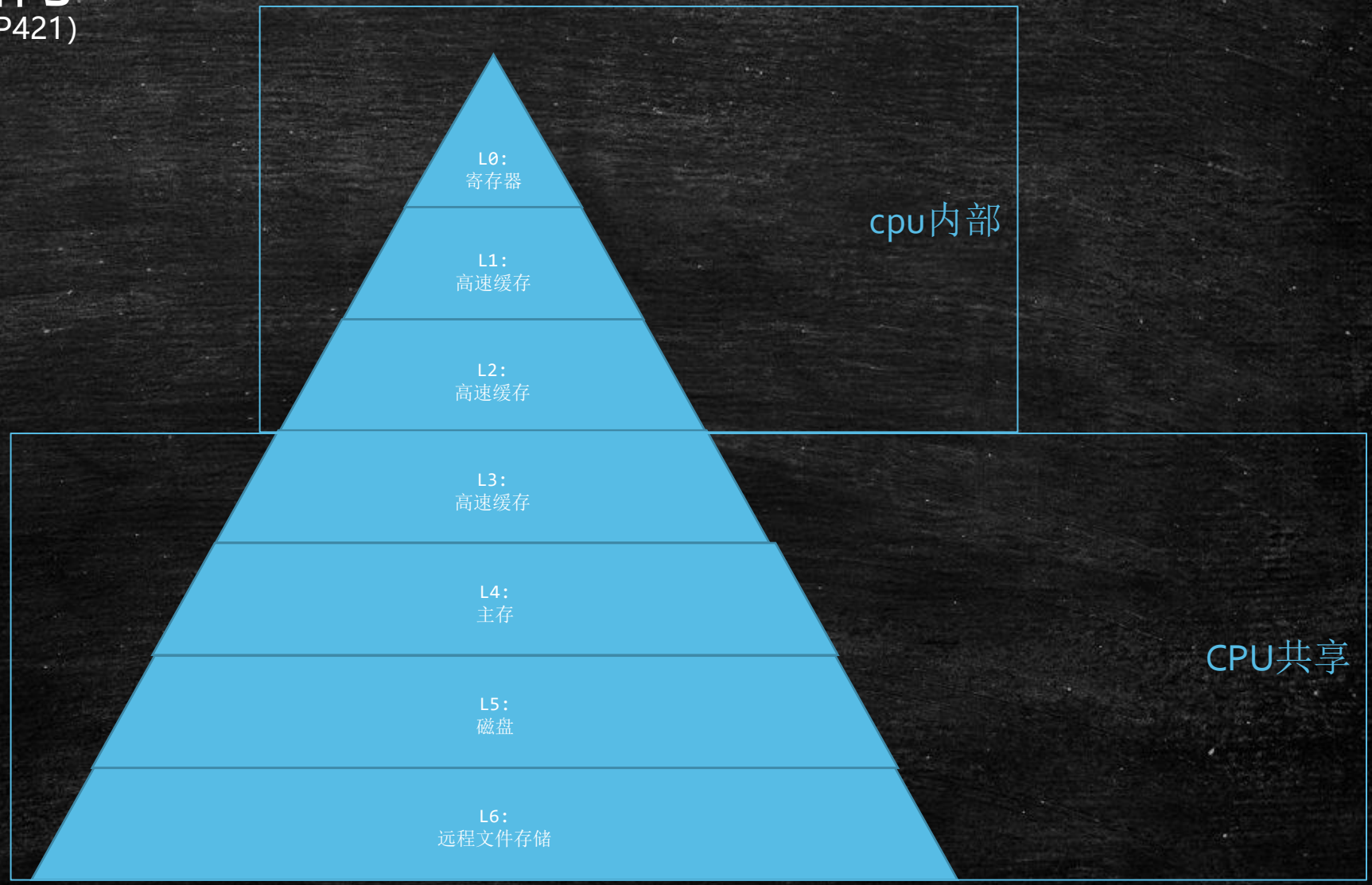
硬件层的并发优化基础知识

存储器的层次结构

（深入理解计算机系统 原书第三版 P421）

更小 更快 成本更高

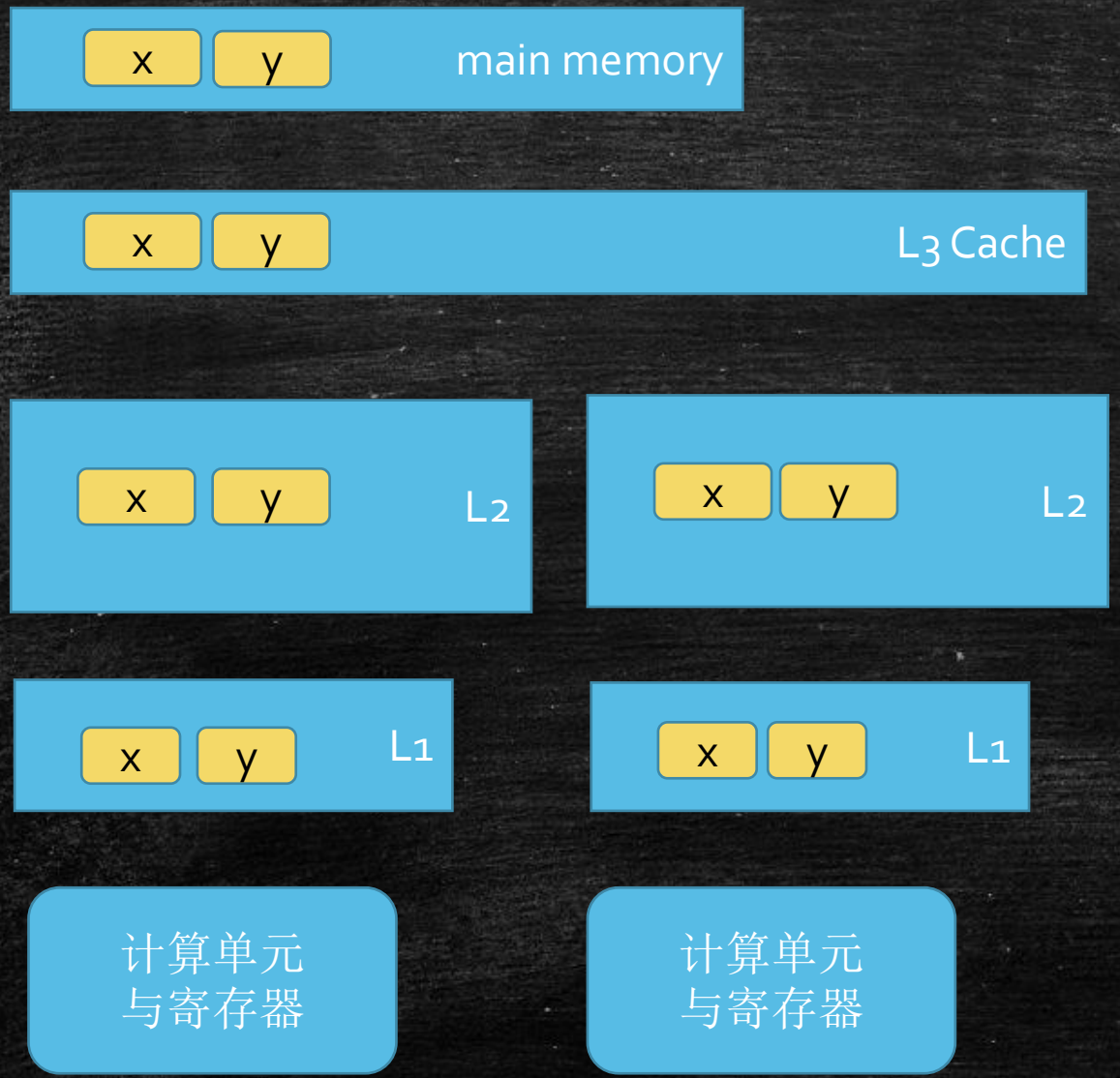
更大 更慢 成本更低



http://mashibing.com

从CPU到	大约需要的 CPU 周期	大约需要的时间
主存		约60-80纳秒
QPI 总线传输 (between sockets, not drawn)		约20ns
L3 cache	约40-45 cycles,	约15ns
L2 cache	约10 cycles,	约3ns
L1 cache	约3-4 cycles,	约1ns
寄存器	1 cycle	

cache line的概念 缓存行对齐 伪共享

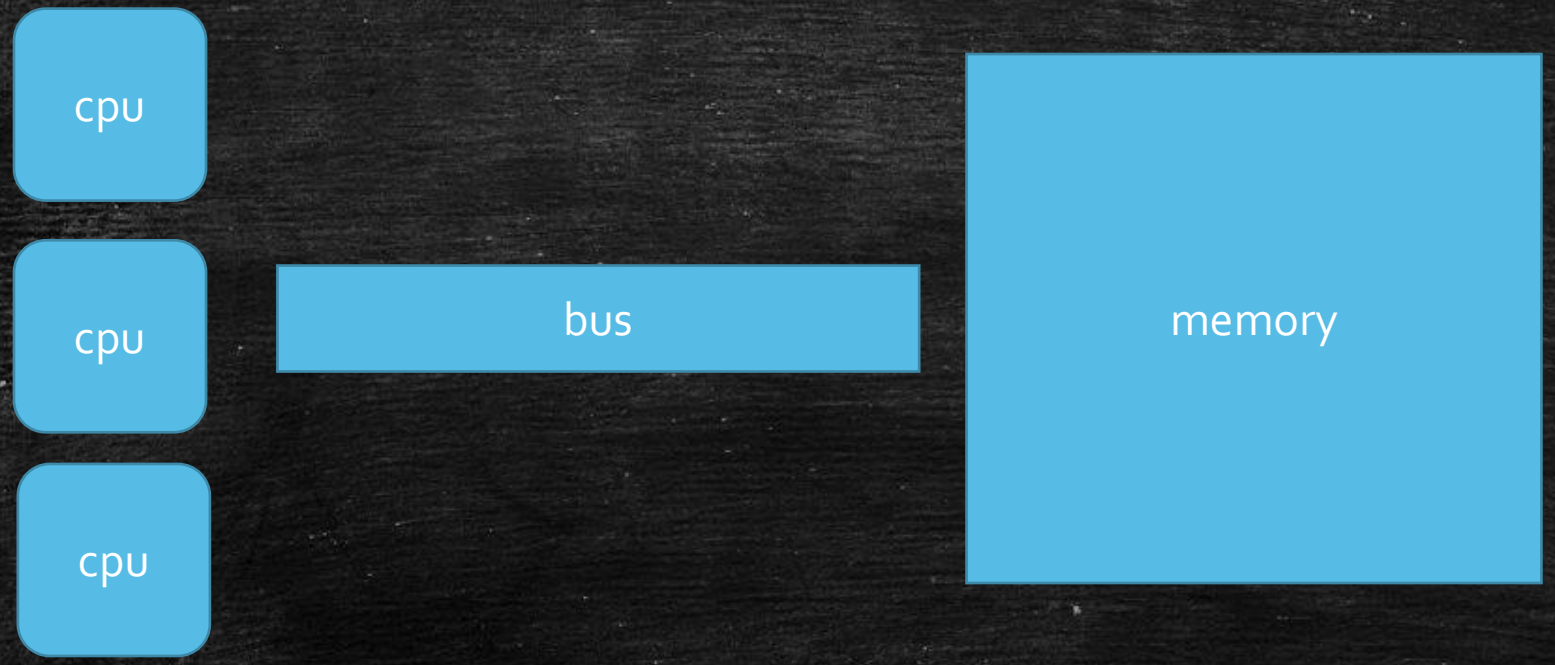


JUC/c_028_FalseSharing

disruptor

```
public long p1, p2, p3, p4, p5, p6, p7; // cache line padding
    private volatile long cursor = INITIAL_CURSOR_VALUE;
    public long p8, p9, p10, p11, p12, p13, p14; // cache line padding
```

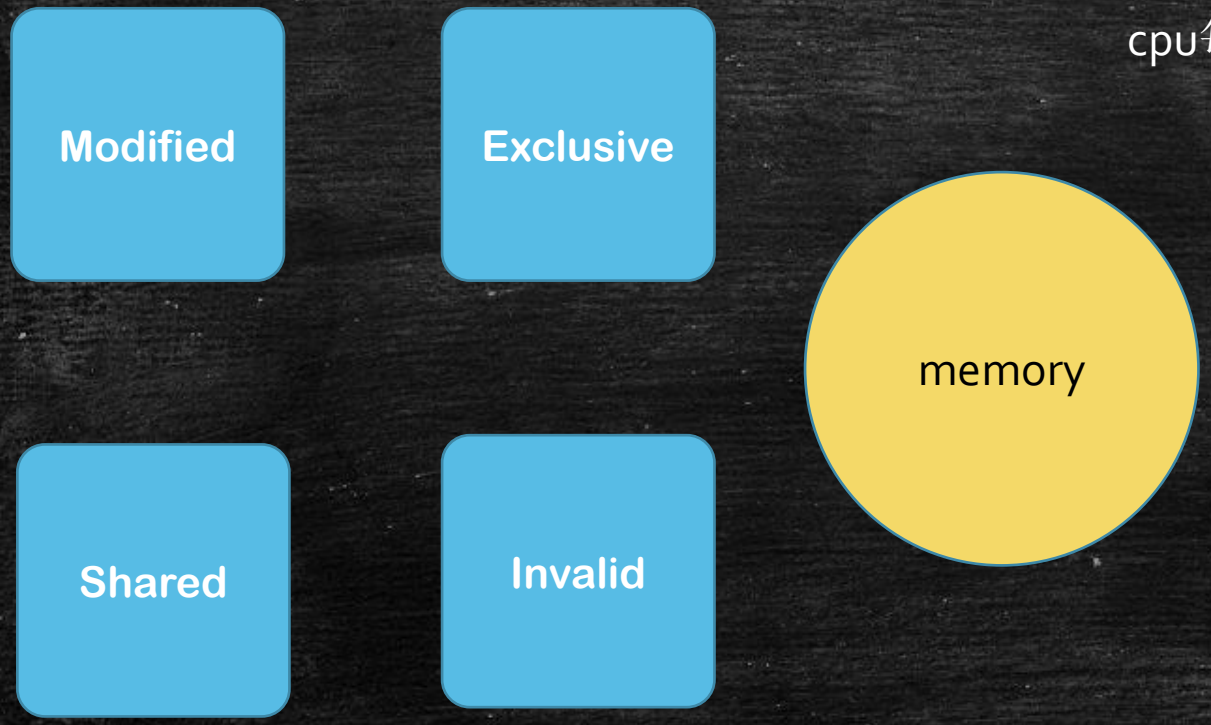
多线程一致性的硬件层支持



总线锁会锁住总线，使得其他CPU甚至不能访问内存中其他的地址，因而效率较低

MESI Cache一致性协议

https://www.cnblogs.com/z00377750/p/9180644.html



cpu每个cache line标记四种状态（额外两位）

缓存锁实现之一
有些无法被缓存的数据
或者跨越多个缓存行的数据
依然必须使用总线锁

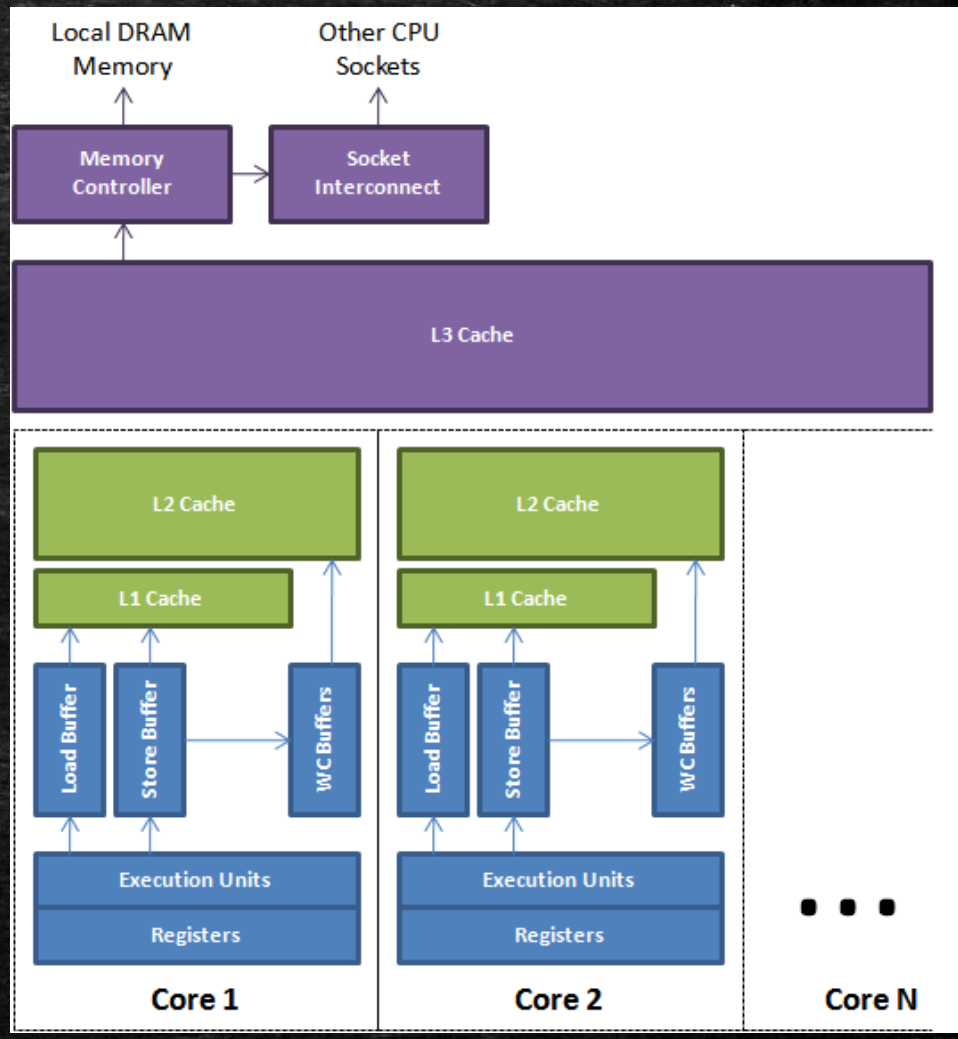
MSI MESI MOSI Synapse Firefly Dragon

cpu的读等待同时指令执行
cpu乱序执行的根源
https://www.cnblogs.com/liushaodong/p/4777308.html

读指令的同时可以同时执行不影响的其他指令
而写的同时可以可以进行合并写
WCBuffer

这样CPU的执行就是乱序的

必须使用Memory Barrier来做好指令排序
volatile的底层就是这么实现的（windows是lock指令）



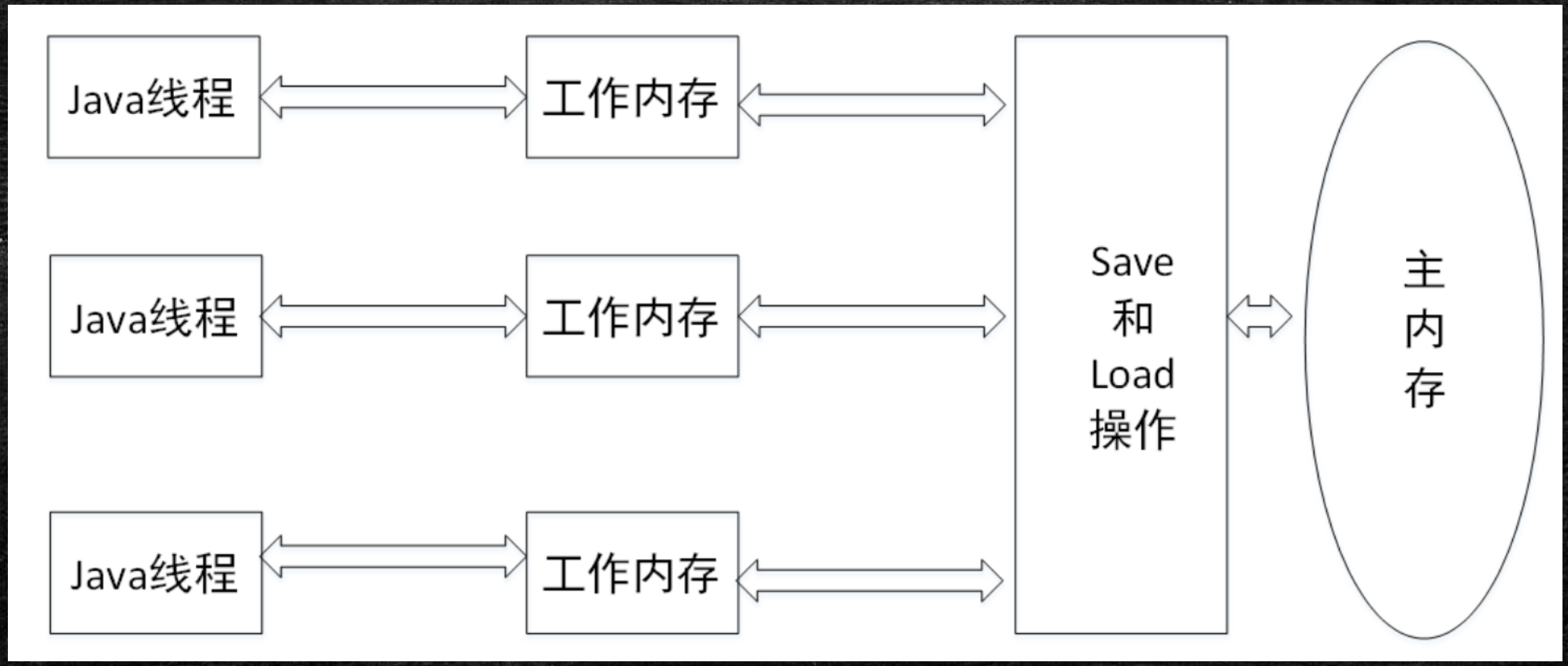
jmm/Disorder.java

https://preshing.com/20120515/memory-reordering-caught-in-the-act/

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
第2728842次 (0,0)

Process finished with exit code 0
```

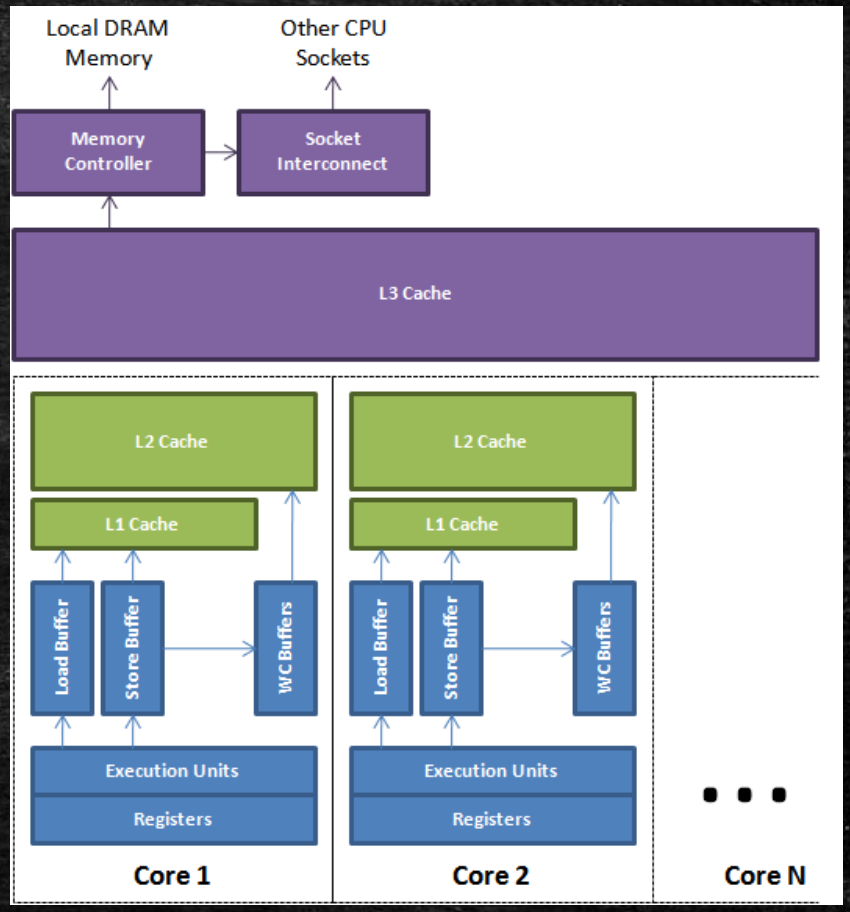
```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
第113299次 (0,0)

Process finished with exit code 0
```

java并发内存模型



cpu合并写的技术

https://www.cnblogs.com/liushaodong/p/4777308.html



JUC/028_WriteCombining

保障有序性

有序性保障
X86 CPU内存屏障

sfence:在sfence指令前的写操作当必须在sfence指令后的写操作前完成。
lfence：在lfence指令前的读操作当必须在lfence指令后的读操作前完成。
mfence：在mfence指令前的读写操作当必须在mfence指令后的读写操作前完成。

有序性保障
intel lock汇编指令

原子指令，如x86上的”lock …”指令是一个Full Barrier，执行时会锁住内存子系统来确保执行顺序，甚至跨多个CPU。Software Locks通常使用了内存屏障或原子指令来实现变量可见性和保持程序顺序

JSR内存屏障

LoadLoad屏障：

对于这样的语句Load1; LoadLoad; Load2，

在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。

StoreStore屏障：

对于这样的语句Store1; StoreStore; Store2，

在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。

LoadStore屏障：

对于这样的语句Load1; LoadStore; Store2，

在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。

StoreLoad屏障：对于这样的语句Store1; StoreLoad; Load2，

在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。

volatile的实现细节
一：编译器层面

jmm/TestVolatile.java

volatile int j

Name:	cp_info #6	<j>
Descriptor:	cp_info #5	<I>
Access flags:	0x0040 [volatile]	

volatile的实现细节
二：JVM层面

StoreStoreBarrier
volatile 写操作
StoreLoadBarrier

LoadLoadBarrier
volatile 读操作
LoadStoreBarrier

volatile的实现细节
三：操作系统及硬件层面

使用hsdis观察汇编码
lock指令 xxx 执行 xxx指令的时候保证对内存区域加锁

https://blog.csdn.net/qq_26222859/article/details/52235930

synchronized实现细节一：编译器层面

monitor enter

monitor exit

synchronized 的实现细节
二：JVM层面

C C++ 的锁实现
操作系统的一些辅助类和数据结构

synchronized 的实现细节
三：CPU层面

https://blog.csdn.net/21aspnet/article/details/88571740

使用lock comxchg实现

java8大原子操作（虚拟机规范）
（已弃用，了解即可）
最新的JSR-133已经放弃这种描述，但JMM没有变化
《深入理解Java虚拟机》P364

lock：主内存，标识变量为线程独占
unlock：主内存，解锁线程独占变量
read：主内存，读取内容到工作内存
load：工作内存，read后的值放入线程本地变量副本
use：工作内存，传值给执行引擎
assign：工作内存，执行引擎结果赋值给线程本地变量
store：工作内存，存值到主内存给write备用
write：主内存，写变量值

hanppens-before原则（JVM规定重排序必须遵守的规则）
JLS17.4.5

- 程序次序规则：同一个线程内，按照代码出现的顺序，前面的代码先行于后面的代码，准确的说是控制流顺序，因为要考虑到分支和循环结构。
- 管程锁定规则：一个unlock操作先行发生于后面（时间上）对同一个锁的lock操作。
- **volatile变量规则：对一个volatile变量的写操作先行发生于后面（时间上）对这个变量的读操作。**
- 线程启动规则：Thread的start( )方法先行发生于这个线程的每一个操作。
- 线程终止规则：线程的所有操作都先行于此线程的终止检测。可以通过Thread.join( )方法结束、Thread.isAlive( )的返回值等手段检测线程的终止。
- 线程中断规则：对线程interrupt( )方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupt( )方法检测线程是否中断
- 对象终结规则：一个对象的初始化完成先行于发生它的finalize()方法的开始。
- 传递性：如果操作A先行于操作B，操作B先行于操作C，那么操作A先行于操作C

as if serial

不管如何重排序，单线程执行结果不会改变

对象的内存布局

一线互联网企业面试题：
关于对象

1. 请解释一下对象的创建过程？
2. 对象在内存中的存储布局？
   - JavaAgent_AboutObject.md
3. 对象头具体包括什么？
   - JavaAgent_AboutObject.md
4. 对象怎么定位？
   - https://blog.csdn.net/clover_lily/article/details/80095580
5. 对象怎么分配？
   - GC相关内容
6. Object o = new Object在内存中占用多少字节？
   - JavaAgent_AboutObject.md

1. 对象的创建过程

1. class loading
2. class linking (verification, preparation, resolution)
3. class initializing
4. 申请对象内存
5. 成员变量赋默认值
6. 调用构造方法<init>
    1. 成员变量顺序赋初始值
    2. 执行构造方法语句

有关markword的细节

markword的结构，定义在markOop.hpp文件：

```
  32 bits:
  --------
  hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object)
  JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
  size:32 ---------------------------------------->| (CMS free block)
  PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)

  64 bits:
  --------
  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted object)
  size:64 ----------------------------------------------------->| (CMS free block)

  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && normal object)
  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && biased object)
  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS promoted object)
  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS free block)
[ptr             | 00]  locked             ptr points to real header on stack
[header      | 0 | 01]  unlocked           regular object header
[ptr             | 10]  monitor            inflated lock (header is wapped out)
[ptr             | 11]  marked             used by markSweep to mark an object

```

COOPs Compressed Ordinary Object Pointers

markword 64位

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否偏向锁	锁标志位
无锁态	对象的hashCode		分代年龄	0	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量(重量级锁)的指针				10
GC标记	空				11
偏向锁	线程ID	Epoch	分代年龄	1	01

1. hashCode部分：
   31位hashcode -> System.identityHashCode(…)
   按原始内容计算的hashcode，重写过的hashcode方法计算的结果不会存在这里。

   如果对象没有重写hashcode方法，那么默认是调用**os::random**产生hashcode,可以通过System.identityHashCode获取；os::random产生hashcode的规则为:next_rand = (16807*seed) mod (2*31-1),因此可以使用31位存储;另外一旦生成了hashcode,JVM会将其记录在markword中；

   什么时候会产生hashcode?当然是调用未重写的hashcode()方法以及System.identityHashCode的时候

   问题：为什么GC年龄默认为15？（最大为15）

学生问题

需要注意下，当调用锁对象的 `Object#hash` 或 `System.identityHashCode()` 方法会导致该对象的偏向锁或轻量级锁升级。这是因为在Java中一个对象的hashcode是在调用这两个方法时才生成的，如果是无锁状态则存放在 `mark word` 中，如果是重量级锁则存放在对应的monitor中，而偏向锁是没有地方能存放该信息的，所以必须升级。

参考资料：

https://cloud.tencent.com/developer/article/1480590
https://cloud.tencent.com/developer/article/1484167
https://cloud.tencent.com/developer/article/1485795
https://cloud.tencent.com/developer/article/1482500

顺带补充

Synchronized和ReentrantLock的区别

原理弄清楚了，顺便总结了几点Synchronized和ReentrantLock的区别：

1. Synchronized是JVM层次的锁实现，ReentrantLock是JDK层次的锁实现；
2. Synchronized的锁状态是无法在代码中直接判断的，但是ReentrantLock可以通过 `ReentrantLock#isLocked` 判断；
3. Synchronized是非公平锁，ReentrantLock是可以是公平也可以是非公平的；
4. Synchronized是不可以被中断的，而 `ReentrantLock#lockInterruptibly` 方法是可以被中断的；
5. 在发生异常时Synchronized会自动释放锁（由javac编译时自动实现），而ReentrantLock需要开发者在finally块中显示释放锁；
6. ReentrantLock获取锁的形式有多种：如立即返回是否成功的tryLock(),以及等待指定时长的获取，更加灵活；