

多核编程入门

作者：chengjia4574@gmail.com

sinaweibo: jiayy

时间：2012-8-8

说明：本文是多核编程的入门资料汇总，来源主要是国外国内的一些网站及自己使用过程中一些记录，写作目的主要是内部分享用（@NSFOCUS）。在多核使用过程中，得益于很多网络资源，所以也把自己整理的产品无关的东西共享出来，希望对多核感兴趣的同学可以入门用。

目录

一. 并发与并行的区别？

1

1.1 串

行

1 1.2

并发

1 1.3

并行

1 1.4 多核编程的难点

2

二.

多核体系架构

3

2.1

多核处理器定义

3 2.2

多核发展趋势

3 2.3

一个多核处理器架构例子

5 2.4 LINUX

线程核绑定

6

2.4.1 核亲和性绑定

6

2.4.2 资源控制 cgroup

8

三.

内存模型

8

3.1

操作原子性

9

3.1.1 原子性的 3 种保证机制

9

3.1.2 硬件原子操作

9

3.1.3 总线锁-原子操作原语

12 3.2

缓存一致性.....

16

3.2.1 定义

16

3.2.2 CC 协议

17

3.2.3 伪共享

21 3.3

顺序一致性.....

24

3.3.1 定义

24

3.3.2 几种顺序约束

25

3.3.3 乱序执行和 内存屏障

28

四.

并发级别.....

31

4.1 WAIT-FREEDOM 无等待并发

32 4.2 LOCK-FREEDOM

无锁并发.....

32 4.3 OBSTRUCTION-FREEDOM 无阻塞并发

33 4.4 BLOCKING ALGORITHMS

阻塞并发.....

33

五. 锁

34

5.1

信号量.....

34 5.2 自

旋锁.....

35 5.3

读写锁.....

35 5.4

顺序锁.....

37 5.5

RCU.....

38

六.

无锁编程.....

38

6.1

定义.....

39

七. 并发数据结构、

开源库.....

41

7.1 一些开源的并发库

41 7.2 一次无锁哈希表跟基于锁的哈希表性能对比测试

..... 41

7.2.1 测试平台

41

7.2.2 测试过程

42

7.2.3 哈希算法

43

7.2.4	测试结果
44	
八.	
多核工程实践	
44	
8.1	网络设备 : INTEL
DPDK	
44	8.2
网络游戏	
44	8.3
手机开发	
45	

九. 参考

45

表格索引

表 3.1 CC 示意图

24 表 3.2 CC 示意图 2

插图索引

图 1.1	并发和并行的区别.....	
2 图 2.1	PC 和手机核心增长趋势图.....	
4 图 2.2	最新的 MAC PRO 已经配备 12 个核心.....	4
图 2.3	三星推出 了 8 核心的手机处理器.....	
4 图 2.4	共享缓存多核处理器体系架构图实例.....	5
图 2.5	处理器各组件功能说明.....	
5 图 2.6	共享缓存多核处理器架构缓存示意图.....	6
图 3.1	DPDK, CAS 实现代码.....	
14 图 3.2	DPDK: 原子 ADD 实现代码.....	

15 图 3.3 DDPK: 原子自

增实现代码.....

15 图 3.4 MESI 协议

17 图 3.5 MOESI

状态机.....

18 图 3.6 CC 协议示例代码

18 图 3.7

初始状态.....

19 图 3.8 X

已经写入缓存.....

20 图 3.9 X增加了 1 0 0 1 0

20 图 3.10 CORE1 从 CORE0 的缓存里读走数据

21 图 3.11

伪共享.....

22 图 3.12

缓存行伪共享.....

23 图 3.13

缓存行填充.....

23 图 3.14

一些体系架构的内存顺序标准.....

27 图 3.15

强内存顺序模型和弱内存顺序模型一些例子

编译乱序和运行乱序.....	27	图	3.16
28	图	3.17	
乱序执行.....			
30	图	3.18	
内存屏障.....			
31	图	4.1	
几种并发级别的对比.....			
34	图	5.1	
读写锁.....			
35	图	5.2	
申请读锁.....			
36	图	5.3	
释放读锁.....			
36			
图	5.4		
申请写锁.....			
37	图	5.5	
释放写锁.....			
37	图	6.1	
什么是无锁编程.....			

一.

并发与并行的区别？

首先了解几个概念：

1.1 串行

最基本的程序执行方式，

串行程序的整个运行时，

只有一个调用栈和一个运行时上下文。

单进程/单线程程序可以认为是串行程序。

1.2 并发

多线程出现后比较常见的程序执行方式，多线程程序运行时，会有多个运行时上下文和对应的多个调用栈。逻辑上多个线程同时发生，物理上是由操作系统调度，CPU 某一时刻依然只执行一个线程的任务，但是某个执行中的线程随时可能被 OS 调度走，而随后运行的线程操作的数据可能跟刚刚被调度走的线程造成冲突，所以有共享数据同步问题。

多进程如果有共享数据,也符合并发程序的特点.相对于多线程并发,多进程并发解耦更彻底,数据分割更清晰。

另外，前述并发情况是从用户程序的角度，从内核的角度，还会有其他类似的场景，比如中断。

中断处理程序和中断之前执行的程序也有可能共享数据冲突的问题。

1.3 并行

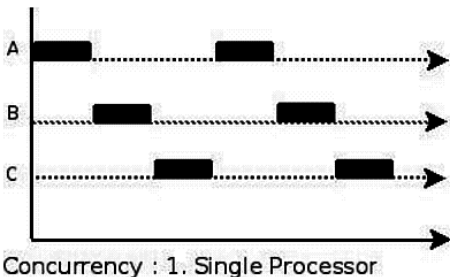
多核处理器出现后会越来越常见的程序执行方式，物理上多个任务可以同时运行，这个概念介于操作系统和体系架构之间，从操作系统而言，依然是调度多个线程/进程去 CPU 执行，只不过有了多个 CPU/核心，不同线程/进程可以绑定从而完全占用一颗核心，所以从体系架构 的角度，同一时刻是有多个任务同时运行，另外一些说法，如‘多处理器程序’‘多核程序’都可以认为属于并行程程序的范畴。

从概念的范围看，并行<并发，即并行的程序肯定是并发的，并发的程序不一定是并行的。

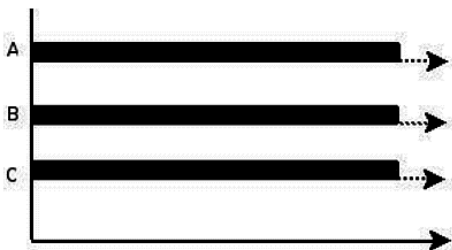
但是， 无论是逻辑上的并发 （单处理器，多线程/多进程） 还是物理上的并发 （并行，多处理 器） ，

所面临的共享数据操作一致性问题是一样的，
在很多情况下，多核编程可以近似认为

是多线程编程，所以本文对 ‘并发’ ‘多线程’ ‘多核’ ‘并行’ 等词汇没有做严格的区分，
根据上下文可以自行理解。



2. logically simultaneous processing



Parallelism : 1. Multiprocessores, Multicore

2. Physically simultaneous processing

图 1.1 并发和并行的区别

1.4 多核编程的难点

难点： 同时从宏观和微观两种角度分析问题，并能灵活在两种角度之间切换。

在之前单处理器的世界里，
计算机科学在编译器优化、

处理器优化、

多线程编程等方面有很深厚的积累，形成了对编程人员而言非常抽象的各种基础库，编程人员无须知道编译器、体系结构、OS 的多线程互斥同步等的实现细节。

进入多处理器后，这一个高度抽象的基础库还没有形成，虽然有一些并行库或者开发套件，但抽象程度还不够，理想上 OS 和编译器应该能对应用开发人员完全屏蔽多核体系架构的特点，但目前无法做到。多核程序开发人员目前需要自己解决核绑定、cgroup 划分、缓存对齐、跨物理 CPU 内存访问以及其他一些 OS 和体系结构层面需要考虑的问题(尤其是性能优化时)

本文的目的是为对多核编程感兴趣的同学提供入门级别的学习材料，

主要包括两个方面：1)

尝试解释清楚几个最基本的概念，

这些概念广泛存在于各种论文、

技术文档甚至源代码注

释里，如果不掌握这些概念，

会对某些多核的关键代码不知其所以然。

2) 提供收集到的比较好的源代码和博客目录，

需要注意的是，多核编程目前属于比较冷门的

领域（对比 app，游戏开发），

好的代码和资料还是比较难收集的。

二. 多核体系架构

2.1 多核处理器定义

多内核处理器架构是指：

芯片设计工程师在单个处理器中集成两个或多个“执行内核（即计算引擎）”。

多内核处理器可直接插入到单一处理器基座中。

但是，操作系统会把它的每个执行

内核作为独立的逻辑处理器，

为其分配相应的执行资源。

要利用多核处理器的运算能力，需

要改写操作系统和编译器，广泛使用的 vista, win7

等都能支持多核体系架构。[百度百科]

2.2 多核发展趋势

首先思考一个问题：

为什么微处理器要从单核转向多核？答案是：
功耗问题限制了单核不断提高性能的发展途径。

有几个简单的公式可以说明这个问题：

1) 处理器性能 = 主频 * IPC ,

主频是指每秒时钟周期数,比如 1Ghz,是每秒 10
亿个时钟周

期。IPC 是每个时钟周期可以执行的指令数。

2) 处理器功耗 正比 电流*电压^2*主频,

而主频正比电压, 所以

处理器功耗 正比 主频^3 ,

通过主频提升性能, 要面临功耗以 3
次方的指数增长问题。

所以主频发展到一定程度后,

自然转到重点依靠提高 ipc 来提升性能, 提升 IPC

可以通过

提高指令的并行度实现，提高并行度，
一是提高微处理器微架构的并行度，二是采用多核
架构，（参考：

http://blogs.intel.com/china/2007/06/03/post_5/）

前者已经发展了很多

年，提升空间和投入产出比明显不如后者，
所有多核处理器是未来的方向。

下图是某公司对 PC 和手机核数的统计和预测，
不管是 PC 端还是移动端，多核的趋势非常明 显，
到目前（2013）市面上 4 核的手机已经非常普遍。

	Number of cores				
	(45 nm) Current	(32nm) 2012	(24 nm) 2014	(19 nm) 2016	(15 nm) 2018
Mobile-devices	1	2	3-4	5-6	9-10
servers	8	16	28	44	72

图 2.1 PC 和手机核心增长趋势图

下面两张图看出家庭版 PC
和手机核心数目也很快突破 10 个。

Apple announces new Mac Pro with cylindrical design, 12-core Intel Xeon E5 CPU, flash storage, Thunderbolt 2.0 and support for up to three 4K displays

By **Daniel Cooper** posted Jun 10th, 2013 at 2:00 PM

图 2.2 最新的 mac pro 已经配备
12 个核心

Samsung announces the 8- core Exynos 5 Octa mobile processor

图 2.3 三星推出了 8

核心的手机处理器

虽然商用多核（multicore）和众核（many-core）系统越来越普遍，成本也越来越低，游戏设备、手机等移动设备也具备越来越多的核心，并发和并行越来越成为必要的技术手段，但多核程序的发展依然没有跟上硬件的发展，很多游戏引擎和网络引擎都还是单线程的。原因就是第一章提到的多核编程的难度。

2.3 一个多核处理器架构例子

Figure 5-1. Intel Xeon Processor

E7-8800/4800/2800 Product Families Block

Diagram

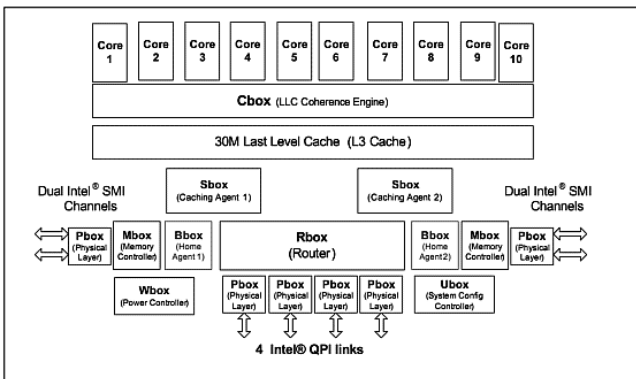


图 2.4 共享缓存多核处理器体系架构图实例

Name	Function
Core	Intel Xeon Processor E7-8800/4800/2800 Product Families core architecture processing unit
Bbox	Home Agent
Cbox	Last Level Cache Coherency Engine
Mbox	Integrated Memory Controller
Pbox	Physical Layer (PHY) for the Intel® QPI Interconnect and Intel® SMI Interconnect memory controller
Rbox	Crossbar Router
Sbox	Caching Agent
Ubox	System Configuration Agent
Wbox	Power Controller
Intel® SMI	Intel® Scalable Memory Interconnect (formerly "FBD2" or "Fully Buffered DIMM 2 interface")
Intel® QPI	Intel® QuickPath Interconnect
LLC	Last Level Cache (Level 3)

图 2.5 处理器各组件功能说明

这是基于共享缓存的多核体系架构的一个例子，一共有 10 个核心，不需要深入了解，这张图唯一的目的是给大家一个概念，

现代的处理器的架构已经比几十年前的冯诺依曼体系复杂多了（各种 box），里边稍微值得关注的是 Cbox，Bbox，这两个组件是缓存控制器，负责非常核心的功能：缓存一致性。

缓存一致性会在第三章描述。

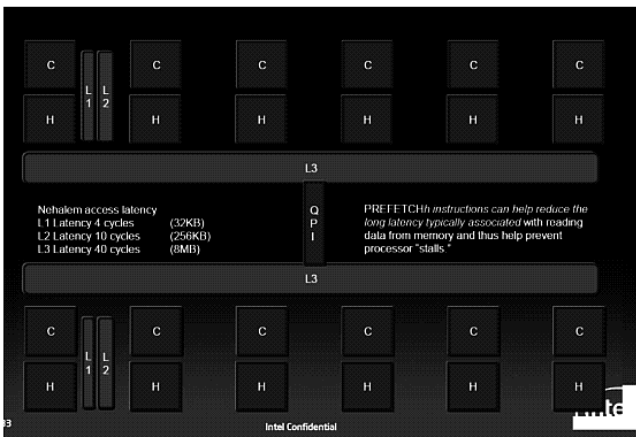


图 2.6 共享缓存多核处理器架构缓存示意图

这张图是 intel 多核体系架构（双路）的缓存示意图，每个 core 拥有自己的 L1 和 L2 缓存，属于一个物理 CPU 的 core 共享 L3 缓存。不同 cpu 之间通过 QPI 交互 L3 数据。每个 CPU 有自己的内存控制器。对多核编程而言，

缓存是非常重要的底层概念。

2.4 Linux 线程核绑定

这么多核具体到编程时候是如何使用的？linux 目前提供了若干机制可以让用户程序使用多核，其中包括进程/线程的核亲和性绑定，以及 cpuset 的资源控制

2.4.1 核亲和性绑定

在 linux 平台提供了核亲和性机制，进程和线程都可以通过设置亲和性绑定到不同的核心上。

进程版本：

```
#include <sched.h>

void setProcessToCPU(int _cpuID)
{
```

```
cpu_set_t mask;
```

```
cpu_set_t get;
```

```
CPU_ZERO(&mask);
```

```
CPU_SET(_cpuID, &mask);
```

```
if (sched_setaffinity(0, sizeof(mask), &mask) < 0) {
```

```
    cout << "set process affinity failed\n" << endl;
```

```
}
```

```
CPU_ZERO(&get);
```

```
if (sched_getaffinity(0, sizeof(get), &get) < 0) {
```

```
    cout << "get process affinity failed\n" << endl;
```

```
}
```

```
}
```

线程版本:

```
#include <pthread.h>
```

```
#include <sched.h>
```

```
void setThreadToCPU(int _cpuID)
{
    cpu_set_t mask;
    cpu_set_t get;

    CPU_ZERO(&mask);
    CPU_SET(_cpuID, &mask);

    if (pthread_setaffinity_np(pthread_self(),
sizeof(mask), &mask) < 0) {
        cout << "set thread affinity failed\n" <<
endl;
    }

    CPU_ZERO(&get);
    if (pthread_getaffinity_np(pthread_self(),
sizeof(get), &get) < 0) {
        cout << "get thread affinity failed\n" <<
endl;
    }
}
```

2.4.2 资源控制 cgroup

Cgroups 是 control groups 的缩写，是 Linux 内核提供的一种可以限制、记录、隔离进程组（process groups）所使用的物理资源（如：cpu,memory,IO 等等）的机制。最初由 google 的工程师提出，后来被整合进 Linux 内核。

Cgroups 也是 LXC 为实现虚拟化所使用的资源管理手段，可以说没有 cgroups 就没有 LXC。

Cgroup 有一个子系统 cpuset，这个子系统为 cgroup 中的任务分配独立 CPU（在多核系统）和内存节点。

<http://www.cnblogs.com/lisperl/archive/2012/04/17/2453838.html>

linux cgroups 详解一至 九

三. 内存模型

问题 1：内存模型是什么？

答案：CPU 硬件有它自己的内存模型，不同的编程语言也有它自己的内存模型。如果用一句 话来介绍什么是内存模型，我会说它就是程序员，编程语言和硬件之间的一个契约，它保证了共享的内存地址里的值在需要的时候是可见的，它保证了机器执行代码的结果与程序员脑子里的预期是一致的。它最大的作用是取得可编程性与性能优化之间的一个平衡。
[parallellabs.com]

问题 2：多核并行编程为什么需要关注内存模型？

答案：目前缺少一个完整的支持并发的内存模型，使得开发多核程序的程序员不得不面对内存操作的一些细节。从语言来说，.net , java 基于虚拟机的语言其内存模型相对完整，c,c++ 的内存模型在并发相关的领域上貌似还不是很完整（引自某个网友）

问题 3：在多核并发环境里，
需要关注内存模型的哪些方面？

答案： 操作原子性、缓存一致性、
顺序一致性。非常重要的一点思想是：
你的代码不 一定向你想象中那样执行

3.1 操作原子性

原子性操作是指，在整个系统可见范围内，一个操作要不就没有发生，要不就执行完毕，没有中间状态出现。

3.1.1 原子性的 3 种保证机制

在多线程程序里，哪些操作具有天然的原子性？哪些操作需要原子操作原语的支持？原子操作原语底层机制是什么？

要回答这些问题，我们首先需要从硬件讲起。以常见的 X86 CPU 来说，根据 Intel 的参考手册，它基于以下三种机制保证了操作原子性（8.1 节）：

（1）Guaranteed atomic operations（注：8.1.1 节有详细介绍）

(2) Bus locking, using the LOCK# signal and the LOCK instruction prefix

(3) Cache coherency protocols that ensure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors

这三个机制相互独立，相辅相成。

简单的理解起来就是

(1)

一些基本的内存读写操作是本身已经被硬件提供了原子性保证（例如读写单个字节的操 作）；

(2) 一些需要保证原子性但是没有被第(1)条机制提供支持的操作（例如 read-modify-write）可以通过使用“LOCK#”来锁定总线，从而保证操作的原子性。

原子操作原语是基于“LOCK#”总线锁实现的。

(3) 因为很多内存数据是已经存放在 L1/L2 cache 中了，对这些数据的原子操作只需要与本 地的 cache 打交道，而不需要与总线打交道，所以 CPU 就提供了 cache coherency 机制来保证其它的那些也 cache 了这些数据的 processor 能读到最新的值。

下面分别对以上 3 点进行说明

3.1.2 硬件原子操作

那么 CPU 对哪些(1)中的基本的操作提供了原子性支持呢？根据 Intel 手册 8.1.1 节的介绍：从 Intel486 processor 开始，以下的基本内存操作是原子的：

- Reading or writing a byte（一个字节的读写）
- Reading or writing a word aligned on a 16-bit boundary（对齐到 16 位边界的字的读写）•

Reading or writing a doubleword aligned on a 32-bit boundary (对齐到 32 位边界的双字的 读写)
从 Pentium processor 开始,

除了之前支持的原子操作外又新增了以下原子操作:

- Reading or writing a quadword aligned on a 64-bit boundary (对齐到 64 位边界的四字的 读写)
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus (未缓存且 在 32 位数据总线范围之内的内存地址的访问)

从 P6 family processors 开始,

除了之前支持的原子操作又新增了以下原子操作:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line (对 单个 cache line 中缓存地址的未对齐的 16/32/64 位访问)

那么哪些操作是非原子的呢?

Accesses to cacheable memory that are split across bus widths, cache lines, and page

boundaries are not guaranteed to be atomic by the Intel Core 2 Duo, Intel® Atom™, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. (说点简单点, 那些被总线带宽、cache line 以及 page 大小给分隔开了 的内存地址的访问不是原子的, 如果你想保证这些操作是原子的, 你就得求助于机制 (2), 对总线发出相应的控制信号才行)。

需要注意的是尽管从 P6 family 开始对一些非对齐的读写操作已经提供了原子性保障, 但是非对齐访问是非常影响性能的, 需要尽量避免。当然了, 对于一般的程序员来说不需要太担心 这个, 因为大部分编译器会自动帮你完成内存对齐。

下面从一道题的分析充分理解硬件原子操作：

以下多线程对 *int* 型变量 *x* 的操作，
哪几个需要进行同步：（ ）

A. $x=y$; B. $x++$; C. $++x$; D. $x=1$;

我们先反汇编一下看看它们到底执行了什么操作：

$x = y$;

mov eax,dword ptr [y]

mov dword ptr [x],eax

$x++$;

mov eax,dword ptr [x]

add eax,1

mov dword ptr [x],eax

$++x$;

```
mov eax,dword ptr [x]  
add eax,1  
mov dword ptr [x],eax
```

```
x = 1;  
mov dword ptr [x],1
```

(1) 很显然，**x=1** 是原子操作。

因为 **x** 是 **int** 类型，32 位 CPU 上 **int** 占 32 位，在 X86 上由硬件直接提供了原子性支持。实际上不管有多少个线程同时执行类似 **x=1** 这样的赋值语句，**x** 的值最终还是被赋的值（而不会出现例如某个线程只更新了 **x** 的低 16 位然后被阻塞，另一个线程紧接着又更新了 **x** 的低 24 位 然后又被阻塞，从而出现 **x** 的值被损坏了的情况）。

(2) 再来看 **x++**和**++x**。

其实类似 $x++$, $x+=2$, $++x$

这样的操作在多线程环境下是需要同步的。因为 X86 会按三条指令 的形式来处理这种语句：

从内存中读 x 的值到寄存器中，对寄存器加 1，
再把新值写回 x 所处 的内存地址

（见上面的反汇编代码）。

例如有两个线程，它们按照如下顺序执行（注意读 x 和写回 x 是原子操作，两个线程不能同时执行）：

time	Thread 1	Thread 2
0	load eax, x	
1		load eax, x
2	add eax, 1	add eax, 1
3	store x, eax	
4		store x, eax

我们会发现最终 x 的值会是 1 而不是 2，因为 Thread 1 的结果被覆盖掉了。这种情况需要借助概述中的机制 2 来实现操作原子性。

(3) 最后来看看 $x=y$ 。

在 X86 上它包含两个操作：读取 y 至寄存器，再把该值写入 x 。读 y 的值这个操作本身是原子的，把值写入 x 也是原子的，

但是两者合起来是不是原子操作呢？我个人认为 $x=y$ 不是原子操作，因为它不是不可再分的操作。但是它需要不需要同步呢？

其实问题的关键在于程序的上下文。

例如有两个线程，线程 1 要执行 $\{y = 1; x = y;\}$ ，线程 2 要执行 $\{y = 2; y = 3;\}$ ，假设它们按如下时间顺序执行：

time	Thread 1	Thread 2
0	store y, 1	
1		store y, 2
2	load eax, y	
3		store y, 3

4 **store x, eax**

那么最终线程 1 中 x 的值为 2，

而不是它原本想要的 1。

我们需要加上相应的同步语句确保 $y = 2$ 不会在线程

1 的两条语句之间发生。 $y = 3$ 那条语句尽管在 **load**

y 和 **store x** 之间执行，但是 却不影响 $x=y$

这条语句本身的语义。所以你可以说 $x=y$ 需要同步，

也可以说 $x=y$ 不需要同步，看你怎么理解题意了。

$x=1$ 是否需要同步也是一样的道理，

虽然它本身是原子操作，

但是如果有另一个线程要读 $x=1$ 之后的值，

那肯定也需要同步，否则另一个线程读到的就是 x

的旧值而不是 1 了。

[参考: <http://www.parallellabs.com> 并行实验室]

3.1.3 总线锁-原子操作原语

对于硬件无法保证的原子操作，
可以通过原子操作原语来保证，
原子操作原语一般要基于总线锁实现：

在 x86 平台上，CPU
提供了在指令执行期间对总线加锁的手段。CPU
芯片上有一条引
线#HLOCK pin，
如果汇编语言的程序中在一条指令前面加上前缀"LOCK"，
经过汇编以后的
机器代码就使 CPU
在执行这条指令的时候把#HLOCK pin 的电位拉低，
持续到这条指令结束时放开，从而把总线锁住，
这样同一总线上别的 CPU
就暂时不能通过总线访问内存了，保证

了这条指令在多处理器环境中的原子性

常见的原子操作原语如下

3.1.3.1 CAS

这是最常见的原子操作原语。

在不同系统下可能有以下命名：

CAS, compare-and-exchange,
compare-and-set,
std::atomic_compare_exchange,
InterlockedCompareExchange,
__sync_val_compare_and_swap, LOCK
CMPXCHG and other.

在某些论文里经常看到 RMW
(read-modify-write)，CAS 就是一种 RMW，
其伪代码是

```

T compare-and-swap(T* location, T cmp,
T xchg)
{
    do atomically
    {
        T val = *location;
        if (cmp == val)
            *location = xchg;
        return val;
    }
}

```

这种 RMW 会将一个新的值 `xchg` 放入地址 `location` 如果该地址是期望的值 `cmp`。否则返回 `location` 的值。

下图是 `dpdk` 的 CAS 实现代码，第一句 `MPLOCKED` 其实是 ‘lock’ 指令，就是锁总线，

确 保同一时间只有一个 CPU 线程能写这块内存，
然后是 `cmpxchgl` 指令，用于比较并交换操作 数，
这个指令是原子的。写完之后，通过 `cache`
一致性模型保证所有核心看到和操作的是同
一块实际内存的值而不是自己缓存内的值。
最后一句 ‘:”memory”’ 是内存屏障，内存屏障属
于顺序一致性的内容，参考第三节。
从这里也可以看出，原子操作原语的 ‘锁’
实际上被移到了 CPU 内部实现。另外一个需要注
意的是，目标内存 `dst` 必须是 `volatile` 修饰的，
意思是编译器每次遇到这个变量都必须从内存 读值，
而不能从核心自己的缓存或寄存器读值。
总结起来就是：

**volatile + lock 指令 + cmpxchgl 指令 +
缓存一致性模型 + memory 指令 =
原子操作原语 CAS**

其他原语的原理只需要将上面公式的 **cmpxchgl** 替换成其他硬件指令即可

```
static inline int
rte_atomic32_cmpset(volatile uint32_t
*dst, uint32_t exp, uint32_t src){
    uint8_t res;
    asm volatile(
        " " MPLOCKED " "
        " cmpxchgl %[src], %[dst] ;

        " " sete %[res] ; "
        : [res] "=a" (res), /* 0 */
          [dst] "=m" (*dst)
          : [src] "r" (src),
            "a" (exp),
            "m" (*dst)
            : "memory");

    return res;
}
```

3.1.3.2 fetch-and-add

```
/* 1 */  
/* 2 */  
/* 3 */  
/* 4 */
```

Dpdk, CAS 实现代码

也是一种 RMW, 在不同系统下可能有以下命名:

`atomic_fetch_add`, `InterlockedExchangeAdd`,

`LOCK XADD`

伪代码

```

T fetch-and-add(T* location, T x)
{
    do atomically
    {
        T val = *location;
        *location = val + x;
        return val;
    }
}

```

同一类型的原语还有 `fetch-and-sub`, `fetch-and-and`, `fetch-and-or`, `fetch-and-xor`, 下面是 `dpdk` 的实现

```

static inline void
rte_atomic32_add
(rte_atomic32_t *v, int32_t
inc)
{
    asm volatile(MPLOCKED
        "addl %[inc], %[cnt]"
        : [cnt] "=m" (v->cnt) /* output
        (0) */: [inc] "ir" (inc), /*

```



```

input (1) */ "m" (v->cnt) /*
                                input (2) */ );
/* no clobber-list */
}

```

图 3.2 Dpdk: 原子 add 实现代码

3.1.3.3 exchange

也是 RMW, `atomic_exchange, XCHG`

`T exchange(T* location, T x)`

```

{
    do atomically
    {
        T val = *location;
        *location = x;
        return val;
    }
}

```

将新值 x 放入位置 location, 将该位置的旧值返回。

Dpdk 没有这个原语的实现。

3.1.3.4 atomic loads and stores

非 RMW 的原子操作，如 `atomic_get`, `atomic_set`, `atomic_inc`, `atomic_dec` 等。下面是 dpdk 自增的原子操作原语实现，在一个总线锁内，对 `v->cnt` 这块内存进行加 1 操作

```
static inline void
rte_atomic32_inc
(rte_atomic32_t *v)
{
    asm volatile (MPLOCKED
"incl %[cnt]"
: [cnt] "=m" (v->cnt) /* output
(0) */: "m" (v->cnt) /*
input (1) */);
/* no clobber-list */
}
```

图 3.3 Dpdk: 原子自增实现代码

3.1.3.5 各个平台的原子操作原语

各个平台或者语言都推出了自己的原子操作原语实现，使用的时候可以直接调用相关 API

Windows

[http://msdn.microsoft.com/en-us/library/ms684122\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms684122(v=VS.85).aspx)

GNU

<http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html>

Solaris

<http://www.oracle.com/technetwork/indexes/documentation/index.html>

Java

<http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/atomic/package-summary.html>

.net

<http://msdn.microsoft.com/en-us/library/system.threading.interlocked.aspx>

Dpdk rte_atomic.h

3.2 缓存一致性

3.2.1 定义

缓存一致性 Cache coherence 简称 CC,

缓存一致性协议是在共享缓存多处理器架构确保最终一致性最突出、最重要的机制。

这些协议在缓存线 (cache-line) 级别实现了对一致性的保证。

缓存线是从主内存中读取数据和向内存中写入数据的缓存单位 (至少从一致性机制的角度看是这样的)。

商用处理器上三个最突出最重要的缓存一致性协议—**MOESI, MESI, and MESIF**—的缩写

都来自它们为缓存线定义的各种状态: **Modified** (已修改), **Owned** (被占用), **Exclusive**

(独占的), Shared (共享的), Invalid (无效的), and Forward (转发的)。缓存一致性协议在对内存确保最终一致性的内存一致性机制的帮助下对这些状态进行管理。

Intel 奔腾: MESI 协议

AMD opteron: MOESI 协议

Intel i7: MESIF 协议

问题: 为什么需要缓存 CC?

答案:

从第二章的[图 2.6]可以看到, 一般每个核心都有一个私有的 L1 级和 L2 级 Cache, 同一个物理 CPU 上的多个核心共享一个 L3 级缓存,

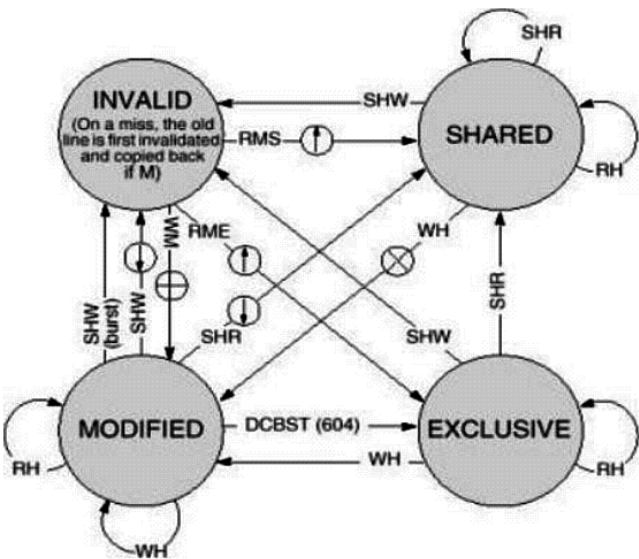
这样的设计是出于提高内存访问性能的考虑。但是这样就有一个问题了, 每个核心之间的私有 L1, L2 级缓存之间需要同步啊。比如, 核心 1 上的线程 A 对一个共享变量 `global_counter` 进行了加 1

操作，这个被写入的新值存到核心 1 的 L1 缓存里了；此时另一个核心 2 上的线程 B 要读 `global_counter` 了，但是核心 2 的 L1 缓存里的 `global_counter` 的值还是旧值，最新被写入的值现在还在核心 1 上。这就需要 CPU 有一个模块来保证，同一个内存的数据在同一时刻对任何对其可见的核心看来，数据是一致的，由第二章[图 2.5]知道，这种专门的组件就是缓存控制器（Cbox,Bbox），其实现了缓存一致性协议。

3.2.2 CC 协议

3.2.2.1 MESI

- MESI Protocol
 - 4 States
 - Invalid
 - 无数据
 - Shared
 - 与Memory一致的数据；
 - 多节点共享；
 - Exclusive
 - 与Memory一致的数据；
 - 单节点持有；
 - Modified
 - 最新修改数据，与memory不一致；
 - 单节点持有；



BUS TRANSACTIONS

- 状态转移
 - 见右图

RH = Read Hit	⬇ = Snoop Push
RMS = Read Miss, Shared	⊗ = Invalidate Transaction
RME = Read Miss, Exclusive	⊕ = Read-with-Intent-to-Modify
WH = Write Hit	⬆ = Cache Block Fill
WM = Write Miss	
SHR = Snoop Hit on a Read	
SHW = Snoop Hit on a Write or Read-with-Intent-to-Modify	

图 3.4 MESI 协议

详细了解参考：

Cache 一致性协议之 MESI：

<http://blog.csdn.net/muxiqingyang/article/details/6615199>

3.2.2.2 MOESI

下面是基于 MOESI 的一个例子，

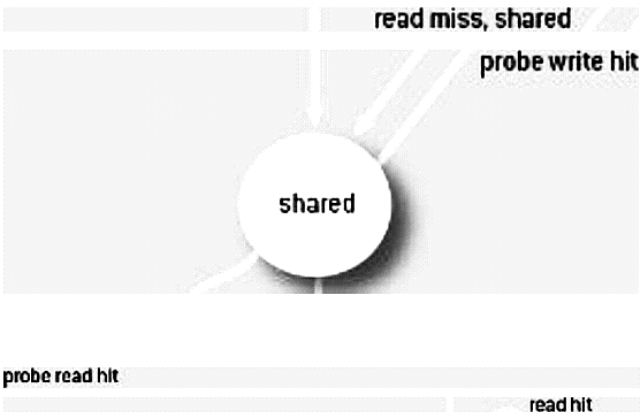
展示的是共享缓存多处理器中共享读写的生命周期。

原文在这里：

[<http://www.oschina.net/translate/nonblocking-algorithms-and-scalable-multicore-programming>]

FIGURE 1

The MOESI State Machine



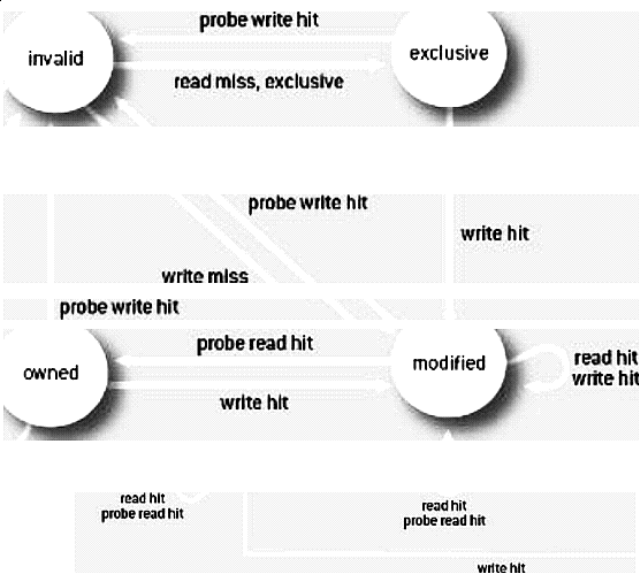


图 3.5 MOESI 状态机

```
volatile int x = 443;
static void * thread(void *unused)
{
    fprintf(stderr, "Read:
```

```
        return NULL;
}

int
main(void)
{
    pthread_t a;

    x = x + 10010;
    if (pthread_create(&a,
%d\n", x);
NULL, thread, NULL) != 0) {
        exit(EXIT_FAILURE);
    }

    pthread_join(a, NULL);
```

```
    return 0;
}
```

图 3.6 CC 协议示例代码

这个程序非常简单，定义了一个 `volatile` 变量 `x`，初始值是 `443`，然后程序开始后将 `x` 的值增加 `10010`，并启动一个线程打印 `x` 的值。在单处理器上，这个程序是非常简单的，下

面的过程显示的是两个核跑这个程序，`core0` 跑 `main`，`core 1` 跑 `thread` 的缓存状态机变迁过程。



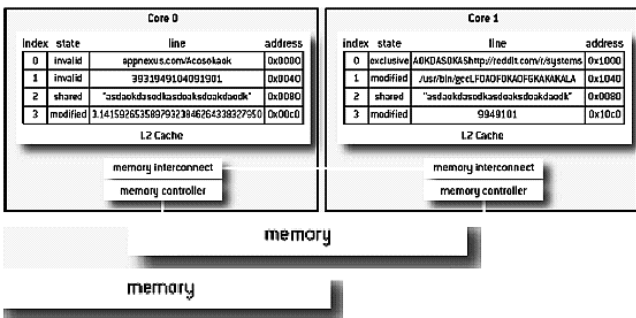


图 3.7 初始状态

上图是两个 core 的 L2 缓存的初始状态，假设每个 core 有一个 256 字节的 L2 缓存，每个 cacheline 长度是 64byte。可以看到初始状态下，两个 core 的 L2 缓存都已经满了，缓存线处于 invalid, shared, modified, exclusive 等多种状态中。

假设 `x` 的地址是 `0x20c4`，
 它被缓存散列函数哈希到 cacheline 3 上，

由初始状态图知道，该缓存线已经有一个数据 0x00c0，对 x 的 probe write hit 到这条缓存，这时候原有缓存的内容会被写回内存，cacheline3 状态变为 invalid，然后 x 的值被写入这个 cacheline，状态变成独占 exclusive，

FIGURE 3

After Initial Load of Variable x

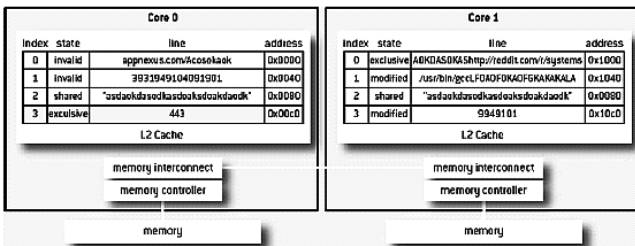


图 3.8 x 已经写入缓存

然后 core0 开始为 x 增加 10010，结果保存在 cacheline3 里，状态如下

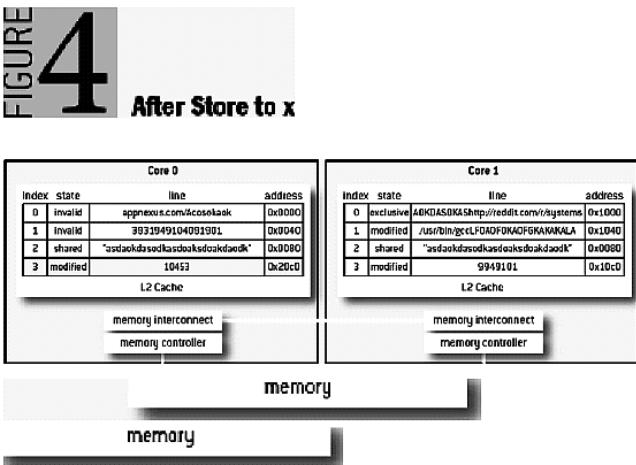


图 3.9 x 增加了 1 0 0 1 0

Core1 跑 thread 函数，需要读取 x 的值并调用 fprintf，这个载入动作会发出一个读取探测，要求从 midified 的状态转换为 owned 状态（如图 5 所示）。

MOESI 协议允许缓存到缓存的数据转移，所以 core1 从 core0 缓存里读走数据，同时 core1 的 cacheline3 变成 shared 状态

FIGURE 5 After Remote Read of x

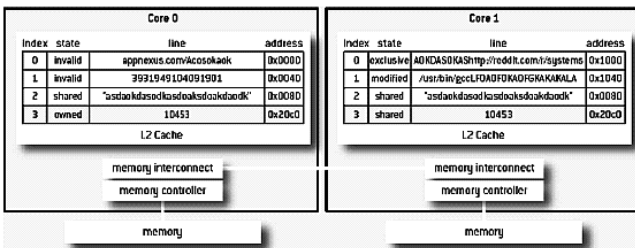


图 3.10 Core1 从 core0 的缓存里读走数据

3.2.2.3 MESIF

Intel 提出了另外一种 MESI 协议的变种，即 MESIF 协议，该协议与 MOESI 协议有较大的不同，也远比 MOESI 协议复杂，该协议由 Intel 的 QPI(QuickPath Interconnect)技术引入，其主要目的是解决“基于点到点的全互连处理器系统”的 Cache 共享一致性问题，而不是“基于共享总线的处理器系统”的 Cache 共享一致性问题。

在基于点到点互连的 NUMA(Non-Uniform Memory Architecture)处理器系统中，包含多个子处理器系统，这些子处理器系统由多个 CPU 组成。如果这个处理器系统需要进行全机 Cache 共享一致性，该处理器系统也被称为 ccNUMA(Cache Coherent NUMA)处理器系统。MESIF 协议主要解决 ccNUMA 处理器结构的 Cache 共享一致性问题，这种结构通常使用目录表，

而 不使用总线监听处理 Cache 的共享一致性。

关于 MESIF ，可以参阅陈怀临的“浅谈 intel qpi 的 MESIF 协议和 home,soure snoop”

http://www.360doc.com/content/10/1207/13/158286_75798413.shtml

3.2.3 伪共享

3.2.3.1 定义

从上一节可以知道，
缓存一致性协议操作的最小对象的缓存行，
缓存行内数据的修改、写入内存、
写入其他缓存等操作都会改变其状态，这样，
在共享缓存多核架构里，数据结构如果

组织不好，就非常容易

出现多个核线程反复修改同一条缓存行的数据导致缓存行状态频繁变

化从而导致严重性能问题,这就是伪共享现象。

下图就是一个伪共享的例子, core1

上运行的线程想修改变量 x,core2

上运行的线程想修改变量 y, 但 x 和 y

刚好在一个缓存行上。

每个线程都要去竞争缓存行的所有权来更新变量。

如果 核心 1 获得了所有权, 缓存子系统将会使核心

2 中对应的缓存行失效。当核心 2 获得了所有

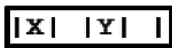
权然后执行更新操作, 核心 1 就要使自

己对应的缓存行失效。这会来来回回的经过 L3

缓存, 大大影响了性能。

如果互相竞争的核心位于不同的插槽,

就要额外横跨插槽连接, 问题可能 更加严重。



Local DRAM

Other CPU

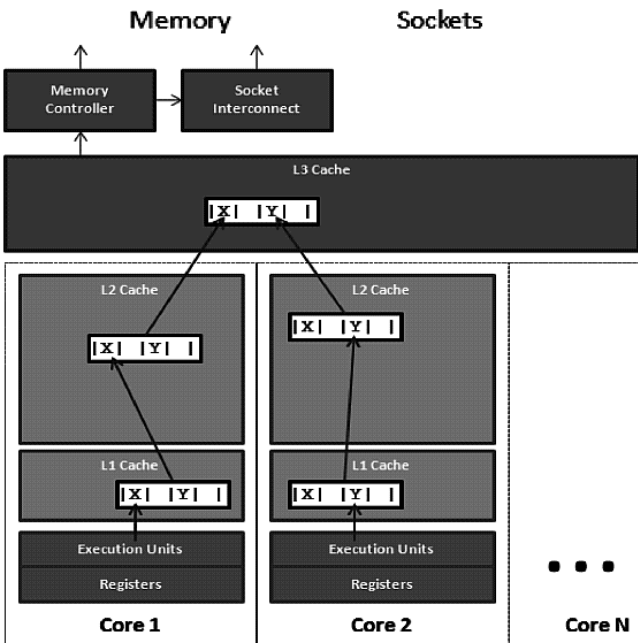


图 3.11 伪共享

3.2.3.2 解决

与缓存行导致性能问题的严重相比，对这个问题的解决方案显得非常简单，这就是**缓存行填充**，通过填充缓存行，使得某个核心线程频繁操作的数据独享缓存行，这样就不会出现伪共享问题了。下面是一个例子。

```
01 | #define N_THR 8
02 |
03 | struct counter {
04 |     unsigned long long value;
05 | };
06 |
07 | static volatile struct counter counters[N_THR];
08 |
09 | void *
10 | thread(void *unused)
11 | {
12 |
13 |     while (leave == false) {
14 |         counters[UNIQUE_THREAD_ID].value++;
15 |     }
16 |
17 |     return NULL;
18 | }
```

图 3.12 缓存行伪共享

```
1 struct counter {  
2     unsigned long long value;  
3     char pad[64 - sizeof(unsigned long long)];  
4 };
```

图 3.13 缓存行填充

32 位机 long long 是 8 字节，这样一个缓存行 64 字节可以存 8 个 counter，这样最差的情况下同时会有 8 个线程争夺同一个缓存行的操作权，性能会非常低。解决方式非常简单，如图 3.13 所示，每个 counter 变量增加一个填充变量 pad，使得一个 counter 变量刚好是一个缓存行大小，这样数组 counters 每个元素占用一个缓存行，所有线程独占自己的缓存行，避免了伪共享问题。

3.3 顺序一致性

3.3.1 定义

Sequential consistency，简称 SC，定义如下

... 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 appear in this sequence in the order specified by its program [lamport]

下面用一个小例子说明这个定义的意思：

假设我们有两个线程（线程 1 和线程 2）

分别运行在两个 CPU 上，有两个初始值为 0 的全局

共享变量 x 和 y ，

两个线程分别执行下面两条指令：

初始条件： $x = y = 0$;

表 3.1 CC 示意图

线程 1	线程 2
x = 1;	y=1;
r1 = y;	r2 = x;

因为多线程程序是交错执行的，
所以程序可能有如下几种执行顺序：

表 3.2 CC 示意图 2

Execution 1	Execution 2	Execution 3
x = 1; r1 = y; y = 1; r2 = x; 结果:r1==0 and r2 == 1	y = 1; r2 = x; x = 1; r1 = y; 结果: r1 == 1 and r2 == 0	x = 1; y = 1; r1 = y; r2 = x; 结果: r1 == 1 and r2 == 1
Execution 1	Execution 2	Execution 3

当然上面三种情况并没包括所有可能的执行顺序，
但是它们已经包括所有可能出现的结果了，
所以我们只举上面三个例子。

我们注意到这个程序只可能出现上面三种结果，

但是不可能出现 $r1==0$ and $r2==0$ 的情况。

SC 其实就是规定了两件事情：

(1)

每个线程内部的指令都是按照程序规定的顺序
(**program order**) 执行的 (单个线

程的视角)

(2) 线程执行的交错顺序可以是任意的，
但是所有线程所看见的整个程序的总体执行顺序都是一样的 (整个程序的视角)

第一点很容易理解，就是说线程 1

里面的两条语句一定在该线程中一定是 $x=1$

先执行， $r1=y$ 后执行。第二点就是说线程 1 和线程

2 所看见的整个程序的执行顺序都是一样的，

举例子就是假设线程 1

看见整个程序的执行顺序是我们上面例子中的

Execution 1，那么线程 2 看见的

整个程序的执行顺序也是 Execution 1，不能是

Execution 2 或者 Execution 3。

有一个更形象点的例子。伸出你的双手，

掌心面向你，两个手分别代表两个线程，从食指到小拇指的四根手指头分别代表每个线程要依次执行的四条指令。

SC 的意思就是说：

（1）对每个手来说，

它的四条指令的执行顺序必须是从食指执行到小拇指

（2）你两个手的八条指令（八根手指头）

可以在满足（1）的条件下任意交错执行（例如可

以是左 1，左 2，右 1，右 2，右 3，左 3，左 4，

右 4，也可以是左 1，左 2，左 3，左 4，右 1，右

2，右 3，右 4，也可以是右 1，右 2，右 3，左 1，

左 2，右 4，左 3，左 4）其实说简单点，SC

就是我们最容易理解的那个多线程程序执行顺序的模型。

CC 保证的是对 一个地址访问的一致性，**SC** 保证的是对一系列地址访问的一致性。

3.3.2 几种顺序约束

顺序的内存一致性模型为我们提供了一种简单的并且直观的程序模型。

但是，这种模型实际

上阻止了硬件或者编译器对程序代码进行的大部分优化操作。

为此，人们提出了很多松弛的

(relaxed) 内存顺序模型，

给予处理器权利对内存的操作进行适当的调整。

例如 Alpha 处 理器， PowerPC

处理器以及我们现在使用的 x86, x64

系列的处理器等等。下面是一些内存顺 序模型

3.3.2.1 TSO (整体存储定序)

- 数据载入间的执行顺序不可改变。
- 数据存储间的顺序不可改变。
- 数据存储同相关的它之前的数据载入间的顺序不可改变。
- 数据载入同其相关的它之前的数据存储的顺序可以改变。
- 向同一个地址存储数据具有全局性的执行顺序。
- 原子操作按顺序执行。
- 这方面的例子包括 x86 TSO²⁶ 和 SPARC TSO.

3.3.2.2 PSO (部分存储定序)

- 数据载入间的执行顺序不可改变。
- 数据存储间的执行顺序可以改变。

- 数据载入同数据存储间相对顺序可以改变。
 - 向同一个地址存储数据具有全局性的执行顺序。
 - 原子操作同数据存储间的顺序可以改变。
- 这方面的例子包括 SPARC PSO.

3.3.2.3 RMO (宽松内存定序)

- 数据载入间的顺序可以改变。
 - 数据载入同数据存储间的顺序可以改变。
- 数据存储间的顺序可以改变。
- 向同一个地址存储数据具有全局性的执行顺序。
 - 原子操作同数据存储和数据载入间的顺序可以改变。
- 这方面的例子包括 Power²⁷ 和 ARM.⁷

Memory ordering in some architectures ^{[1][2]}

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA64	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
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

图 3.14 一些体系架构的内存顺序标准

WEAK

Really weak



**Weak with
data dependency
ordering**



DEC Alpha



ARM



C/C++11

PowerPC

low-level atomics



Source control
analogy



STRONG

Usually strong

(implicit acquire/
release & TSO, usually)



Sequentially
consistent

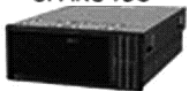
x86/64



dual 386 (circa 1989)



SPARC TSO



Java volatile

C/C++11
default atomics

Or, run on
a single core
without optimization

图 3.15 强内存顺序模型和弱
内存顺序模型一些例子

最左边的内存顺序一致性约束越弱，
右边的约束是在左边的基础上加上更多的约束，
X86/64 算是比较强的约束。

3.3.3 乱序执行和内存屏障

任何非严格满足 SC 规定的

内存顺序模型都产生所谓乱序执行问题，

从编程人员 的代码， 到编译器， 到 CPU 运行，
中间可能至少需要对代码次序做三次调整，
每一次调整都是为了最
终执行的性能更高。如下图

Source code

```
if (!PREACTION(gm)) {  
    void* mem;  
    size_t nb;  
    if (bytes <= MAX_SMALL_R  
        bindex_t idx;  
        binmap_t smallbits;  
        nb = (bytes < MIN_REQU  
        idx = small_index(nb);  
        smallbits = gm->smalllm  
  
        if ((smallbits & 0x3U)  
            schunkptr b, p;  
            idx += ~smallbits &  
            b = mem + 1; b; i = ...
```

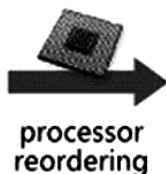


compiler
reordering

Machine code

Memory

cmp	esi,0F4h
ja	dimalloc+1AEh (774)
cmp	esi,0Bh
jae	dimalloc+20h (774)
mov	esi,10h
jnp	dimalloc+26h (774)
add	esi,0Bh
and	esi,0FFFFFFF8h
mov	eax,dword ptr [__
mov	edi,esi
shr	edi,3
mov	ecx,edi
shr	eax,cl
test	al,3
je	dimalloc+9Ah (774)



The hardware memory model matters here.

图 3.16 编译乱序和运行乱序

在串行程序里，编译器和 CPU

对代码所进行的乱序执行的优化对程序员都是封装好了的，无痛的，

所以程序员不需要关心这些代码在执行时被乱序成什么样子。

在并发程序里，

乱序执行可能会造成程序运行结果与编程人员预期不一致，

所以必须使用同步机制，类似 mutex ，

semaphore

等同步机制在实现的时候考虑了编译和执行的乱序问题，可以保证关键代码区不会被乱序执行。

另外一些编程场景，比如为了追求极致性能而自己写更高效的锁实现，或自己编写无锁程序，则会被暴露到这个问题面前。

下面通过一个例子解释乱序执行和内存屏障这两个概念。

[来源：

<http://preshing.com/20120625/memory-ordering-at-compile-time>]

示例代码：

```
int A, B;
```

```
void foo()
```

```
{
```

A = B + 1;

B = 0;

1

普通编译选项:

\$ gcc -S -masm=intel foo.c

\$ cat foo.s

...

mov eax, DWORD PTR B (redo this at home...)

add eax, 1

mov DWORD PTR A, eax

mov DWORD PTR B, 0

加上 -O2 优化编译选项，可以看到，B 的赋值操作顺序变了

\$ gcc -O2 -S -masm=intel foo.c

\$ cat foo.s

...

mov eax, DWORD PTR B

mov DWORD PTR B, 0

add eax, 1

mov DWORD PTR A, eax

上述情况在某些场景下导致的后果是不可接受的，比如下面这段伪代码中，生产者线程执行于一个专门的处理器之上，它先生成一条消息，然后通过更新 **ready** 的值，向执行在另外一个处理器之上的消费者线程发送信号，由于乱序执行，这段代码在目前大部分平台上执行是有问题的：处理器有可能会在将数据存储在 **message->value** 的动作执行完成之前和/或其它处理器能够看到 **message->value** 的值之前，执行 **consume** 函数对消息进行接收或者执行将数据保存到 **ready** 的动作。

```

01 volatile int ready = 0;
02
03 void
04 produce(void)
05 {
06     message = message_new();
07     message->value = 5;
08     message_send(message);
09     ready = 1;
10 }
11
12 void
13 consume(void)
14 {
15
16     while (ready == 0) {
17         ; /* Wait for ready to be non-zero. */
18     }
19
20     message = message_receive();
21     result = operation(&message->value);
22 }

```

图 3.17 乱序执行

回到之前的例子，加上一句内存屏障命令

int A, B;

```
void foo()
```

```
{
```

```
    A = B + 1;
```

```
    asm volatile("" ::: "memory");
```

```
    B = 0;
```

```
}
```

依然采用 o2 优化编译选项，发现这次 B 的赋值操作顺序没有变化

```
$ gcc -O2 -S -masm=intel foo.c
```

```
$ cat foo.s
```

```
...
```

```
    mov     eax, DWORD PTR B
```

```
    add     eax, 1
```

```
    mov     DWORD PTR A, eax
```

```
    mov     DWORD PTR B, 0
```

```
...
```

在内存顺序一致性模型不够强的多核平台上，
例子 2 的正确实现应该是下面这种，需要

加上两个内存屏障语句。

```
01 volatile int ready = 0;
02
03 void
04 produce(void)
05 {
06     message = message_new();
07     message->value = 5;
08     message_send(message);
09
10     /*
11      * Make sure the above memory operations complete before
12      * any following memory operations.
13      */
14     MEMORY_BARRIER();
15     ready = 1;
16 }
17
18 void
19 consume(void)
20 {
21
22     while (ready == 0) {<
23         ; /* Wait for ready to be non-zero */
24     }
25
26     /*
27      * Make sure we have an up-to-date view of memory relative
28      * to the update of the ready variable.
29      */
30     MEMORY_BARRIER();
31     message = message_receive();
32     result = operation(&message->value);
33 }
```

图 3.18 内存屏障

X86 的内存屏障 `#define barrier()`
`__asm__ __volatile__("" : : : "memory")`

更多 X86 内存屏障请参考：

<http://blog.csdn.net/cnctloveyu/article/details/5486339>

四. 并发级别

往往一个并发算法会说自己是 **wait-free** 的或者 **lock-free** 的，或者是 **non-blocking** 的，这些专有词汇其实表示的是并发的程度，或者说并发的级别。

并发级别的理解是阅读各种并发算法设计论文以及并发数据结构实现的必备基础。

4.1 Wait-freedom 无等待并发

Wait-freedom

指的是每一个线程都一直运行下去而无须等待外部条件，整个流程中任何操作 都能在一个有限的步骤内完成，这是最高的并发级别，没有任何阻塞。

结合上面原子操作部分的知识，
可以简单认为能够直接调用一个原子操作实现的算法或程序就属于 **Wait-free**，比如下面的 `increment_reference_counter` 函数就是 **wait-free** 的，它封装了 `atomic_increment` 这个原子自增原语，
多个线程可以同时调用这个函数对同一个内存变量进行自增，而无须任何阻塞（其实也是有阻塞的，

是总线锁级别)

与此做对比，CAS 类的调用就不是 wait-free 的，注意 wait-free 的原语都不能包含内部循环，而 CAS 原语使用时通常包含在“循环直到成功”的循环内部。

```
void increment_reference_counter(rc_base* obj)
{
    atomic_increment(obj->rc);
}
```

4.2 Lock-freedom 无锁并发

Lock-freedom

指的是整个系统作为一个整体一直运行下去，系统内部单个线程某段时间内可能会饥饿，这是比

wait-freedom 弱的并发级别，
但系统整体上看依然是没有阻塞的。所有 **wait-free**
的算法显然都满足 **lock-free** 的要求。

Lock-free 算法通常可以通过同步原语 **CAS** 实现。

```
void stack_push(stack* s, node* n)
{
    node* head;
    do
    {
        head = s->head;
        n->next = head;
    }

    while ( ! atomic_compare_exchange(s->head,
head, n));
}
```

多个线程同时调用上述函数，
理论上某个线程可以一直困在循环内部，
但一旦有一个线程原子操作失败而返回循环，
意味着有其他线程成功执行了原子操作而退出循环，
从而保证系统整体是没有阻塞的。

其实 6.1 的函数也可以用下面的原语实现，
在这种实现里，不再是所有线程都无阻塞了，某些
线程可能会因为 CAS 失败而回绕若干次循环。

```
void increment_reference_counter(rc_base* obj)
{
    Int rc;
    Do {
        rc = obj->rc;
    }
    while(!atomic_compare_exchange(obj->rc,rc,rc+1
));
}
```

4.3 Obstruction-freedom

无阻塞并发

Obstruction-free 是指在任何时间点，一个孤立运行线程的每一个操作可以在有限步之内结束。只要没有竞争，线程就可以持续运行，一旦共享数据被修改，**Obstruction-free** 要求中止已经 完成的部分操作，并进行回滚，**obstruction-free** 是并发级别更低的非阻塞并发，该算法在不出现冲突性操作的情况下提供单线程式的执行进度保证，所有 **Lock-Free** 的算法都是 **Obstruction-free** 的。

4.4 Blocking algorithms

阻塞并发

阻塞类的算法是并发级别最低的同步算法，它一般需要产生阻塞。可以简单认为基于锁的实现是 **blocking** 的算法。详细参考第五章

上述几种并发级别可以使用下图描述：

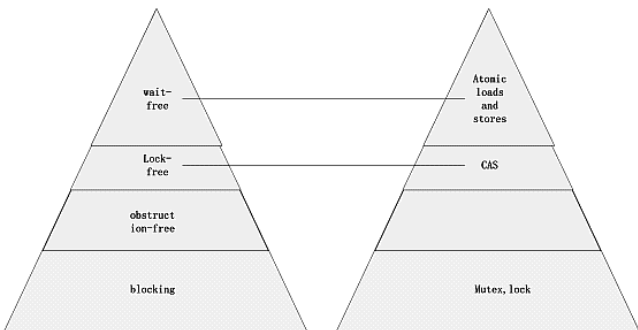


图 4.1 几种并发级别的对比

蓝色是阻塞的算法，绿色是非阻塞算法，

金字塔越上方，并发级别越高，性能越好，右边的金字塔是实现工具（原子操作、锁、互斥体等）

五. 锁

锁的概念是承接原子操作的，根本目的都是为了并发保护，只不过抽象的层次更高，多核环境下的锁的实现要依赖于第三章的理论基础。下面是几种常见的锁机制的简介，其中给出了读写锁的具体实现（来自 DPDK），其他锁的具体实现可以参考 linux 内核代码

5.1 信号量

Linux 中的信号量是一种睡眠锁。如果有一个任务试图获得一个已被持有的信号量时，

信号量会将其推入等待队列，然后让其睡眠。这时处理器获得自由去执行其它代码。当持有信号量的进程将信号量释放后，在等待队列中的一个任务将被唤醒，从而便可以获得这个信号量。

信号量的睡眠特性，使得信号量适用于锁会被长时间持有的情况；只能在进程上下文中使用，因为中断上下文中是不能被调度的；另外当代码持有信号量时，不可以再持有自旋锁。

5.2 自旋锁

自旋锁最多只能被一个可执行线程持有，如果一个执行线程试图请求一个已被争用（已经被持有）的自旋锁，那么这个线程就会一直进行忙循环——旋转——等待锁重新可用。要是锁未被争用，

请求它的执行线程便能立刻得到它并且继续进行。

自旋锁可以在任何时刻防止多于一个的执行线程同时进入临界区。

事实上，自旋锁的初衷就是：在短期间内进行轻量级的锁定。一个被争用的自旋锁使得请求它的线程在等待锁重新可用期间进行自旋（特别浪费处理器时间），所以自旋锁不应该被持有时间过长。另外自旋锁不允许任务睡眠（持有自旋锁的任务睡眠会造成自死锁），它能够在中断上下文中使用。

5.3 读写锁

读写锁实际是一种特殊的自旋锁，它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。这种锁相对于自旋锁而言，能提高并发性，

因为在多处理器系统中，
它允许同时有多个读者来访问共享资源，
最大可能的读者数为实际的逻辑 CPU 数。
写者是排他性的，
一个读写锁同时只能有一个写者或多个读者（与 CPU 数相关），但不能同时既有读者又有写者。

在读写锁保持期间也是抢占失效的。
如果读写锁当前没有读者，也没有写者，
那么写者可以立刻获得读写锁，否则它必须自旋在那里，直到没有任何写者或读者。

如果读写锁没有写者，
那么读者可以立即获得该读写锁，否则读者必须自旋在那里，直到写者释放该读写锁。

DPDK 里有一个读写锁的实现

```
typedef struct {
```

```
volatile int32_t cnt; /**< -1 when  
W lock held, > 0 when R locks held. */ }  
rte_rwlock_t;
```

图 5.1 读写锁

这把锁的核心是一个 `volatile int32` 的变量，
这个变量值为-1 的时候表示处于写锁状态，值大 于
0 表示处于读锁状态。

```
static inline void  
rte_rwlock_read_lock(rte_rwlock_t  
*rwl) {  
    int32_t x;  
    int success = 0;  
  
    while (success == 0) {  
        x = rwl->cnt;  
        /* write lock is held */  
        if (x < 0)  
            continue;
```

```

        success = rte_atomic32_cmpset(
(volatile uint32_t *) &rw1->cnt,
        x, x + 1);
    }
}

```

图 5.2 申请读锁

通过原子 **CAS** 操作实现读锁获取，
 多个线程可以同时调用这个函数获取读锁，
 同时也可能有

其他线程在申请写锁，
 如果这时锁处于写锁持有状态， $x < 0$ ，
 则所有申请读锁的线程会循环

等待，可见这是一种特殊的自旋锁。
 否则说明锁处于读锁状态或 0 状态，可以进入 **CAS**
 操作

尝试申请，每个想持有的线程原子地为变量 **cnt** 加
 1，成功了就相当于获取了读锁，否则说明

其他线程获取了，
这里的其他线程既有可能是其他申请读锁的线程，
也有可能是申请写锁的

线程（x 一开始为 0 时），所以 CAS
失败后需要重新判断 x 是否小于 0，然后再调用
CAS 申

请

```
static inline void
rte_rwlock_read_unlock
(rte_rwlock_t *rw1) {
    /* in debug mode, we should check
that rw1->cnt is > 0 */

    /* same than atomic32_dec */
    asm volatile(MPLOCKED
        "decl %[cnt]"
        : [cnt] "=m" (rw1->cnt)
```

```

/* output (0) */
        : "m" (rwl->cnt)
/* input (1) */
        );
/* no clobber-list */ }

```

图 5.3 释放读锁

释放读锁的操作没有使用原子原语，而是通过总线锁和指令 `decl` 实现为 `cnt` 减 1 操作，也可

以调用 `atomic_dec` 为 `cnt` 做原子减 1 操作

```

static inline void
rte_rwlock_write_lock(rte_rwlock_t
*rwl) {
    int32_t x;
    int success = 0;

    while (success == 0) {

```



```

    x = rwl->cnt;
    /* a lock is held */
    if (x != 0)
        continue;
    success = rte_atomic32_cmpset(
(volatile uint32_t *)&rwl->cnt,
        0, -1);
}
}

```

图 5.4 申请写锁

写锁的申请跟读锁申请类似，如果 x 不为 0，说明还有线程持有读锁，则循环等待，等全部持有的读锁都释放了， x 变回 0，这时通过 CAS 操作为 x 赋值 -1，成功则申请了一把写锁。

```

static inline void
rte_rwlock_write_unlock(rte_rwlock_t
*rwl) {
    /* in debug mode, we should check that

```

```
rw1->cnt is < 0 */
```

```
/* same than atomic32_inc */
asm volatile (MPLOCKED
               "incl %[cnt]"
               : [cnt] "=m" (rw1->cnt) /*
output (0) */
               : "m" (rw1->cnt)
/* input (1) */
               );
/* no clobber-list */
#endif /* _RTE_RWLOCK_H_ */
```

图 5.5 释放写锁

写锁的释放跟读锁释放相反，写锁释放是 `atomic_inc` 的操作，`inc` 加 1 后又变回 0

5.4 顺序锁

顺序锁也是对读写锁的一种优化，对于顺序锁，读者绝不会被写者阻塞，也就是说，读者可以

在写者对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写者完成写操作，

作，

写者也不需要等待所有读者完成读操作才去进行写操作。

但是，写者与写者之间仍然是

互斥的，即如果有写者在进行写操作，

其他写者必须自旋在那里，

直到写者释放了顺序锁。

这种锁有一个限制，

它必须要求被保护的共享资源不含有指针，

因为写者可能使得指针失效，

但读者如果正要访问该指针，将导致 OOPs。

如果读者在读操作期间，写者已经发生了写操作，

那么，读者必须重新读取数据，以便确保得到的数据是完整的。

这种锁对于读写同时进行的概率比较小的情况，性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。

5.5 RCU

RCU(Read-Copy Update)，

顾名思义就是读-拷贝修改，

它是基于其原理命名的。对于被 RCU

保护的共享数据结构，

读者不需要获得任何锁就可以访问它，

但写者在访问它时首先拷贝一

个副本，然后对副本进行修改，最后使用一个回调

(callback) 机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。

这个时机就是所有引用该数据的 CPU
都退出对共享 数据的操作。

RCU 也是读写锁的高性能版本，
但是它比大读者锁具有更好的扩展性和性能。

RCU 既允许 多个读者同时访问被保护的数据，
又允许多个读者和多个写者同时访问被保护的数据

(注意：
是否可以有多个写者并行访问取决于写者之间使用的同步机制)
，读者没有任何同步开销，

而写者的同步开销则取决于使用的写者间同步机制。
但 RCU 不能替代读写锁，因为如果写比 较多时，
对读者的性能提高不能弥补写者导致的损失。

六. 无锁编程

前面简单介绍了锁，这章简单介绍一下无锁编程。

有一个观念需要先了解：

虽然无锁编程对于多核编程作用有限，
但是它对于理解多线程编程的许多深层次
问题还是有很好的借鉴作用。

前半句的意思是说无锁编程的难度导致只能有极少一部分人能真正利用这种技术做出东西，

其他人只能等待这部分人的成果，

后半句的意思是即便你成为不了前面这些人，

但稍微了解 一下他们做的东西，

可以大大帮助你理解多核编程的很多概念。

6.1 定义



or interrupts,
signal handlers, etc.

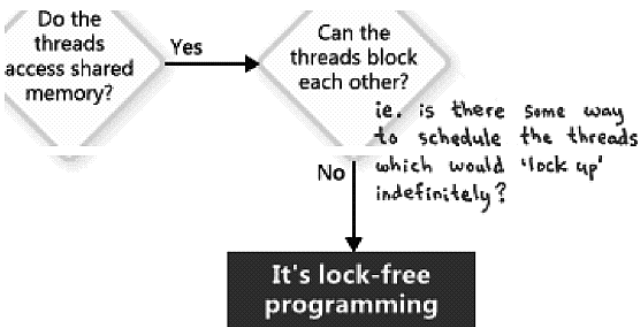


图 6.1 什么是无锁编程

基本的概念与第四章，并发级别是一致的，

就是多线程程序里，不需要依赖于锁这样的阻塞机制就可以保证同步的编程技术。

It's lock-free programming

```
graph TD; A[It's lock-free programming] --> B[Are there multiple writers?]; A --> C[Are you targeting multicore?]; B --> D[Things are easier to implement, but probably slower]; C --> D;
```

Are there
multiple
writers?

Are you
targeting
multicore?
or

Things are easier
to implement, but
probably slower

Yes

eg. Java volatile
or C++11 default
atomic types

Yes

Is there
sequential
consistency?

any

symmetric

multiprocessor

Yes

You may need
atomic RMW
instructions

read -
modify -
write

Is there a
CAS

loop?

Compare -
and - swap

Yes

You may need
a full memory
fence somewhere

No

Are there
producers and

consumers
only?

Yes

No

eg. C++11 atomic
types with low-level
constraints, or
plain C/C++

You may need
barriers or
access semantics

to enforce
memory ordering

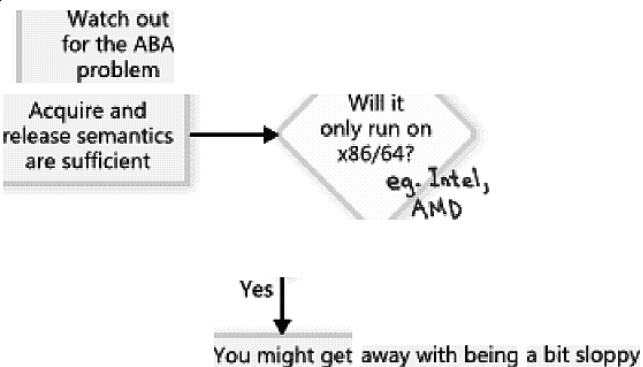


图 6.2 无锁编程涉及的技术

从上图可以看到，如果要自己编写无锁程序，基本上要掌握前面几章涉及的所有概念。是一个非常艰巨的工程。

七. 并发数据结构、

开源库

7.1 一些开源的并发库

<http://openmp.org/wp/> openmp

开放标准的并行程序指导性注释，没用过

<http://software.intel.com/zh-cn/articles/parallelization-using-intel-threading-building-blocks-1>

[intel-tbb](#) intel tbb ， Intel 的 C++多线程库，

没有用过，有人说性能很不好

<https://code.google.com/p/nbds/>

无锁数据结构库，关键是针对缓存做了优化，

需要注意的 是里边的哈希表是没有删除的，

说多了都是泪

<http://mcg.cs.tau.ac.il/projects/hopscotch-hashing-1>

并发的 hopscotch 哈希表实现

da-data.blogspot.nl/2013/03/optimistic-cukoo-hashing-for.html 并发的

cukoo 实现，查找非 常快，

但增加和删除目前是大锁，

作者说以后会推出无锁的增加和删除，

不知道什么时候出 来

<http://ww2.cs.mu.oz.au/~astivala/paralleldp/>

实现了无锁的基于数组的哈希表，无锁的基于链

表的哈希表，对比了多种实现的性能，

无锁实现仍然是没有删除，另外，

无锁实现没有针对 缓存做优化

[http://dpdk.org/ rte_ring.c](http://dpdk.org/rte_ring.c)

实现了无锁的生产者消费者队列

另外，据说 apache portable runtime 里边有很多

lock-free 算法，Windows 里的 Interlocked 系

列函数内部实现也用到了很多 lock-free 算法，

没有验证过。

上述代码里，详细调试过的有 nbds 的无锁 hashtable，用它构造了一个哈希表作为流表，与基于 DPDK example 的一种哈希表+dpdk rwlock 的实现做 L3 转发性能对比，发现不删流的情况下性能差不多，当然测试是在冲突率不是很高的情况下测试的。

7.2

一次无锁哈希表跟基于锁的哈希表性能对比测试

7.2.1 测试平台

测试在一台 intel E5-2658 上进行，这款 CPU 有两个物理 CPU，每个物理 CPU 有 8 个核，每个核有两个核线程。测试只使用到一颗物理 CPU，只用到 core1~core8

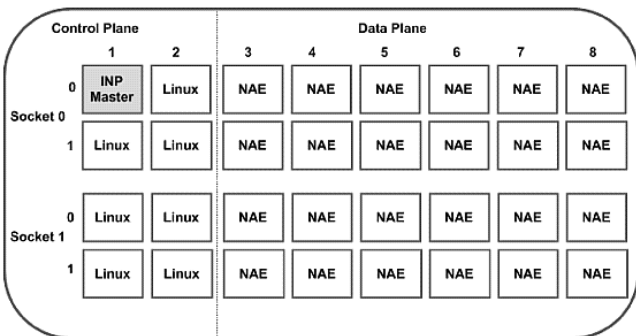


图 7.1 Intel e5-2658

```
root@ubuntu:~# coremap
CPU : Intel(R) Xeon(R) CPU E5-2658 0 @ 2.10GHz
2 sockets, 8 cores per socket and 2 threads per core
0/0/0 0
0/1/0 1
0/2/0 2
0/3/0 3
0/4/0 4
0/5/0 5
0/6/0 6
0/7/0 7
1/0/0 8
1/1/0 9
1/2/0 10
1/3/0 11
1/4/0 12
1/5/0 13
1/6/0 14
1/7/0 15
0/0/1 16
0/1/1 17
0/2/1 18
0/3/1 19
0/4/1 20
0/5/1 21
0/6/1 22
0/7/1 23
1/0/1 24
1/1/1 25
1/2/1 26
1/3/1 27
1/4/1 28
1/5/1 29
1/6/1 30
1/7/1 31
```

图 7.2 E5-2658 核分布

发包和统计工具是 testcenter 发包仪器，

使用两块万兆网卡对发 udp 包。

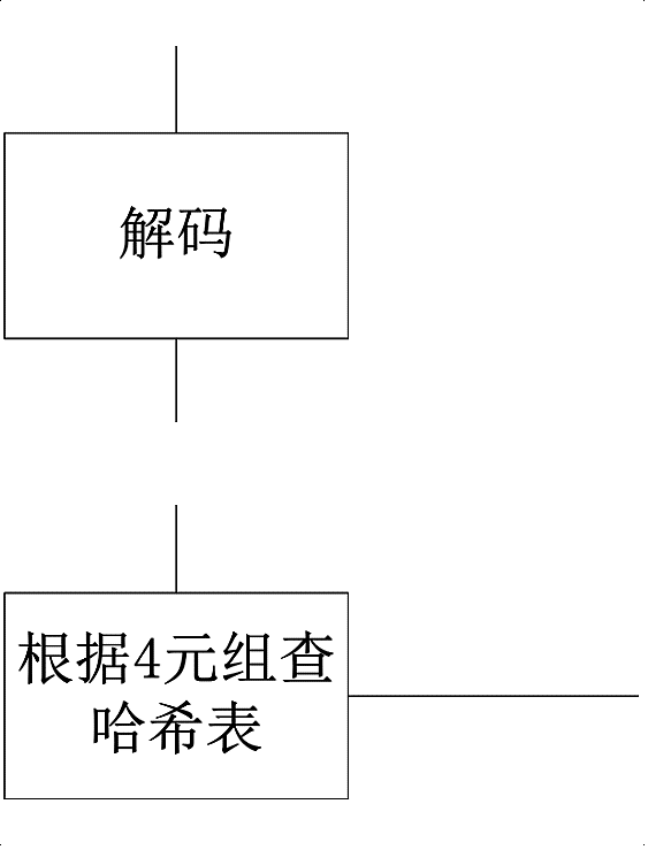
7.2.2 测试过程

1) DPDK 示例程序 examples 里有 l2fwd 示例程序，测试程序在 l2fwd 的基础上改写。

如图，l2fwd 只有收包和发包两个步骤，测试程序增加了解码和查表（增加节点、拷贝 mac）的操作。



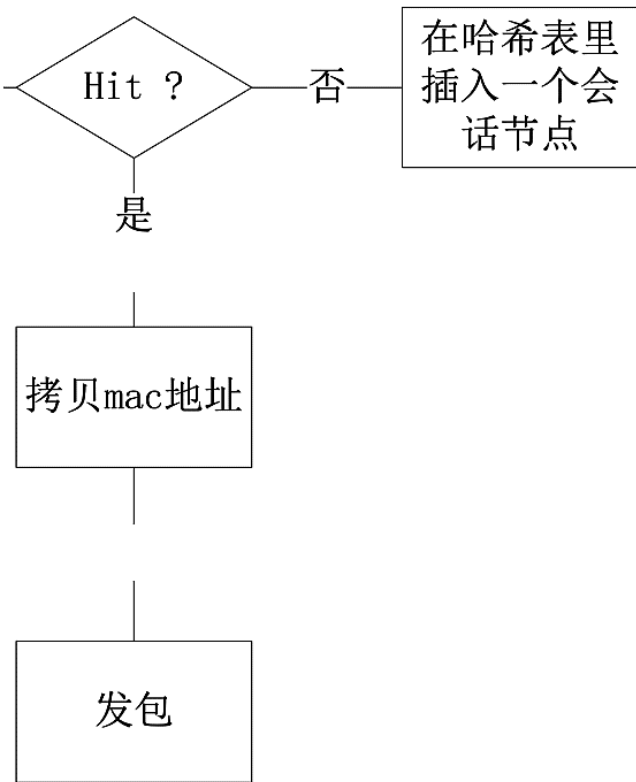
收包



```
graph TD; A[解码] --> B[根据4元组查哈希表];
```

解码

根据4元组查
哈希表





3) 测试时, 固定 2 个 core

线程负责两个网卡的收包和发包,
中间的解码+会话表处理灵活配

置核心, 测试对比了 2 个, 4 个, 6
个核线程的情况

4) 哈希表部分 API 使用函数指针做了一个抽象层
(lookup, insert, delete), 具体实现替换为
不同的哈希表实现

7.2.3 哈希算法

1) 第一种哈希使用

DPDK librte_hash/rte_fbk_hash.c 增加 rte_rwlock

实现，在原有哈希算

法基础上，每个 bucket 增加一个 rwlock，lookup 的时候需要获取读锁，返回前释放。Insert 需要先获取写锁，然后 lookup 一遍没有相同值再插入，返回前释放写锁，delete 也需要获取写锁。

2) 第二种哈希使用 nbds 的 hashtable.c 实现，在原有哈希算法基础上，将 key,value 的真实值替换为 DPDK 的 mempool 实现，将所有内存的申请替换为 DPDK 的 rte_malloc 实现。

7.2.4 测试结果

3) 简单测试发现，随着中间工作核线程数目增加，性能有明显提升

4) 多次反复测试发现，
两种哈希实现性能相差不大。

5) Nbds 没有实现真正的 delete 算法，
研究中发现，

大部分开源的无锁结构都没有实现删除

算法，删除算法会导致“状态爆炸”

即中间可能的状态太多导致正确性很难证明。

结论：

a，对于哈希表这种操作相对复杂的结构，
加上要考虑多核平台的内存模型，无锁的高
效稳定实现非常难，不要自己实现这样的结构。

b.

基于自旋锁/读写锁实现的哈希表性能可以接受

代码请邮件 chengjia4574@gmail.com

理论上任何 X86 平台都可以执行

八. 多核工程实践

8.1 网络设备 : Intel DPDK

Intel 推出的基于 x86 平台的数据包处理方案，
目前已经开源：<http://dpdk.org/>，网上最新版本是
1.3.1， 主要内容包括： 多核框架，
基于大内存页的内存管理方案， 无锁队列，
用户态网 卡驱动，
这几个核心模块的底层实现用到了本文档前面提到的诸多概念，
在前面概念的描述 过程中， 也引用了一些 dpdk

的代码。

8.2 网络游戏

虽然文章一开头已经提到，目前家用 PC 和手机核数最高已经超过 10 颗，但很多网络游戏的主引擎还是单进程的，比较确定的使用了多核系统的，有云风主导开发的一款游戏（据说采用多进程模型），另外，剑侠奇缘现在有选项：多核游戏模式，不知道是不是已经实现多核的游戏引擎。

8.3 手机开发

ios4.0 : Grand Central Dispatch (GCD)

九. 参考

<http://ifeve.com/> 并发编程网

<http://www.pdl.cmu.edu/>

卡耐基梅陇大学并行实验室

<http://www.parallellabs.com> 并行实验室

http://en.wikipedia.org/wiki/Non-blocking_algorithm#Obstruction-freedom

<http://www.cnblogs.com/lxconan/category/429872.html> sql

server 并行相关

<http://preshing.com/20120930/weak-vs-strong-memory-models>

内存模型

<http://preshing.com/20120625/memory-ordering-at-compile-time>

内存顺序一致性模型

<http://preshing.com/20120515/memory-reordering-caught-in-the-act>

内存顺序一致性模型

<http://preshing.com/20120710/memory-barriers-are-like-source-control-operations>

内存屏 障

<http://preshing.com/20111118/locks-arent-slow-lock-contention-is>

锁

附录A