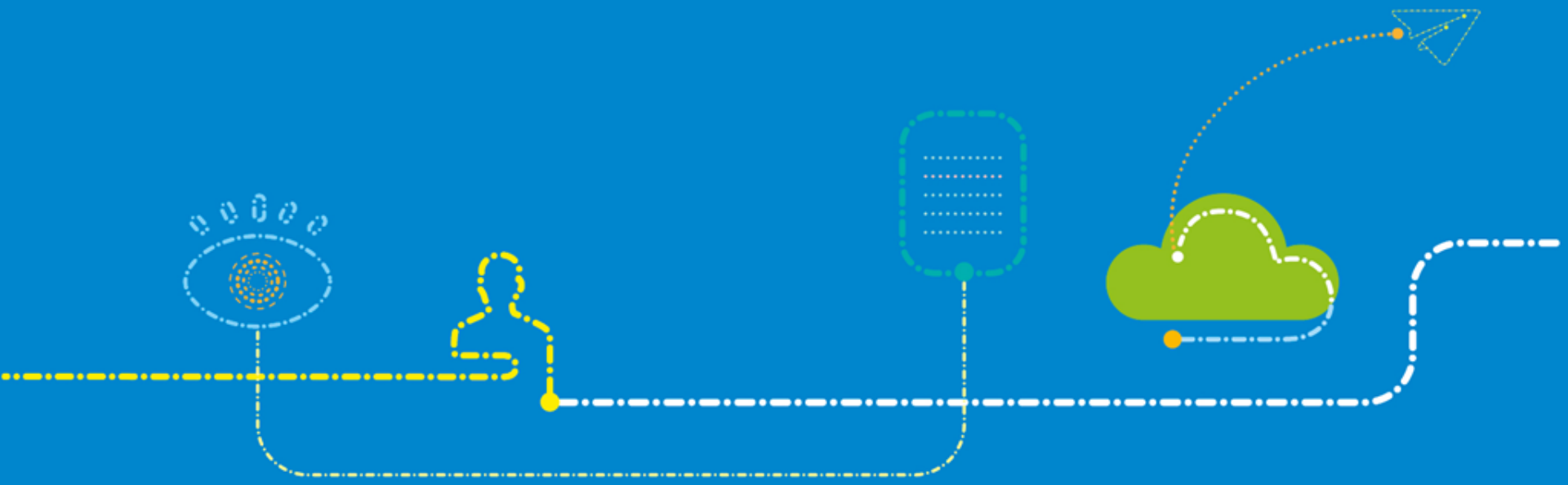
Linux内存屏障

——并行编程的基石

ZTE

未来，不等待

谢宝友　中兴通讯操作系统团队
scxby@163.com



目录

从硬件说起——霍金提出的难题

According to Stephen Hawking

✓the finite speed of light

✓and the atomic nature of matter

How about

✓量子理论？

✓弦论？

从硬件说起——硬件制造商的努力

3D集成

降低光程、减少能耗，但是制造、测试和散热（用钻石）？？

新材料和新工艺

单个电子上存储多个比特位，不稳定

用光代替电子

光速也有极限

专用加速器

GPU、矢量处理器、专用加密硬件，比较靠谱

摩尔定律失效了☹

计数的难题——代码例子

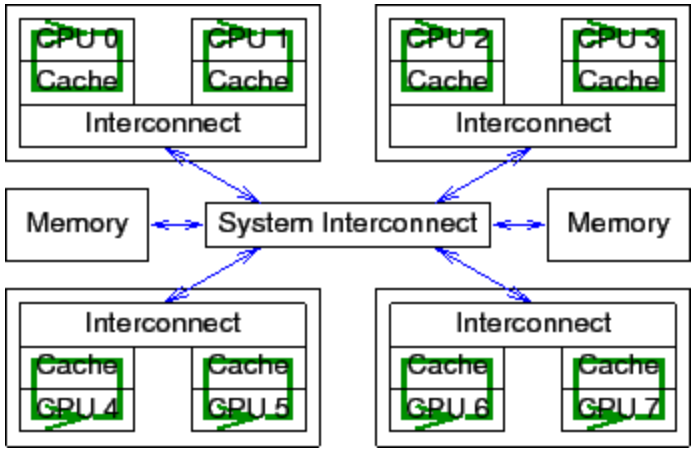
```
long  counter  =  0;

void  inc_count(void)
{
   counter++;
}

long  read_count(void)
{
   return  counter;
}
```

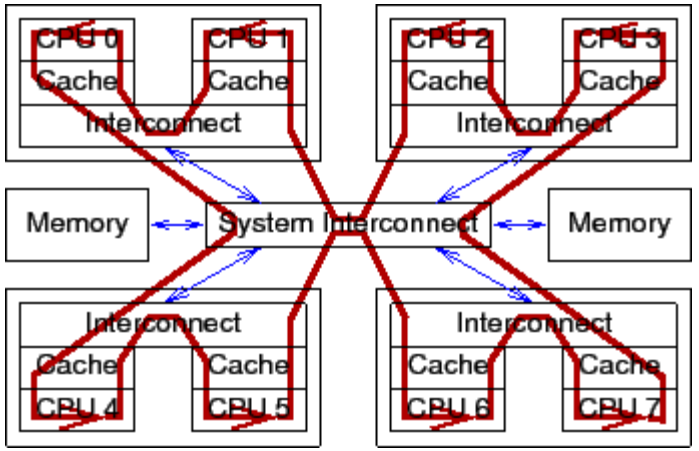
丢失计数

```
atomic_t counter = ATOMIC_INIT(0);

void inc_count(void)
{
   atomic_inc(&counter);
}

long read_count(void)
{
    return atomic_read(&counter);
}
```

正确，但是性能？

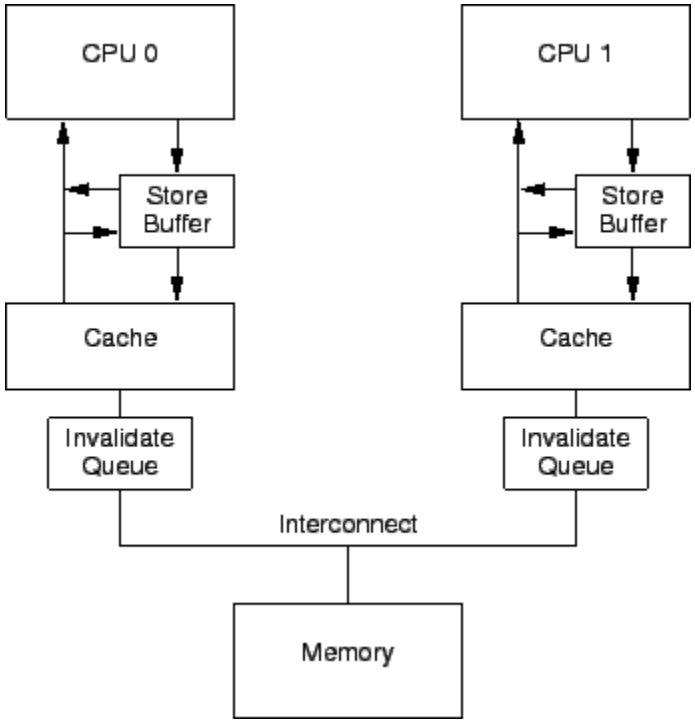
计数的难题——硬件带来的困扰

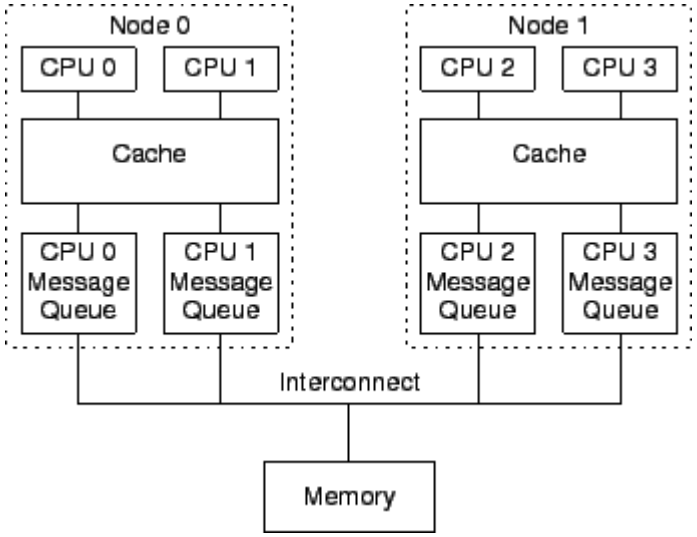




缓存一致性协议及内存屏障

内存与CPU的速度差异超过1个数量级





MESI一致性协议消息是异步的

缓存一致性协议及内存屏障

内存乱序是如何形成的？

- 编译造成的乱序
- CPU执行乱序
- 内存乱序是什么？
- 根本原因还是光速的限制
- 存储缓冲和使无效队列是乱序产生的原因
- 单CPU可以按照编程顺序看到内存序（不考虑编译乱序及极个别情况）
- 多CPU之间无法按照编程顺序看到内存序

缓存一致性协议及内存屏障

```
CPU0：
void foo(void)
{
  a = 1;
  b = 1;
}

CPU1：
void bar(void)
{
  while (b == 0) continue;
  assert(a == 1);
}
```

1. CPU 0 执行a = 1。缓存行不在CPU0的缓存中，因此CPU0将“a”的新值放到存储缓冲区，并发送一个“读使无效”消息。
2. CPU 1 执行 while (b == 0) continue，但是包含“b”的缓存行不在缓存中，它发送一个“读”消息。
3. CPU 0 执行 b = 1，它已经在缓存行中有“b”的值了 (换句话说，缓存行已经处于“modified”或者“exclusive”状态)，因此它存储新的“b”值在它的缓存行中。
4. CPU 0 接收到“读”消息，并且发送缓存行中的最新的“b”的值到CPU1，同时将缓存行设置为“shared”状态。
5. CPU 1 接收到包含“b”值的缓存行，并将其值写到它的缓存行中。
6. CPU 1 现在结束执行while (b == 0) continue，因为它发现“b”的值是1，它开始处理下一条语句。
7. CPU 1 执行assert(a == 1)，并且，由于 CPU 1 工作在旧的“a”的值，因此验证失败。
8. CPU 1 接收到“读使无效”消息, 并且发送包含“a”的缓存行到CPU0，同时使它的缓存行变成无效。但是已经太迟了。
9. CPU 0 接收到包含“a”的缓存行，将且将存储缓冲区的数据保存到缓存行中，这使得CPU1验证失败。

缓存一致性协议及内存屏障

```
CPU0：
void foo(void)
{
  a = 1;
  smp_mb();
  b = 1;
}

CPU1：
void bar(void)
{
  while (b == 0) continue;
  assert(a == 1);
}
```

1. CPU 0 执行 a = 1。CPU0中相应的缓存行是只读的，因此CPU0将“a”的新值放入存储缓冲区，并发送一个“使无效”消息，这是为了使CPU1的缓存中相应的缓存行失效。
2. CPU 1 执行 while (b == 0) continue，但是包含“b”的缓存行不在缓存中，因此它发送一个“读”消息。
3. CPU 1 接收到 CPU 0的“读”消息，将它排队，并立即响应消息。
4. CPU 0 接收到来自于 CPU 1的响应消息，因此它放心的通过 smp_mb()，从存储缓冲区移动“a”的值到缓存行。
5. CPU 0 执行 b = 1。它拥有这个缓存行(也就是说，缓存行已经处于 “modified”或者“exclusive”状态)，因此它将“b”的新值存储到缓存行中。
6. CPU 0 接收到“读”消息，并且发送包含“b”的新值的缓存行到CPU 1，也标记缓存行为“shared”状态。
7. CPU 1 接收到包含“b”的缓存行并且将其应用到本地缓存。
8. CPU 1 现在执行完 while (b == 0) continue，因为它发现“b”的值为1，接着处理下一条语句。
9. CPU 1 执行assert(a == 1)，由于旧的“a”值还在CPU 1的缓存中，因此陷入错误。
10. 虽然陷入错误，CPU 1 处理已经排队的“使无效”消息，并且刷新包含“a”值的缓冲行。

缓存一致性协议及内存屏障

```
CPU0：
void foo(void)
{
  a = 1;
  smp_mb();
  b = 1;
}

CPU1：
void bar(void)
{
  while (b == 0) continue;
  smp_mb();
  assert(a == 1);
}
```

1. CPU 0 执行 a = 1。相应的缓存行在CPU0的缓存中是只读的，因此CPU0将“a”的新值放入它的存储缓冲区，并且发送一个“使无效”消息以刷新CPU1的缓存。
2. CPU 1 执行 while (b == 0) continue，但是包含“b”的缓存行不在它的缓存中，因此它发送一个“读”消息。
3. CPU 1 接收到 CPU 0的“使无效”消息，将它排队，并立即响应它。
4. CPU 0 接收到CPU1的响应，因此它放心的通过smp_mb()语句，将“a”从它的存储缓冲区移到缓存行。
5. CPU 0 执行 b = 1。它已经拥有缓存行(换句话说, 缓存行处于“modified” 或者“exclusive”状态)，因此它存储“b”的新值到缓存行。
6. CPU 0 接收“读”消息，并且发送包含新的“b”值的缓存行给CPU1，同时标记缓存行为“shared”状态。
7. CPU 1 接收到包含“b”的缓存行并更新到它的缓存中。
8. CPU 1 现在结束执行 while (b == 0) continue，因为它发现“b”的值为 1，它处理下一条语句，这是一条内存屏障指令。
9. CPU 1 必须延迟，直到它处理使无效队列中的所有消息。
10. CPU 1 处理已经入队的“使无效”消息，从它的缓存中使无效包含“a”的缓存行。
11. CPU 1 执行assert(a == 1)，由于包含“a”的缓存行已经不在它的缓存中，它发送一个“读”消息。
12. CPU 0 响应“读”消息，发送它的包含新的“a”值的缓存行。
13. CPU 1 接收到缓存行，它包含新的“a”的值1，因此不会陷入失败。

缓存一致性协议及内存屏障——正确的做法

```
CPU0：
void foo(void)
{
  a = 1;
  smp_wmb();
  b = 1;
}

CPU1：
void bar(void)
{
  while (b == 0) continue;
  smp_rmb();
  assert(a == 1);
}
```

1. foo()没有必要做使无效队列相关的任何操作，类似的，bar()也没有必要做与存储缓冲区相关的任何操作。
2. 不准确的说，一个“读内存屏障”仅仅标记它的使无效队列，一个“写内存屏障”仅仅标记它的存储缓冲区，完整的内存屏障同时标记无效队列及存储缓存缓冲区。
3. 读内存屏障仅仅保证装载顺序，因此所有在读内存屏障之前的装载将在所有之后的装载前完成。类似的，写内存屏障仅仅保证写之间的顺序。所有在内存屏障之前的存储操作将在其后的存储操作完成之前完成。完整的内存屏障同时保证写和读之间的顺序。

缓存一致性协议及内存屏障——锁的实现

锁的三个属性

✓一个特定CPU 或者线程必须以编程顺序看到自己的所有加载和存储操作。

✓申请、释放锁必须以全局顺序被看到。全局时间线的概念。

✓如果一个特定变量在临界区中还没有被存储，那么在临界区中执行的加载操作必须看到上一次存储。

缓存一致性协议及内存屏障——锁的实现

违反第二个属性的例子

```
spin_lock(&mylock);
if (p == NULL)
  p = kmalloc(sizeof(*p), GFP_KERNEL);
spin_unlock(&mylock);
```

结果：内存泄漏

缓存一致性协议及内存屏障——锁的实现

违反第三个属性的例子

```
spin_lock(&mylock);

ctr = ctr + 1;

spin_unlock(&mylock);
```

结果：计数倒退

缓存一致性协议及内存屏障——锁的实现

```
void spin_lock(spinlock_t *lck)
{
  while (atomic_xchg(&lck->a, 1) != 0)          //包含smp_mb()实现第二个属性
    while (atomic_read(&lck->a) != 0)
      continue;
}

void spin_unlock(spinlock_t lck)
{
  smp_mb();                                     //实现第三个属性
  atomic_set(&lck->a, 0);
}
```

问题：

内核在实现锁的时候，如何解决编译乱序和指令乱序？？

感谢

感谢主办单位及在座各位☺

感谢Paul E. McKenney的著作：

《Is Parallel Programming Hard, And, If So, What Can You Do About It?》

https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

中文翻译文档：

http://download.csdn.net/detail/xiebaoyou/9083233

题外话：嵌入式OS实时性测试

Thank you

未来，不等待