# 多核环境下锁优化方法分析

2012.6 Version 1.01
yalung929@gmail.com

## 1 前言

锁是并行的敌人。

有效的减少甚至避免锁可以提高多线程程序在多核环境下的性能表现。

我们谈到多核编程就会提到免锁编程（LOCK-FREE），但绝大多数的免锁算法，实现和维护都非常困难，很难应用于工程实践。 完全的免锁[1]，不符合实际；能在一定程度上减少锁的开销、粒度、争抢就已经非常不错了。本文对一些**可以应用到工程实践中的锁优化思路和方法**做出了分析和总结。

——简单是美，合适是道

## 2 范围

由于知识和精力有限，本文的所有内容限定在【X86_64 SMP Linux GCC】这一平台环境。

## 3 测试环境

CPU：Intel(R) Xeon(R)  5138  @ 2.13GHz SMP 2 CPU * 2 CORE

OS：Linux 3.1.10-1.9-default #1 SMP x86_64

编译器和运行库：gcc version 4.6.2 (SUSE Linux) + glibc-2.14.1(with pthread)

## 4 测试方法和代码

### 4.1 锁的对比测试方法

4 个线程争抢同一个锁，进入临界区后做一些计算，并统计和计时。

[1] 实际上Erlang等多核并行语言可以做到完全免锁，但那是另外一个层面，是从软件架构上避免了，C/C++程序如果借鉴erlang的设计思想，数据完全不共享，仅靠收发消息通信，也是完全免锁的。但这个主题不在本文讨论范围之内。Erlang通过语言的特性限定了程序结构，Erlang是一种设计思想。

### 4.2 相关代码

本文涉及的相关代码详见附录 7.2 simple 库，包含了：

1. Spinlock - 带休眠自旋锁的实现。

2. RWSpinlock - 带休眠读写自旋锁的实现。

3. SPSC - 单生产者单消费者的免锁实现。

4. Singleton – 单例模式免锁实现。

5. test - 测试程序。

## 5 优化方法

### 5.1 使用自旋锁

普通的锁，比如 pthread 的 mutex 基于 OS 提供的休眠和唤醒机制，如果锁的临界区很短，会造成 CPU 资源的浪费，并且延迟也比较高。这种情况我们可以考虑使用自旋锁——等待自旋锁的线程不会放弃 CPU 进入休眠，而是不断的检查锁占用标志，一旦持有锁的线程设置标志释放锁，等待线程可以立即进入临界区。

在测试环境上对 pthread 线程库的 mutex 和 spinlock 的做性能对比，结果如下：

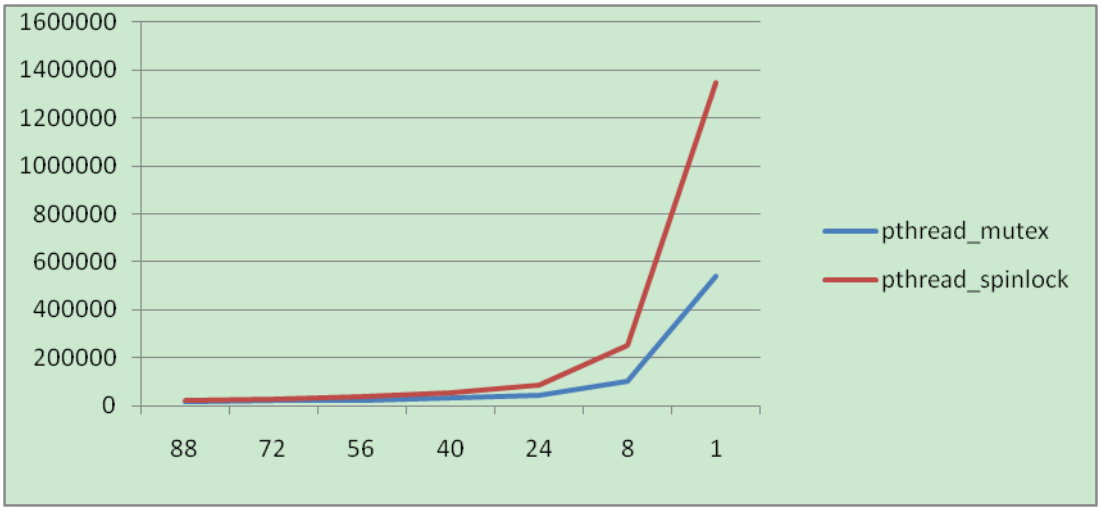


图 1 pthread mutex 和 spinlock 性能对比

随着临界区越来越短，自旋锁相对 MUTEX 的性能优势越来越明显，当临界区缩短到 8us[2]以

[2]这是一个大约值，因环境而已，只用于说明锁性能变化的趋势。在测试环境上，大概是做了1000次内存（8字节对齐）访问的时间。

后，自旋锁可以达到 2 倍以上的提升。

如果临界区持续时间很长，spinlock 虽然不会比 mutex 性能差，但 CPU 占用率会很高，一直是 400%（4 个核，4*100%）；而 mutex 随着临界区时间变长，逐渐接近于 100%。主要是因为 spinlock 等锁时不放弃 CPU，空转。所以 spinlock 应该在临界区比较短的情况下使用，否则很浪费 CPU。

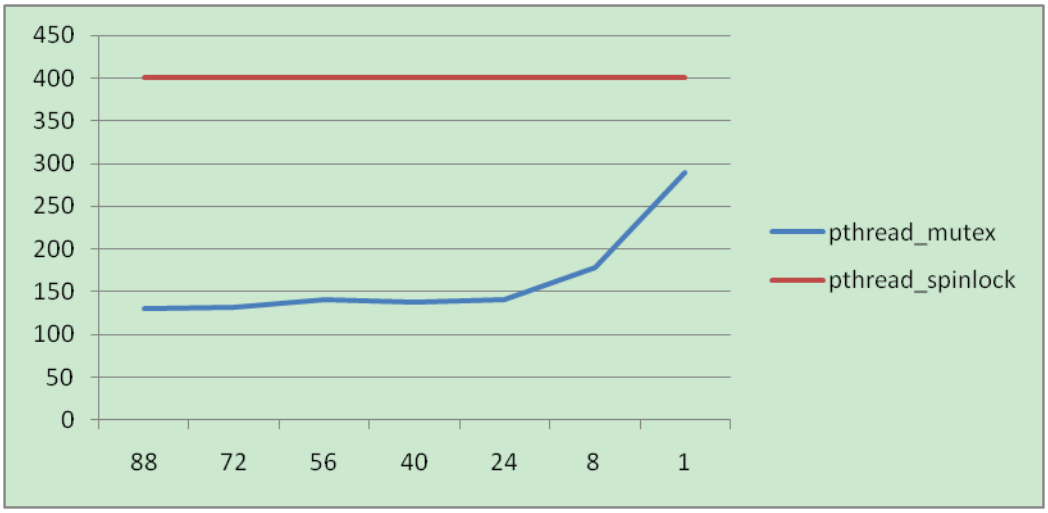


图 2 pthread mutex 和 spinlock CPU 占用对比

然而由于应用程序处于用户态，它的执行可被任意抢占，临界区的执行时间不能准确预期，有可能被耽搁“很久”。所以应用程序中使用自旋锁的实践并不多。

Facebook 于今年六月份放出了他们针对多核环境的 C++高性能库- folly，里面对自旋锁实现做了改进，更适用于用户态程序。这种改进非常简单，就是**先自旋后休眠**：

这个简单的改进，让自旋锁具有了很好的适应性，可以在**应用程序中使用**，多数情况下我可以很快，少数极端情况下临界区的执行被耽搁了，我也不至于傻转，浪费 CPU。

simple::SpinLock 是采用这种方式实现的一个 C++自旋锁。和 mutex 的性能、CPU 占用对比情况如下：

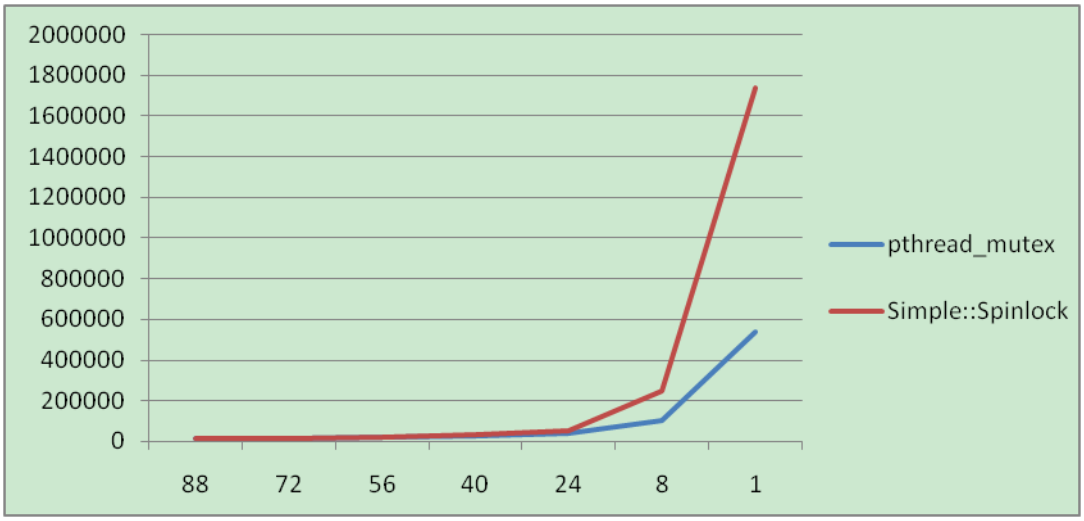


图 3 simple::spinlock 和 pthread mutex 性能对比

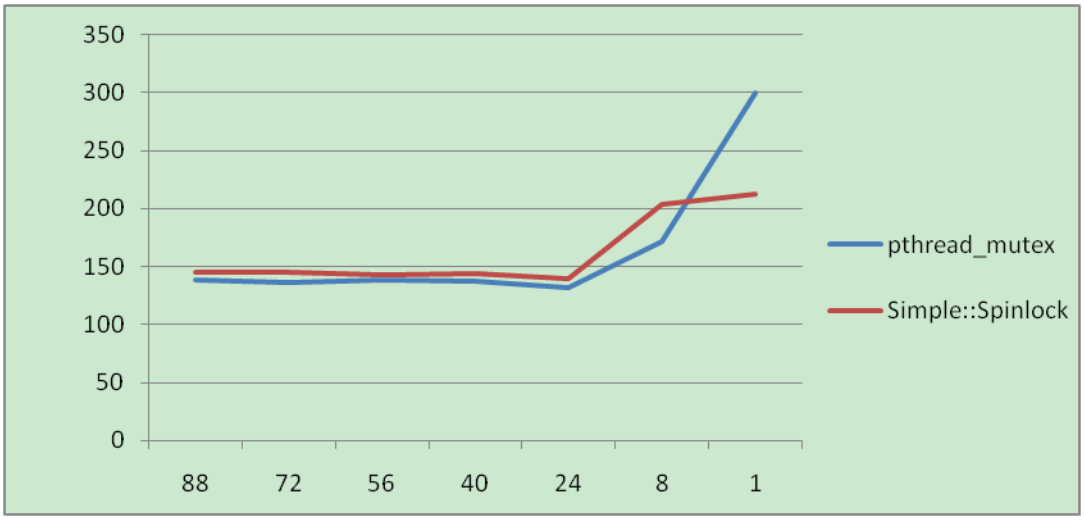


图 4 simple::spinlock 和 pthread mutex CPU 占用对比

simple::SpinLock 在临界区时间很短的时候相对 pthread_mutex 的性能提升明显，和 pthread_spinlock 的表现一致。但 CPU 占用只有 pthread_spinlock 的一半左右。

simple::SpinLock 在临界区时间比较长的时候相对 pthread_mutex 的性能没有提升，和 pthread_spinlock 的表现一致。但 CPU 占用确很低，接近于 pthread_mutex。

综上所述，多核环境下对于**锁争抢比较频繁**、并且**临界区很短**的场景，可以使用 simple::SpinLock 这种带休眠的自旋锁代替 pthread_mutex 提升性能。

## 5.2 使用读写锁

4 个线程争抢一个锁，实际上同一时刻只有 1 个线程可以干活儿，无论我们怎么去优化锁的开销，只要锁存在，临界区的并行度仍然是 1，白白浪费了 3 个 CPU。如果 CPU 个数更多，

参与争抢的线程数更多，这种浪费更明显，达到一种不可接受的程度。

从结构上对锁优化，是进一步的思路。多数场景，对临界区同时读是没有问题的，也就是说只要没有人写临界区，读是没必要等待的，可以做到完全的并行。pthread_rwlock 就是这样一个锁。

随着写占用的比例越来越少，pthread_rwlock 相对 ptread_mutex 提升越来越明显。最后可以达到 3 倍左右的提升。理论上如果写是 0%，读完全并发不等待，应该是 4 倍（4 个线程），这里锁自身开销占了一部分。如果 CPU 个数更多，并发的线程数更多，读写锁带来的性能提升会更多。

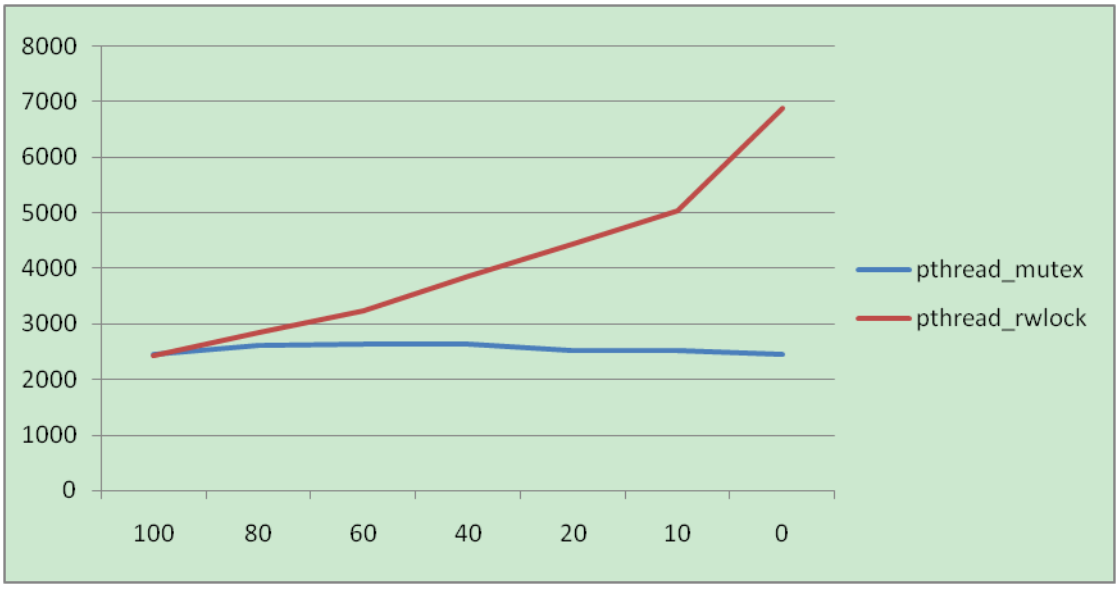


图 5 pthread_mutex 和 pthread_rwlock 性能对比

如果把临界区缩短至 8us[3]，使用自旋读写锁，100%写的情况下和自旋锁的表现一致，都是相对 mutex 提升 2 倍左右，随着写的比率越来越低，低于 10%时，达到 4 倍左右，最终可以达到 6 倍左右。

[3] 同1。

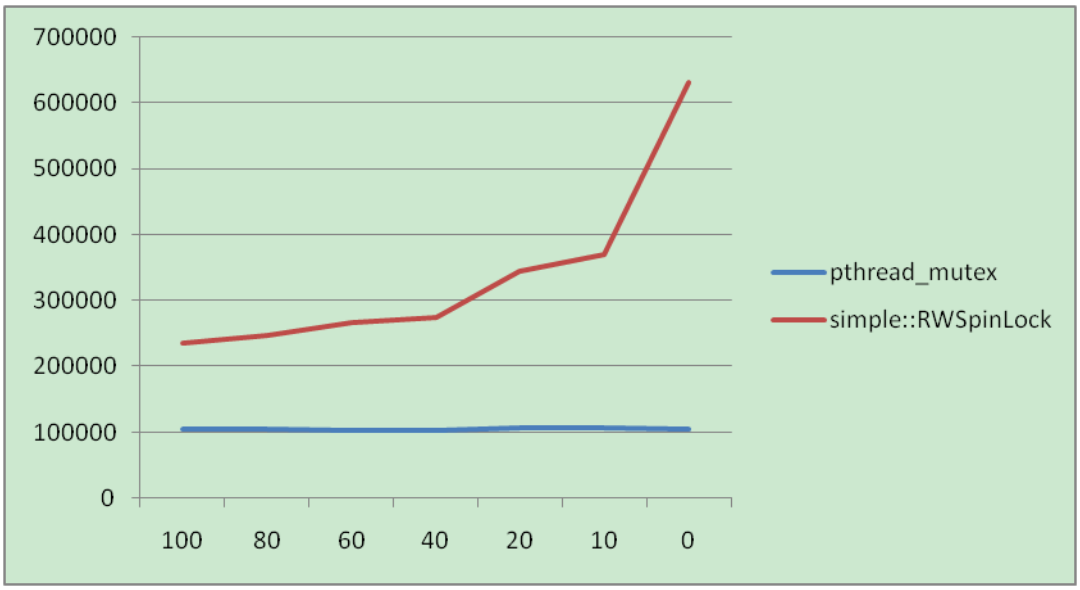


图 6 pthread_mutex 和 simple::RWSpinLock 性能对比

所以，多核环境下对于**锁争抢比较频繁**、并且**写明显少于读**的场景，使用 pthread_rwlock 代替 pthread_mutex 可以获得明显的性能提升。

而同时如果**临界区也很短**的话，使用读写自旋锁代替 pthread_mutex 会有更多的性能提升。

## 5.3 使用 Double Check

考虑如下单例模式，实例初始化完毕以后，后面每次访问都做了一次根本不需要的锁操作。

```
   static Tp* Instance() {
                locker.lock();
                if (inst_ == NULL) {
                      inst_    = new Tp;
                }
                locker.unlock();
           return    inst_;
 }
```

其实上面单例模式可以改为：

```
static Tp* Instance() {
                if (inst_ == NULL) {
                      locker.lock();
```

```
            if (inst_ == NULL) {
                Tp* tmp = new Tp;  [4]
                mb();
                inst_ = tmp;
            }
            locker.unlock();
        }
    return    inst_;
}
```

这样一旦初始化之后，再也不需要获取锁了。

这种 check-lock-check-do 的方式称为 Double Check。Double Check 是一种锁的粒度优化。不只可以用于单例模式，其应对的基本模型是：

```
if    (condition = true)
      do mutual work    //change condition
```

符合这种模型的场景下都可以使用 Double Check（check-lock-check-do），避免不必要的锁开销，尤其是在 if 条件多数情况不达成的场景下。

Double Check 是一种粒度优化，主要是从代码自身的逻辑上去考虑。

## 5.4 免锁

### 5.4.1 原子操作

如果对共享数据的操作都是原子的那就不需要锁了。最简单的在 x86_64 上，如果多个线程同时访问一个**对齐**的 long[5]型是不需要锁的。

```
volatile unsigned long value_;
value_ = 18676783867
```

上面的赋值语句是原子的，所有的线程要么看到旧值，要么看到新值。即使多个线程同时赋值也没关系。

---

[4] 如果直接用 inst_ = new Tp; 对 inst_的赋值语句可能先于 Tp 的构造函数完成。所以插入了一条屏障指令，确保 inst_非空以后，其指向的内容已经是完全初始化的。

[5] 低于8字节的对齐的内存访问自然也可以。

value_++;

但这条自增语句不是原子的，它一般需要读内存、改值、写内存三个指令。

```
400585:    48 8b 45 f8          mov    -0x8(%rbp),%rax
400589:    48 83 c0 01          add    $0x1,%rax
40058d:    48 89 45 f8          mov    %rax,-0x8(%rbp)
```

一般的编译器都会针对这种操作，提供内建的原子方法，比如在 GCC 下，上面的 value++ 可以修改为：

__sync_fetch_and_add(&value_, 1);

__sync_fetch_and_add 是原子的，它的实现如下：

```
40057c:    48 8d 45 f8          lea -0x8(%rbp),%rax
400580:    f0 48 83 00 01       lock addq $0x1,(%rax)
```

直接对内存位置执行 add 指令，原来的 3 条变为 1 条，但这仍然不足以保证它的原子，因为别的线程可以可能正在操作这个内存位置，所以加了一个 lock 前缀，对于 x86 来说，lock 前缀用于锁定总线，保证其后面一条指令对内存的独占访问。

gcc 提供了一组原子方法，包含了：

type __sync_fetch_and_add (type *ptr, type value)

type __sync_fetch_and_sub (type *ptr, type value)

type __sync_fetch_and_or (type *ptr, type value)

type __sync_fetch_and_and (type *ptr, type value)

type __sync_fetch_and_xor (type *ptr, type value)

type __sync_fetch_and_nand (type *ptr, type value)

bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval)

…… 更多的原子方法可以参考 GCC 手册[6]。

这些方法无需声明可直接使用。

这些原子方法的本质是用硬件层面的锁代替了软件层面的锁，从软件层面来看是原子化了。

在 x86_64 上，如果**共享的数据长度小于等于 8 字节**，可以考虑使用原子方法，替代锁。

**使用原子操作时，可能会涉及到内存屏障的使用，**附录 7.1 **对内存屏障做了详细的说明。**

---

[6] GCC online doc http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html

### 5.4.2 CAS

CAS 指 compare and swap，整个过程是原子的，CAS 原语的逻辑是：

```
bool CAS(Type* addr, const Type& compare, const Type& value)
{
    if (*addr == compare)
    {
        *addr = value;
        return true;
    }
    return false;
}
```

CAS 依赖硬件实现，在 X86 上使用的是 cmpxchg 指令，cmpxchg 支持最多 64bit 的比较交换操作。上节提到的 GCC 内建函数__sync_bool_compare_and_swap 就是对 cmpxchg 的封装。依靠 CAS 原语，我们可以实现完全免锁的单例模式：

```
  static Tp* Instance() {
            Tp* newTp;
            if (inst_ == NULL) {
                newTp = new Tp;
                if (!__sync_bool_compare_and_swap(&inst_, NULL, newTp)) {
                    delete newTp;
                }
            }
        return    inst_;
    }
```

CAS 是很多免锁算法的核心。

### 5.4.3 SPSC

SPSC – single producer single consumer 单生产者单消费者模式，本质是一个 FIFO 的环形队列，一个读者从队列中取数据，一个写者往队列中添加数据，读和写都不需要加锁。SPSC

的实现和维护都不难，是可以用于工程实践的一个免锁数据结构之一。6.3 小节中对 simple::SPSC 的实现做了详细的介绍。

# 6 实现

## 6.1 自旋锁

simple::SpinLock 采用了 folly:: MicroSpinLock 的实现方式，先自旋后休眠。

simple::SpinLock 的核心原理是使用 CAS 原语，改变一个 1 字节的内容实现加锁。

```
bool TryLock() {
    return   __sync_bool_compare_and_swap(&lock_, FREE, LOCKED)
}
```

解锁时，进行一个内存写操作。由于单字节必定是原子的，不需要 CAS。

```
void UnLock() {
        asm volatile("" : : : "memory");
        lock_ = FREE;
 }
```

但需要插入内存屏障确保其他线程看到锁 FREE 以后，拿锁的线程已经离开了临界区（所有对共享数据的读写已经完成）。目前 x86 上并不需要硬件屏障[7]，但如果担心特殊环境或未来变化，换成硬件屏障肯定是万无一失的；目前 Glibc 和 folly 以及内核的自旋锁都没有针对 x86 用硬件屏障。

获取锁时，反复尝试直到加锁成功。

```
void Lock()  {
        Waiter w;
        do {
            while(lock_ != FREE) {   //先 check 一次
                w.Sleep();
```

[7] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A-System Programming Guide. 8.2 MEMORY ORDERING
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

```
        }
    } while(!TryLock());          //再次执行 check 原子化（加锁）
}                                 //隐含了 double check 思想，避免反复执行原子操作
                                  //原子操作比普通操作开销大
```

一开始先自旋不放弃 CPU，自旋多次后仍然加不上，就开始执行 usleep 休眠；如果休眠的时间设置为 0，则调用 sched_yield 放弃 CPU 给其他任务执行机会，但不会离开 CPU 运行队列执行休眠，确保低延迟响应。

```
void Sleep() {
        if (loop < SpinMaxCount) {
            asm volatile("pause"); //这个指令让 CPU 节点能，环保用的。
            loop++;
        }
        else {
            if ( SleepTime <= 0 )
                sched_yield();
            else
                usleep(SleepTime);
        }
}
```

## 6.2 读写自旋锁

simple::RWSpinLock，使用一个 int 记录读者和写者。

volatile unsigned int lock_ __attribute__ ((aligned (4)));

读者加锁对 lock_ 加 2，解锁对 lock_ 减 2；最大可以满足 2^31 个读并发。

写者加锁对 lock_ 置 1，解锁对 lock_ 置 0。

只有写者会修改第一个 bit，所以读者可以根据这个 bit 判断是否有写者存在。

其他细节和 simple::SpinLock 一致。

## 6.3 SPSC

simple:: SPSC 内部采用一个 vector 存储环形队列，并且需要预先指定 vector 的大小，例如：

SPSC<string, 1000> spsc;

定义了一个单进单出的环形队列，大小是 1000，存储对象是 string。

两个游标 p_和 c_分别记录生产者和消费者的位置，初始都指向 0，队列为空。

Enqueue 入队，如果返回 false 说明队列满了，由调用者选择处理方式，或休眠或做其他事情。

```
bool Enqueue(const Tp& v) {
        if (p_ + 1 == c_) return false;
        queue_[p_.cur_] = v;
        ++p_;
        return true;

}
```

入队之后，更新游标 p_之前，需要加入内存屏障，确保消费者看到 p_更新时，元素已经在队列里面了。

Dequeue 出队，如果返回 false 说明队列为空，由调用者选择处理方式，或休眠或做其他事情。

```
bool Dequeue(Tp& v) {
        if (c_ == p_) return false;
        v = queue_[c_.cur_];
        ++c_;
        return true;
}
```

出队之后，更新游标 c_之前，需要加入内存屏障，确保生产者看到 c_更新时，元素不在队列里面了。

由于是环形的，p_和 c_到 vector 尾端时需要从头来， Iter 对象对这个过程做了封装。

显然游标值必须是内存变量，所以声明为 volatile，而且应该保持对齐，确保改写它是原子

的。

volatile size_t cur_ __attribute__ ((aligned (8)));

对它改写时，**先在临时变量上生成最终新值（包括了到尾端后的调整）**，然后写进内存，写之前加入屏障 - 这是 SPSC 实现最容易出错的地方，如果直接对游标修改，中间值会被对方看到。

```
Iter& operator++() {
        Iter tmp = *this + 1;
        asm volatile("" : : : "memory");
        cur_ = tmp.cur_; //atomic
        return *this;
}
```

## 6.4 Singleton

演示 CAS 的使用而已，实际上单例也就第一次创建时存在竞争，使用 Double Check 之后，是否免锁没有什么区别。

## 6.5 test

测试程序。

TestPthreadMutex 等锁的测试程序接收三个参数：

第一个参数是线程数；第二个参数用于控制持锁时长； 第三个参数是写操作百分比。

./TestPthreadMutex 4 10 10

输出：

Time used: 5s

| Thread | Read | Read/s | Write | Write/s | R&W/s |
|---|---|---|---|---|---|
| 0 | 2873 | 574 | 273550 | 54710 | 55284 |
| 1 | 1103 | 220 | 103386 | 20677 | 20897 |
| 2 | 1245 | 249 | 115736 | 23147 | 23396 |
| 3 | 1074 | 214 | 103061 | 20612 | 20827 |
| Total | 6295 | 1259 | 595733 | 119146 | 120405 |

Avg locked time: 8948

第一行是测试持续时间

最后一行是持锁平均时间，单位是 ns。

中间是的性能统计值（进入临界区的次数）。

# 7 附录

## 7.1 内存屏障

**考虑如下代码：**

| 线程 A: | 线程 B: |
| --- | --- |
| a=malloc(); | if(b==true) |
| b=true; | {access a} |

a 和 b 均为内存变量，代码逻辑上并没有问题，但遗憾的是，当 b 为 true 时，a 不一定已经被赋值为一个合法的内存地址。也就是说对 a 和 b 指向的两个内存位置的访问顺序并不是严格按照代码顺序来的。

有四个层次的顺序

1. 代码[8]顺序：就是我们 C/C++代码的顺序，这是我们需要的正确的逻辑顺序。
2. 指令顺序：编译器生成的指令顺序，这个顺序可能会和代码顺序不一致，编译器出于优化的目的可能会把没有依赖关系的指令打乱。
3. 执行顺序：CPU 执行指令的顺序，这个顺序可能会和指令顺序不一致，CPU 出于优化的目的可能会把没有依赖关系的指令打乱。
4. 感知顺序：CPU 感知到的自己或其他 CPU 对内存操作的顺序。这个顺序可能和执行顺序不一样，Cache 机制、内存系统方面的优化可能会导致先对内存的操作却后被生效（感知）。我们需要的是感知顺序和代码顺序一致。而内存屏障就是做这个事情的。

内存屏障分为两种:

1. 优化屏障，优化屏障提示编译器不要做优化，确保屏障代码前面的所有对内存的操作均在其后面对内存的操作之前完成。GCC 插入优化屏障的方法是：

| a=malloc(); | if(b==true) |
| --- | --- |

[8] 指C/C++等程序员编写而非编译器生成的代码。

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

b=true;

asm volatile("" : : : "memory"); 是一条内联汇编代码，起到屏障的作用。

优化屏障确保指令顺序和代码顺序的一致。

2. 硬件屏障， 硬件屏障是具体平台架构提供的特殊指令，硬件屏障通常分为读屏障、写屏障、读写屏障。 顾名思义，读屏障确保读顺序、写屏障确保写顺序、读写屏障确保读写顺序。

X86 提供的三个屏障指令。

- LFENCE instructions cannot pass earlier reads.
- SFENCE instructions cannot pass earlier writes.
- MFENCE instructions cannot pass earlier reads or writes.

GCC 环境下插入 X86 硬件屏障也很简单。

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

也可以借鉴内核代码，把这些内联汇编定义成可读性好的宏。

#define mb()  asm volatile("mfence":::"memory")

#define rmb()  asm volatile("lfence":::"memory")

#define wmb()  asm volatile("sfence" ::: "memory")

硬件屏障确保感知顺序和指令顺序一致，但硬件屏障对应的汇编代码，也包括了优化屏障，所以硬件屏障代码可以确保感知顺序和代码顺序一致。

回到我们上面一开始提到的例子，实际上在已有的 x86 架构上只要插入了优化屏障就可以了，

| 线程 A: | 线程 B: |
| --- | --- |
| a=malloc(); | if(b==true) |
| asm volatile("" : : : "memory"); | |
| b=true; | {access a} |

只要编译器不乱序，对于一般的写操作 x86 是不会做乱序处理的。[9]

---

[9] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A-System Programming Guide. 8.2 MEMORY ORDERING
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

但是 INTEL 在他的手册里面也说了，内存顺序的机制随着 x86 的架构演进可能会变化。总之要是实在摸不准，我们直接使用硬件屏障语句就好了。

如果你使用 mutex spinlock 这样的锁机制，你不需要担心内存屏障，这些锁的底层实现隐含了内存屏障的处理。

如果在 **X86** 上使用__sync_fetch_and_add 这一组原子方法，也是不需要屏障的，因为 **lock** 前缀隐含了内存屏障的处理。

### 7.2 simple 库代码

Git: N/a

# 8 参考资料


[1] Linux Source Code http://www.kernel.org/

[2] Memory Ordering in Modern Microprocessors Paul E. McKenney http://www.linuxjournal.com/article/8211

[3] Facebook folly source code https://github.com/facebook/folly/

[4] Intel 64 and IA-32 Architectures Software Developer's Manual http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[5] GCC online doc http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html