原 自旋锁与互斥锁的对比、手

2016年12月16日 17:31:50 阅读数：11324 标签: 旋锁 锁 更多

个人分类： 多线程

本文之前，我只是对自旋锁有所了解，知道 甚至以为自旋锁只有kernel用这个 才发现POSIX有提供自旋锁的接口。下面我会分析一下自旋锁，并代码实现自旋锁和 旋锁。

一：自旋锁（spin lock）

自旋锁是一种用于保护多线程共享资源的锁，与一般互斥锁（mutex）不同之处在于当自旋锁尝试获取锁时以忙等待（busy waiting）的形式不断地循环检查锁是否可用。在多CPU的环境中，对持有锁较短的程序来说，使用自旋锁代替一般的互斥锁往往能够提高程序的性能。

最后标红的句子很重要，本文将针对该结论进行验证。

下面是man手册中对自旋锁pthread_spin_lock()函数的描述：

DESCRIPTION

The pthread_spin_lock() function shall lock the spin lock referenced by lock. The calling thread shall acquire the lock if it is not held by another thread. Otherwise, the thread shall spin (that is, shall not return from the pthread_spin_lock() call) until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made. The pthread_spin_trylock() function shall lock the spin lock referenced by lock if it is not held by any thread. Otherwise, the function shall fail.

The results are undefined if any of these functions is called with an uninitialized spin lock.

可以看出，自选锁的主要特征：当自旋锁被一个线 程尝试去phtread_spin_lock()获得该锁，那么它将不会从该函数返回，而是一直自旋（spin），直到自旋锁可用为止。

使用自旋锁时要注意：

- 由于自旋时不释放CPU，因而持有自旋锁的线程 会一直在哪里自旋，这就会浪费CPU时
- 持有自旋锁的线程在sleep之前应该释放自旋锁以 果持有自旋锁的代码sleep了就可能导 系统挂起。(下面会解释)

使用任何锁都需要消耗系统资源（内存资源和CPU

1.建立锁所需要的资源
2.当线程被阻塞时所需要的资源

POSIX提供的与自旋锁相关的函数有以下几个，都在

```
int pthread_spin_init(pthread_spinlock_t
```

初始化spin lock， 当线程使用该函数初始化一个未初始化或者被destroy过的spin lock有效。该函数会为spin lock申请资源并且初始化spin lock为unlocked状态。
有关第二个选项是这么说的：

If the Thread Process-Shared Synchronization option is supported and the value of pshared is PTHREAD_PROCESS_SHARED, the implementation shall permit the spin lock to be operated upon by any thread that has access to the memory where the spin lock is allocated, even if it is allocated in memory that is shared by multiple processes.

If the Thread Process-Shared Synchronization option is supported and the value of pshared is PTHREAD_PROCESS_PRIVATE, or if the option is not supported, the spin lock shall only be operated upon by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined.

所以，如果初始化spin lock的线程设置第二个参数为PTHREAD_PROCESS_SHARED，那么该spin lock不仅被初始化线程所在的进程中所有线程看到，而且可以被其他进程中的线程看到，PTHREAD_PROESS_PRIVATE则只被同一进程中线程看到。如果不设置该参数，默认为后者。

```
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

销毁spin lock，作用和mutex的相关函数类似，就不翻译了：

The pthread_spin_destroy() function shall destroy the spin lock referenced by lock and release any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to

pthread_spin_init(). The results are undefined if holds the lock, or if this
function is called with an uninitialized thread spin
不过和mutex的destroy函数一样有这样的性质（当初
The result of referring to copies of that object in calls pthread_spin_try-
lock(), or pthread_spin_unlock() is undefined.

```
int pthread_spin_lock(pthread_spinlock_t
```

加锁函数，功能上文都说过了，不过这么一点值得注
EBUSY A thread currently holds the lock.
These functions shall not return an error code of

```
int pthread_spin_trylock(pthread_spinloc
```

还有这个函数，这个一般很少用到。

```
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

解锁函数。不是持有锁的线程调用或者解锁一个没有lock的spin lock这样的行为都是undefined的。

二：自旋锁和互斥锁的区别

以下该段内容引自http://blog.chinaunix.net/uid-28711483-id-4995776.html：

从实现原理上来讲，Mutex属于sleep-waiting类型的 锁。例如在一个双核的机器上有两个线程(线程A和线程B)，它们分别运行在Core0和Core1上。假设线程A想要通过 pthread_mutex_lock 操作去得到一个临界区的锁，而此时这个锁正被线程B所持有，那么线程A就会被阻塞(blocking)，Core0 会在此时进行上下文切换(Context Switch)将线程A置于等待队列中，此时Core0就可以运行其他的任务(例如另一个线程C)而不必进行忙等待。而Spin lock则不然，它属于busy-waiting类型的锁，如果线程A是使用pthread_spin_lock操作去请求锁，那么线程A就会一直在 Core0上进行忙等待并不停的进行锁请求，直到得到这个锁为止。

如果大家去查阅Linux glibc中对pthreads API的实现NPTL(Native POSIX Thread Library) 的源码的话(使用"getconf GNU_LIBPTHREAD_VERSION"命令可以得到我们系统中NPTL的版本号)，就会发现pthread_mutex_lock()操作如果 没有锁成功的话就会调用system_wait()的系统调用并将当前线程加入该mutex的等待队列里。而spin lock则可以理解为在一个while(1)循环中用内嵌的汇编代码实现的锁操作(印象中看过一篇论文介绍说在linux内核中spin lock操作只需要两条CPU指令，解锁操作只用一条指令就可以完成)。有兴趣的朋友可以参考另一个名为sanos的微内核中pthreds API的实现：mutex.c spinlock.c，尽管与NPTL中的代码实现不尽相同，但是因为它的实现非常简单易懂，对我们理解spin lock和mutex的特性还是很有帮助的。

对于自旋锁来说，它只需要消耗很少的资源来建 查看锁是否可用了，也就是说当自旋锁处于等待状态时它会一直消耗CPU时间。

对于互斥锁来说，与自旋锁相比它需要消耗大量 呈的调度状态被修改，并且线程被加入等待线程队列；最后当锁可用 时，在获取锁之前，线程会被从等待队列取出并更改其调度状态；但是在

因此自旋锁和互斥锁适用于不同的场景。自旋锁 类适用于那些可能会阻塞很长时间的场

四：自旋锁与linux内核进程调

现在我们就来说一说之前的问题，如果临界区可能 能引起死锁：

那么为什么信号量保护的代码可以睡眠而自旋锁会

先看下自旋锁的实现方法吧，自旋锁的基本形式如

```
spin_lock(&mr_lock):

//critical region

spin_unlock(&mr_lock);
```

跟踪一下spin_lock(&mr_lock)的实现

```
#define spin_lock(lock) _spin_lock(lock)
#define _spin_lock(lock) __LOCK(lock)
#define __LOCK(lock) \
do { preempt_disable(); __acquire(lock); (void)(lock); } while (0)
```

注意到“preempt_disable()”，这个调用的功能是“关抢占”（在spin_unlock中会重新开启抢占功能）。从中可以看出，使用自旋锁保护的区域是工作在非抢占的状态；即使获取不到锁，在“自旋”状态也是禁止抢占的。了解到这，我想咱们应该能够理解为何自旋锁保护 的代码不能睡眠了。试想一下，如果在自旋锁保护的代码中间睡眠，此时发生进程调度，则可能另外一个进程会再次调用spinlock保护的这段代码。而我们 现在知道了即使在获取不到锁的“自旋”状态，也是禁止抢占的，而“自旋”又是动态的，不会再睡眠了，也就是说在这个处理器上不会再有进程调度发生了，那么 死锁自然就发生了。

总结下自旋锁的特点：

- 单CPU非抢占内核下：自旋锁会在编译时被忽略（因为单CPU且非抢占模式情况下，不可能发生进程切换，时钟只有一个进程处于临界区（自旋锁实际没什么用了）
- 单CPU抢占内核下：自选锁仅仅当作一个设置抢占的开关（因为单CPU不可能有并发访问临界区的情况，禁止抢占就可以保证临街区唯一被拥有）
- 多CPU下：此时才能完全发挥自旋锁的作用，自旋锁在内核中主要用来防止多处理器中并发访问临界区，防止内核抢占造成的竞争。

五：linux发生抢占的时间

linux抢占发生的时间，抢占分为**用户抢占**和**内核抢**

用户抢占在以下情况下产生：

- 从系统调用返回用户空间
- 从中断处理程序返回用户空间

内核抢占会发生在：

- 当从中断处理程序返回内核空间的时候，且当时
- 当内核代码再一次具有可抢占性的时候（如：sp
- 如果内核中的任务显示的调用schedule()　　（这

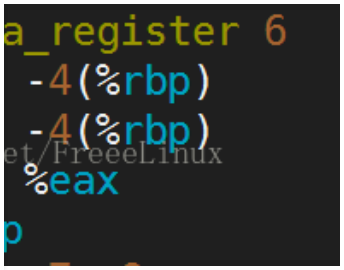
基本的进程调度就是发生在时钟中断后，并且发现进程的时间片已经使用完了，则发生进程抢占。通常我们会利用中断处理程序返回内核空间的时候可进行内核抢占这个特性来提高一些I/O操作的实时性，如：当I/O事件发生的时候，对应的中断处理程序被激活，当它发现有进程在等待这个I/O事件的时候，它 会激活等待进程，并且设置当前正在执行进程的need_resched标志，这样在中断处理程序返回的时候，调度程序被激活，原来在等待I/O事件的进程 （很可能）获得执行权，从而保证了对I/O事件的相对快速响应（毫秒级）。可以看出，在I/O事件发生的时候，I/O事件的处理进程会抢占当前进程，系统 的响应速度与调度时间片的长度无关。

六：spin_lock和mutex实际效率对比

1.++i是否需要加锁?

我分别使用POSIX的spin_lock和mutex写了两个累加的程序，启动了两个线程，并利用时间戳计算它们执行完累加所用的时间。

下面这个是使用spin_lock的代码，我启动两个线程同时对num进行++，使用spin_lock保护临界区，实际上可能会有疑问++i（++i和++num本文中是一个意思）为什么还要加锁?
i++需要加锁是很明显的事情，对i++的操作的印象是，它一般是三步曲，从内存中取出i放入寄存器中，在寄存器中对i执行inc操作，然后把i放回内存中。这三步明显是可打断的，所以需要加锁。
但是++i可能就有点犹豫了。实际上印象流是不行的，来看一下i++和++i的汇编代码，其实他们是一样的，都是三步，我只上一个图就行了，如下：



所以++i也不是原子操作，在多核的机器上，多个线就导致多个线程同时执行+1，但实际得到的结果是一样的，即i只加了一次。还有一点：这几句汇编正说明了++i和i++i对于效率是一样的ss的话，我们都写过类的++运算符的重果一个类在单个语句中不写++i，而是写i++的话，那无疑效率会有很大的损耗。（有点跑题）

2.spin_lock代码

首先是spin_lock实现两个线程同时加一个数，每个线

```
#include <iostream>
#include <thread>

#include <pthread.h>
#include <sys/time.h>
#include <unistd.h>

int num = 0;
pthread_spinlock_t spin_lock;

int64_t get_current_timestamp()
{
    struct timeval now = {0, 0};
    gettimeofday(&now, NULL);
    return now.tv_sec * 1000 * 1000 + now.tv_usec;
}

void thread_proc()
{
    for(int i=0; i<100000000; ++i){
        pthread_spin_lock(&spin_lock);
        ++num;
```

```
        pthread_spin_unlock(&spin_lo
}

int main()
{
    pthread_spin_init(&spin_lock, PT                    ROCESS_PRIVATE or PTHREAD_PROC      ARED

    int64_t start = get_current_time

    std::thread t1(thread_proc), t2(
    t1.join();
    t2.join();

    std::cout<<"num:"<<num<<std::end
    int64_t end = get_current_timest
    std::cout<<"cost:"<<end-start<<s

    pthread_spin_destroy(&spin_lock);

    return 0;
}
```


3.mutex代码

复制

```
#include <iostream>
#include <thread>

#include <pthread.h>
#include <sys/time.h>
#include <unistd.h>

int num = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

```

```
int64_t get_current_timestamp()12 {
    struct timeval now = {0, 0};
    gettimeofday(&now, NULL);
    return now.tv_sec * 1000 * 1000
}

void thread_proc()
{
    for(int i=0; i<1000000; ++i){
        pthread_mutex_lock(&mutex);
        ++num;
        pthread_mutex_unlock(&mutex)
    }
}

int main()
{
    int64_t start = get_current_timestamp();
   std::thread t1(thread_proc), t2(thread_proc);
    t1.join();
    t2.join();
    std::cout<<"num:"<<num<<std::endl;
    int64_t end = get_current_timestamp();
    std::cout<<"cost:"<<end-start<<std::endl;

    pthread_mutex_destroy(&mutex);    //maybe you always foget this

    return 0;
}
```


3

2

4.结果分析

得出的结果如图，num是最终结果，cost是花费时间，单位为us，main2是使用spin lock，

```
vagrant@vagrant-ubuntu-wil
vagrant@vagrant-ubuntu-wil
num:2000000
cost:44208
vagrant@vagrant-ubuntu-wil
num:2000000
cost:53362
vagrant@vagrant-ubuntu-wil
num:2000000
cost:45504
vagrant@vagrant-ubuntu-wil
num:2000000
cost:59681
```



```
ck$ g++ -o main2 sys_spin_lock.cpp -lpthread
ck$ ./main2

ck$ ./main3

et/FreeeLinux
ck$ ./main2

ck$ ./main3
```

显然，在临界区只有++num这一条语句的情况下，spin lock相对花费的时间短一些，实际上它们有可能接近的情况，取决于CPU的调度情况，但始终会是spin lock执行的效率在本情况中花费时间更少。

我修改了两个程序中临界区的代码，改为：

```
1    for(int i=0; i<1000000; ++i){
2         pthread_spin_lock(&spin_lock);
3         ++num;
4         for(int i=0; i<100; ++i){
5             //do nothing
6         }
7         pthread_spin_unlock(&spin_lock);
8     }
```

另一个使用mutex的程序也加了这么一段，然后结果就与之前的情况大相径庭了：

实验结果是如此的明显，仅仅是在临界区内加了一个10圈的循环，spin lock就需要花费比mutex更长的时间了。

所以，spin lock虽然lock/unlock的性能更好（花费很少的CPU指令），但是它只适应于临界区运行时间很短的场景。实际开发中，程序员如果对自己程序的锁行为不是很了解，否则使用spin lock不是一个好主意。更保险的方法是使用mutex，如果对性能有进一步的要求，那么再考虑spin lock。

七：使用C++实现自主实现自旋锁

由于前面原理已经很清楚了，现在直接给代码如下：

```
#pragma once

#include <atomic>

class spin_lock {
private:
    std::atomic<bool> flag = ATOMIC_VAR_INIT(false);
public:
    spin_lock() = default;
    spin_lock(const spin_lock&) = delete;
    spin_lock& operator=(const spin_lock) = delete;
    void lock(){   //acquire spin lock
```

```
        bool expected = false;
            expected = false;
    }
    void unlock(){   //release spin
        flag.store(false);
    }
};
```

测试文件，仅给出关键部分：

```
int num = 0;
spin_lock sm;

void thread_proc()
{
    for(int i=0; i<10000000; ++i){
        sm.lock();
        ++num;
        sm.unlock();
    }
}
```

参考：http://blog.chinaunix.net/uid-28711483-id-4995776.html
http://blog.chinaunix.net/uid-21411227-id-1826888.html
http://blog.poxiao.me/p/spinlock-implementation-in-cpp11/

好的，对自旋锁的总结就先到这里了。

想对作者说点什么

nice_wen： 感谢博主，写的很棒呢！ (06-23 1

来离： 感谢博主 (03-12 17:14 #1楼)

3

2

自旋锁（spinlock） 解释得经典，透彻 1.4万

http://blog.csdn.net/unbutun/article/details/5730037 用者睡眠，如果自旋锁已经被别…

自旋锁和互斥锁区别 6.1万

http://blog.csdn.net/kyokowl/article/details/6294341 行并行编程的一套常...

自旋锁、阻塞锁、重入锁、偏向锁、轻量锁和重量锁 8468

关于并发编程下的各种锁机制的简单介绍和总结

Linux内核同步方法——自旋锁（spin lock） 433

自旋锁 Linux的的内核最常见的锁是自旋锁。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获得一个被已经持有（…

Synchronized的原理及自旋锁，偏向锁，轻量级锁，重量级锁的区别 2405

在多线程并发编程中Synchronized一直是元老级角色，很多人都会称呼它为重量级锁，但是随着Java SE1.6对Synchronized进行了各…

自旋锁 658

Linux 设备驱动中必须解决的一个问题是多个进程对共享资源的并发访问，并发访问会导致竞态，linux 提供了多种解决竞态问题的方式…

程序员不会英语怎么行?

老司机教你一个数学公式秒懂天下英语

自旋锁-概念 814

不放弃CPU时间 线程获取不到锁,就会被阻塞挂起,等… …和唤醒是需要转入到内核态完成…

3

自旋锁和互斥锁的区别 java中lock Syntr 4051

转载自：http://blog.csdn.net/susidian/article/details/5… …类似，只是自旋锁不会引起调用…

2

java 中的锁 -- 偏向锁、轻量级锁、自旋锁 4.1万

之前做过一个测试，详情见这篇文章《多线程 +1操作… …试结果很疑惑，反复执行过多次…

内核锁 spin_lock 与 mutex_lock 区别? 4569

本文由该问题引入到内核锁的讨论，归纳如下 为什么… …于内核态的情况，而在内核态下，…

相关热词 自旋锁衍生自旋锁 win32自旋锁 jdk自旋…

个人资料

wilcohuang 关注

原创	粉丝	喜欢	评论
351	101	73	32

等级： 博客 6 访问： 36万+

积分： 7194 排名： 4376

勋章：

3

2

博主专栏

muduo源码剖析

阅读量：20905 28 篇

libevent源码剖析

阅读量：11251 6 篇

STL源码剖析

阅读量：1515 3 篇

个人分类

PHP	7篇
C/C++	50篇
Python	5篇

汇编语言 4篇

展开

归档

2018年3月 1篇

2018年2月 1篇

2017年12月 10篇

2017年11月 9篇

2017年9月 2篇

展开

联系我们

请扫描二维码联系客服

webmaster@csdn.net

400-660-0108

QQ客服 客服论坛

关于 招聘 广告服务 网站地图

百度提供搜索支持

经营性网站备案信息

网络110报警服务

中国互联网举报中心

北京互联网违法和不良信息举报中心

3

2