C++小品：榨干性能：C++11中的原子操作（atomic operation） - [C++11 FAQ]

http://imcc.blogbus.com/logs/179131763.html

所谓的原子操作，取的就是"原子是最小的、不可分割的最小个体"的意义，它表示在多个线程访问同一个全局资源的时候，能够确保所有其他的线程都不在同一时间内访问相同的资源。也就是他确保了在同一时刻只有唯一的线程对这个资源进行访问。这有点类似互斥对象对共享资源的访问的保护，但是原子操作更加接近底层，因而效率更高。

在以往的C++标准中并没有对原子操作进行规定，我们往往是使用汇编语言，或者是借助第三方的线程库，例如intel的pthread来实现。在新标准C++11，引入了原子操作的概念，并通过这个新的头文件提供了多种原子操作数据类型，例如，atomic_bool,atomic_int等等，如果我们在多个线程中对这些类型的共享资源进行操作，编译器将保证这些操作都是原子性的，也就是说，确保任意时刻只有一个线程对这个资源进行访问，编译器将保证，多个线程访问这个共享资源的正确性。从而避免了锁的使用，提高了效率。

我们还是来看一个实际的例子。假若我们要设计一个广告点击统计程序，在服务器程序中，使用多个线程模拟多个用户对广告的点击：

```
#include <boost/thread/thread.hpp>
#include <atomic>
#include <iostream>
#include <time.h>

using namespace std;
// 全局的结果数据
long total = 0;

// 点击函数
void click()
{
for(int i=0; i<1000000;++i)
{
// 对全局数据进行无锁访问
total += 1;
}
}
```

```
int main(int argc, char* argv[])
{
// 计时开始
clock_t start = clock();
// 创建100个线程模拟点击统计
boost::thread_group threads;
for(int i=0; i<100;++i)
{
threads.create_thread(click);
}

threads.join_all();
// 计时结束
clock_t finish = clock();
// 输出结果
cout<<"result:"<<total<<endl;
cout<<"duration:"<<finish -start<<"ms"<<endl;
return 0;
}
```

从执行的结果来看，这样的方法虽然非常快，但是结果不正确

```
E:\SourceCode\MinGW>thread.exe
result:87228026
duration:528ms
```

很自然地，我们会想到使用互斥对象来对全局共享资源的访问进行保护，于是有了下面的实现：

```
long total = 0;
// 对共享资源进行保护的互斥对象
mutex m;

void click()
{
for(int i=0; i<1000000;++i)
{
// 访问之前，锁定互斥对象
m.lock();
total += 1;
// 访问完成后，释放互斥对象
m.unlock();
}
}
```

互斥对象的使用，保证了同一时刻只有唯一的一个线程对这个共享进行访问，从执行的结果来看，互斥对象保证了结果的正确性，但是也有非常大的性能损失，从刚才的528ms变成了现在的8431，用了原来时间的10多倍的时间。这个损失够大。

```
E:\SourceCode\MinGW>thread.exe
result:100000000
duration:8431ms
```

如果是在C++11之前，我们的解决方案也就到此为止了，但是，C++对性能的追求是永无止境的，他总是想尽一切办法榨干CPU的性能。在C++11中，实现了原子操作的数据类型（atomic_bool,atomic_int,atomic_long等等），对于这些原子数据类型的共享资源的访问，无需借助mutex等锁机制，也能够实现对共享资源的正确访问。

```
// 引入原子数据类型的头文件
#include <atomic>

// 用原子数据类型作为共享资源的数据类型
atomic_long total(0);
//long total = 0;

void click()
{
for(int i=0; i<1000000;++i)
{
// 仅仅是数据类型的不同而以，对其的访问形式与普通数据类型的资源并无区别
total += 1;
}
}
```

我们来看看使用原子数据类型之后的效果如何：

```
E:\SourceCode\MinGW>thread.exe
result:100000000
duration:2105ms
```

结果正确！耗时只是使用mutex互斥对象的四分之一！也仅仅是不采用任何保护机制的时间的4倍。可以说这是一个非常不错的成绩了。

原子操作的实现跟普通数据类型类似，但是它能够在保证结果正确的前提下，提供比mutex等锁机制更好的性能，如果我们要访问的共享资源可以用原子数据类型表示，那么在多线程程序中使用这种新的等价数据类型，是一个不错的选择。

Similar Posts:

- 特权同学笔记-榨干FPGA片上存储资源

  榨干FPGA片上存储资源 记得Long long time ago,特权同学写过一篇简短的博文<M4K使用率>,文章中提到了Cyclone器件的内嵌存储块M4K的配置问题.文中提到了这个M4K块除了存储大小是有限的4Kbit,它的可配置的Port数量也是有限的,通常为最大36个可用port. 当时只是简单的提到有这么回事,提醒使用者注意,也没有具体的谈到如何解决或者确切的说应该是避免这样的状

况出现.因此,本文将结合特权同学近期在使用FPGA时,配置片内存储器遇到的一些片内资源无法得到充分利用的

- C++11中的POD和Trivial

  引子 在介绍C++11的文章或者博客中,经常会出现POD类型和Trivial类型的影子.但是POD类型和Trivial类型到底是什么意思呢? POD类型 POD类型的好处 POD类型 粗略上来讲,POD是C++为兼容C的内存布局而设计的,主要用于修饰用户自定义类型.但POD却远比这个要复杂.POD(Plain Old Data),从字面意思上来看,是平凡普通的老旧格式的数据,POD是一个类型属性,既不是关键字也不会像"volatile"用来修饰类型信息. POD类型,说明该数据是普通的

- 如何榨干cpu的每一滴资源（parallel computing in R）

  前言 现在计算机的cpu和system基本都采用了multicore技术(intel i3 2cores.i5 4cores-),但是Windows下我们常常在设计程序时大多是单线程的,然而这却是一种对cpu资源的极度浪费.网上看到过一个动图觉得很能讽刺这种情况 随着R语言学习的加深,也许大家都会觉得处理速度越来越力不从心,那么我们就该考虑如何让我们的程序跑的更快了: ①少用for循环,替而代之的是apply函数族 ②将底层计算交给C语言(package "Rcpp") ③并行计算(p

- IE 11中 onpropertychange失效

  https://msdn.microsoft.com/zh-cn/library/ie/dn265032(v=vs.85).aspx 将突变事件和属性更改事件迁移到突变观察者 Internet Explorer 11 中的突变观察者提供了对突变事件支持的所有相同方案的快速执行替换,以及对属性更改事件支持的方案的替换. 你可以使用突变事件和/或属性更改事件迁移现有代码,以使用突变观察者. 监视 DOM 突变的传统技术 Mutation events 在 Web 平台中扮演了关键角色.这些事件允许

- C++11中值得关注的几大变化

  赖勇浩(http://laiyonghao.com) 声明:本文源自 Danny Kalev 在 2011 年 6 月 21 日发表的<The Biggest Changes in C++11(and Why You Should Care)>一文,几乎所有内容都搬了过来,但不是全文照译,有困惑之处,请参详原文(http://www.softwarequalityconnection.com/2011/06/the-biggest-changes-in-c11-and-why-you-shou

- paip.提升性能--多核编程中的java .net php c++最佳实践 v2.0 cah

  paip.提升性能--多核编程中的java .net php c++最佳实践 v2.0 cah 作者Attilax 艾龙, EMAIL:1466519819@qq.com 来源:attilax的专栏 地址:http://blog.csdn.net/attilax //////////多核编程的方法: 1.等候jvm等直接支持多核 2.框架实现 OpenMP 3.使用并发api (FutureTask.ExecutorService) 推荐 4.使用传统muilt thread 作者Attila

- C++11 中的双重检查锁定模式

  双重检查锁定模式(DCLP)在无锁编程方面是有点儿臭名昭著案例学术研究的味道.直到2004年,使用java开发并没有安全的方式来实现它.在c++11之前,使用便捷式c+开发并没有安全的方式来实现它.由于引起人们关注的缺点模式暴露在这些语言之中,人们开始写它.一组高调的java聚集在一起开发人员并签署了一项声明,题为:"双重检查锁定坏了".在2004年斯科特 .梅尔斯和安德烈.亚历山发表了一篇文章,题为:"c+与双重检查锁定的危险"对于DCLP是什么?这两篇文章都是伟

- C++11 中的右值引用与转移语义

  本文介绍了 C++11 标准中的一个特性,右值引用和转移语义.这个特性能够使代码更加简洁高效. 新特性的目的 右值引用 (Rvalue Referene) 是 C++ 新标准 (C++11, 11 代表 2011 年) 中

引入的新特性，它实现了转移语义 (Move Sementics) 和精确传递 (Perfect Forwarding).它的主要目的有两个方面: 消除两个对象交互时不必要的对象拷贝,节省运算存储资源,提高效率. 能够更简洁明确地定义泛型函数. 左值与右值的定义 C++(包

- C++11中的string - to_string/stoi

  转自 IBM 编译器中国开发团队的<C++11中的string - atoi/itoa> 在C++11中,由于右值引用的引入,常为人所诟病std::string的性能问题得到了很大的改善.另外一方面,我们也可以看到新语言为std::string类增加了很多新的api.比较引人注意的就是std::string的成员函数stoi系列,以及std::to_string全局函数.这两种API虽然很不起眼,却为C++11的格式化输出(formatted I/O)增加了一种实用的手段.我们可以依序会议一下
- 关于C++11中的std::move和std::forward

  std::move是一个用于提示优化的函数,过去的c++98中,由于无法将作为右值的临时变量从左值当中区别出来,所以程序运行时有大量临时变量白白的创建后又立刻销毁,其中又尤其是返回字符串std::string的函数存在最大的浪费. 比如: 1 std::string fileContent = "oldContent";2 s = readFileContent(fileName); 因为并不是所有情况下,C++编译器都能进行返回值优化,所以,向上面的例子中,往往会创建多个字符串.re