C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B)

多线程编程 (/tags/#%E5%A4%9A%E7%BA%BF%E7%A8%8B%E7%BC%96%E7%A8%8B)

C++ 并发编程笔记(四)

C++ 并发编程笔记(四)

Posted by 敬方 on July 6, 2019

> *2019-07-17 22:16:53*

第八章 并发代码设计

8.1 线程间划分工作的技术

一般的线程划分会，直接将数据划分线程；但是最后的多线程之间的结果数据同步会造成较多的麻烦，数据和算法符划分方式，在多线程编程中较为重要。

8.1.2 递归划分

第四章中的 `std::async()` 方式的快速排序，在对于大量数据进行排序的时候，每一层递归都会产生一个新线程，最终会产生大量的线程；线程的性能开销反而会让程序的执行时间上升，不如单线程的快速排序。

可以通过将数据打包后，交给固定线程处理，或者使用 `std::thread::hardware_concurrency()` 函数来确定线程的数量。

使用栈的并行快速排序算法–等待数据块排序

```
template<typename T>
struct sorter
{
    //准备排序的数据块

    struct chunk_to_sort
    {
        //排序数据

        std::list<T> data;
        //预期结果

        std::promise<std::list<T> > promise;
    };
    //线程安全的排序栈

    thread_safe_stack<chunk_to_sort> chunks;
    //工作线程列表

    std::vector<std::thread> threads;
    //最大线程数量

    unsigned const max_thread_count;
    //原子变量，是否达到数据末尾

    std::atomic<bool> end_of_data;

    sorter():max_thread_count(std::thread::hardware_concurrency()-1),
            end_of_data(false){}
    ~sorter()
    {
        end_of_data=true;
        for(unsigned i=0;i<threads.size();++i)
        {
            //等待线程结束

            threads[i].join();
        }
```

```
    }
    void try_sort_chunk()
    {
        //获取排序线程

        boost::shared_ptr<chunk_to_sort > chunk=chunks.pop();
        if(chunk)
        {
            //开始排序

            sort_chunk(chunk);
        }
    }
    std::list<T> do_sort(std::list<T>& chunk_data)
    {
        if(chunk_data.empty())
        {
            return chunk_data;
        }
        std::list<T> result;
        //拷贝数据的头指针

        result.splice(result.begin(),chunk_data,chunk_data.begin());
        //获取开头的数据

        T const& partition_val=*result.begin();
        //根据partition_val进行分组，获取中间分组的迭代器

        typename std::list<T>::iterator divide_point=std::partition(chunk_data.begin(),chunk_data.end(),[&](T const& val){return va
        chunk_to_sort new_lower_chunk;
        //截取分组的前半段到new_lower_chunk

        new_lower_chunk.data.splice(new_lower_chunk.data.end(),
            chunk_data,
            chunk_data.begin(),
            divide_point
            );
        //获取new_chunk的future值
```

```
        std::future<std::list<T> > new_lower=new_lower_chunk.promise.get_future();
        //压入排序栈

        chunks.push(std::move(new_lower_chunk));
        //如果线程小于最大线程数

        if(threads.size()<max_thread_count)
        {
            //创建排序线程

            threads.push(std::thread(&sorter<T>::sort_thread,this));

        }
        //获取大于的一部分，并返回排序结果

        std::list<T> new_higher(do_sort(chunk_data));
        //接上new_higher

        result.splice(result.end(),new_higher);
        //循环执行排序线程，直到结束

        while(new_lower.wait_for(std::chrono::seconds(0))!=std::future_status::ready)
        {
            try_sort_chunk();
        }
        //将前面排序结果放到result中

        result.splice(result.begin(),new_lower.get());
    }
    void sort_thread()
    {
        while(!end_of_data)
        {
            //没有到达数据末尾，尝试快排

            try_sort_chunk();
            //线程休眠

            std::this_thread::yield();
```

```
        }
    }

};
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;
    return s.do_sort(input);
}
```

8.1.3 通过人去类型划分工作

当通过任务类型对线程间的任务进行划分时,不应该让线程处于完全隔离的状态。当多个输入数据集需要使用同样的操作序列,可以将序列中的操作分成多个阶段,来让每个线程执行。

这里可以使用cpu的流水先工作方式，对于CPU数据的处理有加好的平均性能

8.2 影响并发代码性能的因素

处理器数量

首先线程是操作系统上的概念，在CPU中是不存在线程这个说法的，所以需要注意的时，在多线程编程，不等同于多核编程，中间操作系统起到了非常重要的作用。之间的调度并不明了。需要谨慎使用。

8.2.2 数据争用与乒乓缓存

当两个线程并发的在不同处理器上执行时，对同意数据进行读取，通常不会出现问题；数据将会拷贝到每个线程的缓存中，可以让两个处理器同事进行处理。但是当有线程对数据进行修改的时候，修改数据需要更新到其它核芯的缓存中取，需要耗费一定的时间。通常会让工作中的CPU进行等待，直到缓存中的数据得到更新。

高竞争(high contention):一个处理器准备更新这个值,另一个处理器正在修改这个值,所以该处理器就不得不等,待第二个处理器更新完成,并且完成更新传递时,才能执行更新。 **低竞争(low contention)**: 如果处理器很少需要相互等待。

乒乓缓存(cache ping-pong): 数据在每个缓存中传递若干次。

在多线程编程中，互斥量，通常需要另外一个线程将数据进行转移，保证处理器之间的互斥性。当线程进行完修改后，其它线程对互斥量进行修改，并对线程进行解锁，再将互斥数据传递到下一个需要互斥量的线程上去。这个过程就是互斥量的获取和释放。

当多个线程高竞争访问时，会造成大量的资源浪费

注意：

- 互斥量的竞争通常不同于原子操作的竞争,最简单的原因是,互斥量通常使用操作系统级别的序列化线程,而非处理器级别的。因此不会影响操作系统中的其它线程，但是会影响本程序的线程。
- 尽量避免乒乓现象，减少两个线程对同一个内存位置的竞争。

缓存行：由处理器cache大小，决定的一次读取内存块的一行，内存块通常大小为32或64字节。在内存中称为缓存行(cache line);

伪共享: 即使给定内存位置被一个线程所访问,可能还是会有乒乓缓存的存在,是因为另一种叫做伪共享(false sharing)的效应。即使数据存储在缓存行中，多个线程对数据中的成员进行访问时，硬件缓存还是会产生乒乓缓存。缓存行是多个线程共享的，但实际并不被多个CPU共享，因此使用伪共享来声明这种方式。

伪共享发生的原因: 某个线程所要访问的数据过于接近另一线程的数据,另一个是与数据布局相关的陷阱会直接影响单线程的性能。 **避免伪共享**：避免伪共享的方法是，实现数据分离，努力让不同线程访问不同缓存行。

8.2.4 紧凑的数据

当CPU中线程的关键数据分散在内存中时，会增加内存访问的次数和内存的延迟。因此尽量紧凑的内存设计会降低延迟。

当处理器切换线程时，对不同内存上的数据进行重新加载(当不同线程使用的数据跨越了多个缓存行时)，而非对缓存中的数据保持原样(当线程中的数据都在同一缓存行时)。

当线程数量对一二内核处理器数量，操作系统可能也会选择将一个线程更换芯核，缓存行从一个内核上，转移到另外一个内核上；这样对性能损害比较大。

8.2.5 超额认购和频繁的任务切换

当有超级多的线程准备运行时(非等待状态),任务切换问题就会频繁发生。这个问题我们之前也接触过:超额认购。

8.3 为多线程性能设计数据结构

多线程性能设计考虑因素：

- 竞争

- 伪共享
- 数据距离

8.3.1 为复杂操作划分数组元素

这里主要探究的是矩阵的乘法问题，比较建议的是将矩阵进行分块来，进行计算

8.3.2 其它数据结构中的数据访问模式

当使用的互斥量和数据项在内存中很接近，对于一个需要获取互斥量的线程来说，比较理想；所需要的数据可能早就存入处理器的缓存中了；但是当其他线程尝试锁住互斥量时，线程就能对对应的数据进行访问。对于相同位置的操作都需要先获取互斥量，如果互斥量已锁，那就会调用系统内核。而原子的互斥量操作(“读，写，改”)，可能会让数据存储在缓存中，让线程获取的互斥量变得毫无作用。当互斥量共享同一缓存行时，其中存储的是线程已使用的数据，这时拥有互斥量的线程会遭受到性能打击，因为其他线程也在尝试锁住互斥量。

8.4 设计并发代码的注意事项

注意代码在物理硬件改变时的可扩展性，避免因为物理硬件的改变，造成代码错误。

8.4.1 并行算法中的异常安全

在串行算法中抛出一个异常，算法只需要考虑其本身的处理，多线程中需要考虑到多个线程之间的相互影响。

之前实现的线程安全的求和函数在执行线程创建时并不安全。因此在此基础之上改良线程安全函数。

```
class join_threads
{
    std::vector<std::thread>& threads;
public:
    explicit join_threads(std::vector<std::thread>& threads_):threads(threads_){}
    ~join_threads()
    {
        for(unsigned long i=0;i<threads.size();++i)
        {
            if(threads[i].joinable())
                threads[i].join();
        }
    }
};



template<typename iterator ,typename T>
struct accumulate_block
{
    //构造操作

    T operator()(Iterator first,Iterator last)
    {
        //返回所有数据和

        return std::accumulate(first,last,T());
    }
};
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return init;
    unsigned long const min_pre_thread=25;
    unsigned long const max_thread=(length+min_pre_thread-1)/min_pre_thread;
```

```
    unsigned long const hardware_threads=std::thread::hardware_concurrency();
    unsigned long const num_threads=std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    //分块的大小

    unsigned long const block_size=length/num_threads;
    std::vector<std::future<T> > futures(num_threads-1);
    std::vector<std::thread> threads(num_threads-1);
    //安全线程类

    join_threads joiner(threads);


    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        std::packaged_tack<T(Iterator,Iterator)> task(accumulate_block<Iterator,T>());
        futures[i]=task.get_future();
        threads[i]=std::thread(std::move(task),block_start,block_end);
        block_start=block_end;
    }
    T last_result=accumulate_block()(block_start,last);
    std::for_each(
        thread.begin(),
        thread.end(),
        std::mem_fn(&std::thread::join)
        );

    T result=init;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        result+=futures[i].get();
    }
    result+=last_result;
    return result;

}
```

8.4.2 可扩展性和Amdahl定律

将程序划分为”串行”和”并行”部分。可以使用下面的公式对程序性能的增益进行估计：

$$P = \frac{1}{f_S + \frac{1 - f_S}{N}}$$

其中：

- fs表示串行时间

- P表示性能增益
- N处理器数量

8.4.3 使用多线程隐藏延迟

比起添加线程数量让其对处理器进行充分利用,有时也要在增加线程的同时,确保外部事件被及时的处理,以提高系统的响应能力。

8.4.4 使用并发提高响应能力

通常使用专用的GUI线程来处理这些事件。线程可以通过简单的机制进行通讯，而不是将时间处理代码和任务代码混在一起，GUI线程如下

```
std::thread task_thread;
std::atomic<bool>   task_cancelled(false);
void gui_thread()
{
    while(true)
    {
        event_data  event=get_event();
        if(event.type==quit)
            break;
        process(event);
    }
}
void task()
{
    while(!task_complete()&&!task_cancelled)
    {
        do_next_operation();
    }
    if(task_cancelled)
    {
        perform_cleanup();
    }else{
        post_gui_event(task_complete);
    }
}
void process(event_data const& event)
{
    switch(event.type)
    {
        case start_task:
                task_cancelled=false;
                task_thread=std::thread(task);
                break;
        case stop_task:
                task_cancelled=true;
                task_thread.join();
                break;
        case task_complete:
                task_thread.join();
```

```
                display_results();
                break;
        default:
                //...
        }
}
```

8.5 在实践中设计并发代码

8.5.1 并行实现: `std::for_each`

for_each主要是容器类的内部迭代，因此主要是进行操作时候的存取锁；可以通过使用 `std::packaged_task` 和 `std::future` 机制对线程中的异常进行转移。下面是两种方式实现的 `for_each`

```
//使用 std::packaged_task和std::future

template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return;
    unsigned long const min_pre_thread=25;
    unsigned long const max_threads=(length+min_per_thread-1)/min_pre_thread;
    unsigned long const hardware_threads=std::thread::hardware_concurrency();
    unsigned long const num_threads=std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    unsigned long const block_size=length/num_threads;
    std::vector<std::future<void> > futures(num_threads-1);
    std::vector<std::thread> threads(num_threads-1);
    join_threads joiner(threads);
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        std::packaged_task<void(void)> task(
            [=](){
                //执行相关函数

                std::for_each(block_start,block_end,f);
            }
            );
        futures[i]=task.get_future();
        threads[i]=std::thread(std::move(task));
        block_start=block_end;
    }
    std::for_each(block_start,last,f);
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        futures[i].get();
    }

}
```

```
//使用 std::async实现

template<typename   Iterator,typename   Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return;
    unsigned long const min_per_thread=25;
    if(length<(2*min_per_thread))
    {
        std::for_each(first,last,f);
    }else{
        Iterator const mid_point=first+length/2;
        std::future<void> first_half=std::async(
            &parallel_for_each<Iterator,Func>,
            first,mid_point,
            f
            );
            parallel_for_each(mid_point,last,f);
            first_half.get();
        }
}
```

8.5.2 并行实现: std::find

find需要在找到时，中断其它线程，可以使用一个原子变量作为标示，可以使用 `std::packaged_task` 或者 `std::promise` 对异常和最终值进行设置;现在使用 `std::promise` 的方法如下：

```
template<typename Iterator,typename MatchType>
//并行查找函数

Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
    struct find_element
    {
        //重载操作符

        void operator()(
            Iterator begin,
            Iterator end,
            MatchType match,
            std::promise<Iterator>* result,
            std::atomic<bool>*  done_flag
            )
        {
            try
            {
                //循环查找是否相等

                for(;(begin!=end)&&!done_flag->load();++begin)
                {
                    if(*begin==match)
                    {
                        result->set_value(begin);
                        done_flag->store(true);
                        return;
                    }
                }
            }catch(...)
            {
                try
                {
                    //输出错误信息

                    result->set_exception(std::current_exception());
                    done_flag->store(true);
```

```
                }catch(...){}
            }
        }
    };
    unsigned long const length=std::distance(first,last);
    if(!length)
        return  last;
    unsigned long const min_per_thread=25;
    unsigned long const max_threads=(length+min_per_thread-1)/min_per_thread;
    unsigned long const hardware_threads=std::thread::hardware_concurrency();
    unsigned long const num_threads=std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    //每个线程数据块的大小

    unsigned long const block_size=length/num_threads;
    //result结果

    std::promise<Iterator> result;
    //是否查找到的标志位

    std::atomic<bool> done_flag(false);
    //线程vector

    std::vector<std::thread> threads(num_threads-1);
    {
        //添加和启动线程

        join_threads joiner(threads);
        //迭代创建线程

        Iterator block_start=first;
        for(unsigned long i=0;i<(num_threads-1);++i)
        {
            Iterator block_end=block_start;
            std::advance(block_end,block_size);
            threads[i]=std::thread(
                find_element(),
                block_end,
                match,
                &result,
```

```
                &done_flag
                );
            block_start=block_end;
        }
        if(!done_flag.load())
        {
            return last;
        }
        return result.get_future().get();
    }
```

使用 `std::async` 实现的并行find算法

```
template<typename Iterator,typename MatchType>
Iterator parallel_find_impl(Iterator first,Iterator
last,MatchType match,ne)
{
    try
    {
        unsigned long const length=std::distance(first,last);
        unsigned long const min_per_thread=25;
        //小于最小线程数量的两倍直接查找

        if(length<(2*min_per_thread))
        {
            for(;(first!=last)&&!done.load();++first)
            {
                if(*first==match)
                {
                    done=true;
                    return  first;
                }
            }
            return last;
        }else{
            //中间部分的迭代器

            Iterator const mid_point=first+(length/2);
            //获取中间到最后位置的异步执行的结果

            std::future<Iterator> async_result=std::async(&parallel_find_impl<Iterator,MatchType>,mid_point,last,match,std::ref(don
            //直接获取当前查找的结果

            Iterator const direct_result=parallel_find_impl(first,mid_point,match,done);
            //查找结果是否为中间指针

            return (direct_result==mid_point)?async_result.get():direct_result;
        }catch(...)
        {
            done=true;
            throw;
```

```
        }
    }
    //查找函数

    template<typename Iterator,typename MatchType>
    Iterator parallel_find(Iterator first,Iterator last,MatchType match)
    {
        std::atomic<bool> done(false);
        return parallel_find_impl(first,last,match,done);
    }
```

8.5.3 并行实现： `std::partial_sum`

使用划分的方式实现并行的计算部分和

```
template<typename   Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
    //迭代器类型

    typedef tycodepename Iterator::value_type value_type;
    //定义处理单元类

    struct  process_chunk
        {
            //()操作,主要用于构造函数
            void operator()(
                Iterator begin,
                Iterator last,
                std::future<value_type>* previous_end_value,
                std::promise<value_type>*end_value
                )
            {
                //尝试工作
                try
                {
                    //将end迭代器指向last

                    Iterator end=last;
                    //移动迭代器指针

                    ++end;
                    //对数据进行求和，并将结果存入begin中

                    std::partial_sum(begin,end,begin);
                    //如果预期结果存在

                    if(previous_end_value)
                    {
                        //获取结果
                        value_type& addend=previous_end_value->get();
                        //last值添加addend

                        *last+=addend;
```

```
                    //检查end_value是否为空

                    if(end_value)
                    {
                        //设置值

                        end_value->set_value(*last);
                    }
                    //便利迭代器，将每个值添加addend,即每个值添加前一组的期望值

                    std::for_each(begin,last,[addend](value_type& item){
                        item+=addend;
                    });
                    //如果预期结果值不存在，检查end_value是否存在

                }else if(end_value)
                {
                    //存在直接设置为期望值

                    end_value->set_value(*last);
                }
            }catch(...)
            {
                if(end_value)
                {
                    end_value->set_exception(std::current_exception());
                }else{
                    throw;
                }
            }
        }
    };

    unsigned long const length=std::distance(first,last);
    if(!length)
        return last;
    //最小分块线程数

    unsigned long const min_per_thread=25;
```

```
        //计算最大线程数

        unsigned long const max_threads=(length+min_per_thread-1)/min_per_thread;
        //当前线程允许的最大线程数目

        unsigned long const hardware_threads=std::thread::hardware_concurrency();
        //实际线程数目

        unsigned long const num_threads=std::min(hardware_threads!=0?
hardware_threads:2,max_threads);
        //每个线程块的大小

        unsigned long const block_size=length/num_threads;
        //迭代器数据类型

        typedef typename Iterator::value_type value_type;
        //创建线程vector

        std::vector<std::thread> threads(num_threads-1);
        //创建对应的promise,即最终结果

        std::vector<std::promise<value_type> > end_values(num_threads-1);
        //创建期望

        std::vector<std::future<value_type> > previous_end_values;
        //设置期望大小

        previous_end_values.reserve(num_threads-1);
        //创建添加线程

        join_threads joiner(threads);
        //将block中的开始指针指向first

        Iterator block_start=first;
        //开始构造对应线程

        for(unsigned long i=0;i<(num_threads-1);++i){
            //将尾迭代器指向block start
```

```
            Iterator block_last=block_start;
            //将block_last更新迭代器步长为block_size

            std::advance(block_last,block_size-1);
            //创建线程并，输入对应参数

            threads[i]=std::thread(
                process_chunk(),
                block_start,
                block_last,
                (i!=0)?&previous_end_values[i-1]:0,
                &end_values[i]
                );
            //移动block指针

            block_start=block_last;
            ++block_start;
            //将最后的预计值放入end_values

            previous_end_values.push_back(end_values[i].get_future());
        }
        //最后将指针指向分组后的最后一组

        Iterator final_element=block_start;
        //移动尾指针到末尾

        std::advance(final_element,std::distance(block_start,last)-1);
        //计算剩余的值的和

        process_chunk()(
            block_start,
            final_element,
            (num_threads>1)?&previous_end_values.back():0,
            0);
}
```

实现以2的幂级数为距离部分和算法

将数据进行分离，并实现SIMD，将中间的处理结果传递到下一个结果中去。

简单的栅栏类实现

```
class barrier
{
    unsigned const count;
    //空值

    std::atomic<unsigned> spaces;
    std::atomic<unsigned> generation;
public:
    explicit barrier(unsigned count_):count_(count_),spaces(count),generation(0){}
    void wait()
    {
        //更新当前线程的generation

        unsigned const my_generation=generation;
        //当space为0d的时候，重置space,添加gengeration

        if(!--spaces)
        {
            spaces=count;
            ++generation;
        }else{
            //当space>0 时

            //检查是否相同
            while(generation==my_generation) {
                //当没有改变，即不存在++generation,等待一段时间

                std::this_thread::yield();
            }
        }
    }
};

//总体而言实现了栅栏的核心，主要是使用所有进行等待，当栅栏满足之后，再同一开始工作，count是栅栏管控的线程总数
```

上面的栅栏还是略显简陋，因此需要进一步改进

```
struct barrier
{
    //线程总数统计

    std::atomic<unsigned> count;
    //空余总数统计

    std::atomic<unsigned> spaces;
    //栅栏执行相关次数统计

    std::atomic<unsigned> generation;
    barrier(unsigned count_):count(count_),spaces(count_),generation(0)
    {}
    //wait相关函数

    void wait()
    {
        unsigned const gen=generation.load();
        if(!--spaces)
        {
            spaces=count.load();
            ++generation;
        }else{
            //没有到达条件，等待一会儿

            while(generation.load()==gen)
            {
                std::this_thread::yield();
            }
        }
    }
    //执行等待操作

    void done_waiting()
    {
        --count;
        if(!--spaces)
        {
```

```
            spaces=count.load();
            ++generation;
        }
    }
};


//下面是栅栏的并行计算

template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
    typedef typename Iterator::value_type value_type;
    //处理元素类，主要是来运行一组线程

    struct process_element
    {
        void operator()(
            Iterator first,
            Iterator last,
            sstd::vector<value_type>& buffer,
            unsigned i,
            barrier& b
            )
        {
            //获取尾部元素

            value_type& ith_element=*(first+i);
            //是否更新源

            bool update_source=false;
            for(unsigned step=0,stride=1;stride<=i;++step,stride*=2)
            {
                //step为偶数则返回buffer[i],否则返回当前元素
                //主要是从原始数据或者缓存中添加元素

                value_type const& source=(step%2)?buffer[i]:ith_element;

                value_type& dest=(step%2)?ith_element:buffer[i];
```

```
            value_type const& addend=(step%2)?buffer[i-stride]:*(first+i-stride);
            //将计算后的值，添加到缓存

            dest=source+addend;
            update_source=!(step%2);
            //执行栅栏等待同步

            b.wait();
        }
        if(update_source)
        {
            ith_element=buffer[i];
        }
        //开始等待同步，结束本次新城

        b.done_waiting();
    }
};
unsigned long const length=std::distance(first,last);
if(length<=1)
    return;
//创建缓冲向量

std::vector<value_type> buffer(length);
//创建栅栏

barrier b(length);
//创建线程

std::vector<std::thread> thread(length-1);
join_threads joiner(threads);
//更新线程数

Iterator block_start=first;
//遍历，创建线程

for(unsigned long i=0;i<(length-1);++i)
{
```

```
        threads[i]=std::thread(
            process_element(),
            first,
            last,
            std::ref(buffer),
            i,
            std::ref(b)
            );
    }
    //最后处理剩下的元素
    process_element()(first,last,buffer,length-1,b);
}
```

第9章 高级线程池

关于线程池在之前的文章中有过介绍，因此不再做过多说明

可等待任务的线程池

```
class function_wrapper
{
    struct impl_base
    {
        virtual void call()=0;
        virtual ~impl_base(){}
    };
    std::unique_ptr<impl_base>  impl;
    template<typename   F>
    struct impl_type:impl_base
    {
        F f;
        impl_type(F&& f_):f(std::move(f_)){}
        void call(){f();}
    };
public:
    template<typename F>
    function_wrapper(F&& f):impl(new impl_type<F>(std::move(f))){}
    void operator()(){impl->call();}
    function_wrapper()=default;
    function_wrapper(function_wrapper&& other):impl(std::move(other.impl)){}
    function_wrapper& operator=(function_wrapper&& other)
    {
        impl=std::move(other.impl);
        return *this;
    }
    function_wrapper(const function_wrapper&)=delete;
    function_wrapper(function_wrapper&)=delete;
    function_wrapper& operator=(const function_wrapper&)=delete;
};
class thread_pool
{
    thread_safe_queue<function_wrapper> work_queue; //使用function_wrapper,而非使用std::function
    void worker_thread()
    {
        while(!done)
        {
            function_wrapper task;
```

```
            if(work_queue.try_pop(task))
            {
                task();
            }else{
                std::this_thread::yield();
            }
        }
    }
public:
    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type> submit(FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type result_type;
        std::packaged_task<result_type()> task(std::move(f));
        std::future<result_type> res(task.get_future());
        work_queue.push(std::move(task));
        return  res;
    }
};
```

使用线程池求和

```
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return init;
    unsigned long const block_size=25;
    unsigned long const num_blocks=(length+block_size-1)/block_size;
    std::vector<std::future<T>  > futures(num_blocks-1);
    thread_pool pool;
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        Iteratorblock_end=block_start;
        std::advance(block_end,block_size);
        futures[i]=pool.submit(accumulate_block<Iterator,T>());
        block_start=block_end;
    }
    T last_result=accumulate_block<Iterator,T>()(block_start,last);
    T result=init;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        result+=futures[i].get();
    }
    result+=last_result;
    return result;
}
```

基于线程池的快速排序实现

```
template<typename T>
struct sorter
{
    thread_pool;
    std::list<T> do_sort(std::list<T>& chunk_data)
    {
        if(chunk_data.empty())
        {
            return chunk_data;
        }
        std::list<T> result;
        //分割数据

        result.splice(result.begin(),chunk_data,chunk_data.begin());
        T const& partition_val=*result.begin();
        //分割数组，并返回关键迭代指针

        typename std::list<T>::iterator divide_point=std::partition(
            chunk_data.begin(),
            chunk_data.end(),
            [&](T const& val){return val<partition_val;}
            );
        //创建较小部分的数据块

        std::list<T> new_lower_chunk;
        //赋值初始化

        new_lower_chunk.splice(
            new_lower_chunk.end(),
            chunk_data,
            chunk_data.begin(),
            divide_point
            );
        std::list<T> new_higher(do_sort(chunk_data));
        //将高部数据拷贝到result

        result.splice(result.end(),new_higher);
```

```
        while(!new_lower.wait_for(std::chrono::seconds(0))==std::future_status::timeout)
        {
            pool.run_pending_task();
        }
        result.splice(result.begin(),new_lower.get());
        return result;
    }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;
    return s.do_sort(input);
}
```

9.1.5 窃取任务

为了让没有任务的线程能从其他线程的任务队列中获取任务,就需要本地任务列表可以进行访问,这样才能让run_pending_tasks()窃取任务。需要每个线程在线程池队列上进行注册,或由线程池指定一个线程。同样,还需要保证数据队列中的任务适当的被同步和保护,这样队列的不变量就不会被破坏。

```
class work_stealing_queue
{
private:
    typedef function_wrapper data_type;
    //数据队列

    std::deque<data_type> the_queue;
    mutable std::mutex  the_mutex;
public:
    work_stealing_queue(){}
    work_stealing_queue(const work_stealing_queue& other)=delete;
    work_stealing_queue& operator=(const work_stealing_queue& other)=delete;
    void push(data_type  data)
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        the_queue.push_front(std::move(data));
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        return  the_queue.empty();
    }
    //安全的取出数据

    bool try_pop(data_type& res)
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }
        res=std::move(the_queue.front());
        the_queue.pop_front();
        return  true;
    }
    //对队列后端进行操作

    bool try_steal(data_type& res)
    {
```

```
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }
        res=std::move(the_queue.back());
        the_queue.pop_back();
        return  true;
    }
};
```

使用任务窃取的线程池

```
class thread_pool
{
    typedef function_wrapper task_type;
    std::atomic_bool done;
    thread_safe_queue<task_type> pool_work_queue;
    std::vector<std::unique_ptr<work_stealing_queue> > queues;
    std::vector<std::thread> threads;
    join_threads joiner;
    static thread_local work_stealing_queue* local_work_queue;
    static thread_local unsigned my_index;
    void worker_thread(unsigned my_index_)
    {
        my_index=my_index_;
        local_work_queue=queues[my_index].get();
        while(!done)
        {
            run_pending_task();
        }
    }
    bool pop_task_from_local_queue(task_type& task)
    {
        return local_work_queue && local_work_queue->try_pop(task);
    }
    bool pop_task_from_pool_queue(task_type& task)
    {
        return  pool_work_queue.try_pop(task);
    }
    bool pop_task_from_other_thread_queue(task_type& task)
    {
        for(unsigned  i=0;i<queues.size();++i)
        {
            unsigned const index=(my_index+i+1)%queues.size();
            if(queues[index]->try_steal(task))
            {
                return true;
            }
        }
        return false;
```

```
    }
public:
    thread_pool():done(false),joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency();
        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                queues.push_back(std::unique_ptr<work_stealing_queue>(threads.push_back(std::thread(&thread_pool::worker_thread,thi
            }
        }catch(...)
        {
            done=true;
            throw;
        }
    }
    ~thread_pool()
    {
        done=true;
    }
    template<typename FunctionType>
    std::future<typename std::result_of<FunctionType()>::type> submit(FunctionType f)
    {
        typedef typename std::result_of<FunctionType()>::type result_type;
        std::packaged_task<result_type()> task(f);
        std::future<result_type> res(task.get_future());
        if(local_work_queue)
        {
            local_work_queue->push(std::move(task));
        }else{
            pool_work_queue.push(std::move(task));
        }
        return res;
    }
    void run_pending_task()
    {
        task_type task;
        if(pop_task_from_local_queue(task)||
```

```
            pop_task_from_pool_queue(task)||
            pop_task_from_other_thread_queue(task))
        {
            task();
        }else{
            std::this_thread::yield();
        }
    }
};
```

9.2 线程中断

操作系统中的线程中断和挂起机制，需要使用信号来让未结束线程停止运行。这里需要一种合适的方式让线程主动的停下来,而非让线程戛然而止。

9.2.1 启动和中断线程

线程的中断多需要在线程的原有基础之上，添加线程中断的程序。

std::condition_variable 在interruptible_wait中使用超时

```
class interrupt_flag
{
    //是否中断

    std::atomic<bool> flag;
    //环境变量

    std::condition_variable* thread_cond;
    //清除信号量

    std::mutex set_clear_mutex;
public:
    interrupt_flag():thread_cond(0){}
    void set()
    {
        flag.store(true,std::memory_order_relaxed);
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        //设计环境变量

        if(thread_cond)
        {
            //发射环境信号

            thread_cond->notify_all();
        }
    }
    //查看是否设置

    bool is_set() const
    {
        return flag.load(std::memory_order_relaxed);
    }
    void set_condition_variable(std::condition_variable& cv)
    {
        //信号加锁

        std::lock_guard<std::mutex> lk(set_clear_mutex);
        //更新条件变量
```

```
        thread_cond=&cv;
    }
    //清除环境变量

    void clear_condition_variable()
    {
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        thread_cond=0;
    }
    struct clear_cv_on_destruct
    {
        ~clear_cv_on_destruct()
        {
            this_thread_interrupt_flag.clear_condition_variable();
        }
    };
};
thread_local interrupt_flag this_thread_interrupt_flag;
//检查中断点，通过检查flag值来判断，如果flag为true抛出信号

void interruption_point()
{
    if(this_thread_interrupt_flag.is_set())
    {
        throw thread_interrupted();
    }
}

//中断等待

void interruptible_wait(std::condition_variable& cv,std::unique_lock<std::mutex>& lk)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);
    //临时锁

    interrupt_flag::clear_cv_on_destruct gurad;
    //再次检查
```

```
    cv.wait_for(lk,std::chrono::milliseconds(1));
    interruption_point();
}
//中断等待

template<typename Predicate>
void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk,
                        Predicate pred
                        )
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);
    interrupt_flag::clear_cv_on_destruct gurad;
    while(!this_thread_interrupt_flag.is_set()&&!pred())
    {
        cv.wait_for(lk,std::chrono::milliseconds(1));
    }
    interruption_point();
}
```

为std::condition_variable_any 设计的interruptible_wait

```
class interrupt_flag
{
    std::atomic<bool> flag;
    //环境条件变量

    std::condition_variable* thread_cond;
    std::condition_variable_any* thread_cond_any;
    //访问互斥信号量

    std::mutex set_clear_mutex;
public:
    interrupt_flag():
    thread_cond(0),thread_cond_any(0){}
    void set()
    {
        //更改值

        flag.store(true,std::memory_order_relaxed);
        //加锁

        std::lock_guard<std::mutex> lk(set_clear_mutex);
        //环境变量

        if(thread_cond)
        {
            thread_cond->notify_all();
        }else if(thread_cond_any)
        {
            thread_cond_any->notify_all();
        }
    }
    //等待函数

    template<typename Lockable>
    void wait(std::condition_variable_any& cv,Lockable& lk)
        {
            //传统默认锁
```

```
            struct  custom_lock
            {
                interrupt_flag* self;
                Lockable& lk;

                custom_lock(
                    interrupt_flag* self_,
                    std::condition_variable_any& cond,
                    Lockable& lk_):
                self(self_),
                lk(lk_)
                {
                    self->set_clear_mutex.lock();
                    self->thread_cond_any=&cond;
                }

                void unlock()
                {
                    lk.unlock();
                    self->set_clear_mutex.unlock();
                }

                void lock()
                {
                    std::lock(self->set_clear_mutex,lk);
                }
                ~custom_lock()
                {
                    self->thread_cond_any=0;
                    self->set_clear_mutex.unlock();
                }
            };
            custom_lock cl(this,cv,lk);
            interruption_point();
            cv.wait(cl);
            interruption_point();
        }
};
//中断等待
```

```
template<typename Lockable>
void interruptible_wait(std::condition_variable_any& cv,Lockable& lk)
{
    this_thread_interrupt_flag.wait(cv,lk);
}
```

第10章 多线程程序的测试和调试

10.1 与并发相关的错误类型

- 不必要阻塞：一个线程被阻塞的时候,不能处理任何任务,因为它在等待其他“条件”的达成。即阻塞不是必要的
  - 死锁:相互等待直到永远，无法自己跳出,主要原因是无法检查其它相关变量的变化
  - 活锁:与死锁基本相同但不是线程阻塞等待，而是在循环中持续检查，如:自旋锁。问题可以解决。
  - I/O阻塞或外部输入:当线程被外部输入所阻塞,线程也就不能做其他事情了(即使,等待输入的情况永远不会发生)。
- 条件竞争:
  - 数据竞争：因为未同步访问一块共享内存,将会导致代码产生未定义行为
  - 破坏不变量:主要表现为悬空指针(因为其他线程已经将要访问的数据删除了),随机存储错误(因为局部更新,导致线程读取了不一样的数据),以及双重释放(比如:当两个线程对同一个队列同时执行pop操作,想要删除同一个关联数据),等等。
  - 生命周期问题:线程访问变量时，变量的声明周期已经结束。

10.2 定位并发错误的技术

- 代码审阅——发现潜在的错误，主要考虑的问题
  - 并发访问时,那些数据需要保护?
  - 如何确定访问数据受到了保护?
  - 是否会有多个线程同时访问这段代码?

    - 这个线程获取了哪个互斥量?
    - 其他线程可能获取哪些互斥量?
    - 两个线程间的操作是否有依赖关系?如何满足这种关系?
    - 这个线程加载的数据还是合法数据吗?数据是否被其他线程修改过?
    - 当假设其他线程可以对数据进行修改,这将意味着什么?并且,怎么确保这样的事情不 会发生?
- 通过测试定位并发相关的错误，考虑因素
    - “多线程”是有多少个线程(3个,4个,还是1024个?)
    - 系统中是否有足够的处理器,能让每个线程运行在属于自己的处理器上
    - 测试需要运行在哪种处理器架构上
    - 在测试中如何对“同时”进行合理的安排
- 可测试性设计
    - 每个函数和类的关系都很清楚。
    - 函数短小精悍。
    - 测试用例可以完全控制被测试代码周边的环境。
    - 执行特定操作的代码应该集中测试,而非分布式测试。
    - 需要在完成编写后,考虑如何进行测试。

PREVIOUS
C++ 并发编程笔记(三)
(/2019/07/06/CPLUSPLUS_CONCURRENCY_IN_ACTION_03/)

NEXT
STL 源码剖笔记(一)
(/2019/07/06/CPLUSPLUS_ANNOTATED_STL_SOURCES_01/)

Login with GitHub

FEATURED TAGS (/tags/)

C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B) C/C++ (/tags/#C/C++)
后台开发 (/tags/#%E5%90%8E%E5%8F%B0%E5%BC%80%E5%8F%91) C (/tags/#C) 网络编程 (/tags/#%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B)
STL源码解析 (/tags/#STL%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90) Linux (/tags/#Linux)
操作系统 (/tags/#%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 程序设计 (/tags/#%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1)
优化 (/tags/#%E4%BC%98%E5%8C%96) UML (/tags/#UML) UNIX (/tags/#UNIX) 学习笔记 (/tags/#%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0)
面试 (/tags/#%E9%9D%A2%E8%AF%95) Java (/tags/#Java) 读书笔记 (/tags/#%E8%AF%BB%E4%B9%A6%E7%AC%94%E8%AE%B0) go (/tags/#go)
阅读笔记 (/tags/#%E9%98%85%E8%AF%BB%E7%AC%94%E8%AE%B0)

FRIENDS

WY (http://zhengwuyang.com) 简书·JF (http://www.jianshu.com/u/e71990ada2fd) Apple (https://apple.com)
Apple Developer (https://developer.apple.com/)

(https://www.facebook.com/wangpengcheng) (https://github.com/wangpengcheng)

Theme on GitHub (https://github.com/wangpengcheng/wangpengcheng.github.io.git) | Star 12