
# 并发有序链表

2019/3/13

DaSE
Data Science
&Engineering


# 准备工作

```
find_package(Threads)
target_link_libraries (${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT})
```

DaSE
Data Science
&Engineering

# 并发

「一个人」同时干不同的事情；

「多个人」分别同时干不同的事情

# 目的

设计多线程的程序目的主要有两个：

- **充分利用多核CPU的性能**（利用多核心的计算能力以及让计算和IO重叠来降低RT并提升吞吐量）

- **简化程序逻辑**（即把单线程状态机的逻辑拆分成多个线程彼此同步，这么做虽然不见得能提升代码性能，但是可以简化代码逻辑）。

# 执行流程

# 头文件

- <atomic>：该头文主要声明了两个类, std::atomic 和 std::atomic_flag，另外还声明了一套 C 风格的原子类型和与 C 兼容的原子操作的函数。
- <thread>：该头文件主要声明了 std::thread 类，另外 std::this_thread 命名空间也在该头文件中。
- <mutex>：该头文件主要声明了与互斥量(mutex)相关的类，包括 std::mutex 系列类，std::lock_guard, std::unique_lock, 以及其他的类型和函数。
- <condition_variable>：该头文件主要声明了与条件变量相关的类，包括 std::condition_variable 和 std::condition_variable_any。
- <future>：该头文件主要声明了 std::promise, std::package_task 两个 Provider 类，以及 std::future 和 std::shared_future 两个 Future 类，另外还有一些与之相关的类型和函数，std::async() 函数就声明在此头文件中。

# 例子1

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

void HelloWorld() {
    std::cout << "Hello multithread!" << std::endl;
    return;
}

int main() {
    std::thread t(HelloWorld);
    t.join();
    return 0;
}
```


# 声明方式

```
std::thread t1; // t1 is not a thread
std::thread t2(f1, n + 1); // pass by value
std::thread t3(f2, std::ref(n)); // pass by reference
std::thread t4(std::move(t3)); // t4 is now running f2(). t3 is
no longer a thread
```

DaSE Data Science & Engineering


# Join vs detach

C++11有两种方式来等待线程结束

- detach方式，启动的线程自主在后台运行，当前的代码继续往下执行，不等待新线程结束。
- join方式，等待启动的线程完成，才会继续往下执行。

DaSE
Data Science
&Engineering

# 例子2

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

using namespace std;

void output(int i)
{
cout << "This thread number is " << i << endl;
cout << "Thread " << i << " has finished" << endl;
}

int main()
{
for (int i = 0; i < 5; i++)
{
thread t(output, i);
t.detach(); //t.join();
}
getchar();
return 0;
}
```


# 数据保护

通常来说，避免恶性数据竞争有几个思路。

- 保护数据结构，确保在数据结构更新过程中，其不变量被破坏的中间状态只有一个线程能够看到。
- 修改数据结构的实现，确保任何对数据结构的更新，在外界看来不变量都是成立的。
- 将所有对数据结构的修改，都交给第三方（服务器）串行执行。

# 互斥量

访问某个数据结构时的基本逻辑：

- 首先检查互斥量。若互斥量被锁住，则等待，直到互斥量解锁。
- 若互斥量没有被锁住，则锁住互斥量，而后更新数据，再解锁。

问题：

- 首先，互斥量起作用是有前提的。
- 所有可能引发数据竞争的数据结构都被保护起来了；
- 所有可能引发数据竞争的操作，都正确地使用了互斥量。
- 其次，互斥量可能引发所谓的「死锁」问题。
- 最后，若互斥量过多或过少地保护了数据，都可能出现问题。

# 例子3

```
void safe_increment()
{
        g_i_mutex.lock();
        ++g_i;
        std::cout << std::this_thread::get_id()  << ": " << g_i << '\n';
        g_i_mutex.unlock();
}
```

# Std::mutex

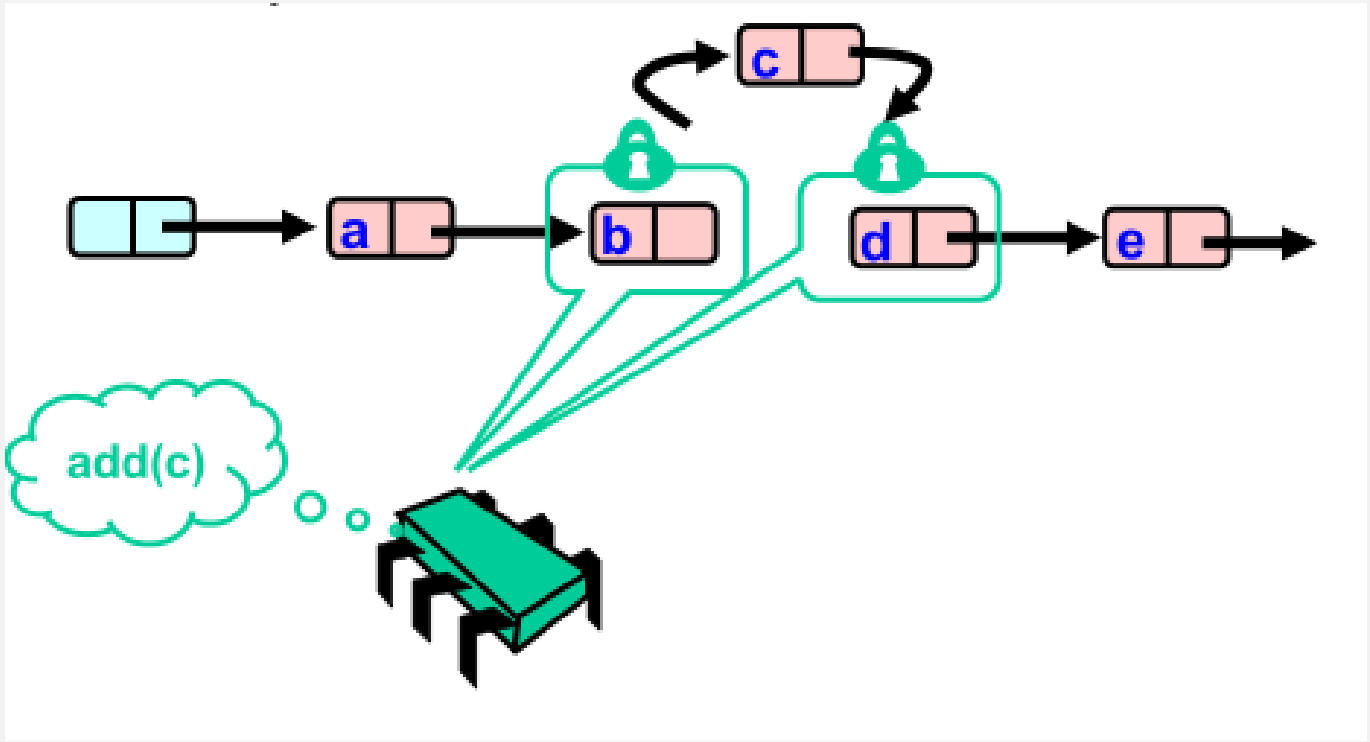
互斥对象的主要操作有两个加锁(lock)和释放锁(unlock)。当一个线程对互斥对象进行lock操作并成功获得这个互斥对象的所有权，在此线程对此对象unlock前，其他线程对这个互斥对象的lock操作都会被阻塞。

| 类模板 | 描述 |
| --- | --- |
| std::lock_guard | 严格基于作用域(scope-based)的锁管理类模板，构造时是否加锁是可选的(不加锁时假定当前线程已经获得锁的所有权)，析构时自动释放锁，所有权不可转移，对象生存期内不允许手动加锁和释放锁。 |
| std::unique_lock | 更加灵活的锁管理类模板，构造时是否加锁是可选的，在对象析构时如果持有锁会自动释放锁，所有权可以转移。对象生命期内允许手动加锁和释放锁。 |
| std::shared_lock(C++14) | 用于管理可转移和共享所有权的互斥对象。 |

# 例子4

```
//死锁
void sharePrint(std::string id, int value) {
                    std::lock_guard<std::mutex> locker(mu);
                    std::lock_guard<std::mutex> locker2(mu2);
                    std::cout << id << value << std::endl;
          }
void sharePrint2(std::string id, int value) {
                    std::lock_guard<std::mutex> locker2(mu2);
                    std::lock_guard<std::mutex> locker(mu);
                    std::cout << id << value << std::endl;
          }
```

# Add-思路



# Add-思路

```
27  void Add(Node* head, Node* node) {
28    if (node == nullptr) {
29        return;
30    }
31    printf("Insert %d Start\n", node->key_);
32    Node* prev = head;
33    Node* current = head->next_;
34    while (current != nullptr) {
35        prev->mutex_.lock();
36        current->mutex_.lock();
37        if (node->key_ < current->key_) {
38            prev->next_ = node;
39            node->next_ = current;
40            printf("Insert %d End\n", node->key_);
41            current->mutex_.unlock();
42            prev->mutex_.unlock();
43            return;
44        } else {
45            prev->mutex_.unlock();
46            current->mutex_.unlock();
47            prev = current;
48            current = current->next_;
49        };
50    }
51    prev->mutex_.lock();
52    printf("Insert %d End\n", node->key_);
53    //prev->mutex_.unlock();
54    prev->next_ = node;
55    prev->mutex_.unlock();
56
57  }
```

# Task

实现一个并发有序链表：假定有序链表不允许重复元素

方法：

- add(x) 增加一个元素
- remove(x) 删除一个元素
- contains(x) 查找一个元素

每个节点包括：

- Key
- Item(value)
- 一个指向后面节点的指针 next