Хитрости мультипоточности

Максим Лысков

Минск, 2016

Знакомимся?

Максим, старший инженер-разработчик в компании EPAM Systems.

В настоящее время участвую в разработке системы иерархического хранения и репликации данных.

Хитрость №1

Начиная с C++11, у нас есть механизм параллельного выполнения задач

Хитрость №2

std::packaged_task<T(U ...Args)>

- Содержит в себе callable объект
- Может быть вызвана асинхронно
- Можно получить результат через std::future

```
  std::packaged_task<int(int,int)> task([](int a, int b) {
      return a * b;
   });
  _std::future<int> result = task.get_future();
   // any other code
   task(2, 9);

  _std::cout << "task result: " << result.get() << '\n';
```

Методы std::packaged_task

- valid()
- get_future() // must be called once
- operator()(Args … args) // execute and fill the future
- make_ready_at_thread_exit(Args … args) // execute and fill on thread exit
- reset() // clear previous call result, necessary to get_future again
- swap()

Хитрость №3

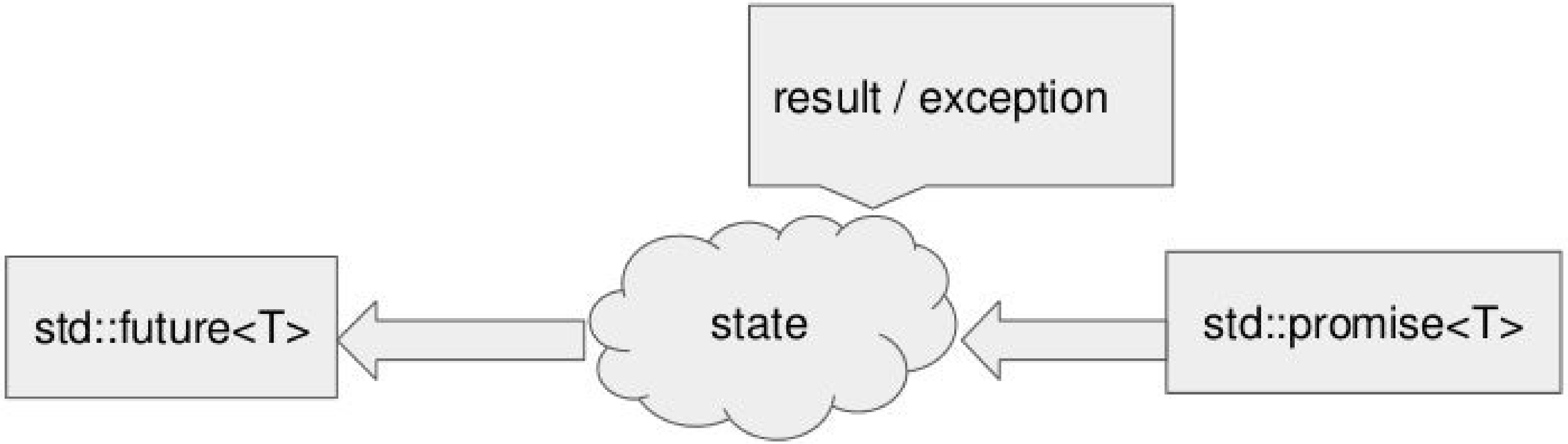
std::future std::async(F, Args … args)
std::future std::async(policy, F, Args … args)

Policy может быть:

- deferred - отложенное выполнение
- async - выполнение в новом потоке

Хитрость №4

- std::promise<T>
- std::future<T> // get result once
- std::shared_future<T> // get result many times



Методы std::promise<T>

- get_future
- set_value
- set_value_at_thread_exit
- set_exception // std::current_exception
- set_exception_at_thread_exit

Методы std::future<T>

- share() // create shared_future
- get() // result or exception
- valid()
- wait() // blocks until result is ready
- wait_for() / wait_until() // deferred/ready/timeout

Хитрость №5

std::thread(F, Args … args)

Методы:

- detach()
- join()
- joinable()
- get_id()
- hardware_concurrency()
- native_handle()

Утилиты текущего потока

- std::this_thread::get_id
- std::this_thread::yield
- std::this_thread::sleep_for / sleep_until

Do and dont

- Не вызывать join() для !joinable() потоков
- Не забывать вызывать join() перед деструктором, если joinable()

Передача параметров в потоки

- copy по умолчанию
- move если есть
- std::ref/std::cref если надо передать ссылки

Хитрость №6

Синхронизация потоков

- mutex // lock/unlock/try_lock
- timed_mutex // try_lock_for/try_lock_until
- recursive_mutex
- recursive_timed_mutex
- shared_mutex / shared_timed_mutex

RAII-обертки для mutex

- std::lock_guard<M> // just RAII, can adopt lock
- std::uniqie_lock<M> // defer/adopt/try_to_lock
- std::lock<M>/std::try_lock<M> // lock several std::uniqie_locks without deadlocks

- std::lock_guard быстрее std::unique_lock

Хитрость №8

std::once_flag / std::call_once

```
int f() {
    std::cout << "Called!\n";
    return 0;
}

int main() {
    std::once_flag callGuard;
    std::call_once(callGuard, f);
    std::call_once(callGuard, f);
}
```

```
Called!
```

Хитрость №9

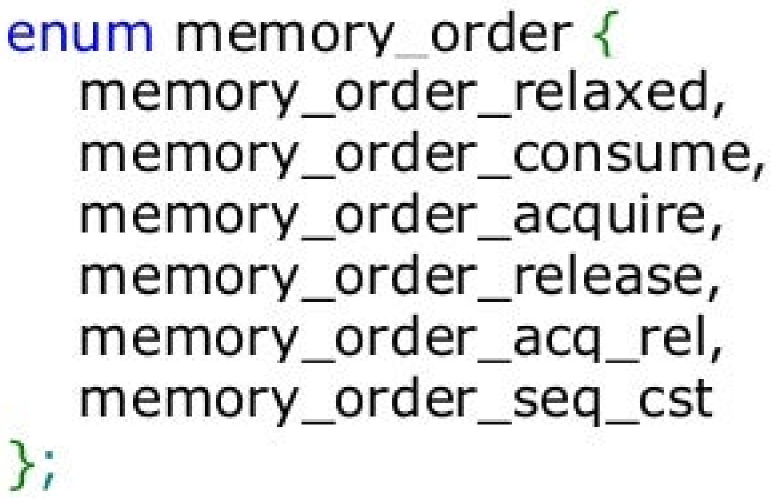
Модель памяти C++

```
struct S
{
    char a; // location 1
    int b:5; // location 2
    unsigned c:11; // still location 2
    unsigned: 0; // location delimeter
    unsigned d:8; // location 3
    struct {int ee:8; } e; // location 4
}
```

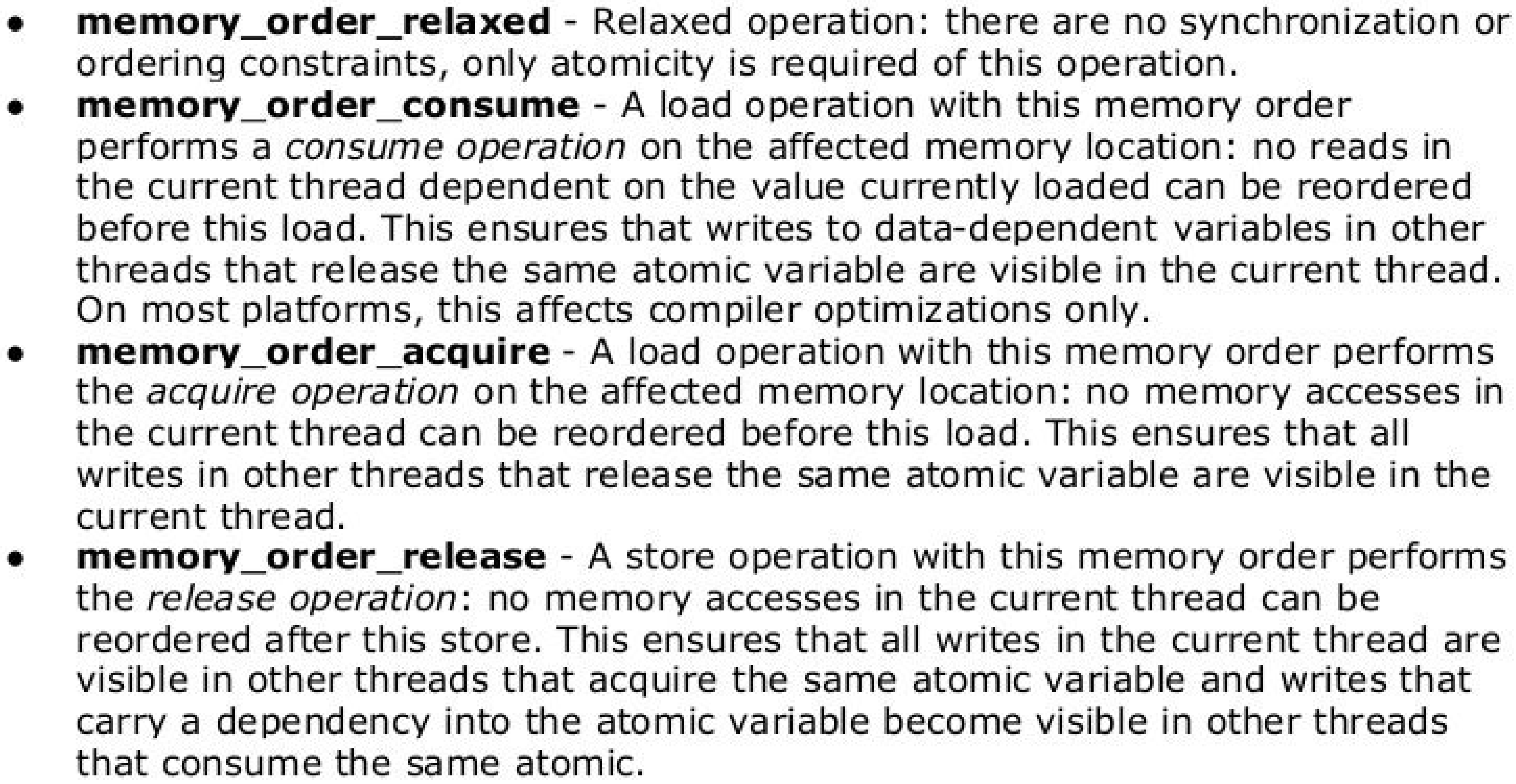
Стандарт говорит, что доступ к разным локациям из разных потоков не пересекается друг с другом.

Хитрость №10*

Memory order в C++



```
enum memory_order {
   memory_order_relaxed,
   memory_order_consume,
   memory_order_acquire,
   memory_order_release,
   memory_order_acq_rel,
   memory_order_seq_cst
};
```



- **memory_order_relaxed** - Relaxed operation: there are no synchronization or ordering constraints, only atomicity is required of this operation.
- **memory_order_consume** - A load operation with this memory order performs a *consume operation* on the affected memory location: no reads in the current thread dependent on the value currently loaded can be reordered before this load. This ensures that writes to data-dependent variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this affects compiler optimizations only.
- **memory_order_acquire** - A load operation with this memory order performs the *acquire operation* on the affected memory location: no memory accesses in the current thread can be reordered before this load. This ensures that all writes in other threads that release the same atomic variable are visible in the current thread.
- **memory_order_release** - A store operation with this memory order performs the *release operation*: no memory accesses in the current thread can be reordered after this store. This ensures that all writes in the current thread are visible in other threads that acquire the same atomic variable and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic.

- **memory_order_acq_rel** - A read-modify-write operation with this memory order is both an *acquire operation* and a*release operation*. No memory accesses in the current thread can be reordered before this load, and no memory accesses in the current thread can be reordered after this store. It is ensured that all writes in other threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable.
- **memory_order_seq_cst** - Any operation with this memory order is both an *acquire operation* and a *release operation*, plus a single total order exists in which all threads observe all modifications (see below) in the same order.

- Все это нужно для **atomic<T>**

Валидное выражение в C++

```
[](){}()
```

?

Спасибо!