Средства и системы параллельного программирования

Семинар #3. Основы многопоточности в C++

Как создавать поток в C++?

Для представления потока выполнения в C++ 11 появился специальный класс std::thread (нужно сделать **#include <thread>**)

```
thread() noexcept;                                          (1) (since C++11)
thread( thread&& other ) noexcept;                          (2) (since C++11)
template< class F, class... Args >
explicit thread( F&& f, Args&&... args );                   (3) (since C++11)
thread( const thread& ) = delete;                           (4) (since C++11)
```

Как создавать поток в C++?

```
thread() noexcept;                                          (1) (since C++11)

thread( thread&& other ) noexcept;                          (2) (since C++11)

template< class F, class... Args >
explicit thread( F&& f, Args&&... args );                   (3) (since C++11)

thread( const thread& ) = delete;                           (4) (since C++11)
```

(1) - создание потока, не представляющего поток управления
(2) - перемещение ресурсов объекта потока other в наш объект
(3) - создание потока, выполняющего функцию f с аргументами args
(4) - копировать другой поток запрещено

Lambda-функция

Представляют собой краткий синтаксис для определения функторов

[] (параметры) { действия }

В квадратных скобках - имена переменных из области видимости **определения** лямбды, которые хотим захватить

Правила проброса параметров такие же, как в обычных функциях. Если параметров нет, () можно не писать

[=] - захват всех переменных по значению

[&] - захват всех переменных по ссылке

[var1, &var2] - разный захват двух переменных

Мьютексы

```
#include <mutex>
```

`constexpr mutex() noexcept;` - конструктор мьютекса

Семантика следующих операций идентична семантике операций для pthread_mutex_t:

```
void lock();

bool try_lock();

void unlock();
```

Функция std::lock

```
template< class Lockable1, class Lockable2, class... LockableN >
void lock( Lockable1& lock1, Lockable2& lock2, LockableN&... lockn );
```

Данная функция использует алгоритмы обхода deadlock() при попытке захвата сразу нескольких мьютексов

Умный захват мьютекса

```
template<class Mutex>
class lock_guard;          (since C++11)
```

Автоматически захватывает мьютекс и управляет разблокировкой - unlock() у мьютекса вызывается в деструкторе lock_guard

Ещё более умный захват мьютекса

```
template< class Mutex >
class unique_lock;
```

Данный класс также предоставляет RAII-интерфейс для мьютекса, однако его семантика шире - он позволяет отпустить мьютекс и вновь его взять

У данного класса есть методы lock(), try_lock(), owns_lock()

Conditional variable

```
#include <condition_variable>
class condition_variable;                    (since C++11)

void wait( std::unique_lock<std::mutex>& lock );
template< class Predicate >
void wait( std::unique_lock<std::mutex>& lock, Predicate pred );
```

pred - условие выхода из цикла, созданного для противодействия spurious wakeups

void notify_one() noexcept;, void notify_all() noexcept; - для сигналов

Timed mutex

class timed_mutex;

мьютекс для взаимодействия с блокировками по времени

Можно обернуть в std::unique_lock и вызывать методы timed_mutex через методы std::unique_lock

Jthread

```
class jthread; (since C++20)
```

Тот же поток, что и std::thread, только

1) автоматически делает join() в деструкторе
2) Хранит разделяемое состояние, через которое можно приказать потоку завершиться

Пользование разделяемым ресурсом

```
{
    lock_guard<mutex> lg{mut};
    if (!res_ptr)
        res_ptr = new Resource();
}
res_ptr->use();  //read-only функция
```

Чем плох такой код?

Double-checked lock

```
if (!res_ptr)
{
    lock_guard<mutex> lg{mut};
    if (!res_ptr)
        res_ptr = new Resource();
}
res_ptr->use();  //read-only функция
```

Call once

```
class once_flag;

template< class Callable, class... Args >

void call_once( std::once_flag& flag, Callable&& f, Args&&... args );
```

С использованием единого глобального flag можно гарантировать, что функция вызовется один раз, даже при условии множественных обращений с потоков

Атомики

`std::atomic_flag` - единственный тип, гарантированно являющийся атомарным на любой архитектуре

`std::atomic_flag lock = ATOMIC_FLAG_INIT;` - инициализация переменной типа

У него есть два метода (C++11):

`clear()` - сбрасывает значение переменной

`bool test_and_set()` - записываем в переменную true и возвращаем старое значение.

Что вообще дают атомики?

Мы “переносим ответственность” за синхронизацию, раньше проводимую мьютексами, на объекты памяти, к которым синхронизация обеспечивается аппаратно.

Там, где это возможно, мы отказываемся от “тяжёлых” мьютексов в пользу более атомарных операций, которые зачастую легковеснее мьютексов.

К атомарным объектам нельзя обращаться напрямую - обычные операции не являются атомарными.

Атомики общего вида

https://cppreference.com/w/cpp/atomic/atomic.html

```
template< class T >
struct atomic;
template< class U >
struct atomic<U*>;
```

С помощью данных операций можно сделать атомарный тип для исходного класса

На класс есть ограничения (он должен быть trivially copyable, ….,) мы это пропустим

Стоит воспринимать atomic<MyType> как тип, имеющий аналогичное MyType внутреннее представление, но имеющий другие, более “строгие” операции

Операции с атомиками:

```
bool is_lock_free() const noexcept;
```

проверка того, реально ли тип является lock-free (можно ли его синхронизировать только за счёт атомарных операций или все операции к нему за собой тянут мьютексы (если ваш тип получился сложным/длинным))

```
void store( T desired, std::memory_order order =
std::memory_order_seq_cst ) noexcept;
```

запись значения desired в атомик

```
T load( std::memory_order order = std::memory_order_seq_cst )
const noexcept;
```

чтение значения из атомика

memory_order оставлять по умолчанию!!

Операции с атомиками (2):

```
bool compare_exchange_strong( T& expected, T desired,
 std::memory_order success, std::memory_order failure ) noexcept;
```

Попытка(!) записать desired значение в атомик, значение которого мы предполагаем равным expected. Если получилось - true, иначе - в поле expected кладётся текущее значение атомика (обновлённое другими потоками, пока наш поток делал свою работу) и выдаём false.

То есть бывает, что мы делаем какие-то действия, а к моменту записи в атомик результаты наших действий (или часть результатов) становятся нерелевантными и как бы "отстают" от других потоков. Поэтому нам надо эти действия повторить и постараться догнать текущее значение атомика.

Все входящие в этот метод операции также являются атомарными

_weak версия аналогична, однако может давать false даже при совпадении expected и значения атомика

Имеет смысл делать эту операцию в цикле. Данные операции называются CAS(-циклами).

Это концептуально (и зачастую производительно) ***лучше*** *спинлоков (не говоря уже о мьютексах), поскольку мы тратим ресурсы не на ожидание входа в критическую секцию, а на потенциально полезные действия.*

Операции с атомиками (3)

Для многих встроенных типов языка введены операции, позволяющие вынести CAS-циклы под детали реализации

```
T fetch_add( T arg, std::memory_order order =
std::memory_order_seq_cst ) noexcept;

T fetch_sub( T arg, std::memory_order order =

std::memory_order_seq_cst ) noexcept;
```

С atomic<int> можно даже делать += или ++ - под капотом он вызовет атомарную операцию

Задание

Реализовать на языке C++ структуру связного списка со следующими условиями:

- Каждый элемент (NodeChain) списка, помимо указателя на следующий элемент, хранит в себе число uint32_t
- Изначально список пустой
- Каждый поток хочет сгенерировать и положить в список K элементов, по одному элементу за раз
- После того, как все потоки произведут описанные действия, главный поток проходится по списку и проверяет корректность заполнения списка (число элементов = K * threads, число уникальных значений uint32_t в элементах)
- После завершения проверки главный поток удаляет список
- В качестве значений, подаваемых в элемент, можно использовать val = K * thread_number + i для $i \in [0, K - 1]$, но можете придумать что-то своё, чтобы проверка была осмысленной
- Для добавления элементов списка необходимо использовать атомарные операции, рассмотренные на занятии. Значение memory_order - по умолчанию
- Убедиться, что проверка в конце программы стабильно (от запуска к запуску) проходит для большого значения K. Попробовать различное число работающих потоков
- *(* - необязательно) Сравнить время заполнения списка с использованием атомарных операций с временем операции, использующей мьютексы для этой же задачи*

Дедлайн: 6.10, 13.10