ПРОГРАММИРОВАНИЕ НА ОСНОВЕ ЗАДАЧ

Скотт Мейерс:

«Предпочитайте программирование на основе задач программированию на основе потоков»

Проблемы подхода на основе потоков:

- Получение формируемого потоком **результата**?
- Обработка сгенерированного потоком **исключения**?
- **Количество запускаемых в системе потоков** – ограниченный ресурс => при превышении – std::system_error
- «**Превышение подписки**» - количество программных потоков (активных и в состоянии готовности) значительно превышает количество аппаратных потоков => частые переключения контекста => время на переключение + перезагрузка кэшей процессора => неэффективно

Большинство перечисленных проблем

- Решается посредством распараллеливания на основе задач =>
- Переложить проблемы программиста на средства, предоставляемые для этих целей стандартной библиотекой

Когда актуально программирование посредством потоков (требуется управление «вручную»):

- Доступ к нижележащим средствам ОС (например, задание приоритетов)
- Выигрыш по эффективности в задачах с фиксированным (заранее известным числом потоков => можно соптимизировать)
- Реализация пулов потоков

Высокоуровневые средства запуска и взаимодействия потоков. Асинхронное программирование. Ожидание одноразовых результатов

#include <future>

STD::ASYNC()

STD::FUTURE

STD::SHARED_FUTURE

Проблема 1: результат работы потоковой функции посредством std::thread?

в качестве одного (нескольких) параметра в потоковую функцию можно отправить адрес, по которому функция «положит» сформированный ею результат

Проблемы:

- при ожидании завершения запущенного потока???
- если detach()???

Проблема 2: если дочерний поток сгенерировал исключение

А обработка требуется в родительском потоке?

Проблема 3: ограниченное количество потоков

- Для каждого процесса
- Для системы в целом

=> Требуется управление потоками вручную!

При исчерпании потоков – std::system_error

Проблема 4 – «превышение подписки»

большое количество запущенных потоков вынуждает ОС все время осуществлять переключение потоков =>

ресурсы на переключение:

- Время на сохранение/восстановление контекста
- Перегрузка кэшей

Задача:

Требуется запуск длительной операции и получение результата. При этом результат понадобится позже (а может быть и вообще не понадобится…)

Варианты:

- вызвать синхронно =>
  ‘+’ - результат гарантированно сформирован после вызова
  ‘- ‘ – время, потраченное на вычисления, можно было бы потратить на выполнение какой-нибудь другой полезной работы, не требующей результата

Проблемы получения результата **detached** потока:

- как узнать о том, что результат сформирован?
- как обеспечить существование объекта, адрес которого передан в поток?

Асинхронное программирование

стиль программирования, при котором :

- «тяжеловесные» задачи исполняются в detached дочерних (фоновых) потоках,
- и результат выполнения которых может быть получен родительским потоком, когда он того пожелает (при условии, что результат доступен)

Основа асинхронного программирования

- - это **разделяемое состояние** (зависит от конкретной реализации – обычно класс с подсчетом ссылок) ,
  доступ к которому осуществляется посредством шаблонов std::future<> и std::promise<>
- и **условные переменные**

Разделяемое состояние (данные):

```
template<class _Ty> class _Associated_state{
  _Atomic_counter_t _Refs;
  …
public:
  _Ty _Result;
  _XSTD exception_ptr _Exception;
  mutex _Mtx;
  condition_variable _Cond;
  bool _Retrieved;
  int _Ready;
  bool _Ready_at_thread_exit;
  bool _Has_stored_result;
  bool _Running;
  …
};
```

Связь <future> с разделяемым состоянием

```
template<class _Ty> class future<_Ty&>
: public _State_manager<_Ty *>
{ …};

template<class _Ty> class _State_manager{// class for managing possibly non-existent associated asynchronous state object
…
private:
_Associated_state<_Ty> *_Assoc_state;
bool _Get_only_once;
};

template<class _Ty> class _Associated_state{// class for managing associated synchronous state
   …//данные для хранения результата и манипуляций разделяемым состоянием
};
```

шаблон функции std::async()

- позволяет **синхронно** или **асинхронно** запустить задачу (потенциально в отдельном потоке)
- передать ей любое количество параметров (аналогично std::thread)
- и получить возвращаемое потоком значение или обработать сгенерированное задачей исключение с помощью шаблона std::future (когда оно понадобится и гарантированно будет сформировано)

Формы async():

template< class Function, class... Args>

std::**future**<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>

async(**Function&& f, Args&&... args**);

template< class Function, class... Args >

std::**future**<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>

async(**std::launch policy, Function&& f,**
Args&&... args);

Специфика **std::launch** :

- По умолчанию std::launch::**deferred |** std::launch::**async** реализация решает сама: запускать новый поток или выполнять синхронно
- std::launch::**deferred** – отложить вызов функции до вызова методов wait() или get() объекта future (*lazy evaluation*)
- std::launch::**async** – запуск функции в отдельном потоке

Важно!

Использование политики

std::launch::**deferred** | std::launch::**async**

- и **thread_local** объектов

комбинируются ПЛОХО!

- а также если для синхронизации в caller и callee используется один и тот же **mutex ?**

Для асинхронного взаимодействия и обмена данными thread support library предоставляет:

- шаблон **future**, который предоставляет доступ к универсальному хранилищу (placeholder)
  - для значения любого типа, которое будет сформировано в будущем
  - или исключения любого типа, которое будет сгенерировано в будущем
- **future**s – средство для однократного
  - получения возвращаемых потоками значений
  - и обработки исключений при асинхронном выполнении

Будущие результаты (future)

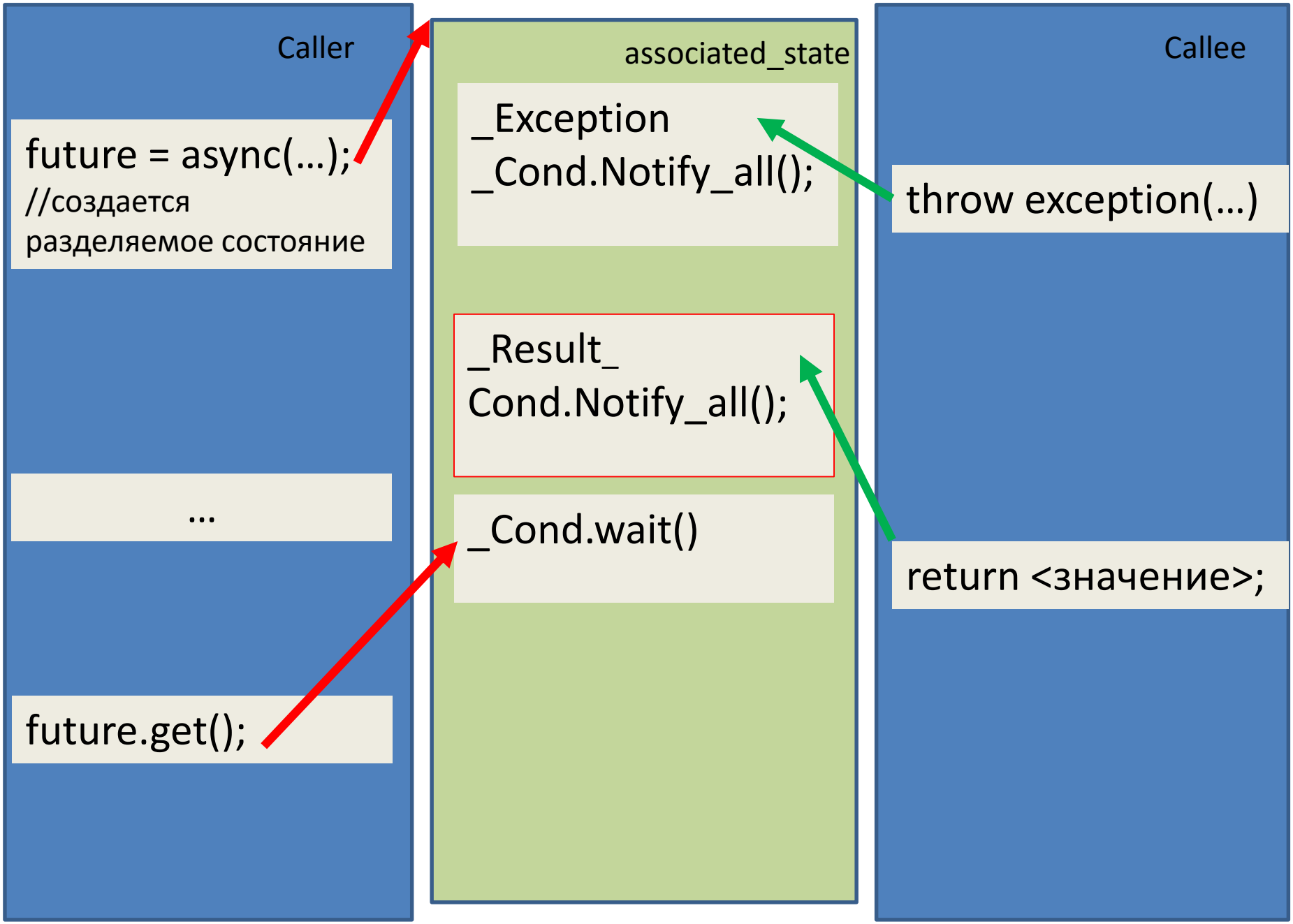
```
std::future<> & std::shared_future<>
```

Что такое future?

- средство **однократной** коммуникации двух потоков посредством «разделяемого состояния» == канал, соединяющий два потока (саморазрушается после приема данных)

Замечание:

- аналог std::unique_ptr, который уничтожается после «чтения»


Caller
associated_state
Callee
future = async(…);
//создается
разделяемое состояние
_Exception
_Cond.Notify_all();
throw exception(…)
Result
Cond.Notify_all();
…
_Cond.wait()
return <значение>;
future.get();

Важно!

- Будущие результаты сами по себе не обеспечивают синхронизацию доступа к разделяемым данным
- => если несколько потоков используют один и тот же объект future,
  - то они сами должны синхронизировать доступ с помощью future::valid(), мьютекса или других средств синхронизации
  - или нужно позволить нескольким потокам пользоваться одним и тем же объектом **std::shared_future** или каждому потоку работать со своей копией - **std::shared_future** без дополнительной синхронизации

Шаблон класса std::future:

- обеспечивает доступ к **обертке** для значения любого типа, вычисление или получение которого происходит отложено (то есть в неизвестном будущем)
- фактически поддерживает механизм однократного уведомления, является получателем будущего значения
- сам по себе future пассивен, он просто предоставляет доступ к некоторому **разделяемому состоянию**, которое состоит из 2-х частей: собственно интересующие **данные** + **флаг готовности** (как только значение вычислено, устанавливается флаг)
- для многих задач менее ресурсоемко, чем condition variable + mutex

Шаблон std::future

предоставляет механизм для однократного получения результата асинхронной операции, инициированной: **std::async**(), std::packaged_task(), std::promise()

- готовность результата посредством методов std::future:
  - **wait()** - блокирует вызвавший поток до получения результата
  - **wait_for()** – ждет завершения или истечения timeout-а
  - **wait_until()** - ждет завершения или наступления момента
  - **get() –** для ожидания вызывает wait() и возвращает результат

std::future::**get()**

- **T get()**; //генеральный шаблон (возвращаемое значение формируется посредством **std::move**) =>
  повторный вызов == неопределенное поведение
- **T& get()**;//для специализации future<T&>
- **void get()**; // для специализации future<void>

Важно!

- Любой из вариантов дожидается результата
- Если было сохранено исключение, оно заново генерируется

std::future::**get()**

При вызове get() могут быть три ситуации:

- если выполнение функции происходило в отдельном потоке и уже закончилось (или результат уже сформирован) – сразу получаем результат
- если выполнение функции происходит в отдельном потоке и еще не закончилось (или результат еще не сформирован) – поток-получатель блокируется
- если запуск был задан синхронно, то происходит просто вызов функции + формирование future

Важно: вторичный вызов get() – неопределенное поведение!

Пример std::async(). Поведение по умолчанию

```
int sum(const std::vector<int>& v){
    int res = 0;
    for (auto i:v) { res += i;}
    return res;
}

int main(){
    std::vector<int> v = { 1, 2, 3, 4 };
    std::future<int> f = std::async(sum, std::cref(v));
    //…какие-то вычисления
    std::cout<<f.get(); //дожидаемся окончания,
                        получаем результат
}
```

Пример std::async() Явное задание условий вызова - std::launch::async:

```
int sum(const std::vector<int>& v){
    int res = 0;
    for (auto i:v) { res += i;}
    return res;
}

int main(){
    std::vector<int> v = { 1, 2, 3, 4 };
    std::future<int> f = std::async(std::launch::async, sum,
                                    std::cref(v));
    //…какие-то вычисления
    std::cout<<f.get(); //дожидаемся окончания,
                        получаем результат
}
```

Пример std::async(). Явное задание условий вызова- std::launch::deferred

```
int sum(const std::vector<int>& v){
    int res = 0;
    for (auto i:v) { res += i;}
    return res;
}

int main(){
    std::vector<int> v = { 1, 2, 3, 4 };
    std::future<int> f = std::async( std::launch::deferred, sum,
                                     std::cref(v));
    //…какие-то вычисления
    if(<условие>) { std::cout<<f.get(); } //синхронный вызов + прием
                                          возвращаемого значения
}
```

std::launch::deferred и std::launch::async

```
void Consumer(std::future<int> f)
        { if(условие) {std::cout << f.get();} }
int Producer() { return rand() % 10; }
int HeavyProducer() { std::this_thread::sleep_for(1s); return 0; }

int main(){
  auto f1 = std::async(std::launch::deferred, Producer);
  std::thread th1(Consumer, std::move(f1));
  auto f2 = std::async(std::launch::async, HeavyProducer);
  std::thread th2(Consumer, std::move(f2));
 //полезная работа
  th1.join(); th2.join();
}
```

Специфика: если функция ничего не возвращает

```
void func(void);

int main(){
   std::future<void> f = std::async( func);
  //…какие-то вычисления
    f.get();  //эквивалентно f.wait(); }
```

std::future_status

- deferred - политика запуска потока была std::launch::deferred
- ready – разделяемое состояние сформировано
- timeout - таймаут истек, а разделяемое состояние **не** сформировано

Пример использования wait_for()

```
//Политику запуска определяет реализация!
using namespace std::chrono_literals;
auto f = std::async([]() {std::this_thread::sleep_for(3s);
        std::cout <<  "Working hard!" << std::endl;});
while (f.wait_for(1s) == std::future_status::timeout) {//если
                политика запуска была deferred, то выход из цикла
        std::cout << "in while" << std::endl; // выполняем
                        другую работу, не требующую результата
}
std::cout << "main" << std::endl;
f.get();
```

Проблемы: отложенный вызов и future::wait_for() или future::wait_until()

Специфика: вызов future::wait_for() или future::wait_until() возвращает для отложенных задач
std::**future_status::deferred =>**

```
int main(){
   using namespace std::chrono_literals;
   auto ft = std::async(std::launch::deferred,[](){…});
   while(ft.wait_for(100ms) !=
                  std:: future_status::ready){/*вечно!*/}
//а сюда мы не попадем НИКОГДА!!!
}
```

Как узнать – какая была политика запуска?

???

Как узнать – какая была политика запуска?

```
auto f = std::async(sum,1,2);
if (f.wait_for(0ms) != std::future_status::deferred)
{
   while (f.wait_for(100ms) != std::future_status::ready) {
   //полезная работа
   }
}
auto res = f.get();
```

std::future::valid()

```
int func(int);
int main(){
   int res=0;
   std::future<int> f1 = std::async( func, 5);
   bool b = f1.valid(); //true
   std::future<int> f2 = std::move( f1);
   b=f1.valid(); //false
   if(f2.valid()){  res = f2.get();} //true
   b=f2.valid(); //false
}
```

Сохранение исключения в объекте future

Если функция, вызванная посредством async(), генерирует исключение,

- это исключение сохраняется в объекте future вместо значения (результата)
- вызов get() повторно возбуждает сохраненное исключение (при этом стандарт не оговаривает: в объекте future создается копия исключения или сохраняется ссылка на оригинал)

Сохранение исключения в объекте future. Пример:

```
double square_root(double x) {
if (x<0) { throw std::out_of_range("x<0"); }
return sqrt(x);
}
int main(){
   std::future<double> f = std::async(square_root, -1);
   double y=0;
   try {
         y = f.get();
    }
   catch (std::out_of_range& e){ std::cout << e.what(); }
//или catch (std::exception& e){ std::cout << e.what(); }
}
```

Пример более жизненный:

```
int main() {
    std::vector<std::future<int>> futures;
    size_t N = <большое!!!>;
    futures.reserve(N);

     for(int i = 0; i < N; ++i) {
      futures.push_back (std::async([](int x){return x*x;},i));
  }
//… занимаемся полезной работой
//нужны ВСЕ результаты!
   for(auto &f : futures) {
    std::cout << f.get() << std::endl;
  }
…
}
```

std::future::~future()

Если деструктор вызывается для последнего «владельца» разделяемого состояния, то разделяемое состояние уничтожается

- standard-30.6.8/3, - The thread object [for the function to be run asynchronously] is stored in the shared state and affects the behavior of any asynchronous return objects [e.g., futures] that reference that state.
- 30.6.8/5 - the thread completion [for the function run asynchronously] synchronizes with [i.e., occurs before]
  [1] the return from the first function that successfully detects the ready status of the shared state or
  [2] with the return from the last function that releases the shared state, whichever happens first.

Как и где происходит «самоуничтожение разделяемого состояния»

Вариант 1.

Caller	Calee

{
auto ft = std::async(callable);
//ref_count++ (2)
 	
int res=ft.get(); //результат «перемещается» из shared_state в res, ref_count– (0) => последний владелец => **уничтожение**!
 	
}//~ft – «пустой» => продолжение вып.
	//создается анонимный функтор, в который запаковываются callable + параметры для callable + указатель на shared_state => ref_count=1
	
	int callable() //ref_count=1
	{
	…
	
	return 1; //установка флага готовности
	}~анонимный функтор => ref_count– (1)

Замечание:

- Если деструктор future вызывается до вызова get() или wait(), деструктор блокирует поток до завершения задачи

Как и где происходит «самоуничтожение разделяемого состояния»

Вариант 2.

Caller	Calee

{
auto ft = std::async(callable);
//ref_count++ (2)
 	
}//~ft – Calee еще выполняется!
ref_count– (1)
ref_count!=0 => **Caller блокируется**,
пока ref_count не станет =0
	//создается анонимный функтор, в который запаковываются callable + параметры для callable + указатель на shared_state => ref_count=1
	
	int callable() //ref_count=1
	{
	…
	
	return 1; //установка флага готовности
	}~анонимный функтор => ref_count–(0); => последний владелец => **уничтожение**!

Как и где происходит «самоуничтожение разделяемого состояния»

Вариант 3.

Caller	Calee

/*auto ft* = /??? std::async(callable);
//ref_count++ (2)
 	
//~возвр_знач – Calee еще выполняется! ref_count– (1)
ref_count!=0 => **Caller блокируется**, пока ref_count не станет =0
	//создается анонимный функтор, в который запаковываются callable + параметры для callable + указатель на shared_state => ref_count=1
	
	int callable() //ref_count=1
	{
	…
	
	return 1; //установка флага готовности
	}~анонимный функтор => ref_count– (0); => последний владелец => **уничтожение**!

Пример:

```
double sumN(int n){
double s = 0;
for (int i = n; i > 0; i--) { s += i; }
std::cout << "done";
return s;
}

int main(){
  {
        std::future<double> f = std::async(sumN, 0x7fffffff);
  } //~f заблокирует родительский поток до завершения дочернего
}
```

STD::SHARED_FUTURE

Если требуется в нескольких потоках ожидать получения одного и того же результата ???

```
int threadFunc(int x, int y){ return x + y; }

void thread1(std::future<int>& f){ int res = f.get(); }
void thread2(std::future<int>& f){ int res = f.get(); }

int main(){
  std::future<int> f = std::async(threadFunc, 1, 2);
  std::thread t1(thread1,std::ref(f));
  std::thread t2(thread2, std::ref(f));
  t1.join();    t2.join();
}
```

Шаблон класса std::shared_future

- позволяет получить результат асинхронной операции, но,
  - в отличие от future (вторичный вызов get() – неопределенное поведение)
  - **shared_future** допускает несколько вызовов get() и возвращает один и тот же результат или генерирует одно и то же исключение
- создать и проинициализировать можно:
  - посредством конструкторов shared_future
  - или методом future::share()

Аналогия:

- std::future – аналог unique_ptr
- std::shared_future – аналог shared_ptr

Формирование std::shared_future посредством std::future

```
std::future<int> f1 = …;
std::shared_future<int> sf1 ( std::move(f1));
bool b1 = f1.valid(); //???

std::future<int> f2 = …;
std::shared_future<int> sf2= f2.share();
bool b2 = f2.valid(); //???
```

Объекты shared_future можно:

- копировать
- присваивать
- перемещать

Модифицируем пример посредством std::shared_future

```
int threadFunc(int x, int y){ return x + y; }

void thread1(std::shared_future<int>& f){ int res = f.get(); …}
void thread2(std::shared_future<int>& f){ int res = f. get(); …}

int main(){
  std::future<int> f = std::async(threadFunc, 1, 2);
  std::shared_future<int> sf(std::move(f));
  std::thread t1(thread1, std::ref(sf));
  std::thread t2(thread2, sf);
  t1.join();    t2.join();
}
```

Или по значению:

```
int threadFunc(int x, int y){ return x + y; }

void thread1(std::shared_future<int> f){ int res = f.get(); …}
void thread2(std::shared_future<int> f){ int res = f. get(); …}

int main(){
  std::future<int> f = std::async(threadFunc, 1, 2);
  std::shared_future<int> sf(std::move(f));
  std::thread t1(thread1, sf);
  std::thread t2(thread2, sf);
  t1.join();    t2.join();
}
```

#include <future>

STD::PROMISE

PROMISES, PROMISES…

std::promise<>

- это средство «активного» формирования результата потока

Специфика std::promise:

- std::promise/std::future обеспечивает механизм (один из возможных), позволяющий передавать значения между потоками
- в отличие от async(), которая явно возвращает объект future, promise «инкапсулирует» future
- механизм формирования результата разный:
  - при вызове async() в разделяемом состоянии **неявно** формируется возвращаемое потоком значение или сгенерированное исключение (при завершении потока)
  - при использовании promise – значение нужно сформировать **явно** (set_value(), set_exception())

Отличия future и promise:

- future позволяет получить данные (get())
- promise позволяет
  передать данные - **set_value**()
  или исключение - **set_exception**()

Связь std::promise<> и разделяемого состояния:

```
template<class _Ty> class promise{// class that defines an asynchronous provider that holds a value
…
private:
        _Promise<_Ty> _MyPromise;
}

template<class _Ty> class _Promise{// class that implements core of promise;
        …
private:
        _State_manager<_Ty> _State;
        bool _Future_retrieved;
};
template<class _Ty> class _State_manager{// class for managing possibly non-existent associated asynchronous state object
…
private:
_Associated_state<_Ty> *_Assoc_state;
bool _Get_only_once;
};

template<class _Ty> class _Associated_state
{// class for managing associated synchronous state
_Atomic_counter_t _Refs;
…
};
```

Связь promise и разделяемого состояния:

```
template<class _Ty> class promise {// class that defines an asynchronous provider that holds a value
public: //основная функциональность
promise() : _MyPromise(new _Associated_state<_Ty>){}

void set_value(const _Ty& _Val)
{ _MyPromise._Get_state_for_set()._Set_value(_Val, false); }

void set_exception(_XSTD exception_ptr _Exc)
{ _MyPromise._Get_state_for_set()._Set_exception(_Exc, false); }

private:
_Promise<_Ty> _MyPromise;
};
```

Связь future и promise:

```
template<class _Ty> class promise {
…
public:
  future<_Ty> get_future(){// return a future object that shares the associated
      return (future<_Ty>(_MyPromise._Get_state_for_future(),
                                                    _Nil()));
  }
…
}
```

Формирование/получение результата

- **std::promise** отвечает за установку значения:
  - std::promise::set_value() – сохраняет значение в разделяемом состоянии и выставляет флаг готовности
  - std::promise::set_value_at_thread_exit()- «-», но не выставляет флаг до завершения потока
  - std::promise::set_exception()
- а **std::future** - за его получение (блокирует поток до установки готовности в разделяемом состоянии):
  - std::future::wait()
  - std::future::get()

Пример promise

```
int main(){
   std::promise<int> p;
   std::future<int> f = p.get_future(); //можем сразу же
                                получить доступ к разделяемому состоянию
   std::thread th( [&p](){p.set_value(33);} );
   th.detach(); //можем подождать завершения – join(), но не интересно
   //делаем что-то полезное
   int res = f.get(); //получаем результат
}
```

Важно!

- Если для одного и того же объекта promise повторно вызвать set_value() или set_exception(), будет сгенерировано исключение!
- Если на момент вызова деструктора результат не сформирован, сохраняет в разделяемом состояние исключение std::future_error с кодом std::future_errc::broken_promise и устанавливает флаг готовности

Пример некорректного использования promise:

```
void fPromise(std::promise<int>& p){
        p.set_value(1);
        //…
        p.set_value(2); //или p.set_exception(std::exception_ptr());
}
int main(){
        std::promise<int> Promise1;
        std::future<int> fut1 = Promise1.get_future();
        std::thread th(fPromise, std::ref(Promise1));
        th.detach();
        std::this_thread::sleep_for(1s);
        int res1 = fut1.get();
}
```

Еще один пример межпоточного обмена данными посредством promise

```
std::promise<int> promise;

void thread_func1(){    promise.set_value(33); }

void thread_func2(){  std::cout << promise.get_future().get(); }

int main(){
   std::thread th1(thread_func1);
   std::thread th2(thread_func2);

   th1.join();    th2.join();
}
```

Пример передачи исключения посредством promise

```
void sum_promise(const std::vector<int>& v,
                                    std::promise<int>& p)
{
  try {
    if (v.empty()) { throw std::runtime_error("empty");    }
    int res = 0;
    for (const auto& i : v){ res += i; }
    p.set_value(res);
   }
     catch (...) {  p.set_exception(std::current_exception()); }
}
```

Продолжение примера

```
int main(){
try {
        std::vector<int> v = { 1,2,3,4,5,6,7 };
        std::promise<int> p;
        std::thread th(sum_promise, std::cref(v),
                                        std::ref(p) );
        th.detach();
        //...
        int sum = p.get_future().get();
 }catch (std::exception& e){…}
}
```

Важно:

- f.get(); блокирует вызвавший поток до формирования результата (когда в другом потоке будет вызвана p.set_value(); или p.set_exception()), а не до завершения поставляющего результат потока!!!
- для ожидания завершения – p. **set_value_at_thread_exit**(res);

Создание потока в приостановленном состоянии

Важно! В тех ситуациях когда, перед тем, как поток начнет выполнение (вернее будет включен в диспетчирование) нужно произвести его «настройку», например:

- установить приоритет (системными средствами)
- получить для других системных действий дескриптор потока

Для этого можно использовать специализацию **std::promise<void>**

Пример создания потока в приостановленном состоянии

```
void SimpleThreadF() { std::cout << "Child" << std::endl; }
int main(){
   std::promise<void> suspendPromise;
   std::thread th([&suspendPromise] {
            suspendPromise.get_future().wait(); SimpleThreadF();});

     //"Настройка" потока:
  HANDLE h = th.native_handle();
  ::SetThreadPriority(h, THREAD_PRIORITY_ABOVE_NORMAL);
   std::cout << "Parent"<<std::endl;
   //Запуск:
   suspendPromise.set_value();
//Возможно еще какая-то полезная работа
   th.join();
}
```

Как получить несколько результатов?

```
void fMultiplePromises(std::promise<int>& p1, std::promise<int>& p2,
                                                    std::promise<int>& p3){
        p1.set_value(1);
        p2.set_value(2);
        p3.set_value(3);
}

int main(){
  std::promise<int> Promise1, Promise2, Promise3;
  std::future<int> fut1 = Promise1.get_future(), fut2 =
        Promise2.get_future(), fut3 = Promise3.get_future();
  std::thread th(fMultiplePromises, std::ref(Promise1), std::ref(Promise2),
                                                    std::ref(Promise3));
  th.detach();
  int res1 = fut1.get();    int res2 = fut2.get();    int res3 = fut3.get();
}
```

Передача исключения посредством promise

- Для передачи исключения - метод std::promise::set_exception (который принимает объект типа std::exception_ptr)
- Напоминание: получить объект std::exception_ptr можно:
  - вызовом std::current_exception() из блока catch
  - либо создать объект этого типа напрямую посредством std::make_exception()

Пример обработки исключения

```
std::promise<int> promise;

void thread_func1(){
promise.set_exception(std::make_exception(std::runtime_error("error
"))); }

void thread_func2(){
   try   {
      std::cout << promise.get_future().get() << std::endl;
   } catch (const std::exception& e) {  std::cout << e.what() << std::endl;  }
}

int main(){
   std::thread th1(thread_func1);    std::thread th2(thread_func2);
    th1.join();    th2.join();
}
```

#include <future>

STD::PACKAGED_TASK

Зачем нужен **packaged_task<>**

Проблема: есть обычная callable, которую хочется выполнить в отдельном потоке и получить **результат**!

Варианты:

1. Изменить сигнатуру функции, чтобы она принимала параметр типа promise – не хочется
2. Изменить сигнатуру функции, чтобы она принимала адрес, по которому сформирует результат – тоже не хочется
3. Использовать async() + future – происходит немедленный запуск потока (launch::async), а хочется только подготовить все к запуску (пул)
4. => «Завернуть» callable в обертку packaged_task

std::packaged_task

- реализует понятие «задачи» - является хранилищем для пары callable + promise => удобно использовать при асинхронном программировании
- расширяет возможности async()
- реализован как функтор (перегружен operator() – возвращаемое значение или исключение запаковывается в «разделяемое состояние» => доступны посредством future<>)

Пояснение:

int sum(int, int);

При создании объекта типа packaged_task:

std::packaged_task<int(int, int)> task1(sum);

компилятор генерирует анонимный функтор типа:

```
template<> class packaged_task<int(int, int)> {
public:
        template<typename Callable> explicit
                        packaged_task(Callable&& f);
        std::future<int> get_future();
        void operator()(int, int);
…
};
```

Специфика std::packaged_task

- исключительно перемещаемый класс
- перегружен operator() таким образом, что он:
  - вызывает callable с сохраненными в объекте packaged_task значениями в качестве параметров
  - сохраняет возвращаемое значение или исключение в разделяемом состоянии и устанавливает в разделяемом состоянии признак готовности
- попытка повторного использования объекта packaged_task – исключение std::future_error

Попытка повторного использования объекта packaged_task:

```
std::packaged_task<double(int)> task1(sumN);
task1(1000);
try{
        task1(2000);
}
catch (std::future_error){ … }
```

Для получения future из packaged_task

std::future<R>

std::packaged_task::get_future();

Важно! Можно вызвать только один раз, так как происходит **перемещение**, а не копирование ассоциированного с packaged_task объекта future (то есть счетчик ссылок в разделяемом состоянии не инкрементируется)

Разница в использовании packaged_task:

```
int sum(int x, int y) { return x + y; }

int main(){
  std::packaged_task<int(int, int)> task(sum); //это просто
                                               подготовка к запуску
  std::future<int> f = task.get_future(); //получаем доступ к
                                          разделяемому состоянию
  task(1, 2); //!!! но! вызов функции в текущем потоке!!!
 //делаем что-то полезное
  int res = f.get(); //получаем результат
}
```

Разница в использовании packaged_task:

```
int sum(int x, int y) { return x + y; }

int main(){
   std::packaged_task<int(int, int)> task(sum); //это просто
                                                подготовка к запуску
   std::future<int> f = task.get_future(); //получаем доступ к
                                           разделяемому состоянию
   std::thread th(std::move(task),1, 2); //!!! запуск потока!!!
    th.detach();
 //делаем что-то полезное
   int res = f.get(); //получаем результат
}
```

Важно! «Заготовить» future необходимо до move(task) !!!

```
std::packaged_task<int(int, int)> task(sum);
bool b1 = task.valid(); //???
std::thread th(std::move(task),1, 2); // запуск потока

bool b2 = task.valid(); //???

//task стал «недействительным»! Его разделяемое состояние и callable перемещены в другой поток
std::future<int> f = task.get_future();

//до «полезного» дело не дойдет, так как будет сгенерировано исключение std::future_error
```

Специфика future и packaged_task:

деструктор future, полученного из packaged_task НЕ блокирует родительский поток до завершения дочернего

```
{
        std::packaged_task<int(int, int)> task(sum);
        std::future<int> f = task.get_future();
        std::thread th(std::move(task), 1, 2);
        th.detach();
} //
```

Специфика packaged_task

так как перегружен operator(), объект std::packaged_task в свою очередь можно использовать везде, где требуется callable:

- в частности «завернуть» в std::function
- объекты такого типа можно хранить в контейнерах

Храним задачи в контейнере:

```
double sumN(int n) {  double s = 0; for (int i = n; i > 0; i--) {s += i;}  return s; }
int main(){
  std::vector<std::packaged_task<double(int)> > tasks;
  tasks.reserve(10);
  std::vector<std::future<double>> futures;
  futures.reserve(10);

  for (int i = 0; i < 10; i++) {
        tasks.emplace_back(sumN);
        futures.push_back(tasks[i].get_future());
  }

  int n = 10;
  for(auto& t:tasks){ std::thread th(std::move(t),n); th.detach(); n *= 2; }
  for (auto& f : futures) { std::cout<< f.get()<<" "; }
}
```

Тип callable в шаблоне packaged_task и неявное приведение типа

```
double sumN(int n);

int main(){
   std::packaged_task<double(int)> task1(sumN); //ок
   std::packaged_task<int(int)> task2(sumN); //ок
   std::packaged_task<double(double)>
                                   task3(sumN); //ок
…
}
```

Пример packaged_task:

```
double accum(const std::vector<double>& v) {
  typedef  decltype(v.begin()) IT;
  std::packaged_task<double(IT, IT, double)> pt1{ std::accumulate<IT,double> };
  std::packaged_task<double(IT, IT, double)> pt2{ std::accumulate<IT,double> };

  auto f0 = pt1.get_future();
  auto f1 = pt2.get_future();

  std::thread t1(std::move(pt1),v.begin(), v.begin() + v.size() / 2, 0); // запускаем
  std::thread t2(std::move(pt2), v.begin() + v.size() / 2,v.end(), 0);
  t1.detach(); t2.detach();
//Полезная работа
  return f0.get() + f1.get();// получаем результаты
}
```

Пример межпоточного использования packaged_task

```
int f(){   return 1;}

std::packaged_task<int()> task(f);

void thread_func1(){   task();}

void thread_func2(){
   std::cout << task.get_future().get() << std::endl;
}

int main(){
   std::thread th1(thread_func1);
   std::thread th2(thread_func2);
   th1.join();    th2.join();
}
```

void std::packaged_task:: **make_ready_at_thread_exit**(ArgTypes... args);

- принимает параметры для вызова *packaged_task*.
- выполняет код функции, сохраненной в *packaged_task*, результат (возвращаемое значение или исключение) сохраняется в разделяемом состоянии (future), НО флаг готовности будет выставлен только после того, как все деструкторы локальных, для данного потока, объектов будут выполнены (т.е. прямо перед завершением потока)

СРАВНЕНИЕ НИЗКОУРОВНЕВЫХ И ВЫСОКОУРОВНЕВЫХ СРЕДСТВ СОЗДАНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Когда и что удобнее использовать: std::thread

std::thread – программирование на уровне потоков:

- «+»
  - для получения доступа к системным средствам
  - для организации пула потоков
  - для задания thread local – данных
  - для использования средств синхронизации
- «-»
  - трудно/небезопасно получать результат работы потока
  - только два варианта взаимодействия

condition_variable + mutex

- Для многократного уведомления
- в задачах, где есть субъект, гарантированно/постоянно ожидающий наступления некоторого события

Когда и что удобнее использовать: std::async и std::future

std::async – программирование на уровне задач (однократное получение результата):

- «+»
  - взаимодействие посредством разделяемого состояния (данные + флаг готовности)
  - возможность синхронного/асинхронного запуска
- «-»
  - по умолчанию неизвестно, как (синхронно/асинхронно) выполняется задача (=> проблемы thread_local переменных и мьютексов)
  - если запуск асинхронный, стартует сразу!
  - std::future::get() (wait()) блокирует родительский поток до завершения дочернего (return <значение> или throw <исключение>)
  - ~future() может блокировать родительский поток

Когда и что удобнее использовать: std::shared_future

для многократного получения одного и того же результата (в разных потоках)

Когда и что удобнее использовать: std::promise

std::promise – хранилище для результата (который можно получить посредством std::future):

- позволяет подготовить/настроить пару callable + future
- явная установка результата (или генерация исключения) – set_value() / set_exception()
- родительский поток блокируется не до завершения дочернего, а до формирования результата

Когда и что удобнее использовать: std::packaged_task

std::packaged_task – «упаковывает» callable + promise

- можно «вызвать» напрямую (перегружен operator()) в текущем потоке
- можно в свою очередь «завернуть» в объект function
- можно передать в качестве параметра в потоковую функцию (и в любую другую функцию)
- объединить в коллекцию

Программирование без блокировок

Герб Саттер:

«Если Вы думаете, что программирование без блокировок это просто, значит или Вы – один из тех 50, которые умеют это делать, или же используете атомарные инструкции недостаточно аккуратно.»

Цели многопоточного программирования **без блокировок**:

- Исключение накладных расходов на блокировку потока (переключение контекста) => (в большинстве случаев) повышение эффективности
- Без использования мьютексов (т.е. без явной организации критической секции) исключить неопределенное поведение при работе с разделяемыми данными
- Решение проблемы выбора размера критической секции (простота или масштабируемость)
- В некоторых случаях решение проблем, связанными с взаимными блокировками (dead locks) в случае необдуманного использования мьютексов (порядок блокировки разных мьютексов в разных потоках)

Виды алгоритмов, свободных от блокировок:

- без ожиданий (wait-free) – “**никто никогда не ждет**”. Каждая операция завершается за N шагов без каких-либо условий. Гарантии:
  - максимум пропускной способности системы
  - отсутствие голодания
- без блокировок (lock-free) – “**всегда какой-то из потоков работает**”. Гарантии:
  - максимум пропускной способности системы
  - один из потоков может постоянно ожидать
- без остановок (obstruction-free) – “**поток работает, если нет конфликтов**”. При возникновении конфликта за ограниченное число шагов один поток достигает результата при условии, что конфликтующие потоки остановлены
  - все потоки не блокируются из-за проблем с другими потоками
  - не гарантируется прогресс, если одновременно работают два и больше потоков

MEMORY MODEL C++11

Б. Страуструп – multithreading in C++11

«С точки зрения **параллельного** программирования ключевые новшества C++11 состоят в:

- **организации/модели памяти**,
- портируемости многопоточных программ»

До стандарта C++11

Порядок вычисления выражения компилятором? Точки следования?

Примеры точек следования:

```
int f(int, const char*);
const char* fStr();

int main()
{
        int n = <выражение>, m = <выражение>;
        int res = f(n+m, fStr()); //???
}
```

Зачем нужна модель памяти

- модель памяти регламентирует **разрешенное** поведение многопоточных программ при обращении потоков к **разделяемым данным**
- программист (хороший), управляя моделью памяти, получает возможность создавать не только **безопасные**, но и **эффективные** параллельные программы

Модель памяти

Программист должен представлять:

- размещение составляющих программы в памяти (код, локальные данные, статические данные, динамические данные, thread local данные) – **структурный аспект модели памяти**
- для создания параллельных программ добавляется обеспечение синхронизации доступа к разделяемым данным - **параллельный аспект модели памяти,** ! иначе **неопределенное поведение**!

При обращении к разделяемым данным в многопоточной программе

Модель памяти == контракт:

- программист обеспечивает корректную синхронизацию,
- система в целом (компилятор + ОС + процессор + кэш) обеспечивает иллюзию того, что наш код исполняется так, как мы задумали

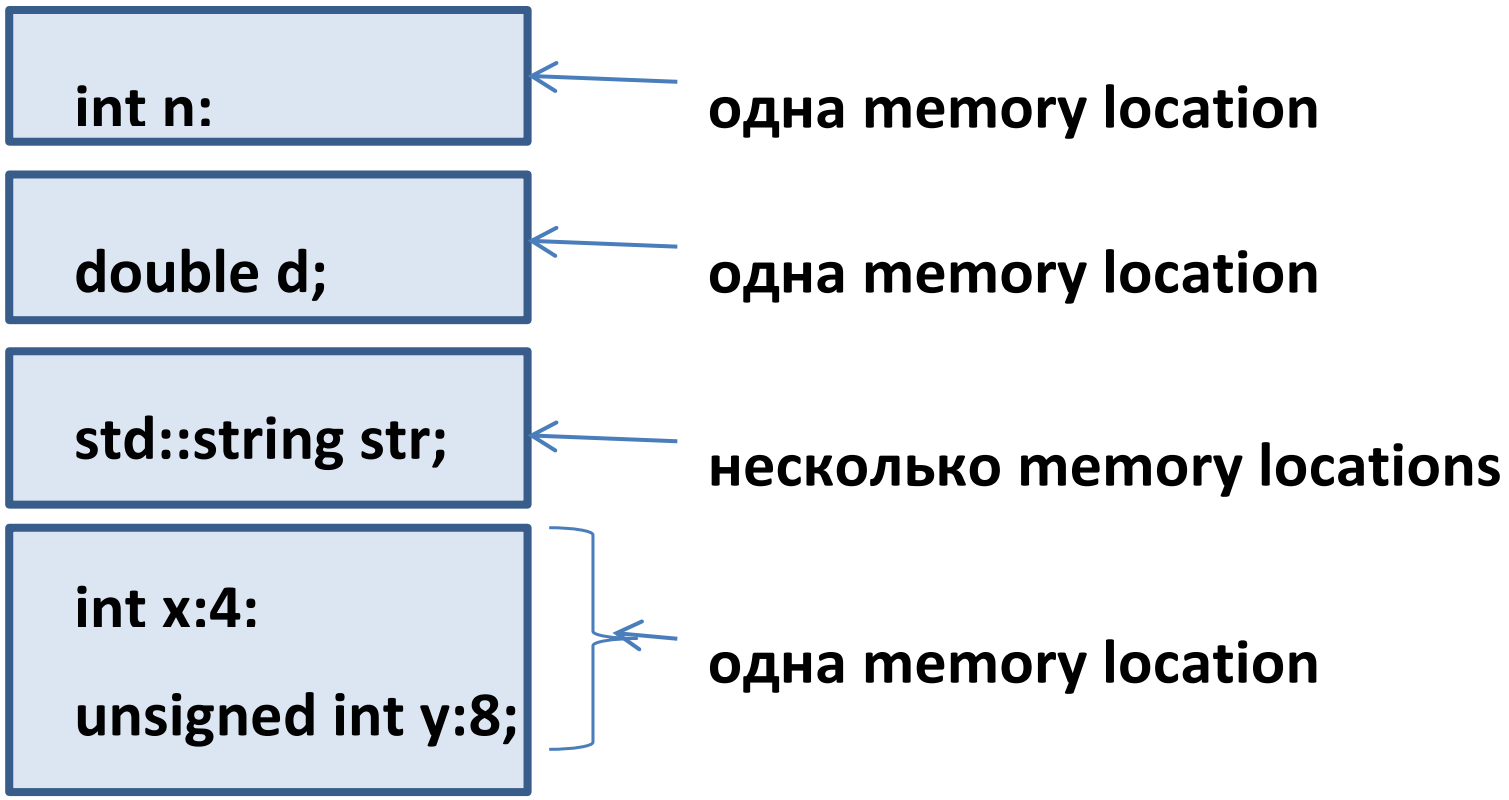
Правила формируются относительно понятия memory location

memory location – структурный аспект memory model (размещение данных в памяти)

- в C++11 введено понятие **memory location** (до многопоточных приложений было неактуально)
- смысл memory location:
  - все данные в C++ программе строятся из объектов
  - struct, class – объект, который является контейнером для других подобъектов
  - в стандарте C++ программный объект определяется как «область памяти»
  - программный объект может занимать одну и более memory locations
  - переменные базовых типов (int, char…), указатель занимают **одну memory location** независимо от размера
  - смежные битовые поля (adjacent bit fields) принадлежат одной и той же memory location

Примеры memory location

```
struct Sample{
    int n:
    double d;
    std::string str;
    int x:4:
    unsigned int y:8;
};
```



Цель введения в C++11 понятия memory locations?

для формирования правил упорядочения обращения к данных в **многопоточном** приложении

memory location – аспект параллелизма memory model

- Модификация различными потоками **различных** memory locations → OK
- Чтение различными потоками **одной** memory location → OK
- Чтение/запись одним потоком memory location, модифицируемой другим потоком → Data race → Undefined Behavior!!!

Модель памяти C++11:

- - учитывает специфику многопоточных приложений
- - предоставляет программисту **средства** для **принудительного** упорядочения параллельного выполнения кода

Актуальны только при многопоточной организации программы

ПРОБЛЕМЫ КОНКУРЕНТНОГО ПРОГРАММИРОВАНИЯ

Актуальность проблем

- До поддержки многопоточности (C++11) – НЕ актуальны
- При создании многопоточных приложений:
  - несинхронизированный доступ к данным
  - неатомарные операции чтения/записи данных
  - переупорядочение выполнения кода

Несинхронизированный доступ к данным.

Пример:

```
if( value <0 )
{ value = -value;}
```

//используем **только** положительное значение value

Проблемы:

- в однопоточном приложении???
- в многопоточном приложении???

Несинхронизированный доступ к данным.

Еще один пример:

std::vector<int> v;

//поток 1
if(!v.empty())
{
 	
std::cout << v.front() ; //???
…
}
	//поток 2
	…
	
	v.pop();

Несинхронизированный доступ к данным.

Замечание:

При использовании средств стандартной библиотеки следует учитывать:

функции стандартной библиотеки не поддерживают (пока) **конкурентные операции чтения и записи разными потоками одних и тех же данных!**

http://libcds.sourceforge.net/

libCDS –**Concurrent Data Structure** - open source C++ библиотека **lock-free** контейнеров и алгоритмов безопасного освобождения памяти (safe memory reclamation).

Параллельная STL (Parallel Studio XE)

- предусмотрена интеграция с VS17 (?)
- Threading Building Blocks – 2018:

например,

for_each(exec_policy, …); //seq, par, vect (?)

Неатомарные операции модификации данных – «частично» модифицированные данные

Поток 1

count++;

Поток 2

count++;

Поток 3

count++;

Глобальная переменная

int **count**=0;

Когда все потоки завершатся, значение count?

Детализация неопределенного поведения: count = ???

Поток 1

count++;

Поток 2

count++;

Поток 3

count++;

Глобальная переменная

int **count**=0;

```
count++;
mov eax,dword ptr ds:[смещ]
add eax,1
mov dword ptr ds:[смещ],eax
```

volatile тоже не помогает!

Поток 1:
count++;

Поток 2:
count++;

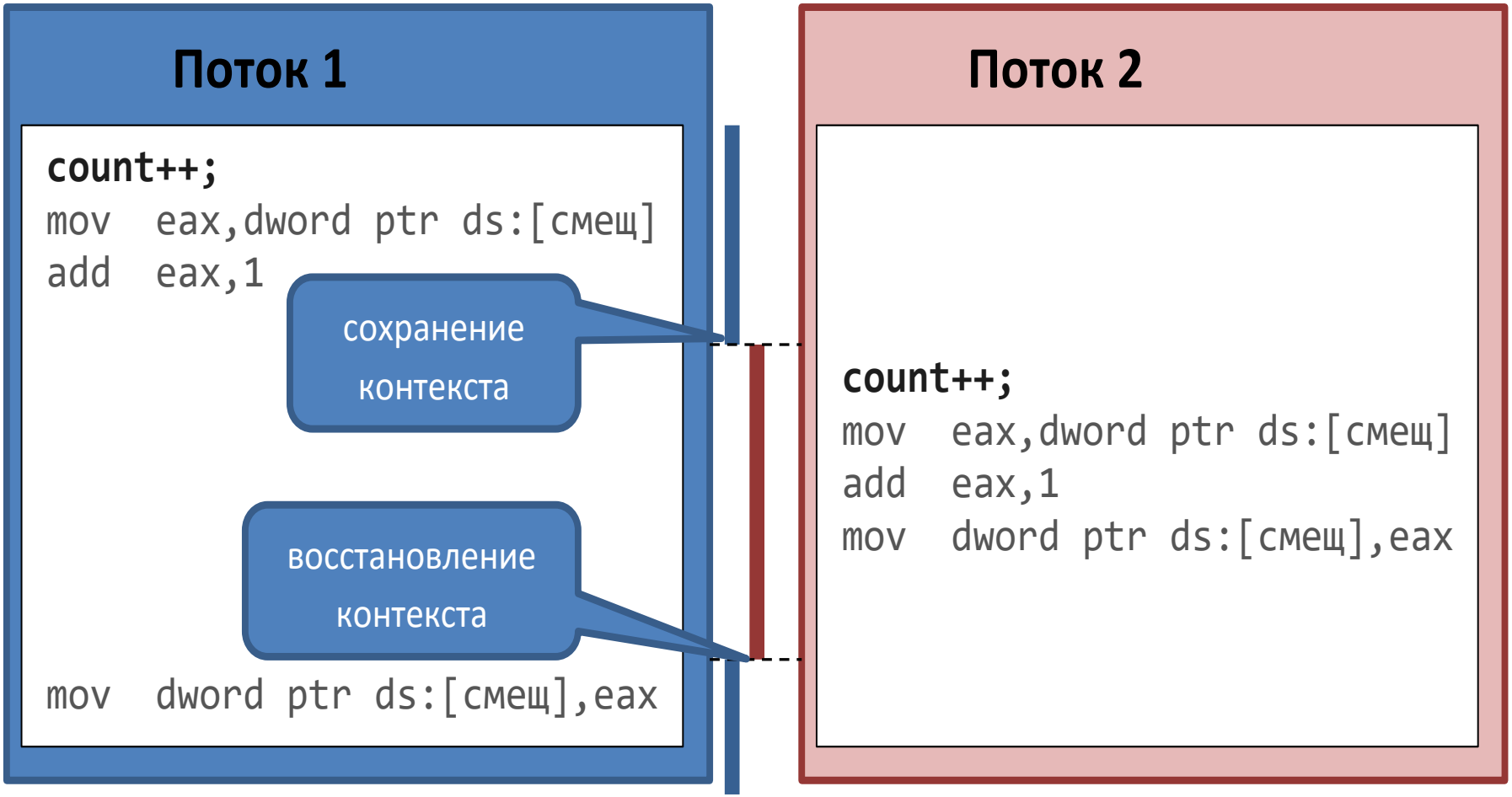
Поток 3:
count++;

Глобальная переменная:
volatile int count=0;

count++;

mov eax,dwordptr ds:[смещ]
add eax,1
mov dwordptr ds:[смещ],eax

Демонстрация проблемы



Разные данные, принадлежащие одному memory location ???

```
struct Test{
int a:4;
int b:5;
};
```

Test global;

Поток 1

global.a=1;

Поток 2

global.b=2;

В общем случае стандарт не дает гарантий даже относительно разных memory locations!

global.**a** = ? global.**b** = ?

```
struct Test{

int a;

int b;

};
```

Test global;

Поток 1

global.a=1;

Поток 2

global.b=2;

Согласно стандарту компилятор имеет полное право преобразовать код:

```
Test    global; //Глобальная
...
//Поток 1
Test tmp = global;
tmp.a = 5;
global = tmp;
...
//Поток 2
Test tmp = global;
tmp.b = 4;
global = tmp;
```

Переупорядочение кода (reodering)

Эпиграф:

Если Вы думаете, что программа всегда выполняется так и в таком порядке, как Вы ее написали,

Вы ошибаетесь!

Стандарт C++11:

«Реализация может свободно **игнорировать** любое требование международного стандарта, если результирующее поведение программы выглядит так, «**как будто**» это требование было выполнено.

Например, фактическая реализация не обязана вычислять часть выражения, которая в дальнейшем не используется, и при этом не возникают побочные эффекты»

Следовательно

сгенерированный (для одного и того же высокоуровневого текста на C++) код – это

«черный ящик»,

- содержимое которого может быть разным
- в то время, как **наблюдаемое поведение** должно быть одинаковым (независимо от реализации)

Надежды и реальность:

- Программист надеется на то, что порядок действий будет таким, как он запланировал
- Но! в современных сложно организованных архитектурах **в целях повышения производительности** этот порядок может **не** соблюдаться!

=> в общем случае (без использования средств C++11) соответствия

«расположено перед» == «происходит раньше»

нет!

Кто и зачем меняет наш код:

- **Компилятор** – оптимизация, переупорядочивание кода (reordering), линеаризация циклов, исключает то, что с его точки зрения бесполезно (“мертвый код”). . .
- **Процессор** – предвыборка кода (prefetch), спекулятивное выполнение (speculative execution), вычислительный конвейер . . .
- **Кэши** – могут содержать разные значения для одной и той же ячейки памяти (memory location) => синхронизация кэшей

Предвыборка кода

- — это выдача запросов со стороны процессора в оперативную память для считывания инструкций **заблаговременно**, до того момента как эти инструкции потребуется исполнять.
- В результате этих запросов, инструкции загружаются из памяти в **кэш** => когда инструкции потребуется исполнять, доступ к ним будет осуществляться значительно быстрее, так как задержка при обращении в кэш на порядки меньше, чем при обращении в оперативную память.

Параллелизм на уровне команд (ILP)

- **Спекулятивное выполнение** - опережающее выполнение команд прежде, чем становится известно, что их выполнение необходимо (speculative execution)
- **Вычислительный конвейер** - выполнение нескольких инструкций может частично перекрываться

Пример переупорядоченных компилятором инструкций – взгляд программиста

```
int data;
bool readyFlag = false;
```

Поток, формирующий данные	**Поток, обрабатывающий данные**
data = <значение>; readyFlag = true;	while(!readyFlag){;} //Считаем, что данные готовы => обрабатываем

Пример переупорядоченных компилятором инструкций – взгляд компилятора

- Компилятор может сгенерировать код в том порядке, который задал программист:

  data = <значение>;

  readyFlag = true;

- А может и в другой (оптимизируя код):

  readyFlag = true;

  data = <значение>;

так как с точки зрения компилятора на конечный результат порядок не влияет (компилятор ничего не знает про второй поток)

Пример переупорядочения компилятором низкоуровневых инструкций при включении оптимизаций

```
int A, B=33;
void f()
{
   A = B + 1;
   B = 0;
}
```

//Без оптимизаций (псевдокод)	//с включенными оптимизациями
mov eax, DWORD PTR [B] add eax, 1 mov DWORD PTR [A], eax **mov DWORD PTR [B], 0**	mov eax, DWORD PTR [B] **mov DWORD PTR [B], 0** add eax, 1 mov DWORD PTR [A], eax

Модель памяти C++11:

предоставляет различные **ограничения** на переупорядочение компилятором операций при выполнении действий над **атомарными типами**

Важно! эти ограничения формируются вокруг атомарных операций, но воздействуют как на атомарные операции, так и на обычные (неатомарные)

Итог – проблемы, возникающие при конкурентном программировании:

- несинхронизированный доступ к данным (порядок чтения и записи в разных потоках)
- частично записанные данные (чтение в одном потоке началось раньше, чем закончилась запись в другом)
- переупорядоченные операции (компилятор может изменять порядок выполнения инструкций, если при перестановке поведение **данного** конкретного потока остается корректным)

Проблемы, порождающие неопределенное поведение:

- несинхронизированный доступ к данным
- неатомарные операции модификации данных
- переупорядочение выполнения кода

СПОСОБЫ РЕШЕНИЯ ПРОБЛЕМ

При конкурентном программировании

необходимо **принудительно** упорядочить обращения к данным из разных потоков!

Способы решения проблем:

- **атомарность** (в широком смысле – операция или последовательность операций в одном потоке не может быть прервана другим потоком)
- **порядок** – гарантия того, что порядок выполнения операций будет упорядочен

Средства, предоставляемые **C++11** для решения проблем конкурентного доступа:

- **атомарные типы данных**
- **атомарные операции**
- условные переменные (conditional variables)
- мьютексы и блокировки
- futures и promises

Недостатки использования примитивов синхронизации, предоставляемых ОС:

- в некоторых реализациях являются объектами исполняющей системы
- => обращение к такому объекту может быть очень дорогим: может потребоваться переключение контекста, переход на уровень ядра ОС, поддержка очередей ожидания доступа к защищаемым примитивом синхронизации данным и пр.

И если это нужно только для того, чтобы есть выполнить **одну-две ассемблерных инструкци**и, то это:

- неэффективно
- + объект ядра ОС – это ограниченный по количеству ресурс.

Важно!

- **Высокоуровневые средства** (futures, promises, mutexes, conditional variables)
  - относительно просты в использовании
  - и безопасны
  - но! менее эффективны
- **Низкоуровневые средства** (атомарные переменные и атомарные операции)
  - обеспечивают бОльшую производительность
  - являются более универсальными
  - но! риск их неправильного использования велик!

Неблокирующая синхронизация

Википедия:

«Неблокирующая синхронизация - подход в параллельном программировании на симметрично-многопроцессорных системах, проповедующий **отказ от традиционных примитивов блокировки, таких, как семафоры, мьютексы и события**. Разделение доступа между потоками идёт за счёт **атомарных операций и специальных, разработанных под конкретную задачу, механизмов блокировки**»

Важно!

- Писать lock-free код не просто.
- Писать **правильный** lock-free код невероятно сложно.

Цели lock-free параллельных программ:

- Повышение масштабируемости путем сокращения блокировок и ожиданий
- При этом нужно гарантировать отсутствие неопределенного поведения
- Решение проблем, связанных с блокировками, например, dead-lock-ов

#include <atomic>

АТОМАРНЫЕ ОПЕРАЦИИ И АТОМАРНЫЕ ТИПЫ

М.Полубенцева

Атомарные объекты и атомарные операции

- это самый низкий уровень синхронизации в многопоточных приложениях, доступный в C++

Атомарность может быть реализована:

- на аппаратном уровне (когда непрерывность обеспечивается аппаратурой) – действительная атомарность
- или эмулироваться программно.

Замечание: **volatile** и параллелизм в C++

volatile в C++ :

- только предотвращает агрессивную оптимизацию
- не гарантирует
  - ни атомарности
  - ни порядка

=> для C++11 указание volatile для атомарной переменной - только для предотвращения оптимизаций компилятора => не обязательно

Напоминание - пример неопределенного поведения

Поток 1

count++;

Поток 2

count++;

Поток 3

count++;

Глобальная переменная

int **count**=0;

```
count++;
mov eax,dword ptr ds:[смещ]
add eax,1
mov dword ptr ds:[смещ],eax
```

Модифицируем пример, используя атомарные типы и операции

Поток 1

```
++atomicCount;
```

Поток 2

```
++atomicCount;
```

Поток 3

```
++atomicCount;
```

Глобальная переменная

```
std::atomic<int> atomicCount(0);
```

```
++atomicCount;
mov ecx,<&atomicCount> //this
call std::_Atomic_int::operator++
```

Важно!

Использование атомарных операций

- **не** предотвращает гонку (какая атомарная операция выполнится раньше, не гарантируется)
- но! позволяет избежать **неопределенного поведения**

Атомарные операции – один из способов избежать неопределенного поведения

- -операции, которые гарантированно будут выполнены **целиком** даже в том случае, когда в процессе выполнения такой операции квант времени потока истекает или данный поток должен быть вытеснен другим более приоритетным потоком

  Замечание: - это «облегченная» альтернатива мьютексу
- согласно стандарту C++11 в атомарных операциях используются специальные атомарные типы

Атомарные операции – альтернатива мьютексам

Почему введены (помимо мьютексов) атомарные операции:

- ‘+’
  - эффективность (выигрыш по производительности, так как не требуется переключение контекста, переход на уровень ядра…)
  - программисту не нужно **явно** обеспечивать неделимость (реализуется средствами стандартной библиотеки посредством команд процессора или эмулируется)
- ‘-’
  - относительная сложность корректного использования

Замечание:

- мьютексы **могут быть** (**необязательно**) медленнее операций над атомарными объектами
- => не нужно воспринимать это как мьютексы плохие, а атомарные объекты – «наше всё».

Мьютексы и атомарные объекты созданы для разных ситуаций => прежде чем использовать атомарные объекты, нужно оценить **cost/benefit**

Когда выигрыша от использования атомарных операций можно не получить:

- атомарный доступ может понадобится к данным любой сложности и размера
- настоящей атомарной операцией над данными можно считать лишь ту, которую процессор может выполнить **одной командой**
- процессор не имеет таких команд для структур данных любой сложности
- => чтобы оставить возможность атомарного доступа, и не скатиться лишь к базовому набору типов, **атомарность должна эмулироваться для всех типов**, которые процессор не может обрабатывать атомарно! Возможно, за счёт тех же самых мьютексов.

Возможные способы реализации/эмуляции атомарных операций:

- **использование атомарных инструкций процессора**
- запрет/разрешение прерываний (для очень коротких действий?)
- синхронизирующие примитивы (для длинных?)
- запрещение переключения контекста потока
- блокировка шины
- повышение приоритета

Атомарные операции

- делятся на простые
  - чтение
  - запись
- и операции атомарного изменения (read-modify-write, RMW)

Пример: атомарные инструкции x86

- **CMPXCHG/CMPXCHG8B/CMPXCHG16B** — основная атомарная команда x86 (сравнение и обмен). При использовании с префиксом **LOCK** атомарно выполняет сравнение переменной с указанным значением и пересылку в зависимости от данного сравнения. Является основой реализации всех безблокировочных алгоритмов. Часто используется в реализации спинлоков и RWLock'ов, а также практически всех высокоуровневых синхронизирующих элементов, таких как семафоры, мьютексы, события и пр. в качестве внутренней реализации
- **XCHG** — Операция обмена между памятью и регистром. Выполняется атомарно на x86-процессорах. Часто используется в реализации спинлоков.

Продолжение

Кроме того, многие команды вида Чтение-Модификация-Запись могут быть сделаны искусственно атомарными с помощью префикса **LOCK**:

- Команды сложения и вычитания **ADD/ADC/SUB/SBB**, где операнд-приемник является ячейкой памяти
- Команды инкремента/декремента **INC/DEC**
- Логические команды **AND/OR/XOR**
- Однооперандные команды **NEG/NOT**
- Битовые операции **BTS/BTR/BTC**
- Операция сложение и обмен **XADD**

Префикс **LOCK** вызывает блокировку доступа к памяти на время выполнения инструкции. **Блокировка может распространяться на область памяти шире, чем длина операнда, например, на длину кэш-линии**.

Замечание:

- В современных процессорах гарантируется настоящая атомарность чтения/записи/обмена только **выровненных** простых (integral) типов – целых чисел и указателей.

Атомарные инструкции и компилятор

Компиляторы языков высокого уровня сами никогда не используют при генерации кода атомарные инструкции, так как:

- атомарные операции во много раз более ресурсоёмкие, чем обычные
- у компилятора нет информации, когда доступ должен осуществляться атомарными инструкциями

=> программист использует один из следующих подходов:

- ассемблерная вставка соответствующей атомарной инструкции
- использование расширения компилятора (функции семейства __builtin_ или __sync_)
- использование «высокоуровневой» обертки в виде специальной библиотеки
- вызов системной функции (Windows – семейство interlock-функций)
- **Использование C++11, поддерживающего типы atomic и функции семейства atomic_**

Ассемблерная вставка:

```
    int nGlobal=0;

int main()
{
    __asm LOCK INC dword ptr [nGlobal]
                        //ds:[адрес]
}
```

класс **atomic_flag**

- не шаблон!
- единственная **гарантированная** (независимо от реализации) lock-free структура данных
- предоставляет минимум функциональности
- => используется в очень простых задачах

Инициализация **atomic_flag:**

- обязательно (независимо от времени жизни) должен быть проинициализирован значением **ATOMIC_FLAG_INIT** (создается в сброшенном состоянии)
- atomic_flag(const atomic_flag&) = delete;
- atomic_flag(); //unspecified state

Специфика **atomic_flag:**

- atomic_flag& operator=(const atomic_flag&) = delete;
- void atomic_flag::**clear**(<порядок>);
  глобальной функции вида –
  void **atomic_flag_clear**(std::atomic_flag* p);
  void **atomic_flag_clear_explicit**(std::atomic_flag* p, <порядок>);
- bool atomic_flag::**test_and_set**(<порядок>);
  bool **atomic_flag_test_and_set**(std::atomic_flag* p);
  bool **atomic_flag_test_and_set_explici**t
  (std::atomic_flag* p, <порядок>);

Замечание: единственный и обязательный способ инициализации

Замечание: реализация ATOMIC_FLAG_INIT зависит от реализации:

```
#define ATOMIC_FLAG_INIT /* implementation-defined */

//std::atomic_flag f(ATOMIC_FLAG_INIT);
                    //ошибка - конструктор копирования delete
                //даже, если компилятор ошибки не выдает - unspecified

std::atomic_flag f1{ ATOMIC_FLAG_INIT }; //OK
std::atomic_flag f2 = ATOMIC_FLAG_INIT; //OK
```

Замечание:

```
std::atomic_flag f1; //unspecified state
```

bool atomic_flag::**test_and_set()**

- устанавливает флаг в true и
- возвращает
  - true, если флаг был на момент вызова установлен (true)
  - иначе false

Замечание: такой проверкой можно «отследить» сброс флага в другом потоке!!!

Пример: реализация spin-lock “мьютекса” посредством **atomic_flag**

```
class handmade_mutex{
   std::atomic_flag flag;
public:
  handmade_mutex():flag{ATOMIC_FLAG_INIT}{}
  void lock() { while(flag.test_and_set()){;}  }
  void unlock() { flag.clear(); }
};
```

Реализация try_lock()

???

Продолжение примера. Использование hand made мьютекса

```
handmade_mutex m; //глобальная переменная

//поток, выполняющий работу над разделяемыми данными
void myThread()
{
        m.lock();
        //работа с разделяемыми данными
        m.unlock();
}
```

Продолжение примера. Использование lock_guard

```
handmade_mutex m; //глобальная переменная

//поток, выполняющий работу над разделяемыми данными
void thread()
{
        std::lock_guard<handmade_mutex> lk(m);
        //работа с разделяемыми данными
} //???
```

Шаблон структуры std::**atomic**<T>

- является **оберткой** для атомарного значения
- предоставляет основные операции для выполнения атомарных действий над хранящимся значением:
  конструкторы,
  load(),
  store(),
  is_lock_free(),
  exchange(), …
- запрещает копирование и присваивание объектов типа atomic
- большинство параметров принимаются **по значению**
- результат возвращается **по значению**
- для большинства методов есть перегруженные
  () volatile

Псевдонимы:

typedef	тип
std::atomic_bool	std::atomic<bool>
std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
…	…

Специализации шаблона atomic<T>:

- **bool**
- **любых целых типов**
- **указателей**

Замечание: класс **atomic_flag** – это **не** специализация atomic<T>:

- гарантированно lock-free
- и не эквивалентен atomic<bool>

Замечание:

большинство методов класса atomic, например:

T std::atomic<T>::**load();**

дублируются шаблонами глобальных функций вида:

template< class T > T **atomic_load**
(const std::atomic<T>* obj);

Инициализация

- **atomic()**=default ;

  для завершения инициализации можно использовать
  template< class T > void **atomic_init**(
  std::atomic<T>* obj, T desired);
  ,которая не является атомарной!

- **atomic(T desired)**;

- **atomic(const atomic&) = delete**;

store() и load()

- методы класса вида:
  T **load**(<порядок>) const;
  void **store**(T desired, <порядок>);
- шаблоны глобальных функций вида:
  template< class T > void
  atomic_store(std::atomic<T>* obj, T desr);

  template< class T > void
  atomic_store_explicit (std::atomic<T>* obj,
  T desr , <порядок>);

is_lock_free()

метод bool **is_lock_free**() const;

шаблон глобальной функции - template< class T > bool **atomic_is_lock_free**(const std::atomic<T>* obj);

Проверка:

- true – если операции для данного типа действительно реализуются атомарно (посредством одной команды процессора)
- false – если операции эмулируют атомарность

operator=

atomic& **operator=**(**const atomic&**) **= delete**;

T **operator=**(**T** desired); //возвращает копию desired
//эквивалентно store(desired)

std::atomic<int> n(1);
int m=33;
n=m;

operator T ()

- эквивалентен load()

```
std::atomic<int> n(1);
int m = n;
```

exchange()

метод:

T **exchange**(T desired, <порядок>);

шаблон глобальной функции:

template< class T > T
atomic_exchange(std::atomic<T>* obj, T desr);

template< class T > T
atomic_exchange_explicit(std::atomic<T>* obj,
T desr, <порядок>);

???

```
std::atomic<int> n(1);
int m = n.exchange(2);
// n==? m==?
```

compare_exchange_weak(), compare_exchange_strong()

перегруженные методы вида:

bool **compare_exchange_strong**
(T& expected, T desired, <порядок>);

- сравнивают expected (ожидаемое значение) с хранящимся атомарным значением
- и, если они совпадают, заменяет хранящееся значение на desired, возвращает true
- если не совпадают, загружает текущее значение по адресу expected, возвращает false (текущее при этом не меняется!)

Замечание: соответствующие шаблоны глобальных функций – **atomic_compare_exchange_strong**()

Как работает (псевдокод)

```
bool cmp_exch(T* cur, T* expected, T desired)
{
        if(*cur==*expected){
                * cur = desired;
                return true;
        }else{
                *expected = *cur;
                return false;
        }
}
```

Пример:

```
std::atomic<int> current(1);
int expected = 0;
bool b1 = std::atomic_compare_exchange_strong(&current,
                    &expected, 3); //false, expected = 1, current не изм.

//или
//bool b1 = current.compare_exchange_strong( expected, 3);

expected = 1;
bool b2 = std::atomic_compare_exchange_strong(&current,
          &expected, 3);
//true, expected не изм., current =3
```

Пример обычного использования

```
extern std::atomic<bool> b; //внешняя

//Поток
{
   bool expected = false;
   do{expected=false;}
    while(//цикл продолжается, пока expected==false
      !b. compare_exchange_strong(expected, true) ) ;

}
```

Замечание:

версия **weak** – эффективнее, но отличается тем, что сохранение desired может **не** произойти даже в том случае, когда текущее значение совпадает с expected (текущее значение не изменится, а функция вернет false). Такое возможно, если в наборе команд процессора нет аппаратной команды сравнить-и-обменять => поток может быть вытеснен в середине требуемой последовательности => процессор не может гарантировать атомарность => “ложный” отказ => обычно используется в цикле (с маленьким телом) -> возможна дополнительная проверка

Пример использования compare_exchange_weak()

```
   extern std::atomic<bool> b; //внешняя
//Поток
{
   bool expected = false;
   while(
      !b. compare_exchange_weak(expected, true)
         && !expected) {expected=false;}
         //работаем «под защитой»
         b.store(false);
}
```

Замечание 1: только для специализаций шаблона atomic<Integral> :

- добавлены специализированные методы вида:
  T **fetch_add**(T , <порядок>);
  …
  Важно! атомарно модифицируют хранящееся значение, а возвращают то значение, которое было до выполнения операции
- перегружены операторы:
  operator++ //Важно! **T** operator++(); экв. fetch_add(1)+1
  operator--
  совмещенные операторы присваивания:
  operator+=
  operator&=
  …
- методам соответствуют шаблоны глобальных функций вида:
  template< class Integral > Integral
  atomic_fetch_add(std::atomic<Integral>* obj, Integral arg);

Замечание 2: только для специализаций шаблона atomic<T*> :

- добавлены специализированные методы вида:
  T* **fetch_add**(std::ptrdiff_t , <порядок>);
  …
- перегружены операторы:
  operator++
  operator--
  operator+=
  operator-=

Важно!

запрещено использовать с нетривиальными типами, например:

```
std::atomic<std::vector<int>>
```

так как:

- у вектора нетривиальный конструктор копирования
- нетривиальный operator=

Переписываем пример в атомарных терминах

Поток 1

```
++count;
```

Поток 2

```
++count;
```

Поток 3

```
++count;
```

Глобальная переменная

```
std::atomic<int> count(0);
```

Использование атомарных операций

```
#include <atomic>
std::atomic<int> count (0);

поток_1
{
count++; //count==1

//или
int previous = count.fetch_add(1);
                    //например, previous==0, count==1…
}
```

Важно!

- каждая из операций может быть атомарной, но их комбинация атомарной НЕ ЯВЛЯЕТСЯ. Например:

```
std::atomic<int> integer(0);
std::atomic<int> otherInteger(0);
integer++;    //Атомарно
otherInteger += ++integer;   //Не атомарно!
```

Атомарные пользовательские типы

- Если размер пользовательского типа <= sizeof(int??? size_t), то в большинстве случаев компилятор может сгенерировать код, состоящий из атомарных операций
- если размер > sizeof(int), то в большинстве случаев эмуляция атомарности (=> выигрыша по сравнению с использованием мьютекса можно не получить)
- ограничения на пользовательские типы:
  - тривиальные конструктор копирования и operator= (которые реализуются компилятором автоматически посредством побитового копирования - memcpy),
  - сравнение должно осуществляться посредством побитового сравнения – memcmp()
  - нет виртуальных функций и виртуальных базовых классов

Пример атомарного пользовательского типа

```
class Test {
   int a, b;
public:
   Test(int _a=0, int _b=0)noexcept :a(_a), b(_b) {}
};

std::atomic<Test> t(Test(1, 2));
```

Специфика:

```
class Test {
   int a, b;
public:
   Test(int _a=0, int _b=0)noexcept :a(_a), b(_b) {}
   Test(const Test& other) { a = other.a; b = other.b; };
};

int main(){
   Test   t1(1, 2),  t2(3, 4);
   std::atomic<Test> a1(Test(1, 2));  //ошибка - atomic<T> requires T to be
                                      trivially copyable
}
```

Специфика:

```
class Test {
  int a, b;
public:
  Test(int _a=0, int _b=0)noexcept :a(_a), b(_b) {}
  bool operator==(const Test& right) { return a == right.a && b == right.b; }
};

int main(){
  Test   t1(1, 2),  t2(3, 4);
  std::atomic<Test> a1(Test(1, 2));  //OK
  bool b = a1.compare_exchange_strong(t1,t2); //operator==  для сравнения не
                                                вызывается
}
```

std::atomic<float>
std::atomic<double>

- формально разрешены, так как удовлетворяют тем же ограничениям, которым должны следовать пользовательские типы (побитовое копирование и сравнение)
- но! могут иметь разное внутреннее представление ??? => при сравнении равных значений можно получить false
- не определены атомарные арифметические операции

Memory order

ПОРЯДОК ИСПОЛНЕНИЯ

Порядок исполнения

- До C++11 – правила для однопоточного приложения:
  - зависимые друг от друга вычисления выполняются в предусмотренном программистом порядке,
  - остальные вычисления могут выполняться в том порядке, который компилятор + процессор считает оптимальным
- В многопоточном приложении может возникать необходимость упорядочить в одном потоке исполнение вычислений
  => *порядок изменения(**ПИ**)* учитывает наличие потоков, и является глобальным по отношению к ним, т.е. *ПИ* выстраивает порядок не относительно какого-либо потока а является общим для всех потоков, в котором участвуют **атомарные объекты данного *ПИ***. *ПИ* имеет отношение лишь к атомарным объектам, следовательно он влияет только на порядок вычисления выражений, в которых вовлечены атомарные объекты

Пример неопределенного порядка исполнения

```
void f(int a, int b){std::cout<<a<<‘ ‘<<b;}
int get(){
   static int val=0;
   return ++val;
}
int main(){
   f( get(), get() ); //порядок вызова не определен => ???
}
```

До C++11 - точка следования (sequence point)

- точка программы, в которой все побочные эффекты от предыдущих вычислений должны завершиться, а от последующих еще не начаться
- C++11 - sequence point заменено на:
  - **sequenced before**
  - sequenced after
  - indeterminately sequenced (один из before/after – неизвестно, какой конкретно)
  - unsequenced

- C++11 - появилось отношение «Synchronized with»

Специфика:

- Отношение – «**synchronized with**» возможно только между операциями над **атомарными** типами!
- Отношения – «**sequenced before/ sequenced after**» характеризуют, каким образом неатомарные операции группируются вокруг атомарных в одном потоке

Демонстрация отношений

(только два потока)

std::vector<int> data;
std::atomic<bool> ready_flag(false);

```
void writer(){
//запись
  data.push_back(33);
  ready_flag.store( true);
}
```

```
void reader(){
 while(!ready_flag.load()){…}
 //данные готовы =>чтение
 std::cout<<data.back();
}
```

В результате чтение происходит гарантированно после записи!
=> принудительное упорядочение (пока по умолчанию) !

БАРЬЕРЫ

Средства принудительного упорядочения выполнения:

- **compiler barrier** - запрет компилятору на переупорядочение (это просто указание компилятору не переставлять генерируемые инструкции «за барьер» или наоборот)
- **memory barrier** - запретить переупорядочение инструкций процессором, то есть на момент «прохождения» барьера:
  - заставляет выполнится весь конвейер до барьера => все предшествующие барьеру операции должны быть завершены
  - сброс предвыборки (PrefetchFlush) - благодаря чему следующие за барьером инструкции будут заново выбраны и декодированы из памяти (или кэша???).

Барьеры компилятору

Для запрета переупорядочения кода существует универсальный метод — установка барьера компилятору

- Барьеры приводят к частичному упорядочиванию операций доступа к памяти по обе (или по указанную) стороны барьера.
- Переупорядочение инструкций возможно только до барьера или только после барьера, но не через барьер!
- Барьер ничего не блокирует – просто препятствует оптимизации
- Существует несколько видов барьеров памяти: полный, release fence и acquire fence.

Характеристики барьеров:

- **Полный барьер** гарантирует, что все чтения и записи расположенные до/после барьера будут выполнены также до/после барьера, то есть никакая инструкция обращения к памяти не может «перепрыгнуть» барьер.
- **Acqire fence (полубарьер)** гарантирует что инструкции, стоящие после барьера, не будут перемещены до барьера.
- **Release fence (полубарьер)** гарантирует, что инструкции, стоящие до барьера, не будут перемещены после барьера.

Барьеры памяти

- для корректного выполнения параллельного кода **процессору** необходимо «подсказывать», до каких пределов ему разрешено проводить свои внутренние оптимизации чтения/записи.
- Эти подсказки – барьеры памяти. Барьеры памяти позволяют в той или иной мере упорядочить обращения к памяти (точнее, кэшу, — процессор взаимодействует с внешним миром только через кэш).
- Степень упорядочения может быть разной, — каждая архитектура может предоставлять целый набор барьеров “на выбор”. Используя те или иные барьеры памяти, мы можем построить разные модели памяти — набор гарантий, которые будут выполняться для наших программ.

Барьер памяти

- Процессоры и другие системные устройства используют множество приёмов для повышения производительности, в их числе переупорядочивание операций, откладывание и совмещение операций доступа к памяти, раннее чтение данных, предсказание переходов и различные типы кеширования. **Барьеры доступа к памяти** служат для подавления этих механизмов.

Демонстрация использования барьеров. Псевдокод:

```
void executedOnCpu0() {
   value = 10;
   storeMemoryBarrier();
   finished = true;
}

void executedOnCpu1() {
   while(!finished);
   loadMemoryBarrier();
   assert (value == 10);
}
```

УПОРЯДОЧЕНИЕ ДОСТУПА К ПАМЯТИ ДЛЯ АТОМАРНЫХ ОПЕРАЦИЙ

Напоминание. Важно!

Основное время при выполнении программы уходит на **обращения к памяти =>**

- на вычисления – мало!
- на чтение/запись – много!

=> компилятор оптимизирует низкоуровневый код, а схемотехники оптимизируют все действия, связанные с чтением/запись

=> порядок выполнения программы не определен (гарантируется только корректное получения результата в пределах одного потока)

Средства C++11 управления упорядочением

- Для задания компилятору правил упорядочения вокруг действий с атомарными переменными в большинстве методов и соответствующих глобальных функций-аналогов вводится дополнительный параметр **std::memory_order**
  Замечание: при этом барьер устанавливается внутри функции (switch)
- + функции для «ручной» установки барьеров:
  std::atomic_thread_fence(<порядок>**) –** барьер памяти
  std::atomic_signal_fence(<порядок>**) –** барьер компилятора

enum std::memory_order

предписание компилятору (опосредованно процессору):

- разрешается ему или нет переупорядочивать (reoder) другие (неатомарные) обращения к памяти до и/или после атомарной операции

Упорядочение доступа к памяти для атомарных операций

Варианты задаются константами memory_order:

- memory_order_**relaxed**
- memory_order_**consume**
- memory_order_**acquire**
- memory_order_**release**
- memory_order_**acq_rel**
- **memory_order_seq_cst** - умолчание

Шесть вариантов представляют три модели (разная эффективность, зависят от архитектуры):

- **sequentially consistent** - последовательно согласованное упорядочение (глобальное) - memory_order_seq_cst
- **acquire-release** - упорядочение захват/освобождение (между парами потоков) - memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel
- ослабленное упорядочение - memory_order_**relaxed**

Замечание:

- три модели упорядочения обычно влекут за собой различные издержки для процессоров с разной архитектурой => разная эффективность => в общем случае
  - захват/освобождение «дешевле», чем последовательно согласованное
  - а ослабленное «дешевле», чем захват/освобождение
- последовательно согласованное упорядочение
  - интуитивно понятнее
  - проще в использовании
- Важно! наличие разных моделей упорядочения доступа к памяти позволяет **эксперту** добиться повышения производительности за счет более точного управления отношениями упорядочения (по сравнению с использованием последовательно согласованного упорядочения)

memory_order_seq_cst – принудительное последовательно согласованное упорядочение (по умолчанию)

- если все операции над экземпляром **атомарного типа** последовательно (принудительно) согласованы, то поведение многопоточной программы в целом такое же, как если бы эти операции выполнялись в определенной последовательности в одном потоке
- упорядочение глобальное, то есть ВСЕ потоки «видят» один и тот же порядок операций изменения разделяемых данных
- гарантирует: на момент чтения атомарных данных в одном потоке
  - все действия, предшествующие записи в другом потоке, должны быть выполнены!
  - все действия в данном потоке, следующие за чтением, еще не выполнены

Важно!

- если в одном потоке задано последовательно согласованное сохранение данных (запись)
- то чтение в другом потоке должно быть тоже последовательно согласовано (если задан ослабленный порядок, то никаких гарантий нет)

=> Для «**глобального**» упорядочения при доступе к одним и тем же разделяемым данным (memory location) в разных потоках требуется использовать последовательно согласованное упорядочение

В противном случае

- В отсутствие явных ограничений на упорядочение кэши различных процессоров и внутренние буферы могут содержать различные значения для одной и той же memory location

Производительность

Последовательно согласованное упорядочение в **некоторых** системах реализуется посредством дополнительных команд синхронизации => может привести к потерям производительности системы в целом

Замечание:

некоторые архитектуры (x86, x86-64) обеспечивают последовательную согласованность с относительно низкими издержками => прежде, чем отказываться от этого средства, смотри документацию по конкретному процессору!

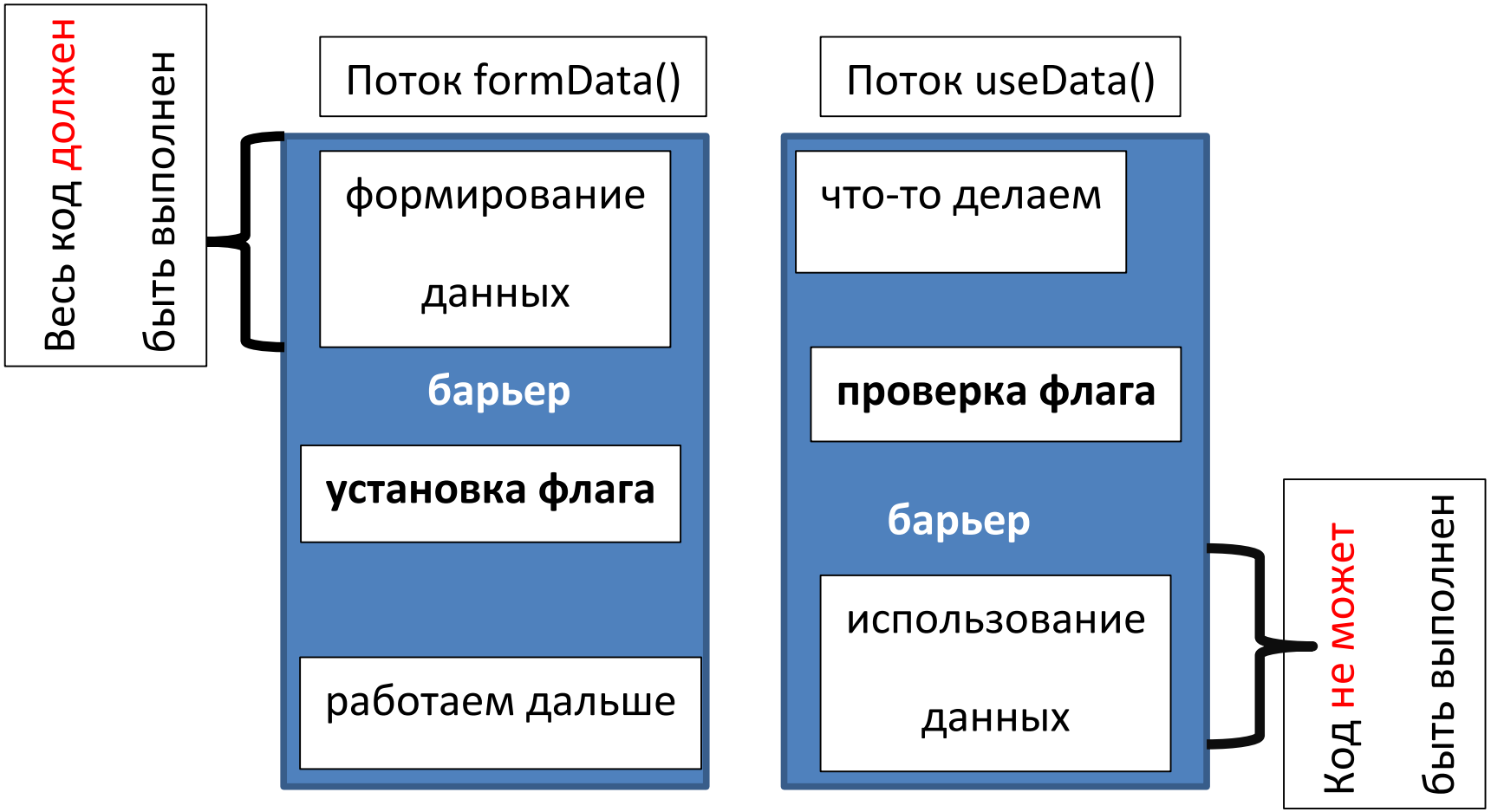
Проблемы?

int valGlobal;
bool readyFlag = false;

```
//поток, формирующий данные
void formData()
{
   //…
    valGlobal = 33;
    readyFlag=true;
  //…
}
```

```
//поток, обрабатывающий данные
void useData()
{
    while (!readyFlag)
    {  /*что-то делаем*/   }
    assert(valGlobal==33);
   //…
}
```

Требуется:


Весь код должен быть выполнен
Поток formData()
формирование данных
барьер
установка флага
работаем дальше
Поток useData()
что-то делаем
проверка флага
барьер
использование данных
Код не может быть выполнен

Пример – синхронизация и принудительное упорядочение

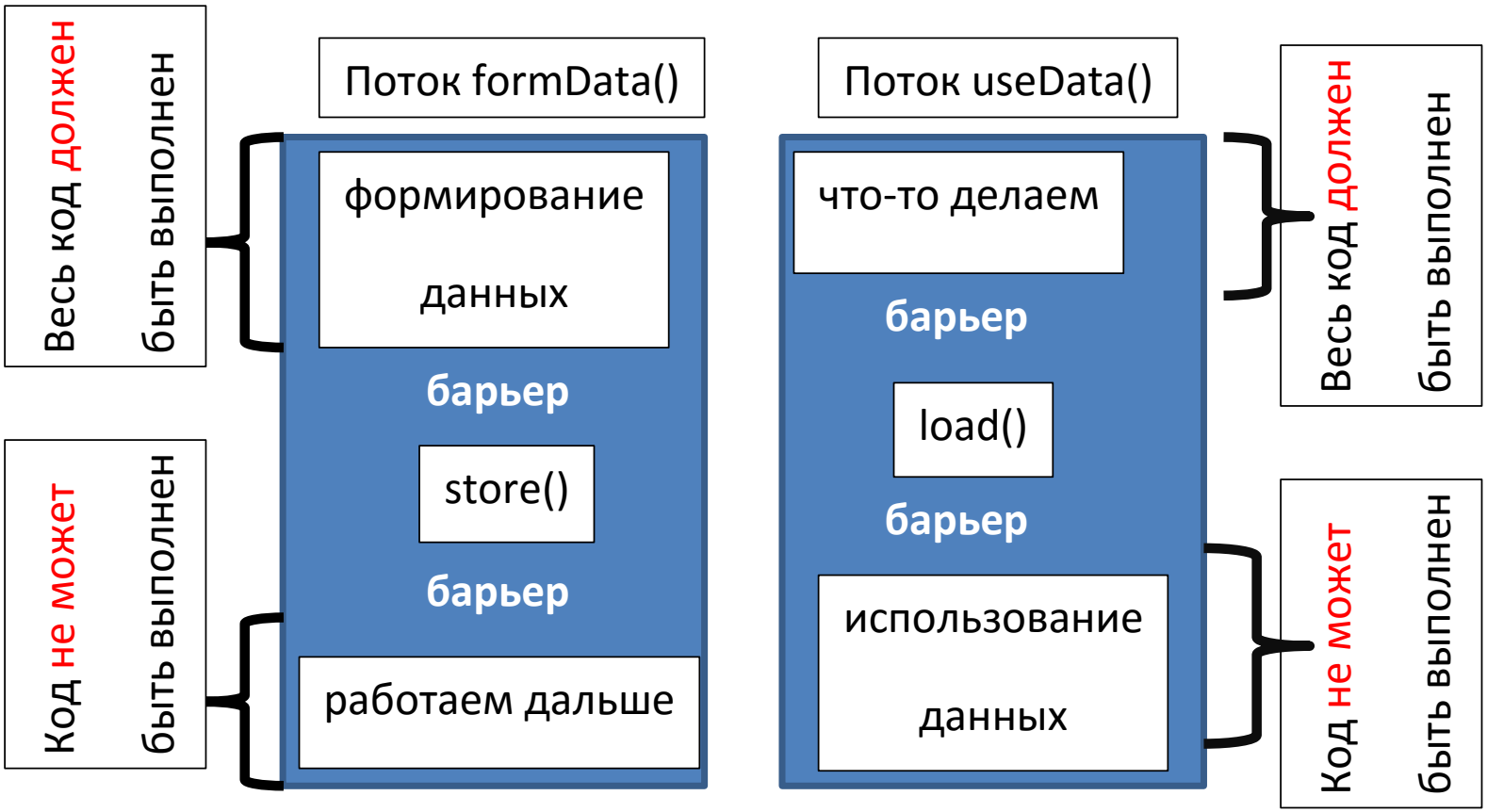
int valGlobal; //неатомарные данные

std::atomic<bool> readyFlag(false);

```
//поток, формирующий данные
void formData()
{
   //…
    valGlobal = 33;
    readyFlag.store(true,
          std::memory_order_seq_cst);
  //…
}
```

```
//поток, обрабатывающий данные
void useData()
{
    while (!readyFlag.load(
std::memory_order_seq_cst))
    {  /*что-то делаем*/   }
    assert(valGlobal==33);
   //…
}
```

А на самом деле ограничения гораздо сильнее



Пример:

```
std::atomic<bool> x(false),y(false);
std::atomic<int> z = {0};

void write_x() {x.store(true, std::memory_order_seq_cst);}
void write_y(){y.store(true, std::memory_order_seq_cst);}

void read_x_then_y(){
  while (!x.load(std::memory_order_seq_cst)){std::cout << 'x';}
  if (y.load(std::memory_order_seq_cst)) {++z;}
}

void read_y_then_x(){
  while (!y.load(std::memory_order_seq_cst)){ std::cout << 'y'; }
  if (x.load(std::memory_order_seq_cst)) { ++z; }
}
```

Продолжение

```
int main(){
        std::thread a(write_x);
        std::thread b(write_y);
        std::thread c(read_x_then_y);
        std::thread d(read_y_then_x);

        a.join(); b.join(); c.join();  d.join();

        // z = ???
}
```

memory_order_relaxed

- нет ограничений на переупорядочение неатомарных операций вокруг атомарной в одном потоке =>
  - эффективность высокая
  - но используется только тогда, когда упорядочение действительно не требуется или упорядочение программист обеспечивает «вручную» - std::atomic_thread_fence()
- единственное ограничение – операции доступа **к одной и той же атомарной переменной** внутри потока переупорядочивать нельзя!

Специфика memory_order_relaxed

- обеспечивается только атомарность выполнения
- операции в этом режиме не поддерживают отношение «синхронизируется с»
- для атомарной relaxed-записи стандартом запрещена спекулятивная запись
- ограничение: атомарные операции доступа над **одним и тем же объектом** в данном потоке нельзя переупорядочить

Хороший пример слабого упорядочения

std::atomic<int> atomicCount(0);	

//Поток 1
 	
…
atomicCount. fetch_add(1, std::memory_order_relaxed);
…
	//Поток 2
	
	…
	atomicCount. fetch_add(1, std::memory_order_relaxed);
	…

Пример:

```
std::atomic<bool> x(false),y(false);
std::atomic<int> z = {0};


void write_x_then_y(){
   x.store(true,std::memory_order_relaxed);
   y.store(true,std::memory_order_relaxed);
}

void read_y_then_x(){
  while (!y.load(std::memory_order_relaxed)){std::cout << 'y'; }
   if (x.load(std::memory_order_relaxed)) { ++z; }
}
```

Продолжение

```
int main()
{
        std::thread a(write_x_then_y);
        std::thread b(read_y_then_x);
        a.join(); b.join();

        // z = ???
}
```

Упорядочение захват/освобождение acquire/release

- атомарные операции load() – захват (acquire)
- атомарные операции store() - освобождение (release)
- атомарные операции exchange() – и захват, и освобождение

Модель захват-освобождение acquire - release

- в отличие от слабого упорядочения предоставляет **некоторую** попарную синхронизацию между потоком, захватившим ресурс, и потоком, освободившим ресурс
- операции load() – захват (memory_order_acquire) – чтение из памяти,
  операции store() – освобождение (memory_order_release) – запись в память
- атомарные операции чтение-модификация-запись (memory_order_acq_rel) – exchange(), fetch_add()… -

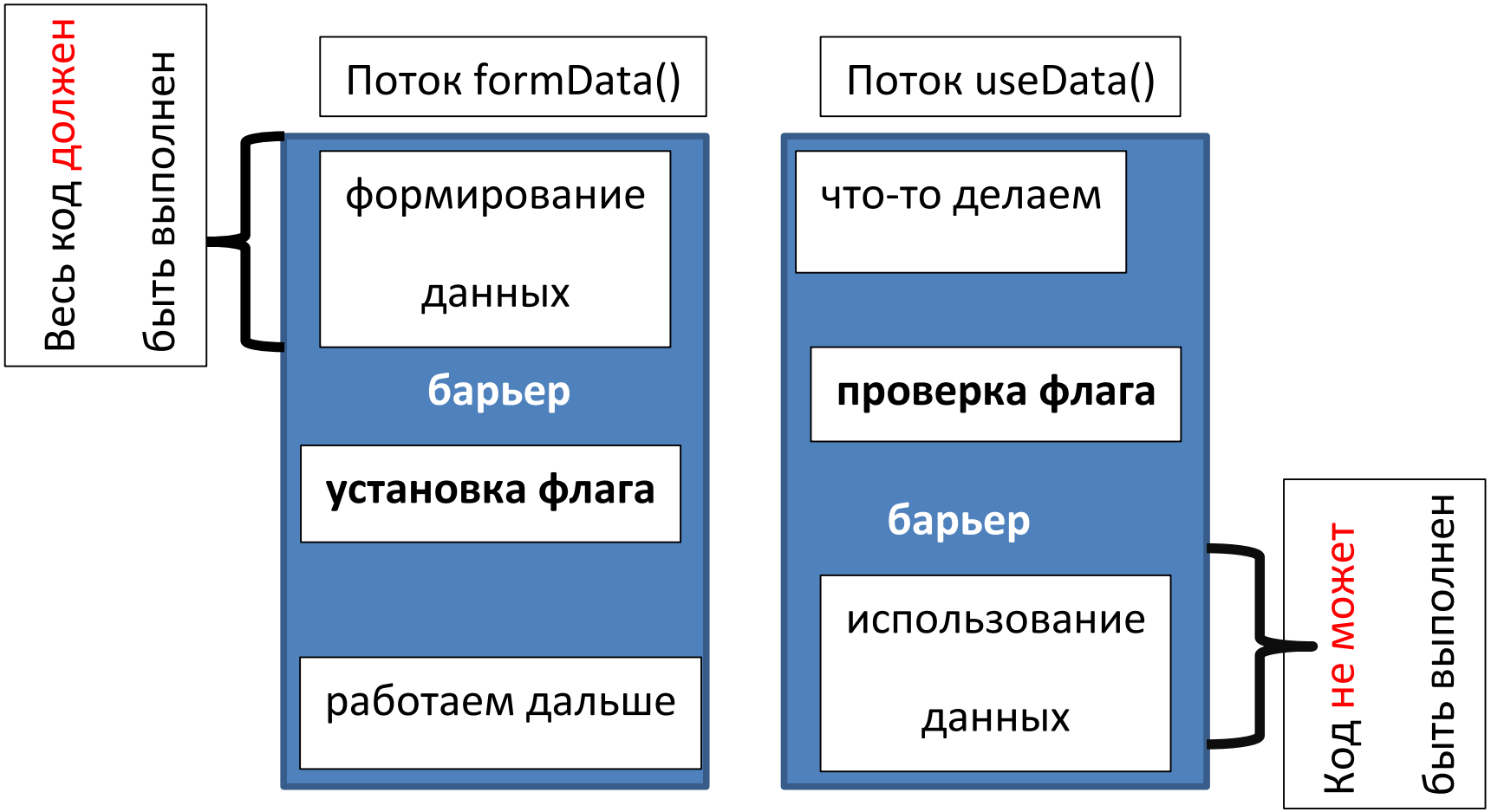
Отличия от memory_order__seq_cst

- memory_order__seq_cst – глобальное упорядочение
- memory_order_acquire,
  memory_order_release,
  memory_order_acq_rel
  попарное упорядочение между двумя потоками

Правила *acquire| release*:

- предшествующие операции записи не могут выполняться после *store() – установка барьера записи (release)!*
- следующие операции чтения не могут выполниться до *load() – установка барьера чтения (acquire)!*

Модифицируем пример:



Пример – *acquire| release*

int valGlobal; //неатомарные данные
std::atomic<bool> readyFlag(false);

```
//поток, формирующий данные
void formData()
{
   //…
    valGlobal = 33;
    readyFlag.store(true,
          std::memory_order_release);
  //…
}
```

```
//поток, обрабатывающий данные
void useData()
{
    while (!readyFlag.load(
std::memory_order_acquire))
    {  /*что-то делаем*/   }
    assert(valGlobal==33);
   //…
}
```

Пример посложнее

```
std::atomic<bool> x(false),y(false);
std::atomic<int> z = {0};

void write_x() {…x.store(true, std::memory_order_release);…}
void write_y(){…y.store(true, std::memory_order_release);…}

void read_x_then_y(){
  while (!x.load(std::memory_order_aquire)){std::cout << 'x';}
  if (y.load(std::memory_order_aquire)) {++z;}
}

void read_y_then_x(){
  while (!y.load(std::memory_order_aquire)){ std::cout << 'y'; }
  if (x.load(std::memory_order_aquire)) { ++z; }
}
```

Продолжение ???

```
int main()
{
        std::thread a(write_x);
        std::thread b(write_y);
        std::thread c(read_x_then_y);
        std::thread d(read_y_then_x);
        a.join(); b.join(); c.join();   d.join();

        // z = ???
}
```

std::memory_order_consume

- специализированная разновидность **acquire**-release семантики
- вводит новые отношения упорядочения – «по данным»:
  - предшествует-по-зависимости (dependency-ordered-before)
  - переносит-зависимость-в (carries-a-dependency-to)
- самое важное использование – атомарная загрузка (load()) указателя

Использование std::memory_order_consume с указателями

std::atomic<int*> pVal;
int val;

```
void write_val(){
val = 33;
pVal.store(&val,
   std::memory_order_release);
}
```

```
void read_val(){
int* p;
while ( !(p=pVal.load(
   std::memory_order_consume)) ) {}
int n = *p; //гарантирует: read/write операции с «указываемыми» данными не будут переупорядочены перед load. Для всех остальных данных переупорядочение возможно
}
```

Пример std::memory_order_consume поинтереснее

```
struct X {
int i;
std::string s;
};
//глобальные данные
std::atomic<X*> p;
std::atomic<int> a;
```

Продолжение примера std::memory_order_consume

```
void create_x(){
  X* x = new X;
  x->i = 33;
  x->s = "abc";
  a.store(1, std::memory_order_relaxed);
  p.store(x, std::memory_order_release);
}
void use_x(){
  X* x=nullptr;
  while(!(x=p.load(std::memory_order_consume))){}
  bool b1 = x->i == 33;   //гарантировано true
  bool b2 = x->s == "abc";   //гарантировано true
  int n = a.load(std::memory_order_relaxed); // a.load() может быть
                                              переупорядочено до p.load()
  bool b3 = n == 1; //никаких гарантий
}
```

STD::ATOMIC_THREAD_FENCE()

Барьеры можно ставить «вручную» std::atomic_thread_fence()

- C++11
  atomic_thread_fence()
- GCC
  __asm volatile("mfence" ::: "memory");
  //аппаратный барьер памяти - процессору
  __asm volatile("" ::: "memory"); //программный барьер памяти - компилятору

Пример:

```
data.store(3, std::memory_order_relaxed);
std::atomic_thread_fence(std::memory_order_release);
flag.store(1, std::memory_order_relaxed);
flag2.store(2, std::memory_order_relaxed);
```

не эквивалентно:

```
data.store(3, std::memory_order_relaxed);
flag.store(1, std::memory_order_release);
flag2.store(2, std::memory_order_relaxed);
```

так как flag2.store() может быть перемещен перед data.store()

ПРИМЕР LOCK-FREE СТЕКА

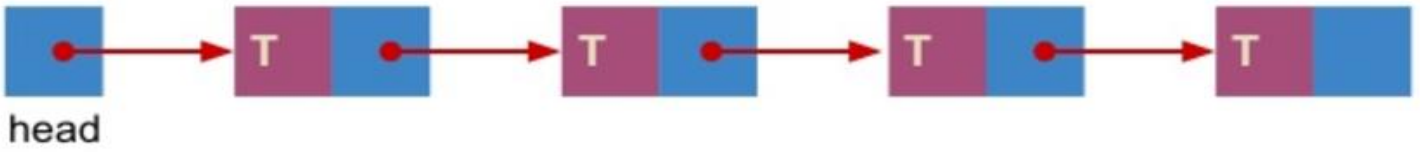
Реализация push() в однопоточной программе. Для многопоточной???

```
template<typename T> class Stack {
struct Node {
   T data;
   Node* pNext=nullptr;
  };
  Node* pHead = nullptr;
public:
  void push( const T& t)  {
     Node* const pNewNode = new Node(t);
     pNewNode->pNext = pHead;
     pHead = pNewNode
    }
};
```

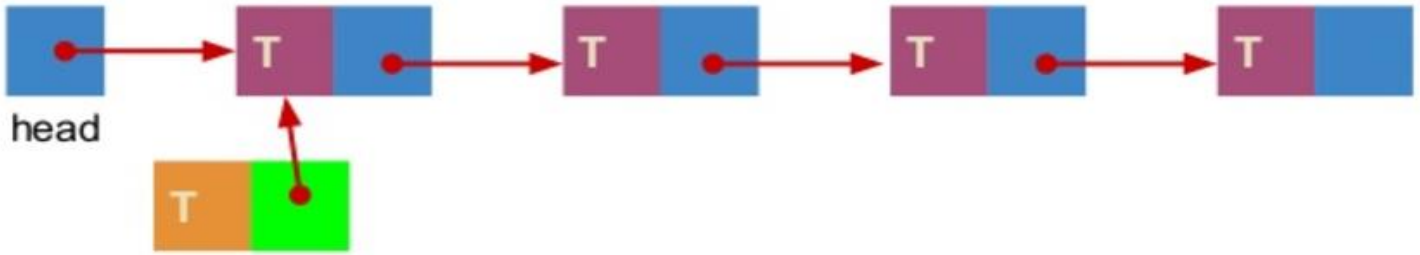
Функция push

Clip slide

Начальная стадия



Промежуточная стадия



Конечная стадия

Проблемы при разделении данных

```
void push( T t)  {
  //Пока не добавляем в список, все ОК
  Node* const pNewNode = new Node(std::move(t));
  pNewNode->pNext = pHead;  //неатомарная операция
 //в это время другой поток тоже может добавлять новый Node
                    (push) или удалять верхний (pop)
  pHead = pNewNode; // неатомарная операция
}
```

Решаем проблемы push()

???

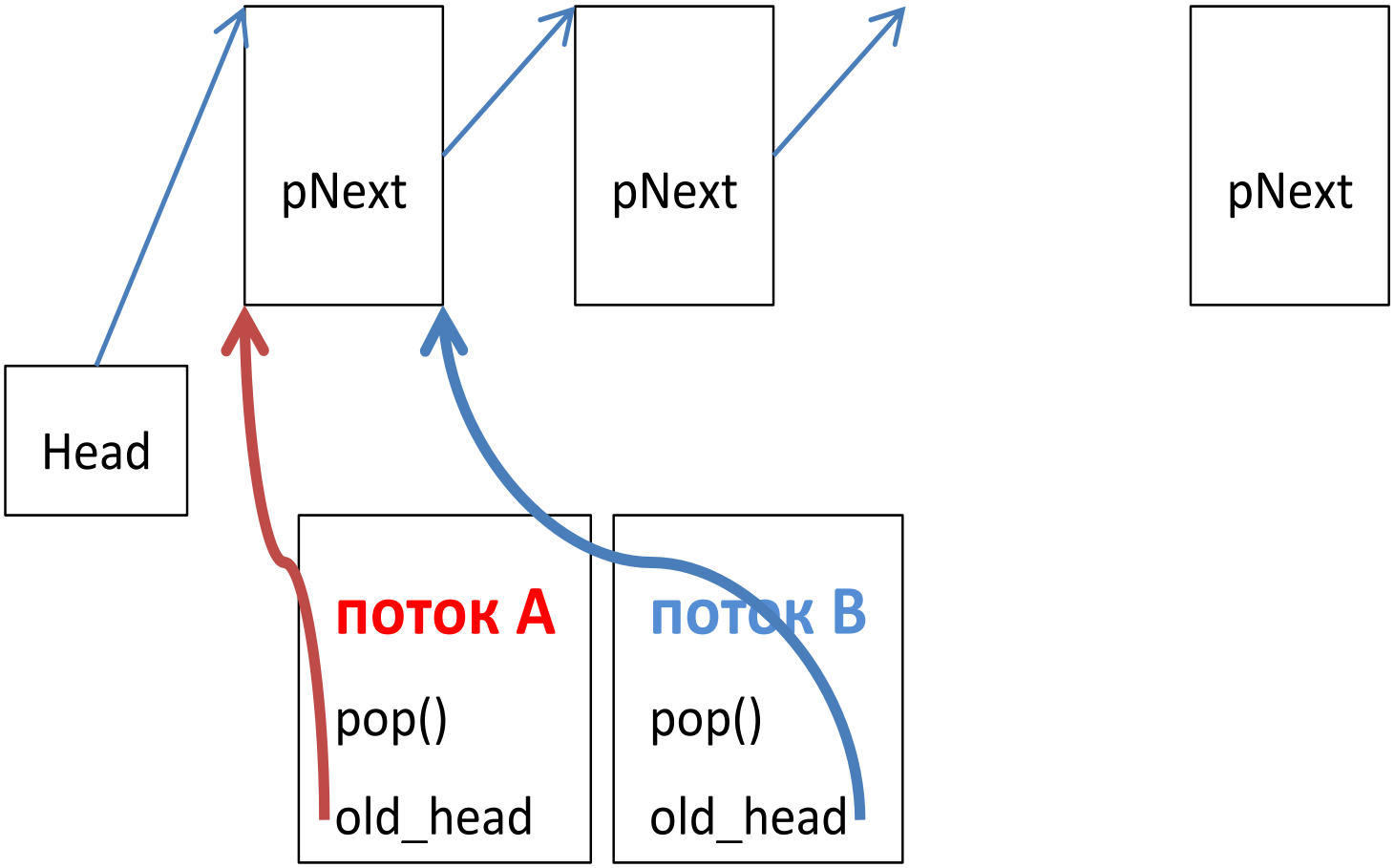
Решаем проблемы push()

```
template<typename T> class Stack {
  …
  std::atomic<Node*> Head = nullptr;
public:
  void push( T t)  {
  //подготовка нового узла
  Node* const pNewNode = new Node(std::move(t));
  pNewNode->pNext = Head.load(); //заносим текущий Head
  while (!Head.compare_exchange_strong(…, …) ){;}
//записать в Head pNewNode, НО! при этом гарантировать, что другой поток не успел модифицировать Head
}
```

Однопоточная версия pop(). Проблемы для многопоточного использования?

```
void pop(T& res)
{
        Node* old_head = Head.load();
        if(old_head)
        {
                res = old_head->data;
                Head = Head->pNext;
                delete old_head;
        }
}
```

Иллюстрация проблемы



Опасная реализация pop()

```
void pop(T& res)
{
  Node* old_head = Head.load();
  while(old_head &&
                !Head.compare_exchange_weak
                (…, …)){;}
  res = //безопасно только если Node уже удален из стека
        old_head ? old_head->data : ???; //что делать?
  delete old_head; //если другой поток хранит этот указатель?
}
```

Решаем проблемы с возвратом данных из pop()

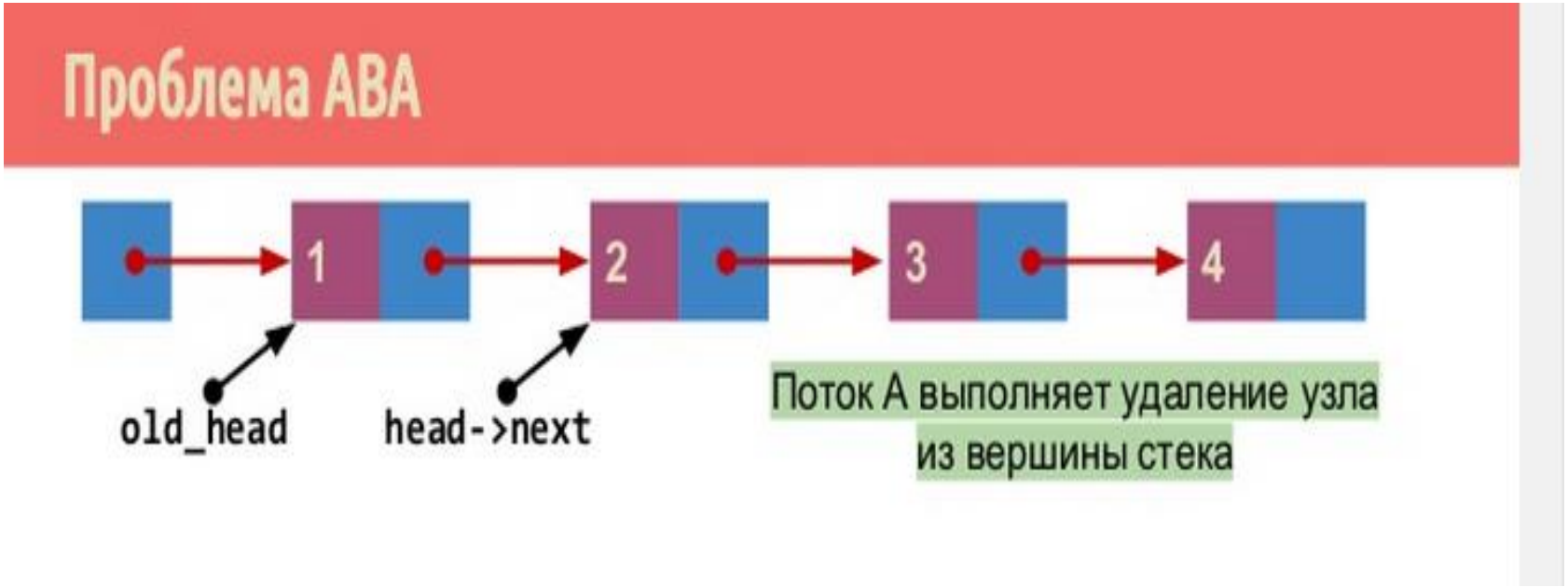
```
struct Node {
   std::shared_ptr<T> data;
   Node* pNext=nullptr;
…
  };

 std::shared_ptr<T> pop(){
  Node* old_head = Head.load();
  while(old_head &&
                        !Head.compare_exchange_weak
                        (…, …)){;}
 std::shared_ptr<T> res =  old_head ? old_head->data : std::shared_ptr<T> ();
  delete old_head; //если другой поток хранит этот указатель?
 return res;
}
```

Удаление узла

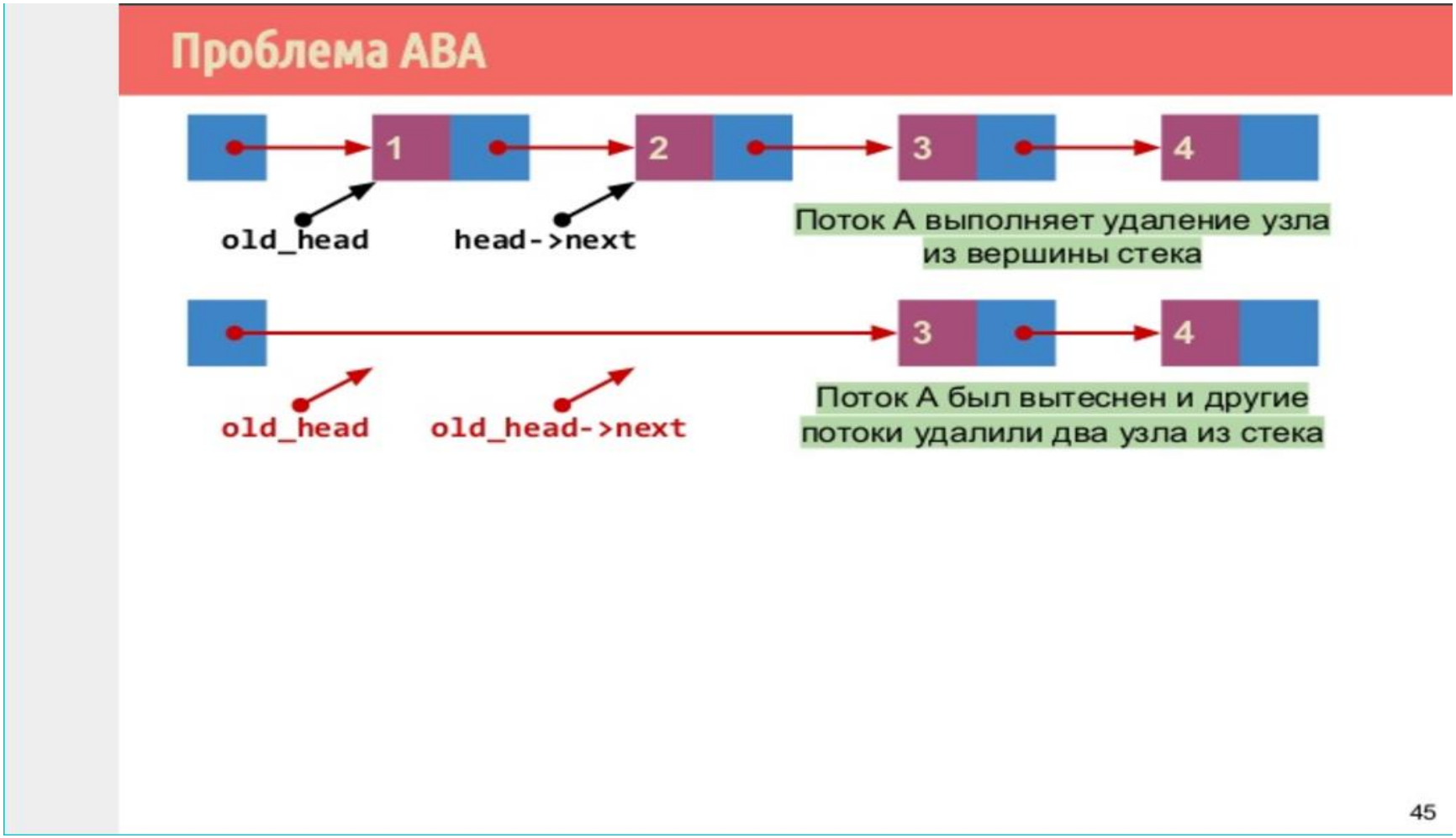
- Внутри метода pop() к одному и тому же узлу могут иметь доступ несколько потоков
- => если несколько потоков одновременно могут вызвать pop(), требуется обеспечить реальное удаление только в том случае, когда узлом ни один поток не пользуется
- => реализовать сборщик мусора узлов

Замечание: метод push не обращается к уже существующим узлам => защиту нужно ставить только в pop()

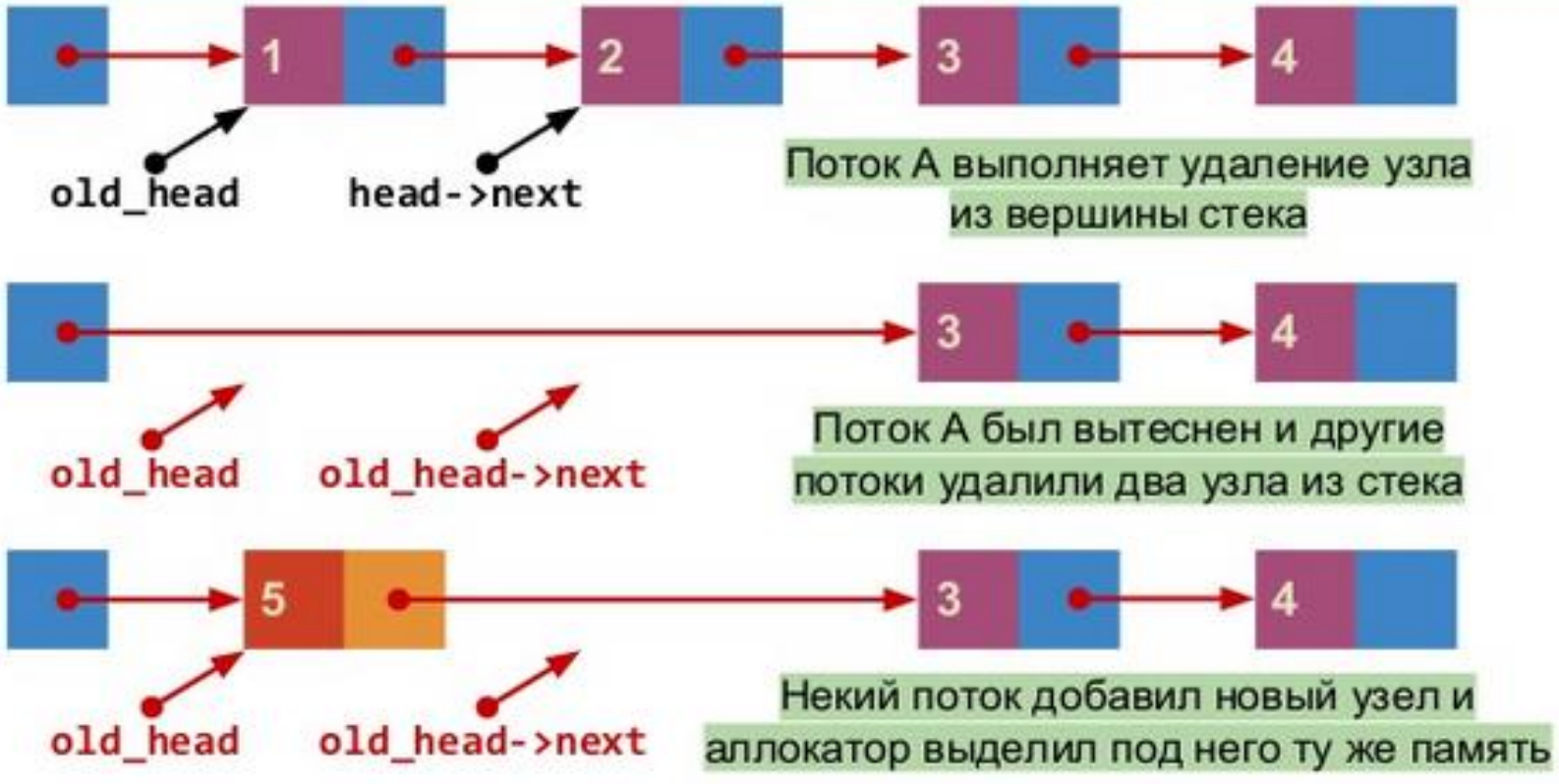
Проблема ABA



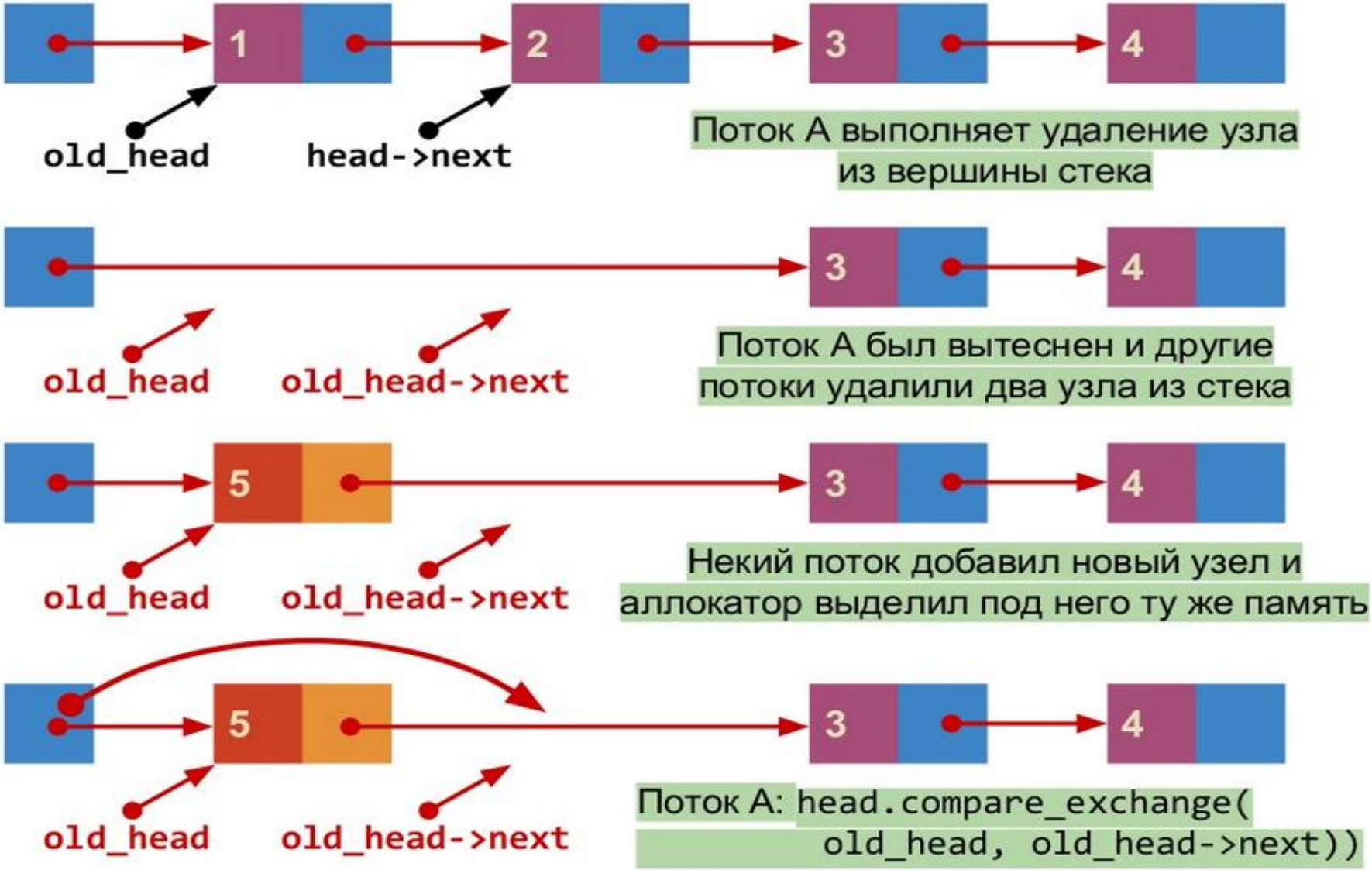
Продолжение



Продолжение



Финал


1
2
3
4
old_head
head->next
Поток A выполняет удаление узла из вершины стека
3
4
old_head
old_head->next
Поток A был вытеснен и другие потоки удалили два узла из стека
5
3
4
old_head
old_head->next
Некий поток добавил новый узел и аллокатор выделил под него ту же память
5
3
4
old_head
old_head->next
Поток A: head.compare_exchange(old_head, old_head->next))

Решение (одно из возможных) – «ленивый» сборщик ненужных узлов

в методе pop():

- при входе «считаем» поток, вызвавший pop() «++», при выходе снова «считаем» - «--»
- после извлечения данных из узла, заносим узел в специальный список узлов, подлежащих удалению
- если счетчик потоков, вызывающих pop(), нулевой => узлы можно удалять