Объектно-ориентированное программирование

2023

Лямбда

01_ParameterFunction

```
using FuncType = int(int,int);

void PassingFunc(FuncType fn, int x, int y)
{
    std::cout << "Result = " << fn(x, y) << std::endl;
}
```

Lambda
02_Lambda

Лямбда-выражения в C++ — это краткая форма записи анонимных функторов.

Например:

```
[](int _n)   {    cout << _n << " ";}
```

Примерно соответствует:

```
class MyLambda
{
   public: void operator ()(int _x) const { cout << _x << " "; }
};
```

Лямбда функции могут возвращать значения 03_LambdaReturn

В случае, если в лямбда-функции только один оператор return то тип значения можно не указывать. Если несколько, то нужно явно указать.

```
[] (int i) -> double
    {
        if (i < 5)
            return i + 1.0;
        else if (i % 2 == 0)
            return i / 2.0;
        else
            return i * i;
    }
```

Lambda == Functor

```
[ captures ]   ( params ) -> ret   { statements; }
```

```
class functor {
private:
  CaptureTypes __captures;
public:
  __functor( CaptureTypes captures )

  auto operator() ( params ) -> ret
    { statements; }
```

Захват переменных из внешнего контекста

04_LambdaCapture

```
[]                      // без захвата переменных из внешней области видимости
[=]                     // все переменные захватываются по значению
[&]                     // все переменные захватываются по ссылке
[this]                  // захват текущего класса
[x, y]                  // захват x и y по значению
[&x, &y]                // захват x и y по ссылке
[in, &out]              // захват in по значению, а out — по ссылке
[=, &out1, &out2]       // захват всех переменных по значению, кроме out1 и out2,
                        // которые захватываются по ссылке
[&, x, &y]              // захват всех переменных по ссылке, кроме x…
```

Изменение объектов, захваченных по значению

Caller	Closure	C++11

	• a,b lives inside c	void caller()
		{
		int a = 42;
		int b = 15;
		auto c = [=]() mutable{a = 33;++b; int z = a + b; return z; };
		c();
		c();
		c();
		}

▪ a=42		
▪ b=15		
▪ a=42 b=15	• a=33 b=16 z=49	
▪ a=42 b=15	• a=33 b=17 z=50	
▪ a=42 b=15	• a=33 b=18 z=51	

Генерация лямбда-выражений 05_LambdaGen

Начиная со стандарта C++11 шаблонный класс std::function является полиморфной оберткой функций для общего использования. Объекты класса std::function могут хранить, копировать и вызывать произвольные вызываемые объекты - функции, лямбда-выражения, выражения связывания и другие функциональный объекты. Говоря в общем, в любом месте, где необходимо использовать указатель на функцию для её отложенного вызова, или для создания функции обратного вызова, вместо него может быть использован std::function, который предоставляет пользователю большую гибкость в реализации.

Определение класса

```
template<class> class function; // undefined

template<class R, class... ArgTypes>
class function<R(ArgTypes...)>;
```

C++11

```
auto lambda = [](int x, int y)
  {return x + y;}
struct unnamed_lambda {
  auto operator()(int x, int y)
   const {return x + y;}
};
```

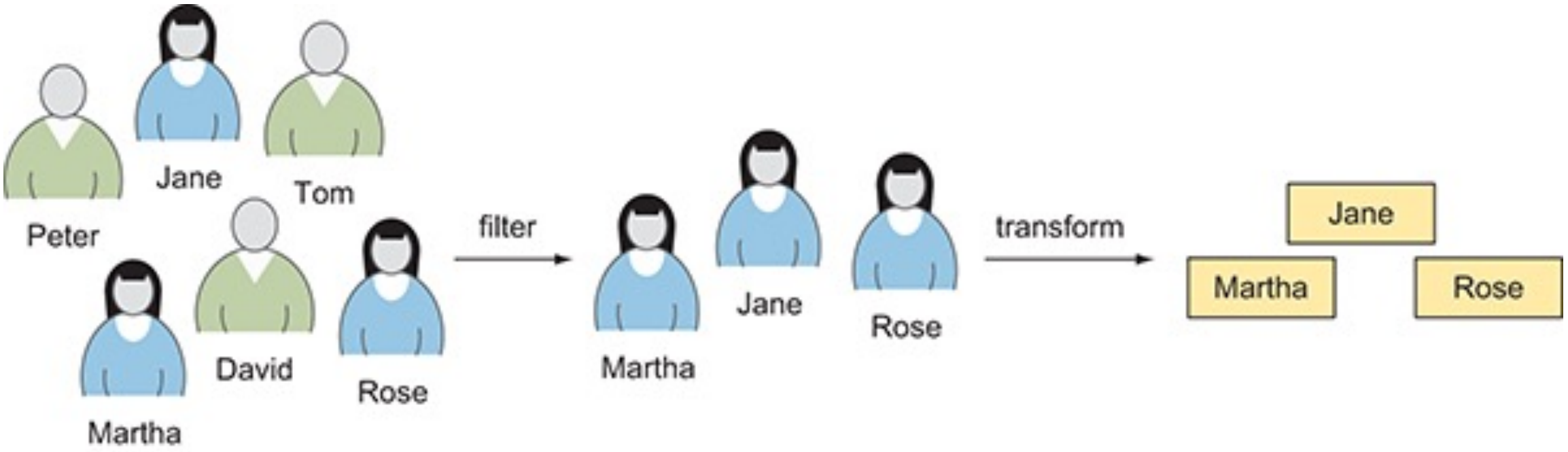
C++14

```
auto lambda = [](auto x,
  auto y) {return x + y;}
struct unnamed_lambda {
  template<typename T,
  typename U> auto
  operator()(T x, U y) const
  {return x + y;}
```

Лямбда в C++14

Лямбды + variadic template
06_LambdaVariadic

```
template <typename T, typename ...Ts>
auto concat(T t, Ts ...ts)
{
if constexpr (sizeof...(ts) > 0)
    return [=](auto ...parameters)
        {
         return
t(concat(ts...)(parameters...));
        };
  } else {return t;}
}
```



Пример 07_LambdaExample

for_each.cpp
#include <algorithm>

```
std::for_each(
              begin(vehicles),
              end(vehicles),
              [](const Vehicle &vehicle){
                  cout << vehicle.name << endl;
             });
```

transform.cpp
#include <algorithm>

```
      std::transform(
              std::begin(vect),
              std::end(vect),
              std::begin(vect2),
              [](int n) {
                return n * n;
            });
```

Мультипрограммирование

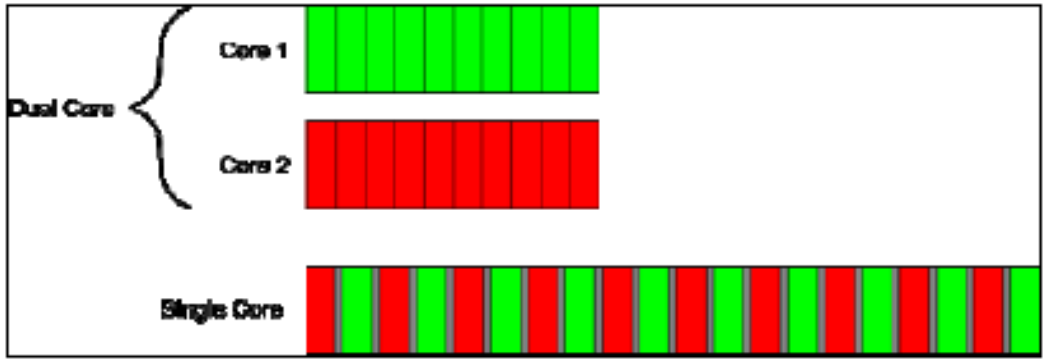
Процессы и потоки

- Программа (program) – это последовательность команд, реализующая алгоритм решения задачи.
- Процесс (process) – это программа (пользовательская или системная) в ходе выполнения.
- В современных операционных системах процесс представляет собой объект – структуру данных, содержащую информацию, необходимую для выполнения программы. Объект "Процесс" создается в момент запуска программы (например, пользователь дважды щелкает мышью на исполняемом файле) и уничтожается при завершении программы.
- Процесс может содержать один или несколько потоков (thread) – объектов, которым операционная система предоставляет процессорное время. Сам по себе процесс не выполняется – выполняются его потоки. Таким образом, машинные команды, записанные в исполняемом файле, выполняются на процессоре в составе потока. Если потоков несколько, они могут выполняться одновременно.

Основные понятия

- **Мультипроцессирование** - использование нескольких процессоров для одновременного выполнения задач.
- **Мультипрограммирование** - одновременное выполнение нескольких задач на одном или нескольких процессорах.

Мультипроцессирование vs Мультипрограммирование



Несколько вычислительных ядер процессора позволяют выполнять несколько задач одновременно.

Одно ядро процессора может выполнять несколько задач, только переключаясь между ними.

За чем применять многозадачность?

1. Разделение программы на независимые части. Один процесс выполняет одну задачу (например, взаимодействие с пользователем), а другой – другую (например, вычисления).
2. Для увеличения производительности.

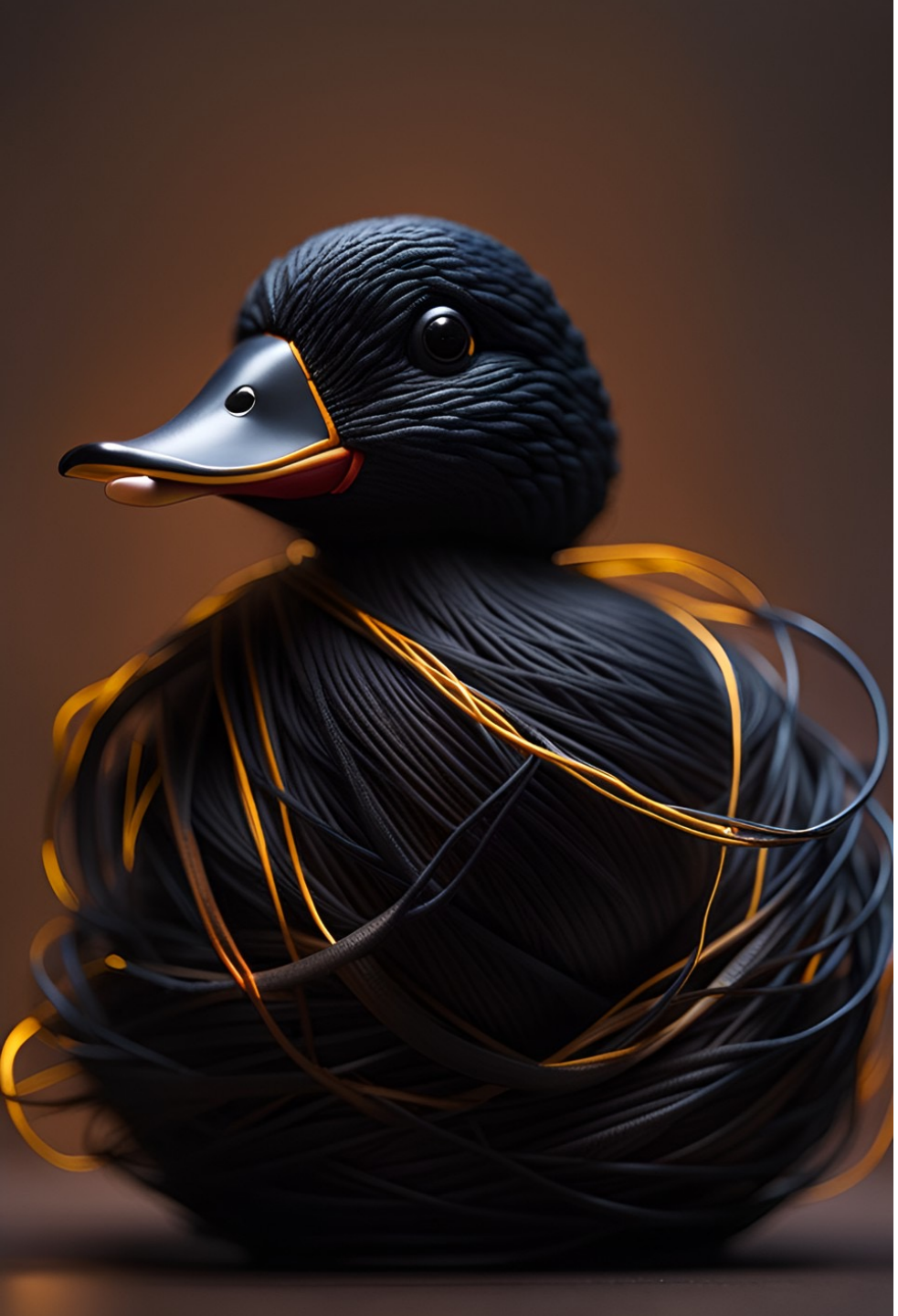
Увеличение числа параллельных процессов не всегда приводит к ускорению программы.

08_Thread

```
1.  #include <iostream>
2.  #include <thread>
3.  void hello() {
4.  std::cout<<"Hello Concurrent World\n";
5.  }

6.  int main(int argc,char * argv[]){
7.  std::thread t(hello); // launch new thread
8.  t.join(); //wait for finish of threads

9.  cin.get();
10. return 0;
11. }
```



std::thread
стандарт C++11

Конструктор

template <class Fn, class... Args> explicit thread (Fn&& fn, Args&&... args);

Принимает в качестве аргумента функтор и его параметры.

void join();

Ожидает когда выполнение потока закончится.

Как передать объект? 09_ThreadParameter

```
1.  #include <thread>
2.  #include <iostream>
3.  class MyClass{
4.  public:
5.      // thread вызовет переопределенный оператор
6.      void operator()(const char* param){
7.      std::cout << param << std::endl;
8.     }
9.  };
10. int main() {
11.     // передается копия объекта!
12.     std::thread my_thread(MyClass(),"Hello world!");
13.     my_thread.join();
14.     return 0;
15. }
```

Полезные функции std::this_thread

std::thread::hardware_concurrency()

Возвращает количество Thread которые могут выполняться параллельно для данного приложения.

std::this_thread::get_id()

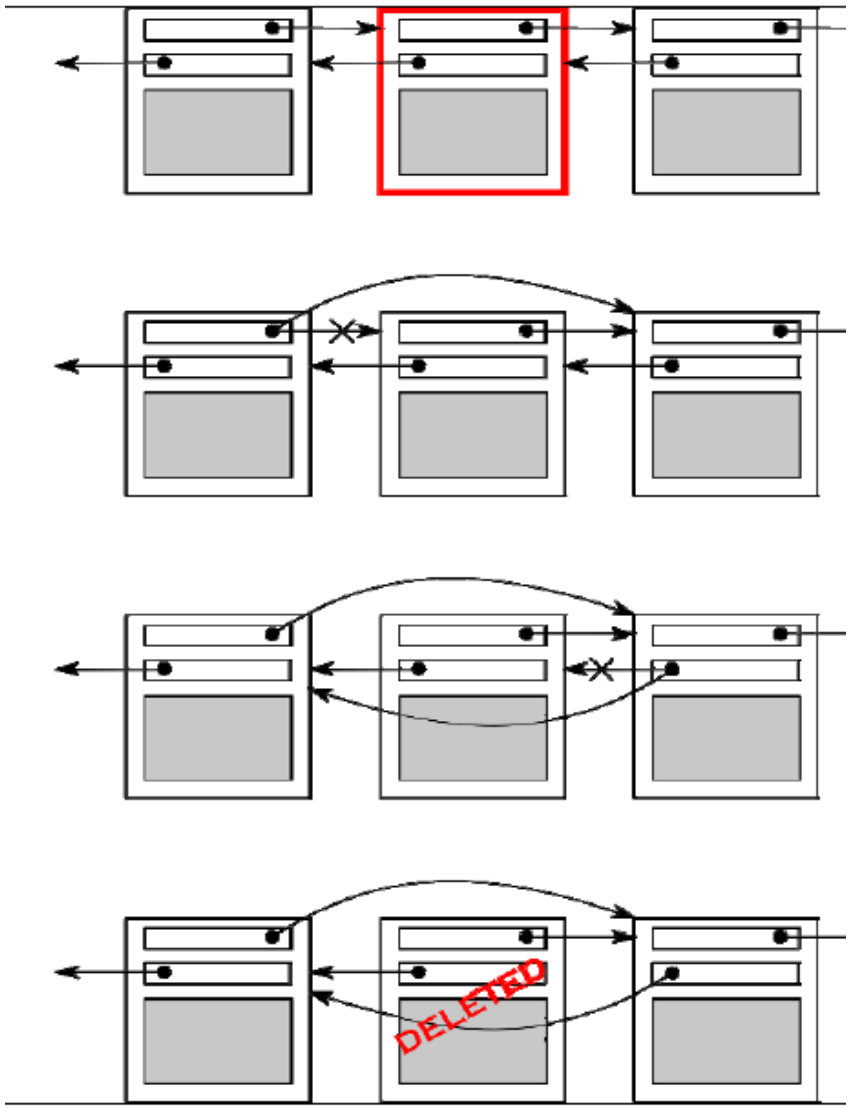
Возвращает идентификатор потока.

std::this_thread::sleep_for(std::chrono::milliseconds)

Позволяет усыпить поток на время

Как дождаться завершения потока красиво? 10_ScopedThread

```
class Scoped_Thread {
 std::thread t;
public:
 Scoped_Thread(std::thread&& t_) : t(std::move(t_)) {
  if (!t.joinable()) throw std::logic_error("No thread");
 };
 Scoped_Thread(std::thread& t_) : t(std::move(t_)) {
  if (!t.joinable()) throw std::logic_error("No thread");
 };
 Scoped_Thread(Scoped_Thread & other) : t(std::move(other.t)) { };
 Scoped_Thread(Scoped_Thread && other) : t(std::move(other.t)) {};
 Scoped_Thread& operator=(Scoped_Thread &&other) {
  t = std::move(other.t);       return *this;}
 ~Scoped_Thread() {
  if (t.joinable()) t.join();
  };
};
```



Проблемы работы с динамическими структурами данных в многопоточной среде

При удалении элемента из связанного списка производится несколько операций:

- удаление связи с предыдущим элементом
- удаление связи со следующим элементом
- удаление самого элемента списка

Во время выполнения этих операций к этими элементами обращаться из других потоков нельзя!

Потоковая безопасность

Свойство кода, предполагающее его корректное функционирование при одновременном исполнении несколькими потоками.

Основные методы достижения:

- Реентрабельность;
- Локальное хранилище потока;
- Взаимное исключение;
- Атомарные операции;

Реентерабельность

Свойство функции, предполагающее её корректное исполнение во время повторного вызова (например, рекурсивного).

- Не должна работать со статическими (глобальными данными);
- Не должна возвращать адрес статических (глобальных данных);
- Должна работать только с данными, переданными вызывающей стороной;
- Не должна модифицировать своего кода;
- Не должна вызывать нереентерабельных программ и процедур.

Потоконебезопасный код 11_RaceCondition

```
void add_function( long * number){
 for( long i=0;i<1000000000L;i++) (*number)++;}

void subst_function( long * number){
 for( long i=0;i<1000000000L;i++) (*number)--;}

int main() {
long number = 0;
{
 Scoped_Thread
th1(std::move(std::thread(add_function,&number)));
 Scoped_Thread
th2(std::move(std::thread(subst_function,&number)));
}
std::cout << "Result:" << number << std::endl;
return 0;
}
```

Состояния гонки

- проблема разделения объектов
- когда только читаем, проблем нет
- когда пишем, все может сломаться
- непредсказуемость момента переключения контекста

Взаимное исключение 12_Mutex

Мьютекс — базовый элемент синхронизации и в C++11 представлен в 4 формах в заголовочном файле <mutex>:

mutex
обеспечивает базовые функции lock() и unlock() и не блокируемый метод try_lock()

timed_mutex
в отличие от обычного мьютекса, имеет еще два метода: try_lock_for() и try_lock_until() позволяющих контролировать время ожидания вхождения в mutex

Рекурсивное вхождение Mutex 13_MutexRecursive

1. **recursive_mutex**
   может войти «сам в себя», т.е. Поддерживает многократный вызов lock из одного потока. Содержит все функции mutex.
2. **recursive_timed_mutex**
   это комбинация timed_mutex и recursive_mutex

RAII - lock_guard 14_LockGuard

1. Захват в конструкторе
2. Освобождение в деструкторе
3. Используются методы мьютексов
   - Захват: void lock();
   - Освободить: void unlock();

unique_lock

15_Deadlock

1. То же, что lock_guard
2. Используются методы мьютексов
   - Попытаться захватить: bool try_lock();
   - Захват с ограничением: void timed_lock(...);
3. + Дополнительные функции получения мьютекса, проверки «захваченности»...

16_SharedLock

В языке C++ std::shared_lock - это механизм, позволяющий нескольким потокам совместно использовать блокировку, но при этом только один поток может одновременно производить запись в общий ресурс. Это полезно в тех случаях, когда нагрузка на чтение велика и требуется разрешить нескольким потокам одновременное чтение из одного и того же ресурса.

```
std::lock_guard<std::mutex>
lock(a);
std::lock_guard<std::mutex>
lock(b);
```

```
std::lock_guard<std::mutex>
lock(b);
std::lock_guard<std::mutex>
lock(a);
```

Возникает когда несколько потоков пытаются получить доступ к нескольким ресурсам в разной последовательности.

Ловим таймауты что бы определить deadlock
17_DeadLock2

Потоко-безопасный Stack 18_Stack

Классы «обертки» позволяют непротиворечиво использовать мьютекс в RAII-стиле с автоматической блокировкой и разблокировкой в рамках одного блока. Эти классы:

lock_guard
когда объект создан, он пытается получить мьютекс (вызывая lock()), а когда объект уничтожен, он автоматически освобождает мьютекс (вызывая unlock())

Печать на экран это то же разделяемый ресурс – так что печать то же защищаем мьютексом.

Пример опасной ситуации pass_out.cpp

Передача данных из за границу lock.

Правило:

Ни когда не передавайте во вне указатели или ссылки на защищаемые данные. Это касается сохранение данных в видимой области памяти или передачи данных как параметра в пользовательскую функцию.

future + async
20_Future

future – служит для получения результата вычислений из другого потока.

Т.е. Функция выполняемая thread теперь может возвращать значение.

– это шаблон (параметр – тип возвращаемого значения)

– конструируется с помощью std::async

— результат выполнения получается методом get()

— async запускает поток и синхронизирует результат с возвращаемым future

```
T function() {

 return T();

}

int main (){

std::future<T> fut = std::async (function);

T x = fut.get();

}
```

std::future<T> ограничения

1. Нельзя вызывать сразу несколько get() из разных std::thread. Если нужно синхронизировать сразу несколько потоков – то лучше использовать условные переменные (будет рассмотрено далее).
2. Нельзя копировать (только передача по ссылке).

future работает на promise 21_Promise

template <class T> promise – шаблон который сохраняет значение типа T, которое может быть получено с помощью future

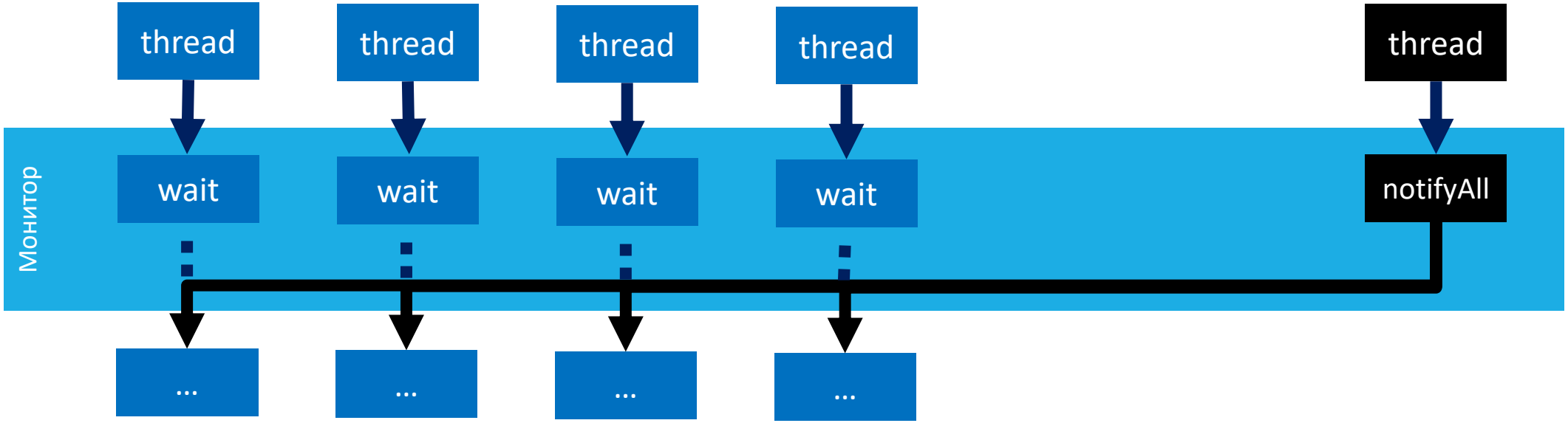
get_future – запрос future, связанного со значением внутри promise

set_value – устанавливает значение (и передает его в вызов get_future)

set_exception – передача исключения

Условная переменная

примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. Условные переменные используются вместе с ассоциированным мьютексом и являются элементом некоторых видов мониторов.



Условные переменные <condition_variable> 22_Conditional

требует от любого потока перед ожиданием сначала выполнить std::unique_lock

1. Должен быть хотя бы один поток, **ожидающий**, пока какое-то условие станет истинным. Ожидающий поток должен сначала выполнить unique_lock.
2. Должен быть хотя бы один поток, **сигнализирующий** о том, что условие стало истинным. Сигнал может быть послан с помощью notify_one(), при этом будет разблокирован один (любой) поток из ожидающих, или notify_all(), что разблокирует все ожидающие потоки.
3. В виду некоторых сложностей при создании пробуждающего условия, которое может быть предсказуемых в многопроцессорных системах, могут происходить **ложные пробуждения** (spurious wakeup). Это означает, что поток может быть пробужден, даже если никто не сигнализировал условной переменной. Поэтому необходимо еще проверять, верно ли условие пробуждение уже после то, как поток был пробужден.

std::condition_variable

- ожидание уведомления
- механизм синхронизации потоков
- довольно сложно использовать (мьютекс, цикл, условие)
- есть проблемы spurious wakeup / lost wakeup

Проблема блокировок

1. Взаимоблокировки (Deadlocks)
2. Надежность — вдруг владелец блокировки помрет?
3. Performance
   - Параллелизма в критической секции нет!
   - Владелец блокировки может быть вытеснен планировщиком

Закон Амдала

Джин Амдал (Gene Amdahl) - один из разработчиков всемирно известной системы IBM 360 в 1967 году предложил формулу, отражающую зависимость ускорения вычислений, достигаемого на многопроцессорной ВС, как от числа процессоров, так и от соотношения между последовательной и распараллеливаемой частями программы. Проблема рассматривалась Амдалом исходя из положения, что объем решаемой задачи (рабочая нагрузка - число выполняемых операций) с изменением числа процессоров, участвующих в ее решении, остается неизменным.

Пусть
f - доля операций, которые должны выполняться последовательно одним из процессоров и
1-f - доля, приходящаяся на распараллеливаемую часть программы.
Тогда ускорение, которое может быть получено на ВС из *n* процессоров, по сравнению с однопроцессорным решением не будет превышать величины:

S(n)=T(1)/T(n)=1/[f+(1-f)/n].

Например, если половина операций подлежит распараллеливанию на 4 машинах, то ускорение равно:

S(4) = 1/(0.5+0.5/4)=1.6 т.е. Только в полтора раза!

Итого

- ✓ Многопоточность
- ✓ Параллельность
- ■ Асинхронность

Итого: Многопоточность

возможность операционной системы (системы поддержки выполнения программ) создать несколько дополнительных потоков выполнения в рамках одного процесса.

Итого: Параллельность

выполнение некоторых участков кода программы одновременно на разных вычислительных мощностях системы.

Итого: Асинхронность

Отделение вызывающего потока от потока исполнения запроса.

Что когда применять?

- **Многопоточность** – в любой программе, где есть несколько потоков.
- **Параллельность** – где важна вычислительная скорость И задачи можно эффективно распараллеливать.
- **Асинхронность** – где можем найти занятие, пока ждём результата.

26_CustomAsync

```
auto main() -> int {
 PrintHandler printHandler;
 EventLoop eventLoop;

 eventLoop.addHandler(&printHandler);
 eventLoop.send({EventCode::start,"starting"});
 std::thread workerThread{userThread, std::ref(eventLoop)};

 eventLoop.exec();
 workerThread.join();

return 0;
}
```

Проблемы асинхронности

Асинхронный код требует:

1. хранение контекста операции
2. умение этот контекст восстанавливать
3. специальный функционал завершения
4. механизм возврата результата

Corutines

Corutines

1. Корутины в C++ - это новый механизм, добавленный в стандарт C++20, который позволяет создавать функции, которые могут приостанавливаться и возобновляться в произвольный момент времени.

2. Корутины позволяют упростить асинхронное программирование и улучшить производительность за счет эффективного управления потоками выполнения.

3. Корутины имеют свой собственный тип **promise_type**, который определяет, какие значения могут быть возвращены из корутины и как они могут быть переданы между вызовами **coroutine_handle::resume**().

Домашнее задание

Спасибо!

НА СЕГОДНЯ ВСЕ