Лекция 12. Инструменты новых стандартов C++ для создания многопоточных приложений

*Лекция составлена по материалам книги* [*Энтони Уильямс «C++. Практика многопоточного программирования»*](https://www.piter.com/collection/top-prodazh/product/c-praktika-mnogopotochnogo-programmirovaniya)

*и материалам из сети* [*https://habr.com/ru/post/443406/*](https://habr.com/ru/post/443406/)

*Ссылка на дополнительные материалы по теме* [*https://en.cppreference.com/w/cpp/thread*](https://en.cppreference.com/w/cpp/thread)

# Многозадачность и многопоточность

Процессом в программировании принято называть отдельную выполняемую программу, связанную с исполняемым файлом. Каждый процесс имеет свое собственное адресное пространство, исполняемый код, данные, дескрипторы объектов и базовый приоритет. Считается, что при запуске процесса создается поток команд (thread), называемый первичным потоком (начальным потоком), который, в свою очередь, может порождать дочерние потоки (вторичные потоки).

Потоки, принадлежащие одному процессу, имеют общее адресное пространство. Каждый поток имеет собственные обработчики исключительных ситуаций, базовый приоритет. Поток – объект, для которого система выделяет ресурсы процессора.

Поскольку каждый процесс имеет изолированное адресное пространство, данные одного процесса недоступны для другого. Потоки же одного процесса имеют общее адресное пространство. Это позволяет обмениваться данными между потоками с минимальными затратами системных ресурсов. Потоки могут выполняться одновременно (параллельно) на нескольких процессорах. Кроме ускорения работы программы за счёт распараллеливания вычислений потоки применяются и для реализации некоторых алгоритмов – при многопоточной архитектуре алгоритмы становятся понятнее и проще.

Наличие общего адресного пространства и отсутствие защиты данных между потоками существенно сокращают издержки, связанные с использованием нескольких потоков, по сравнению с издержками при выполнении нескольких процессов, так как снижается нагрузка на операционную систему. Но гибкость, обусловленная применением общей памяти, сопряжена и с затратами: если доступ к данным получают несколько потоков, программист должен убедиться, что представление данных, видимых каждым потоком, согласовано при каждом обращении к этим данным. Низкие издержки на запуск потоков внутри процесса и обмен данными между ними стали причиной популярности этого подхода во всех распространенных языках программирования, включая С++, даже несмотря на потенциальные проблемы, связанные с разделением памяти.

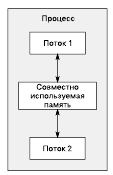


Рис. 1. Обмен данными между двумя потоками, запущенными одновременно в одном процессе

Для *многопоточного* (multithreading) кода часто употребляется другое слово – *параллелизм*, *параллельное программирование* (parallel programming). Существует также близкий по смыслу термин *конкурентность* (concurrency). Разница между конкурентностью и параллелизмом заключается в нюансах, фокусе и намерениях. Оба термина относятся к одновременному выполнению нескольких задач с использованием доступного оборудования, но параллелизм более явно ориентирован на производительность. О параллелизме говорят, когда главной заботой становится использование имеющегося оборудования для повышения производительности массовой обработки данных, тогда как при конкурентности основная забота – разделение задач, требующих одновременного решения, или оперативность реагирования. Соответствующий API в программных платформах также обозначается термином *конкурентность* (например, Java Concurrency).

Исторически язык программирования С++ не содержал инструментов для создания многопоточных программ. Многопоточные программы создавались с использованием инструментария API функций операционных систем, специализированных библиотек (например, MPI) и библиотек некоторых IDE. Это делало программы платформозависимыми. Ситуация поменялась только с введением стандарта С++11. В нем был реализован класс std::thread, снабжённый всем необходимым для создания и управления потоками.

# Создание потоков

Многопоточная программа похожа на любую другую программу на C++ с обычным сочетанием переменных, классов и функций. Единственное реальное отличие заключается в том, что некоторые функции могут выполняться одновременно, поэтому необходимо обеспечить безопасность совместно используемых данных при конкурентном доступе к ним. Чтобы одновременно запустить функции для управления различными потоками, необходимо воспользоваться особыми функциями и объектами. Рассмотрим листинг простейшей многопоточной программы.

#include <iostream>

#include <thread>

void hello()

{

std::cout<<"Hello Concurrent World\n";

}

int main()

{

std::thread t(hello);

t.join();

}

Объявления, предназначенные для поддержки многопоточности в стандартной библиотеке C++, представлены в новых заголовках: функции и классы для управления потоками объявляются в <thread> (функции и классы для защиты совместно используемых данных – в некоторых других заголовках, будет рассмотрено ниже).

Каждый поток должен иметь *функцию потока* - исходную функцию, с которой начинается новый поток команд. Для начального потока в приложении это main(), а для любого другого потока такая функция указывается в конструкторе объекта std::thread. В данном случае это объект std::thread с именем t, имеющий в качестве исходной новую функцию hello(). Новый поток с именем t начнёт выполнение сразу после создания.

После запуска нового потока исходный продолжает выполняться. Если бы он не ждал завершения нового потока, то продолжил бы работу до конца функции main() и завершил программу, возможно, еще до того, как смог бы запуститься новый поток. В некоторых случаях такое завершение может быть ошибочным. Поэтому используется вызов метода потока join() – он заставляет вызывающий поток main() ожидать тот поток, что связан с объектом std::thread (в данном случае поток t). join() можно вызвать для конкретного потока только один раз.

Существует также метод потока detach(). Он делает поток независимым от первичного потока программы и позволяет ему выполняться, даже когда породивший его процесс уже завершён. Такие отсоединенные потоки часто называются *потоками-демонами*.

# Передача аргументов функции потока

При передаче аргументов функции потока по умолчанию аргументы копируются во внутреннее хранилище, где к ним может получить доступ вновь созданный поток, а затем передаются вызываемому объекту или функции как временные значения. Это делается, даже если соответствующий параметр в функции передаётся по ссылке. Рассмотрим пример:

void f(int i, std::string const& s)

{

……

}

std::thread t(f,3,"hello");

В результате создается новый поток выполнения, связанный с t, который вызывает функцию f(3,"hello"). Обратите внимание: даже если f в качестве второго параметра принимает std::string, строковый литерал передается как const char\* и преобразуется в std::string только в контексте нового потока.

Использование ссылок как аргументов функции потока бывает удобно в тех случаях, когда поток связан с обработкой данных и необходимо вернуть результат этой обработки. В этом случае нужно передать в поток ссылку на переменную-результат и указать это при вызове с помощью функции std::ref. Пример:

void update\_data\_for\_widget(widget\_id w, widget\_data& data)

{

……..

}

int main()

{

widget\_data data;

std::thread t(update\_data\_for\_widget, w, std::ref(data)); display\_status();

t.join();

process\_widget\_data(data);}

В данном случае переменная data будет передана по ссылке и будучи измененной в потоке t сохранит изменения и вне потока.

# Остановка и продолжение потока

Иногда при выполнении программы требуется приостановить поток на некоторое время. Для этого используется функция std::this\_thread::sleep\_for(sleep\_duration). Параметр sleep\_duration имеет особый тип временных литералов const [std::chrono::duration](http://en.cppreference.com/w/cpp/chrono/duration).

Пример использования функции.

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

В данном случае текущий поток приостановит работу на 100 миллисекунд.

Есть также функция std::this\_thread::sleep\_until( ), приостанавливающая работу потока до наступления некоторого момента времени.