# Multithreading

**Многопоточность** — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких *потоков*, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины (с) Wiki.

**Поток (thread)** – средство выполнения кода программы — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов.

Для компьютеров с одним процессором истинная многопоточность невозможна, для них возможна только **вытесняющая многозадачность** – когда диспетчер задач отдает квант времени одному потоку, затем приостанавливает его, отдает процессорное время другому потоку и т.д.

**Многопоточность** в условиях нескольких ядер – одновременно может выполняться количество потоков равное количеству процессоров.

**Синхронизация** – разграничение доступа к каким-либо ресурсам программы/системы для нескольких потоков либо программ.

**Что синхронизировать**: работу с контейнерами/объектами, к которым происходит доступ из нескольких потоков, работу системными ресурсами (файлы/память), если это необходимо.

**Как реализовать**: с помощью средств синхронизации, в некоторых случаях с помощью умных указателей и определения стратегии владения объектами.

Условная классификация **средств синхронизации**:

1. **Основные примитивы(Mutable execution)**

mutex

timed\_mutex

recursive\_mutex

recursive\_timed\_mutex

shared\_mutex

shared\_timed\_mutex

Название происходит от использования – для синхронизации блока выполняемого кода (функции, блока внутри функции). Особенности:

- наличие владельца: объект синхронизации содержит в себе указатель на поток либо id потока;

- повторное вхождение: объект синхронизации может быть захвачен более одного раза одним потоком. Освободить нужно столько же раз, сколько раз объект был захвачен.

1. **Condition variables**

condition\_variable

condition\_variable\_any

Примитив, позволяющий ожидать, пока не случится какое-то событие с последующим атомарным захватом лока при выходе из функции ожидания. Рассмотрим на примере простого случая распространенной задачи producer/consumer:

std::mutex mutex;

std::condition\_variable cv;

int g\_x = 0;

bool ready = false;

void Producer() {

std::unique\_lock<std::mutex> lock(mutex); // acquire lock

for (; g\_x < 100;) {

cv.wait(lock, [=] { return !ready; });

++g\_x;

ready = true;

cv.notify\_one();

}

}

void Consumer() {

std::unique\_lock<std::mutex> lock(mutex);

for (; g\_x < 100;) {

cv.wait(lock, [=] { return ready; });

std::cout << g\_x << "\n";

ready = false;

cv.notify\_one();

}

}

1. **Дополнительные средства**:

**Semaphore(std::counting\_semaphore, std::binary\_semaphore)**  – примитив, позволяющий войти в заданный участок кода не более чем N потокам. Содержит счетчик, который инкрементится/декрементится с вызовом enter/leave функций.

**Read-Write locks**. Применяется в случае, когда чтение ресурса происходит много чаще его изменения. Потоков, захвативших read lock, может быть несколько, write lock может захватить только один поток.

**Распространенные проблемы при разработке многопоточных программ:**

* **deadlock** - неосвобожденный лок, как результат – вечное ожидание в одном/нескольких потоках;
* **livelock** - почти то же, что и дедлок, но лок проверяется в цикле с каким-то интервалом времени;
* **starvation** – проблема, когда поток не может получить доступ к ресурсу. Чаще всего спровоцирована слишком простым алгоритмом планировшщика, когда он, например, всегда отдает ресурс потоку с наивысшим приоритетом и потока с более низким приоритетом просто нет возможности захватить ресурс.
* **race condition** – гонки потоков – ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

**Рекомендации по разработке многопоточных приложений**:

* Не совершайте под локом сетевых вызовов или любых других вызовов, которые могут длиться неопределенно долго.
* Не ждите какого-либо события под локом.
* Всегда четко определяйте к каким ресурсам будет происходить доступ из нескольких потоков. Разграничивайте этот доступ с помощью примитивов синхронизации, дабы в любой момент времени, когда поток получает доступ к объекту, объект находился в согласованном состоянии. Чаще всего в роли объектов будут выступать контейнеры и указатели. Тщательно следите за временем жизни объекта, дабы поток не обратился к удаленному объекту либо удаленному элементу контейнера.
* Используйте умные указатели для совместного владения.
* Используйте гарды-врапперы для освобождения объекта синхронизации. Guard представляет собой класс, принимающий в конструкторе константную ссылку на объект синхронизации (cs, mutex etc), и вызывающий в деструкторе освобождение лока. Это даст вам гарантию того, что ваш объект не останется неосвобожденным, если в функции вылетит exception.
* Помните, что в любой момент у потока могут отобрать управление.

**Литература:**

* С++ concurrency in action, Anthony Williams
* "Джеффри Рихтер. Windows для профессионалов. Создание эффективных WIN32-приложений с учетом специфики 64-разрядной версии Windows”. 2001 год
* MSDN
* Wikipedia
* <https://en.cppreference.com/w/cpp/thread>

**Упражнения для самостоятельной работы:**

* Реализуйте задачу single producer/single consumer
* Реализуйте простейший thread pool
* Напишите свою реализацию rwlocks