C++. Базовый уровень

Многопоточность в C++. Создание потоков через класс thread. Синхронизация потоков через мьютексы и условные переменные.

КОД будущего

Минцифры России

инжиниринум
МГТУ им. Н.Э. Баумана

Национальные проекты России
ЦИФРОВАЯ ЭКОНОМИКА

20.35 УНИВЕРСИТЕТ

2023

Многопоточность

Многозадачность (multitasking) — свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач.

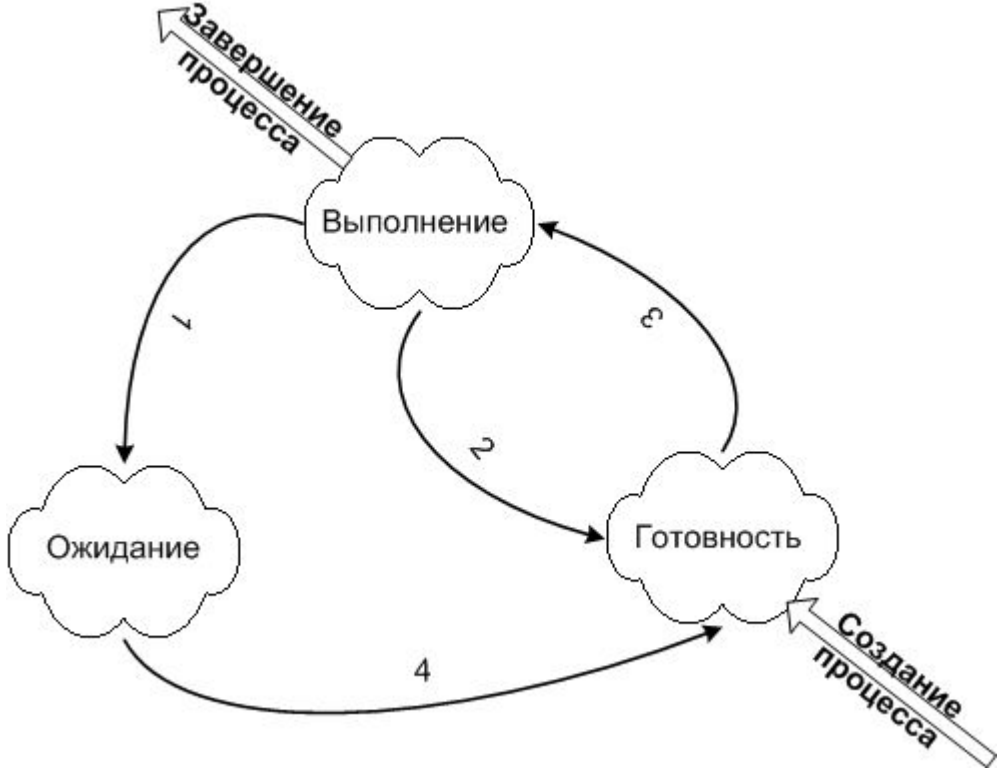
Многопоточность (multithreading) — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что **процесс**, порождённый в операционной системе, может состоять из нескольких **потоков**, выполняющихся «параллельно», то есть без предписанного порядка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

Многопоточность

Параллельное выполнение задач возможно только в многопроцессорной системе!

В остальных случаях используется псевдопараллельное исполнение, где процессы выполняются параллельно за малые кванты времени.

Многопоточность



Многопоточность

Поток - это наименьшая единица выполнения внутри процесса, в то время как процесс - это выполняющаяся программа, которой выделены системные ресурсы.
Процессы обычно независимы друг от друга, а потоки существуют как подмножества процесса и совместно используют память и ресурсы процесса.

Поток может принадлежать только одному процессу, а процесс может иметь несколько потоков, которые выполняют разные задачи одновременно.

Процессы используют разные адресные пространства, в то время как потоки используют то же самое адресное пространство процесса. Это означает, что один процесс не может получить доступ к данным другого процесса без межпроцессного взаимодействия, а потоки могут обмениваться данными между собой.

Многопоточность

```cpp
#include <iostream>
#include <thread>

// Функция, которая будет выполняться в отдельном потоке
void print_hello(int n) {
    std::cout << "Hello from thread " << n << "\n";
}

int main() {
    // Создаем два потока, передавая им функцию print_hello и номер потока
    std::thread t1(print_hello, 1);
    std::thread t2(print_hello, 2);

    // Ждем, пока оба потока завершат свою работу
    t1.join();
    t2.join();

    std::cout << "Hello from main thread\n";
    return 0;
}
```

```
Hello from thread Hello from thread 1
2
Hello from main thread
```

Многопоточность

Однако, если несколько потоков работают с общими данными или ресурсами, то может возникнуть проблема согласованности или гонки данных (race condition). Это означает, что результат выполнения программы может зависеть от того, в каком порядке и скорости выполняются операции в разных потоках.

Многопоточность

```cpp
#include <iostream>
#include <thread>
#include <vector>

// Глобальная переменная, которая будет изменяться в разных потоках
int sum = 0;
// Функция, которая добавляет n к sum
void add(int n) {
    sum += n;
}
int main() {
    // Создаем вектор из четырех потоков, передавая им функцию add и разные значения
    std::vector<std::thread> threads;
    threads.emplace_back(add, 10);    threads.emplace_back(add, 20);
    threads.emplace_back(add, 30);    threads.emplace_back(add, 40);

    // Ждем, пока все потоки завершат свою работу
    for (auto& t : threads) t.join();

    // Выводим на экран значение sum
    std::cout << "Sum = " << sum << "\n";
    return 0;
}
```

```
Sum = 100
```

Многопоточность

Чтобы избежать проблем, нужно использовать механизмы **синхронизации потоков,** которые обеспечивают взаимное *исключение* (mutual exclusion) или *координацию* (coordination) потоков. В стандартной библиотеке <thread> есть два основных механизма синхронизации: мьютексы (mutexes) и условные переменные (condition variables).

Мьютекс - это объект, который может быть захвачен (locked) или освобожден (unlocked) одним потоком за раз. Когда поток захватывает мьютекс, он получает эксклюзивный доступ к общим данным или ресурсам, которые защищаются этим мьютексом.

Когда поток освобождает мьютекс, он позволяет другим потокам захватить его и получить доступ к общим данным или ресурсам. Если поток пытается захватить мьютекс, который уже захвачен другим потоком, то он блокируется до тех пор, пока мьютекс не будет освобожден.

Многопоточность

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

// Глобальная переменная, которая будет изменяться в разных потоках
int sum = 0;
// Глобальный объект мьютекса, который будет защищать доступ к sum
std::mutex mtx;

// Функция, которая добавляет n к sum с использованием мьютекса
void add(int n) {
    mtx.lock(); // Захватываем мьютекс перед изменением sum
    sum += n;
    mtx.unlock(); // Освобождаем мьютекс после изменения sum
}
int main() {
    // Создаем вектор из четырех потоков, передавая им функцию add и разные значения
    std::vector<std::thread> threads;
    threads.emplace_back(add, 10);    threads.emplace_back(add, 20);
    threads.emplace_back(add, 30);    threads.emplace_back(add, 40);

    for (auto& t : threads) t.join(); // Ждем, пока все потоки завершат свою работу

    std::cout << "Sum = " << sum << "\n"; // Выводим на экран значение sum
    return 0;
}
```

Многопоточность

Мьютекс используется для того, чтобы избежать гонки данных (race condition) или нарушения согласованности данных при одновременном доступе к ним из разных потоков.

Условная переменная - это объект, который позволяет одному или нескольким потокам ожидать выполнения некоторого условия, связанного с общими данными или ресурсами, которые защищаются мьютексом.

Когда другой поток изменяет данные или ресурсы таким образом, что условие становится истинным, он может уведомить об этом ожидающие потоки. Тогда один или все ожидающие потоки просыпаются и пытаются захватить мьютекс, чтобы продолжить свою работу.

Условная переменная используется для того, чтобы организовать взаимодействие между потоками, которые работают с общими данными или ресурсами, и координировать их действия в зависимости от состояния данных или ресурсов.

Зачем условной переменной нужен мьютекс? Потому что условная переменная не может сама по себе защитить доступ к общим данным или ресурсам от других потоков. Она только позволяет ожидающему потоку узнать, когда условие стало истинным, но не гарантирует, что данные или ресурсы не будут изменены другими потоками в то время, как ожидающий поток просыпается и пытается захватить мьютекс. Поэтому нужно использовать мьютекс в сочетании с условной переменной для того, чтобы обеспечить атомарность проверки условия и доступа к данным или ресурсам.

Многопоточность

```
#include <iostream>
#include <thread>
#include <string>
#include <mutex>
#include <condition_variable>

const int MAX_SIZE = 5; // Максимальный размер буфера
std::string buffer; // Общий буфер сообщений
std::mutex mtx; // Общий мьютекс для защиты доступа к буферу
std::condition_variable cv; // Общая условная переменная для синхронизации потоков

// Функция, которая выполняется в потоке-писателе
void writer(int n) {
    for (int i = 0; i < n; i++) {
        std::unique_lock<std::mutex> lock(mtx); // Захватываем мьютекс перед изменением буфера
        cv.wait(lock, [] {return buffer.size() < MAX_SIZE; }); // Ждем, пока буфер не освободится
        buffer += "A"; // Добавляем сообщение в буфер
        std::cout << "Writer " << std::this_thread::get_id() << " wrote A\n";
        lock.unlock(); // Освобождаем мьютекс
        cv.notify_one(); // Уведомляем читателя
    }
}
```

Многопоточность

```
// Функция, которая выполняется в потоке-читателе
void reader(int n) {
    for (int i = 0; i < n; i++) {
        std::unique_lock<std::mutex> lock(mtx); // Захватываем мьютекс перед изменением буфера
        cv.wait(lock, [] {return buffer.size() > 0; }); // Ждем, пока буфер не заполнится
        char value = buffer.back(); // Извлекаем сообщение из буфера
        buffer.pop_back();
        std::cout << "Reader " << std::this_thread::get_id() << " read " << value << "\n";
        lock.unlock(); // Освобождаем мьютекс
        cv.notify_one(); // Уведомляем писателя
    }
}
```

Многопоточность

```
int main() {
    // Создаем два потока: писатель и читатель
    std::thread t1(writer, 10);
    std::thread t2(reader, 10);

    // Ждем, пока оба потока завершат свою работу
    t1.join();
    t2.join();

    return 0;
}
```

```
Writer 5212 wrote A
Writer 5212 wrote A
Writer 5212 wrote A
Reader 53636 read A
Reader 53636 read A
Reader 53636 read A
Writer 5212 wrote A
Writer 5212 wrote A
Writer 5212 wrote A
Writer 5212 wrote A
Writer 5212 wrote A
Reader 53636 read A
Reader 53636 read A
Reader 53636 read A
Reader 53636 read A
Reader 53636 read A
Writer 5212 wrote A
Writer 5212 wrote A
Reader 53636 read A
Reader 53636 read A
```