# /Инструменты синхронизации доступа к общим ресурсам – мьютексы

Когда дело доходит до совместной работы с данными нескольких потоков, то все проблемы возникают из-за последствий изменения этих данных. Если все совместно используемые данные доступны только для чтения, проблем не будет, поскольку данные, считываемые одним потоком, не зависят от того, читает другой поток те же данные или нет. Но если один или несколько потоков, совместно использующих данные, начинают вносить в них изменения, создаются серьезные предпосылки для возникновения проблем. В таком случае следует обеспечить приемлемость конечных результатов.

Один из подходов к решению - помечать все фрагменты кода, обращающиеся к совместно используемым данным, как взаимоисключающие, чтобы при выполнении одного из них каким-либо потоком любой другой поток, пытающийся получить доступ к этой структуре данных, был бы вынужден ждать, пока первый поток не завершит выполнение помеченного фрагмента. Тогда поток не смог бы обратиться к данным, кроме тех случаев, когда он сам выполнял бы их модификацию.

Для защиты фрагментов кода от использования другими потоками предусмотрены специальные объекты – мьютексы std::mutex. В примере ниже разделяемым ресурсом является std::cout является разделяемым ресурсом. С ним работают 6 потоков – первичный и 5 вторичных.

#include <iostream>

#include <string>

#include <thread>

#include <mutex>

std::mutex mu; // Создаём мьютекс

// Функция потока, выводящая сообщение на консоль

void CallHome(string message)

{

mu.lock(); // Блокируем доступ к коду другим потокам

std::cout << "Thread " << this\_thread::get\_id() << " says " << message << endl;

mu.unlock();// Снимаем блокировку

}

int main()

{

thread t1(CallHome, "Hello from Jupiter");

thread t2(CallHome, "Hello from Pluto");

thread t3(CallHome, "Hello from Moon");

CallHome("Hello from Main/Earth");

thread t4(CallHome, "Hello from Uranus");

thread t5(CallHome, "Hello from Neptune");

t1.join();

t2.join();

t3.join();

t4.join();

t5.join();

return 0;

}

В результате все сообщения будут выведены «красиво». Если из функции потока убрать блокировки – символы на консоль будут выводиться хаотично.

Вызов методов lock() и unlock() непосредственно у мьютекса не является предпочтительным вариантом потому, что вы можете забыть отдать удерживаемую блокировку. Такой подход применим лишь в самых простых, «очевидных» случаях. Если не разблокировать ресурс, то все остальные потоки, которые ожидают освобождения ресурса, будут бесконечно заблокированы и программа может зависнуть.

В нашем примере, если забыть разблокировать мьютекс в вызове функции CallHome(), в стандартный поток будет выведено первое сообщение из потока t1 и программа зависнет. Так происходит из-за того, что поток t1 получил блокировку мьютекса, а остальные потоки ждут освобождения этой блокировки.

Подобные ошибки часто случаются, именно поэтому нежелательно использовать методы lock()/unlock() напрямую из мьютекса. Вместо этого следует использовать шаблонный класс std::lock\_guard. Когда объект lock\_guard создаётся, он пытается завладеть мьютексом. Когда программа выходит из области видимости lock\_guard объекта, вызывается деструктор, который освобождает мьютекс.

Перепишем функцию CallHome() с применением std::lock\_guard объекта:

void CallHome(string message)

{

std::lock\_guard<std::mutex> lock1(mu); // пытаемся захватить блокировку

std::cout << "Thread " << this\_thread::get\_id() << " says " << message << endl;

}// объект lock\_guard уничтожится и освободит мьютекс

Другими словами std::lock\_guard обеспечивает блокировку кода в пределах своей области видимости.

Следует отметить, что для объекта мьютекс не определён конструктор копирования, поэтому передавать его параметром в функцию можно только по ссылке.

# Асинхронный вызов функций

Если вам нужно, чтобы код какой либо функции выполнился асинхронно, т.е. без блокировки основного потока выполнения, наилучшим выбором будет использование std::async(). Это равносильно созданию потока и передаче необходимого кода на выполнение в этот поток через указатель на функцию или параметр в виде лямбда функции. Однако, в последнем случае вам необходимо следить за созданием, присоединением/отсоединением этого потока, а также за обработкой всех исключений, которые могут возникнуть в этом потоке. Если вы используете std::async(), вы избавляете себя от этих проблем, а также резко снижаете свои шансы попасть в deadlock.

Значительное преимущество использования асинхронного вызова заключается в возможности получить результат выполнения асинхронной операции обратно в вызывающий поток с помощью std::future объекта.

Представим, что у нас есть функция ConjureMagic(), которая возвращает int. Мы можем запустить асинхронную операцию, которая установит значение в будущем в future объект, когда выполнение задачи завершится, и мы сможем извлечь результат выполнения из этого объекта в том потоке выполнения, из которого операция была вызвана.

Делается это следующим образом.

// запуск асинхронной операции и получение обработчика для future объекта

std::future<int> myFuture = std::async(&ConjureMagic);

//... выполнение каких-то операций

if (myFuture.valid()) // Проверяем закончена ли операция

{

int val = myFuture.get(); // Получаем значение

}

С точки зрения производительности, накладные расходы на создание потока на порядок больше, чем использование асинхронного вызова функций.

# Закон Амдала

Существуют специальные оценки эффективности решения задач на с использованием параллельного программирования. Одна из наиболее известных таких оценок называется *законом Амдала*.

Пусть в имеющейся программе доля операций, которые нужно выполнять последовательно, равна f, причём 0£f£1. Крайние случаи в значениях f соответствуют полностью параллельным (f=0) или полностью последовательным (f=1) программам. Ускорение, которое можно получить на компьютере, состоящем из p процессоров, при данном значении f (в случае, если последовательные и параллельные участки кода программы распределены равномерно), можно получить из следующей формулы:

s£1/{f+(1-f)/p}.

Например, предположим, что в программе лишь 10% последовательных операций, т.е. f=0.1. В этом случае из формулы следует: сколько бы процессоров ни использовали, ускорение работы программы более чем в 10 раз не получится.

Отсюда следует вывод, что при работе на параллельных компьютерах необходима оценка заложенного в программе алгоритма и если доля последовательных операций в нём велика, то на значительное ускорение рассчитывать не придётся.

Путём к увеличению производительности вычислений до значений, превышающих прогнозируемые законом Амдала, является так называемое *естественное распараллеливание* алгоритмов, нарушающее равномерность распределения параллельных и последовательных участков кода программы и позволяющее в идеале добиться наивысшего ускорения вычислений, кратного числу используемых процессоров.