МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

(ДГТУ)

Кафедра "Программное обеспечение вычислительной техники и автоматизированных систем"

Синхронизация потоков выполнения в Си/C++

Методические указания к практическим работам по дисциплине "Параллельные вычисления"

Ростов-на-Дону

20 г.

Составитель: к.ф.-м.н., доц. Габрельян Б.В.

УДК 512.3

Синхронизация потоков выполнения в Си/С++: методические указания – Ростов н/Д: Издательский центр ДГТУ, 20 . – с.

В методической разработке рассматриваются вопросы синхронизации работы потоков команд в С++11. Даны задания по выполнению лабораторной работы. Методические указания предназначены для направления 01.05.00 – «Математическое обеспечение и администрирование информационных систем».

Цель работы: Ознакомление с конструкциями синхронизации потоков команд на уровне языка, реализованной в стандарте C++11.

I. Синхронизация доступа к разделяемому ресурсу.

Если несколько потоков обращаются к одному и тому же (разделяемому) ресурсу и при этом хотя бы один из них изменяет этот ресурс необходимо синхронизировать работу потоков. Пока ресурс не будет полностью модифицирован все потоки кроме того, который выполняет изменение должны приостановить свою работу. Для этого все потоки, работающие с данным ресурсом должны действовать согласованно. Поток изменяющий ресурс должен как-то сообщить, что он занял ресурс, а другие потоки должны проверять, не занят ли сейчас этот ресурс и, если занят, ожидать его освобождения. C++11 предлагает класс mutex (mutual exclusion – взаимное исключение), который можно использовать для блокировки доступа потоков к разделяемому ресурсу. Мьютекс можно заблокировать с помощью метода lock() и разблокировать с помощью метода unlock(). Описание класса mutex содержится в файле заголовков mutex. Все потоки, работающие с общим ресурсом должны использовать один и тот же объект класса mutex. Каждый поток, желающий получить эксклюзивный допуск к ресурсу должен вначале заблокировать мьютекс, а после того как ресурс будет изменен или использован без изменения разблокировать мьютекс. Попытка заблокировать мьютекс может не удастся, если он уже заблокирован другим потоком, тогда метод lock() заставляет поток ждать освобождения мьютекса.

```
#include <thread>
#include <mutex>
#include <queue>
#include <chrono>
#include <iostream>
using namespace std;
queue<int> que; // разделяемый ресурс
mutex mu;
void set() {
```

```
        while(1) {
            mu.lock(); // блокировка мьютекса
            que.push( rand()%101 ); /* добавление в очередь случайного значения из диапазона [0..100] */
            mu.unlock(); // разблокировка мьютекса
            this_thread::sleep_for(chrono::milliseconds(100)); /* эмуляция длительных вычислений, передача кванта времени другим потокам */
        }
}
void get() {
        while(1) {
            mu.lock(); // блокировка мьютекса
            int value = que.front(); // получение очередного значения из очереди
            que.pop(); // удаление значения из очереди
            mu.unlock(); // разблокировка мьютекса
            cout << “get: “ << value << endl;
            this_thread::sleep_for(chrono::milliseconds(100));
        }
}

int main() {
        thread   t1(set), t2(get);
        …
        t1.join();
        t2.join();
}
```

Нельзя в том же самом потоке снова заблокировать уже заблокированный мьютекс. Если требуется это сделать необходимо использовать объект класса recursive_mutex.

Поток, заблокировавший мьютекс должен до завершения своей работы освободить этот мьютекс. Это должно быть сделано при всех условиях, в том числе и при возникновении исключительных ситуаций. Для этого удобно использовать какой-нибудь класс-оболочку, при создании объекта которого (в конструкторе) мьютекс блокируется, а при уничтожении (в деструкторе) разблокируется. C++11 предлагает для этого шаблоны классов lock_guard<mutex> и unique_lock<mutex>. lock_guard не позволяет освобождать мьютекс (все время существования объекта этого класса мьютекс находится в заблокированном состоянии), а unique_lock позволяет управлять состоянием мьютекса.

```
mutex mu;
…
void set() {
        while(1) {
                {
                        lock_guard<mutex> lg(mu); // блокировка мьютекса
                        que.push( rand()%101 ); /* добавление в очередь значения */
                } // уничтожение объекта lg и разблокировка мьютекса
                this_thread::sleep_for(chrono::milliseconds(100));
        }
}
```

II. Условные переменные

Пусть какому-то потоку или группе потоков необходимо дождаться результатов работы некоторого другого потока. В принципе, они могут использовать некоторый разделяемый флажок (булеву переменную), изначально имеющий значение false (флаг сброшен) и устанавливаемый в true (флаг установлен) потоком генерирующим данные для других потоков после того, как нужные данные готовы. Тогда остальные потоки должны проверять значение флага и,

если он не установлен засыпать на некоторый промежуток времени. Когда флаг выставлен эти потоки могут использовать готовые данные. В этой схеме, во-первых, всем потокам нужно обеспечивать согласованный доступ к флагу и, во-вторых, не всегда понятно на какой промежуток времени должны засыпать ожидающие потоки. Если промежуток мал, будет проводиться слишком много проверок значения флага, если велик, то даже когда данные уже готовы потоки еще некоторое время будут продолжать спать. Для таких задач можно использовать условные переменные. Условные переменные - это примитив синхронизации обеспечивающий ожидание одним или несколькими потоками сигнала от другого потока. До получения сигнала потоки остаются заблокированными. Сигнал получают один или все ожидающие потоки, это приводит к разблокировке соответствующего потока (или потоков). C++11 предлагает класс condition_variable описанный в файле заголовков <condition_variable>. Потоки ожидающие события вызывают метод этого класса wait. Поток генерирующий событие вызывает метод класса notify_one для посылки сигнала какому-то одному (какому решает среда выполнения) из ожидающих потоков или метод notify_all для посылки сигнала всем ожидающим потокам. Метод wait использует в своей работе мьютекс, точнее класс-оболочку над мьютексом. При этом он может как блокировать, так и разблокировать этот мьютекс, поэтому класс-оболочка это не lock_guard, a unique_lock. Все ожидающие потоки должны использовать один и тот же мьютекс.

```
#include <iostream>

#include <thread>

#include <mutex>

#include <condition_variable>

#include <queue>


using namespace std;


// глобальные переменные доступны всем потокам

mutex mu;

condition_variable cv;
```

```
queue<int> que;

void setData() {
	{
		lock_guard<mutex> lg(mu);
		que.push( rand() % 101 );
	}
	cv.notify_one();
}
void getData() {
	unique_lock<mutex> ul(mu);
	cv.wait(ul);
	if(!que.empty()) { /// !!! Возможны ложные срабатывания
		cout << “get “ << this_thread::get_id() << “: ” << que.front() << endl;
		que.pop();
	}
}
void generator() {
	while(1) setData();
}
void reader() {
	while(1) getData();
}

int main() {
```

```
    thread t1(generator), t2(reader), t3(reader);
    t1.join();
    t2.join();
    t3.join();
}
```

Зачем нужно условие if(!que.empty()) в методе getData()?

При использовании условных переменных, в принципе возможны ложные срабатывания, поэтому нужна дополнительная проверка готовности данных для чтения. Класс condition_variable предлагает еще одну версию метода wait с двумя аргументами: первый по-прежнему ссылка на объект класса unique_lock, а второй функция (точнее то, что ведет себя как функция), возвращающая булево значение (предикат). Поток пробуждается, когда получен сигнал от условной переменной и, одновременно, предикат возвращает значение true. Это позволяет переписать метод getData(), например, следующим образом:

```
bool predicate() {
    return !que.empty();
}

void getData() {
    unique_lock<mutex> ul(mu);
    cv.wait(ul, predicate);
    cout << “get “ << this_thread::get_id() << “: ” << que.front() << endl;
    que.pop();
}
```

или так

```
void getData() {
    unique_lock<mutex> ul(mu);
```

```
        cv.wait( ul, [] { return !que.empty();} );
        cout << "get " << this_thread::get_id() << ": " << que.front() << endl;
        que.pop();
}
```

III. Атомарные типы.

C++11 поддерживает шаблон класса atomic<T>. Шаблон задан в файле заголовков atomic. Этот шаблон позволяет создавать и использовать в программе атомарные типы. Атомарные типы дают возможность использовать атомарные операции – такие операции, которые гарантированно выполняются как единое целое. Проблемы с одновременным доступам к разделяемому ресурсу в том случае, если какой-либо (или какие-либо) поток изменяет ресурс связаны с тем, что изменение ресурса может состоять из отдельных этапов, каждый из которых приводит к частичным изменениям и только все вместе они обеспечивают полное изменение ресурса. При выполнении отдельных, частичных изменений ресурс находится в неопределенном состоянии. Атомарные операции выглядят так, как будто частичных изменений нет, и изменяется сразу весь ресурс, тем самым при выполнении атомарной операции ресурс не может оказаться в неопределенном состоянии. В принципе можно построить атомарный тип на основе разных типов (этот тип, это тип ресурса), но выгоду от использования таких типов можно получить, если атомарный тип основан на каком-то простом типе, изменения значения которого происходит быстро. Мы знаем, что для того, чтобы изменение ресурса для всех параллельно работающих потоков не приводило к доступу к частично модифицированному значению можно использовать блокировки на основе мьютекса. Но тогда будет расходоваться дополнительная память и нужно будет генерировать команды по блокировке и разблокировке мьютекса. В случае простых атомарных типов могут быть сгенерированы машинные команды, которые обеспечивают атомарность операций для таких типов без необходимости выполнять блокировку потоков (неблокирующая синхронизация). Существуют специализации шаблона atomic<T> для булева типа, целочисленных типов и указателей.

Основные атомарные операции это void store(T value) и T load(). Первая позволяет задать, а вторая получить значение соответствующей атомарной переменной. Кроме того, поддерживается операция преобразования к типу, на основе которого построен атомарный тип. Например, для atomic<int> к типу int. Для

целочисленных типов и указателей разрешены также операции ++, --, +=, -=, &=, |=, ^=.

IV. Задания.

1. Создайте класс Queue, реализующий понятие "потокобезопасная очередь", то есть очередь, объекты которой можно использовать из разных потоков команд.

2. Реализуйте схему поставщик/потребитель (producer/consumer). В этой схеме один или несколько потоков генерируют некоторые данные и помещают их в разделяемый потоками ресурс, причем ресурс имеет ограничения по размеру (используйте в качестве ресурса массив фиксированного размера), а другой поток или другие потоки извлекают эти данные из ресурса. Используйте условные переменные.

3. Реализуйте схему поставщик/потребитель (producer/consumer), но вместо условной переменной используйте атомарный тип atomic<bool>.

4. Реализуйте схему читатели/писатели (reader/writer). В этой схеме один или несколько потоков имеют доступ к разделяемому потоками ресурсу только на чтение, а другой поток или другие потоки только на запись.

*5. Создайте класс Barrier позволяющий реализовать примитив синхронизации "барьер". Объект класса должен быть доступен всем потокам, использующим барьер. При создании объекта класса Barrier через аргумент конструктора, или после создания (но до использования в качестве барьера) с помощью соответствующего метода задается число потоков, использующих барьер. Далее в нужный момент каждый поток вызывает специальный метод класса Barrier для установки барьера. Во всех участвующих в этом потоков команды следующие за командой установки барьера должны выполняться только после того, как все потоки достигли этого барьера. Класс Barrier должен быть организован таким образом, чтобы его объекты можно было использовать повторно.

*6. Используйте класс Barrier для реализации распараллеленного алгоритма редукции. Алгоритму передается массив элементов типа float, его размер, признак, определяющий какую операцию нужно выполнять в процессе редукции – сложение или умножение, максимальное число потоков, которые должен использовать алгоритм.

V. Контрольные вопросы.

1. Для чего используются мьютексы?
2. Какие виды мьтексов есть в C++11?
3. В чем назначение шаблона lock_quard<>?
4. Чем lock_guard<> отличается от unique_lock<>?
5. Каково назначение условных переменных?
6. Для чего предназначен метод wait класса condition_variable?
7. Для чего предназначен методы notify_one и notify_all класса condition_variable?
8. В чем смысл использования атомарных переменных?
9. Какие операции должны поддерживать все атомарные типы?
10. Какие операции поддерживает тип atomic<int>?

Литература

1. Энтони Уильямс "Параллельное программирование на C++ в действии" – М.: ДМК Пресс, 2012. – 672 с.

2. Джеффри Рихтер "Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows" – СПб.: Питер; М.: Русская редакция, 2001. – 752 с.