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"

Использование 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

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

В методической разработке рассматриваются вопросы создания и использования 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 многопоточностью на уровне языка, реализованной в стандарте C++11 и возможностями, предлагаемыми WinAPI.

I. Потоки выполнения по стандарту C++11.

Поток выполнения в C++11 представлен классом std::thread. Этот класс и другие (но далеко не все) классы и функции, предназначенные для поддержки потоков, объявлены в файле заголовков <thread>. Конструктор класса thread многократно перегружен и позволяет передавать вновь созданному объекту указатель на функцию, которая будет выполняться в этом потоке. Завершение работы этой функции (функции потока) означает нормальное завершение работы потока. Вызов функции осуществляется из конструктора класса thread. Простейший вариант создания и запуска нового потока выглядит следующим образом:

```
#include <thread>

using namespace std;


void f() { /* тело функции потока */ }


int main() {

        thread   t(f); // создание потока и запуск в нем функции f()

        …

        return 0;

}
```

Функция main запускается в начальном (основном) потоке. Потоки, создаваемые в функции main, по умолчанию становятся рабочими потоками. Такие потоки принудительно завершаются при завершении порождающего их потока. Чтобы заставить порождающий поток ждать завершения порожденного потока нужно вызвать для последнего метод join().

```
        thread   t(f);

        t.join();
```

Теперь функция main не завершится, пока не завершится работа потока t. Вызвать join для потока можно только единожды, после этого поток становится "неприсоединяемым". Узнать, можно ли "присоединить" поток, можно, вызвав для него метод joinable().

Поток не будет принудительно завершаться при завершении порождающего потока и в том случае, если он фоновый (демон в терминологии Unix). Для того чтобы сделать поток фоновым нужно вместо join() вызвать для него метод detach().

```
thread   t(f);
t.detach();
```

Функции потока можно передавать аргументы. Например,

```
void fun1(int);


int main() {
    thread   t(fun1,10); // в новом потоке вызывается fun1(10);
    t.join();
    return 0;
}
```

Можно передавать также объекты-функции (объекты классов, в которых переопределена операция вызова функции). Например,

```
class A {
    int a;
public:
    A(int  x = 0) : a(x) {}
    void operator ()() { cout<<++a<<endl; }
    void operator ()(int  x) { cout<<(a += x)<<endl; }
};
```

```
int main() {
	A x(7);
	thread   t1( x ); // вызов x.operator();
	thread   t2( x, 10 ); // вызов x.operator(10);
	t1.join();
	t2.join();
	return 0;
}
```

Кроме того, в качестве функции потока можно использовать метод класса, но в этом случае, очевидно, нужно передавать в качестве первого аргумента указатель (или ссылку) на объект этого класса. Например,

```
class A {
	int a;
public:
	A(int  x = 0) : a(x) {}
	void add(int  x) { a +=x; }
};


int main() {
	A x(7);
	thread   t(&A::add,&x,10); // вызов x.add(10);
	t.join();
	return 0;
}
```

Последний пример может привести к ошибке, если новый поток t будет фоновым (t.detach() вместо t.join()) т.к. в этом случае локальная переменная x уни-

чтожается при завершении основного потока, а поток t может еще продолжать свою работу.

Если функции потока требуется передать не значение, а ссылку то возникает следующая проблема.

```
class M {…};

void  fun2(M& m);


int main() {

        M  x;

        thread   t( fun2, x );

        t.join();

        return 0;

}
```

Сама функция fun2 ожидает ссылку на объект класса M, но конструктор класса thread получает аргумент x по значению, поэтому в нем создается копия x и эта копия передается fun2 по ссылке. Чтобы fun2 получила не копию, а сам объект нужно явно указать это при передаче x конструктору thread. Для этого используют шаблон функции ref из стандартной библиотеки C++.

```
        thread   t( fun2, std::ref(x) );
```

Количество создаваемых потоков определяется программистом. Их реальную привязку к процессорам или вычислительным ядрам выполняет операционная система. Но выгоду от распараллеливания задачи можно получить только при эффективном использовании имеющихся аппаратных ресурсов. Статический метод класса tread hardware_concurrency() возвращает количество процессоров или вычислительных ядер доступных в системе. Если этот метод не может получить такую информацию, возвращается значение 0.

```
        int  tCount = thread::hardware_concurrency();
```

С каждым потоком связывается идентификатор – объект класса std::thread::id. Для идентификаторов разрешены операции сравнения (== != < <= > >=), присваивание и помещение в поток вывода. Получить идентификатор потока

можно либо по ссылке на поток, с помощью метода get_id(), либо с помощью std::this_thread::get_id().

Потоки должны быть уникальными, т.е. нельзя создавать копии потока и нельзя присвоить один поток другому. Для этого в реализации класса thread конструктор копирования и операция присваивания закрыты. Но можно передавать владение потоком от одного объекта класса thread другому объекту этого класса. Тогда первый объект уже не будет связан с потоком, владеть им будет только второй объект. В общем случае передачу владения можно выполнить с помощью алгоритма стандартной библиотеки C++ move. Например,

```
void fun3();


thread   t1( fun3 );

thread   t2;

t2 = move( t1 );
```

II. Асинхронные вызовы

Класс thread предоставляет низкоуровневый интерфейс для создания потока команд и запуска функциональной сущности в этом потоке. Стандарт C++11 предлагает также высокоуровневую возможность запуска задач не в последовательном, но в асинхронном режиме. Эту возможность дают функция async и шаблон класса future<>. Например, если необходимо выполнить сложные вычисления используя некоторые исходные данные и получить результат типа int, можно создать функцию int fun(float a[], int count) и запустить ее на выполнение в отдельном потоке, продолжая другие вычисления в текущем потоке команд. Вопрос заключается в том, как получить результат, который вернет функция fun завершив свои вычисления. На момент запуска функции результата еще нет, он появится когда-то в будущем. Здесь и помогает класс, построенный по шаблону future, параметр которого заменяется типом возвращаемого функцией значения. Объект такого класса возвращается функцией async. А в качестве фактических аргументов ей передаются функция, запускаемая асинхронно по отношению к вычислениям в текущем потоке и конкретные значения ее аргументов:

```
float x[1024]{};
```

```
int main() { // выполняется в текущем потоке команд
        …
        future<int> f = async(fun, x, 1024);
        …
}
```

Когда текущему потоку понадобится результат, сгенерированный функцией fun, в нем нужно вызвать метод get() класса future.

```
int result = f.get();
```

Если к этому моменту функция fun еще не закончила свою работу (или даже еще не была запущена на выполнение) текущий поток будет ожидать ее завершения.

III. Использование WinAPI для создания и управления потоками MS Windows.

В Windows каждый поток связан с уникальным целочисленным идентификатором, но управляется с помощью дескриптора, имеющего тип HANDLE. Поток создается функцией WinAPI с именем CreateThread. CreateThread получает шесть аргументов: атрибуты защиты, размер стека для созданного потока, указатель на функцию потока, параметр, передаваемый функции потока, флаги создания (обычно 0), адрес переменной, в которую будет помещено значение идентификатора потока. Функция CreateThread возвращает дескриптор (хэндл) вновь созданного потока или NULL, если поток создать не удалось. Хэндл нужно освободить, когда он уже не нужен, с помощью функции CloseHandle.

Функция потока должна иметь следующую сигнатуру:

```
DWORD WINAPI  fun(LPVOID param);
```

Здесь DWORD синоним unsigned int, а LPVOID - синоним void *.

Чтобы заставить порождающий поток ждать завершения порожденного или порожденных потоков (точнее перехода соответствующих объектов в сигнальное состояние) можно использовать функции WaitForSingleObject и WaitForMultipleObjects.

```
WaitForSingleObject( HANDLE h, DWORD milliseconds );
```

В качестве второго параметра нужно задать либо достаточно большое значение, либо константу INFINITE.

WaitForMultipleObjects(DWORD count, HANDLE *hPtr, BOOL wait, DWORD milliseconds);

Ожидает перехода в сигнальное состояние count объектов ядра, дескрипторы которых помещены в массив hPtr. Если значение параметра wait TRUE, то ожидается переход всех объектов, если FALSE – какого-нибудь из этих объектов.

Например,

```
#include <windows.h>

#include <iostream>


using namespace std;


int x = 1;


DWORD WINAPI fun(LPVOID) {
	x = 100;
	return 0;
}


int main() {

	HANDLE hThread;

	DWORD id;

	hThread = CreateThread(NULL, 0, fun, 0, 0, &id);

	WaitForSingleObject(hThread, INFINITE);

	/*	или

		WaitForMultipleObjects(1, &hThread, TRUE, INFINITE);

	*/
```

```
    CloseHandle(hThread);


    cout<<"x="<<x<<endl; // x=100


    return 0;
}
```

IV. Задания.

1. Создайте класс Matrix, реализующий понятие "матрица действительных чисел". Класс должен поддерживать инициализацию, заполнение, присваивание, копирование, вывод матрицы на консоль, а также методы, обеспечивающие последовательные версии сложения, вычитания и умножения матриц.

2. Создайте программу для тестирования класса Matrix. В процессе тестирования нужно генерировать матрицы разного размера с помощью генератора псевдослучайных чисел. Для сгенерированных матриц разного размера протестируйте работу операций сложения и умножения, замеряя при этом время выполнения этих операций.

3. Создайте для класса Matrix параллельные версии операций сложения и умножения, используя потоки C++11.

4. Создайте для класса Matrix параллельные версии операций сложения и умножения, используя асинхронные вызовы C++11.

5. Протестируйте параллельные версии операций сложения и умножения для матриц того же размера, для которых тестировались последовательные версии этих операций. Измерьте время выполнения операций, сравните с последовательными версиями и сделайте выводы.

6. Создайте и протестируйте параллельные версии операций сложения и умножения для матриц для потоков MS Windows на основе WinAPI.

V. Контрольные вопросы.

1. Как в C++11 реализованы потоки команд?
2. Как запустить функцию на выполнение в новом потоке команд в C++11?

3. Как организовать ожидание завершения вызванного потока команд в вызвавшем потоке в C++11?
4. Как различать потоки команд в C++11?
5. Как создать поток команд с помощью Win API?
6. Как организовать ожидание завершения вызванного потока команд в вызвавшем потоке с помощью Win API?
7. Как различать потоки команд созданные с помощью WinAPI?

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.