Лабораторная работа №3. Многопоточность в языке C++

3.1 Цель лабораторной работы

Получить навыки работы с потоками в языке C++.

3.2. Теоретический материал

3.2.1 Создание потоков в языке C++

Начиная со стандарта C++11 в стандартную библиотеку языка C++ была включена библиотека многопоточности, которая содержит потоки (thread), взаимные исключения (mutex) и условные переменные (condition_variable). Также появилась поддержка асинхронного программирования. В стандартах C++14, C++17 и C++20 возможности многопоточного и асинхронного программирования были расширены.

Перед созданием потоков необходимо подключить заголовочный файл thread (`#include <thread>`) и создать объект потока.

```
std::thread myThread; //создает объект потока, не создавая поток выполнения
```

Далее запускаем поток на выполнение

```
myThread = std::thread(foo);
```
//конструктор создает поток и связывает его с потоком выполнения, **`foo`** – функция, которую выполняет поток.

Если необходимо дождаться завершения «дочернего» потока, то используют функцию `join` (вызов данной функции блокирует текущий поток до завершения «дочернего»). В противном случае необходимо отсоединить созданный поток от «родительского» функцией `detach` (переводит созданный поток «в свободное плавание», «дочерний» поток завершается при завершении «родительского»). Второй вариант используется редко.

Чтобы обратиться внутри функции к потоку, который её выполняет, можно использовать `std::this_thread`.

Пример 1. В следующем примере создается один поток, который выводит сообщение о своем старте, затем выводит свой id 10 раз, после каждого вывода засыпает на 1 секунду (имитация сложных вычислений) и выводит сообщение о

завершении. Главный поток ждёт окончания выполнения дочернего и после этого также завершается.

```
void foo(void)
{
	std::cout << "Thread start..." << std::endl;

	for (int i = 0; i < 10; ++i)
	{
		std::cout << "Thread id = " << std::this_thread::get_id()
<< std::endl;
		std::this_thread::sleep_for(std::chrono::seconds(1));
	}
	std::cout << "Thread finish!" << std::endl;

	return;
}
int main()
{
	std::thread myTh(foo);
	std::cout << "Main thread id = " << std::this_thread::get_id()
<< std::endl;
	myTh.join();
}
```

Замечание. Функция `sleep_for` приостанавливает выполнение на заданный промежуток времени.

Пример 2. Модифицируем программу таким образом, чтобы поток выводил свой id *N* раз. Число *N* вводится пользователем в главной функции и может быть передано потоку в качестве параметра.

```
void foo(int cnt)
{
	std::cout << "Thread start..." << std::endl;

	for (int i = 0; i < cnt; ++i)
	{
		std::cout << "Thread id = " << std::this_thread::get_id()
<< std::endl;
		std::this_thread::sleep_for(std::chrono::seconds(1));
	}
	std::cout << "Thread finish!" << std::endl;

	return;
}
int main()
{
	int N;
	std::cout << "Input N: ";
	std::cin >> N;
```

```
        std::thread myTh(foo, N);

        std::cout << "Main thread id = " << std::this_thread::get_id()
    << std::endl;

        myTh.join();
    }
```

3.2.2 Простейшая синхронизация потоков в языке C++

Так как потоки выполняются в одном и том же адресном пространстве (имеют общую память), то возникает необходимость конкурентного доступа к разделяемому ресурсу. Если запустить на выполнение программу из примера 2, то можно заметить, что потоки конкурируют за доступ к объекту потокового вывода. В итоге вывод может выглядеть не так, как предполагал программист.

В данном случае вывод нужно поместить в критическую секцию. В таком случае новый поток не сможет начать вывод до того, как предыдущий выведет сообщение целиком.

Наиболее простой механизм – взаимоблокировка потоков (mutex, от mutual exclusion).

Метод `lock` блокирует мьютекс, метод `try_lock` пытается заблокировать мьютекс. Для разблокировки используется метод `unlock`.

Вызывающий поток владеет мьютексом со времени успешного вызова `lock` или `try_lock` и до момента вызова `unlock`. Пока поток владеет мьютексом, все остальные потоки не могут получить доступ к ресурсу (блокируются при вызове `lock` или получают `false` при вызове `try_lock`). Разблокировать мьютекс может только тот поток, который его захватил. Этим мьютекс отличается от бинарного семафора.

Рекомендуется использовать класс `lock_guard`, который реализует принцип RAII (Resource acquisition is initialization – получение ресурса есть инициализация) и является «оберткой» для `mutex`. При создании объекта `lock_guard` захватывается мьютекс, переданный ему в конструкторе. В деструкторе, же, происходит освобождение мьютекса. Также, `lock_guard` содержит дополнительный конструктор, который позволяет инициализировать объект

`lock_guard` с мьютексом, который уже был захвачен. В случае, если нужно иметь возможность разблокировать мьютекс, используйте объект `unique_lock`.

Захват мьютекса изменяет его состояние, поэтому мьютекс нужно передавать в функцию потока по ссылке (`ref`) или делать глобальной переменной.

Пример 3. Добавим в предыдущий пример второй дочерний поток и модифицируем его так, чтобы вывод мог осуществлять только один поток в один момент времени.

```
void foo(int cnt, std::mutex& mx)
{
	std::cout << "Thread start..." << std::endl;

	for (int i = 0; i < cnt; ++i)
	{
		std::this_thread::sleep_for(std::chrono::milliseconds(
rand()%1000));
		std::lock_guard<std::mutex> lgmx(mx);
		std::cout << "Thread id = " << std::this_thread::get_id()
<< std::endl;
	}
	std::cout << "Thread finish!" << std::endl;

	return;
}
int main()
{
	int N;
	std::cout << "Input N: ";
	std::cin >> N;

	std::thread myTh1, myTh2;
	std::mutex mx;
	myTh1 = std::thread(foo, N, ref(mx));
	myTh2 = std::thread(foo, 10, ref(mx));

	std::unique_lock<std::mutex> ulmx(mx);
	std::cout << "Main thread id = " << std::this_thread::get_id()
<< std::endl;
	ulmx.unlock();

	myTh1.join();
	myTh2.join();
}
```

Вопрос: подумайте, почему вызов join происходит в самом конце. Что будет, если вызвать join для первого потока перед созданием второго?

3.3. Задание на лабораторную работу

0. Запустить на выполнение пример 1. Заменить join на detach. Прокомментировать результат.

1. Разработать программу, выводящую числа от 1 до 100 в двух потоках, в первом – чётные, во втором – нечётные. Запустить несколько раз на выполнение. Вопрос. В каком порядке будут выводиться числа в консоли? Почему?

2. Разработать программу, генерирующую P потоков (число $P \leq 10$ задаётся при запуске), каждый из которых сначала выводит сообщение о старте, а потом выводит свой номер 100 раз. Количество потоков P задает пользователь с клавиатуры. Не забудьте имитировать сложные вычисления в потоке. Длительность сложных вычислений – случайное число миллисекунд из диапазона [1000, 2000].

3. Создать P потоков (число $P \leq 10$ задаётся при запуске), конкурирующих за общий ресурс – целочисленную переменную. Каждый поток увеличивает значение переменной на свой индекс. Потоки должны завершиться после того как значение общей переменной превысило 100 (начальное значение – 0). Организовать корректное взаимодействие потоков с использованием объекта `std::mutex`.

4. Написать программу, которая каждый элемент массива размера N заменяет на его наибольший простой делитель. Число N задается пользователем. Элементы массива – случайные натуральные числа из диапазона $[10^5, 10^6]$. Распараллелить с использованием `std::thread`. Замерить время работы программы для $N = 2 \cdot 10^7$, $5 \cdot 10^7$ и 10^8 на 1, 2, 4 и 8 потоках. Для проверки результатов реализовать данную задачу последовательно, сравнить результаты.

Замечание. Размер тестируемых массивов выбирать максимально возможным. Размеры приведены для компьютеров университета. Для более мощных компьютеров размеры тестируемых массивов нужно увеличить!

На каждом примере запустить не менее 3 раз. В таблицу занести среднее время выполнения на одном примере в секундах.

Таблица 1 – Время обработки массивов в секундах

Размерность	**Число потоков**			
	1	**2**	**4**	**8**
$2 \cdot 10^7$				
$5 \cdot 10^7$				
10^8				

Замечание. Если N не делится на число потоков, то нужно очень аккуратно раздавать нагрузку! Проверяйте, что у Вас два потока одновременно не делают одно и то же, также необходимо проверить, что нет «отдыхающих» потоков.

Вычислить ускорения для каждого значения N (заполнить таблицу 2) и построить диаграмму зависимости ускорения от числа потоков (три графика на одной диаграмме).

Замечание. *Ускорение* вычисляется как отношение времени работы последовательной программы к времени работы параллельной программы.

Таблица 2 – Ускорение параллельного суммирования массивов в секундах

Размерность	**Число потоков**		
	2	**4**	**8**
$2 \cdot 10^7$			
$2 \cdot 10^7$			
10^8			

5. Разработать программу, включающую в себя последовательный и параллельный алгоритм вычисления произведения квадратной матрицы на вектор. Каждый алгоритм реализуется в отдельном методе. Матрица и вектор генерируются некоторым способом по заданной размерности N. Параллельный алгоритм должен учитывать доступное ему количество процессоров. Программа запрашивает у пользователя размерность N, после чего выводит время вычислений для последовательного и параллельного алгоритмов. Сами матрицы выводить не нужно.

Провести тестирование программ на матрицах размерности N = 5000, 10000 и 20000. На каждом примере запустить не менее 3 раз. В таблицы занести среднее время выполнения на одном примере в секундах и ускорение. Построить диаграмму зависимости ускорения от числа потоков (три графика на одной диаграмме).

Таблица 3 – Время выполнения алгоритма умножения матрицы на вектор

Размерность	**Алгоритм**	
	последовательный	**параллельный**
5000		
10000		
20000		

6. Написать программу, вычисляющую сумму чисел от 1 до N. Число N вводится как параметр командной строки. Замерить время работы программы. Результат вывести на консоль.

Распараллелить вычисление суммы с использованием `std::thread`. Результат, полученный каждым потоком, записать в разделяемую переменную. Для синхронизации потоков использовать `std::mutex`.

Замерить время работы программы для N = **10^7, 10^8 и 10^9** на 1, 2, 4 и 8 потоках. На каждом примере запустить не менее 3 раз. В таблицы занести среднее время выполнения на одном примере в секундах и ускорение. Построить диаграмму зависимости ускорения от числа потоков (три графика на одной диаграмме).

3.4. Результаты лабораторной работы

Результаты лабораторной работы представляются в виде отчета по лабораторной работе. В отчет включается титульный лист, цель работы, задание на лабораторную работу, описание и обоснование правильности алгоритма, листинг с комментариями, скриншоты, доказывающие правильность работы программы, полученные результаты и выводы по лабораторной работе.

Пример оформления титульного листа приведен на следующей странице.

Отчет оформляется в электронном виде и высылается на e-mail vbyzov.vyatsu@gmail.com (в теме или тексте письма, а также в названии документа с отчетом должны фигурировать ФИ студента, его группа, номер лабораторной работы).

Лабораторная работа считается зачтенной после её устной защиты у преподавателя.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Отчёт по лабораторной работе №3

по дисциплине «Параллельное программирование»

Многопоточность в языке C++

Выполнил: студент группы ФИб-4301-51-00 _____ / И.И. Иванов /

Проверил: ст.преподаватель каф. ПМиИ _____ / В.А. Бызов /

Киров 2021