Лабораторная работа №4
Тема: Подключение внешних модулей и применение параллельного подхода

Задание.

1. Выделить в отдельный файл функции: расчёта функционала, пересчёта новой позиции точки (вспомогательные, если есть).
2. Реализовать их в виде подключаемых библиотек (создать заголовочный файл *.h и пр.)
3. Подключить вашу библиотеку к новому проекту(#include).
4. Создать в новом проекте популяцию из 1000 элементов (A[1000][3]).
5. Реализовать популяционный алгоритм поиска экстремума. Реализация должна использовать метод параллельных потоков. Распараллеливание алгоритма предлагается сделать в месте пересчёта новых значений точек (т.к. их количество 1000 и изменение каждой из них не зависит от других).

Алгоритм поиска экстремума, используя популяцию:
1. Инициализировать: Популяцию, Массив, хранящий текущее значение функции для этой точки. Задать стартовое значение шага и параметр его затухания.
2. Для каждого элемента пытаемся передвинуть точку в новое положение (см. пред. Лаб. работу), т.е. если значение функции будет улучшено происходит изменение положения точки, иначе возвращается предыдущее значение точки (вызов библиотечной функции // можно вспомнить: передача массива в/из функций).
3. После движения всех точек изменить шаг (см. пред. Лаб.работу).
4. Проверка точки Останова, если выполняется, завершаем алгоритм, иначе переходим к п.2. Условие Останова: значение шаг стало меньше минимального. Решением задачи будет: точка из популяции, которой соответствует наименьшее значение функции.

Примечание 1. Добавление и реализация статических библиотек C/C++.

Для создание своего подключаемого модуля (библиотеки). Необходимо реализовать, в соответствии с подходом разделения описания и реализации, файлы *.h (описание) и *.cpp (.c) (реализация).
Например, были реализованы функции:

```
double function(double x1, double x2)
{
   // какие-то вычисления
   return xx;
}

double* functionUpdate(double array[])
{
   // …
   // тысячи строк спустя
   return pointer;
}
```

Что бы дальше использовать эти функции (классы) в других проектах, нужно реализовать заголовочный файл, например mylib.h:

```
mylib.h :
// ----------------------
#ifndef MYLIB_H
#define MYLIB_H

double function(double x1, double x2);
double* functionUpdate(double x[]);

#endif /* MYLIB_H */
// ----------------------
```

как видите там содержаться объявление всех функций (и классов), которые будут описаны области реализации и могут быть использованы в проекте, к которому мы подключим эту библиотеку.

Тогда файл mylib.cpp выглядит:

```
// ----------------------
// some includes
#include <iostream>
#include <math.h>

// подключим наш заголовок
#include "mylib.h"

double function(double x1, double x2)
{
   // какие-то вычисления
   return xx;
}
```

```
double* functionUpdate(double array[])
{
   // …
   // тысячи строк спустя
   return pointer;
}
// ----------------------
```

В итоге мы в нашем проекте можем с помощью директивы include подключить данную библиотеку.
#include "mylib.h"

Примечание 2. Подключение классов в Java.

Для того, чтобы найти класс по имени когда вы его вызываете, в Java существует стандартный загрузчик классов. Он оперирует понятием classpath. Список classpath — это набор путей, где следует искать файлы классов. Каждый classpath может указывать как на директорию, так и на так называемый jar-файл. По-умолчанию в classpath входят файлы стандартных библиотек и директория, из которой вы вызвали саму Java.
Пакет (package) представляют собой набор классов, объединённых по смыслу. Пакеты обычно вкладываются друг в друга, образуя иерархию, дающую понять, что зачем. На файловой системе такая иерархия выглядит в виде вложенных друг в друга директорий с исходниками.

Так, исходники пакета A лежат в папке B, исходники пакета A.B — в папке A/B и так далее. Типичный путь к пакету выглядит примерно так:

org.apache.commons.collectons.

Чтобы использовать какой-то класс в коде другого класса, вы должны импортировать его, написав до объявления класса строку:

import путь.к.классу.ИмяКласса;

Кроме того, если вы используете классы одного пакета часто, вы можете импортировать весь пакет:

import путь.к.классу.*;

Это относится ко всем пакетам, кроме java.lang — он импортирован по умолчанию, именно из него были в прошлом примере взяты классы System и String. На самом деле они лежат в некоем jar'e, в каталоге java/lang.

Организация кода

В серьёзном проекте вы всегда должны будете разложить свои классы по пакетам. Обычно корневые пакеты создаются такими, чтобы ясно давать понять к чему он относится.

Например:

ru.project.photomaker выбирается корневым пакетом

ru.project.photomaker.core сюда мы пишем классы, отвечающие за логику

ru.project.photomaker.visual сюда, все наши визуальные объекты для приложения

и так далее.

Чтобы создать класс

ru.project.photomaker.Paint

вы должны:

создать файл Paint.java в папке ru/project/photomaker/

прописать в нём первой строчкой (точнее говоря, до импортов):

package ru.project.photomaker;

Примечание 3. Многопоточное программирование.

3.1. C/C++ на примере OpenMP

OpenMP (http://openmp.org) - API, предназначенное для программирования многопоточных приложений на многопроцессорных системах с общей памятью. Разработку спецификации OpenMP ведут несколько крупных производителей вычислительной техники и программного обеспечения. OpenMP поддерживается основными компиляторами (http://openmp.org/wp/openmp-compilers/ , см. «OpenMP Compilers - OpenMP.pdf»).

В OpenMP вы «не увидите» потоки в коде. Вместо этого, вы сообщаете компилятору с помощью директив #pragma, что блок кода может быть распараллелен. Зная данную информацию, компилятор в состоянии сгенерировать приложение, состоящее из одного главного потока, который создаёт множество других потоков для параллельного блока кода. Эти потоки синхронизируются в конце параллельного блока кода, и мы снова возвращаемся к одному главному потоку.

Так как OpenMP контролируется #pragma, то код на C++ корректно скомпилируется любым компилятором C++, потому что неподдерживаемые #pragma должны игнорироваться. Однако OpenMP API содержит также несколько функций, и, чтобы ими воспользоваться, необходимо подключить заголовочный файл.

Самый легкий способ определить, поддерживает ли компилятор OpenMP - попробовать подключить файл omp.h:

```
#include <omp.h>
```

Для компиляции программы с использованием библиотеки openmp нужно настроить используемую вами среду, зачастую нужно зайти в настройки проекта и найти либо область дополнительных параметров компилятора, либо найти где фигурирует название openmp. Например,

- Для Microsoft VS 2010 для подключения библиотеки openmp нужно зайти «Проект»-«Свойства»-«C/C++»-«Язык»-«Поддержка Open MP» и выбрать пункт меню «Да /openmp»

Если openMP поддерживается, тогда нужно его включить с помощью параметров компиляции:

gcc -fopenmp

Intel -openmp (linux, Mac) -Qopenmp (windows)

Microsoft -openmp

1. Программа HelloWorld:

```
#include <iostream>

int main()
{
   #pragma omp parallel
   {
      std::cout << "Hello World!\n";
   }
}
```

Компиляция и запуск:

```
user@ubuntu g++ test1.cpp -fopenmp
user@ubuntu ./a.out
Hello World!
```

Hello World!
Hello World!

Когда проект включает библиотечный файл, хранящийся в текущей деректории, можно компилировать:
user@ubuntu g++ proj.cpp mylib.cpp –fopenmp

2. Параллельные циклы

```
// 1
#pragma omp parallel
{
// 2
#pragma omp for
for ( int  n=0; n<10; ++n)
{
   cout<<n<<" "<<endl;
 }
}
```

Данный цикл выведет числа от 0 до 9 по одному разу, однако порядок будете неизвестен:
0 5 6 1 7 2 8 3 9 4

В примере
- блок 1: parallel — создает новую группу потоков.
- блок 2: for - в текущей группе потоков реализует параллельное вычисление цикла for.

Можно записать короче:

```
#pragma omp parallel for

for ( int  n=0; n<10; ++n)
{
   cout<<n<<" "<<endl;
}
```

3. Что бы задать количество потоков можно воспользоваться параметром num_threads():
Например создадим пул из 3ёх потоков:

```
#pragma omp parallel num_threads(3)
{
#pragma omp for
for ( int  n=0; n<10; ++n)
{
   cout<<n<<" "<<endl;
 }
}
```

4. Планирование.
Можно контролировать то, каким образом потоки будут загружаться работой при обработке цикла. Существует несколько вариантов.

```
#pragma omp for schedule(static)
```

static является вариантом по умолчанию. Ещё до входа в цикл каждый поток «знает», какие части цикла он будет обрабатывать.
Второй вариант - ключевое слово dynamic:

#pragma omp for schedule(dynamic)

В данном случае невозможно предсказать порядок, в котором итерации цикла будут назначены потокам. Каждый поток выполняет указанное число итераций. Если это число не задано, по умолчанию оно равно 1. После того как поток завершит выполнение заданных итераций, он переходит к следующему набору итераций. Так продолжается, пока не будут пройдены все итерации. Последний набор итераций может быть меньше, чем изначально заданный. Такой вариант очень полезен, когда разные итерации цикла обсчитываются разное время.

#pragma omp for schedule(dynamic, 3)

В данном примере, каждый поток выполняет три итерации, затем «берёт» следующие три и так далее. Последние, конечно, могут иметь размер меньше трёх.

5. Упорядочивание.
Порядок, в котором будут обрабатываться итерации цикла, вообще говоря, непредсказуем. Тем не менее, возможно «заставить» OpenMP выполнять выражения в цикле по порядку. Для этого существует ключевое слово ordered:
#pragma omp for ordered schedule(dynamic)
Если, например, поток выполнил седьмую итерацию, но шестую к этому моменту ещё не выполнил, поток будет ожидать выполнение шестой итерации. Разрешено использовать только один ordered блок на цикл.

Конечно это не полный перечень функций, но этого достаточно, что бы реализовать параллельную обработку критического участка кода, в небольшом проекте. Остальное можно изучить самостоятельно.
(см. «openmp-4.5.pdf» - офф. документация ноябрь 2015)

3.2. Java, Thread.
Каждый процесс имеет хотя бы один выполняющийся поток. Тот поток, с которого начинается выполнение программы, называется главным. В языке Java, после создания процесса, выполнение главного потока начинается с метода main(). Затем, по мере необходимости, в заданных программистом местах, и при выполнении заданных им же условий, запускаются другие, побочные потоки.

В языке Java поток представляется в виде объектапотомка класса Thread. Этот класс инкапсулирует стандартные механизмы работы с потоком. Запустить новый поток можно двумя способами:

1. Создать объект класса Thread, передав ему в конструкторе нечто, реализующее интерфейс Runnable. Этот интерфейс содержит метод run(), который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод run().

```
class   SomeThing      //Нечто, реализующее интерфейс  Runnable

implements   Runnable     //(содержащее метод  run ())
{
  public   void   run ()
  {
        //Этот метод будет выполняться в побочном потоке
        System.out.println("Привет из побочного потока!");
  }
}

public   class   Program
//Класс с методом  main ()
{
        static  SomeThing mThing; //mThing - объект класса, реализующего интерфейс Runnable

        public   static   void   main (String[] args)
        {
                mThing =  new  SomeThing();

                Thread myThready =  new  Thread(mThing);  //Создание потока "myThready"

                myThready.start();  // Запуск потока
                System.out.println("Главный поток завершён...");
        }
}
```

2. Создать потомка класса Thread и переопределить его метод run():

```
class MyThread extends Thread
{
        @Override
        public void run() //Этот метод будет выполнен в побочном потоке
        {
                System.out.println("Привет из потока!");

        }
```

```
}

public   class   Program
{
        static  MyThread mSecondThread;

        public static void main (String[] args)
        {
                mSecondThread =  new  MyThread();  //Создание потока

                mSecondThread.start();     //Запуск потока

                System.out.println("Главный поток завершён...");
        }
}
```

Для того чтобы запустить поток необходимо вызвать метод start().

```
Thread t = new Thread();
t.start();
```

Если вы вместо метода start() выполните метод run(), то run() выполнит свой код, но только в том же потоке, в котором и был вызван. Отдельный поток при этом не создастся.

Для того чтобы приостановить выполнение текущего потока необходимо выполнить статический метод sleep().

Данный метод задерживает поток на заданное время в миллисекундах и наносекундах, и имеет две перегруженные реализации:

sleep(long millis) - задает задержку в миллисекундах;
sleep(long millis, int nanos) – задает задержку в миллисекундах и наносекундах.

Приостановка потока, с передачей управления другому потоку производится статическим методом yield(). Метод не выбрасывает никаких исключений.

Thread.currentThread() – получает объект Thread текущего потока;

.getName() – получает имя потока. По умолчанию имя потока состоит из слова Thread и номера потока.

При создании потока, по умолчанию, задается приоритет родительского потока. Но вы всегда можете изменить заданный параметр приоритета с помощью метода setPriority().

Например:

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.setPriority(8);
```

В Java можно задать приоритет в диапазоне от 1 до 10.

Так же приоритет потока можно задать через определенные, в классе константы:

- Thread.MIN_PRIORITY – равняется самому низкому приоритету 1;

- Thread.NORM_PRIORITY – равняется среднему приоритету 5 (данный приоритет задан по умолчанию);

- Thread.MAX_PRIORITY – равняется самому высокому приоритету 10.

Узнать приоритет потока можно с помощью метода getPriority().

Устанавливая приоритеты потока, вы всегда должны помнить, что указание высокого или низкого приоритета не гарантирует, что поток, в очереди, будет выполнен раньше или позже. В данном случае планировщик потоков сам решает, какому потоку дать более высокий приоритет. Многое зависит от реализации потоков в операционной системе.

Чтобы определить состояние потока используется метод isAlive(). Если поток запушен или заблокирован, то возвращается значение true, если поток является созданным (еще не запущенным) или остановленным, то возвращается значение false. Определить, запущен поток или заблокирован – невозможно. Так же невозможно понять создан поток или остановлен.
Для выполнения потока необходимо дождаться завершения другого потока. В этих случаях вам поможет метод join(). Если поток вызывает метод join(), для другого потока, то вызывающий поток приостанавливается до тех пор, пока вызываемый поток не завершит свою работу.
Данный метод имеет две перегруженные реализации:

join() – ожидает пока вызываемый поток не завершит свою реализации;

join(long milis) – ожидает завершения вызываемого потока указанное время, после чего передает управление вызывающему потоку.

Вызов метода join() может быть прерван вызовом метода interrupt() для вызывающего потока, поэтому метод join() размещают в блоке try - catch.

Потоки-демоны
Любой поток, кроме main (главный) можно сделать потокомдемоном. Для этого необходимо перед запуском потока вызвать метод setDaemon(), с аргументом true.

```
Thread t = new Thread();

t.setDaemon(true);

t.start();
```

Основным назначением потока-демона является выполнение какой-то работы (например, таймер, проверка входящих сообщений) в фоновом режиме во время выполнения программы. При этом данный поток не является неотъемлемой частью программы. А это значит, что в случае завершения работы всех потоков, не являющихся демонами программа, завершает свою работу, не дожидаясь завершения работы потоков-демонов.

Чтобы узнать, является ли поток демоном, необходимо вызвать метод isDaemon(). Если поток

является демоном, то все потоки, которые он производит, также будут демонами.

Дальше для более детального изучения стоит рассмотреть модификаторы final и volatile, synchronized, java.util.concurrent (механизмы блокировок), механизм отложенного выполнения , Фреймворк Fork/Join (java 7) и т.д.