**Министерство образования и науки**

**Российской Федерации**

**Федеральное государственное бюджетное образовательное**

**учреждение высшего профессионального образования**

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)**

Кафедра автоматизированных систем управления(АСУ)

**РЕАЛИЗАЦИЯ МНОГОПОТОЧНОЙ ОБРАБОТКИ ДАННЫХ**

отчет по лабораторной работе №2

по дисциплине «Теория вычислительных процессов»

Вариант № 4

Выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

Проверил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

Томск— 2016

Оглавление

[1. Задание 3](#_Toc447732652)

[2. Анализ задачи 5](#_Toc447732653)

[2.1 Описание входных данных 5](#_Toc447732654)

[2.2 Описание данных 5](#_Toc447732655)

[3. Сеть Петри 9](#_Toc447732656)

[4. Реализация сети Петри 10](#_Toc447732657)

[5. Описание способов многозадачного анализа 11](#_Toc447732658)

[6. Пример работы программы 15](#_Toc447732667)

[Приложение А (Обязательное) Исходный код 16](#_Toc447732668)

# Задание

Цель выполнения лабораторной работы № 2 — освоить реализацию многопоточной обработки данных, а также пула потоков и механизма асинхронного ввода/вывода.

В работе необходимо реализовать многопоточную обработку массива структур данных (из N элементов) четырьмя способами:

1. При помощи массива из M потоков (M ≤ N), используя для синхронизации объект ядра — семафор.
2. При помощи массива из M потоков (M ≤ N), используя для синхронизации сеть Петри, моделирующую семафор.
3. При помощи пула из M потоков (M ≤ N), используя системный пул потоков или асинхронные потоки ввода/вывода.
4. При помощи пула из M потоков (M ≤ N), моделируя его при помощи сети Петри.

При обработке массива данных массивом потоков каждый поток либо заранее выбирает диапазон элементов массива данных, которые он будет обрабатывать, либо просто берет первый необработанный элемент. Завершив обработку одного элемента, поток переходит к обработке следующего.

При обработке массива данных пулом потоков, завершив обработку одного элемента массива данных, освободившийся в пуле поток переходит к обработке следующего необработанного элемента.

Чтобы не требовалось создавать слишком большие массивы (для которых эффект от параллельной обработки будет наиболее очевидным), можно имитировать ситуацию, когда обработка одного элемента массива требует больше процессорного времени, чем в действительности. Для этого после обработки очередного элемента массива поток может делать паузу на указанное количество миллисекунд.

Входной файл должен иметь имя «input.txt» или «input.xml».

Формат входного файла для лабораторной работы

|  |  |
| --- | --- |
| Поле | Значение |
| PA | Выбранный способ обработки массива |
| N | Размер массива структур данных (значения полей каждой структуры генерируются программой случайным образом) |
| M | Количество параллельных потоков (если 0, то принимается равным числу процессорных ядер в системе) |
| PT | Пауза после обработки каждого элемента массива (мс) |
| … | Дополнительные входные данные (зависят от варианта) |

Выходной файл должен иметь имя «output.txt». Формат вы- ходного файла представлен в таблице

|  |  |
| --- | --- |
| Поле | Значение |
| T1 | Время, требующееся на линейную обработку массива (без распараллеливания вычислений) |
| TP | Время, требующееся на параллельную обработку массива выбранным способом |
| … | Результаты обработки (зависят от варианта) |

Для ввода и вывода данных допускается использование в программе визуального интерфейса вместо файлового ввода/вывода.

Структура содержит данные о точке на плоскости (координаты и тип координат — полярные или декартовые). Требуется преобразовать все декартовые координаты в полярные, и наоборот. Также требуется определить точку, наиболее удаленную от точки с номером P. Вывести в выходной файл координаты этих точек и расстояние между ними.

# Анализ задачи

В данной задаче есть несколько частей, которые можно реализовывать по очереди.

* Чтение параметров.
* Выделение памяти и создание данных.
* Расчет алгоритма при простом линейном способе обхода данных.
* Простой многопоточный алгоритм, синхронизация с помощью семафора
* Многопоточный, с синхронизацией с помощью сети Петри
* Системный пул потоков
* Системный пул потоков с синхронизацией с помощью сети Петри.
* Итоговый анализ данных

Рассмотрим каждый из этих пунктов.

## 2.1 Описание входных данных

В данном пункте просто открывается файл и из него читаются числа-параметры. Все это реализуется в функции

void read\_file ()

При вызове этой процедуры открывается файл input.txt и читаются из него параметры, необходимые для работы.

## 2.2 Описание данных

Опишет структуру точки

typedef struct tagPoint

{

float x, y;

float r, a;

int typ;

int i;

} TPoint;

Данные, которые есть в структуре следующие:

* x, y – координаты в декартовых координатах
* r, a – координаты в полярных координатах
* typ – флаговый регистр, описывающий, какие координаты есть
  + 1 – есть декартовы
  + 2 – есть полярные
  + 3 – есть и те, и другие
* i – индекс сткрутуры в общем списке

При начальной инициализации в структуре заполняется только один тип координат, остальные остаются пустыми. Данные дополняются до верных при обработке

Для хранения всех структур в памяти заводится массив:

TPoint \*points

int point\_c;

Который хранит указатель на начало структуры. Переменная point\_c указывает количество точек.

Также есть переменная

TPoint \*p\_p;

int p\_i = 10;

Которая указывает номер точки, до которой искать расстояние.

Процедура инициализации следующая:

const float pi = 3.14159265;

// Создаем случайные точки

void generate\_points()

{

points = (TPoint\*) malloc( n \* sizeof(TPoint) );

memset( points, 0, n \* sizeof(TPoint) );

p\_p = &points[p\_i];

for( int i = 0; i < n; i++ )

{

points[i].typ = 1 << (rand() % 2);

points[i].i = i;

if (points[i].typ == 1)

{

points[i].x = (rand() % 1000 - 500) / 10.0f;

points[i].y = (rand() % 1000 - 500) / 10.0f;

}

else

{

points[i].r = (rand() % 500) / 10.0f;

points[i].a = (rand() % 3600) / 10.0f \* pi / 180.0;

}

}

}

Опишем обработки одного элемента. Так как обработка должна где-то хранить данные о предыдущих обработках, то заведем массив обработанных ранее данных и на вход будем подавать номер из этого массива

// Данные для каждой из цепочек

TPoint\* max\_p[10];

float max\_v[10];

Также заведем специальный массив, который будет нужен для APC-цепочек и который будет описывать каждой такой цепочке, когда завершится:

// Для APC цепочек

bool terminate[10];

Здесь поля описывают следующее:

* map\_p – указатель на максимальный элемент
* mav\_x – максимальное значение разницы, найденное ранее

Будем передавать индекс из этого массива в функцию обработки вместе с текущей обрабатываемой записью, таким образом каждая цепочка будет обрабатывать данные независимо. Но после завершения нескольких параллельных процессов итоговые данные надо обобщить, чтобы получить единый итог, а не несколько. Для этого в процедуре вывода результата сложим итоговые данные и найдем самые оптимальные:

// Вывод итогов

void display\_data( int threads )

{

// Собираем данные со всех цепочек и выводим общий итог в первой ячейке

for( int i = 1; i < threads; i++ )

{

if (max\_v[i] > max\_v[0])

{

max\_v[0] = max\_v[i];

max\_p[0] = max\_p[i];

}

}

TPoint \*p = max\_p[0];

printf( " Max distance\n from point %d (%f; %f)\n is point %d (%f; %f)\n Distance = %f\n",

p\_p->i, p\_p->x, p\_p->y, p->i, p->x, p->y, max\_v[0] );

}

В данной конкретной задаче она находит максимальный элемент из массива, но в общем виде может происходить и более сложный анализ.

Для обработки одного элемента данных используется следующая процедура

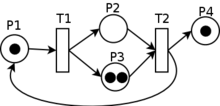
// Обрабатываем данные

void calc\_data( TPoint \*p, int ind )

Также она в ходе работы обеспечивает задержку для более реалистичной демонстрации пользы многозадачности.

# Сеть Петри

Сети Петри — математический аппарат для моделирования динамических дискретных систем. Впервые описаны Карлом Петри в 1962 году.

Сеть Петри представляет собой двудольный ориентированный мультиграф, состоящий из вершин двух типов — позиций и переходов, соединённых между собой дугами. Вершины одного типа не могут быть соединены непосредственно. В позициях могут размещаться метки (маркеры), способные перемещаться по сети.

Событием называют срабатывание перехода, при котором метки из входных позиций этого перехода перемещаются в выходные позиции. События происходят мгновенно, либо разновременно, при выполнении некоторых условий.

# Реализация сети Петри

В программе сеть Петри реализована с помощью двух классов:

class TCritPetry

{

private:

static int nodes[2];

static int moves\_out[2][2];

static int moves\_in[2][2];

void move( int m );

public:

void Enter ()

{

move(0);

}

void Leave()

{

move(1);

}

};

class TSemPetry

{

private:

static int nodes[3];

static int moves\_out[2][3];

static int moves\_in[2][3];

void move( int m );

public:

void Finished ()

{

move(0);

}

void Finish ()

{

move(1);

}

static void init\_sem\_petry( int n );

};

CRITICAL\_SECTION crit\_section\_petry;

У классов есть методы, описывающие переходы. У критической секции:

void Enter ()

void Leave()

У семафора:

void Finished ()

void Finish ()

Каждый из которых просто вызывает нужный переход в сети Петри.

# Описание способов многозадачного анализа

## Линейный анализ данных

Для сравнительного анализа данные в начале обсчитываются с помощью линейного анализа. Для этого процедура обработки по очереди вызывается для каждого элемента. Все данные накапливаются в первой ячейке массива хранения итогов.

## Анализ времени работы

В начале любой процедуры обработки засекается время работы.

void start\_timer()

В конце работы производится вызов

void end\_timer( char \*cap )

Который считает время от запуска start\_timer до вызова себя и показывает его на экране с сопровождающей записью.

## Выбор следующего элемента для анализа

Для простого многопоточного анализа каждой цепочке предоставляется регион, в котором она производит вычисления. Такие регионы описываются следующими массивами:

int start\_el[10], end\_el[10];

## Простой многопоточный алгоритм

В данном варианте работы создается нужное количество цепочек, каждая из которых просто линейно обрабатывает свой регион. Отличие от начальной однопоточной обработки только в том, что регионы для обработки меньше и каждая обработка происходит в отдельной цепочке.

Все цепочки работают до тех пор, пока есть свободные элементы.

Функция создания цепочек просто создает нужное количество цепочек, после чего ждет завершения всех цепочек. После завершения происходит суммирование результатов через функцию агрегации и вывод результата.

## Многопоточная обработка с помощью сети Петри

В данной части задачи необходимо использовать сеть Петри для обработки. Но ранее нам не нужна была эта сеть (или иные способы блокировки), так как цепочки имели непересекающиеся области.

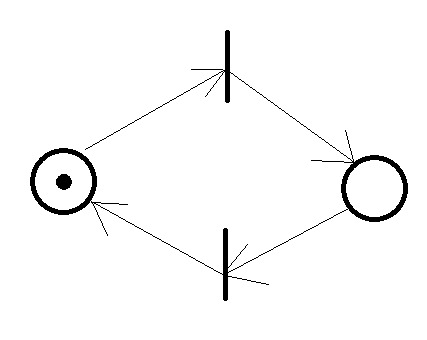
Сейчас создадим функцию

int next\_element()

которая возвращает следующий элемент в массиве. Каждая цепочка в цикле будет спрашивать следующий элемент, если он есть – обрабатывать, а если его нет – завершатся.

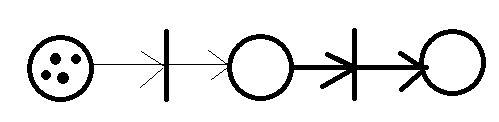
При таком изменении программы необходимо использовать один из способов блокировки доступа к этой функции. В данной части работы будет использоваться сеть Петри.

Для использования сети Петри в данной части работы используется сеть Петри со следующим графом:



Один узел данной сети считается блокирующим. Все цепочки при попытке входа в критическую секцию пытаются переместить фишку из левой части в правую, но это получается только у одной. После окончания обработки фишка попадает обратно в этот узел и следующая цепочка может попасть в критическую секцию.

Также сеть Петри использутся в семафоре, который контролирует завершение основной цепочки в соответствии со следюущей схемой:



Основная цепочка, после запуска всех дочерних цепочек пытается перейти из среднего узла в правый (с кратностью равной количеству цепочек), но не может – в узле нет фишек.

Каждая из рабочих цепочек обрабатывает данные, а после их завершения совершает один переход из левой ячейки в среднюю (с кратностью равной 1). После завершения всех цепочек в среднем узле накапливается достаточно большое количество фишек, чтобы совершить финальный переход.

## Многопоточная обработка с помощью системного пула потоков

В данном режиме основная цепочка потока работает в обычном режиме, а в моменты ее засыпания (Sleep или иное ожидание) цепочка может обрабатывать данные в дополнительной функции. Для этого используется функция QueueUserAPC, которая запускает асинхронную функцию в пределах другой цепочки.

В программе запускается нужное количество цепочек, каждая из которых просто спит, пока цепочке не будет требоваться завершение. Завершение описывается массивом terminate. После запуска всех цепочек в каждую из них запускается асинхронная функция, которая и делает в задаче основную работу. По завершению всех функций вызывается завершение всех цепочек и они прекращают работу.

## Многопоточная обработка сетью Петри и пулом потоков

Это комбинация двух предыдущих способов – потоки выполнения находятся в пользовательских асинхронных функциях (основная функция потока спит), а логика синхронизации находится в сети Петри, а не в семафоре при вызове функции get\_next.

## Выбор количества потоков

Если входной параметр m равен 0, то по условию задачи необходимо определить количество ядер процессора и использовать его, как количество цепочек. Для этого используется функция:

int get\_proc\_num()

{

SYSTEM\_INFO sysinfo;

GetSystemInfo( &sysinfo );

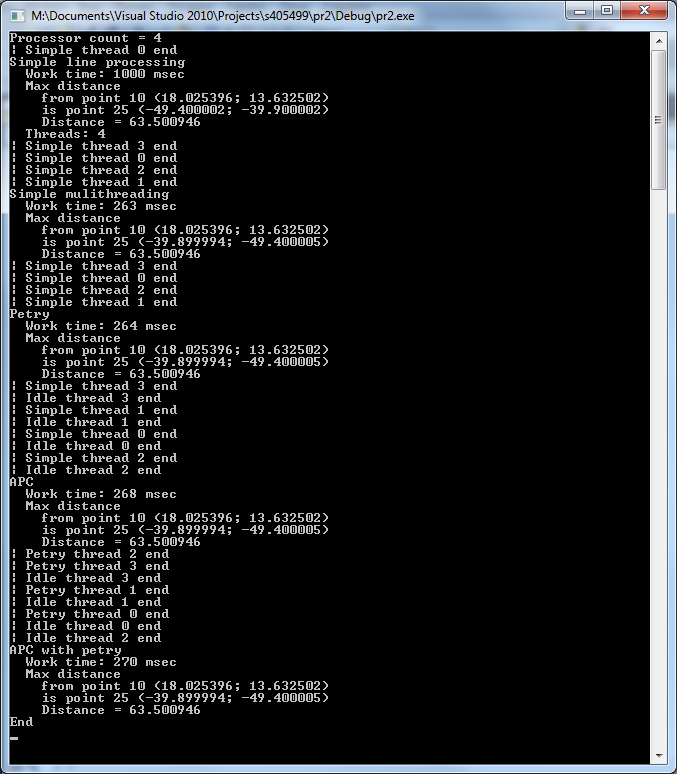
m = sysinfo.dwNumberOfProcessors;

printf( "Processor count = %d\n", m );

return m;

}

# Пример работы программы



Для отладки и информации каждая цепочка выводит при своем выходе сообщение о завершении.

# Приложение А (Обязательное) Исходный код

// pr2.cpp: определяет точку входа для консольного приложения.

//

#include "stdafx.h"

#include <Windows.h>

#include <math.h>

#include <ctime>

#include <conio.h>

typedef struct tagPoint

{

float x, y;

float r, a;

int typ;

int i;

} TPoint;

// Данные о точках

TPoint \*points, \*p\_p;

int point\_c,

p\_i = 10; // Какая точка центр для поиска максимума

int pa = 2, m=0, pt=10;

int n = 100;

bool terminate = false;

bool term\_thr[10];

// Узнать количество ядер

int get\_proc\_num()

{

SYSTEM\_INFO sysinfo;

GetSystemInfo( &sysinfo );

m = sysinfo.dwNumberOfProcessors;

printf( "Processor count = %d\n", m );

return m;

}

float sqr( float v )

{

return v\*v;

}

// Читаем параметры из файла

void read\_file ()

{

FILE \*in = fopen( "input.txt", "rt" );

fscanf( in, "%d", &pa );

fscanf( in, "%d%d%d", &n, &m, &pt );

fscanf( in, "%d", &p\_i );

fclose(in);

}

// Создаем случайные точки

void generate\_points()

{

points = (TPoint\*) malloc( n \* sizeof(TPoint) );

memset( points, 0, n \* sizeof(TPoint) );

p\_p = &points[p\_i];

for( int i = 0; i < n; i++ )

{

points[i].typ = 1 << (rand() % 2);

points[i].i = i;

if (points[i].typ == 1)

{

points[i].x = (rand() % 1000 - 500) / 10.0f;

points[i].y = (rand() % 1000 - 500) / 10.0f;

}

else

{

points[i].r = (rand() % 500) / 10.0f;

points[i].a = (rand() % 3600) / 10.0f \* M\_PI / 180.0;

}

}

}

// Данные для каждой из цепочек

TPoint\* max\_p[10];

float max\_v[10];

// Для APC цепочек

// bool terminate[10];

void reset\_calc()

{

memset( max\_p, 0, sizeof(max\_p) );

memset( max\_v, 0, sizeof(max\_v) );

terminate = false;

}

void calc\_data( TPoint \*p, int ind )

{

if (p->typ == 1)

{

p->typ = 3;

p->r = sqrt( p->x \* p->x + p->y \* p->y );

p->a = atan2f( p->x, p->y );

}

else // if (p->typ == 2)

{

p->typ = 3;

p->x = p->r \* cos( p->a );

p->y = p->r \* sin( p->a );

}

float l;

l = sqrt( sqr(p->x - p\_p->x) + sqr(p->y - p\_p->y) );

if (max\_v[ ind ] < p->r)

{

max\_v[ ind ] = p->r;

max\_p[ ind ] = p;

}

Sleep( pt );

}

// Вывод итогов

void display\_data( int threads )

{

// Собираем данные со всех цепочек и выводим общий итог в первой ячейке

for( int i = 1; i < threads; i++ )

{

if (max\_v[i] > max\_v[0])

{

max\_v[0] = max\_v[i];

max\_p[0] = max\_p[i];

}

}

TPoint \*p = max\_p[0];

printf( " Max distance\n from point %d (%f; %f)\n is point %d (%f; %f)\n Distance = %f\n",

p\_p->i, p\_p->x, p\_p->y, p->i, p->x, p->y, max\_v[0] );

}

// Время линейной обработки

int l\_time;

clock\_t start\_t, end\_t;

void start\_timer()

{

start\_t = clock();

}

void end\_timer( char \*cap )

{

end\_t = clock ();

clock\_t delta = end\_t - start\_t;

printf( "%s\n Work time: %d msec\n", cap, delta );

}

// Массивы указывают каждой цепочке, какую часть обработать.

// Последний элемент включается

int start\_el[10], end\_el[10];

DWORD \_\_stdcall thr\_simple( void \*ptr )

{

int thr\_i = (int) ptr, i;

for( i = start\_el[thr\_i]; i <= end\_el[thr\_i]; i++ )

calc\_data( &points[i], thr\_i );

// For APC

term\_thr[ thr\_i ] = true;

printf( "| Simple thread %d end\n", thr\_i );

return 0;

}

// Линейно обработать

void run\_no\_threads()

{

reset\_calc ();

reset\_calc();

start\_timer ();

start\_el[0] = 0;

end\_el[0] = n-1;

thr\_simple( (LPDWORD) 0 );

end\_timer ( "Simple line processing" );

display\_data( 1 );

}

// Запустить несколько простых цепочек

void run\_simple()

{

HANDLE threads[100];

reset\_calc();

start\_timer ();

start\_el[0] = 0;

int di = n / m;

for( int i = 0; i < m; i++ )

{

if (i == (m-1))

end\_el[i] = n-1;

else

{

end\_el[i] = start\_el[i] + di;

start\_el[i+1] = end\_el[i]+1;

}

threads[i] = CreateThread( NULL, 0, (LPTHREAD\_START\_ROUTINE) thr\_simple,

(LPDWORD) i, 0, NULL );

}

WaitForMultipleObjects( m, &threads[0], TRUE, INFINITE );

end\_timer( "Simple mulithreading" );

display\_data(1);

}

CRITICAL\_SECTION crit\_section\_petry;

class TCritPetry

{

private:

static int nodes[2];

static int moves\_out[2][2];

static int moves\_in[2][2];

void move( int m );

public:

void Enter ()

{

move(0);

}

void Leave()

{

move(1);

}

};

int TCritPetry::nodes[2] =

{ 1, 0 };

int TCritPetry::moves\_out[2][2] =

{

{ 1,0 },

{ 0,1 }

};

int TCritPetry::moves\_in[2][2] =

{

{ 0,1 },

{ 1,0 }

};

void TCritPetry::move( int m )

{

bool fl = false;

while (!fl)

{

EnterCriticalSection( &crit\_section\_petry );

fl = true;

for( int i = 0; i < 2; i++ )

if (nodes[i] < moves\_out[m][i])

fl = false;

// Перемещаемся

if (fl)

{

for( int i = 0; i < 2; i++ )

nodes[i] = nodes[i] - moves\_out[m][i] + moves\_in[m][i];

}

LeaveCriticalSection( &crit\_section\_petry );

if (!fl) Sleep(1);

}

}

class TSemPetry

{

private:

static int nodes[3];

static int moves\_out[2][3];

static int moves\_in[2][3];

void move( int m );

public:

void Finished ()

{

move(0);

}

void Finish ()

{

move(1);

}

static void init\_sem\_petry( int n );

};

int TSemPetry::nodes[3] =

{ 9,0,0 };

int TSemPetry::moves\_out[2][3] =

{

{ 1,0,0 },

{ 0,9,0 },

};

int TSemPetry::moves\_in[2][3] =

{

{ 0,1,0 },

{ 0,0,9 },

};

// Инициалиируем кратные вершины нужной кратностью

void TSemPetry::init\_sem\_petry( int n )

{

int i,j;

for( i = 0; i < 3; i++ )

{

if (nodes[i] == 9)

nodes[i] = n;

for( j = 0; j < 2; j++ )

{

if (moves\_in[j][i] == 9)

moves\_in[j][i] = n;

if (moves\_out[j][i] == 9)

moves\_out[j][i] = n;

}

}

}

void TSemPetry::move( int m )

{

bool fl = false;

while (!fl)

{

EnterCriticalSection( &crit\_section\_petry );

fl = true;

for( int i = 0; i < 3; i++ )

if (nodes[i] < moves\_out[m][i])

fl = false;

// Перемещаемся

if (fl)

{

for( int i = 0; i < 3; i++ )

nodes[i] = nodes[i] - moves\_out[m][i] + moves\_in[m][i];

}

LeaveCriticalSection( &crit\_section\_petry );

if (!fl) Sleep(1);

}

}

/\* class TPetry

{

public:

int petry\_1;

void move\_12()

{

EnterCriticalSection( &crit\_section\_petry );

petry\_1 = 0;

petry\_2 = 1;

LeaveCriticalSection( &crit\_section\_petry );

}

void move\_21()

{

while (petry\_1 == 0)

{

EnterCriticalSection( &crit\_section\_petry );

if (petry\_2 == 1)

{

petry\_1 = 1;

petry\_2 = 0;

}

LeaveCriticalSection( &crit\_section\_petry );

if (petry\_1 == 0) Sleep(10);

}

}

}; // \*/

int last\_el;

int next\_element()

{

if (last\_el < n)

return last\_el++;

return -1;

}

void \_\_stdcall thr\_petry( void \*ptr )

{

TCritPetry \*crit = new TCritPetry();

int thr\_i = (int) ptr, i;

while (true)

{

// printf( " petry %d enter\n", thr\_i );

crit->Enter();

i = next\_element();

// printf( " petry %d leave\n", thr\_i );

crit->Leave ();

if (i < 0) break;

calc\_data( &points[i], thr\_i );

}

// printf( " petry %d finish\n", thr\_i );

TSemPetry \*sem = new TSemPetry ();

sem->Finished();

}

// Запустить несколько простых цепочек

void run\_petry()

{

HANDLE threads[100];

reset\_calc();

start\_timer ();

for( int i = 0; i < m; i++ )

threads[i] = CreateThread( NULL, 0, (LPTHREAD\_START\_ROUTINE) thr\_simple,

(LPDWORD) i, 0, NULL );

/\* TSemPetry \*sem = new TSemPetry();

sem->Finish (); // \*/

WaitForMultipleObjects( m, &threads[0], TRUE, INFINITE );

end\_timer( "Petry" );

display\_data(1);

}

void \_\_stdcall thr\_idle( void \*ptr )

{

int thr\_i = (int) ptr, i;

while (!term\_thr[ thr\_i ])

SleepEx( 100, TRUE );

printf( "| Idle thread %d end\n", thr\_i );

}

void run\_apc()

{

HANDLE threads[10];

reset\_calc();

start\_timer ();

for( int i = 0; i < m; i++ )

threads[i] = CreateThread( NULL, 0, (LPTHREAD\_START\_ROUTINE) thr\_idle,

(LPDWORD) i, 0, NULL );

for( int i = 0; i < m; i++ )

QueueUserAPC( (PAPCFUNC) thr\_simple, threads[i], (ULONG\_PTR) i );

WaitForMultipleObjects( m, &threads[0], TRUE, INFINITE );

end\_timer( "APC" );

display\_data(1);

}

void run\_apc\_petry ()

{

HANDLE threads[10];

reset\_calc();

start\_timer ();

for( int i = 0; i < m; i++ )

threads[i] = CreateThread( NULL, 0, (LPTHREAD\_START\_ROUTINE) thr\_idle,

(LPDWORD) i, 0, NULL );

for( int i = 0; i < m; i++ )

QueueUserAPC( (PAPCFUNC) thr\_petry, threads[i], (ULONG\_PTR) i );

TSemPetry \*sem = new TSemPetry();

sem->Finish ();

terminate = true;

// WaitForMultipleObjects( m, &threads[0], TRUE, INFINITE );

end\_timer( "APC with petry" );

display\_data(1);

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

read\_file();

generate\_points();

InitializeCriticalSection( &crit\_section\_petry );

if (m == 0)

m = get\_proc\_num();

// Поправить семафор для количества цепочек

TSemPetry::init\_sem\_petry( m );

run\_no\_threads();

printf( " Threads: %d\n", m );

run\_simple ();

run\_petry ();

memset( term\_thr, 0, sizeof(term\_thr) );

run\_apc ();

memset( term\_thr, 0, sizeof(term\_thr) );

run\_apc\_petry ();

printf( "End\n" );

\_getch ();

return 0;

}