Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

**ОТЧЁТ**

к лабораторной работе

на тему

Управление потоками и нитями

Выполнил: студент группы 153505

Кудласевич Артур Иванович

Проверил: Сиротко Сергей Иванович

Минск 2023

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc146631498)

[2 Краткие теоретические сведения 4](#_Toc146631499)

[3 Результаты выполнения лабораторной работы 6](#_Toc146631500)

[Выводы 7](#_Toc146631501)

[Приложение А (обязательное) Листинг кода 8](#_Toc146631503)

**1 ПОСТАНОВКА ЗАДАЧИ**

Целью лабораторной работы является изучение концепции вычислительных потоков и ее реализация в Windows. Изучение основных этапов жизненного цикла потоков и элементарное управление ими: порождение, завершение, получение и изменение состояния. Изучение типичных ситуаций многопоточности, типичного использования многопоточности.

**2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

В процессоре может быть несколько вычислительных потоков, и каждый из них может выполнять свои инструкции независимо от других потоков. Таким образом, вычислительные потоки позволяют параллельно выполнять несколько задач, увеличивая общую производительность процессора.

Каждый вычислительный поток имеет свою набор регистров, счетчик команд и другие ресурсы, необходимые для выполнения инструкций. Кроме того, вычислительные потоки могут иметь доступ к общей памяти, что позволяет им обмениваться данными и совместно работать над решением задач.

Вычислительные потоки могут быть реализованы на аппаратном или программном уровне. В первом случае, каждый поток имеет свои собственные физические ресурсы, такие как регистры и функциональные блоки. Во втором случае, несколько потоков могут совместно использовать одни и те же аппаратные ресурсы, но выполнять разные программы.

Вычислительные потоки могут быть реализованы различными способами, включая аппаратную поддержку в виде многопоточных ядер или программную поддержку в виде многопоточных библиотек. В зависимости от конкретной архитектуры процессора, потоки могут быть организованы по разным принципам, таким как симметричное мультипоточное исполнение (SMT) или мультипоточность с полосной организацией.

В итоге, вычислительные потоки являются важной составляющей современных процессоров, позволяющей увеличить общую производительность и обеспечить более эффективное использование аппаратных ресурсов. Их использование позволяет сократить время выполнения задач и повысить общую производительность вычислительной системы.

Вычислительный поток процессора, или поток выполнения, представляет собой некоторую последовательность инструкций, которую процессор может выполнять независимо от других потоков. Каждый поток обрабатывает свою часть задачи и может быть выполнен параллельно с другими потоками.

Принцип работы вычислительного потока заключается в распределении нагрузки на несколько обработчиков или ядер процессора. Каждый поток процессора имеет свой собственный контрольный блок и набор регистров, что позволяет процессору выполнять несколько задач одновременно.

Потоки выполнения могут быть реализованы с помощью аппаратной многопоточности или программного механизма многопоточности. Аппаратная многопоточность подразумевает наличие нескольких физических вычислительных ядер в процессоре, каждое из которых может выполнять свои потоки параллельно. Программная многопоточность, с другой стороны, использует одно ядро для выполнения нескольких потоков, переключаясь между ними в зависимости от их приоритета.

Вычислительный поток процессора позволяет увеличить общую производительность системы, ускоряя выполнение задач, которые могут параллельно выполняться. Он нашел применение во многих современных компьютерных системах, таких как многопоточные микропроцессоры, многопроцессорные системы и графические процессоры.

**3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ     РАБОТЫ**

В рамках выполнения лабораторной работы было создано приложение, которое разбивает массив на несколько частей, сортирует каждую часть отдельным потоком и производит окончательную “сборку”.

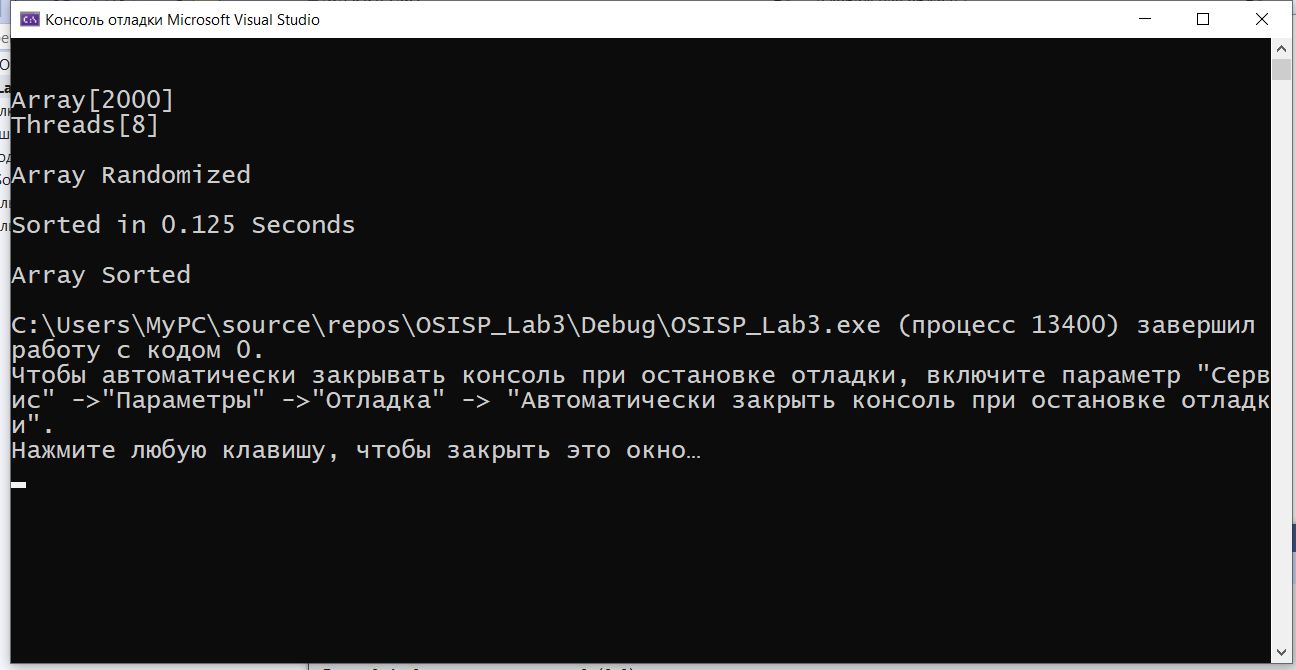


Рисунок 1 – Результат работы программы

**ВЫВОДЫ**

В результате выполнения данной лабораторной работы была разработана программа с использованием C++, которая разбивает массив на несколько частей, сортирует каждый отдельным потоком и окончательно реализует “сборку”.

**ПРИЛОЖЕНИЕ А**

## **(обязательное)**

**Листинг кода**

#include <iostream>

#include <time.h>

#include <thread>

using namespace std;

#if defined (WIN32) || (\_WIN64)

#include <windows.h>

#define pthread\_t DWORD

#define pthread\_create(THREAD\_ID\_PTR, ATTR, ROUTINE, PARAMS) CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)ROUTINE,(void\*)PARAMS,0,THREAD\_ID\_PTR)

#define sleep(ms) Sleep(ms)

#else

#include <pthread.h>

#include <unistd.h>

#endif

class QS

{

public:

int\* c\_array;

int c\_size;

int c\_thread\_id;

QS\*\* c\_threads;

QS(int\* array, int size, QS\*\* threads, int thread\_id)

{

c\_array = array;

c\_size = size;

c\_threads = threads;

c\_thread\_id = thread\_id;

threads[thread\_id] = this;

}

~QS()

{

c\_threads[c\_thread\_id] = nullptr;

}

};

void qsort(int\* array, const unsigned int size)

{

if (size <= 30)

{

// insertion sort algorithm

for (unsigned int i = 1; i < size; i++) {

int temp = array[i];

unsigned int j = i;

while (j > 0 && temp < array[j - 1]) {

array[j] = array[j - 1];

j--;

}

array[j] = temp;

}

}

else

{

// quick sort algorithm

int pivot = array[size / 2];

int\* left = array;

int\* right = array + size - 1;

while (true) {

while ((left <= right) && (\*left < pivot)) left++;

while ((left <= right) && (\*right > pivot)) right--;

if (left > right) break;

int temp = \*left;

\*left++ = \*right;

\*right-- = temp;

}

qsort(array, right - array + 1);

qsort(left, array + size - left);

}

}

void\* qsort\_thread(void\* obj)

{

qsort(((QS\*)obj)->c\_array, ((QS\*)obj)->c\_size);

delete ((QS\*)obj);

return nullptr;

}

struct RAND

{

int\* s\_array;

int s\_size;

RAND(int\* array, int size)

{

s\_array = array;

s\_size = size;

}

};

int RANDOM\_THREAD\_COUNT = 0;

void\* randomize\_thread(void\* obj)

{

RAND\* r = (RAND\*)obj;

for (int i = 0; i < r->s\_size; i++)

r->s\_array[i] = rand();

delete (RAND\*)r;

RANDOM\_THREAD\_COUNT--;

return nullptr;

}

// driver

int main(int argc, char\*\* argv)

{

char\* sz;

int MAX\_ARRAY\_ELEMENTS = 2000;

int MAX\_THREADS = thread::hardware\_concurrency();

// parse command line arguments

for (--argc, ++argv; argc > 0; --argc, ++argv)

{

sz = \*argv;

if (\*sz != '-')

break;

switch (sz[1])

{

case 'A': // array max

MAX\_ARRAY\_ELEMENTS = atoi(sz + 2);

break;

case 'T': // thread count

MAX\_THREADS = atoi(sz + 2);

break;

}

}

if (MAX\_THREADS < 1)

MAX\_THREADS = 1;

cout << "\n\nArray[" << MAX\_ARRAY\_ELEMENTS << "]\nThreads[" << MAX\_THREADS << "]";

// allocate the array itself

int\* g\_array = new int[MAX\_ARRAY\_ELEMENTS];

int len = MAX\_ARRAY\_ELEMENTS / MAX\_THREADS;

// generating random values in array

srand(clock());

// create threads for randomizing array

for (int i = 0, ai = 0; i < MAX\_THREADS; i++, ai += len)

{

RANDOM\_THREAD\_COUNT++;

pthread\_t t;

int size = len + (i == (MAX\_THREADS - 1) ? (MAX\_ARRAY\_ELEMENTS % MAX\_THREADS) : 0);

pthread\_create(&t, 0, randomize\_thread, new RAND(&g\_array[ai], size));

}

// join all current randomizing threads

while (RANDOM\_THREAD\_COUNT)

sleep(10);

cout << "\n\nArray Randomized";

// allocate memeory for threads

QS\*\* threads = new QS \* [MAX\_THREADS];

clock\_t time = clock();

// create threads for qsort

for (int i = 0, ai = 0; i < MAX\_THREADS; i++, ai += len)

{

threads[i] = nullptr;

pthread\_t t;

int size = len + (i == (MAX\_THREADS - 1) ? (MAX\_ARRAY\_ELEMENTS % MAX\_THREADS) : 0);

pthread\_create(&t, 0, qsort\_thread, new QS(&g\_array[ai], size, threads, i));

}

// join all current threads

for (int i = 0; i < MAX\_THREADS;)

{

sleep(10);

if (threads[i])

continue;

i++;

}

// now that threads are finished, do a final sort on the app's main thread

if (MAX\_THREADS > 1)

qsort(g\_array, MAX\_ARRAY\_ELEMENTS);

cout << "\n\nSorted in " << ((clock() - time) / 1000.0L) << " Seconds";

int last = 0;

for (int i = 0; i < MAX\_ARRAY\_ELEMENTS; i++)

{

if (g\_array[i] < last)

{

cout << "\n\nArray Not Sorted";

return 0;

}

last = g\_array[i];

}

cout << "\n\nArray Sorted";

if (MAX\_ARRAY\_ELEMENTS < 50)

for (int i = 0; i < MAX\_ARRAY\_ELEMENTS; i++)

cout << " " << g\_array[i];

cout << "\n";

delete[]g\_array;

delete[]threads;

}