Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

**УПРАВЛЕНИЕ ПОТОКАМИ**

Студент: Муратов Артём Алексеевич

Группа: М8О–212Б–22

Вариант: 2

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2023.

**Постановка задачи**

## Цель работы

Целью является приобретение практических навыков в:

* Управление потоками в ОС
* Обеспечение синхронизации между потоками

## Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При

обработки использовать стандартные средства создания потоков операционной системы

(Windows/Unix). Ограничение максимального количества потоков, работающих в один момент

времени, должно быть задано ключом запуска вашей программы.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей

программой с помощью стандартных средств операционной системы.

В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных

данных и количества потоков. Получившиеся результаты необходимо объяснить.

**Общие сведения о программе**

Программа компилируется из файла main.c. Для сборки используется CMake. Также используется заголовочные файлы:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы потоков в C.
2. Создать структуру ThreadInfo для потоков.
3. Реализовать в main.c
   1. Ввод с помощью ключей длину массива и макимальное число потоков
   2. Заполнение массива случайными величинами
   3. Разделение массива на части, присвоение частей потокам.
   4. Дождаться завершения работы потоков и объединить результаты их работы.
   5. Вывод отсортированного и изначального массивов.
   6. Измерение времени работы программы.
4. Создать соответствующие функции merge(), quicksort\_thread()

**Основные файлы программы**

**Main.c**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

#define MAX\_THREADS 4 // макс потоков по умолчанию

// структура для сортировки в каждом потоке

struct ThreadInfo

{

int\* array;

int left;

int right;

};

void quicksort(int\* array, int left, int right){

int i\_left = left;

int j\_right = right;

int pivot = array[(left + right) / 2];

do

{

while(array[i\_left] < pivot) i\_left++;

while(array[j\_right] > pivot) j\_right--;

if ( i\_left <= j\_right){

if (array[i\_left] > array[j\_right]){

int temp = array[i\_left];

array[i\_left] = array[j\_right];

array[j\_right] = temp;

}

i\_left++;

j\_right--;

}

} while (i\_left <= j\_right);

if (i\_left < right)

quicksort(array, i\_left, right);

if (left < j\_right)

quicksort(array, left, j\_right);

}

void\* quicksort\_thread(void\* arg){

struct ThreadInfo\* info = (struct ThreadInfo\*)arg;

quicksort(info->array, info->left, info->right);

pthread\_exit(NULL);

}

void merge(int\* array, int left, int right){

int i, j, k;

int left\_el\_num = left;

int size\_of\_chunk = right - left + 1;

int L\_array[left\_el\_num]; // массив для элементов левее куска потока i

int Chunk\_array[size\_of\_chunk]; // массив для элементов в куске потока i

for(int i = 0; i < left\_el\_num; i++){

L\_array[i] = array[i];

}

for(int j = 0; j < size\_of\_chunk; j++){

Chunk\_array[j] = array[left + j];

}

// Мешаем массивы для упорядочивания

i = 0;

j = 0;

k = 0;

while(i < left\_el\_num & j < size\_of\_chunk){

if (L\_array[i] <= Chunk\_array[j]) {

array[k] = L\_array[i];

i++;

} else {

array[k] = Chunk\_array[j];

j++;

}

k++;

}

// Копируем оставшиеся элементы L\_array

while(i < left\_el\_num) {

array[k] = L\_array[i];

i++;

k++;

}

// Копируем оставшиеся элементы Chunk\_array

while(j < size\_of\_chunk) {

array[k] = Chunk\_array[j];

j++;

k++;

}

}

int main(int argc, char\* argv[]){

if (argc < 2) {

printf("Недостаточно аргументов \n");

return 1;

}

int array\_size = atoi(argv[1]);

if (array\_size <= 0){

printf("размер массива не может быть меньше 0 \n");

return 1;

}

int\* array = malloc(sizeof(int) \* array\_size);

srand(time(NULL));

// заполняем случайными числами

for (int i = 0; i < array\_size; i++){

array[i] = rand() % 100;

}

printf("Исходный массив \n");

for(int i = 0; i < array\_size; i++){

printf("%d ", array[i]);

}

printf("\n");

int max\_threads = MAX\_THREADS;

if ( argc == 3){

if (atoi(argv[2]) > 0 ) {

max\_threads = atoi(argv[2]);

}

}

clock\_t start\_time = clock();

struct ThreadInfo thread\_info[max\_threads];

pthread\_t threads[max\_threads];

if(max\_threads > array\_size)

max\_threads = array\_size;

// разделяем массив на куски и распределяем их по потокам

int chunk\_size = array\_size / max\_threads;

for(int i = 0; i < max\_threads; i++){

thread\_info[i].array = array;

thread\_info[i].left = i \* chunk\_size;

thread\_info[i].right = (i + 1) \* chunk\_size - 1;

pthread\_create(&threads[i], NULL, quicksort\_thread, &thread\_info[i]);

}

// соединяем результаты в один массив

for(int i = 0; i < max\_threads; i++){

pthread\_join(threads[i], NULL);

merge(array, thread\_info[i].left, thread\_info[i].right);

}

clock\_t end\_time = clock();

double exec\_time = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

printf("Отсортированный массив \n");

for (int i = 0; i < array\_size; i++)

{

printf("%d ", array[i]);

}

printf("\n");

printf("Время выполнения программы %f \n", exec\_time);

free(array);

return 0;

}

**CMakeLists.txt**

cmake\_minimum\_required(VERSION 3.22)

project(lab\_os\_2)

set(CMAKE\_C\_STANDARD 17)

set(CMAKE\_C\_STANDARD\_REQUIRED ON)

add\_executable(main main.c)

**Пример работы**

**1.**

virtbox@virtbox-VirtualBox:~/OS\_labs/lab\_os\_1/build$ ./main 20

Исходный массив

31 45 65 87 39 33 88 62 2 6 24 22 33 14 61 83 28 76 74 42

Отсортированный массив

2 6 14 22 24 28 31 33 33 39 42 45 61 62 65 74 76 83 87 88

Время выполнения программы 0.000897

**2.**

virtbox@virtbox-VirtualBox:~/OS\_labs/lab\_os\_1/build$ ./main 150

Исходный массив

82 59 68 96 56 16 0 28 50 18 49 49 77 35 65 32 82 34 42 42 58 67 0 11 68 95 24 2 91 16 25 73 27 45 21 83 13 21 64 64 92 13 65 69 0 30 54 82 16 96 76 74 63 76 85 83 23 9 85 15 77 62 88 4 8 10 88 21 83 52 85 75 17 51 45 17 33 51 51 50 47 27 24 62 4 62 45 27 23 30 42 1 93 83 5 53 45 45 74 28 97 12 56 66 63 53 35 48 4 86 98 51 66 75 13 22 89 10 1 12 40 96 13 85 79 71 38 24 16 65 4 66 77 60 32 92 65 68 92 69 6 91 20 72 18 85 94 7 95 96

Отсортированный массив

0 0 0 1 1 2 4 4 4 4 5 6 7 8 9 10 10 11 12 12 13 13 13 13 15 16 16 16 16 17 17 18 18 20 21 21 21 22 23 23 24 24 24 25 27 27 27 28 28 30 30 32 32 33 34 35 35 38 40 42 42 42 45 45 45 45 45 47 48 49 49 50 50 51 51 51 51 52 53 53 54 56 56 58 59 60 62 62 62 63 63 64 64 65 65 65 65 66 66 66 67 68 68 68 69 69 71 72 73 74 74 75 75 76 76 77 77 77 79 82 82 82 83 83 83 83 85 85 85 85 85 86 88 88 89 91 91 92 92 92 93 94 95 96 96 96 97 98 95 96

Время выполнения программы 0.001109

**3.**

virtbox@virtbox-VirtualBox:~/OS\_labs/lab\_os\_1/build$ ./main 20 1

Исходный массив

54 5 96 0 21 98 31 30 98 75 59 67 13 2 17 89 93 72 28 40

Отсортированный массив

0 2 5 13 17 21 28 30 31 40 54 59 67 72 75 89 93 96 98 98

Время выполнения программы 0.000220

**4.**

virtbox@virtbox-VirtualBox:~/OS\_labs/lab\_os\_1/build$ ./main 20 20

Исходный массив

71 35 45 34 88 79 44 98 15 79 17 74 62 66 42 55 65 7 32 10

Отсортированный массив

7 10 15 17 32 34 35 42 44 45 55 62 65 66 71 74 79 79 88 98

Время выполнения программы 0.001911

**5.**

virtbox@virtbox-VirtualBox:~/OS\_labs/lab\_os\_1/build$ ./main

Недостаточно аргументов

**6.**

virtbox@virtbox-VirtualBox:~/OS\_labs/lab\_os\_1/build$ ./main -1

размер массива не может быть меньше 0

**Вывод**

В ходе выполнения данной работы я изучил многопоточное программирование на C.Первой проблемой стал неправильный подход к написанию алгоритма: я использовал обычную быструю сортировку, но для каждой из 2-х частей массива вызывал свой поток. При таком решении оказалось сложно ограничить число доступных потоков, как того требует задание. Из-за этого я решил перейти к другому подходу, разделению массива на куски, и создание потока для сортировки каждого куска, потом их объединение. Второй проблемой, с которой я столкнулся при создании программы, собственно объединение результатов работы различных потоков в один упорядоченный массив. Так как каждый поток сортирует лишь положенный ему кусок, после объединения всех потоков остается

частично упорядоченный массив, к которому нужно применить написанную функцию merge().

Далее идёт анализ времени выполнения програмы.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Количество элементов>  ---------------  Количество потоков | **1** | **5** | **10** | **25** | **50** | **75** | **100** |
| **1** | 0.025 | 0.052 | 0.042 | 0.043 | 0.047 | 0.045 | 0.046 |
| **5** | 0.042 | 0.291 | 0.263 | 0.273 | 0.266 | 0.279 | 0.312 |
| **10** | 0.048 | 0.321 | 0.764 | 0.770 | 0.727 | 0.715 | 0.726 |
| **25** | 0.053 | 0.256 | 0.716 | 2.236 | 2.055 | 2.169 | 2.054 |
| **50** | 0.048 | 0.263 | 0.703 | 2.029 | 4.255 | 4.373 | 4.281 |
| **75** | 0.047 | 0.260 | 0.720 | 2.038 | 4.268 | 6.590 | 6.551 |
| **100** | 0.049 | 0.259 | 0.713 | 2.056 | 4.231 | 6.625 | 8.976 |

Таблица среднего (10 повторов с одинаковыми вводными ) времени выполнения программы (в мс ) в зависимости от количества элементов начального массива и максимального числа потоков.

Как можно увидеть из таблицы для массивов с 1-100 элементов наиболее выгодно использовать лишь 1 поток для вычисления. Я считаю, что связано это со слишком малым количеством элементов. Затраты времени на создание потоков не оправдываются в данном случае.

Однако, после этого я решил **сильно** увеличить количество элементов, и получил такие данные.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов>  ---------------  Количество потоков | **10000** | **100000** | **1000000** | **5000000** | **10000000** |
| **1** | 1.189 | 11.798 | 133.567 | 678.810 | 1378.201 |
| **5** | 1.384 | 11.594 | 118.514 | 636.934 | 1305.132 |
| **10** | 1.819 | 11.785 | 116.919 | 621.904 | 1278.502 |
| **25** | 3.345 | 13.126 | 116.015 | 595.224 | 1216.096 |
| **50** | 5.166 | 15.848 | 116.885 | 592.818 | 1207.902 |
| **75** | 7.718 | 18.105 | 120.844 | 601.141 | 1186.517 |
| **100** | 9.804 | 21.005 | 120.357 | 589.527 | 1103.034 |

Из данной таблицы можно увидеть, что при значениях около 100000 увеличение времени с увеличением количества процессов становиться незначительным. При большем же количестве элементов выгода использования распараллеливания становится очевидна. Отмечу, что в обоих таблицах значение для каждой ячейка получено из среднего арифметического 10 выполнений программы на данных одинакового объёма, чтобы уменьшить случайный разброс значений.