Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №X по курсу**

**«Операционные системы»**

Студент: Друхольский А.К.

Группа: М8О-207Б-21

Вариант: 17

Преподаватель: Черемисинов Максим

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

https://github.com/ssForz/OS-labs

**Постановка задачи**

**Цель работы**

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 потоков 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 входящих

данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант 17) Найти в большом целочисленном массиве минимальный элемент

**Общие сведения о программе**

Программа компилируется из файла main.c при помощи cmake. В программе реализована многопоточность. Функции для работы с потоками. которые я использовал:

* pthread\_create() — создание потока с передачей ему аргументов. В случае успеха возвращает 0.
* pthread\_join() — ожидает завершения потока обозначенного THREAD\_ID. Если этот поток к тому времени был уже завершен, то функция немедленно возвращает значение.
* pthread\_mutex\_init() — инициализация мьютекса
* pthread\_mutex\_lock() — блокировка мьютекса
* pthread\_mutex\_lock() — открытие доступа к мьютексу
* pthread\_mutex\_destroy() - удаление мьютекса

**Общий метод и алгоритм решения**

Требуется найти минимальный элемент в массиве. Суть будет в том, что массив разобьётся на количество частей, равное количеству потоков. В каждом потоке будет искаться минимальный элемент подмассива (изначально была попытка сравнивать минимум каждого потока с глобальной переменной, используя при этом мьютекс, но это сказалось отрицательно на эффективности, поэтому было принято решение выделить из каждого подмассива минимальный элемент отдельно). Количество потоков логично будет ограничить количеством элементов в массиве.

Так же в функции, которую выполняет поток добавим вывод надписи, что сейчас процедура выполняется в данном потоке (на этапе тестирования). Сделаем это для наглядности параллельности процедур.

**Исходный код**

#include<pthread.h>

#include<iostream>

#include <ctime>

#include<vector>

#include<fstream>

#include<chrono>

using namespace std;

static int minimal = 1e9;

pthread\_mutex\_t mutex;

int flag = 0;

vector<int> result(100);

struct arg\_to\_thread {

int\* big\_array;

int partition;

int num\_of\_thread;

int count\_threads;

int size\_array;

};

void\* thread\_func(void \*args)

{

int minimum = 1e9;

arg\_to\_thread\* arguments = (arg\_to\_thread\*) args;

int num\_of\_thread = arguments->num\_of\_thread;

int partition = arguments->partition;

flag = 1;

int count\_threads = arguments->count\_threads;

int size\_array = arguments->size\_array;

int\* big\_array = arguments->big\_array;

if (num\_of\_thread != count\_threads - 1) {

for (int j = num\_of\_thread \* partition; j < num\_of\_thread \* partition + partition; ++j) {

if (big\_array[j] < minimum) {

minimum = big\_array[j];

result[num\_of\_thread] = big\_array[j];

}

}

} else {

for (int j = size\_array - 1; j > size\_array - partition - 1; --j) {

if (big\_array[j] < minimum) {

minimum = big\_array[j];

result[num\_of\_thread] = big\_array[j];

}

}

}

return 0;

}

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

{

ifstream file("./src/big\_massives.txt");

int count\_threads;

cout<<"Введите количество потоков: ";

cin>>count\_threads;

cout<<endl;

int size\_array;

cout<<"Введите размер массива: ";

cin>>size\_array;

cout<<endl;

vector<int>big\_array(size\_array);

cout<<"Введите массив: "<<endl;

for(int i = 0; i < size\_array; ++i) {

string s;

getline(file, s);

big\_array[i] = atoi(s.c\_str());

}

if (count\_threads > size\_array) {

cout<<"Количество потоков больше максимального"<<endl;

count\_threads = size\_array;

}

pthread\_t threads[count\_threads];

pthread\_mutex\_init(&mutex, NULL);

int partition = size\_array / count\_threads;

struct arg\_to\_thread arg;

arg.partition = partition;

arg.big\_array = big\_array.data();

arg.count\_threads = count\_threads;

arg.size\_array = size\_array;

chrono::steady\_clock::time\_point begin = chrono::steady\_clock::now();

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

arg.num\_of\_thread = i;

if (i == count\_threads - 1) {

partition += size\_array % count\_threads;

arg.partition = partition;

}

int status = pthread\_create(&threads[i], NULL, thread\_func, (void\*)&arg);

while(flag != 1) {

//ожидание пока функции передадутся аргументы

}

flag = 0;

if (status != 0) {

cout<<"Create thread error"<<endl;

}

}

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

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

}

chrono::steady\_clock::time\_point end = chrono::steady\_clock::now();

pthread\_mutex\_destroy(&mutex);

int minimum2 = 1e9;

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

if (result[i] < 1e9) {

minimum2 = result[i];

}

}

cout<<"Минимальное значение: "<<minimum2<<endl;

cout<<chrono::duration\_cast<chrono::microseconds>(end-begin).count();

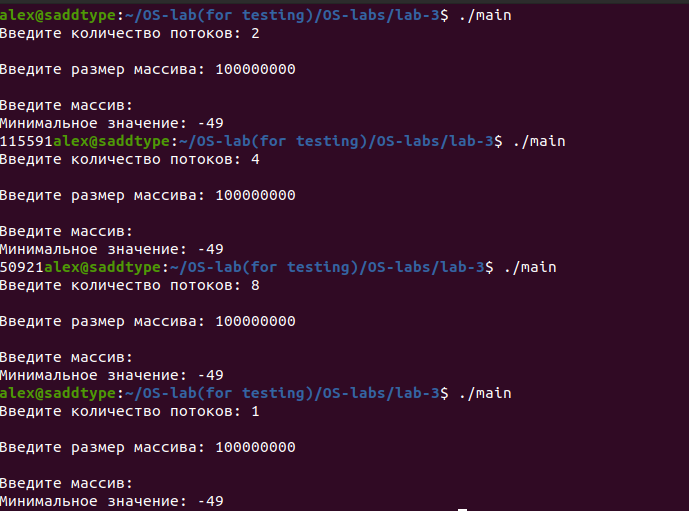
return 0;

}

//find\_minimal(big\_array, i, partition)

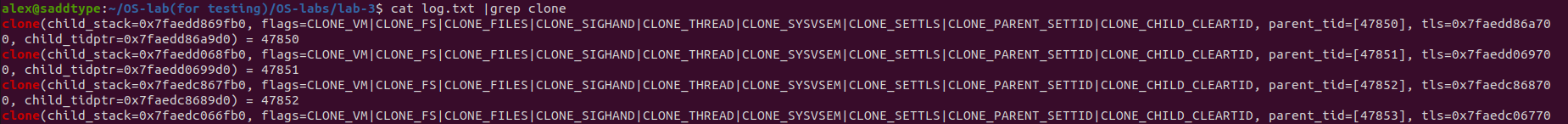
**Демонстрация работы программы**

Продемонстрируем работу многопоточного алгоритма на массиве из 10^8 чисел



Из тестирования видно, что значения при 1, 2, 4, 8, 16 потоках сильно отличаются. Особенно это разница заметна при малом количестве потоков (отличие почти в два раза при 1 и 2 потоках).

Проверим вывод утилиты strace (запишем в отдельный файл, а потом найдём вызовы создания потоков). Проверим на тесте с 4-мя потоками.



Можно сделать вывод, что создание потоков, как и создание процессов, реализовано в Linux с помощью утилиты clone(). Отличие в более тонкой настройке. С помощью этой утилиты в целом можно самостоятельно создавать потоки.

**Выводы**

Я познакомился с многопоточностью и смог реализовать её для моей задачи. Так же мне удалось выявить зависимость между количеством потоков и временем выполнения. При малом количестве потоков (<=4) разницы во времени выполнения очень значительна. Дальнейшее увеличение количества потоков приводит лишь к незначительному набору эффективности (много времени тратиться на инициализацию потоков). Так же, с помощью утилиты strace, я узнал как реализовано создание потоков в линуксе. Многопоточность – полезная вещь в различных проектах, а благодаря этой лабораторной работе я получил базовые навыки в работе с потоками.