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 В UNIX И ОБЕСПЕЧЕНИЕ ИХ СИНХРОНИЗАЦИИ

Студент: Слесарчук Василий Анатольевич
Группа: М8О–210Б–22
Вариант: 20
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023.

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- Обеспечении межпроцессорного взаимодействия посредствам технологии file mapped
- Освоение принципов работы с файловыми системами

Задание

Составить программу на языке Си, осуществляющую работу с процессами и их взаимодействие в ОС на базе UNIX.

Родительский процесс должен открыть файл из которого дочерний процесс читает все числа типа int и передает родительскому процессу их сумму.

Общие сведения о программе

Программа компилируется с помощью Makefile, сгенерированным cmake.

Также используется заголовочные файлы: stdio.h, stdlib.h, string.h, chrono, time.h, thread .

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы многопоточных программ.
2. Продумать реализацию функций, по возможности, без блокировок и простаивании процессора.
3. Написать генератор точек.
4. Написать и отладить работу основной функции по созданию процессов и их работе.
5. Придумать тесты и ответы к этим тестам.
6. Реализовать тесты.

Основные файлы программы

main.cpp

```
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>
#include <mutex>
#include <algorithm>
#include <chrono>
#include <time.h>
#include "point.h"
#include "func.h"
#include "points.h"



int main(int argc, char* argv[]) {
    size_t maxThreads = 1; // Значение по умолчанию

    if (argc > 1) {
        maxThreads = std::stoul(argv[1]);
    }



    std::cout << maxThreads;
    if (maxThreads == 0){
        std::cout << "---> 100 Points / Threads from 1 to 10 <---\n";
        for (int i = 1; i < 11; i++)
        {
            auto start = std::chrono::steady_clock::now();
            std::vector<Point> result = findLargestTriangleMultiThread(P100, i);
            auto end = std::chrono::steady_clock::now();
            std::cout << "Triangle find in: " <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "
ms" << " -- threads: " << i << "\n";
            std::cout << "Треугольник наибольшей площади:" << std::endl;
            for (const auto& point : result) {
                std::cout << "(" << point.x << ", " << point.y << ", " <<
point.z << ")";
            }
            std::cout << "\n\n";
        }

        std::cout << "---> 200 Points / Threads from 1 to 10 <---\n";
        for (int i = 1; i < 11; i++)
        {
            auto start = std::chrono::steady_clock::now();
            std::vector<Point> result = findLargestTriangleMultiThread(P200, i);
            auto end = std::chrono::steady_clock::now();
            std::cout << "Triangle find in: " <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "
ms" << " -- threads: " << i << "\n";
```

```
            std::cout << "Треугольник наибольшей площади:" << std::endl;
            for (const auto& point : result) {
                std::cout << "(" << point.x << ", " << point.y << ", " <<
point.z << ") ";
            }
            std::cout << "\n\n";
        }

        std::cout << "---> 300 Points / Threads from 1 to 10 <---\n";
        for (int i = 1; i < 11; i++)
        {
            auto start = std::chrono::steady_clock::now();
            std::vector<Point> result = findLargestTriangleMultiThread(P300, i);
            auto end = std::chrono::steady_clock::now();
            std::cout << "Triangle find in: " <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "
ms" << " -- threads: " << i << "\n";
            std::cout << "Треугольник наибольшей площади:" << std::endl;
            for (const auto& point : result) {
                std::cout << "(" << point.x << ", " << point.y << ", " <<
point.z << ") ";
            }
            std::cout << "\n\n";
        }

        std::cout << "---> 400 Points / Threads from 1 to 10 <---\n";
        for (int i = 1; i < 11; i++)
        {
            auto start = std::chrono::steady_clock::now();
            std::vector<Point> result = findLargestTriangleMultiThread(P400, i);
            auto end = std::chrono::steady_clock::now();
            std::cout << "Triangle find in: " <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "
ms" << " -- threads: " << i << "\n\n";
            std::cout << "Треугольник наибольшей площади:" << std::endl;
            for (const auto& point : result) {
                std::cout << "(" << point.x << ", " << point.y << ", " <<
point.z << ") ";
            }
            std::cout << "\n\n";
        }
    } else {
        auto start = std::chrono::steady_clock::now();
            std::vector<Point> result = findLargestTriangleMultiThread(P400,
maxThreads);
            auto end = std::chrono::steady_clock::now();
            std::cout << "Triangle find in: " <<
std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() << "
ms" << " -- threads: " << maxThreads << "\n\n";
            std::cout << "Треугольник наибольшей площади:" << std::endl;
            for (const auto& point : result) {
                std::cout << "(" << point.x << ", " << point.y << ", " <<
point.z << ") ";
            }
            std::cout << "\n\n";
    }
```

```
    return 0;
}
```

func.h

```
#pragma once
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>
#include <mutex>
#include <algorithm>
#include <chrono>
#include <time.h>
#include "point.h"
#include "points.h"

std::mutex maxAreaMutex;


Point getRPoint(int Min, int Max) {
    Point result;

    // Инициализация генератора случайных чисел

    // Генерация случайных чисел в заданных границах
    result.x = std::rand() % (Max - Min + 1) + Min;
    result.y = std::rand() % (Max - Min + 1) + Min;
    result.z = std::rand() % (Max - Min + 1) + Min;

    return result;
}

void getRPoints(int n){
    std::srand(std::time(NULL));
    std::cout << "{";
    for (int i = 0; i < n; i++)
    {
        Point p = getRPoint(-100, 100);
        std::cout << "{" << p.x << ", " << p.y << ", " << p.z << "}, \n";
    }
    std::cout << "}";

}

double distance(const Point& p1, const Point& p2) {
    return std::sqrt(std::pow(p2.x - p1.x, 2) + std::pow(p2.y - p1.y, 2) +
std::pow(p2.z - p1.z, 2));
}

double calcArea(const Point& p1, const Point& p2, const Point& p3) {
```

```
    double a = distance(p1, p2);
    double b = distance(p2, p3);
    double c = distance(p3, p1);
    double s = (a + b + c) / 2;
    return std::sqrt(s * (s - a) * (s - b) * (s - c));
}

void findLargestTriangleThread(const std::vector<Point>& points, size_t start,
size_t end, double& maxArea, std::vector<Point>& maxTriangle) {
    for (size_t i = start; i < end; ++i) {
        for (size_t j = 0; j < points.size(); ++j) {
            for (size_t k = 0; k < points.size(); ++k) {
                double currentArea = calcArea(points[i], points[j], points[k]);
                if (currentArea > maxArea) {
                    std::lock_guard<std::mutex> lock(maxAreaMutex);
                    if (currentArea > maxArea) {
                        maxArea = currentArea;
                        maxTriangle = {points[i], points[j], points[k]};
                    }
                }
            }
        }
    }
}

std::vector<Point> findLargestTriangleMultiThread(const std::vector<Point>&
points, size_t maxThreads) {
    double maxArea = 0;
    std::vector<Point> maxTriangle;

    size_t chunkSize = points.size() / maxThreads;
    std::vector<std::thread> threads;

    for (size_t i = 0; i < maxThreads; ++i) {
        size_t start = i * chunkSize;
        size_t end = (i == maxThreads - 1) ? points.size() : (i + 1) *
chunkSize;
        threads.emplace_back(findLargestTriangleThread, std::ref(points), start,
end, std::ref(maxArea), std::ref(maxTriangle));
    }

    for (auto& thread : threads) {
        thread.join();
    }

    return maxTriangle;
}

int min(int a, int b){
    if (a < b) return a;
    return b;
}
```

Пример работы

./findTriangle.exe 1

./findTriangle.exe 4

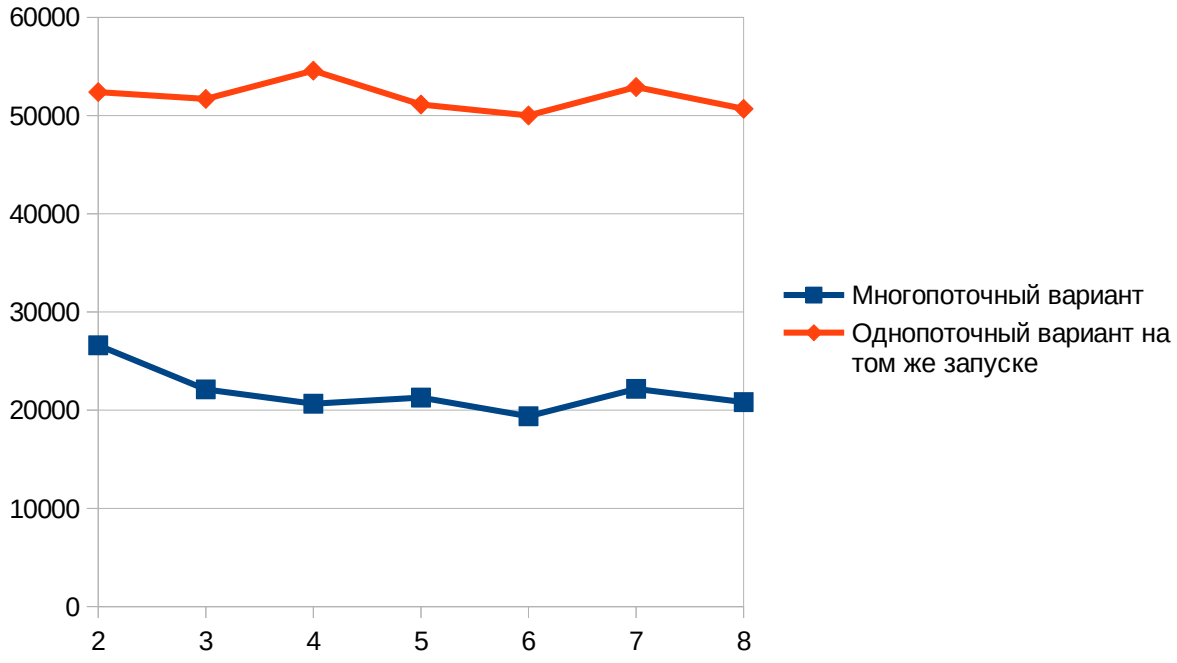
Затраченное время для 4 потоков: 20062 nanoseconds

Затраченное время для одного потока: 51689 nanoseconds

Эффективность многопоточной программы

Тесты осуществлялись для 400 точек, результаты представлены в наносекундах.

Количество потоков	2	3	4	5	6	7	8
Многопоточный вариант	26603	22118	20658	21277	19387	22174	20814
Однопоточный вариант на том же запуске	52392	51689	54572	51120	50012	52902	50685



Вывод

Во время выполнения это лабораторной работы я успел узнать, что при создании вектора потоков и добавлении туда потоков, реализуется многопоточность, даже если явно не указан поток. Как выяснилось, компилятор может самостоятельно распараллеливать некоторые вычисления, а также делать их быстрее. К этому можно отнести, что деление на const int почему то дольше, чем на обычный int. Также я узнал, что clock() будет считать общее процессорное время. Возможность распараллеливать вычисления позволяет выигрывать по времени до 3 раз (в моих тестах) и наиболее ощутимые сокращения времени получаются на больших массивах.