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: Забелкин Андрей Алексеевич
Группа: М8О–210Б–22
Вариант: 8
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 из с помощью Makefile, сгенерированным cmake.

Также используется заголовочные файлы: stdio.h, stdlib.h, string.h, pthread.h, time.h .Для создания многопоточности используется:

1. **pthread_create()** - эта функция запускает новый поток.
2. **pthread_join()** - эта функция ожидает завершения процесса, указанного в аргументах.

## 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 матриц.
4. Написать и отладить работу основной функции по созданию процессов и их работе.
5. Придумать тесты и ответы к этим тестам.
6. Написать bash-скрипт, который запускает и проверяет программу на тестах.

## Основные файлы программы

## lab2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <time.h>

struct argum {
    int** arr;
    long long* res;
    int t;
    int N_ARRAYS;
    int NUM_FOR_ARRAYS;
    int NUM_THREADS;
};

void* thread_function(void* arg) {
    struct argum* id = ((struct argum*)arg);
    int incr = id->NUM_THREADS;
    int len = id->NUM_FOR_ARRAYS;
    for (int i = id->t; i < len; i += incr) {
        for (int j = 0; j < id->N_ARRAYS; ++j) {
            id->res[i] += id->arr[j][i];
        }
    }
    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    int N_ARRAYS;
    int NUM_FOR_ARRAYS;
    int NUM_THREADS;
    int** arrays = NULL;
    long long* array_of_sums = NULL;
    if (argc != 2) {
        printf("%s <number_of_threads>\n", argv[0]);
        return 1;
    }

    NUM_THREADS = atoi(argv[1]);
    scanf("%d", &N_ARRAYS);
    scanf("%d", &NUM_FOR_ARRAYS);
    if (NUM_THREADS <= 0) {
        printf("Ну ты придумал конечно.\n");
        return 1;
    }
    arrays = (int**)realloc(arrays, N_ARRAYS * sizeof(int*));
    for (int i = 0; i < N_ARRAYS; ++i) {
        arrays[i] = (int*)realloc(arrays[i], NUM_FOR_ARRAYS * sizeof(int));
    }
    array_of_sums = (long long*)realloc(array_of_sums, NUM_FOR_ARRAYS * sizeof(long
long));
    memset(array_of_sums, 0, NUM_FOR_ARRAYS * sizeof(long long));
    for (int i = 0; i < N_ARRAYS; ++i) {
        for (int j = 0; j < NUM_FOR_ARRAYS; ++j) {
```

```
            scanf("%d", &arrays[i][j]);
        }
    }
    if (arrays == NULL || array_of_sums == NULL) {
        printf("Ошибка выделения памяти.\n");
        return 1;
    }
    struct timespec start, end;

    struct argum* args_array = (struct argum*)malloc(NUM_THREADS * sizeof(struct
argum));

    if (args_array == NULL) {
        perror("Memory allocation failed");
        return 1;
    }

    for (int t = 0; t < NUM_THREADS; t++) {
        args_array[t].arr = arrays;
        args_array[t].res = array_of_sums;
        args_array[t].N_ARRAYS = N_ARRAYS;
        args_array[t].NUM_THREADS = NUM_THREADS;
        args_array[t].NUM_FOR_ARRAYS = NUM_FOR_ARRAYS;
        args_array[t].t = t;
    }

    clock_gettime(CLOCK_MONOTONIC, &start);
    pthread_t threads[NUM_THREADS];
    for (int t = 0; t < NUM_THREADS; ++t) {
        if(pthread_create(&threads[t], NULL, thread_function, (void*)&args_array[t])) {
            perror("pthread_create");
            return 1;
        }
    }

    for (int t = 0; t < NUM_THREADS; ++t) {
        pthread_join(threads[t], NULL);
    }

    clock_gettime(CLOCK_MONOTONIC, &end);
    long long elapsed_time = (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec -
start.tv_nsec);
    printf("Затраченное время: %lld nanoseconds\n", elapsed_time);
    clock_gettime(CLOCK_MONOTONIC, &start);
    pthread_t thread;
    args_array[0].NUM_THREADS = 1;
    pthread_create(&thread, NULL, thread_function, (void*)&args_array[0]);

    pthread_join(thread, NULL);

    clock_gettime(CLOCK_MONOTONIC, &end);
    elapsed_time = (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec -
start.tv_nsec);
    printf("Затраченное время: %lld nanoseconds\n", elapsed_time);

    pthread_exit(NULL);
```

```
}
```

## Пример работы

**./src/build/lab2 4 < ./test/matrix1.txt**

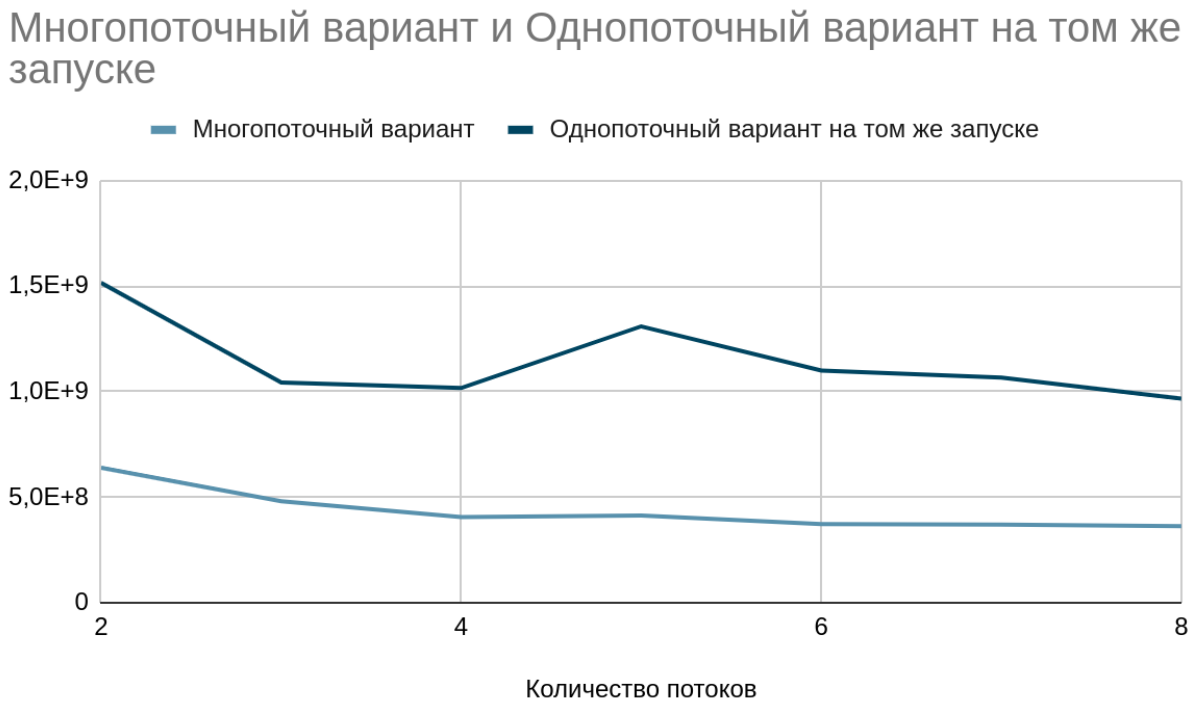
**Затраченное время для 4 потоков: 343202950 nanoseconds**

**Затраченное время для одного потока: 931669784 nanoseconds**

## Эффективность многопоточной программы

**Тесты осуществлялись для матрицы размером 194 648 015 байт, результаты представлены в наносекундах.**

| Количество потоков | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| Многопоточный вариант | 639312603 | 480716663 | 405134158 | 412915523 | 372762279 | 369623273 | 362637896 |
| Однопоточный вариант на том же запуске | 1515581506 | 1042892441 | 1016758092 | 1308920375 | 1099969358 | 1066441450 | 966547155 |



# Вывод

Во время выполнения это лабораторной работы я успел реализовать неправильное сложение массивов (складывал все элементы), узнать, что при старте процесса лучше не менять никакие переменные, переделать реализацию на глобальных переменных на способ, при котором в параметры исполняемой функции я передаю указатель на структуру, которая в свою очередь содержит общие для потоков ресурсы. Интересным открытием стало то, что сложение массива на одной ядре, или на 8 (максимальном для моей машины значении) я все равно считаю медленнее, чем если бы я не реализовывал многопоточность. 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.