Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

Студент: Юнусов Руслан Асифович
Группа: М8О-209Б-24
Вариант: 10
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

## Содержание

## Репозиторий

https://github.com/Rissochek/OSLabs/tree/main/lab2

## 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 обработке использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение потоков должно быть задано ключом запуска программы.
Необходимо уметь продемонстрировать количество потоков, используемых программой, с помощью стандартных средств операционной системы.
Привести исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков. Объяснить получившиеся результаты.
Вариант 10: Решить систему линейных уравнений методом Гаусса.

## Общие сведения о программе

Программа написана на языке Си в UNIX-подобной ОС. При запуске программы указывается аргумент, который отвечает за количество потоков. Программа реализует метод Гаусса для матриц с возможностью многопоточного режима.

## Общий метод и алгоритм решения

В работе реализован метод Гаусса. Он представляет собой алгоритм действий для приведения матриц любого вида к ступенчатому виду, посредством выбора ведущего элемента и последующими действиями, такими как сложение и вычитание на строку ведущего элемента помноженную на некоторое число n, при котором элемент ниже ведущего будет обнулен. Эти действия проводятся со строками лежащими ниже строки с ведущим элементом. В программе реализован именно такой вариант решения матриц при помощи метода Гаусса.

## Исходный код

main.c

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


typedef struct {
   size_t row; size_t row_start; size_t row_end; size_t columns_count;
double** massive; double gen_value;
} GaussArgs;


void* func_for_threads(void* args){
   GaussArgs *gauss_args = (GaussArgs *)args;
   size_t row = gauss_args->row;
   size_t row_start = gauss_args->row_start;
   size_t row_end = gauss_args->row_end;
   size_t columns_count = gauss_args->columns_count;
   double** massive = gauss_args->massive;
```

```
    double gen_value = gauss_args->gen_value;
    double to_replace = 0;
    for (size_t sub_row = row_start; sub_row < row_end; sub_row++){
        double value_under_gen = massive[sub_row][row];
        double coff = value_under_gen/gen_value;
        for (size_t col = row; col < columns_count; col++){
            to_replace = massive[sub_row][col] - massive[row][col]*coff;
            massive[sub_row][col] = to_replace;
        }
    }
    pthread_exit(0);
}


//massive[строки][колонны]  massive[row][column]
void Gauss_func(size_t columns_count, size_t rows_count, double** massive,
size_t thread_count) {
    pthread_t tid[thread_count];
    size_t rows_per_thread = 1;
    size_t remaining_rows = 0;
    GaussArgs* args_array = malloc(thread_count * sizeof(GaussArgs));
    size_t pthread_used = 0;
    for (size_t row = 0; row < rows_count; row++) {
        double gen_value = massive[row][row];
        if (gen_value != 0.0) {
            if (row < rows_count - 1) {
                for (size_t i = 0; i < thread_count; i++) {
                    if (thread_count < (rows_count - 1) - row ){
                        rows_per_thread = ((rows_count- 1) - row) /
thread_count;
                        remaining_rows = (rows_count - 1 - row) %
thread_count;
```

```
                    }
                    size_t row_start = (i * rows_per_thread) + 1 + row;
                    size_t row_end = row_start + rows_per_thread;
                    if (i == thread_count - 1) {
                        row_end += remaining_rows;
                    }
                    if (row_end <= rows_count){
                        args_array[i] = (GaussArgs){row, row_start,
row_end, columns_count, massive, gen_value};
                        pthread_create(&tid[i], NULL, func_for_threads,
&args_array[i]);
                        pthread_used++;
                    }else{
                        break;
                    }
                }
                for (size_t i = 0; i < pthread_used; i++) {
                        pthread_join(tid[i], NULL);
                    }
                rows_per_thread = 1;
                remaining_rows = 0;
                pthread_used = 0;
                }
            }
        }
    free(args_array);
}

void print_matrix(double** massive, size_t rows_count, size_t col_count){
    for (size_t row = 0; row < rows_count; row++){
        for (size_t col = 0; col < col_count; col++){
```

```
            printf("%f ", massive[row][col]);
        }
            printf("\n");
    }
}

void input_matrix(double** massive, size_t rows_count, size_t col_count){
    for (size_t row = 0; row < rows_count; row++){
        for (size_t col = 0; col < col_count; col++){
            scanf("%lf", &massive[row][col]);
        }
    }
}

void generate_random_matrix(double** massive, size_t rows_count, size_t
col_count) {
    srand(time(NULL));
    for (size_t row = 0; row < rows_count; row++) {
        for (size_t col = 0; col < col_count; col++) {
            massive[row][col] = (double)(rand() % 10);
        }
    }
}

int main(int argc, char *argv[]){
    size_t rows_count = 3;
    size_t col_count = 4;
    double**massive = malloc(sizeof(double *) * rows_count);
    for (size_t line = 0; line < rows_count; line++){
        massive[line] = malloc(col_count * sizeof(double));
```

```
    }


    //generate_random_matrix(massive, rows_count, col_count);
    input_matrix(massive, rows_count, col_count);
    print_matrix(massive, rows_count, col_count);
    const size_t thread_count = (size_t)atoi(argv[1]);
    clock_t start_time = clock();
    Gauss_func(col_count, rows_count, massive, thread_count);
    clock_t end_time = clock();
    double elapsed_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
    printf("Время выполнения Gauss_func: %f секунд\n", elapsed_time);
    print_matrix(massive, rows_count, col_count);
    for (size_t i = 0; i < rows_count; i++){
        free(massive[i]);
    }
    free(massive);
}
```

## Демонстрация работы программы

**parol1@riss:~/Labs/OSLabs/lab2$ gcc main.c**
**parol1@riss:~/Labs/OSLabs/lab2$ ./a.out 2**
**#input matrix**
**1 2 3 4**
**5 6 7 8**
**9 10 11 12**
**#matrix after input**
**1.000000 2.000000 3.000000 4.000000**
**5.000000 6.000000 7.000000 8.000000**
**9.000000 10.000000 11.000000 12.000000**
**Время выполнения Gauss_func: 0.000143 секунд**
**#matrix after Gauss_func**

**1.000000 2.000000 3.000000 4.000000**
**0.000000 -4.000000 -8.000000 -12.000000**
**0.000000 0.000000 0.000000 0.000000**

## Выводы

Язык Си позволяет пользователю взаимодействовать с потоками операционной системы. Для этого на Unix-подобных системах требуется подключить библиотеку pthread.h.

Создание потоков происходит быстрее, чем создание процессов, а все потоки используют одну и ту же область данных. Поэтому многопоточность – один из способов ускорить обработку каких-либо данных: выполнение однотипных, не зависящих друг от друга задач, можно поручить отдельным потокам, которые будут работать параллельно. Однако нельзя забывать о таком понятии как **race condition**, которое может привести к некорректной работе программы. Для избежания этого можно пользоваться мьютексами, но в моем варианте в этом нет необходимости, так как у меня каждый поток работает со своей строкой.

Средствами языка Си можно совершать системные запросы на создание потока, ожидания завершения потока, а также использовать различные примитивы синхронизации.