Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3 по курсу**

**«Операционные системы»**

# Процессы и потоки

Студент: Шевлякова София Сергеевна
Группа: М8О–208Б–21
Вариант: 8
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

## 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 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 8. Есть К массивов одинаковой длины. Необходимо сложить эти массивы. Необходимо предусмотреть стратегию, адаптирующуюся под количество массивов и их длину (по количеству операций).

## Общие сведения о программе

Программа компилируется из файла main.c. Также используется заголовочные файлы: stdio.h, stdlib.h, pthread.h. В программе используются следующие системные вызовы:

1. **pthread_create (&mutex, NULL, start, &arg) –** создает новый поток mutex, NULL - атрибуты потока по умолчанию, start - функция, которая будет выполняться в новом потоке, arg - аргументы, которые передаются этой функции. Возвращает 0 в случае успеха. В mutex сохраняется id потока;

2. **pthread_join(tid, value_ptr) -** откладывает выполнение вызывающего потока, до тех пор, пока не будет выполнен поток tid. Когда pthread_join выполнилась успешно, то она возвращает 0. Если поток явно вернул значение, то оно будет помещено в переменную value_ptr.

## Общий метод и алгоритм решения

Сначала в переменную CountThreads считываем количество потоков, введенных пользователем. Далее пользователь вводит количество и длину массивов, если длина будет более, чем в два раза, больше количества массивов, то используется функция вертикального суммирования, иначе – горизонтального. Все введённые пользователем элементы массивов преобразуются в двумерный массив (матрицу). В случае вертикального суммирования, двумерный массив разделяется между потоками по столбцам, и каждый поток складывает элементы в одном или нескольких столбцах матрицы. Когда применяется горизонтальное суммирование, то матрица делится по строчкам, то есть каждый поток суммирует по столбцам один или несколько введённых пользователем массивов.

Стоит обратить внимание, что в этом случае мы применяем mutex, так как при обращении к переменной token.res[i] одним потоком, его может прервать другой поток, возникает так называемая «гонка потоков». Поток, который захватил (заблокировал) mutex, работает с участком кода. Остальные потоки, когда достигают mutex, ждут его разблокировки. Для этого используем функции pthread_mutex_lock, которая блокирует mutex и возвращает 0 после успешного завершения. После этого участок кода token.res[i] += c становится недоступным остальным потомкам - их выполнение блокируется до тех пор, пока mutex не будет разблокирован. Освобождение должен провести поток, заблокировавший mutex, вызовом pthread_mutex_unclock.

## Основные файлы программы

========== main.c ==========

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

pthread_mutex_t mutex;

typedef struct ThreadToken {
    int** arr;
    int* res;
    int start;
    int steps;
    int K;
    int N;
} ThreadToken;

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

void exit_with_msg(const char* message, int return_code) {
    printf("%s\n", message);
    exit(return_code);
}

void* vertical_sum_arrays(void* arg) {
    ThreadToken token = *((ThreadToken*) arg);
    for (int i = token.start; i < token.start + token.steps; ++i) {
        int c = 0;
        for (int j = 0; j < token.K; ++j) {
            c += token.arr[j][i];
        }
        token.res[i] = c;
    }
    return arg;
}

void* horizontal_sum_arrays(void* arg) {
    ThreadToken token = *((ThreadToken*) arg);
    for (int i = 0; i < token.N; ++i) {
        int c = 0;
        for (int j = token.start; j < token.start + token.steps; ++j) {
            c += token.arr[j][i];
```

```
        }
        pthread_mutex_lock(&mutex);
        token.res[i] += c;
        pthread_mutex_unlock(&mutex);
    }
    return arg;
}

int main(int argc, const char** argv) {
    if (argc < 2) {
        exit_with_msg("Missing arguments", 1);
    }
    int CountThreads = 0;
    for (int i = 0; argv[1][i] > 0; ++i) {
        if (argv[1][i] >= '0' && argv[1][i] <= '9') {
            CountThreads = CountThreads * 10 + argv[1][i] - '0';
        }
    }
    int N, K;
    printf("Length of arrays N: ");
    scanf("%d", &N);
    printf("Number of arrays K: ");
    scanf("%d", &K);
    int** all = malloc(sizeof(int*) * K);
    if (all == NULL) {
        exit_with_msg("Cannot allocate memory", 2);
    }
    for (int i = 0; i < K; ++i) {
        all[i] = malloc(sizeof(int) * N);
        if (all[i] == NULL) {
            for (int j = 0; j < i; ++j) {
                free(all[j]);
                all[j] = NULL;
            }
            free(all);
            all = NULL;
            exit_with_msg("Cannot allocate memory", 2);
        }
        for (int j = 0; j < N; ++j) {
            scanf("%d", &all[i][j]);
        }
    }

    void* function;
    int end;
    if (N > K * 2) {
```

```
    function = &vertical_sum_arrays;
    end = N;
    printf("vertical\n");
} else {
    function = &horizontal_sum_arrays;
    end = K;
    printf("horizontal\n");
    if (pthread_mutex_init(&mutex, NULL) != 0) {
        exit_with_msg("Cannot init mutex", 3);
    }
}
CountThreads = min(CountThreads, end);
pthread_t* th = malloc(sizeof (pthread_t) * CountThreads);       //id потока
ThreadToken* token = malloc(sizeof(ThreadToken) * CountThreads);
int* result = malloc(sizeof(int) * N);
if (th == NULL || token == NULL || result == NULL) {
    exit_with_msg("Cannot allocate memory", 2);
}
for (int i = 0; i < N; ++i) {
    result[i] = 0;
}
int start = 0;
int steps = (end + CountThreads - 1) / CountThreads;
for (int i = 0; i < CountThreads; ++i) {
    token[i].arr = all;
    token[i].res = result;
    token[i].start = start;
    token[i].K = K;
    token[i].N = N;
    token[i].steps = min(end - start, steps);
    start += steps;
}

for (int i = 0; i < CountThreads; ++i) {
    if (pthread_create(&th[i], NULL, function, &token[i]) != 0) {
        exit_with_msg("Cannot create thread", 4);
    }
}
for (int i = 0; i < CountThreads; ++i) {
    if (pthread_join(th[i], NULL) != 0) {
        exit_with_msg("Cannot join threads", 5);
    }
}
for (int i = 0; i < N; ++i) {
    printf("%d ", result[i]);
}
```

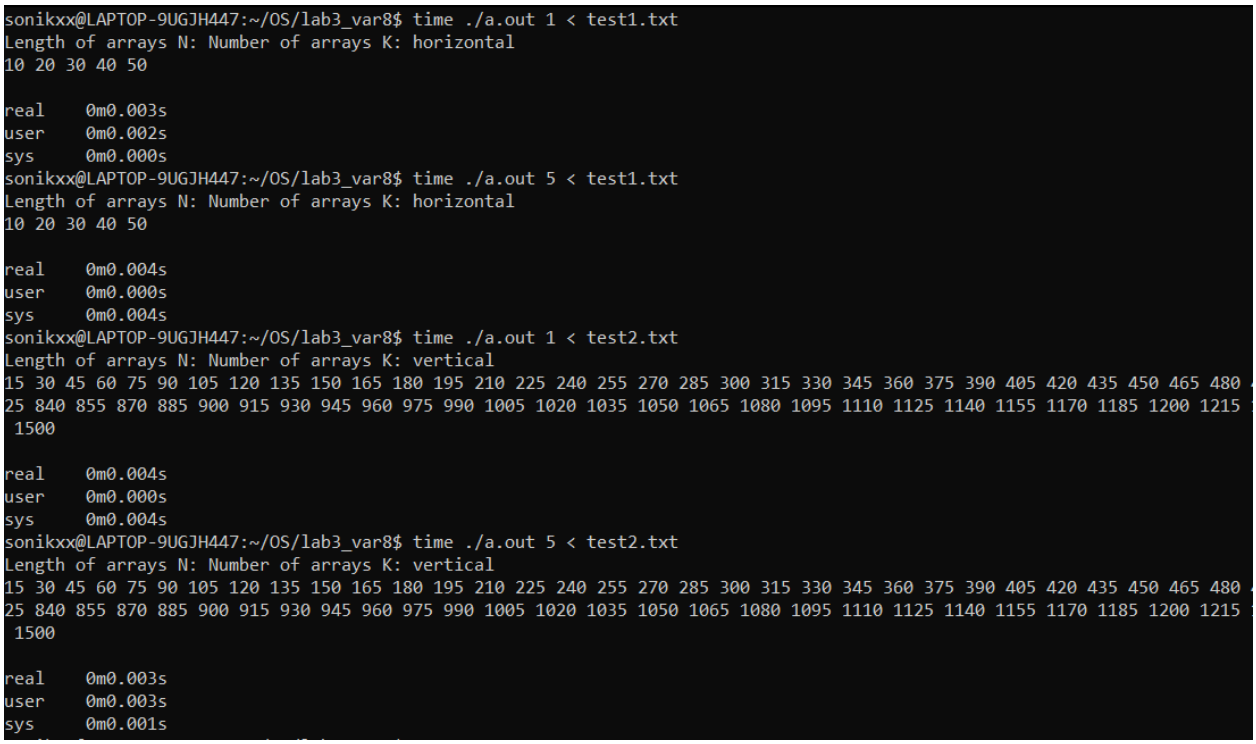
```
    printf("\n");
    for (int i = 0; i < K; ++i) {
        free(all[i]);
        all[i] = NULL;
    }
    if (end == K) {
        pthread_mutex_destroy(&mutex);
    }
    free(all);
    free(token);
    free(th);
    free(result);
    return 0;
}
```

**test1.txt** – файл содержит 10 массивов длины - 5.

**test2.txt** – файл содержит 15 массивов длины - 100.

**test3.txt** – файл содержит 400 массивов длины - 10000.

## Пример работы




На небольших данных можно посмотреть, как программа выбирает способ сложения массивов, но судить о времени выполнения не стоит, так как настолько малые временные промежутки могут быть погрешностью.

Посмотрим, как меняется время выполнения в зависимости от количества потоков на test3.txt. User time указывает время, затраченное программой в пользовательском режиме:

| количество потоков | user time |
|---|---|
| 1 | 0m0.393s |
| 2 | 0m0.392s |
| 3 | 0m0.415s |
| 4 | 0m0.362s |
| 5 | 0m0.457s |
| 6 | 0m0.473s |
| 7 | 0m0.516s |
| 8 | 0m0.285s |
| 9 | 0m0.415s |
| 10 | 0m0.460s |

Мы видим, что на 8 потоках программа работает наиболее эффективно по времени, так как на моем ноутбуке 8 ядер, а значит только 8 потоков могут выполняться одновременно, поэтому дальнейшее увеличение потоков не ускорит программу. Стоит помнить, что на создание потока тоже уходит время, поэтому программа может выполняться на одном потоке быстрее, чем на 10. Можно сказать, что метод параллельных вычислений действительно даёт выигрыш в скорости, хотя не всегда покрывает расходы на создание потока.

## Вывод

Проделав лабораторную работу, я приобрела практические навыки в управлении потоками в ОС и обеспечила синхронизацию между ними. Создание потоков не так просто, как кажется на первый взгляд, ведь

необходимо воплотить специальную функцию, которая будет выполняться в отдельном потоке исполнения, к реализации данной функции мы должны подходить с особой осторожностью из-за доступа к одной и той же памяти родительского процесса многими потоками.

Также я узнала, что использование потоков может пригодиться в любой системе: в однопроцессорной, в которой достаточная часть времени уходит на ожидание ввода-вывода, и в многопроцессорной, где задачи могут выполняться параллельно на разных процессорах, что даёт рост производительности программы. К сожалению, не любой алгоритм хорошо выполняется как многопоточная программа, однако для некоторых из них есть параллельные реализации, которые ускоряют работу программы.