Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

Потоки в операционных системах

Студент	Гришин Алексей Юрьевич
Группа	М8О-208Б-21
Вариант	6
Преподаватель	Соколов Андрей Алексеевич
Оценка	5
Дата	03.10.2022
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 C, осуществляющую перемножение 2-ух матриц, содержащих комплексные числа, в многопоточном режиме. При обработке 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.

Общие сведения о программе

Описание

Программа состоит из одного главного файла `main.c` и вспомогательных компонентов: `matrix`, `io_matrix`, `manager`, `worker`.

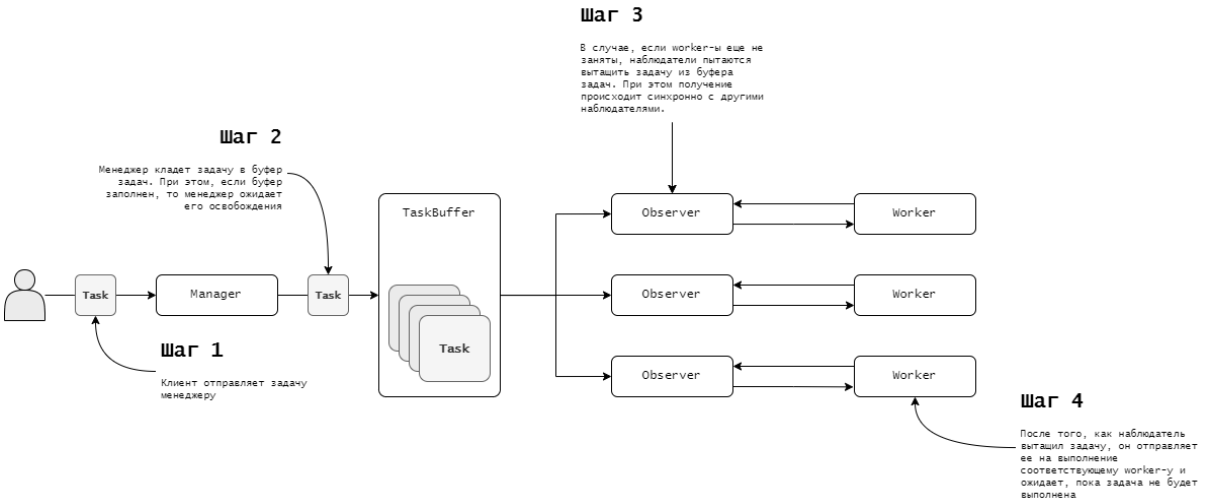
Модуль `matrix` вводит новый тип данных матрицы и добавляет набор методов, для работы с этим типом данных. Список методов приведен ниже:

1. `create_matrix` – создает объект матрицы
2. `delete_matrix` – удаляет объект матрицы, освобождая память, которую он до этого занимал
3. `rows`, `cols` – возвращает количество строк и столбцов матрицы соответственно

4. `get_cell`, `set_cell` – возвращает или устанавливает значение клетки матрицы по ее координатам

Модуль `io_matrix` внедряет функционал, позволяющий считать матрицы с входного потока и вывести ее в отформатированном виде. Из описания данного модуля очевидно, что он содержит 2 метода: `print_matrix`, `scan_matrix`.

Основными модулями являются `manager` и `worker`. Хоть `manager` и `worker` и являются представлены в виде отдельных модулей, они являются единым компонентом. При реализации модулей я придерживался следующей архитектуры



Модуль `worker` представляет абстракцию потока, исполняющего задачу. Все, что необходимо для работы с такими объектами – наличие способа отправки задания и способа узнать о завершении выполнения задания. Следовательно, ключевую роль среди набора методов для работы с worker являются `run_task` и `wait_worker`.

Далее, согласно диаграмме выше, каждому `worker`-у соответствует наблюдатель. Задача наблюдателя заключается в обеспечении `worker`-а задачами. Делает он это путем получения задач из буфера задач. Буфер задач обеспечивает синхронные операции вставки и получения сообщений. По своей сути, данная задача очень похожа на задачу Consumer - Producer, поэтому основные идеи решения я взял оттуда. Также, стоит отметить, что при вставке задач в буфер необходимо учитывать момент, когда он заполнен. В этом случае мы должны заблокировать Producer-а до тех пор, пока в буфере

не появится свободное место. Аналогичная ситуация и с получением задачи из буфера. Нам необходимо учитывать ситуацию, когда буфер пуст. В этом случае мы должны заблокировать Consumer-а до тех пор, пока в буфере не появится сообщение. Использование семафоров значительно облегчает реализацию этой задачи, так как их механика с использованием счетчика очень подходит задачи. В итоге, для буфера задач я использовал 2 семафора, значения которых обозначают количество свободных ячеек в буфере и количество свободных задач. Для обеспечения синхронизации получения и вставки сообщений я использовал блокировку `mutex`, так как она проста в использовании (все, что нам необходимо для работы с `mutex` – методы `lock`, `unlock`).

Задача менеджера заключается в структуризации и объединении отдельных частей всей системы в единое целое (своего рода реализация паттерна проектирования ”Фасад”). Внедрение менеджера позволяет сократить взаимодействие с системой до 2-х методов: `do_task` и `wait_for_task_complete`. Метод `do_task` отправляет задачу на выполнение, а метод `wait_for_task_complete` блокирует поток исполнения до тех пор, пока все задачи не будут выполнены. К тому же, при инициализации менеджера мы можем настроить размер буфера и количество `worker`-ов, что окажет большое удобство при дальнейшем анализе производительности программы.

Исходный код

`io_matrix/io_matrix.h`

```
#ifndef __IO_MATRIX_H__
#define __IO_MATRIX_H__

#include "../matrix/matrix.h"
#include <stdio.h>

typedef enum {
    SCAN_MATRIX_SUCCESS = 0,
    INVALID_INPUT_FORMAT = 1
} ScanMatrixExitCode;

void print_matrix(FILE *stream, PMatrix matrix);
ScanMatrixExitCode scan_matrix(FILE *stream, PMatrix *matrix);

#endif
```

io_matrix/io_matrix.c

```
#include "io_matrix.h"

void print_matrix(FILE * stream, PMatrix matrix) {
    for (int i = 0; i < rows(matrix); i++) {
        for (int j = 0; j < cols(matrix); j++) {
            MatrixElement el;
            get_cell(i, j, matrix, & el);
            fprintf(stream, "%.2f + %.2fi\t", creal(el), cimag(el));
        }
        fprintf(stream, "\n");
    }
}

ScanMatrixExitCode scan_matrix(FILE * stream, PMatrix * matrix) {
    int rows, cols;
    float real, imag;

    if (fscanf(stream, "%d %d", & rows, & cols) != 2)
        return INVALID_INPUT_FORMAT;

    create_matrix(rows, cols, matrix);

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (fscanf(stream, "%f + %fi", & real, & imag) != 2)
                return INVALID_INPUT_FORMAT;

            set_cell(i, j, CMPLX(real, imag), * matrix);
        }
    }

    return SCAN_MATRIX_SUCCESS;
}
```

manager/observer/observer.c

```
#include "observer.h"
#include <pthread.h>
#include <stdlib.h>

struct __Observer {
    TaskBuffer observed_buffer;
    Worker worker;
```

```
    pthread_t observer_thread_id;

    pthread_mutex_t end_search_lock;
};

void __observer_thread_code(Observer observer) {
    while (1) {
        Task task = get_task(observer -> observed_buffer);

        pthread_mutex_lock( & (observer -> end_search_lock));
        run_task(task, observer -> worker);
        pthread_mutex_unlock( & (observer -> end_search_lock));

        wait_worker(observer -> worker);
    }
}

void * __observer_thread_code_wrapper(void * data) {
    Observer observer = (Observer) data;
    __observer_thread_code(observer);
    pthread_exit(0);
}

void create_observer(TaskBuffer buffer, Worker worker, Observer *
observer) {
    * observer = (Observer) malloc(sizeof(struct __Observer));

    ( * observer) -> observed_buffer = buffer;
    ( * observer) -> worker = worker;

    pthread_mutex_init( & ( * observer) -> end_search_lock, NULL);
}

void start_search(Observer observer) {
    pthread_create( &
        observer -> observer_thread_id,
        NULL,
        __observer_thread_code_wrapper,
        observer);
}

void end_search(Observer observer) {
    pthread_mutex_lock( & (observer -> end_search_lock));
    pthread_cancel(observer -> observer_thread_id);
    pthread_mutex_unlock( & (observer -> end_search_lock));
}

void delete_observer(Observer observer) {
    pthread_mutex_destroy( & (observer -> end_search_lock));
    free(observer);
}
```

manager/observer/observer.h

```
#ifndef __OBSERVER_H__
#define __OBSERVER_H__

#include "../task_buffer/task_buffer.h"
#include "../../worker/worker.h"

typedef struct __Observer * Observer;

void create_observer(
    TaskBuffer buffer, Worker worker,
    Observer * observer);
void start_search(Observer observer);
void end_search(Observer observer);
void delete_observer(Observer observer);

#endif
```

manager/task_buffer/task_buffer.c

```
#include "task_buffer.h"

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

struct __TaskBuffer {
    int free_cell_index;
    Task * buffer;

    pthread_mutex_t edit_buffer_mutex;
    sem_t free_cells;
    sem_t untaken_tasks;
};

void create_task_buffer(int buffer_size, TaskBuffer * buffer) {
    * buffer = (TaskBuffer) malloc(sizeof(struct __TaskBuffer));
    ( * buffer) -> free_cell_index = 0;
    ( * buffer) -> buffer = (Task * ) calloc(buffer_size, sizeof(Task));

    pthread_mutex_init( & ( * buffer) -> edit_buffer_mutex, NULL);
    sem_init( & ( * buffer) -> free_cells, 0, buffer_size);
    sem_init( & ( * buffer) -> untaken_tasks, 0, 0);
}

void add_task(Task task, TaskBuffer buffer) {
```

```
    sem_wait( & (buffer -> free_cells));
    pthread_mutex_lock( & (buffer -> edit_buffer_mutex));
    buffer -> buffer[buffer -> free_cell_index] = task;
    buffer -> free_cell_index++;
    pthread_mutex_unlock( & (buffer -> edit_buffer_mutex));
    sem_post( & (buffer -> untaken_tasks));
}

Task get_task(TaskBuffer buffer) {
    sem_wait( & (buffer -> untaken_tasks));
    pthread_mutex_lock( & (buffer -> edit_buffer_mutex));
    Task task = buffer -> buffer[buffer -> free_cell_index - 1];
    buffer -> free_cell_index--;
    pthread_mutex_unlock( & (buffer -> edit_buffer_mutex));
    sem_post( & (buffer -> free_cells));

    return task;
}

bool is_empty(TaskBuffer buffer) {
    pthread_mutex_lock( & (buffer -> edit_buffer_mutex));
    bool result = buffer -> free_cell_index == 0;
    pthread_mutex_unlock( & (buffer -> edit_buffer_mutex));

    return result;
}

void delete_task_buffer(TaskBuffer buffer) {
    pthread_mutex_destroy( & (buffer -> edit_buffer_mutex));
    free(buffer -> buffer);
    free(buffer);
}
```

manager/task_buffer/task_buffer.h

```
#ifndef __TASK_BUFFER_H__
#define __TASK_BUFFER_H__

#include "../../task/task.h"
#include <stdbool.h>


typedef struct __TaskBuffer *TaskBuffer;

void create_task_buffer(int buffer_size, TaskBuffer *buffer);
void add_task(Task task, TaskBuffer buffer);
Task get_task(TaskBuffer buffer);
void delete_task_buffer(TaskBuffer buffer);
```

```
bool is_empty(TaskBuffer buffer);

#endif
```

manager/manager.c

```
#include "manager.h"

#include "observer/observer.h"

#include "task_buffer/task_buffer.h"

#include "../worker/worker.h"

#include <stdlib.h>

struct __TaskManager {
    Worker * workers;
    Observer * observers;
    TaskBuffer buffer;
    int workers_amount;
};

void __init_workers(int amount, Worker ** arr) {
    * arr = (Worker * ) calloc(amount, sizeof(Worker));
    for (int i = 0; i < amount; i++) create_worker( & ( * arr)[i]);
}

void __init_observers(
    Worker * workers,
    int workers_amount,
    TaskBuffer buffer,
    Observer ** arr)
{
    * arr = (Observer * ) calloc(workers_amount, sizeof(Observer));

    for (int i = 0; i < workers_amount; i++) {
        create_observer(buffer, workers[i], & ( * arr)[i]);
        start_search(( * arr)[i]);
    }
}

void create_task_manager(
    int workers_amount,
    int buffer_size,
    TaskManager * manager)
{
    * manager = (TaskManager) malloc(sizeof(struct __TaskManager));
    ( * manager) -> workers_amount = workers_amount;
```

```
    create_task_buffer(
        buffer_size, &
        ( * manager) -> buffer);

    __init_workers(
        workers_amount, &
        ( * manager) -> workers);

    __init_observers(
        ( * manager) -> workers, workers_amount,
        ( * manager) -> buffer, &
        ( * manager) -> observers);
}

void do_task(Task task, TaskManager manager) {
    add_task(task, manager -> buffer);
}

void wait_for_task_complete(TaskManager manager) {
    while (!is_empty(manager -> buffer));

    for (int i = 0; i < manager -> workers_amount; i++)
        end_search(manager -> observers[i]);

    for (int i = 0; i < manager -> workers_amount; i++)
        if (is_busy(manager -> workers[i]))
            wait_worker(manager -> workers[i]);

    for (int i = 0; i < manager -> workers_amount; i++)
        start_search(manager -> observers[i]);
}

void delete_task_manager(TaskManager manager) {
    for (int i = 0; i < manager -> workers_amount; i++) {
        end_search(manager -> observers[i]);
        delete_observer(manager -> observers[i]);
        delete_worker(manager -> workers[i]);
    }

    delete_task_buffer(manager -> buffer);

    free(manager -> workers);
    free(manager -> observers);
    free(manager);
}
```

manager/manager.h

```
#ifndef __TASK_MANAGER_H__
#define __TASK_MANAGER_H__

#include "../task/task.h"

typedef struct __TaskManager *TaskManager;

void create_task_manager(int workers_amount, int buffer_size, TaskManager
*manager);
void do_task(Task task, TaskManager manager);
void wait_for_task_complete(TaskManager manager);
void delete_task_manager(TaskManager manager);

#endif
```

matrix/matrix.c

```
#include "matrix.h"
#include <stdlib.h>
#include <stdio.h>

struct __MatrixSize {
    int columns;
    int rows;
};

struct __Matrix {
    struct __MatrixSize size;
    MatrixElement * elements;
};

MatrixExitCode create_matrix(int n, int m, PMatrix * matrix) {
    if (n <= 0 || m <= 0)
        return INVALID_MATRIX_SIZE;

    * matrix = (PMatrix) malloc(sizeof(struct __Matrix));

    if ( * matrix == NULL)
        return CREATE_MATRIX_FAILED;

    ( * matrix) -> size.rows = n;
    ( * matrix) -> size.columns = m;

    ( * matrix) -> elements = (MatrixElement * ) calloc(
        n * m, sizeof(MatrixElement));

    if (( * matrix) -> elements == NULL) {
```

```
        free( * matrix);
        return CREATE_MATRIX_FAILED;
    }

    return MATRIX_SUCCESS;
}

int rows(PMatrix matrix) {
    return (matrix -> size).rows;
}

int cols(PMatrix matrix) {
    return (matrix -> size).columns;
}

MatrixExitCode get_cell(
    int i, int j, PMatrix matrix,
    MatrixElement * out_value)
{
    if (i < 0 || i >= rows(matrix) || j < 0 || j >= cols(matrix))
        return INVALID_COORDINATES;

    * out_value = matrix -> elements[i * cols(matrix) + j];
    return MATRIX_SUCCESS;
}

MatrixExitCode set_cell(int i, int j, MatrixElement value, PMatrix
matrix) {
    if (i < 0 || i >= rows(matrix) || j < 0 || j >= cols(matrix))
        return INVALID_COORDINATES;

    matrix -> elements[i * cols(matrix) + j] = value;
    return MATRIX_SUCCESS;
}

void delete_matrix(PMatrix matrix) {
    free(matrix -> elements);
    free(matrix);
}
```

matrix/matrix.h

```
#ifndef __MATRIX_H__
#define __MATRIX_H__

#include <complex.h>

typedef double complex MatrixElement;
```

```
typedef struct __Matrix * PMatrix;

typedef enum {
    MATRIX_SUCCESS = 0,
        CREATE_MATRIX_FAILED = 1,
        INVALID_MATRIX_SIZE = 2,
        INVALID_COORDINATES = 3
}
MatrixExitCode;

MatrixExitCode create_matrix(int n, int m, PMatrix * matrix);

int rows(PMatrix matrix);

int cols(PMatrix matrix);

MatrixExitCode get_cell(int i, int j, PMatrix matrix, MatrixElement *
out_value);

MatrixExitCode set_cell(
    int i, int j, MatrixElement value,
    PMatrix matrix);

void delete_matrix(PMatrix matrix);

#endif
```

task/task.h

```
#ifndef __TASK_H__
#define __TASK_H__

#include "../matrix/matrix.h"

typedef struct {
  PMatrix a, b;
  int i, j;
  PMatrix out_matrix;
} Task;


#endif
```

worker/worker.c

```
#include "worker.h"
```

```
#include "../matrix/matrix.h"
#include <pthread.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>

struct __Worker {
    pthread_t id;
    Task task;
    bool is_busy;
};

void __worker_code(PMatrix a, PMatrix b, int i, int j, PMatrix c) {
    MatrixElement result = 0;

    for (int p = 0; p < cols(a); p++) {
        MatrixElement a_element;
        MatrixElement b_element;

        get_cell(i, p, a, & a_element);
        get_cell(p, j, b, & b_element);

        result += a_element * b_element;
    }

    set_cell(i, j, result, c);
}

void * __worker_code_wrapper(void * data) {
    Worker worker = (Worker) data;

    __worker_code(
        worker -> task.a, worker -> task.b,
        worker -> task.i, worker -> task.j,
        worker -> task.out_matrix
    );

    worker -> is_busy = false;
    pthread_exit(0);
}

void create_worker(Worker * worker) {
    * worker = (Worker) malloc(sizeof(struct __Worker));
    ( * worker) -> is_busy = false;
}

void run_task(Task task, Worker worker) {
    worker -> task = task;
    worker -> is_busy = true;
    pthread_create( & worker -> id, NULL, __worker_code_wrapper, worker);
}
```

```
void wait_worker(Worker worker) {
    pthread_join(worker -> id, NULL);
}

void delete_worker(Worker worker) {
    free(worker);
}

bool is_busy(Worker worker) {
    return worker -> is_busy;
}
```

worker/worker.h

```
#ifndef __WORKER_H__
#define __WORKER_H__

#include "../task/task.h"
#include <stdbool.h>

typedef struct __Worker *Worker;

void create_worker(Worker *worker);
void run_task(Task task, Worker worker);
void wait_worker(Worker worker);
void delete_worker(Worker worker);
bool is_busy(Worker worker);

#endif
```

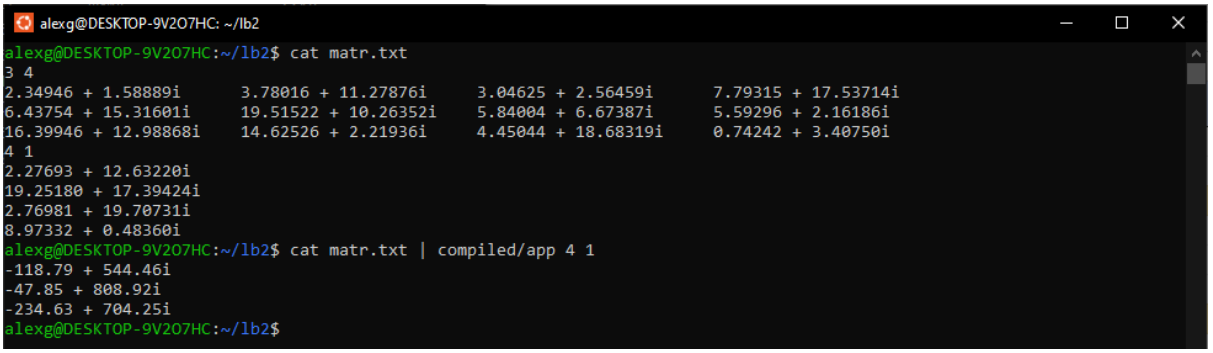
main.c

```
#include "matrix/matrix.h"
#include "io_matrix/io_matrix.h"
#include "manager/manager.h"
#include "task/task.h"
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int workers_amount;
int buffer_size;

void handle_scan_error(ScanMatrixExitCode code) {
    if (code == INVALID_INPUT_FORMAT)
```

```
        perror("[ScanMatrixError] Invalid input data format!\n");

    exit(20 + code);
}

PMatrix multiply(PMatrix a, PMatrix b) {
    if (cols(a) != rows(b)) {
        perror("[MultiplyError] Invalid matrix sizes\n");
        exit(31);
    }

    PMatrix c;
    TaskManager manager;

    create_matrix(rows(a), cols(b), & c);
    create_task_manager(workers_amount, buffer_size, & manager);

    for (int i = 0; i < rows(c); i++) {
        for (int j = 0; j < cols(c); j++) {
            Task task = {
                a,
                b,
                i,
                j,
                c
            };
            do_task(task, manager);
        }
    }

    wait_for_task_complete(manager);
    delete_task_manager(manager);
    return c;
}

int main(int argc, char * argv[]) {
    PMatrix a, b, c;
    int scan_result;

    workers_amount = (argc >= 2) ? atoi(argv[1]) : 1;
    buffer_size = (argc >= 3) ? atoi(argv[2]) : 1;

    scan_result = scan_matrix(stdin, & a);
    if (scan_result != SCAN_MATRIX_SUCCESS)
        handle_scan_error(scan_result);

    scan_result = scan_matrix(stdin, & b);
    if (scan_result != SCAN_MATRIX_SUCCESS)
        handle_scan_error(scan_result);

    c = multiply(a, b);
    print_matrix(stdout, c);
```

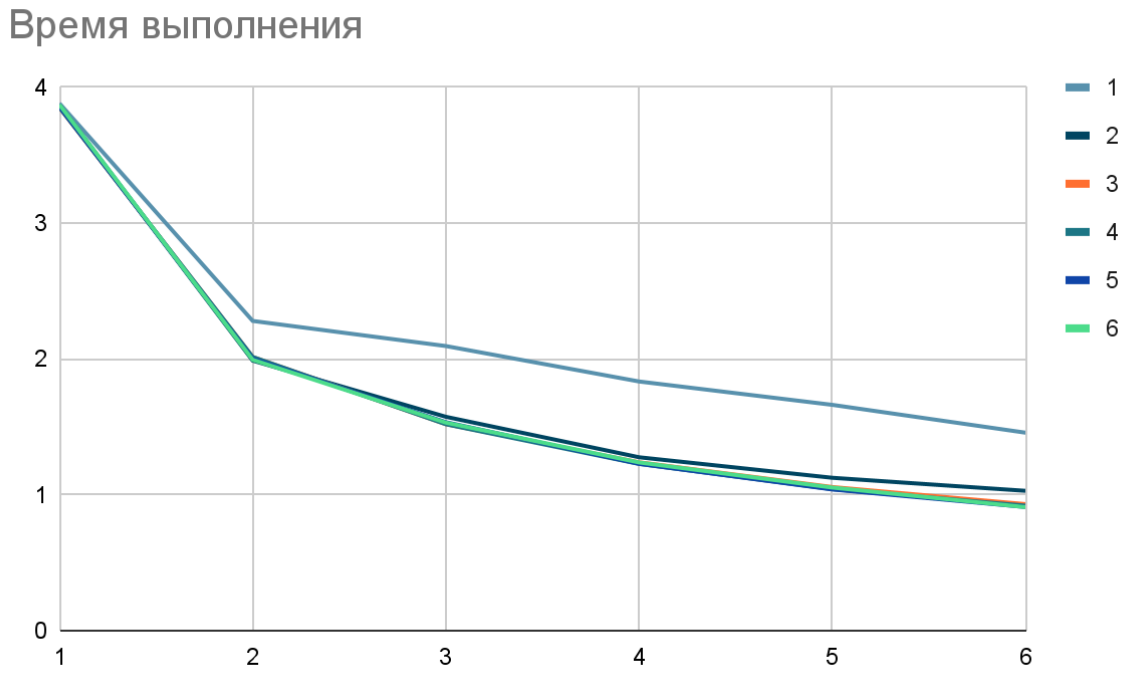
```
    delete_matrix(a);
    delete_matrix(b);
    delete_matrix(c);

    return 0;
}
```

Пример работы



Исследование программы

Исследуем программы на производительность. По вышеперечисленным причинам при оценке эффективности системы я буду производить по двум параметрам: количеству потоков и размеру буфера сообщений. Ниже представлен график времени выполнения программы. Каждый график соответствует определенному размеру буфера, ось X показывает количество потоков, а ось Y – время исполнения. Анализ производился на двух матрицах размерами 200×200 (в итоге, программа должна совершить 40 000 операций умножения комплексных чисел).



Как видим, увеличение количества потоков положительно влияет на время выполнения программы. Более того, мы можем заметить резкий спад времени при переходе от 1-го поточного исполнения к многопоточному. Также, мы можем заметить, что снижение времени исполнения происходит менее интенсивно после момента, когда размер буфера становится больше, чем количество потоков. Итого, мы можем сделать вывод, что самое эффективное соотношение потоков к размеру буфера будет 1 к 1.

Вывод

В ходе выполнения данной лабораторной работы я ознакомился с реализацией многопоточных программ на языке программирования C, изучил основные подходы к написанию асинхронных программ, узнал о недостатках асинхронного подхода и способах синхронизации потоков через такие блокировку как семафор и `mutex`. Также, я изучил библиотеку pthread систем Unix, с помощью которой можно писать многопоточные программы.