Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №2 по курсу

«Операционные системы»

Группа: М8О-210Б-23
Студент: Дворников М.Д.
Преподаватель: Бахарев В.Д.
Оценка: _____
Дата: 18.12.24

Москва, 2024

Постановка задачи

Цель работы

Приобретение практических навыков в:

- Управление потоками в ОС
- Обеспечение синхронизации между потоками

Задание

Составить программу на языке Си(C++), обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, 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 входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

6 вариант:

Произвести перемножение 2-ух матриц, содержащих комплексные числа. Произвести перемножение 2-ух матриц, содержащих комплексные числа

Общий метод и алгоритм решения

Использованные системные вызовы:

- int pthread_create(pthread_t *restrict newthread, const pthread_attr_t *restrict attr, void *(*start_routine)(void *), void *restrict arg)
  Создаёт поток, выполняющий указанную функцию (start_routine) с заданными аргументами (arg).
- int pthread_join(pthread_t th, void **thread_return)
  Ожидает завершения указанного потока и получает результат его выполнения.
- ssize_t write(int fd, const void *buf, size_t n)
  Записывает n байт из буфера buf в файл с файловым дескриптором fd. Возвращает количество записанных байт или -1 в случае ошибки.
- void exit(int status)
  Завершает выполнение программы, закрывая все потоки и освобождая ресурсы.
- Для реализации с использованием atomic:
  Библиотека <stdatomic.h>:
  - Тип данных _Atomic double для представления атомарных значений действительных и мнимых частей матриц.
  - Макросы atomic_store(PTR, VAL) для безопасной записи значения и atomic_load(PTR) для безопасного чтения.
- Для реализации с использованием mutex:
  - pthread_mutex_t — тип данных для работы с мьютексами.
  - int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
    Инициализация мьютекса.
  - int pthread_mutex_lock(pthread_mutex_t *mutex)
    Блокирует мьютекс, ограничивая доступ других потоков к защищаемому ресурсу.

- int pthread_mutex_unlock(pthread_mutex_t *mutex)
  Разблокирует мьютекс, разрешая доступ другим потокам.
- int pthread_mutex_destroy(pthread_mutex_t *mutex)
  Уничтожает мьютекс, освобождая связанные с ним ресурсы.

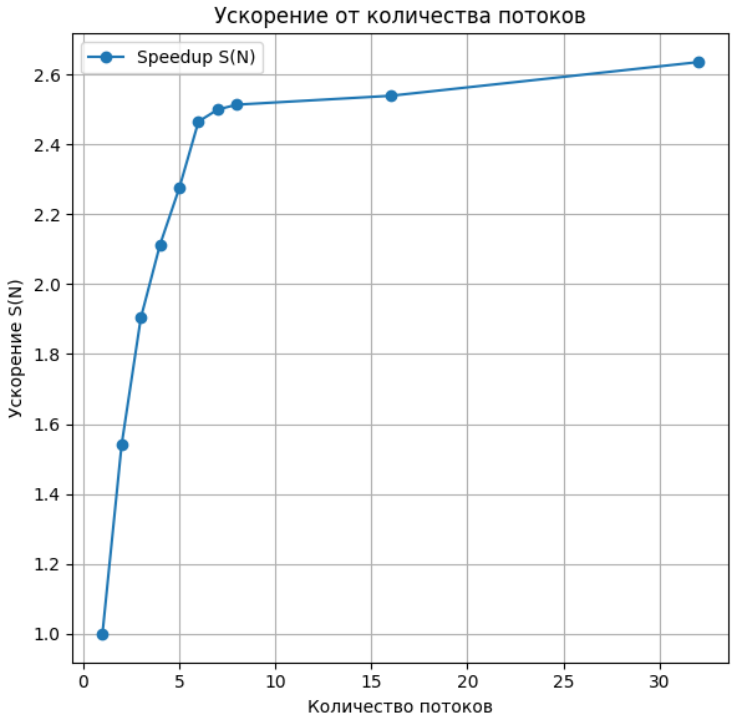
Матрицы генерируются случайным образом. Элементы матриц — комплексные числа, где действительная и мнимая части находятся в диапазоне от -10 до 10. Для генерации случайных чисел используется rand() и диапазон задаётся константой RAND_RANGE.
В программе создаются потоки в количестве, указанном в аргументе командной строки. Каждый поток обрабатывает определённое количество строк итоговой матрицы.
Для реализации с использованием атомиков:
Итоговая матрица представлена как массив атомарных структур, содержащих действительную и мнимую части.
Потоки безопасно записывают значения в итоговую матрицу с помощью операций atomic_store.
Для реализации с использованием мьютексов:
Потоки защищают доступ к итоговой матрице с помощью мьютекса.
Каждая запись в итоговую матрицу выполняется внутри защищённого блока между вызовами pthread_mutex_lock и pthread_mutex_unlock.
После завершения работы всех потоков родительский поток объединяет результаты и выводит итоговую матрицу.
Синхронизация потоков осуществляется с использованием атомиков или мьютексов для предотвращения состояния гонки и обеспечения корректности вычислений.

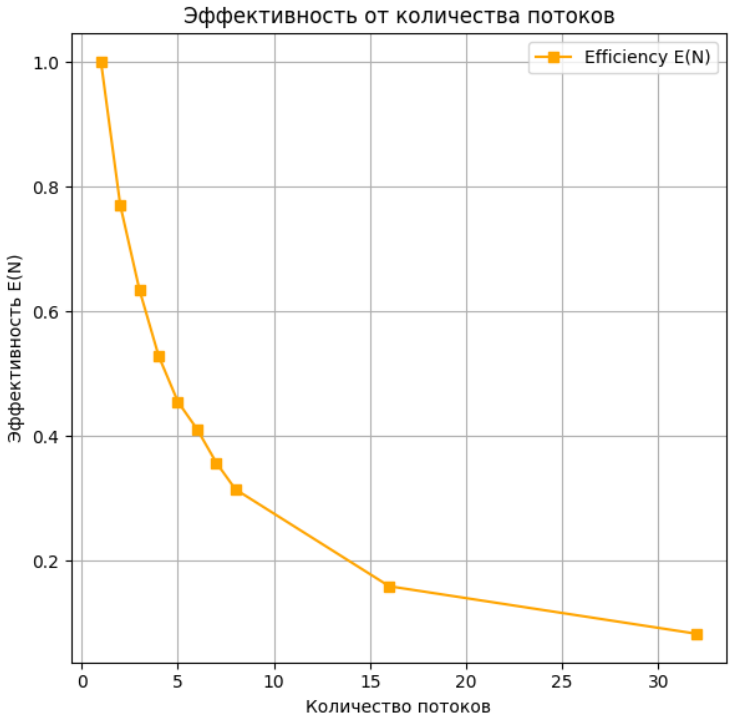
```
matrix_size = 1000
```

Число потоков	Время выполнения, с	Ускорение	Эффективность
1	20.148	1,00	1,00
2	13.073	1.54	0.77
3	10.584	1.90	0,923
4	9.535	2.11	0,895
5	8.858	2.27	0,816
6	8.174	2.46	0,748
7	8.062	2.50	0,69
8	8.017	2.51	0,639
16	7.937	2.54	0,586
32	7.647	2.64	0,545

Ускорение $S_N=T_1/T_N$:
Например, для 2 потоков:
$S_2=20.148/13.073\approx1.54$

Эффективность $E_N=S_N/N$:
Например, для 2 потоков:
$E_2=1.54/2\approx0.77$





1. Ускорение S_N от количества потоков:
Ускорение S_N увеличивается с ростом числа потоков **N**, но этот рост постепенно замедляется. На начальных этапах (до 4-8 потоков) ускорение растет довольно быстро, а затем приближается к плато.
Рост ускорения ограничивается законом Амдала. Чем больше потоков используется, тем меньше времени уходит на параллельные задачи. Однако накладные расходы на управление потоками (координация, синхронизация) и доля последовательного кода начинают преобладать, что ограничивает максимальное ускорение.
При увеличении потоков ускорение растет, но не пропорционально числу потоков, а рост становится всё менее эффективным.

2. Эффективность E_N от количества потоков:
Эффективность E_N значительно падает при увеличении количества потоков. На малом числе потоков эффективность близка к 1 (100%), а при большем количестве потоков она снижается до 0.2−0.30.2-0.30.2−0.3 и ниже.
Эффективность показывает, насколько хорошо используются ресурсы. При большем числе потоков накладные расходы на синхронизацию и управление потоками начинают занимать большую долю времени, а прирост производительности от каждого дополнительного потока уменьшается. Кроме того, при увеличении потоков могут возникать конфликты за ресурсы (например, доступ к памяти), что дополнительно снижает эффективность.
Максимальная эффективность достигается при оптимальном соотношении количества потоков к характеру задачи. При избыточном числе потоков эффективность снижается.

Таким образом, мы подтвердили действие закона Амдала: параллельные вычисления имеют предел, после которого добавление потоков перестает давать значительный прирост производительности.

Для данной задачи оптимальное количество потоков находится в диапазоне от 4 до 8, когда ускорение близко к линейному, а эффективность всё ещё достаточно высока.
Увеличение потоков сверх этого диапазона приводит к существенному снижению эффективности, хотя ускорение продолжает расти, но с минимальной отдачей.

Код программы

mutex.c

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

#define MAX_THREADS 32
#define RAND_RANGE 10

typedef double complex cplx;

pthread_mutex_t mutex;
cplx **result;

typedef struct {
    size_t start_row;
    size_t end_row;
    size_t size;
    cplx **matrix1;
    cplx **matrix2;
} ThreadArgs;

void HandleError(const char *msg) {
    write( fd: STDERR_FILENO, buf: msg, nbyte: strlen( s: msg));
    exit(EXIT_FAILURE);
}

void AllocateMatrix(cplx ***matrix, size_t size) {
    *matrix = malloc( size: size * sizeof(cplx *));
    if (*matrix == NULL) {
        HandleError( msg: "Ошибка выделения памяти для матрицы.\n");
    }
    for (size_t i = 0; i < size; i++) {
        (*matrix)[i] = malloc( size: size * sizeof(cplx));
        if ((*matrix)[i] == NULL) {
            HandleError( msg: "Ошибка выделения памяти для строки матрицы.\n");
        }
    }
}

void FreeMatrix(cplx **matrix, size_t size) {
    for (size_t i = 0; i < size; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

cplx GenerateRandomComplex() {
    double real = (rand() % (2 * RAND_RANGE + 1)) - RAND_RANGE;
    double imag = (rand() % (2 * RAND_RANGE + 1)) - RAND_RANGE;
    return real + imag * I;
}

void *MatrixMultiply(void *args) {
    ThreadArgs *data = (ThreadArgs *)args;

    for (size_t i = data->start_row; i < data->end_row; i++) {
        for (size_t j = 0; j < data->size; j++) {
            cplx sum = 0;
            for (size_t k = 0; k < data->size; k++) {
                sum += data->matrix1[i][k] * data->matrix2[k][j];
            }

            // Синхронизация доступа к результату
            if (pthread_mutex_lock(&mutex) != 0) {
                HandleError( msg: "Ошибка блокировки мьютекса.\n");
            }
            result[i][j] = sum;
            if (pthread_mutex_unlock(&mutex) != 0) {
                HandleError( msg: "Ошибка разблокировки мьютекса.\n");
            }
        }
    }

    return NULL;
}

int main(int argc, char **argv) {
    if (argc != 3) {
        HandleError( msg: "Использование: ./mutex <количество потоков> <размер матрицы>\n");
    }

    size_t threads_count = strtoul( str: argv[1], endptr: NULL, base: 10);
    if (threads_count > MAX_THREADS) {
        HandleError( msg: "Ошибка: Превышено максимальное количество потоков.\n");
    }

    size_t matrix_size = strtoul( str: argv[2], endptr: NULL, base: 10);

    cplx **matrix1, **matrix2;

    AllocateMatrix( matrix: &matrix1, size: matrix_size);
    AllocateMatrix( matrix: &matrix2, size: matrix_size);
    AllocateMatrix( matrix: &result, size: matrix_size);

    srand(time(NULL));

    for (size_t i = 0; i < matrix_size; i++) {
        for (size_t j = 0; j < matrix_size; j++) {
            matrix1[i][j] = GenerateRandomComplex();
            matrix2[i][j] = GenerateRandomComplex();
        }
    }

    if (pthread_mutex_init(&mutex, NULL) != 0) {
        HandleError( msg: "Ошибка инициализации мьютекса.\n");
    }

    pthread_t threads[MAX_THREADS];
    ThreadArgs thread_args[MAX_THREADS];

    size_t rows_per_thread = matrix_size / threads_count;
    for (size_t i = 0; i < threads_count; i++) {
        thread_args[i].start_row = i * rows_per_thread;
        thread_args[i].end_row = (i == threads_count - 1) ? matrix_size : (i + 1) * rows_per_thread;
        thread_args[i].size = matrix_size;
        thread_args[i].matrix1 = matrix1;
        thread_args[i].matrix2 = matrix2;

        if (pthread_create(&threads[i], NULL, MatrixMultiply, &thread_args[i]) != 0) {
            HandleError( msg: "Ошибка создания потока.\n");
        }
    }

    for (size_t i = 0; i < threads_count; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            HandleError( msg: "Ошибка ожидания потока.\n");
        }
    }

    pthread_mutex_destroy(&mutex);

    for (size_t i = 0; i < matrix_size; i++) {
        for (size_t j = 0; j < matrix_size; j++) {
            char buffer[64];
            snprintf(buffer, sizeof(buffer), "(%.2f + %.2fi) ", creal(result[i][j]), cimag(result[i][j]));
            write( fd: STDOUT_FILENO, buf: buffer, nbyte: strlen( s: buffer));
        }
        write( fd: STDOUT_FILENO, buf: "\n", nbyte: 1);
    }

    FreeMatrix( matrix: matrix1, size: matrix_size);
    FreeMatrix( matrix: matrix2, size: matrix_size);
    FreeMatrix( matrix: result, size: matrix_size);

    return EXIT_SUCCESS;
}
```

atomic.c

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

#define MAX_THREADS
#define RAND_RANGE 10

typedef double complex cplx;

typedef struct {
    _Atomic double real;
    _Atomic double imag;
} atomic_cplx;

atomic_cplx **atomic_result;

typedef struct {
    size_t start_row;
    size_t end_row;
    size_t size;
    cplx **matrix1;
    cplx **matrix2;
} ThreadArgs;

void HandleError(const char *msg) {
    write( fd: STDERR_FILENO,  buf: msg,  nbyte: strlen( s: msg));
    exit(EXIT_FAILURE);
}

void AllocateMatrix(cplx ***matrix, size_t size) {
    *matrix = malloc( size: size * sizeof(cplx *));
    if (*matrix == NULL) {
        HandleError( msg: "Ошибка выделения памяти для матрицы.\n");
    }
    for (size_t i = 0; i < size; i++) {
        (*matrix)[i] = malloc( size: size * sizeof(cplx));
        if ((*matrix)[i] == NULL) {
            HandleError( msg: "Ошибка выделения памяти для строки матрицы.\n");
        }
    }
}

void FreeMatrix(cplx **matrix, size_t size) {
    for (size_t i = 0; i < size; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void AllocateAtomicMatrix(atomic_cplx ***matrix, size_t size) {
    *matrix = malloc( size: size * sizeof(atomic_cplx *));
    if (*matrix == NULL) {
        HandleError( msg: "Ошибка выделения памяти для атомарной матрицы.\n");
    }
    for (size_t i = 0; i < size; i++) {
        (*matrix)[i] = malloc( size: size * sizeof(atomic_cplx));
        if ((*matrix)[i] == NULL) {
            HandleError( msg: "Ошибка выделения памяти для строки атомарной матрицы.\n");
        }
    }
}

void FreeAtomicMatrix(atomic_cplx **matrix, size_t size) {
    for (size_t i = 0; i < size; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void *MatrixMultiply(void *args) {
    ThreadArgs *data = (ThreadArgs *)args;

    for (size_t i = data->start_row; i < data->end_row; i++) {
        for (size_t j = 0; j < data->size; j++) {
            cplx sum = 0;
            for (size_t k = 0; k < data->size; k++) {
                sum += data->matrix1[i][k] * data->matrix2[k][j];
            }

            atomic_store(&atomic_result[i][j].real, creal(sum));
            atomic_store(&atomic_result[i][j].imag, cimag(sum));
        }
    }

    return NULL;
}

cplx GenerateRandomComplex() {
    double real = (rand() % (2 * RAND_RANGE + 1)) - RAND_RANGE;
    double imag = (rand() % (2 * RAND_RANGE + 1)) - RAND_RANGE;
    return real + imag * I;
}

int main(int argc, char **argv) {
    if (argc != 3) {
        HandleError( msg: "Использование: ./atomic <количество потоков> <размер матрицы>\n");
    }

    size_t threads_count = strtoul( str: argv[1],  endptr: NULL,  base: 10);
    if (threads_count > MAX_THREADS) {
        HandleError( msg: "Ошибка: Превышено максимальное количество потоков.\n");
    }

    size_t matrix_size = strtoul( str: argv[2],  endptr: NULL,  base: 10);

    cplx **matrix1, **matrix2;

    AllocateMatrix( matrix: &matrix1,  size: matrix_size);
    AllocateMatrix( matrix: &matrix2,  size: matrix_size);
    AllocateAtomicMatrix( matrix: &atomic_result,  size: matrix_size);

    srand(time(NULL));

    for (size_t i = 0; i < matrix_size; i++) {
        for (size_t j = 0; j < matrix_size; j++) {
            matrix1[i][j] = GenerateRandomComplex();
            matrix2[i][j] = GenerateRandomComplex();
        }
    }

    pthread_t threads[MAX_THREADS];
    ThreadArgs thread_args[MAX_THREADS];

    size_t rows_per_thread = matrix_size / threads_count;
    for (size_t i = 0; i < threads_count; i++) {
        thread_args[i].start_row = i * rows_per_thread;
        thread_args[i].end_row = (i == threads_count - 1) ? matrix_size : (i + 1) * rows_per_thread;
        thread_args[i].size = matrix_size;
        thread_args[i].matrix1 = matrix1;
        thread_args[i].matrix2 = matrix2;

        if (pthread_create(&threads[i], NULL, MatrixMultiply, &thread_args[i]) != 0) {
            HandleError( msg: "Ошибка создания потока.\n");
        }
    }

    for (size_t i = 0; i < threads_count; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            HandleError( msg: "Ошибка ожидания потока.\n");
        }
    }

    for (size_t i = 0; i < matrix_size; i++) {
        for (size_t j = 0; j < matrix_size; j++) {
            double real = atomic_load(&atomic_result[i][j].real);
            double imag = atomic_load(&atomic_result[i][j].imag);

            char buffer[64];
            snprintf(buffer, sizeof(buffer), "(%.2f + %.2fi) ", real, imag);
            write( fd: STDOUT_FILENO,  buf: buffer,  nbyte: strlen( s: buffer));
        }
        write( fd: STDOUT_FILENO,  buf: "\n",  nbyte: 1);
    }

    FreeMatrix( matrix: matrix1,  size: matrix_size);
    FreeMatrix( matrix: matrix2,  size: matrix_size);
    FreeAtomicMatrix( matrix: atomic_result,  size: matrix_size);

    return EXIT_SUCCESS;
}
```

Протокол работы программы

Тестирование:

```
./mutex 1 1000  15.63s user 0.49s system 78% cpu 20.620 total
./mutex 2 1000  18.18s user 0.49s system 132% cpu 14.134 total
./mutex 8 1000  21.24s user 0.51s system 273% cpu 7.944 total
./mutex 16 1000  18.82s user 0.51s system 251% cpu 7.686 total
```

dtrace:
SYSCALL(args) = return
(-85.00 + 60.00i) (249.00 + 55.00i)
(-2.00 + 129.00i) (176.00 + -54.00i)
munmap(0x100474000, 0x84000) = 0 0
munmap(0x1004F8000, 0x8000) = 0 0
munmap(0x100500000, 0x4000) = 0 0
munmap(0x100504000, 0x4000) = 0 0
munmap(0x100508000, 0x48000) = 0 0
munmap(0x100550000, 0x4C000) = 0 0
crossarch_trap(0x0, 0x0, 0x0) = -1 Err#45
open(".\0", 0x100000, 0x0) = 3 0
fcntl(0x3, 0x32, 0x16FA030E8) = 0 0
close(0x3) = 0 0
fsgetpath(0x16FA030F8, 0x400, 0x16FA030D8) = 56 0
fsgetpath(0x16FA03108, 0x400, 0x16FA030E8) = 14 0
csrctl(0x0, 0x16FA0350C, 0x4) = -1 Err#1
__mac_syscall(0x1976ABD62, 0x2, 0x16FA03450) = 0 0
csrctl(0x0, 0x16FA034FC, 0x4) = -1 Err#1
__mac_syscall(0x1976A8B95, 0x5A, 0x16FA03490) = 0 0
sysctl([unknown, 3, 0, 0, 0, 0] (2), 0x16FA029F8, 0x16FA029F0, 0x1976AA888, 0xD) = 0 0
sysctl([CTL_KERN, 157, 0, 0, 0, 0] (2), 0x16FA02AA8, 0x16FA02AA0, 0x0, 0x0) = 0 0
open("/\0", 0x20100000, 0x0) = 3 0
openat(0x3, "System/Cryptexes/OS\0", 0x100000, 0x0) = 4 0
dup(0x4, 0x0, 0x0) = 5 0
fstatat64(0x4, 0x16FA02581, 0x16FA024F0) = 0 0
openat(0x4, "System/Library/dyld/\0", 0x100000, 0x0) = 6 0
fcntl(0x6, 0x32, 0x16FA02580) = 0 0
dup(0x6, 0x0, 0x0) = 7 0
dup(0x5, 0x0, 0x0) = 8 0
close(0x3) = 0 0
close(0x5) = 0 0
close(0x4) = 0 0
close(0x6) = 0 0
__mac_syscall(0x1976ABD62, 0x2, 0x16FA02F70) = 0 0
shared_region_check_np(0x16FA02B90, 0x0, 0x0) = 0 0
fsgetpath(0x16FA03110, 0x400, 0x16FA03038) = 82 0
fcntl(0x8, 0x32, 0x16FA03110) = 0 0
close(0x8) = 0 0
close(0x7) = 0 0
getfsstat64(0x0, 0x0, 0x2) = 10 0
getfsstat64(0x1003FC050, 0x54B0, 0x2) = 10 0
getattrlist("/\0", 0x16FA03050, 0x16FA02FC0) = 0 0
stat64("/System/Volumes/Preboot/Cryptexes/OS/System/Library/dyld/dyld_shared_cache_arm64e\0"
, 0x16FA033B0, 0x0) = 0 0
dtrace: error on enabled probe ID 1696 (ID 845: syscall::stat64:return): invalid address (0x0) in
action #11 at DIF offset 12
stat64("/Users/matveyd/CLionProjects/OS-labs-git/lab2/src/mutex\0", 0x16FA02860, 0x0) = 0 0

```
open("/Users/matveyd/CLionProjects/OS-labs-git/lab2/src/mutex\0", 0x0, 0x0)          = 3 0
mmap(0x0, 0x87C8, 0x1, 0x40002, 0x3, 0x0)             = 0x1003FC000 0
fcntl(0x3, 0x32, 0x16FA02978)          = 0 0
close(0x3)             = 0 0
munmap(0x1003FC000, 0x87C8)            = 0 0
open("/Users/matveyd/CLionProjects/OS-labs-git/lab2/src/mutex\0", 0x0, 0x0)          = 3 0
__mac_syscall(0x1976ABD62, 0x2, 0x16FA001F0)          = 0 0
map_with_linking_np(0x16FA00080, 0x1, 0x16FA000B0)             = 0 0
close(0x3)             = 0 0
mprotect(0x1003F0000, 0x4000, 0x1)             = 0 0
open("/dev/dtracehelper\0", 0x2, 0x0)          = 3 0
ioctl(0x3, 0x80086804, 0x16F9FF578)            = 0 0
close(0x3)             = 0 0
shared_region_check_np(0xFFFFFFFFFFFFFFFF, 0x0, 0x0)          = 0 0
access("/AppleInternal/XBS/.isChrooted\0", 0x0, 0x0)           = -1 Err#2
bsdthread_register(0x1979AE0F4, 0x1979AE0E8, 0x4000)          = 1073746399 0
getpid(0x0, 0x0, 0x0)          = 1769 0
shm_open(0x197845F41, 0x0, 0xFFFFFFFF979EC000)         = 3 0
fstat64(0x3, 0x16F9FFBF0, 0x0)         = 0 0
mmap(0x0, 0x8000, 0x1, 0x40001, 0x3, 0x0)             = 0x100404000 0
close(0x3)             = 0 0
csops(0x6E9, 0x0, 0x16F9FFD2C)         = 0 0
ioctl(0x2, 0x4004667A, 0x16F9FFC9C)            = 0 0
mprotect(0x100414000, 0x4000, 0x0)             = 0 0
mprotect(0x100420000, 0x4000, 0x0)             = 0 0
mprotect(0x100424000, 0x4000, 0x0)             = 0 0
mprotect(0x100430000, 0x4000, 0x0)             = 0 0
mprotect(0x100434000, 0x4000, 0x0)             = 0 0
mprotect(0x100440000, 0x4000, 0x0)             = 0 0
mprotect(0x10040C000, 0xC8, 0x1)               = 0 0
mprotect(0x10040C000, 0xC8, 0x3)               = 0 0
mprotect(0x10040C000, 0xC8, 0x1)               = 0 0
mprotect(0x100444000, 0x4000, 0x1)             = 0 0
mprotect(0x100448000, 0xC8, 0x1)               = 0 0
mprotect(0x100448000, 0xC8, 0x3)               = 0 0
mprotect(0x100448000, 0xC8, 0x1)               = 0 0
mprotect(0x10040C000, 0xC8, 0x3)               = 0 0
mprotect(0x10040C000, 0xC8, 0x1)               = 0 0
mprotect(0x100444000, 0x4000, 0x3)             = 0 0
mprotect(0x100444000, 0x4000, 0x1)             = 0 0
issetugid(0x0, 0x0, 0x0)               = 0 0
getentropy(0x16F9FF308, 0x20, 0x0)             = 0 0
getattrlist("/Users/matveyd/CLionProjects/OS-labs-git/lab2/src/mutex\0", 0x16F9FFB90,
0x16F9FFBAC) = 0 0
access("/Users/matveyd/CLionProjects/OS-labs-git/lab2/src\0", 0x4, 0x0)        = 0 0
open("/Users/matveyd/CLionProjects/OS-labs-git/lab2/src\0", 0x0, 0x0)          = 3 0
fstat64(0x3, 0x12BE04470, 0x0)         = 0 0
csrctl(0x0, 0x16F9FFD7C, 0x4)          = 0 0
fcntl(0x3, 0x32, 0x16F9FFA78)          = 0 0
close(0x3)             = 0 0
open("/Users/matveyd/CLionProjects/OS-labs-git/lab2/src/Info.plist\0", 0x0, 0x0)             = -1 Err#2
proc_info(0x2, 0x6E9, 0xD)             = 64 0
csops_audittoken(0x6E9, 0x10, 0x16F9FFE00)             = 0 0
sysctl([unknown, 3, 0, 0, 0, 0] (2), 0x16FA00158, 0x16FA00150, 0x19B0BFD3A, 0x15)             = 0 0
sysctl([CTL_KERN, 155, 0, 0, 0, 0] (2), 0x16FA001E8, 0x16FA001E0, 0x0, 0x0)            = 0 0
bsdthread_create(0x1003EF528, 0x16FA12B18, 0x16FA9B000)        = 1873391616 0
bsdthread_create(0x1003EF528, 0x16FA12B40, 0x16FB27000)        = 1873965056 0
```

thread_selfid(0x0, 0x0, 0x0) = 16822 0
__disable_threadsignal(0x1, 0x0, 0x0) = 0 0
thread_selfid(0x0, 0x0, 0x0) = 16823 0
ulock_wake(0x1000002, 0x16FA9B034, 0x0) = 0 0
__disable_threadsignal(0x1, 0x0, 0x0) = 0 0
ulock_wait(0x1020002, 0x16FA9B034, 0xB07) = 0 0
write(0x1, "(-85.00 + 60.00i) \0", 0x12) = 18 0
write(0x1, "(249.00 + 55.00i) \0", 0x12) = 18 0
write(0x1, "\n\0", 0x1) = 1 0
write(0x1, "(-2.00 + 129.00i) \0", 0x12) = 18 0
write(0x1, "(176.00 + -54.00i) \0", 0x13) = 19 0
write(0x1, "\n\0", 0x1) = 1 0

Вывод

В процессе выполнения данной лабораторной работы я освоил навыки разработки программ, использующих многопоточность, а также научился эффективно синхронизировать потоки между собой. Анализ результатов тестирования программы позволил изучить влияние количества потоков на её производительность и коэффициент ускорения. Было установлено, что при большом объёме данных значительное число потоков может существенно ускорить выполнение задачи, однако эффективность использования ресурсов возрастает только при количестве потоков, сопоставимом с числом логических ядер процессора. Работа над лабораторной стала ценным опытом, так как она впервые позволила мне погрузиться в изучение многопоточности и механизмов синхронизации в языке программирования Си.