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-210БВ-24

Студент: Дмитренко Я.С.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 13.10.25

Москва, 2025

Постановка задачи

Вариант 16.

Задаётся радиус окружности. Необходимо с помощью метода Монте-Карло рассчитать её площадь

Общий метод и алгоритм решения

Использованные системные вызовы:

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg); – Создает поток с заданными атрибутами, который начинает выполнение функции start_routine
- int pthread_join(pthread_t thread, void **retval); – ожидает завершения указанного потока.
- int pthread_mutex_lock(pthread_mutex_t *mutex); – блокирует мьютекс. Предотвращает состояние гонки при одновременном доступе из нескольких потоков
- int pthread_mutex_unlock(pthread_mutex_t *mutex); – разблокирует мьютекс.\
- int clock_gettime(clockid_t clk_id, struct timespec *tp); – получает текущее монотоное время системы

  struct timespec {
  time_t tv_sec; - секунды
  long tv_nsec; - наносекунды
  };

Я реализовал программу, которая использует многопоточность для вычисления площади окружности с использованием метода Монте-Карло. Сначала нужно вписать окружность в квадрат длиной ее радиуса. Далее для этого метода необходимо брать случайные точки и проверять, попали ли они в квадрат или в окружность. Затем площадь окружности вычисляется как (колво точек в окружности/колво точек всего) * площадь квадрата.

Я зафиксировал количество точек в main.c. Количество потоков подается как ключ к моей программе. Затем я отдаю на каждый поток какое то колво точек, чтобы они посчитали, что куда попало. Я делю это равно, то есть общее колво точек/колво потоков.

Также я использую mutex, чтобы предотвратить гонку.

Код программы

main.c

```c
#include <stdint.h>
#include <stdbool.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <unistd.h>
#include <pthread.h>
```

```c
#include <errno.h>

#include <time.h>

typedef struct {
    size_t number;
    double R;
    int count_iterations;
    unsigned int seed;
} ThreadArgs;

static volatile int32_t count_in_circle = 0;
static volatile int32_t count_total = 0;

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

double sequential_calculate(int count_iterations, double R, int seed) {
    // функция для последовательного вычисления
    double x, y;
    int count_in_circle = 0;
    int count_total = 0;

    for (size_t i = 0; i < count_iterations; ++i) {
        x = (double)rand_r(&seed) / RAND_MAX * R;
        y = (double)rand_r(&seed) / RAND_MAX * R;


        if (x * x + y * y <= R * R) {
            ++count_in_circle;
        }
    }

    count_total += count_iterations;
    return 4 * R * R * (count_in_circle/(double)count_total);
}

static void *work(void *_args) {
    ThreadArgs *args = (ThreadArgs *)_args;
    int n = args->count_iterations;
    double x, y;
    int local_count_in_circle = 0;

    unsigned int seed = args->seed;

    for (size_t i = 0; i < n; ++i) {
        x = (double)rand_r(&seed) / RAND_MAX * args->R;
        y = (double)rand_r(&seed) / RAND_MAX * args->R;


        if (x * x + y * y <= args->R * args->R) {
            ++local_count_in_circle;
        }
    }
```

```
    pthread_mutex_lock(&mutex);
    count_total += n;
    count_in_circle += local_count_in_circle;
    pthread_mutex_unlock(&mutex);

    // printf("Threads #%ld says count_in_circle: %d, count_total: %d\n", args->number,
count_in_circle, count_total);

    return NULL;
}

void print_usage(const char* program_name) {
    printf("usage: %s --threads <number>\n", program_name);
    printf("       %s -t <number>\n", program_name);
}

int validate_flags(size_t *n_threads, int argc, char* argv[]) {
    if (argc != 3) {
        print_usage(argv[0]);
        return 1;
    }

    if (strcmp(argv[1], "--threads") == 0 || strcmp(argv[1], "-t") == 0) {
        char* endptr;
        errno = 0;
        *n_threads = strtoll(argv[2], &endptr, 10);

        if (errno == ERANGE) {
            printf("Произошло переполнение\n");
            print_usage(argv[0]);
            return 1;
        }
        else if (endptr == argv[2] || *endptr != '\0') {
            printf("Введите число\n");
            print_usage(argv[0]);
            return 1;
        }
    }
    else {
        print_usage(argv[0]);
        return 1;
    }
}


int main(int argc, char* argv[]) {
    int count = 1000000000;

    size_t n_threads = 0;
    int flag = validate_flags(&n_threads, argc, argv);
    if (flag) {
        return 1;
    }
```

```
// --n_threads; // один поток выполняет main

double R;
printf("Введите радиус окружности: ");
scanf("%lf", &R);


pthread_t *threads = malloc(n_threads * sizeof(pthread_t));
ThreadArgs *thread_args = malloc(n_threads * sizeof(ThreadArgs));

struct timespec start;

clock_gettime(CLOCK_MONOTONIC, &start);


for (size_t i = 0; i < n_threads; ++i) {
    thread_args[i] = (ThreadArgs) {
        .number = i,
        .R = R,
        .count_iterations = count / n_threads,
        .seed = rand(),
    };

    pthread_create(&threads[i], NULL, work, &thread_args[i]);
}


for (size_t i = 0; i < n_threads; ++i) {
    pthread_join(threads[i], NULL);
}


struct timespec end;
clock_gettime(CLOCK_MONOTONIC, &end);

double time = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;

double square = 4 * (count_in_circle/(double)count_total) * R * R;
printf("Parallel version:\n");
printf("***\nSquare circle: %lf\n", square);
printf("Working time: %lfs\n", time);
printf("Count threads: %ld\n", n_threads);
printf("***\n\n");

free(thread_args);
free(threads);


printf("Sequntial version:\n");

clock_gettime(CLOCK_MONOTONIC, &start);
double result = sequential_calculate(count, R, rand());
clock_gettime(CLOCK_MONOTONIC, &end);
```

```
    time = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;

    printf("***\nSquare circle: %lf\n", result);
    printf("Working time: %lfs\n", time);
    printf("***\n");


    return 0;
}
```

Протокол работы программы

❯ ./main.out -t 1

Введите радиус окружности: 1

****

Square circle: 3.141585

Working time: 9.196175s

Count threads: 1

❯ ./main.out -t 2

Введите радиус окружности: 1

****

Square circle: 3.141581

Working time: 4.617311s

Count threads: 2

❯ ./main.out -t 8

Введите радиус окружности: 1

****

Square circle: 3.141580

Working time: 1.687081s

Count threads: 8

❯ ./main.out -t 12

Введите радиус окружности: 1

****

Square circle: 3.141591

Working time: 1.261245s

Count threads: 12

❯ ./main.out -t 16

Введите радиус окружности: 1

****

Square circle: 3.141585

Working time: 1.032265s

Count threads: 16

❯ ./main.out -t 1024

Введите радиус окружности: 1

****

Square circle: 3.141578

Working time: 0.957375s

Count threads: 1024

❯ ./main.out -t 8096

Введите радиус окружности: 1

****

Square circle: 3.141578

Working time: 1.085246s

Count threads: 8096

❯ ./main.out -t 16192

Введите радиус окружности: 1

****

Square circle: 3.141559

Working time: 1.272385s

Count threads: 16192

****

Вывод

В ходе данной лабораторной работы я научился работать с потоками. Разобрался с различными проблемами и нюансами при работе с ними (гонка). Также получил следующие выводы, которые я изображу в виде таблицы

Число потоков	Время исполнения (мс)	Ускорение	Эффективность
1	**13408.602**	**1.00**	**1.00**
2	**6739.873**	**1.99**	**0.99**
8	**1948.613**	**5.45**	**0.68**
12	**1587.313**	**7.29**	**0.61**
16	**1495.700**	**8.91**	**0.56**
1024	**1612.172**	**9.61**	**0.0094**
8096	**1807.809**	**8.47**	**0.00105**
16192	**2102.531**	**7.23**	**0.00045**

Расчеты:

- **Ускорение: T1/Tn, где T1 – время выполнения с 1 потоком,**
  Tn – время выполнения с n потоками
- **Эффективность: Ускорение/n**

Анализ результатов:

1. Количество потоков МЕНЬШЕ логических ядер процессора (1–8 потоков)

- Наблюдается близкое к линейному ускорение (1.99× при 2 потоках, 5.45× при 8 потоках)
- Эффективность остаётся относительно высокой (0.68–0.99)
- Накладные расходы на создание и управление потоками минимальны
- Потоки практически не конкурируют за ресурсы процессора
- Использование мьютекса не оказывает существенного влияния на производительность из-за редких обращений

Вывод:
В данном диапазоне многопоточность обеспечивает наибольшую отдачу, так как вычислительные ресурсы процессора используются эффективно, а накладные расходы минимальны.

2. **Количество потоков, БЛИЗКОЕ к числу логических ядер процессора (12–16 потоков)**

- Ускорение продолжает расти (до 8.91× при 16 потоках)
- Эффективность снижается до 0.56
- Процессор оказывается практически полностью загружен
- Начинают проявляться накладные расходы на управление потоками и переключения контекста

Вывод:
Достигается практический предел эффективного распараллеливания. Дальнейшее увеличение числа потоков приводит к снижению эффективности, несмотря на небольшой прирост ускорения.

3. Количество потоков БОЛЬШЕ логических ядер процессора (более 16 потоков)

- Наблюдается резкое падение эффективности (0.0094 при 1024 потоках)
- Ускорение перестаёт расти пропорционально числу потоков
- При дальнейшем увеличении числа потоков производительность начинает деградировать

Основные причины:

- Значительные накладные расходы на создание и завершение большого числа потоков
- Перегрузка планировщика операционной системы
- Частые переключения контекста
- Потоки значительную часть времени находятся в ожидании выполнения, а не выполняют полезные вычисления

При числе потоков порядка десятков тысяч время работы планировщика становится сопоставимым с временем выполнения полезной работы.

Вывод:
Использование количества потоков, значительно превышающего число логических ядер процессора, приводит к крайне низкой эффективности и деградации производительности. Небольшое улучшение времени выполнения при 1024 потоках по сравнению с 16 потоками обусловлено особенностями планирования потоков операционной системой и не отражает реального роста вычислительной эффективности.