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»

Выполнил: Ю.В. Павлова
Группа: М80-207БВ-24
Преподаватель: Е. С. Миронов

Москва, 2025

Условие

Цель работы: Приобретение практических навыков в:

- Управлении процессами в ОС
- Обеспечение обмена данных между процессами посредством каналов

Задание: Составить программу на языке Си, 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.

Вариант 16: Задаётся радиус окружности. Необходимо с помощью метода Монте-Карло рассчитать её площадь.

Метод решения

Алгоритм решения задачи:

1. Пользователь запускает программу с двумя параметрами: радиус окружности и максимальное количество потоков.
2. Программа генерирует 1 000 000 000 (миллиард) случайных точек в квадрате, описанном вокруг окружности заданного радиуса.
3. Точки равномерно распределяются между потоками.
4. Каждый поток независимо генерирует свои точки и подсчитывает, сколько из них попадают внутрь окружности.
5. Для синхронизации доступа к общему счетчику попаданий используется мьютекс.
6. После завершения работы всех потоков вычисляется площадь окружности по формуле:

$$S = 4 \times R^2 \times \frac{\text{количество попавших точек}}{\text{общее количество точек}}$$

Математическая основа метода Монте-Карло:

Метод основан на использовании случайных выборок для решения детерминированных задач. Вероятность того, что случайно выбранная точка внутри квадрата со стороной $2R$ попадет в окружность радиуса R, равна отношению площади окружности к площади квадрата:

$$P = \frac{\pi R^2}{(2R)^2} = \frac{\pi}{4}$$

Следовательно, площадь окружности можно оценить как:

$$S \approx 4 \times R^2 \times \frac{M}{N}$$

где M — количество точек внутри окружности, N — общее количество точек.

Описание программы

Архитектура программы:

```
lab2/
 build/
 include/
    exceptions.h
    threads.h
 src/
    threads.cpp
 main.cpp
```

Основные компоненты:

- `main.cpp` — основная программа, реализующая метод Монте-Карло для вычисления площади окружности.
- `include/exceptions.h` — объявление классов исключений.
- `include/threads.h` — объявление класса Thread для работы с потоками.
- `src/threads.cpp` — реализация методов класса Thread.

Основные функции и структуры:

- `struct GlobalResult` — хранит общие данные для всех потоков (общее количество попаданий и мьютекс для синхронизации).
- `struct ThreadArgs` — аргументы, передаваемые в каждый поток (радиус, количество точек для генерации, указатель на глобальные данные).
- `void* calculate_area_chunk(void* args)` — функция, выполняемая в каждом потоке. Генерирует случайные точки и подсчитывает попадания внутрь окружности.
- `int main(int argc, char* argv[])` — точка входа в программу, управляет созданием и синхронизацией потоков.

Программа использует мьютекс (`pthread_mutex_t`) для синхронизации доступа к общему счетчику, что предотвращает гонки данных при одновременной записи от нескольких потоков.

Результаты

Для исследования зависимости производительности от количества потоков были проведены замеры времени выполнения программы с различным количеством потоков на фиксированном объеме данных (1 млрд точек) и радиусе окружности $R = 1.0$.

Количество потоков	**Время (сек)**	**Ускорение**	**Эффективность**
1	253.97	1.00	1.00
2	120.96	2.10	1.05
4	60.02	4.23	1.06
8	37.72	6.73	0.84
16	31.80	7.99	0.50
32	28.61	8.88	0.28

Таблица 1: Зависимость времени выполнения от количества потоков

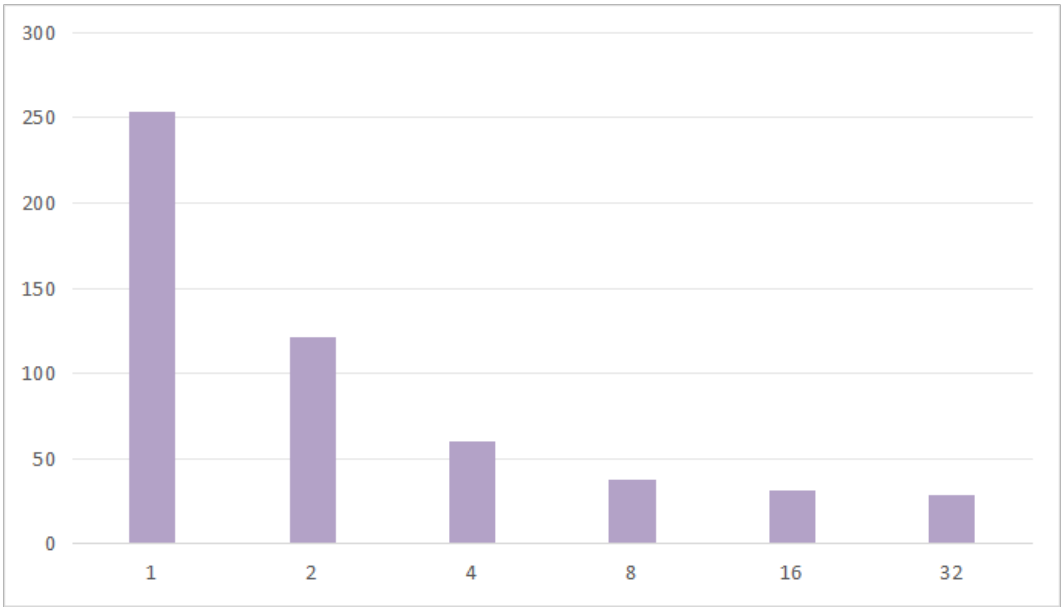

Рис. 1: Диаграмма зависимости времени от количества потоков

Выводы

- Наилучшая эффективность (106%) достигается при 4 потоках
- Максимальное ускорение составляет 8.9 раз при 32 потоках
- Оптимальный диапазон — 4-8 потоков для данного алгоритма
- После 8 потоков наблюдается значительное снижение эффективности
- Алгоритм хорошо масштабируется до количества физических ядер процессора

Исходный код

exceptions.h

```
#pragma once

#include <exception>
#include <string>

namespace exceptions {
    class ThreadException: public std::exception {
    public:
        explicit ThreadException(const std::string& text): error_message_(text)
    {}
        const char* what() const noexcept override {
            return error_message_.c_str();
        }
    private:
        std::string error_message_;
    };

    class ArgumentException: public std::exception {
    public:
```

```
19          explicit ArgumentException(const std::string& text): error_message_(
    text) {}
20          const char* what() const noexcept override {
21              return error_message_.c_str();
22          }
23      private:
24          std::string error_message_;
25      };
26  }
```

Листинг 1: Заголовочный файл исключений

threads.h

```
1  #pragma once
2
3  #include "exceptions.h"
4
5  namespace thread {
6      using threadFunc = void* (*)(void*);
7      struct threadInfo;
8
9      class Thread {
10         private:
11             threadFunc func;
12             threadInfo* pimpl;
13             bool is_joined = true;
14
15         public:
16
17             explicit Thread(threadFunc func);
18             Thread(const Thread&) = delete;
19             Thread& operator=(const Thread&) = delete;
20             Thread(Thread&& other) noexcept;
21             Thread& operator=(Thread&& other) noexcept;
22             void Run(void* threadData);
23             void Join();
24             ~Thread() noexcept;
25
26     };
27
28 }
```

Листинг 2: Заголовочный файл потоков

threads.cpp

```
1  #include "threads.h"
2  #include <iostream>
3  #include <utility>
4  #include <pthread.h>
5
6  namespace thread {
7          struct threadInfo {
8              pthread_t thread = 0;
9          };
10
11         Thread::Thread(threadFunc func) : func(func), is_joined(true) {
```

```
12              pimpl = new threadInfo();
13          }
14
15          Thread::Thread(Thread&& other) noexcept
16          : func(other.func), pimpl(other.pimpl), is_joined(other.is_joined) {
17              other.pimpl = nullptr;
18              other.is_joined = true;
19          }
20
21          Thread& Thread::operator=(Thread&& other) noexcept {
22              if (pimpl != nullptr && !is_joined) {
23                  pthread_detach(pimpl->thread);
24              }
25              delete pimpl;
26
27              Thread temp = std::move(other);
28              std::swap(func, temp.func);
29              std::swap(pimpl, temp.pimpl);
30              std::swap(is_joined, temp.is_joined);
31
32              return *this;
33          }
34
35          void Thread::Run(void* threadData) {
36              if (pimpl == nullptr) {
37                  throw exceptions::ThreadException("Thread is not initialized.")
    ;
38              }
39
40              if (!is_joined) {
41                  throw exceptions::ThreadException("Thread is already running.")
    ;
42              }
43
44              int result = pthread_create(
45                  &(pimpl->thread),
46                  nullptr,
47                  func,
48                  threadData
49              );
50
51              if (result != 0) {
52                  pimpl->thread = 0;
53                  throw exceptions::ThreadException("Failed to create thread");
54              }
55              is_joined = false;
56          }
57
58          void Thread::Join() {
59              if (pimpl == nullptr || is_joined) {
60                  return;
61              }
62
63              int result = pthread_join(pimpl->thread, nullptr);
64
65              if (result != 0) {
66                  throw exceptions::ThreadException("Failed to join thread");
67              }
68
69              pimpl->thread = 0;
```

```
            is_joined = true;
        }

        Thread::~Thread() noexcept {
            if (pimpl != nullptr) {
                if (!is_joined) {
                    std::cerr << "Warning: Thread resource leaked (not joined).
    Detaching..." << std::endl;
                    pthread_detach(pimpl->thread);
                }
                delete pimpl;
            }
        }
}
```

Листинг 3: Реализация работы с потоками

main.cpp

```
#include "threads.h"
#include "exceptions.h"

#include <iostream>
#include <vector>
#include <iomanip>
#include <cmath>
#include <stdexcept>
#include <chrono>
#include <random>
#include <string>

struct GlobalResult {
    long long total_hits = 0;
    pthread_mutex_t mutex;
};

struct ThreadArgs {
    double radius;
    long long points_to_generate;
    GlobalResult* global_res;
};

void* calculate_area_chunk(void* args) {
    ThreadArgs* thread_args = static_cast<ThreadArgs*>(args);

    double R = thread_args->radius;
    long long N = thread_args->points_to_generate;
    double R_squared = R * R;
    long long local_hits = 0;

    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_real_distribution<> distrib(-R, R);

    for (long long i = 0; i < N; ++i) {
        double x = distrib(generator);
        double y = distrib(generator);

        if ((x * x + y * y) <= R_squared) {
```

```
            local_hits++;
        }
    }

    GlobalResult* global_res = thread_args->global_res;

    pthread_mutex_lock(&global_res->mutex);
    global_res->total_hits += local_hits;
    pthread_mutex_unlock(&global_res->mutex);

    return nullptr;
}

int main(int argc, char* argv[]) {
    double radius;
    int max_threads;
    const long long TOTAL_POINTS = 1000000000LL;

    try {
        if (argc != 3) {
            throw exceptions::ArgumentException("Incorrect number of arguments.
    ");
        }

        radius = std::stod(argv[1]);
        max_threads = std::stoi(argv[2]);

        if (radius <= 0 || max_threads <= 0) {
            throw exceptions::ArgumentException("Radius and Max_Threads must be
     positive numbers.");
        }
    }

    catch (const std::exception& e) {
        std::cerr << "Startup Error: " << e.what() << std::endl;
        std::cerr << "Usage: " << argv[0] << " <Radius> <Max_Threads>" << std::
    endl;
        return 1;
    }

    long long points_per_thread = TOTAL_POINTS / max_threads;

    GlobalResult global_res;
    if (pthread_mutex_init(&global_res.mutex, nullptr) != 0) {
        std::cerr << "Fatal error: Failed to initialize mutex." << std::endl;
        return 1;
    }

    std::vector<thread::Thread> thread_pool;
    std::vector<ThreadArgs> args_data(max_threads);

    for (int i = 0; i < max_threads; ++i) {
        args_data[i].radius = radius;
        args_data[i].points_to_generate = points_per_thread;
        args_data[i].global_res = &global_res;

        thread_pool.emplace_back(calculate_area_chunk);
    }

    auto start_time = std::chrono::high_resolution_clock::now();
```

```
98
99      try {
100         for (int i = 0; i < max_threads; ++i) {
101             thread_pool[i].Run(&args_data[i]);
102         }
103
104         for (int i = 0; i < max_threads; ++i) {
105             thread_pool[i].Join();
106         }
107     }
108     catch (const exceptions::ThreadException& e) {
109         std::cerr << "Critical Error (Create): " << e.what() << std::endl;
110         pthread_mutex_destroy(&global_res.mutex);
111         return 1;
112     }
113
114     auto end_time = std::chrono::high_resolution_clock::now();
115     double execution_time = std::chrono::duration<double>(end_time - start_time
    ).count();
116
117     long long total_hits = global_res.total_hits;
118     double area_square = 4.0 * radius * radius;
119
120     double estimated_area = area_square * (static_cast<double>(total_hits) /
    TOTAL_POINTS);
121     double analytical_area = M_PI * radius * radius;
122
123     std::cout << "\n--- Monte Carlo Simulation Results ---" << std::endl;
124     std::cout << "Threads Used:      " << max_threads << std::endl;
125     std::cout << "Total Points:      " << TOTAL_POINTS << std::endl;
126     std::cout << "Total Hits:        " << total_hits << std::endl;
127     std::cout << "Estimated Area:    " << std::fixed << std::setprecision(8) <<
     estimated_area << std::endl;
128     std::cout << "Analytical Area:   " << std::fixed << std::setprecision(8) <<
     analytical_area << std::endl;
129     std::cout << "Execution Time:    " << std::fixed << std::setprecision(6) <<
     execution_time << " seconds" << std::endl;
130
131     pthread_mutex_destroy(&global_res.mutex);
132
133     return 0;
134 }
```

Листинг 4: Основная программа