Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"

Кафедра 806 вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

**ПОТОКИ**

Студент: Парастаев Сослан Геннадьевич

Группа: М8О–203БВ–25

Вариант: 20

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_ \_\_\_\_\_

Дата: \_18.10.2025

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва 2025

**Постановка задачи**

## Цель работы

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 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.

**Общие сведения о программе**

Дан массив координат (x, y, z). Необходимо найти три точки, которые образуют треугольник максимальной площади.

В программе используются следующие средства:

1. **pthread\_t** – Это тип данных, который описывает поток (его идентификатор, дескриптор).
2. **pthread\_create()** – Создаёт новый поток выполнения.
   * &threads[t] // Куда сохранить идентификатор потока
   * NULL // Атрибуты (NULL = по умолчанию, создание потока с настройками по умолчанию)
   * worker\_1 // Функция, которую поток должен выполнить
   * &data[t] // аргумент, который передаётся в эту функцию
3. **pthread\_join()** – Главный (или любой другой) поток ждёт завершения указанного потока. Это блокирующая операция.
4. **pthread\_mutex\_t** – Объект синхронизации, который предотвращает одновременный доступ к критической секции.
5. **PTHREAD\_MUTEX\_INITIALIZER** – Макрос для статической инициализации мьютекса.
6. **pthread\_mutex\_lock(&mutex) –** Поток пытается захватить мьютекс.
   * Если мьютекс свободен — поток захватывает и продолжает
   * Если мьютекс занят — поток блокируется (ждёт), пока другой поток не выполнит pthread\_mutex\_unlock
7. **pthread\_mutex\_unlock(&mutex) –** освобождает мьютекс, который ранее был захвачен текущим потоком.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы вышенаписанных средств
2. Подготовить структуру репозитория
3. Объявляем переменные и новые типы для удобства:

* pthread\_t threads[max\_threads] – массив с идентификаторами потоков
* Point – Набор 3 вещественных координат x, y, z
* Triangle – Набор 3 точек типа Point
* Thread\_data – Тип данных для передачи в поток:
  + Point\* points – Массив точек с координатами (x, y, z)
  + int count\_points – Количество всего точек
  + int start\_i – Начальный индекс части массива для потока
  + int end\_i – Конечный индекс массива для потока
  + double thread\_max\_area – локальный максимум
* triangle\_area() – Расчёт площади треугольника по 3 точкам
* worker\_1() – Метод 1 с фиксированным диапазоном (делим массив на части для индекса i)
* worker\_2() – Метод 2 с динамическим распределением (поочередно потоки берут индекс i) (эти функции передаются в pthread\_create, как функция, которую поток выполнит, второй метод, как тесты показывают, быстрее)

1. Алгоритм:
   * + 1. Создаю новые типы данных для удобства работы с координатами точек и с точками треугольника
       2. Создаю функцию подсчёта площади треугольника по 3 точкам на основе длины векторного произведения двух сторон деленное по полам
       3. Передаю при вызове программы имя файла и количество потоков.

./main test\_0.txt 20

* + - 1. Создаю массив pthread\_t threads[max\_threads], содержащий идентификаторы потоков и Point points[1000], содержащий до 1000 точек
      2. В цикле считываю файл построчно и записываю в i индекс точку point
      3. Замеряю время для будущего исследования эффективности многопоточности
      4. Использую 1-ый метод – разбиение массива на части для индекса i, одна часть = [start\_i:end\_i] размером chunk\_size= ceil((count\_points - 2) / max\_threads). После перебираем все точки после i (j и k) для расчёта локального максимума площади.
      5. В цикле “собираю” данных, которые понадобятся для потока в специальный тип данных Thread\_data, создаю поток через pthread\_create(), передавая куда сохранить идентификатор потока, функцию worker\_1(), которую надо выполнить и данные для этой функции
      6. После в цикле ожидаю завершение всех потоков, считая глобальный максимум площади из всех локальных максимумов.
      7. Замеряю время для будущего исследования эффективности многопоточности
      8. Использую 2-ой метод – динамическое распределение, потоки поочередно берут индекс i. Для этого надо использовать mutex, чтобы один и тот же индекс i не был использован разными потоками. Для этого до и после “захвата” индекса делает mutex\_lock и mutex\_unlock. После перебираем все точки после i (j и k) для расчёта локального максимума площади.
      9. Повторяю пункты 8-10 с worker\_2().
      10. Вывод результата в терминал.

**Основные файлы программы**

**main.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <string.h>  #include <time.h>  #include <math.h> // gcc -o main main.c -lm  /\*  Функция sqrt() объявлена в <math.h>,  но сама она живёт в отдельной библиотеке — libm (math library).  Поэтому при компиляции нужно добавить ключ -lm, чтобы подключить её.  \*/  #include "functions.h"  void\* worker\_1(void\* arg){  Thread\_data\* d = (Thread\_data\*)arg;  double local\_max = 0.0;  for (int i = d->start\_i; i < d->end\_i; ++i){  for (int j = i + 1; j < d->count\_points - 1; ++j){  for (int k = j + 1; k < d->count\_points; ++k){  double area = triangle\_area(d->points[i], d->points[j], d->points[k]);  if (area > local\_max) {local\_max = area;}  }  }  }  d->thread\_max\_area = local\_max;  return NULL;  }  int current\_i = 0;  pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  void\* worker\_2(void\* arg){  Thread\_data\* data = (Thread\_data\*)arg;  double local\_max = 0.0;  while (1) {  int i;  // захватываем индекс i  pthread\_mutex\_lock(&mutex); // Возвращаемое значение: 0 при успехе, код ошибки при неудаче.  if (current\_i >= data->count\_points - 2) {  pthread\_mutex\_unlock(&mutex); // Возвращаемое значение: 0 при успехе, ошибка если мьютекс не был захвачен этим потоком.  break; // больше i нет  }  i = current\_i;  current\_i++;  pthread\_mutex\_unlock(&mutex);  // вычисляем комбинации для текущего i  for (int j = i+1; j < data->count\_points-1; ++j) {  for (int k = j+1; k < data->count\_points; ++k) {  double area = triangle\_area(data->points[i], data->points[j], data->points[k]);  if (area > local\_max) local\_max = area;  }  }  }  data->thread\_max\_area = local\_max;  return NULL;  }  int main(int argc, char \*argv[]){  if (argc < 3){  printf("Usage: %s {input\_file} {max\_threads}\n", argv[0]);  return 1;  }  const char\* filename = argv[1]; // Имя файла с строками, содержащие координаты x, y, z  int max\_threads;    // sscanf(string, "%d %d %d", &a, &b, &c) строку записываем в три числовые переменные  // Считываем строку argv[2] - заданное кол-во потоков от пользователя и "считываем" её как число, записывая в max\_threads  if (sscanf(argv[2], "%d", &max\_threads) != 1 || max\_threads < 0 || max\_threads > 20){  printf("Error: max\_threads must be 0 < max\_threads <= 20\n");  return 1;  };  pthread\_t threads[max\_threads]; // pthread\_t — Это тип данных, который описывает поток (его идентификатор, дескриптор).  Point points[1000]; // Массив, содержащий в каждой ячейке класс Point, которые представляет из себя 3 атрибута x, y, z  const char\* base\_path = "../test/";  char full\_path[256];  // sprintf - Записываем строки base\_path и filename как "%s%s", то есть соединяем в переменную full\_path  sprintf(full\_path, "%s%s", base\_path, filename);  // printf("%s", full\_path);  // Чтение с файла точек в массив  FILE\* file = fopen(full\_path, "r");  if (!file) {  fprintf(stderr, "Error opening file\n");  exit(1);  }  int count\_points = 0;  while (fscanf(file, "%lf %lf %lf", &points[count\_points].x, &points[count\_points].y, &points[count\_points].z) == 3){  count\_points++;  }  fclose(file);  if (count\_points < 3)  {  fprintf(stderr, "Error: Few points for triangle\n");  return 1;  }  // printf("Count of points: %d\n", count\_points);  // for (int i = 0; i < count\_points; ++i){  // printf("%.2lf %.2lf %.2lf\n", points[i].x, points[i].y, points[i].z);  // }  // ================================== Метод 1 =======================================  // Замер времени  struct timespec t\_start, t\_end;  double time\_worker1, time\_worker2;  clock\_gettime(CLOCK\_MONOTONIC, &t\_start);  int chunk\_size = max(1, ceil((count\_points - 2) / max\_threads));  Thread\_data data[max\_threads];  for (int t = 0; t < max\_threads; ++t){  data[t].points = points;  data[t].count\_points = count\_points;  data[t].start\_i = t \* chunk\_size;  data[t].end\_i = data[t].start\_i + chunk\_size;  if (data[t].end\_i > count\_points - 2) {data[t].end\_i = count\_points - 2;}  pthread\_create(  &threads[t], // Куда сохранить идентификатор потока  NULL, // Атрибуты (NULL = по умолчанию, создание поток с настройками по умолчанию)  worker\_1, // Функция, которую поток должен выполнить  &data[t] // аргумент, который передаётся в эту функцию  );  }  double global\_max = 0.0;  for (int t = 0; t < max\_threads; ++t){  pthread\_join(threads[t], NULL); // второй аргумент – указатель, куда будет записан возврат из потока. NULL, так как ничего не возвращаем  if (data[t].thread\_max\_area > global\_max)  {global\_max = data[t].thread\_max\_area;}  }  clock\_gettime(CLOCK\_MONOTONIC, &t\_end);  time\_worker1 = elapsed\_ms(t\_start, t\_end);  printf("Метод 1 с фиксированным диапазоном (делим массив на части для индекса i):\nМаксимальная площадь = %.2f\nвремя = %.2f ms\n", global\_max, time\_worker1);    // ================================== Метод 2 ======================================  clock\_gettime(CLOCK\_MONOTONIC, &t\_start);  for (int t = 0; t < max\_threads; ++t){  data[t].points = points;  data[t].count\_points = count\_points;  data[t].start\_i = t \* chunk\_size;  data[t].end\_i = data[t].start\_i + chunk\_size;  if (data[t].end\_i > count\_points - 2) {data[t].end\_i = count\_points - 2;}  pthread\_create(  &threads[t],  NULL,  worker\_2,  &data[t]  );  }    global\_max = 0.0;  for (int t = 0; t < max\_threads; ++t){  pthread\_join(threads[t], NULL);  if (data[t].thread\_max\_area > global\_max)  {global\_max = data[t].thread\_max\_area;}  }  clock\_gettime(CLOCK\_MONOTONIC, &t\_end);  time\_worker2 = elapsed\_ms(t\_start, t\_end);  printf("Метод 2 с динамическим распределением (поочередно потоки берут индекс i):\nМаксимальная площадь = %.2f\nвремя = %.2f ms\n", global\_max, time\_worker2);  return 0;  } |

**functions.c:**

|  |
| --- |
| int max(int a, int b) {  return (a > b) ? a : b;  }  // Возвращает время в миллисекундах между start и end  double elapsed\_ms(struct timespec start, struct timespec end) {  return (end.tv\_sec - start.tv\_sec) \* 1000.0 +  (end.tv\_nsec - start.tv\_nsec) / 1000000.0;  }  typedef struct {  double x, y, z;  } Point;  typedef struct {  Point A, B, C;  } Triangle;  typedef struct {  Point\* points; // Массив точек с координатами (x, y, z)  int count\_points; // Кол-во всего точек  int start\_i; // Начальный индекс части массива для потока  int end\_i; // Конечный индекс массива для потока  double thread\_max\_area; // локальный максимум  } Thread\_data;  /\*  pthread\_create() всегда вызывает функцию, которая выглядит так:  void\* thread\_function(void\* arg);  \*/  double triangle\_area(Point A, Point B, Point C) {  // Векторы AB и AC  double ABx = B.x - A.x;  double ABy = B.y - A.y;  double ABz = B.z - A.z;  double ACx = C.x - A.x;  double ACy = C.y - A.y;  double ACz = C.z - A.z;  // Векторное произведение AB × AC  double cross\_x = ABy \* ACz - ABz \* ACy;  double cross\_y = ABz \* ACx - ABx \* ACz;  double cross\_z = ABx \* ACy - ABy \* ACx;  // Длина этого вектора  double cross\_len = sqrt(cross\_x \* cross\_x + cross\_y \* cross\_y + cross\_z \* cross\_z);  return 0.5 \* cross\_len; // площадь  } |

**run.sh:**

|  |
| --- |
| #!/bin/bash  # Путь к исходникам  SRC\_DIR="./src"  EXEC="./main"  THREADS=20 # кол-во потоков (можно менять)  echo Количество потоков: $THREADS  TEST\_DIR="../test"  # Замер времени начала  START=$(date +%s%3N)  # Запуск для каждого файла test\_\*.txt  for file in $TEST\_DIR/test\_\*.txt; do  echo "=============================="  echo "Файл: $file"  $EXEC $(basename $file) $THREADS  done  # Замер времени конца  END=$(date +%s%3N)  # Вывод времени выполнения в миллисекундах  TOTAL\_MS=$((END - START))  echo "Общее время для всех тестов: ${TOTAL\_MS} ms" |

Ссылка на github репозиторий:

<https://github.com/SoslanPar/os_lab_2>

**Пример работы**

**./main test\_0.txt 20**

**Метод 1 с фиксированным диапазоном (делим массив на части для индекса i):**

**Максимальная площадь = 0.87**

**Время = 1.31 ms**

**Метод 2 с динамическим распределением (поочередно потоки берут индекс i):**

**Максимальная площадь = 0.87**

**Время = 1.15 ms**

**(test\_0.txt:**

**1.0 0.0 0.0**

**0.0 1.0 0.0**

**0.0 0.0 1.0)**

**./run.sh**

**Количество потоков: 20**

**==============================**

**Файл: ../test/test\_-1.txt**

**Error: Few points for triangle**

**==============================**

**Файл: ../test/test\_0.txt**

**Метод 1 с фиксированным диапазоном (делим массив на части для индекса i):**

**Максимальная площадь = 0.87**

**время = 0.84 ms**

**Метод 2 с динамическим распределением (поочередно потоки берут индекс i):**

**Максимальная площадь = 0.87**

**время = 0.69 ms**

**==============================**

**Файл: ../test/test\_1.txt**

**Метод 1 с фиксированным диапазоном (делим массив на части для индекса i):**

**Максимальная площадь = 2064620.77**

**время = 0.74 ms**

**Метод 2 с динамическим распределением (поочередно потоки берут индекс i):**

**Максимальная площадь = 2064620.77**

**время = 0.84 ms**

**==============================**

**Файл: ../test/test\_2.txt**

**Метод 1 с фиксированным диапазоном (делим массив на части для индекса i):**

**Максимальная площадь = 2909771.72**

**время = 53.98 ms**

**Метод 2 с динамическим распределением (поочередно потоки берут индекс i):**

**Максимальная площадь = 2909771.72**

**время = 33.96 ms**

**==============================**

**Файл: ../test/test\_3.txt**

**Метод 1 с фиксированным диапазоном (делим массив на части для индекса i):**

**Максимальная площадь = 2954329.54**

**время = 293.10 ms**

**Метод 2 с динамическим распределением (поочередно потоки берут индекс i):**

**Максимальная площадь = 2954329.54**

**время = 172.94 ms**

**==============================**

**Файл: ../test/test\_4.txt**

**Метод 1 с фиксированным диапазоном (делим массив на части для индекса i):**

**Максимальная площадь = 2777523.44**

**время = 3.96 ms**

**Метод 2 с динамическим распределением (поочередно потоки берут индекс i):**

**Максимальная площадь = 2777523.44**

**время = 2.77 ms**

**==============================**

**Файл: ../test/test\_5.txt**

**Метод 1 с фиксированным диапазоном (делим массив на части для индекса i):**

**Максимальная площадь = 2648615.53**

**время = 5.57 ms**

**Метод 2 с динамическим распределением (поочередно потоки берут индекс i):**

**Максимальная площадь = 2648615.53**

**время = 3.90 ms**

**Общее время для всех тестов: 651 ms**

**Вывод**

В ходе лабораторной работы было проведено исследование эффективности многопоточного выполнения программы с использованием библиотеки POSIX Threads (pthread).

Для реализации параллельных вычислений применялись потоки, мьютексы и динамическое распределение задач между потоками.

Программа выполнялась при различном числе потоков: 1, 5, 10, 15 и 20. Для каждой конфигурации было замерено как общее время выполнения, так и время отдельных этапов обработки.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во потоков /  Время (ms) | 1 | 5 | 10 | 15 | 20 |
| Общее время | 3000 | 1200 | 800 | 700 | 600 |
| Для наибольшего теста (двумя методами) | 1200  1200 | 650  300 | 450  250 | 300  200 | 300  170 |

Результаты эксперимента показали, что увеличение количества потоков приводит к существенному снижению общего времени выполнения программы.

Наибольший прирост производительности наблюдается при переходе от одного потока к 5–10 потокам.

При дальнейшем увеличении числа потоков ускорение становится менее выраженным — это объясняется тем, что накладные расходы на создание и синхронизацию потоков начинают компенсировать выгоду от параллельного выполнения.

Работа помогла лучше понять принципы параллелизма, синхронизации и оптимизации вычислительных задач, а также на практике увидеть, как многопоточность ускоряет выполнение программ при правильной организации.