Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3 по курсу**

**«Операционные системы»**

Студент: Старцев Иван Романович

Группа: М8О-201Б-21

Вариант: 16

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

https://github.com/IvanTvardovsky/OS-labs

**Постановка задачи**

**Цель работы**

Целью является 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 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.

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

**Общие сведения о программе**

Программа компилируется из файла main.c. Также используется вспомогательные программы general.c и utils.c, а также заголовочные файлы general.h и utils.h

В программе используются следующие системные вызовы:

1. pthread\_create() – создаёт новый поток
2. pthread\_join(pthread\_t THREAD\_ID, void \*\* DATA) - ожидает завершения потока обозначенного THREAD\_ID

**Общий метод и алгоритм решения**

Разделим область нашего интегрирования и количество точек между процессами, разделение областей круга будет происходить по оси Х, на K, где К – количество потоков, равных частей. (то есть координата х случайных точек будет в пределах выделенной области для каждого потока, а координата У у всех генерироваться в одном и том же диапазоне). Так мы разделяем всю работу на К потоков. Заметим, что такого феномена, как «Race condition» у нас не возникнет, так как каждый поток будет считать только свои точки, а после завершения всех процессов мы сложим полученные ответы.

**Исходный код**

**main.c**

#include "general.h"

int main(int argc, const char\*\* argv) {

if (argc < 2) {

printf("Not enough arguments\n");

exit(EXIT\_FAILURE);

}

int countThreads = atoi(argv[1][0]);

double r;

printf("Given radius is: ");

scanf("%lf", &r);

if (r < 0) {

printf("Given radius is negative\n");

exit(EXIT\_FAILURE);

}

printf("Answer is approximately %.20lf\n", CalculateDiameter(r, countThreads, 1000000000));

return 0;

}

**general.c**

#include "utils.h"

#include "general.h"

double CalculateDiameter(double r, int countThreads, int totalPoints) {

double diameterArea;

diameterArea = r \* 2;

pthread\_t\* th = malloc(sizeof(pthread\_t) \* countThreads);

TThreadToken\* token = malloc(sizeof(TThreadToken) \* countThreads);

unsigned int\* states = malloc(sizeof(unsigned long int) \* countThreads);

if (th == NULL || token == NULL || states == NULL) {

printf("Can't allocate memory\n");

exit(EXIT\_FAILURE);

}

double start = -r;

double step = (diameterArea / (double)countThreads);

int points = (totalPoints + countThreads - 1) / countThreads;

for (int i = 0; i < countThreads; ++i) {

token[i].start = start;

token[i].step = &step;

token[i].r = &r;

token[i].points = Min(points, totalPoints - i \* points);

token[i].state = &states[i];

start += step;

}

for (int i = 0; i < countThreads; ++i) {

if (pthread\_create(&th[i], NULL, &Integral, &token[i]) != 0) {

printf("Can't create thread\n");

exit(EXIT\_FAILURE);

}

}

points = 0;

for (int i = 0; i < countThreads; ++i) {

if (pthread\_join(th[i], NULL) != 0) {

printf("Can't join threads\n");

exit(EXIT\_FAILURE);

}

points += token[i].points;

}

free(token);

free(th);

free(states);

return(diameterArea\*diameterArea \* ((double) points / (totalPoints)));

}

**utils.c**

#include "utils.h"

void\* Integral(void\* arg) {

TThreadToken token = \*((TThreadToken\*)arg);

double x, y, r;

r = \*token.r;

int \*ustate = token.state;

\*ustate = time(NULL) ^ getpid() ^ pthread\_self();

int attempts = token.points;

token.points = 0;

for (int i = 0; i < attempts; ++i) {

x = token.start + ((double)rand\_r(ustate) / (double)(RAND\_MAX)) \* (\*token.step);

y = (((double)rand\_r(ustate) / (double)(RAND\_MAX)) - 0.5) \* 2 \* r;

if (InCircle(x, y, r)) {

++token.points;

}

}

((TThreadToken\*)arg) -> points = token.points;

return arg;

}

int InCircle(double x, double y, double r) {

return x \* x + y \* y <= r \* r;

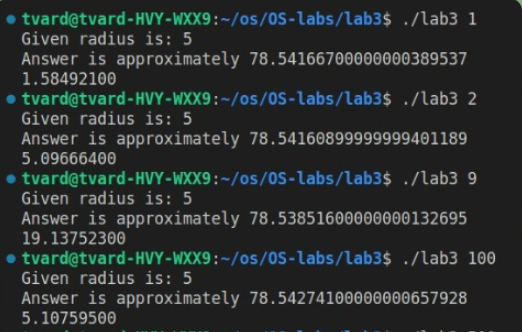
}

int Min(int a, int b) {

return a < b ? a : b;

}

**Демонстрация работы программы**



**Выводы**

За время выполнения лабораторной работы я научился управлять потоками в ОС, а также разобрался с обеспечение синхронизации между потоками. Во время выполнения работы я понял, что создать и синхронизировать много потоков может быть более накладно, чем выполнять код на одном ядре. Из-за этого при создании двух и более потоков программа сильно замедляется (по сравнению с одним потоком).

Как видно из демонстрации работы программы, самый быстрый способ выполнить поставленную задачу – это рассчитать всё одним потоком без использования распараллеливания. Это связано с издержками, которые мы несём при создании и выполнении потоков. Однако дальше видна тенденция к ускорения работы, при увеличении количества потоков, это уже говорит о том, что метод параллельных вычислений действительно даёт выигрыш в скорости, хотя и конкретно в данной задаче не покрывает расходов на создание потоков.