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: Ползикова Алина Владимировна

Группа: М8О-208Б-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/polzzzik/lab\_os

**Постановка задачи**

**Цель работы**

Изучить управление потоками, обеспечение синхронизации между потоками.

**Задание**

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

**Общие сведения о программе**

Программа компилируется из файла main.c. Также используется заголовочные файлы: unistd.h, pthread.h, stdio.h, math.h. В программе используются следующие системные вызовы:

1. pthread\_create() - создаёт новый поток, принимает параметры.
2. pthread\_join() - ждёт завершения работы потока.

**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 <stdio.h>

#include <math.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

typedef struct TThreadToken {

double\* R;

double\* step;

double start;

int Cpoints;

} ThreadToken;

void exit\_with\_msg(const char\* msg, int return\_code);

void\* integral(void\* arg);

int min(int a, int b);

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

int Total\_points = 10000000;

double R, Total\_Aria\_Size;

int CountThreads;

if (argc < 2) {

exit\_with\_msg("missing arguments", -1);

}

for (int i = 0; argv[1][i] > 0; ++i) {

if (argv[1][i] >= '0' && argv[1][i] <= '9') {

CountThreads = CountThreads \* 10 + argv[1][i] - '0';

}

}

printf("enter Radius: ");

scanf("%lf", &R);

Total\_Aria\_Size = R \* 2;

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

ThreadToken\* token = malloc(sizeof(ThreadToken) \* CountThreads);

double start = -R;

double step = (Total\_Aria\_Size / (double ) CountThreads);

int Cpoints = (Total\_points + CountThreads - 1)/CountThreads;

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

token[i].start = start;

token[i].step = &step;

token[i].R = &R;

token[i].Cpoints = min(Cpoints, Total\_points - i\*Cpoints);

start += step;

}

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

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

exit\_with\_msg("cannot create thread", 2);

}

}

Cpoints = 0;

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

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

exit\_with\_msg("cannot join threads", 3);

}

Cpoints += token[i].Cpoints;

}

printf("Exact answer is : %.20lf\n", acos(-1)\*R\*R);

printf("Answer is approximately: %.20lf\n",

Total\_Aria\_Size\*Total\_Aria\_Size\*((double) Cpoints / (Total\_points)));

free(token);

free(th);

return 0;

}

int min(int a, int b) {

if (a < b) return a;

return b;

}

void exit\_with\_msg(const char\* msg, int return\_code) {

printf("%s\n", msg);

exit(return\_code);

}

int in\_circle(double x, double y, double R) {

return (x\*x + y\*y <= R\*R);

}

void\* integral(void\* arg) {

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

double x, y, R;

R = \*token.R;

int attempts = token.Cpoints;

token.Cpoints = 0;

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

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

y = (((double )rand()/(double )(RAND\_MAX)) - 0.5) \* 2\*R;

if (in\_circle(x,y, R)) {

token.Cpoints++;

}

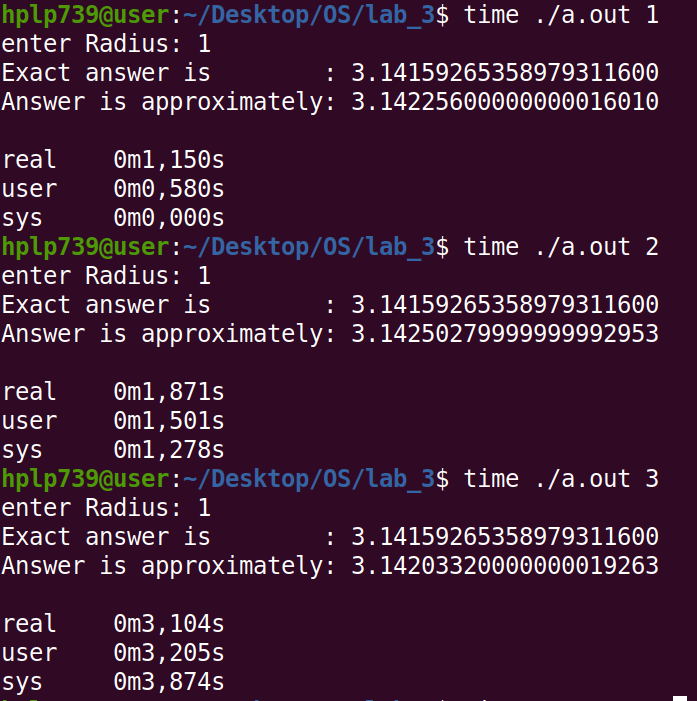
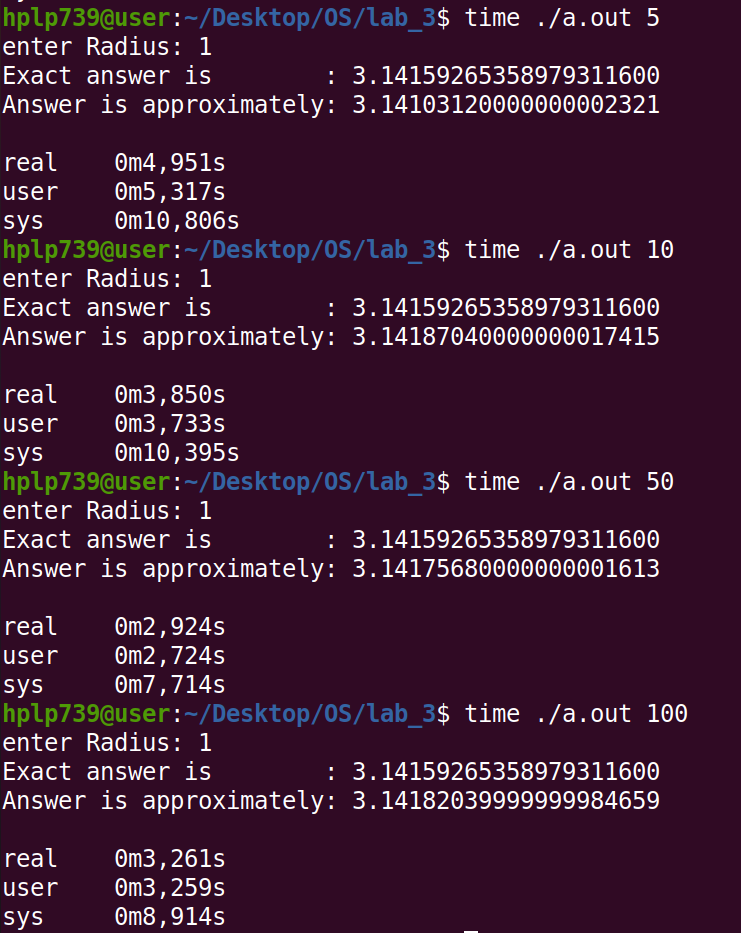
}

((ThreadToken\*) arg)->Cpoints = token.Cpoints;

return arg;

}

**Демонстрация работы программы**



**Выводы**

В результате данной лабораторной работы, научилась работать с потоками (Threads), составила и отладила программу вычисления площади круга методом Монте-Карла. Во время выполнения работы я поняла, 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). Если задачу в программировании можно выполнять по частям, независимо друг от друга, то многопоточный подход ускорит работу программы (если синхронизации не обойдётся дороже!) в несколько раз. Так в играх вычисления (например, перемножение матриц) производятся на большом количестве графических ядер параллельно.

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.