Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работ №3 по курсу**

**«Операционные системы»**

**Управление потоками в ОС**

Студент: Базаргармаев Нима Дондокович

Группа: М80 – 301Б-18

Вариант: 13

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2020

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Основные файлы программы
5. Тестирование
6. Демонстрация работы программы
7. Вывод

**Постановка задачи**.

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке использовать стандартные средства создания потоков операционной системы (Windows/Unix). При создании необходимо предусмотреть ключи, которые позволяли бы задать максимальное количество потоков, используемое программой. При возможности необходимо использовать максимальное количество возможных потоков. Ограничение потоков может быть задано или ключом запуска вашей программы, или алгоритмом.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

**Задание согласно 12 варианту:**

Два человека играют в кости. Правила игры следующие: каждый игрок делает бросок 2-ух костей K раз; побеждает тот, кто выбросил суммарно большее количество очков. Задача программы экспериментально определить шансы на победу каждого из игроков. На вход программе подается K, какой сейчас тур, сколько очков суммарно у каждого из игроков и количество экспериментов, которые должна произвести программа.

**Общие сведения о программе**

Программа компилируется из одного файла 3.c. В 3.c используются заголовочные файлы pthread.h, stdio.h, stdlib.h и time.h. В программе используются следующие системные вызовы:

1. **pthread\_create()** – предназначен для создания новой нити исполнения внутри текущего процесса. При удачном завершении функция возвращает значение 0 и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр thread. В случае ошибки возвращается положительное значение.
2. **pthread\_join()** – Функция *pthread\_join* блокирует работу вызвавшей ее нити исполнения до завершения thread'а с идентификатором thread. Аргумент функции, после разблокирования в указатель, расположенный по адресу status\_addr, заносится адрес, который вернул завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции *pthread\_exit()*. Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение NULL.
3. **pthread\_mutex\_lock()/pthread\_mutex\_unlock()** – системный вызов захвата и освобождения мьютекса, или приостановления выполнения вызывающего потока, пока другой поток «владелец» мьютекса, не освободит его вызовом **pthread\_mutex\_unlock()**.

**Общий метод и алгоритм решения**.

Заводится специальная структура для входных данных, в которой имеется массив целых чисел размера количества проводимых экспериментов. Массив необходим, чтобы затем посчитать вероятности. Далее данные считываются из стандартного потока ввода. Затем заводится массив потоков размера максимального количества потоков, вводимого в момент считывания входных данных. После этого в цикле for производится создание новых потоков, в которых будет выполняться функция проведения эксперимента. В функции проведения эксперимента вызывается pthread\_mutex\_lock(), чтобы обеспечить синхронизацию доступа к данным, затем проводятся необходимые вычисления. После завершения цикла результаты выводятся в стандартный поток вывода.

**Основные файлы программы**.

**Файл 3.c**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

#define SUCCESS 1

#define FAIL 11

int max\_thread\_count = 0;

pthread\_mutex\_t mutex;

typedef struct thread\_data {

int K;

int current\_tour;

int first\_player\_score;

int second\_player\_score;

int experiments\_count;

int\* first\_won;

int current\_experiment;

} thread\_data;

void CheckBounds(int to\_check, int lower\_bound) {

if (to\_check < lower\_bound) {

printf("The value must be greater or equal to %d\n", lower\_bound);

exit(-1);

}

}

thread\_data ScanData() {

thread\_data data;

printf("Enter thread count: ");

scanf("%d", &max\_thread\_count);

printf("\n");

CheckBounds(max\_thread\_count, 1);

printf("Enter the number of throws: ");

scanf("%d", &data.K);

printf("\n");

CheckBounds(data.K, 1);

printf("Enter the tour number: ");

scanf("%d", &data.current\_tour);

printf("\n");

CheckBounds(data.current\_tour, 1);

printf("Enter first and second player's score: ");

scanf("%d", &data.first\_player\_score);

scanf("%d", &data.second\_player\_score);

printf("\n");

CheckBounds(data.first\_player\_score, 2);

CheckBounds(data.second\_player\_score, 2);

printf("Enter count of experiments to conduct: ");

scanf("%d", &data.experiments\_count);

printf("\n");

CheckBounds(data.experiments\_count, 1);

int arr[data.experiments\_count];

data.first\_won = arr;

data.current\_experiment = 0;

return data;

}

void\* conduct\_experiment(void\* args) {

thread\_data\* data = (thread\_data\*) args;

pthread\_mutex\_lock(&mutex);

int first = data->first\_player\_score;

int second = data->second\_player\_score;

for (int i = data->current\_tour; i <= data->K; ++i) {

first += rand() % 12;

second += rand() % 12;

}

int is\_won = (first > second) ? 1 : 0;

data->first\_won[data->current\_experiment++] = is\_won;

pthread\_mutex\_unlock(&mutex);

return (void\*) SUCCESS;

}

int main() {

thread\_data data = ScanData();

pthread\_t threads[max\_thread\_count];

int status\_addrs[max\_thread\_count];

int current\_thread = 0;

int current\_thread\_count = max\_thread\_count;

for (int i = 0; i < data.experiments\_count; ++i) {

if (current\_thread\_count > 0) {

if (pthread\_create(&threads[current\_thread], NULL, conduct\_experiment, (void\*) &data) == 0) {

current\_thread\_count--;

pthread\_join(threads[current\_thread], (void \*\*) &status\_addrs[current\_thread]);

if (status\_addrs[current\_thread] != SUCCESS) {

printf("ERROR\n");

exit(-1);

}

}

current\_thread++;

} else {

current\_thread = 0;

current\_thread\_count = max\_thread\_count;

}

}

int first\_won\_count = 0;

for (int i = 0; i < data.experiments\_count; ++i) {

if (data.first\_won[i] == 1) {

first\_won\_count++;

}

}

double first\_chance = (1.0 \* first\_won\_count) / (1.0 \* data.experiments\_count);

double second\_chance = 1.0 - first\_chance;

printf("First player's chances to win: %f\n", first\_chance);

printf("Second player's chances to win: %f\n", second\_chance);

return 0;

}

**Тестирование.**

См. Следующий пункт.

**Демонстрация работы программы.**

user@DESKTOP-4RFVSK2:~/os/3\_easy$ ./a.out

Enter thread count: 4

Enter the number of throws: 10

Enter the tour number: 1

Enter first and second player's score: 0 0

The value must be greater or equal to 2

user@DESKTOP-4RFVSK2:~/os/3\_easy$ ./a.out

Enter thread count: 4

Enter the number of throws: 10

Enter the tour number: 1

Enter first and second player's score: 4 6

Enter count of experiments to conduct: 8

First player's chances to win: 0.000000

Second player's chances to win: 1.000000

user@DESKTOP-4RFVSK2:~/os/3\_easy$ ./a.out

Enter thread count: 4

Enter the number of throws: 6

Enter the tour number: 3

Enter first and second player's score: 6

13

Enter count of experiments to conduct: 18

First player's chances to win: 0.000000

Second player's chances to win: 1.000000

user@DESKTOP-4RFVSK2:~/os/3\_easy$ ./a.out

Enter thread count: 4

Enter the number of throws: 6

Enter the tour number: 1

Enter first and second player's score: 2 2

Enter count of experiments to conduct: 50

First player's chances to win: 0.020000

Second player's chances to win: 0.980000

**Вывод.**

Управление потоками в OC очень непросто. Правильное создание потоков может сильно ускорить программу, но неправильное использование мьютекса и ожидание завершения thread’a могут только замедлить программу, нежели чем реализовывать последовательный алгоритм выполнения программы. Помимо этого стоит отметить, что многопоточность стоит реализовывать при трудоемких операциях или когда выполнение процесса имеет гораздо большие затраты, чем затраты на создание потока. Также можно столкнуться с проблемой взаимных блокировок, при реализации сложного взаимодействия потоков друг с другом. Но в случае почти идеальной реализации работы с потоками мы можем получить увеличение скорости работы программы.