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»**

Группа: М8О-212БВ-24

Студент: Авезов Р.А.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 03.10.25

Москва, 2025

**Постановка задачи**

**Вариант 7.**

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

**Общий метод и алгоритм решения**

Использованные системные вызовы и API:

pthread\_create() — создаёт новый поток выполнения. Использовался для запуска рабочих потоков, каждый из которых обрабатывает часть экспериментов.

pthread\_join() — ожидает завершения указанного потока. Обеспечивает синхронизацию: основной поток дожидается окончания всех рабочих потоков перед выводом результатов.

pthread\_mutex\_lock() / pthread\_mutex\_unlock() — обеспечивают взаимное исключение при доступе к разделяемым переменным (счётчикам побед). Это предотвращает состояние гонки (race condition) при одновременной записи из нескольких потоков.

clock\_gettime(CLOCK\_MONOTONIC, ...) — высокоточный системный вызов для измерения времени выполнения. Использовался для замера времени последовательной и параллельной версий.

rand\_r() — потокобезопасная функция генерации псевдослучайных чисел. Каждый поток использует собственный seed, что исключает конкуренцию за общий генератор.

**Код программы**

**dice\_sim.c**

#define \_POSIX\_C\_SOURCE 200809L

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <pthread.h>

#include <unistd.h>

long long wins\_p1 = 0;

long long wins\_p2 = 0;

long long draws = 0;

pthread\_mutex\_t result\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

static inline int roll\_die(unsigned int \*seed) {

return (rand\_r(seed) % 6) + 1;

}

void run\_experiment(int K, unsigned int \*seed, long long \*p1, long long \*p2, long long \*dr) {

int s1 = 0, s2 = 0;

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

s1 += roll\_die(seed) + roll\_die(seed);

s2 += roll\_die(seed) + roll\_die(seed);

}

if (s1 > s2) (\*p1)++;

else if (s2 > s1) (\*p2)++;

else (\*dr)++;

}

void run\_sequential(int K, long long N, long long \*p1, long long \*p2, long long \*dr) {

unsigned int seed = (unsigned int)(time(NULL) ^ (unsigned int)getpid());

\*p1 = \*p2 = \*dr = 0;

for (long long i = 0; i < N; i++) {

run\_experiment(K, &seed, p1, p2, dr);

}

}

typedef struct {

long long exp\_count;

int K;

unsigned int seed;

} thread\_arg\_t;

void\* worker(void\* arg) {

thread\_arg\_t\* a = (thread\_arg\_t\*)arg;

long long p1 = 0, p2 = 0, dr = 0;

unsigned int seed = a->seed;

for (long long i = 0; i < a->exp\_count; i++) {

run\_experiment(a->K, &seed, &p1, &p2, &dr);

}

pthread\_mutex\_lock(&result\_mutex);

wins\_p1 += p1;

wins\_p2 += p2;

draws += dr;

pthread\_mutex\_unlock(&result\_mutex);

free(a);

return NULL;

}

void run\_parallel(int K, long long N, int T) {

wins\_p1 = wins\_p2 = draws = 0;

unsigned int base\_seed = (unsigned int)(time(NULL) ^ (unsigned int)getpid());

long long remaining = N;

unsigned int current\_seed = base\_seed;

while (remaining > 0) {

long long threads\_to\_launch = (remaining > T) ? T : remaining;

pthread\_t\* threads = calloc(threads\_to\_launch, sizeof(pthread\_t));

if (!threads) {

perror("calloc");

exit(EXIT\_FAILURE);

}

long long exp\_per = remaining / threads\_to\_launch;

long long extra = remaining % threads\_to\_launch;

for (long long i = 0; i < threads\_to\_launch; i++) {

thread\_arg\_t\* args = malloc(sizeof(thread\_arg\_t));

if (!args) {

perror("malloc");

exit(EXIT\_FAILURE);

}

args->K = K;

args->exp\_count = exp\_per + (i < extra ? 1 : 0);

args->seed = current\_seed++;

if (pthread\_create(&threads[i], NULL, worker, args) != 0) {

fprintf(stderr, "pthread\_create failed\n");

exit(EXIT\_FAILURE);

}

}

for (long long i = 0; i < threads\_to\_launch; i++) {

pthread\_join(threads[i], NULL);

}

free(threads);

remaining -= threads\_to\_launch \* exp\_per + extra;

}

}

int main(int argc, char\* argv[]) {

if (argc != 3) {

fprintf(stderr, "Usage: %s <K> <N>\n", argv[0]);

fprintf(stderr, " K - number of throws per player\n");

fprintf(stderr, " N - total number of experiments\n");

return EXIT\_FAILURE;

}

int K = atoi(argv[1]);

long long N = atoll(argv[2]);

if (K <= 0 || N <= 0) {

fprintf(stderr, "Error: K and N must be positive integers.\n");

return EXIT\_FAILURE;

}

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC, &start);

long long p1, p2, dr;

run\_sequential(K, N, &p1, &p2, &dr);

clock\_gettime(CLOCK\_MONOTONIC, &end);

double t\_seq\_sec = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9;

long long T\_seq\_ms = (long long)(t\_seq\_sec \* 1000 + 0.5);

int thread\_counts[] = {1, 2, 4, 6, 8, 16, 128, 1024};

int num\_configs = sizeof(thread\_counts) / sizeof(thread\_counts[0]);

printf("\n");

printf("Число потоков | Время исполнения (мс) | Ускорение | Эффективность\n");

printf("--------------|------------------------|-----------|----------------\n");

double speedup = (double)T\_seq\_ms / T\_seq\_ms; // = 1.0

double efficiency = speedup / 1.0;

printf("%-13d | %-23lld | %-9.2f | %-14.4f\n", 1, T\_seq\_ms, speedup, efficiency);

for (int i = 1; i < num\_configs; i++) {

int T = thread\_counts[i];

clock\_gettime(CLOCK\_MONOTONIC, &start);

run\_parallel(K, N, T);

clock\_gettime(CLOCK\_MONOTONIC, &end);

double t\_par\_sec = (end.tv\_sec - start.tv\_sec) + (end.tv\_nsec - start.tv\_nsec) / 1e9;

long long T\_par\_ms = (long long)(t\_par\_sec \* 1000 + 0.5);

speedup = (double)T\_seq\_ms / T\_par\_ms;

efficiency = speedup / T;

printf("%-13d | %-23lld | %-9.2f | %-14.4f\n", T, T\_par\_ms, speedup, efficiency);

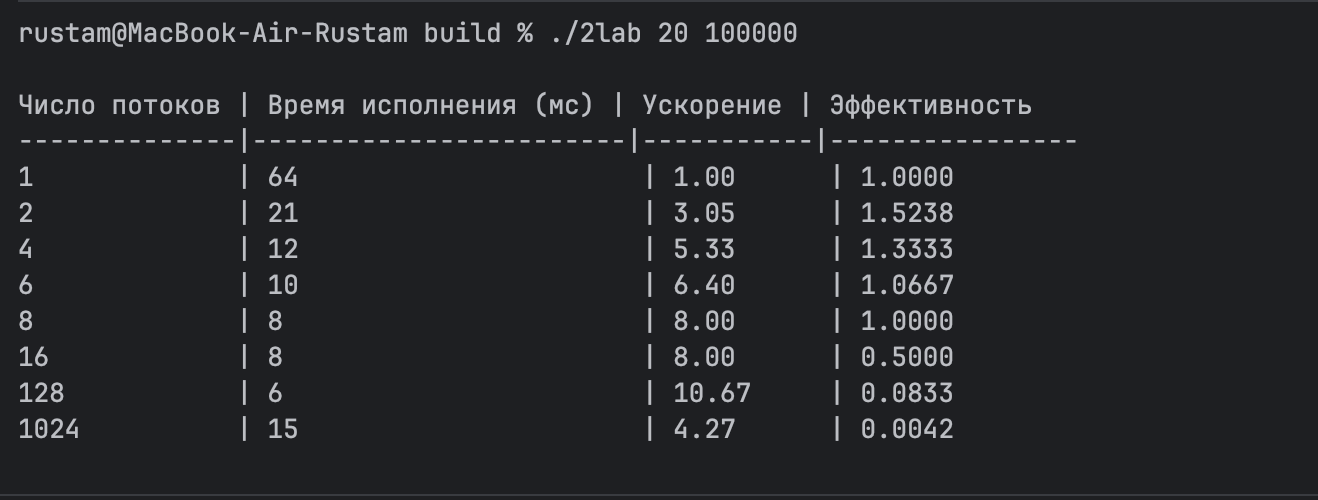
}

printf("\n");

return EXIT\_SUCCESS;

}

**Протокол работы программы**



**Вывод**

В ходе выполнения лабораторной работы была успешно реализована многопоточная программа, демонстрирующая ускорение вычислений при параллельной обработке независимых задач. Эксперименты показали, что максимальная эффективность достигается при числе потоков, близком к количеству логических ядер процессора (8 на используемом MacBook Air), а дальнейшее увеличение числа потоков приводит к снижению производительности из-за накладных расходов на переключение контекста и синхронизацию.