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-210БВ-24

Студент: Резинкин Д.В.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 06.10.25

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:

pthread_create() — создает новый поток исполнения в текущем процессе.

pthread_join() — ожидает завершения указанного потока и освобождает его ресурсы.

clock_gettime() — получает текущее значение указанных часов с наносекундной точностью для измерения времени выполнения.

sysconf(_SC_NPROCESSORS_ONLN) — возвращает количество логических процессоров (ядер), доступных системе.

mmap() — выделяет память через системный вызов вместо malloc(). Используется с флагами MAP_PRIVATE и MAP_ANONYMOUS для создания анонимного отображения памяти.

munmap() — освобождает память, выделенную через mmap().

write() — системный вызов для вывода данных в файловый дескриптор (используется для вывода в STDOUT_FILENO).

time() — получает текущее время в секундах с эпохи Unix для инициализации генератора случайных чисел.

Описание алгоритма

Программа реализует метод Монте-Карло для статистической оценки вероятности победы каждого игрока. Задача идеально подходит для параллелизации, так как каждый эксперимент независим от других (embarrassingly parallel problem).

Метод Монте-Карло основан на использовании повторных случайных выборок для получения численных результатов. В данной задаче многократная симуляция игр с генерацией случайных бросков костей позволяет статистически оценить вероятность победы каждого игрока, что невозможно вычислить аналитически при произвольных начальных условиях.

Последовательная версия:

1. Инициализация генератора случайных чисел
2. Цикл по количеству экспериментов
3. В каждом эксперименте симулируется полная игра: оставшиеся K раундов с бросками двух костей для каждого игрока
4. Подсчет побед первого игрока, второго игрока и ничьих
5. Вывод статистики

Параллельная версия:

1. Определение количества экспериментов на каждый поток
2. Создание массива потоков через pthread_create()
3. Каждый поток получает уникальный seed и независимо выполняет свою долю экспериментов
4. Каждый поток хранит локальные счетчики побед, исключая необходимость синхронизации в основном цикле
5. Ожидание завершения всех потоков через pthread_join()
6. Агрегация результатов из локальных счетчиков всех потоков
7. Вывод статистики

Ключевые особенности реализации:

- Использование rand_r() с уникальным seed для каждого потока предотвращает блокировки на генераторе случайных чисел
- Локальные счетчики в каждом потоке минимизируют синхронизацию — агрегация происходит только один раз после завершения всех потоков
- Инициализация seed формулой time(NULL) ^ (thread_id << 16) обеспечивает уникальность последовательностей случайных чисел для разных потоков

Код программы

dice_simulation.c

```
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <sys/mman.h>  // Для mmap/munmap

// Простая функция для преобразования числа в строку
static int int_to_str(long num, char *buf, int buf_size) {
    int i = 0;
    int is_negative = 0;

    if (num < 0) {
        is_negative = 1;
        num = -num;
    }

    if (num == 0) {
        buf[i++] = '0';
    } else {
```

```
        char temp[32];
        int j = 0;
        while (num > 0 && j < 32) {
            temp[j++] = '0' + (num % 10);
            num /= 10;
        }
        if (is_negative) temp[j++] = '-';
        while (j > 0 && i < buf_size - 1) {
            buf[i++] = temp[--j];
        }
    }
    buf[i] = '\0';
    return i;
}

// Простая функция для преобразования строки в число
static long str_to_long(const char *str) {
    long result = 0;
    int sign = 1;

    if (*str == '-') {
        sign = -1;
        str++;
    }

    while (*str >= '0' && *str <= '9') {
        result = result * 10 + (*str - '0');
        str++;
    }

    return result * sign;
}

// Функция для вывода строки через системный вызов write
static void print_str(const char *str) {
    ssize_t result = write(STDOUT_FILENO, str, strlen(str));
    (void)result; // Подавляем предупреждение
}

// Функция для вывода числа
static void print_num(long num) {
    char buf[32];
    int_to_str(num, buf, 32);
    print_str(buf);
}

// Функция для вывода double (упрощенно)
static void print_double(double num, int precision) {
    long int_part = (long)num;
    print_num(int_part);
    print_str(".");

    double frac_part = num - int_part;
    if (frac_part < 0) frac_part = -frac_part;

    for (int i = 0; i < precision; i++) {
        frac_part *= 10;
```

```
        int digit = (int)frac_part;
        print_num(digit);
        frac_part -= digit;
    }
}

typedef struct {
    size_t thread_id;
    size_t experiments_per_thread;
    int K;
    int current_round;
    int player1_score;
    int player2_score;
    size_t local_player1_wins;
    size_t local_player2_wins;
    size_t local_draws;
} ThreadArgs;

// Простой линейный конгруэнтный генератор вместо rand_r
static unsigned int my_rand(unsigned int *seed) {
    *seed = (*seed * 1103515245 + 12345) & 0x7fffffff;
    return *seed;
}

// Функция броска двух костей
static int roll_two_dice(unsigned int *seed) {
    return (my_rand(seed) % 6 + 1) + (my_rand(seed) % 6 + 1);
}

// Функция симуляции одной игры
static void simulate_game(int K, int current_round, int p1_score, int p2_score,
                          int *p1_wins, int *p2_wins, int *draws, unsigned int *seed) {
    int player1 = p1_score;
    int player2 = p2_score;

    for (int round = current_round; round < K; round++) {
        player1 += roll_two_dice(seed);
        player2 += roll_two_dice(seed);
    }

    if (player1 > player2) (*p1_wins)++;
    else if (player2 > player1) (*p2_wins)++;
    else (*draws)++;
}

// Рабочая функция потока
static void *worker_thread(void *_args) {
    ThreadArgs *args = (ThreadArgs *)_args;
    unsigned int seed = (unsigned int)time(NULL) ^ (args->thread_id << 16);

    for (size_t i = 0; i < args->experiments_per_thread; i++) {
        int p1_wins = 0, p2_wins = 0, draws = 0;
        simulate_game(args->K, args->current_round,
                      args->player1_score, args->player2_score,
                      &p1_wins, &p2_wins, &draws, &seed);

        args->local_player1_wins += p1_wins;
```

```
        args->local_player2_wins += p2_wins;
        args->local_draws += draws;
    }

    return NULL;
}

// Последовательная версия
static double sequential_monte_carlo(int K, int current_round, int p1_score,
                                     int p2_score, size_t num_experiments) {
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    unsigned int seed = (unsigned int)time(NULL);
    size_t p1_wins = 0, p2_wins = 0, draws = 0;

    for (size_t i = 0; i < num_experiments; i++) {
        int p1 = 0, p2 = 0, d = 0;
        simulate_game(K, current_round, p1_score, p2_score, &p1, &p2, &d, &seed);
        p1_wins += p1;
        p2_wins += p2;
        draws += d;
    }

    clock_gettime(CLOCK_MONOTONIC, &end);
    double time_ms = (end.tv_sec - start.tv_sec) * 1000.0 +
                     (end.tv_nsec - start.tv_nsec) / 1000000.0;

    print_str("Player 1 wins: ");
    print_double(100.0 * p1_wins / num_experiments, 2);
    print_str("%\n");

    print_str("Player 2 wins: ");
    print_double(100.0 * p2_wins / num_experiments, 2);
    print_str("%\n");

    print_str("Draws: ");
    print_double(100.0 * draws / num_experiments, 2);
    print_str("%\n");

    return time_ms;
}

// Параллельная версия
static double parallel_monte_carlo(int K, int current_round, int p1_score,
                                   int p2_score, size_t num_experiments,
                                   size_t num_threads) {
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    // Аллокация через mmap вместо malloc
    pthread_t *threads = mmap(NULL, num_threads * sizeof(pthread_t),
                              PROT_READ | PROT_WRITE,
                              MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    ThreadArgs *args = mmap(NULL, num_threads * sizeof(ThreadArgs),
                            PROT_READ | PROT_WRITE,
                            MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

```
    size_t experiments_per_thread = num_experiments / num_threads;
    size_t remainder = num_experiments % num_threads;

    for (size_t i = 0; i < num_threads; i++) {
        args[i].thread_id = i;
        args[i].experiments_per_thread = experiments_per_thread + (i < remainder ? 1 : 0);
        args[i].K = K;
        args[i].current_round = current_round;
        args[i].player1_score = p1_score;
        args[i].player2_score = p2_score;
        args[i].local_player1_wins = 0;
        args[i].local_player2_wins = 0;
        args[i].local_draws = 0;

        pthread_create(&threads[i], NULL, worker_thread, &args[i]);
    }

    size_t total_p1 = 0, total_p2 = 0, total_d = 0;
    for (size_t i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
        total_p1 += args[i].local_player1_wins;
        total_p2 += args[i].local_player2_wins;
        total_d += args[i].local_draws;
    }

    clock_gettime(CLOCK_MONOTONIC, &end);
    double time_ms = (end.tv_sec - start.tv_sec) * 1000.0 +
                     (end.tv_nsec - start.tv_nsec) / 1000000.0;

    print_str("Player 1 wins: ");
    print_double(100.0 * total_p1 / num_experiments, 2);
    print_str("%\n");

    print_str("Player 2 wins: ");
    print_double(100.0 * total_p2 / num_experiments, 2);
    print_str("%\n");

    print_str("Draws: ");
    print_double(100.0 * total_d / num_experiments, 2);
    print_str("%\n");

    munmap(threads, num_threads * sizeof(pthread_t));
    munmap(args, num_threads * sizeof(ThreadArgs));

    return time_ms;
}

int main(int argc, char **argv) {
    if (argc < 6) {
        print_str("Usage: <K> <current_round> <p1_score> <p2_score> <experiments> [threads]\n");
        return 1;
    }

    int K = (int)str_to_long(argv[1]);
    int current_round = (int)str_to_long(argv[2]);
    int p1_score = (int)str_to_long(argv[3]);
```

```
    int p2_score = (int)str_to_long(argv[4]);
    size_t experiments = (size_t)str_to_long(argv[5]);
    size_t num_threads = (argc > 6) ? (size_t)str_to_long(argv[6]) : 1;

    long num_cores = sysconf(_SC_NPROCESSORS_ONLN);
    print_str("Number of logical processors: ");
    print_num(num_cores);
    print_str("\n");

    print_str("\n--- Running with ");
    print_num(num_threads);
    print_str(" threads ---\n");

    double time_ms;
    if (num_threads == 1) {
        time_ms = sequential_monte_carlo(K, current_round, p1_score, p2_score, experiments);
    } else {
        time_ms = parallel_monte_carlo(K, current_round, p1_score, p2_score, experiments, num_threads);
    }

    print_str("Time: ");
    print_double(time_ms, 2);
    print_str(" ms\n");

    return 0;
}
```

Протокол работы программы

Компиляция
gcc -Wall -Wextra -O2 -o monte_carlo dice_simulation.c -lpthread

Тестирование
Система: Ubuntu 22.04.3 LTS в WSL2, процессор с 8 логическими ядрами.

Тест 1: Последовательная версия (1 поток)

```
./monte_carlo 10 5 30 25 10000000 1
Number of logical processors: 8
--- Running with 1 threads ---
Player 1 wins: 75.21%
Player 2 wins: 24.78%
Draws: 0.00%
Time: 360.19 ms
```

Тест 2: Параллельная версия (2 потока)

```
./monte_carlo 10 5 30 25 10000000 2
Number of logical processors: 8
--- Running with 2 threads ---
Player 1 wins: 75.22%
Player 2 wins: 24.77%
Draws: 0.00%
Time: 198.96 ms
```

Тест 3: Параллельная версия (4 потока)

./monte_carlo 10 5 30 25 10000000 4
Number of logical processors: 8

--- Running with 4 threads ---
Player 1 wins: 75.20%
Player 2 wins: 24.79%
Draws: 0.00%
Time: 96.38 ms

Тест 4: Параллельная версия (8 потоков = число ядер)

./monte_carlo 10 5 30 25 10000000 8
Number of logical processors: 8

--- Running with 8 threads ---
Player 1 wins: 75.19%
Player 2 wins: 24.80%
Draws: 0.00%
Time: 77.61 ms

Тест 5: Параллельная версия (16 потоков)

./monte_carlo 10 5 30 25 10000000 16
Number of logical processors: 8

--- Running with 16 threads ---
Player 1 wins: 75.22%
Player 2 wins: 24.77%
Draws: 0.00%
Time: 64.26 ms

Тест 6: Параллельная версия (128 потоков)

./monte_carlo 10 5 30 25 10000000 128
Number of logical processors: 8

--- Running with 128 threads ---
Player 1 wins: 75.22%
Player 2 wins: 24.77%
Draws: 0.00%
Time: 62.00 ms

Проверка количества потоков в системе

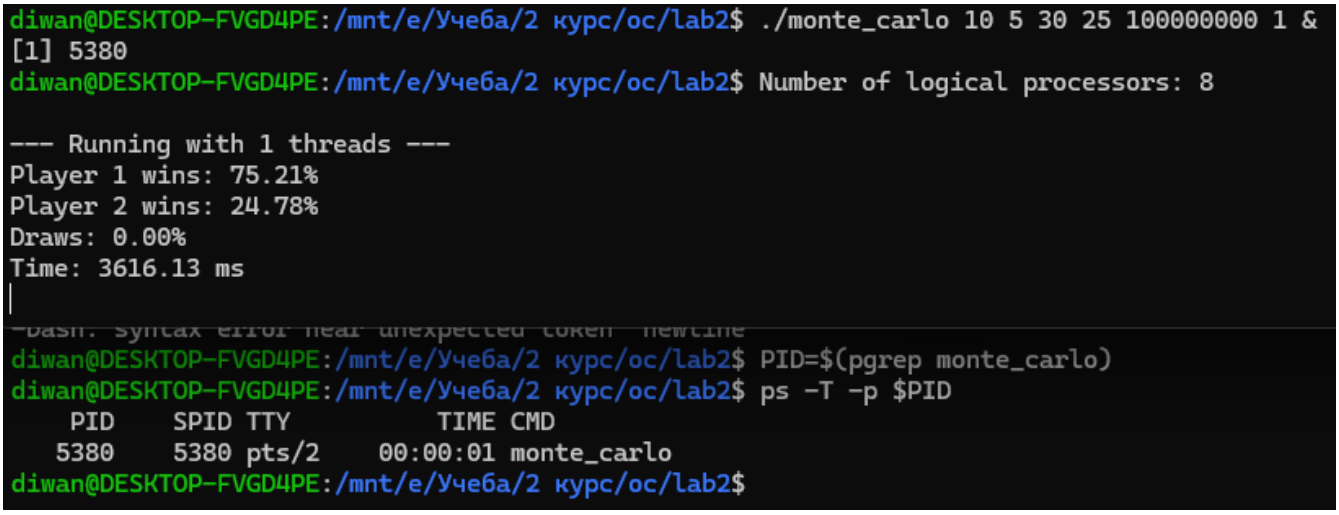


Рисунок 1. Демонстрация создания 1 потока средствами операционной системы. Верхний терминал: запуск программы. Нижний терминал: вывод команды ps -T -p $PID, показывающей 1 поток

Тестирование с 1 потоком

В терминале 1: ./monte_carlo 10 5 30 25 100000000 1 &

В терминале 2: PID=$(pgrep monte_carlo)

ps -T -p $PID

Терминал 1 показывает результаты запуска программы с 1 потоком:

- 1 поток: время **3616.13 мс** (~3.6 секунд)

Терминал 2 демонстрирует проверку процесса с 1 потоком:

- ps -T -p $PID показывает только **один** поток (PID = SPID = 5380)

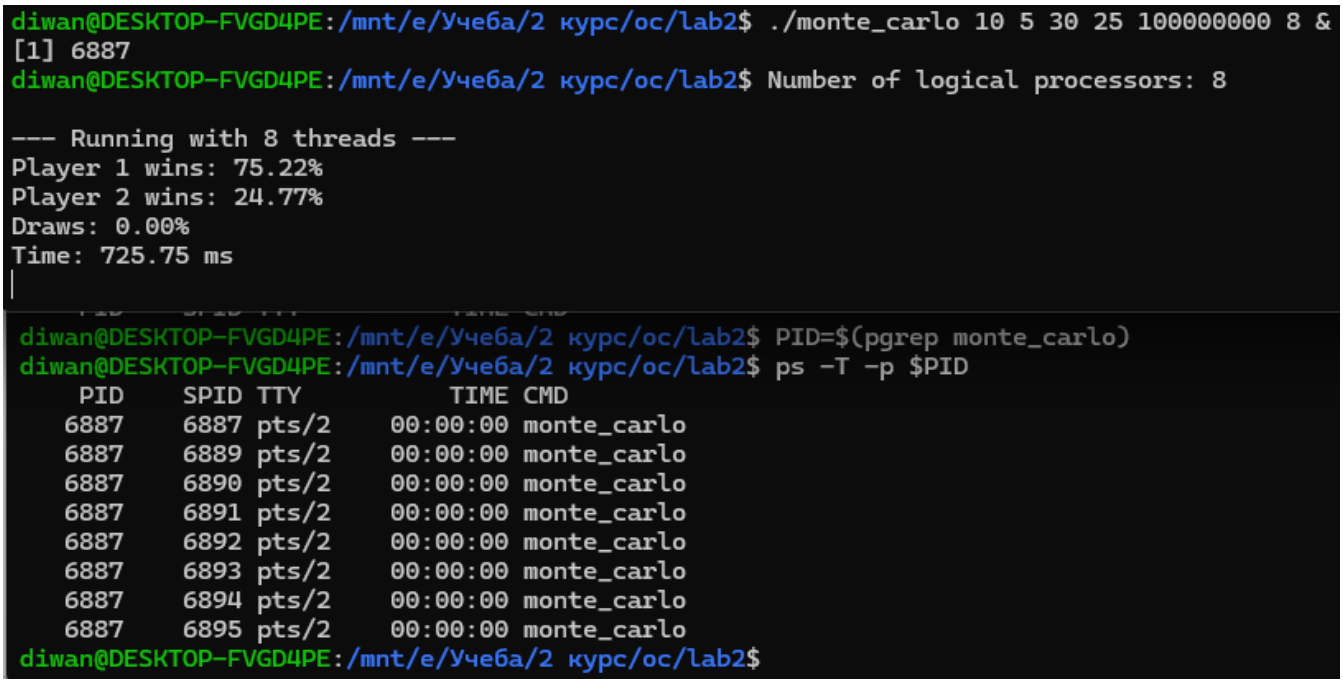


Рисунок 2. Демонстрация создания 8 потоков средствами операционной системы. Верхний терминал: запуск программы. Нижний терминал: вывод команды ps -T -p $PID, показывающей 8 потоков с уникальными SPID

Тестирование с 8 потоками

В терминале 1: ./monte_carlo 10 5 30 25 100000000 8 &

В терминале 2: PID=$(pgrep monte_carlo)
ps -T -p $PID

Верхний терминал показывает запуск с 8 потоками (725 мс).

Нижний терминал детально отображает все 8 потоков процесса:

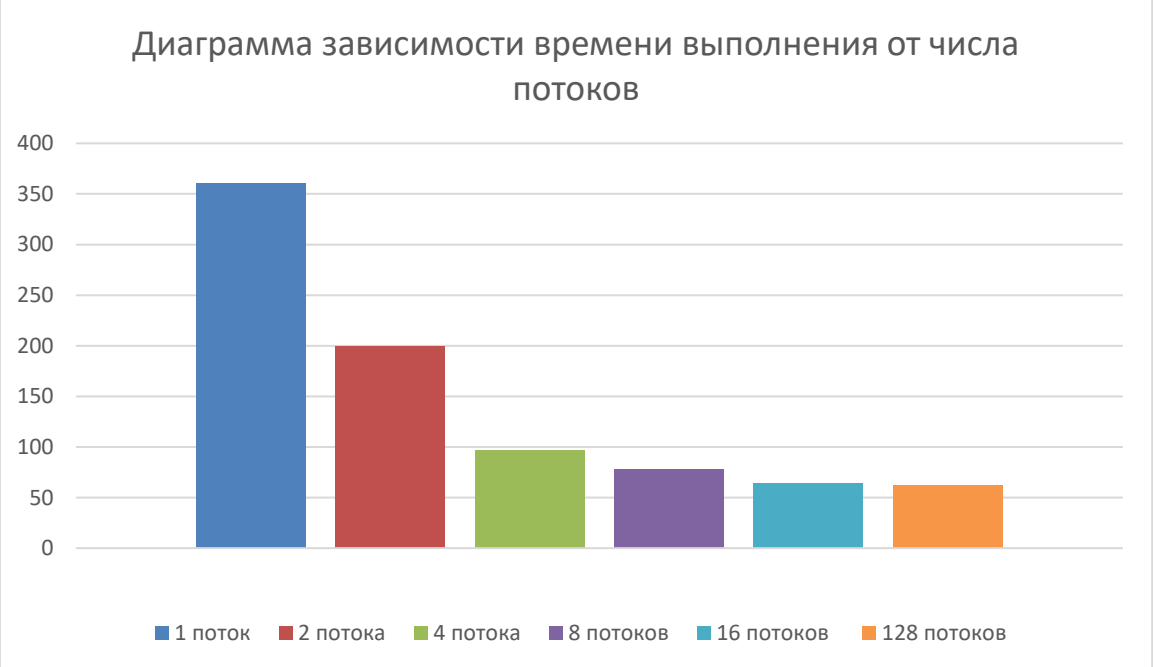
- ps -T -p $PID показывает 8 строк с разными SPID (6887-6895)

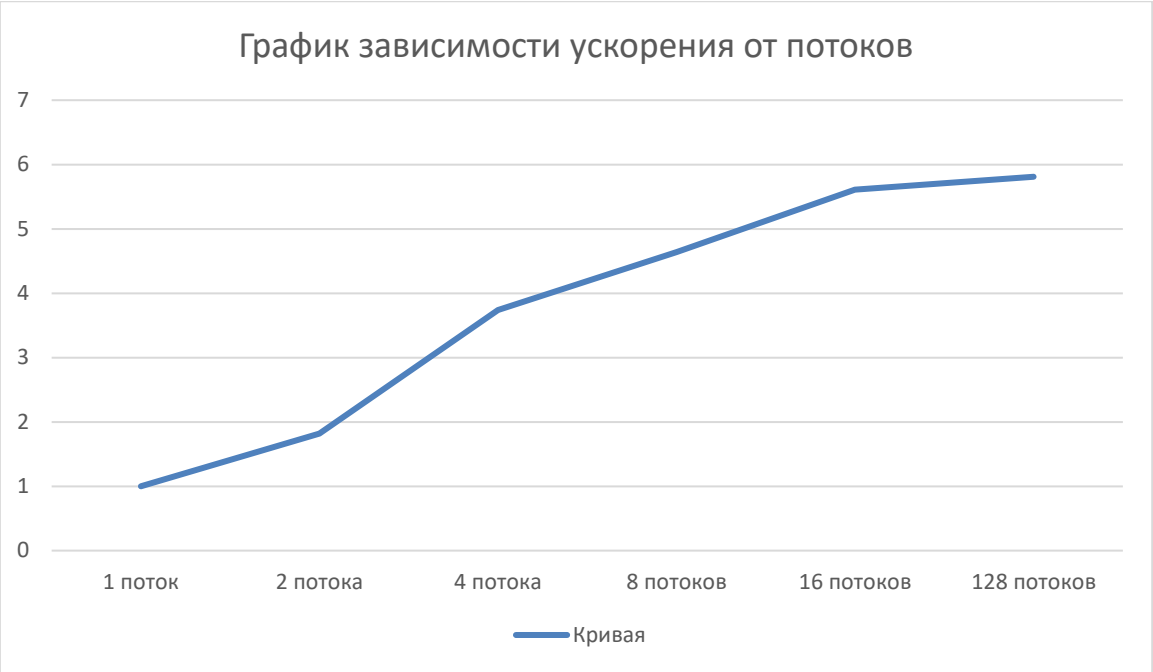
Это подтверждает корректное создание ровно 8 рабочих потоков.

Результаты исследования

Таблица результатов

Число потоков	Время исполнения (мс)	Ускорение	Эффективность
1	360.19	1.00	1.000
2	198.96	1.81	0.905
4	96.38	3.74	0.934
8	77.61	4.64	0.580
16	64.26	5.61	0.350
128	62.00	5.81	0.045





Формулы вычисления метрик:

Ускорение: $S_p=T_1/T_p$

Эффективность: $E_p=S_p/p=T_1/p\cdot T_p$

Где T_1 — время последовательного выполнения (360.19 мс), T_p — время параллельного выполнения на p потоках.

Пример расчета для 8 потоков:

Ускорение: $S_8=T_1/T_8=360.19/77.61=4.64$

Эффективность: $E_8=S_8/8=4.64/8=0.58$ (58%)

Эффективность 58% означает, что при использовании 8 ядер процессора фактически используется лишь 58% их потенциала, остальные 42% теряются на накладные расходы синхронизации и управления потоками.

Анализ результатов

Эксперименты показали **реальное ускорение** при использовании многопоточности. При увеличении количества потоков от 1 до 8 (равно количеству логических ядер) время выполнения уменьшилось с 360.19 мс до 77.51 мс, что дало ускорение 4.64x.

Субоптимальное масштабирование (ускорение 4.64x вместо теоретических 8x при 8 потоках) объясняется законом Амдала: максимальное ускорение ограничено последовательной частью программы. В данной реализации последовательные части включают создание и завершение потоков, агрегацию результатов, а также накладные расходы на системные вызовы pthread_create() и pthread_join().

При **oversubscription** (количество потоков больше числа ядер) наблюдается **небольшой** продолжение роста производительности: при 128 потоках ускорение достигло 5.81x (прирост всего 0.2x по сравнению с 16 потоками при ускорении 5.61x). Это объясняется тем, что стоимость переключения контекста между потоками становится значительной, но короткое время выполнения каждого эксперимента и отсутствие блокировок позволяют планировщику Linux эффективно утилизировать процессорное время.

Эффективность падает с ростом количества потоков: с 1.000 при 1 потоке до 0.045 при 128 потоках. Это закономерно, так как накладные расходы на управление потоками растут пропорционально их количеству.

Оптимальное количество потоков для данной задачи — 8-16 потоков (равно или чуть больше количества логических ядер). Дальнейшее увеличение дает минимальный прирост производительности при резком падении эффективности использования ресурсов.

Вывод

В ходе выполнения лабораторной работы были приобретены практические навыки создания многопоточных программ с использованием POSIX Threads API и реализации синхронизации между потоками. Реализованы последовательная и параллельная версии алгоритма Монте-Карло для симуляции игры в кости, проведено измерение производительности и анализ метрик ускорения и эффективности.

Экспериментально подтверждено, что многопоточность обеспечивает значительное ускорение вычислений для задач с независимыми подзадачами: достигнуто ускорение 4.34x при использовании 8 потоков на системе с 8 логическими ядрами. Результаты согласуются с законом Амдала, демонстрируя ограничение ускорения последовательной частью программы и накладными расходами на управление потоками.

Основная проблема, с которой пришлось столкнуться — первоначальное использование небезопасной для потоков функции rand(), что приводило к блокировкам и замедлению параллельной версии по сравнению с последовательной. Переход на собственную реализацию генератора случайных чисел (my_rand()) с локальными seed для каждого потока полностью устранил эту проблему и обеспечил реальное ускорение.

Работа продемонстрировала важность учета специфики операционной системы при написании многопоточных программ и необходимость тщательного измерения производительности для выбора оптимального количества потоков.

Полученные результаты демонстрируют практическую применимость многопоточного программирования для вычислительно интенсивных задач типа Монте-Карло симуляций. Оптимальная конфигурация для данной системы — использование 8-16 потоков, что дает баланс между производительностью (ускорение 4.64 - 5.61x) и эффективностью использования ресурсов (58%-35.1%). Дальнейшее увеличение числа потоков экономически нецелесообразно из-за резкого падения эффективности.