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**

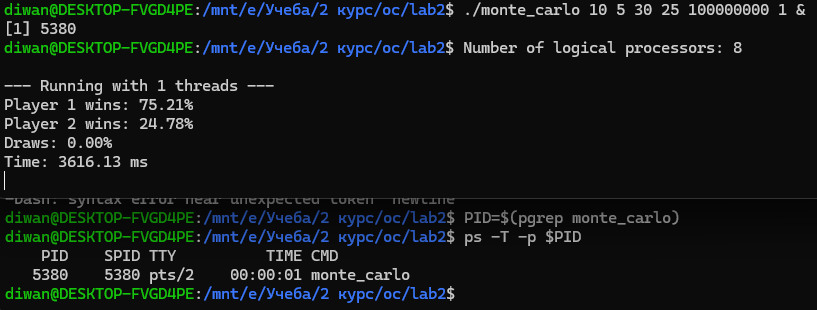
****

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)

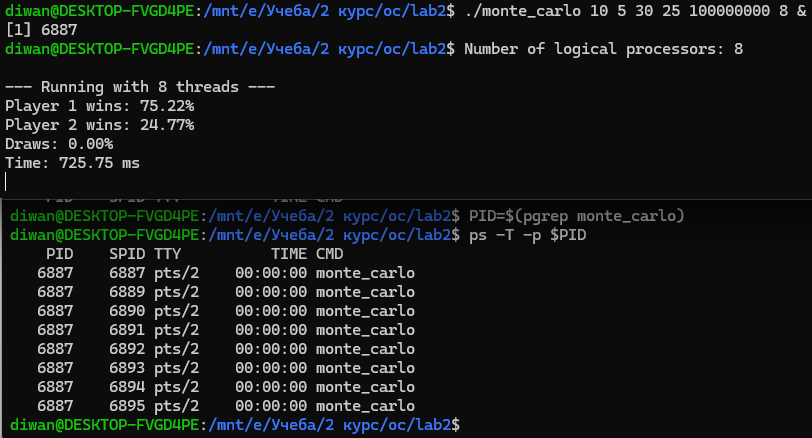


Рисунок 2. Демонстрация создания 8 потоков средствами операционной системы. Верхний терминал: запуск программы. Нижний терминал: вывод команды ps -T -p $PID, показывающей 8 потоков с уникальными SPID

**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: S­p=T1/Tp

Эффективность: Ep=Sp/p=T1/p⋅Tp

Где T1 — время последовательного выполнения (360.19 мс), Tp — время параллельного выполнения на p потоках.

**Пример расчета для 8 потоков:**

Ускорение: S8=T1/T8=360.19/77.61=4.64

Эффективность: E8=S8/8=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.