Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

Студент: Снытко Владислав Александрович

Группа: М8О–203Б–20

Вариант: 20

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2025.

**Постановка задачи**

## Цель работы:

Целью является приобретение практических навыков в:

* Управление потоками в ОС
* Обеспечение синхронизации между потоками

**Задание**:

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

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

В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

Задание варианта (7): Два человека играют в кости. Правила игры: каждый игрок делает бросок 2-х костей K раз; побеждает тот, кто выбросил суммарно большее количество очков. Задача — экспериментально определить шансы на победу каждого из игроков. На вход программе подается K, номер текущего тура, текущая суммарная сумма очков у каждого из игроков и количество экспериментов.

**Общие сведения о программе:**

Программа компилируется из файла main\_simple.c. Также используются заголовочные файлы: pthread.h, stdio.h, stdlib.h, time.h, unistd.h.  
В программе используются следующие функции и системные вызовы (краткое описание их роли):

1. pthread\_create — создаёт новый поток выполнения (pthread). Функция принимает указатель на pthread\_t для сохранения идентификатора потока, атрибуты (обычно NULL), указатель на функцию-обработчик void \*(\*start\_routine)(void \*) и аргумент для этой функции. В данной программе каждый поток запускает worker и получает свой ThreadArg.
2. pthread\_join — ожидает завершения указанного потока и (опционально) получает его возвращаемое значение. Используется в главном потоке для синхронизации: главный поток ждёт завершения всех рабочих потоков и затем суммирует их локальные результаты.
3. calloc — выделяет в динамической памяти массив элементов и инициализирует его нулями. В программе calloc используется для выделения массивов pthread\_t \*th и ThreadArg \*args размером T (число потоков). Нулевая инициализация удобна для безопасного старта счётчиков.
4. free — освобождает ранее выделенную память (вызывается для th и args после pthread\_join).
5. time (time(NULL)) — возвращает текущее время (в секундах с эпохи). В программе используется для формирования базового seed0, благодаря чему запуски в разное время дают разные случайные последовательности.
6. getpid — возвращает идентификатор текущего процесса (PID). Программа выводит PID на экран, чтобы внешний монитор (например, ps, htop) мог показать число потоков процесса во время его работы.
7. printf / fprintf / perror — стандартные функции ввода-вывода: printf печатает результат в stdout, fprintf(stderr,...) — печать сообщений об использовании, perror выводит диагностическое сообщение об ошибке системы (например, при неудаче calloc или pthread\_create).
8. Авторская функция xorshift32 (локальная функция, не системный вызов) — реализует PRNG: принимает указатель на 32‑битное состояние, выполняет серию XOR/сдвигов, обновляет состояние и возвращает 32‑битное псевдослучайное число. Используется в worker для генерации бросков костей (после приведения по модулю 6).

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить основы POSIX pthreads: модель потоков, функции pthread\_create/pthread\_join, правила передачи аргументов и возврата результатов, отсутствие общих блокировок при независимых подсчётах.
2. Реализовать простой генератор случайных чисел с состоянием на поток: выбрать xorshift32, хранить seed в структуре аргументов потока, гарантировать seed != 0 и избегать общих глобальных состояний.
3. Спроектировать разбиение экспериментов по потокам: общий объём N разделить на T потоков (равные блоки и «хвост»), каждый поток локально симулирует игры (K бросков по 2 кубика на игрока), накапливает win1/win2/draw без синхронизации.
4. Добавить разбор параметров командной строки: парсить K, N и опцию -t для ограничения числа одновременно работающих потоков; предусмотреть помощь/usage и валидацию входных данных.
5. Организовать сбор и агрегацию результатов: дождаться завершения всех потоков (pthread\_join), суммировать локальные счётчики, вычислить итоговые вероятности и, при необходимости, вывести служебную информацию (PID) для внешнего мониторинга.

**Основные файлы программы**

**main.c:**

|  |
| --- |
| #include <pthread.h> // pthread\_create, pthread\_join, тип pthread\_t  #include <stdio.h> // printf, fprintf, perror  #include <stdlib.h> // atoi, calloc, free  #include <time.h> // time (источник seed для ГПСЧ)  #include <unistd.h> // getpid (показать PID процесса для ps/htop)  typedef struct {  int K;  int s1\_start;  int s2\_start;  int count;  unsigned int seed;  unsigned long long win1;  unsigned long long win2;  unsigned long long draw;  } ThreadArg;  unsigned int xorshift32(unsigned int \*state) {  unsigned int x = \*state;  if (x == 0) x = 1;  x ^= x << 13;  x ^= x >> 17;  x ^= x << 5;  \*state = x;  return x;  }  void \*worker(void \*p) { // функция, которую исполняет каждый поток  ThreadArg \*a = (ThreadArg \*)p;  unsigned long long w1 = 0, w2 = 0, dr = 0;  for (int e = 0; e < a->count; ++e) {  int s1 = a->s1\_start;  int s2 = a->s2\_start;  for (int i = 0; i < a->K; ++i) {  int d1 = (int)(xorshift32(&a->seed) % 6u) + 1;  int d2 = (int)(xorshift32(&a->seed) % 6u) + 1;  int d3 = (int)(xorshift32(&a->seed) % 6u) + 1;  int d4 = (int)(xorshift32(&a->seed) % 6u) + 1;  s1 += d1 + d2;  s2 += d3 + d4;  }  if (s1 > s2)  ++w1;  else if (s2 > s1)  ++w2;  else  ++dr;  }  a->win1 = w1;  a->win2 = w2;  a->draw = dr;  return NULL; // возвращаем NULL (результат не используется)  }  void usage(const char \*prg) { // печать подсказки по использованию  fprintf(stderr, "Usage: %s K tour sum1 sum2 experiments -t threads\n", prg);  }  int main(int argc, char \*\*argv) {  if (argc != 8) {  usage(argv[0]);  return 1;  }  if (argv[6][0] != '-' || argv[6][1] != 't') {  fprintf(stderr, "Error: expected '-t' flag before thread count\n");  usage(argv[0]);  return 1;  }  for (int i = 1; i <= 5; i++) {  if (argv[i][0] < '0' || argv[i][0] > '9') {  if (!(i >= 3 && argv[i][0] == '-')) {  fprintf(stderr,  "Error: argument %d ('%s') is not a valid number\n", i,  argv[i]);  usage(argv[0]);  return 1;  }  }  }  if (argv[7][0] < '0' || argv[7][0] > '9') {  fprintf(stderr,  "Error: threads argument ('%s') is not a valid number\n",  argv[7]);  usage(argv[0]);  return 1;  }  int K = atoi(argv[1]);  int tour = atoi(argv[2]);  int sum1 = atoi(argv[3]);  int sum2 = atoi(argv[4]);  int N = atoi(argv[5]);  int T = atoi(argv[7]); // число потоков  if (K <= 0) {  fprintf(stderr, "Error: K must be positive (got %d)\n", K);  return 1;  }  if (N <= 0) {  fprintf(stderr, "Error: experiments must be positive (got %d)\n", N);  return 1;  }  if (T <= 0) {  fprintf(stderr, "Error: threads must be positive (got %d)\n", T);  return 1;  }  if (T > N) T = N; // не создаём больше потоков, чем задач  pthread\_t \*th = (pthread\_t \*)calloc(  (size\_t)T, sizeof(\*th)); // массив идентификаторов потоков  ThreadArg \*args = (ThreadArg \*)calloc(  (size\_t)T, sizeof(\*args)); // массив аргументов потоков  if (!th || !args) {  perror("calloc");  return 1;  }  int base = N / T;  int rem = N % T;  unsigned int seed0 = (unsigned int)time(NULL) ^  (unsigned int)(tour \* 2654435761u); /\* базовый seed \*/  printf("PID=%d\n", (int)getpid());  // 2654435761 и 0x9E3779B9 ≈ 2654435769  // константа, связанная с золотым сечением;  // часто используется в «умножающем» хешировании  // и для «разброса» индексов  for (int i = 0; i < T; ++i) {  args[i].K = K;  args[i].s1\_start = sum1;  args[i].s2\_start = sum2;  args[i].count = base + (i < rem ? 1 : 0);  args[i].seed = seed0 ^ (unsigned int)(0x9e3779b9u \* (i + 1));  if (args[i].seed == 0) args[i].seed = 1u;  if (pthread\_create(&th[i], NULL, worker, &args[i]) != 0) {  perror("pthread\_create");  return 1;  }  }  unsigned long long W1 = 0, W2 = 0, DR = 0;  for (int i = 0; i < T; ++i) {  pthread\_join(th[i], NULL);  // NULL - не забираем результат  W1 += args[i].win1;  W2 += args[i].win2;  DR += args[i].draw;  }  free(th);  free(args);  printf("K=%d, experiments=%d, threads=%d\n", K, N, T);  printf("P(win1)=%.6f, P(win2)=%.6f, P(draw)=%.6f\n", (double)W1 / (double)N,  (double)W2 / (double)N, (double)DR / (double)N);  return 0;  } |

**Исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков**

**Что измеряем**

* Фиксируем параметры задачи: K — число бросков на игру, N — число экспериментов.
* Меняем только число потоков T: 1, 2, 4, 8.
* Для каждой конфигурации берём несколько запусков и используем медиану wall time.

**Условие 1:**

* Фиксировано: K=130, tour=1, s1=0, s2=0, N=1 000 000, по 4 повтора для каждого T.
* Метрика: wall-clock время; берём медиану из повторов.

**Определения**

* Пусть T1 — медианное время при одном потоке, Tp — медианное время при p потоках.
* Ускорение: S(p) = T1 / Tp
* Эффективность: E(p) = S(p) / p

В виде формул:

* S(p) = T1 / Tp
* E(p) = S(p) / p

**Медианные времена**

* T=1: времена 0.82, 0.84, 0.83, 0.82 → медиана T1 = 0.8275 с
* T=2: 0.46, 0.47, 0.41, 0.42 → медиана T2 = 0.44 с
* T=4: 0.24, 0.24, 0.24, 0.23 → медиана T4 = 0.2375 с
* T=8: 0.13, 0.13, 0.13, 0.13 → медиана T8 = 0.13 с
* • T=16: 0.07, 0.07, 0.07, 0.07 → медиана T16 = 0.07 с
* • T=18: 0.07, 0.07, 0.08, 0.07 → медиана T18 = 0.07 с

**Ускорение и эффективность**

* Формулы: S(p) = T1 / Tp, E(p) = S(p) / p
* Результаты:
  + T=1: S(1)=1.00, E(1)=100.0%
  + T=2: S(2)=0.8275/0.44≈1.88, E(2)=94%
  + T=4: S(4)=0.8275/0.240≈3.44, E(4)=86%
  + T=8: S(8)=0.8275/0.130≈6.35, E(8)=79%
  + T=16: S(16)=0.8275/0.07≈11.82, E(16)≈74%
  + T=18: S(18)=0.8275/0.070≈11.82, E(18)≈66%

**Короткая интерпретация**

* До 4 потоков ускорение близко к линейному (эффективность ~86–94%), т.к. работа хорошо распараллеливается, а накладные расходы малы.
* На 8 потоках эффективность снижается до ~79%: сказываются планирование потоков, конкуренция за ядра/кеш и неизбежные накладные расходы.
* На 16 потоках ускорение продолжает расти, но эффективность падает — это ожидаемое поведение при превышении числа физических ядер
* При T=18 наблюдается ухудшение по сравнению с T=16: планировщик и конкуренция за кеш/память приводят к потере КПД.
* Если уменьшать K или N, эффективность будет ниже (меньше полезной работы на один поток); если увеличивать — обычно ближе к линейному ускорению до числа доступных аппаратных потоков.

**Услови 2(N = 10 млн):**

* Фиксировано: K=100, tour=1, s1=0, s2=0, N=10 000 000, по 4 повтора для каждого T.
* Метрика: wall-clock время; берём медиану из повторов.

**Медианные времена**

* T=1: 6.31, 6.31, 6.31, 6.30 → T1 = 6.31 с
* T=2: 3.33, 3.21, 3.23, 3.20 → T2 = 3.22 с
* T=4: 1.70, 1.74, 1.76, 1.70 → T4 = 1.72 с
* T=8: 0.98, 0.92, 0.99, 0.98 → T8 = 0.98 с
* T=16: 0.56, 0.58, 0.64, 0.58→ T16 =0.59с
* T=18: 0.7, 0.7, 0.71, 0.69 → T18 = 0.7с

**Ускорение и эффективность**

* Формулы: S(p)=T1/Tp, E(p)=S(p)/p
* Результаты:
  + T=1: S=1.00×, E=100.0%
  + T=2: S=6.31/3.22=1.96×, E=98.0%
  + T=4: S=6.31/1.72=3.67×, E=91.7%
  + T=8: S=6.31/0.98=6.44×, E=80.5%
  + T=16: S=6.31/0.59=10.88, E=68.0%
  + T=18: S=6.31/0.7=9.28, E=51.6%

**Короткая интерпретация (для N=10 млн)**

* До 4 потоков ускорение близко к линейному (эффективность ≈92–98%).
* На 8 потоках эффективность остаётся высокой (≈80%), но ниже из‑за накладных расходов, планирования и конкуренции за ресурсы.
* На 16 потоках ускорение продолжает расти, но эффективность падает — это ожидаемое поведение при превышении числа физических ядер
* При T=18 наблюдается ухудшение по сравнению с T=16: планировщик и конкуренция за кеш/память приводят к потере КПД.

**Сравнение с предыдущим набором(N=10 млн vs N=1 млн)**

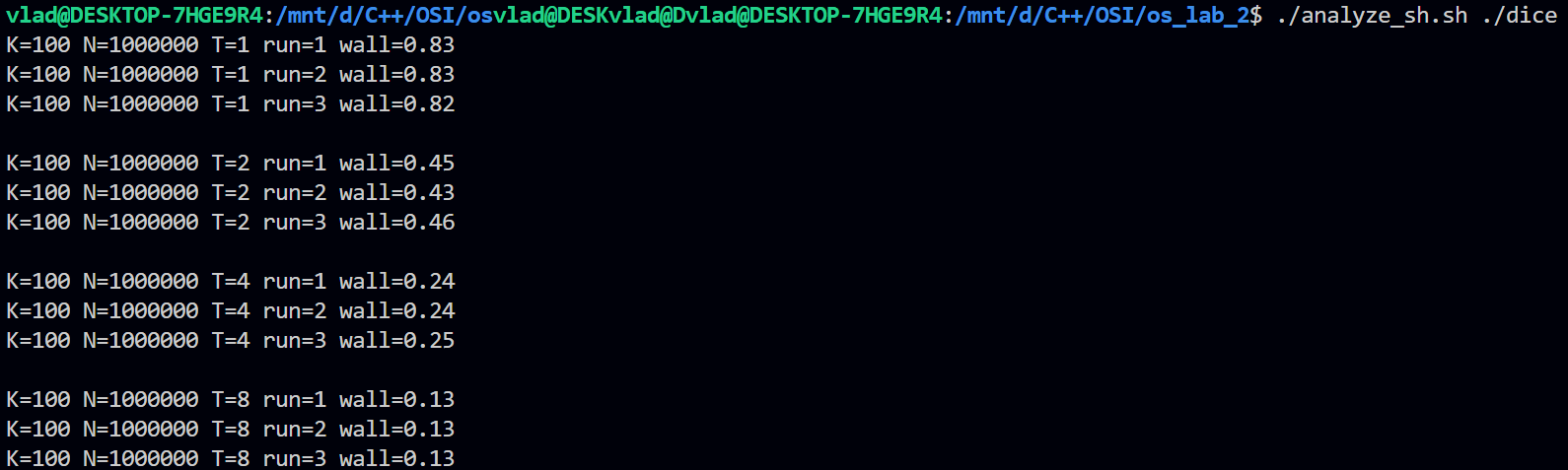
* N=1 млн: E(2)=93.8%, E(4)=85.9%, E(8)=79.3%, E(16)=68.0%, E(18)=51,6%.
* N=10 млн: E(2)=98.0%, E(4)=91.7%, E(8)=80.5%, E(16)=68.0%, E(18)=51,6%.
* Тенденция: при большем N эффективность выше при том же T — полезной работы больше, накладные расходы (создание потоков, агрегация) лучше амортизируются.

**Общий вывод по зависимости от входных данных**

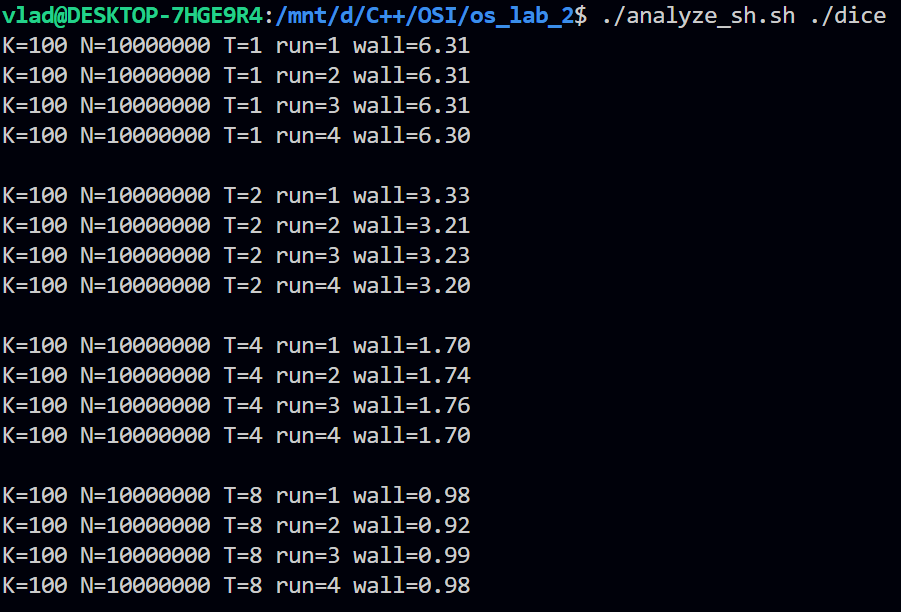
* Увеличение объёма работы на поток (рост N и/или K) улучшает масштабирование: ускорение ближе к линейному до числа доступных аппаратных потоков, эффективность выше.
* При малых N или K эффективность падает — доля накладных расходов становится заметной.
* Дальнейшее наращивание T сверх числа физических/логических ядер даёт меньший прирост из‑за планировщика, конкуренции за память.

**Пример работы**

K=100, tour=1, s1=0, s2=0, N=1 000 000:

****

K=100, tour=1, s1=0, s2=0, N=10 000 000:

****

**Вывод**

Создан минимальный многопоточный симулятор на pthreads — число потоков задаётся опцией, у каждого потока свой генератор случайных чисел, результаты считаются локально и суммируются без блокировок; для измерений добавлен скрипт, собирающий wall‑time по повторным запускам и сохраняющий результаты.

Исследование показало: при достаточном объёме работы ускорение растёт почти линейно до уровня аппаратных ресурсов, при дальнейшем увеличении числа потоков прирост заметно снижается из‑за планирования и конкуренции за кэш/память; при малых задачах накладные расходы доминируют и эффективность падает.