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: Беляк Андрей Игоревич

Группа: М8О–203БВ–24

Вариант: 7

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 <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <string.h>  #include <time.h>  typedef struct  {      int k;      int experiments\_per\_thread;      int \*wins1;      int \*wins2;      int \*draws;      unsigned int seed;  } ThreadArgs;  void \*simulate\_games(void \*arg)  {      ThreadArgs \*args = (ThreadArgs \*)arg;      int wins1 = 0, wins2 = 0, draws = 0;      for (int i = 0; i < args->experiments\_per\_thread; i++)      {          int sum1 = 0, sum2 = 0;          for (int j = 0; j < args->k; j++)          {              sum1 += rand\_r(&args->seed) % 6 + rand\_r(&args->seed) % 6 + 2;              sum2 += rand\_r(&args->seed) % 6 + rand\_r(&args->seed) % 6 + 2;          }          if (sum1 > sum2)              wins1++;          else if (sum1 < sum2)              wins2++;          else              draws++;      }      \*(args->wins1) += wins1;      \*(args->wins2) += wins2;      \*(args->draws) += draws;      return NULL;  }  int main(int argc, char \*argv[])  {      struct timespec start\_time, end\_time;      clock\_gettime(CLOCK\_MONOTONIC, &start\_time);      int k = 0, total\_experiments = 0, max\_threads = 1;      int current\_round = 0, score1 = 0, score2 = 0;      for (int i = 1; i < argc; i++)      {          if (strcmp(argv[i], "-k") == 0)              k = atoi(argv[++i]);          else if (strcmp(argv[i], "-r") == 0)              current\_round = atoi(argv[++i]);          else if (strcmp(argv[i], "-s1") == 0)              score1 = atoi(argv[++i]);          else if (strcmp(argv[i], "-s2") == 0)              score2 = atoi(argv[++i]);          else if (strcmp(argv[i], "-n") == 0)              total\_experiments = atoi(argv[++i]);          else if (strcmp(argv[i], "-t") == 0)              max\_threads = atoi(argv[++i]);      }      if (k <= 0 || total\_experiments <= 0)      {          char error\_msg[256];          int len = snprintf(error\_msg, sizeof(error\_msg),                             "Usage: %s -k K -r ROUND -s1 SCORE1 -s2 SCORE2 -n EXPERIMENTS [-t THREADS]\n", argv[0]);          write(2, error\_msg, len);          return 1;      }      int wins1 = 0, wins2 = 0, draws = 0;      pthread\_t \*threads = malloc(max\_threads \* sizeof(pthread\_t));      ThreadArgs \*thread\_args = malloc(max\_threads \* sizeof(ThreadArgs));      int experiments\_per\_thread = total\_experiments / max\_threads;      for (int i = 0; i < max\_threads; i++)      {          thread\_args[i].k = k;          thread\_args[i].experiments\_per\_thread = experiments\_per\_thread;          if (i == max\_threads - 1)          {              thread\_args[i].experiments\_per\_thread += total\_experiments % max\_threads;          }          thread\_args[i].wins1 = &wins1;          thread\_args[i].wins2 = &wins2;          thread\_args[i].draws = &draws;          thread\_args[i].seed = time(NULL) ^ getpid() ^ i;          pthread\_create(&threads[i], NULL, simulate\_games, &thread\_args[i]);      }      for (int i = 0; i < max\_threads; i++)      {          pthread\_join(threads[i], NULL);      }      free(threads);      free(thread\_args);      char buffer[512];      int len;      len = snprintf(buffer, sizeof(buffer), "Total experiments: %d\n", total\_experiments);      write(1, buffer, len);      len = snprintf(buffer, sizeof(buffer), "Player 1 wins: %d %.2f%%\n",                     wins1, (100.0 \* wins1 / total\_experiments));      write(1, buffer, len);      len = snprintf(buffer, sizeof(buffer), "Player 2 wins: %d %.2f%%\n",                     wins2, (100.0 \* wins2 / total\_experiments));      write(1, buffer, len);      len = snprintf(buffer, sizeof(buffer), "Draws: %d %.2f%%\n",                     draws, (100.0 \* draws / total\_experiments));      write(1, buffer, len);      clock\_gettime(CLOCK\_MONOTONIC, &end\_time);      double execution\_time = (end\_time.tv\_sec - start\_time.tv\_sec) +                              (end\_time.tv\_nsec - start\_time.tv\_nsec) / 1e9;      len = snprintf(buffer, sizeof(buffer), "Execution time: %.6f seconds\n", execution\_time);      write(1, buffer, len);      return 0;  }  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**

./run\_file -k 5 -r 0 -s1 0 -s2 0 -n 1000000 -t 5

Total experiments: 1000000

Player 1 wins: 473740 47.37%

Player 2 wins: 475031 47.50%

Draws: 51229 5.12%

Execution time: 0.041933 seconds

**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.