Белорусский государственный технологический университет
Факультет информационных технологий
Кафедра программной инженерии

# Лабораторная работа 5

По дисциплине «Операционные системы»

На тему «Диспетчеризация и планирование»

Выполнил:
Студент 3 курса 9 группы
Павлович Ян Андреевич
Преподаватель: Савельева М.Г.

Минск, 2025

# Введение

**Цель работы:**

Освоение практических аспектов планирования и управления выполнением потоков в операционных системах.

**Задачи исследования:**

Необходимо разработать набор приложений для операционных систем Windows и Linux, предназначенных для управления потоками, организации взаимодействия между ними и анализа их состояния. В среде Windows требуется реализовать создание потоков с применением WinAPI, управление их жизненным циклом, операции приостановки и завершения, а также использование потоково-локального хранилища (TLS) через WinAPI и модификатор __declspec(thread).

Для Linux необходимо создать аналогичные программы с использованием POSIX Threads, реализовать TLS с помощью pthread_once и потоково-локальные переменные с модификатором __thread.

Во всех программах нужно предусмотреть передачу параметров через аргументы командной строки, корректную обработку ошибок, эффективное управление ресурсами, а также анализ процессов и потоков с использованием системных утилит (ps, /proc, Process Explorer).

Также требуется выполнить сравнение производительности многопоточных и многопроцессных реализаций, включая измерение времени выполнения при различном количестве потоков и процессов.

**Используемые инструменты:**

- Parallels Desktop
- Командная оболочка cmd
- Утилита Sysinternals Process Explorer
- MinGW-w64 и g++
- Visual Studio

## 1 Windows

### 1.1 Приложение Lab-05a

Приложение Lab-05a

Это простое консольное приложение, которое выводит на экран информацию о текущем процессе и потоке. В частности, отображаются идентификаторы процесса и потока, класс приоритета процесса и приоритет потока, маска родственности процесса и системная маска в двоичном представлении, число процессоров, доступных для выполнения процесса, а также номер процессора, на котором в данный момент выполняется поток.

Для реализации создаётся файл с расширением .txt на языке C++, в который помещается код, приведенный в листинге 1.1.

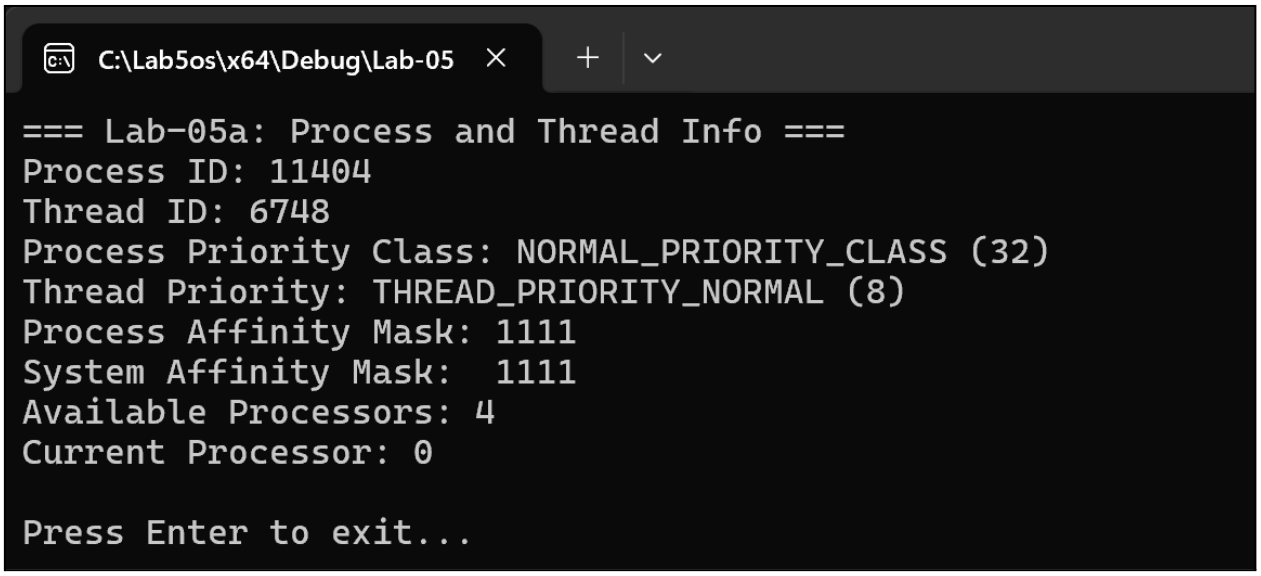


Рисунок 1.1 – Результат работы Lab-05a

Результаты компиляции и выполнения программы показаны на соответствующем рисунке 1.1.

Приложение функционирует без ошибок и корректно выполняет поставленную задачу.

### 1.2 Приложение Lab-05x

Данное консольное приложение выполняет цикл, состоящий из одного миллиона итераций. Через каждые 1000 шагов выполнение приостанавливается на 200 мс, после чего в консоль выводится текущая информация: номер итерации, идентификаторы процесса и потока, класс приоритета процесса, приоритет потока, а также номер процессора, на котором осуществляется выполнение.

После завершения цикла программа выводит время, прошедшее с момента ее запуска, используя функцию clock.

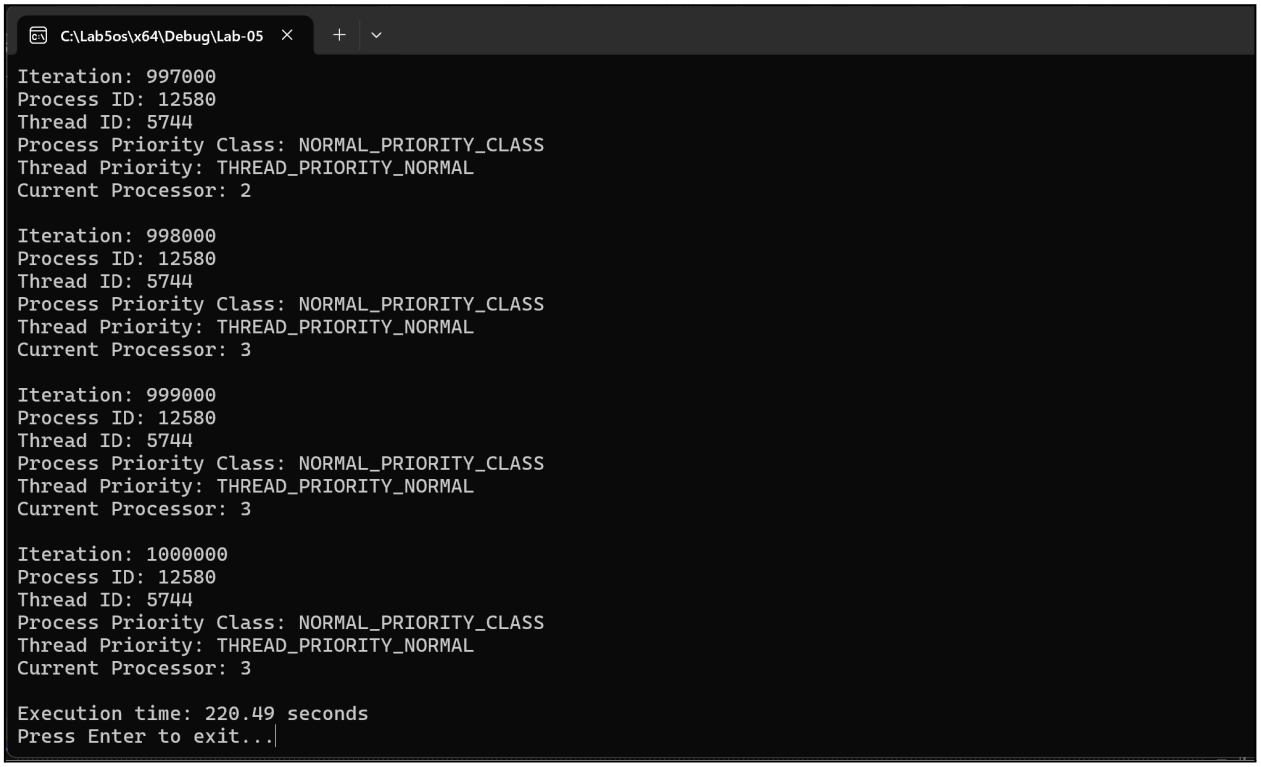


Рисунок 1.2 – Результат работы Lab-05x

Для реализации создается файл Lab-05x.cpp, в который помещается код, приведенный в листинге 1.2. Результат работы программы представлен на рисунке 1.2.

### 1.3 Приложение Lab-05b

Данное консольное приложение принимает три аргумента командной строки. Первый параметр задаёт маску родственности процессоров, второй определяет класс приоритета первого дочернего процесса, а третий — класс приоритета второго дочернего процесса.
Программа выводит полученные значения на экран и запускает два идентичных дочерних процесса, основанных на приложении Lab-05x. Каждый из этих процессов работает в собственном консольном окне и запускается с заданным приоритетом.

Таблица 1.1 – Параметры для Lab-05b.exe

| | | |
|---|---|---|
| 4 | 32 | 32 |
| 4 | 16384 | 128 |
| 1 | 16384 | 128 |

Для реализации создается файл Lab-05b.cpp, в который записывается код, приведенный в листинге 1.3. После этого выполняется компиляция исходного файла в исполняемый файл и его запуск.

| Process | CPU | Private Bytes | Working Set | PID | Description | Co... | Priority | Tree CPU Usage |
|---|---|---|---|---|---|---|---|---|
| Lab-05b.exe | | 712 K | 4,884 K | 7072 | | | 8 | 0.72 |
| Lab-05x.exe | 0.36 | 640 K | 4,308 K | 9404 | | | 8 | 0.36 |
| Lab-05x.exe | 0.36 | 648 K | 4,312 K | 11724 | | | 8 | 0.36 |

Рисунок 1.3 – Приоритеты при первом тесте

В первом эксперименте оба процесса имеют класс приоритета Normal и могут выполняться на всех доступных процессорах. В таких условиях оба экземпляра Lab-05x.exe работают параллельно на разных процессорах, а планировщик равномерно распределяет процессорное время.

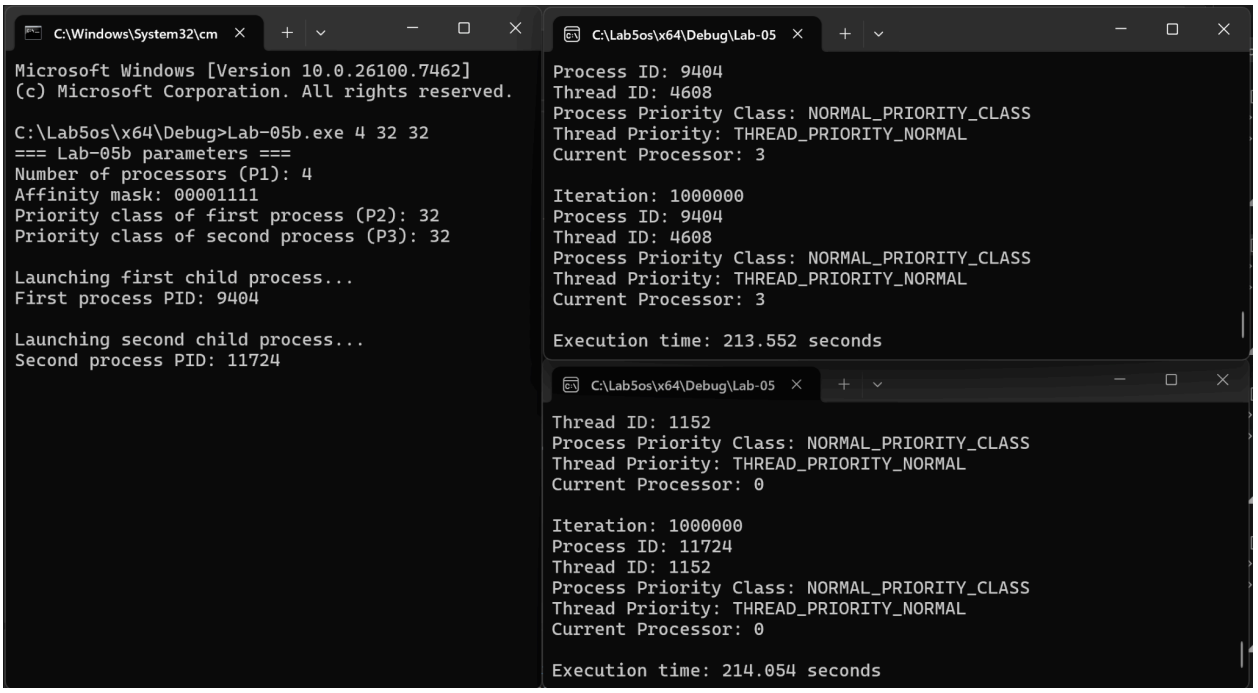


Рисунок 1.4 – Результат работы первого теста

За счёт отсутствия конкуренции за CPU достигается высокая производительность, что также подтверждается отображением одинакового приоритета Normal для обоих процессов в Process Explorer.

Во втором эксперименте процессам назначаются разные приоритеты — Below Normal и High, при этом сохраняется доступ ко всем процессорам.

| Process | CPU | Private Bytes | Working Set | PID | Description | Compa... | Priority | Tree CPU Usage |
|---|---|---|---|---|---|---|---|---|
| Lab-05b.exe | | 700 K | 4,776 K | 3036 | | | 8 | 0.72 |
| Lab-05x.exe | 0.72 | 632 K | 4,308 K | 13188 | | | 6 | 0.72 |
| Lab-05x.exe | < 0.01 | 632 K | 4,304 K | 13128 | | | 13 | |

Рисунок 1.5 – Приоритеты при втором тесте

Процесс с пониженным приоритетом функционирует преимущественно в фоновом режиме, тогда как процесс с высоким приоритетом получает преимущество при распределении ресурсов.

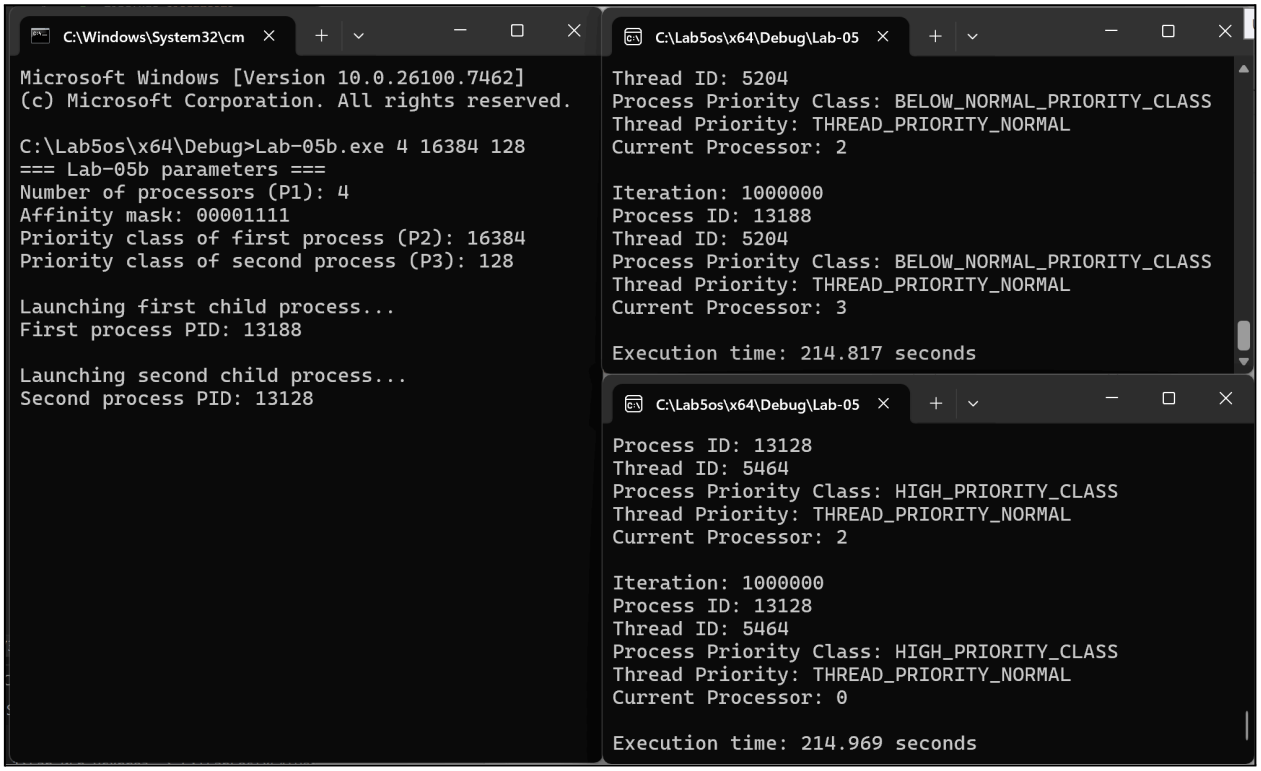


Рисунок 1.6 – Результат работы второго теста

Несмотря на различия в приоритетах, процессы могут выполняться параллельно на разных процессорах, а в Process Explorer отображаются соответствующие значения приоритетов.

В третьем эксперименте процессы также имеют разные приоритеты, однако их выполнение ограничено одним процессором.

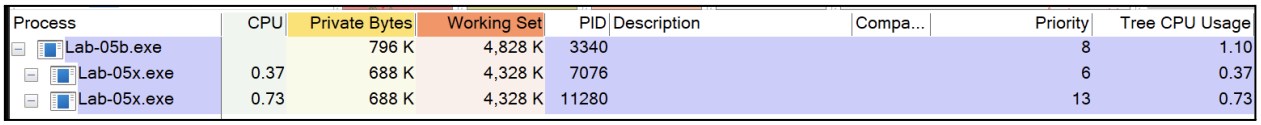


Рисунок 1.7 – Приоритеты при третьем тесте

В результате оба процесса вынуждены конкурировать за один CPU, и процесс с высоким приоритетом получает значительно большую долю процессорного времени.

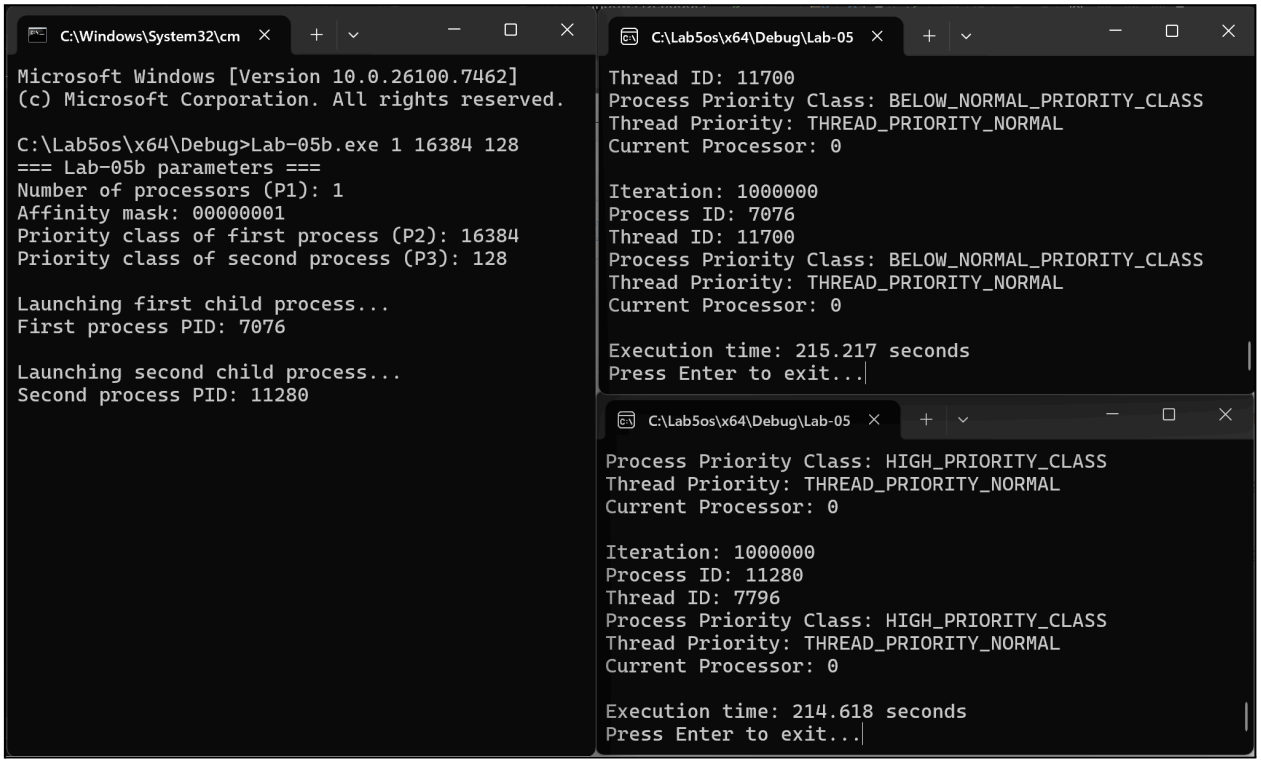


Рисунок 1.8 – Результат работы третьего теста

Процесс с приоритетом Below Normal часто вытесняется и выполняется заметно медленнее. Общая производительность системы снижается из-за ограничения по количеству доступных процессоров, при этом влияние приоритетов становится особенно наглядным.

### 1.4 Приложение Lab-05

Данное приложение представляет собой простую консольную программу, которая принимает четыре аргумента командной строки. Первый параметр задаёт маску родственности процессоров, второй определяет класс приоритета процесса, третий и четвертый — классы приоритетов первого и второго дочерних потоков соответственно.

В программе используется потоковая функция, аналогичная той, что применялась в приложении Lab-05x. Приложение выводит полученные параметры на экран и создает два одинаковых дочерних потока, каждый из которых выполняет вывод в консоль с установленным для него приоритетом.
Для реализации создается файл с расширением cpp, в который помещается код, приведенный в листинге 1.4. После этого приложение компилируется и запускается для тестирования.

В ходе работы программа проверялась с параметрами, указанными в таблице 1.2, после чего были выполнены три тестовых запуска.

```
C:\Lab5os\x64\Debug>Lab-05c.exe 4 32 8 8
=== Lab-05c parameters ===
Processors used: 4
Affinity mask: 00001111
Process priority: NORMAL_PRIORITY_CLASS
Thread priorities: THREAD_PRIORITY_NORMAL , THREAD_PRIORITY_NORMAL
```

Рисунок 1.9 – Параметры запуска первого теста

```
--- Thread 2 ---
Iteration: 1000000
Process ID: 11508
Thread ID: 4652
Process Priority: NORMAL_PRIORITY_CLASS
Thread Priority: THREAD_PRIORITY_NORMAL
Current Processor: 3
Affinity Mask: 00001111

--- Thread 1 ---
Iteration: 1000000
Process ID: 11508
Thread ID: 11396
Process Priority: NORMAL_PRIORITY_CLASS
Thread Priority: THREAD_PRIORITY_NORMAL
Current Processor: 3
Affinity Mask: 00001111

Thread 2 finished. Time elapsed: 221.566 seconds
Thread 1 finished. Time elapsed: 221.815 seconds
```

Рисунок 1.10 – Результат работы первого теста

В первом эксперименте с параметрами 4/32/8/8 оба потока имели одинаковый нормальный приоритет, из-за чего их выполнение происходило равномерно и без заметного перекоса в сторону одного из потоков.

```
C:\Lab5os\x64\Debug>Lab-05c.exe 4 32 2 31
=== Lab-05c parameters ===
Processors used: 4
Affinity mask: 00001111
Process priority: NORMAL_PRIORITY_CLASS
Thread priorities: THREAD_PRIORITY_LOWEST , THREAD_PRIORITY_HIGHEST
```

Рисунок 1.11 – Параметры запуска второго теста

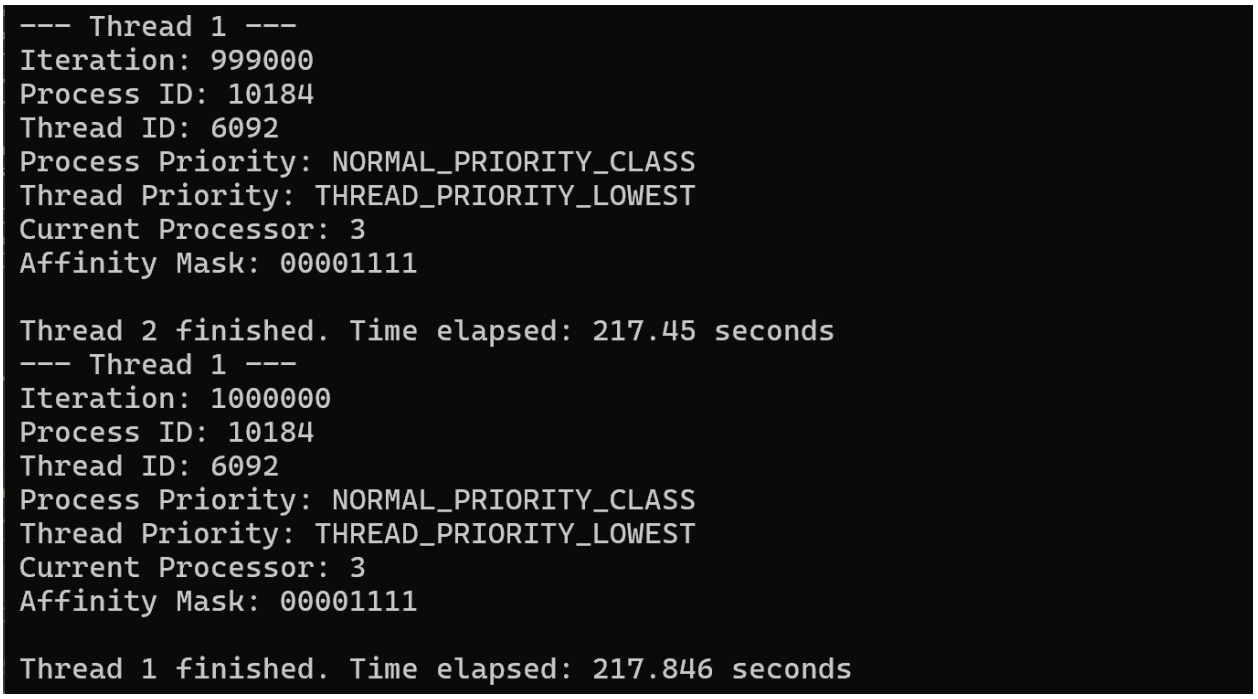


Рисунок 1.12 – Результат работы второго теста

Во втором эксперименте, при параметрах 4/32/2/31, потокам были назначены различные приоритеты — низкий и высокий. В результате поток с более высоким приоритетом получал существенно больше процессорного времени и выполнялся быстрее.

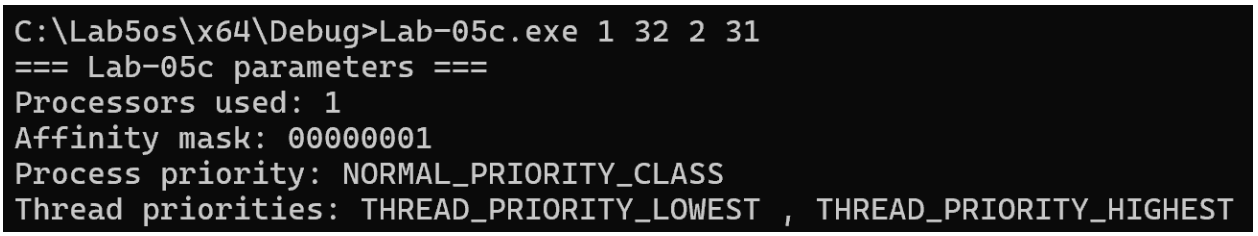


Рисунок 1.13 – Параметры запуска третьего теста

В третьем эксперименте параметры имели вид 1/32/2/31, то есть потоки с разными приоритетами были ограничены выполнением на одном ядре. В этих условиях различия в скорости работы потоков стали ещё более заметными.


Рисунок 1.14 – Результат работы третьего теста

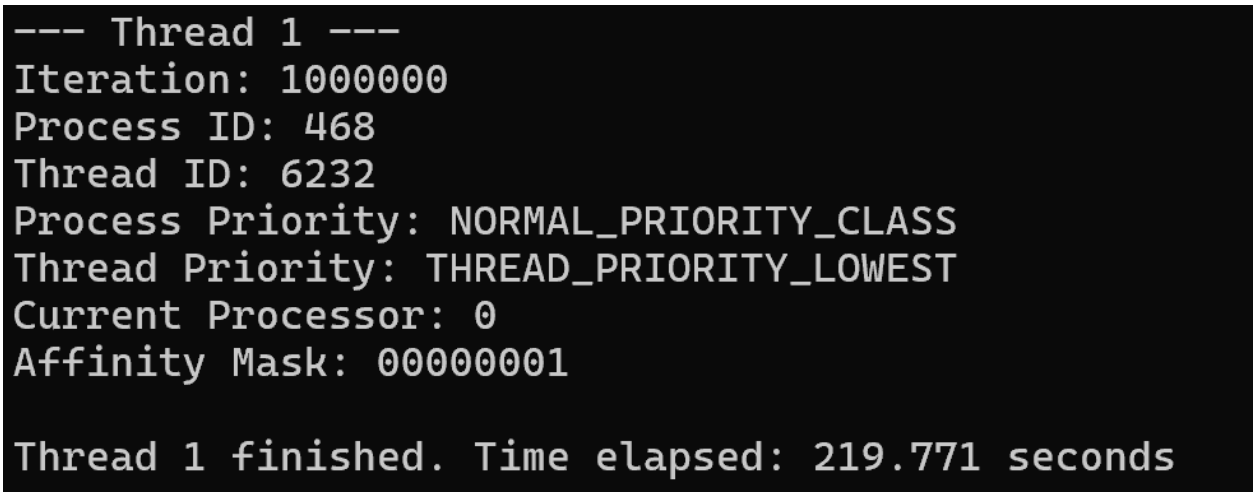


Рисунок 1.15 – Результат работы третьего теста

Таким образом, при одинаковых приоритетах планировщик операционной системы распределяет процессорное время между потоками достаточно равномерно. При различии приоритетов преимущество получает поток с более высоким приоритетом, а ограничение выполнения одним ядром усиливает конкуренцию за ресурсы и делает влияние приоритета наиболее выраженным.

## 2 Linux

### 2.1 Приложение Lab-05a

Данное приложение по своей функциональности повторяет вариант, реализованный для Windows, за исключением вывода системной маски родственности, которая в среде Linux не требуется.

Для реализации создаётся файл Lab_05a.cpp, в который помещается код, приведенный в листинге 2.1. После этого исходный файл компилируется в исполняемый файл и выполняется запуск приложения.

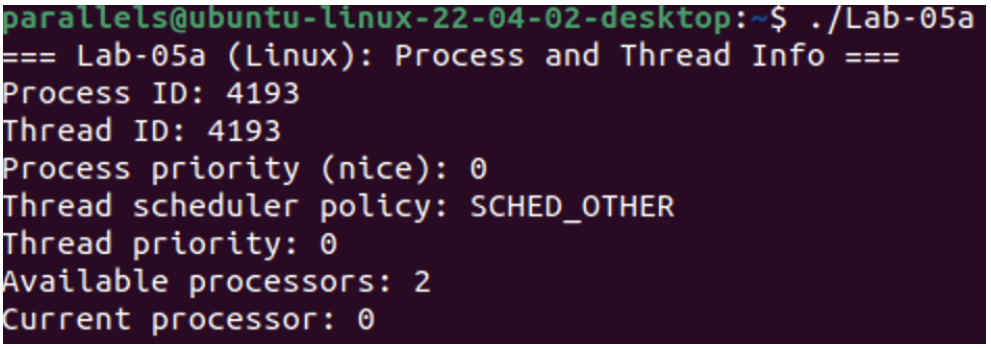


Рисунок 2.1 – Результат работы Lab-05a

Результаты работы программы показаны на соответствующем рисунке. Приложение функционирует корректно.

### 2.2 Приложение Lab-05x

Это простое консольное приложение, выполняющее цикл из одного миллиона итераций. После каждых 1000 итераций программа делает паузу продолжительностью 200 мс и выводит на экран информацию о текущем состоянии выполнения, включая номер итерации, идентификаторы процесса и потока, значение уровня любезности, а также номер процессора, на котором выполняется поток.

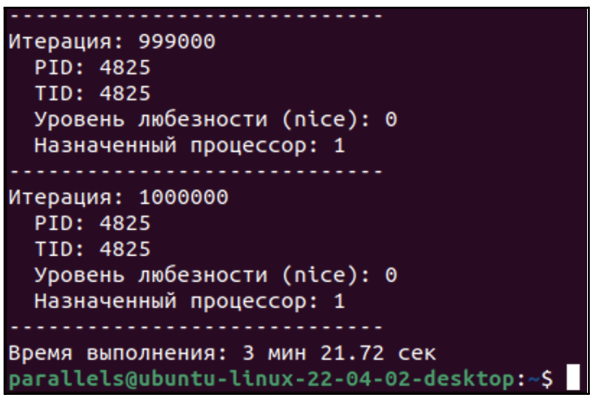


Рисунок 2.2 – Результат работы Lab-05x

По завершении работы приложение отображает время, прошедшее с момента его запуска.

Для реализации создается файл Lab-05x, в который записывается код, представленный в листинге 2.2. После компиляции программа запускается для проверки. Приложение функционирует корректно.

### 2.3 Приложение Lab-05b

Данное консольное приложение принимает три входных параметра. Первый аргумент определяет маску родственности процессоров, второй задаёт приоритет первого дочернего процесса, а третий — приоритет второго дочернего процесса.

Программа выводит принятые параметры на экран и создает два одинаковых дочерних процесса на основе приложения Lab-05x. Оба процесса выполняют вывод в консоль и запускаются с установленными приоритетами, которые назначаются с помощью механизма nice.

Таблица 2.1 – Параметры для Lab-05b.exe

| P1 | P2 | P3 |
|---|---|---|
| 2 | 0 | 0 |
| 2 | 19 | -20 |
| 1 | 19 | -20 |

Для серии экспериментов используются параметры, приведенные в таблице 2.1. В процессе выполнения приложения необходимо получить сведения о приоритетах процессов с помощью утилиты ps и файловой системы /proc, а также определить маску родственности процессоров через каталог /proc.

```
parallels@ubuntu-linux-22-04-02-desktop:~$ ./Lab-05b 2 0 0
Маска процессоров (P1): 2
Приоритет дочернего процесса 1 (P2): 0
Приоритет дочернего процесса 2 (P3): 0
Процесс 1 запущен: PID = 8204
Процесс 2 запущен: PID = 8205
```

Рисунок 2.3 – Результат работы первого теста

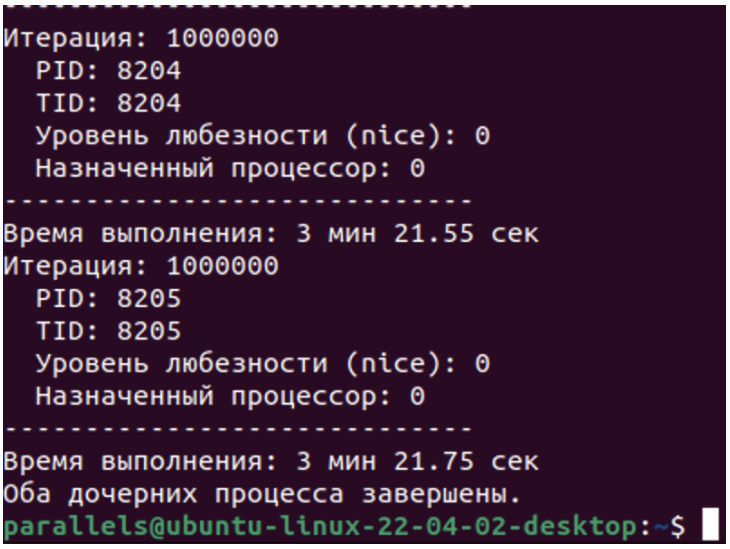


Рисунок 2.4 – Результат работы первого теста

Для реализации создаётся файл Lab-05b, в который записывается код, представленный в листинге Ж. После компиляции выполняется запуск приложения и наблюдение за его работой, включая мониторинг информации о процессах с использованием командной строки.

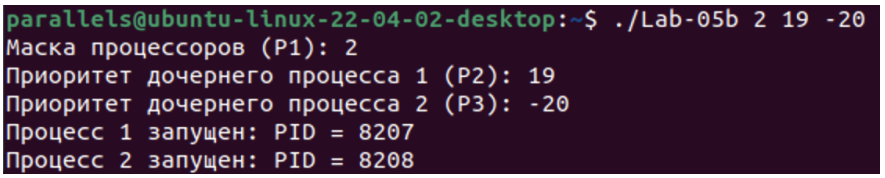


Рисунок 2.5 – Результат работы второго теста

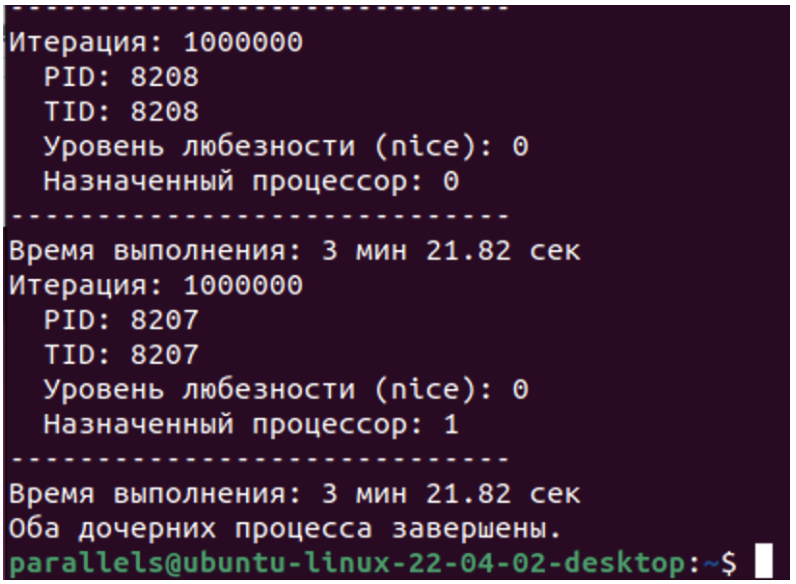


Рисунок 2.6 – Результат работы второго теста

В ходе эксперимента были исследованы процессы с различными значениями приоритетов и масок процессоров. Приложение запускалось три

раза с разными настройками, что позволило сделать следующие выводы. При одинаковых приоритетах и полном доступе ко всем ядрам оба процесса выполнялись практически с одинаковой скоростью, а различия в количестве итераций и времени работы были минимальными, что указывает на равномерное распределение ресурсов планировщиком Linux.

```
parallels@ubuntu-linux-22-04-02-desktop:~$ ./Lab-05b 1 19 -20
Маска процессоров (P1): 1
Приоритет дочернего процесса 1 (P2): 19
Приоритет дочернего процесса 2 (P3): -20
Процесс 1 запущен: PID = 8213
Процесс 2 запущен: PID = 8214
```

Рисунок 2.7 – Результат работы третьего теста

```
------------------------------
Итерация: 1000000
  PID: 8213
  TID: 8213
  Уровень любезности (nice): 0
  Назначенный процессор: 1
------------------------------
Время выполнения: 3 мин 21.72 сек
Итерация: 1000000
  PID: 8214
  TID: 8214
  Уровень любезности (nice): 0
  Назначенный процессор: 0
------------------------------
Время выполнения: 3 мин 21.74 сек
Оба дочерних процесса завершены.
parallels@ubuntu-linux-22-04-02-desktop:~$
```

Рисунок 2.8 – Результат работы третьего теста

При сохранении доступа ко всем ядрам, но при разных значениях nice, процесс с более высоким приоритетом выполнялся быстрее и успевал выполнить больше итераций. Это демонстрирует, что планировщик выделяет больше процессорного времени задачам с повышенным приоритетом даже при отсутствии ограничений по вычислительным ресурсам.


Рисунок 2.9 – Результат работы Lab-05b

В случае ограничения выполнения одним ядром при одновременном различии приоритетов конкуренция за процессор усиливалась. Процесс с высоким приоритетом получал основную долю CPU-времени, тогда как второй процесс значительно замедлялся из-за нехватки ресурсов.

### 2.4 Приложение Lab-05c

Это консольное приложение принимает три аргумента командной строки: первый задает маску родственности процессоров, второй — приоритет первого дочернего потока, третий — приоритет второго дочернего потока.

Таблица 2.2 – Параметры для Lab-05c.exe

| P1 | P2 | P3 |
|---|---|---|
| 2 | 0 | 0 |
| 2 | 19 | -20 |
| 1 | 19 | -20 |

Программа использует потоковую функцию, аналогичную Lab-05x, выводит на экран полученные параметры и создает два одинаковых дочерних потока, выполняющих вывод в консольные окна с заданными приоритетами, назначаемыми через setpriority.

Для серии экспериментов используются параметры, приведенные в таблице 2.2. В момент завершения одного из потоков фиксируется различие в количестве выполненных итераций, после чего результаты анализируются и сравниваются с поведением процессов.

Для реализации создаётся файл Lab-05c.cpp, в который помещается код из листинга 3. После компиляции программа запускается для тестирования.

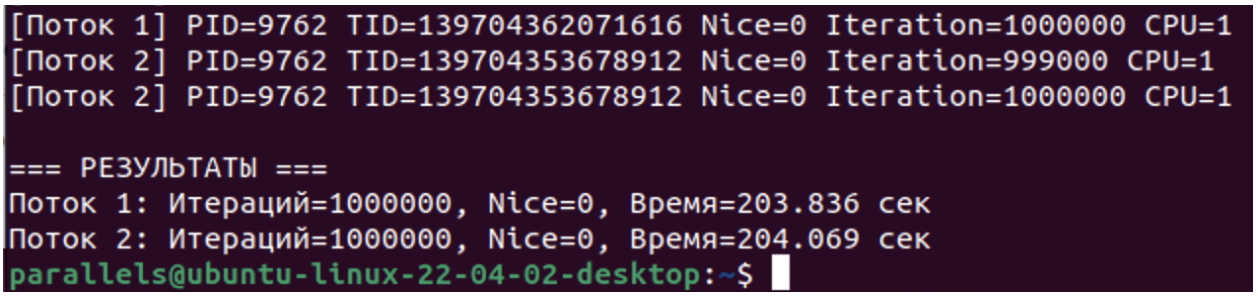


Рисунок 2.10 – Результат работы первого теста

В первом эксперименте, при одинаковых приоритетах и полной доступности всех процессорных ядер, оба потока завершали работу почти одновременно. Количество выполненных итераций и общее время выполнения были близки, что указывает на равномерное распределение ресурсов планировщиком ОС. Процессы и потоки показывают аналогичное поведение в условиях одинаковых приоритетов и неограниченного доступа к ядрам.

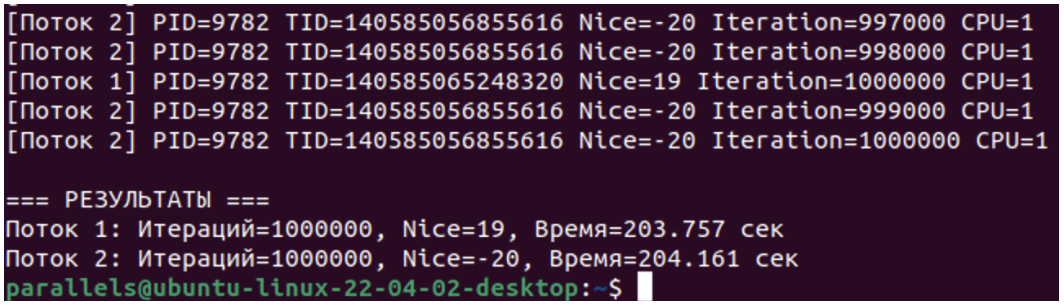


Рисунок 2.11 – Результат работы второго теста

Во втором эксперименте при различии приоритетов поток с более высоким приоритетом получал явное преимущество: он завершался быстрее и выполнял больше вычислений, тогда как поток с низким приоритетом значительно отставал. Это подтверждает, что планировщик отдаёт больше ресурсов более приоритетным задачам, вне зависимости от того, являются они потоками или процессами.

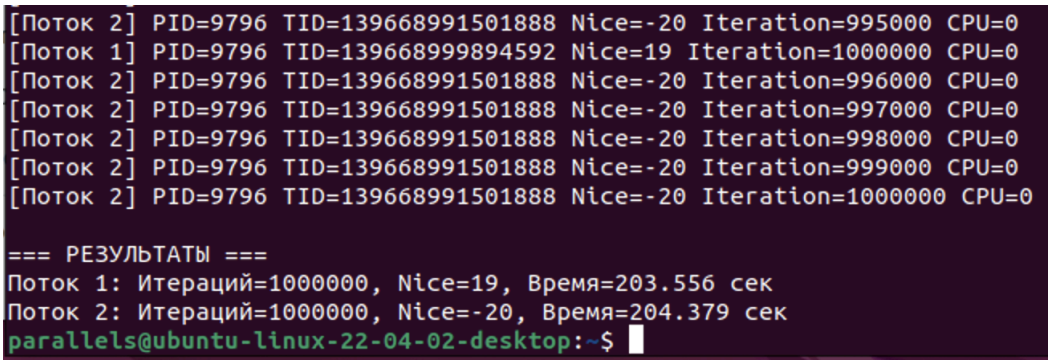


Рисунок 2.12 – Результат работы третьего теста

В третьем эксперименте сочетание разных приоритетов с ограничением работы одним ядром усилило эффект приоритета: поток с приоритетом -20 занимал почти всё доступное процессорное время, а выполнение второго потока практически блокировалось. Потоки, функционируя внутри одного процесса и разделяя общее адресное пространство и маску процессорных ядер, демонстрируют ограниченную гибкость управления. В отличие от потоков, процессы позволяют более точно назначать маски процессоров и приоритеты, что делает их поведение более наглядным при сравнительном анализе.

## Заключение

В ходе лабораторной работы были успешно освоены практические аспекты работы с потоками в операционных системах Windows и Linux. Для Windows разработаны и протестированы приложения, демонстрирующие создание потоков через WinAPI, их синхронизацию, управление жизненным циклом (приостановку, возобновление, принудительное завершение), а также использование Tool Help Library для анализа системных потоков. Для Linux реализовано аналогичное приложение на основе POSIX Threads (pthreads), что позволило изучить кросс-платформенный подход и освоить методы мониторинга потоков через файловую систему `/proc` и утилиту `ps`.

Проведенное сравнение показало, что, несмотря на различия в API (WinAPI и pthreads) и инструментах мониторинга (Process Explorer в Windows, `/proc` и `ps` в Linux), фундаментальные принципы управления потоками остаются общими. Приобретённые навыки написания, отладки и анализа многопоточных приложений позволяют создавать эффективные параллельные программы, адаптированные к различным операционным средам, что соответствует современным требованиям разработки программного обеспечения.

**Приложение А** – Листинг А.1

```
#include <windows.h>
#include <stdio.h>

// Функция для получения имени класса приоритета процесса
const char* getProcessPriorityClassName(DWORD priorityClass) {
    switch (priorityClass) {
    case IDLE_PRIORITY_CLASS: return "IDLE_PRIORITY_CLASS (64)";
    case BELOW_NORMAL_PRIORITY_CLASS: return "BELOW_NORMAL_PRIORITY_CLASS
(16384)";
    case NORMAL_PRIORITY_CLASS: return "NORMAL_PRIORITY_CLASS (32)";
    case ABOVE_NORMAL_PRIORITY_CLASS: return "ABOVE_NORMAL_PRIORITY_CLASS
(32768)";
    case HIGH_PRIORITY_CLASS: return "HIGH_PRIORITY_CLASS (128)";
    case REALTIME_PRIORITY_CLASS: return "REALTIME_PRIORITY_CLASS (256)";
    default: return "UNKNOWN";
    }
}

// Функция для получения имени приоритета потока
const char* getThreadPriorityName(int threadPriority) {
    switch (threadPriority) {
    case THREAD_PRIORITY_IDLE: return "THREAD_PRIORITY_IDLE (1)";
    case THREAD_PRIORITY_LOWEST: return "THREAD_PRIORITY_LOWEST (2)";
    case THREAD_PRIORITY_BELOW_NORMAL: return "THREAD_PRIORITY_BELOW_NORMAL
(4)";
    case THREAD_PRIORITY_NORMAL: return "THREAD_PRIORITY_NORMAL (8)";
    case THREAD_PRIORITY_ABOVE_NORMAL: return "THREAD_PRIORITY_ABOVE_NORMAL
(16)";
    case THREAD_PRIORITY_HIGHEST: return "THREAD_PRIORITY_HIGHEST (31)";
    case THREAD_PRIORITY_TIME_CRITICAL: return "THREAD_PRIORITY_TIME_CRITICAL
(63)";
    default: return "UNKNOWN";
    }
}

// Подсчёт количества доступных процессоров в маске
DWORD countBits(DWORD_PTR mask) {
    DWORD count = 0;
    while (mask) {
        if (mask & 1) count++;
        mask >>= 1;
    }
    return count;
}

int main() {
    DWORD processId = GetCurrentProcessId();
    DWORD threadId = GetCurrentThreadId();
    DWORD processPriorityClass = GetPriorityClass(GetCurrentProcess());
    int threadPriority = GetThreadPriority(GetCurrentThread());

    DWORD_PTR processAffinityMask = 0;
    DWORD_PTR systemAffinityMask = 0;
```

```
    HANDLE hProcess = GetCurrentProcess();
    if (!GetProcessAffinityMask(hProcess, &processAffinityMask,
&systemAffinityMask)) {
        printf("Error getting process affinity mask: %lu\n", GetLastError());
    }

    DWORD availableProcessors = countBits(processAffinityMask);
    DWORD currentProcessor = GetCurrentProcessorNumber();

    printf("=== Lab-05a: Process and Thread Info ===\n");
    printf("Process ID: %lu\n", processId);
    printf("Thread ID: %lu\n", threadId);
    printf("Process Priority Class: %s\n",
getProcessPriorityClassName(processPriorityClass));
    printf("Thread Priority: %s\n", getThreadPriorityName(threadPriority));
    printf("Process Affinity Mask: 0x%08lx\n", (unsigned
long)processAffinityMask);
    printf("System Affinity Mask: 0x%08lx\n", (unsigned
long)systemAffinityMask);
    printf("Available Processors: %lu\n", availableProcessors);
    printf("Current Processor: %lu\n", currentProcessor);

    printf("\nPress Enter to exit...\n");
    getchar();
    return 0;
}
```

**Приложение Б** – Листинг Б.1

```
#include <windows.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

// Преобразование числового значения приоритета в класс процесса
DWORD getProcessPriorityClassFromValue(DWORD value) {
    switch (value) {
    case 64: return IDLE_PRIORITY_CLASS;
    case 16384: return BELOW_NORMAL_PRIORITY_CLASS;
    case 32: return NORMAL_PRIORITY_CLASS;
    case 32768: return ABOVE_NORMAL_PRIORITY_CLASS;
    case 128: return HIGH_PRIORITY_CLASS;
    case 256: return REALTIME_PRIORITY_CLASS;
    default: return NORMAL_PRIORITY_CLASS;
    }
}

// Преобразование числового значения в приоритет потока
int getThreadPriorityFromValue(DWORD value) {
    switch (value) {
    case 1: return THREAD_PRIORITY_IDLE;
    case 2: return THREAD_PRIORITY_LOWEST;
    case 4: return THREAD_PRIORITY_BELOW_NORMAL;
    case 8: return THREAD_PRIORITY_NORMAL;
    case 16: return THREAD_PRIORITY_ABOVE_NORMAL;
    case 31: return THREAD_PRIORITY_HIGHEST;
    case 63: return THREAD_PRIORITY_TIME_CRITICAL;
    default: return THREAD_PRIORITY_NORMAL;
    }
}

// Получение имени класса процесса
const char* getProcessPriorityClassName(DWORD priorityClass) {
    switch (priorityClass) {
    case IDLE_PRIORITY_CLASS: return "IDLE_PRIORITY_CLASS (64)";
    case BELOW_NORMAL_PRIORITY_CLASS: return "BELOW_NORMAL_PRIORITY_CLASS
(16384)";
    case NORMAL_PRIORITY_CLASS: return "NORMAL_PRIORITY_CLASS (32)";
    case ABOVE_NORMAL_PRIORITY_CLASS: return "ABOVE_NORMAL_PRIORITY_CLASS
(32768)";
    case HIGH_PRIORITY_CLASS: return "HIGH_PRIORITY_CLASS (128)";
    case REALTIME_PRIORITY_CLASS: return "REALTIME_PRIORITY_CLASS (256)";
    default: return "UNKNOWN";
    }
}

// Получение имени приоритета потока
const char* getThreadPriorityName(int threadPriority) {
    switch (threadPriority) {
    case THREAD_PRIORITY_IDLE: return "THREAD_PRIORITY_IDLE (1)";
    case THREAD_PRIORITY_LOWEST: return "THREAD_PRIORITY_LOWEST (2)";
    case THREAD_PRIORITY_BELOW_NORMAL: return "THREAD_PRIORITY_BELOW_NORMAL
```

```
(4)";
    case THREAD_PRIORITY_NORMAL: return "THREAD_PRIORITY_NORMAL (8)";
    case THREAD_PRIORITY_ABOVE_NORMAL: return "THREAD_PRIORITY_ABOVE_NORMAL
(16)";
    case THREAD_PRIORITY_HIGHEST: return "THREAD_PRIORITY_HIGHEST (31)";
    case THREAD_PRIORITY_TIME_CRITICAL: return "THREAD_PRIORITY_TIME_CRITICAL
(63)";
    default: return "UNKNOWN";
    }
}

// Подсчёт единиц в маске
DWORD countBits(DWORD_PTR mask) {
    DWORD count = 0;
    while (mask) {
        if (mask & 1) count++;
        mask >>= 1;
    }
    return count;
}

// Функция потока
DWORD WINAPI threadFunction(LPVOID param) {
    struct ThreadParams {
        int threadId;
        int threadPriority;
        DWORD_PTR affinityMask;
    };

    struct ThreadParams* p = (struct ThreadParams*)param;
    int threadId = p->threadId;
    int threadPriority = p->threadPriority;
    DWORD_PTR affinityMask = p->affinityMask;

    HANDLE currentThread = GetCurrentThread();
    SetThreadAffinityMask(currentThread, affinityMask);
    SetThreadPriority(currentThread, threadPriority);

    const int TOTAL_ITERATIONS = 1000000;
    const int OUTPUT_INTERVAL = 1000;

    DWORD processId = GetCurrentProcessId();
    DWORD currentThreadId = GetCurrentThreadId();

    clock_t startTime = clock();

    for (int i = 1; i <= TOTAL_ITERATIONS; i++) {
        if (i % OUTPUT_INTERVAL == 0) {
            DWORD processorNumber = GetCurrentProcessorNumber();
            printf("--- Thread %d ---\n", threadId);
            printf("Iteration: %d\n", i);
            printf("Process ID: %lu\n", processId);
            printf("Thread ID: %lu\n", currentThreadId);
            printf("Process Priority Class: %s\n",
```

```
getProcessPriorityClassName(GetPriorityClass(GetCurrentProcess())));
            printf("Thread Priority: %s\n",
getThreadPriorityName(GetThreadPriority(currentThread)));
            printf("Current Processor: %lu\n", processorNumber);
            printf("Affinity Mask: 0x%lx\n\n", (unsigned long)affinityMask);
            Sleep(200); // задержка 200 мс
        }

        // нагрузка
        volatile double calc = 0.0;
        for (int j = 0; j < 50; j++)
            calc += sin(i * 0.001) * cos(j * 0.001);
    }

    clock_t endTime = clock();
    double elapsedSec = (double)(endTime - startTime) / CLOCKS_PER_SEC;
    printf("Thread %d finished! Time elapsed: %.2f seconds.\n", threadId,
elapsedSec);
    return 0;
}

int main(int argc, char* argv[]) {
    if (argc != 5) {
        printf("Usage: %s P1 P2 P3 P4\n", argv[0]);
        printf("  P1 - process affinity mask (integer)\n");
        printf("  P2 - process priority class (integer)\n");
        printf("  P3 - priority of first child thread (integer)\n");
        printf("  P4 - priority of second child thread (integer)\n");
        return 1;
    }

    DWORD_PTR affinityMask = strtoul(argv[1], NULL, 10);
    DWORD processPriority = getProcessPriorityClassFromValue(strtoul(argv[2],
NULL, 10));
    int threadPriority1 = getThreadPriorityFromValue(strtoul(argv[3], NULL,
10));
    int threadPriority2 = getThreadPriorityFromValue(strtoul(argv[4], NULL,
10));

    printf("=== Application Parameters ===\n");
    printf("Process Affinity Mask: 0x%lx\n", (unsigned long)affinityMask);
    printf("Process Priority Class: %s\n",
getProcessPriorityClassName(processPriority));
    printf("First Thread Priority: %s\n",
getThreadPriorityName(threadPriority1));
    printf("Second Thread Priority: %s\n\n",
getThreadPriorityName(threadPriority2));

    HANDLE currentProcess = GetCurrentProcess();
    SetProcessAffinityMask(currentProcess, affinityMask);
    SetPriorityClass(currentProcess, processPriority);

    printf("=== Launching child threads ===\n");
```

```
    struct ThreadParams t1Params = {1, threadPriority1, affinityMask};
    struct ThreadParams t2Params = {2, threadPriority2, affinityMask};

    HANDLE hThread1 = CreateThread(NULL, 0, threadFunction, &t1Params, 0,
NULL);
    Sleep(50);
    HANDLE hThread2 = CreateThread(NULL, 0, threadFunction, &t2Params, 0,
NULL);

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    printf("\nAll threads finished execution.\nPress Enter to exit...");
    getchar();
    return 0;
}
```

**Приложение В** – Листинг В.1

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    if (argc != 4) {
        printf("Usage: %s P1 P2 P3\n", argv[0]);
        printf("  P1 - process affinity mask (integer)\n");
        printf("  P2 - priority class of first child process (integer)\n");
        printf("  P3 - priority class of second child process (integer)\n");
        printf("Example: %s 15 32 128\n", argv[0]);
        return 1;
    }

    DWORD_PTR affinityMask = strtoul(argv[1], NULL, 10);
    DWORD priority1 = strtoul(argv[2], NULL, 10);
    DWORD priority2 = strtoul(argv[3], NULL, 10);

    printf("=== Application Parameters ===\n");
    printf("Process Affinity Mask: 0x%lx\n", (unsigned long)affinityMask);
    printf("Priority Class of first process: %lu\n", (unsigned
long)priority1);
    printf("Priority Class of second process: %lu\n\n", (unsigned
long)priority2);

    HANDLE currentProcess = GetCurrentProcess();
    if (!SetProcessAffinityMask(currentProcess, affinityMask)) {
        printf("Failed to set process affinity mask. Error: %lu\n",
GetLastError());
    } else {
        printf("Process affinity mask successfully set.\n");
    }

    const char* childApp = "Lab-05x.exe"; // имя дочернего приложения

    char cmdLine1[256];
    char cmdLine2[256];

    snprintf(cmdLine1, sizeof(cmdLine1), "%s %lu %lu 8 8", childApp, (unsigned
long)affinityMask, (unsigned long)priority1);
    snprintf(cmdLine2, sizeof(cmdLine2), "%s %lu %lu 8 8", childApp, (unsigned
long)affinityMask, (unsigned long)priority2);

    STARTUPINFOA si1 = {0};
    si1.cb = sizeof(si1);
    PROCESS_INFORMATION pi1;

    STARTUPINFOA si2 = {0};
    si2.cb = sizeof(si2);
    PROCESS_INFORMATION pi2;

    printf("\nLaunching first child process...\n");
    if (CreateProcessA(NULL, cmdLine1, NULL, NULL, FALSE, CREATE_NEW_CONSOLE,
```

```
NULL, NULL, &si1, &pi1)) {
        printf("First process started. PID: %lu\n", (unsigned
long)pi1.dwProcessId);
    } else {
        printf("Failed to start first process. Error: %lu\n", GetLastError());
    }

    printf("\nLaunching second child process...\n");
    if (CreateProcessA(NULL, cmdLine2, NULL, NULL, FALSE, CREATE_NEW_CONSOLE,
NULL, NULL, &si2, &pi2)) {
        printf("Second process started. PID: %lu\n", (unsigned
long)pi2.dwProcessId);
    } else {
        printf("Failed to start second process. Error: %lu\n",
GetLastError());
    }

    printf("\nBoth child processes are running in separate console
windows.\n");

    WaitForSingleObject(pi1.hProcess, INFINITE);
    WaitForSingleObject(pi2.hProcess, INFINITE);

    CloseHandle(pi1.hThread);
    CloseHandle(pi1.hProcess);
    CloseHandle(pi2.hThread);
    CloseHandle(pi2.hProcess);

    printf("Both child processes have finished execution.\nPress Enter to
exit...");
    getchar();
    return 0;
}
```

**Приложение Г** – Листинг Г.1

```
#include <windows.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

// Преобразование числового значения приоритета в системные константы
DWORD getProcessPriorityClass(DWORD priorityValue) {
    switch (priorityValue) {
        case 64: return IDLE_PRIORITY_CLASS;
        case 16384: return BELOW_NORMAL_PRIORITY_CLASS;
        case 32: return NORMAL_PRIORITY_CLASS;
        case 32768: return ABOVE_NORMAL_PRIORITY_CLASS;
        case 128: return HIGH_PRIORITY_CLASS;
        case 256: return REALTIME_PRIORITY_CLASS;
        default: return NORMAL_PRIORITY_CLASS;
    }
}

int getThreadPriority(DWORD priorityValue) {
    switch (priorityValue) {
        case 1: return THREAD_PRIORITY_IDLE;
        case 2: return THREAD_PRIORITY_LOWEST;
        case 4: return THREAD_PRIORITY_BELOW_NORMAL;
        case 8: return THREAD_PRIORITY_NORMAL;
        case 16: return THREAD_PRIORITY_ABOVE_NORMAL;
        case 31: return THREAD_PRIORITY_HIGHEST;
        case 63: return THREAD_PRIORITY_TIME_CRITICAL;
        default: return THREAD_PRIORITY_NORMAL;
    }
}

// Функция потока
DWORD WINAPI threadFunction(LPVOID param) {
    int threadId = ((int*)param)[0];
    int threadPriority = ((int*)param)[1];
    DWORD_PTR affinityMask = ((DWORD_PTR*)param)[2];

    HANDLE currentThread = GetCurrentThread();
    SetThreadAffinityMask(currentThread, affinityMask);
    SetThreadPriority(currentThread, threadPriority);

    const int TOTAL_ITERATIONS = 1000000;
    const int OUTPUT_INTERVAL = 1000;
    DWORD processId = GetCurrentProcessId();
    DWORD currentThreadId = GetCurrentThreadId();

    clock_t startTime = clock();

    for (int i = 1; i <= TOTAL_ITERATIONS; i++) {
        if (i % OUTPUT_INTERVAL == 0) {
            DWORD processorNumber = GetCurrentProcessorNumber();
            printf("--- Thread %d ---\n", threadId);
            printf("Iteration: %d\n", i);
```

```
            printf("Process ID: %lu\n", (unsigned long)processId);
            printf("Thread ID: %lu\n", (unsigned long)currentThreadId);
            printf("Process Priority: %lu\n", (unsigned
long)GetPriorityClass(GetCurrentProcess()));
            printf("Thread Priority: %d\n", GetThreadPriority(currentThread));
            printf("Current Processor: %lu\n", (unsigned
long)processorNumber);
            printf("Affinity Mask: 0x%lx\n\n", (unsigned long)affinityMask);
            Sleep(100); // задержка 100 мс
        }

        // Простая нагрузка
        volatile double calc = 0.0;
        for (int j = 0; j < 50; j++)
            calc += sin(i * 0.001) * cos(j * 0.001);
    }

    clock_t endTime = clock();
    double elapsedSec = (double)(endTime - startTime) / CLOCKS_PER_SEC;
    printf("Thread %d finished. Time elapsed: %.2f seconds\n", threadId,
elapsedSec);

    return 0;
}

int main(int argc, char* argv[]) {
    if (argc != 5) {
        printf("Usage: %s P1 P2 P3 P4\n", argv[0]);
        return 1;
    }

    DWORD_PTR affinityMask = strtoul(argv[1], NULL, 10);
    DWORD processPriority = getProcessPriorityClass(strtoul(argv[2], NULL,
10));
    int threadPriority1 = getThreadPriority(strtoul(argv[3], NULL, 10));
    int threadPriority2 = getThreadPriority(strtoul(argv[4], NULL, 10));

    HANDLE currentProcess = GetCurrentProcess();
    SetProcessAffinityMask(currentProcess, affinityMask);
    SetPriorityClass(currentProcess, processPriority);

    int param1[3] = {1, threadPriority1, (int)affinityMask};
    int param2[3] = {2, threadPriority2, (int)affinityMask};

    HANDLE hThread1 = CreateThread(NULL, 0, threadFunction, param1, 0, NULL);
    Sleep(50); // задержка перед запуском второго потока
    HANDLE hThread2 = CreateThread(NULL, 0, threadFunction, param2, 0, NULL);

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    CloseHandle(hThread1);
    CloseHandle(hThread2);
```

```
    printf("\nAll threads finished.\nPress Enter to exit...");
    getchar();
    return 0;
}
```

**Приложение Д** – Листинг Д.1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sched.h>
#include <errno.h>

int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf("Использование: %s <P1: маска (игнорируется)> <P2:
nice-значение>\n", argv[0]);
        return 1;
    }

    int niceValue = atoi(argv[2]);

    // Установка приоритета процесса (nice)
    if (setpriority(PRIO_PROCESS, 0, niceValue) != 0) {
        perror("Ошибка установки приоритета (setpriority)");
    }

    pid_t processId = getpid();
    pid_t threadId = syscall(SYS_gettid);

    errno = 0;
    int priority = getpriority(PRIO_PROCESS, 0);
    if (errno != 0) {
        perror("Ошибка получения приоритета");
    }

    int processorCount = sysconf(_SC_NPROCESSORS_ONLN);

    int currentProcessor = sched_getcpu();
    if (currentProcessor == -1) {
        perror("sched_getcpu");
    }

    printf("Идентификатор процесса (PID): %d\n", processId);
    printf("Идентификатор потока (TID): %d\n", threadId);
    printf("Приоритет процесса (nice): %d\n", priority);
    printf("Доступно процессоров: %d\n", processorCount);
    printf("Номер текущего процессора: %d\n", currentProcessor);

    return 0;
}
```

**Приложение Е** – Листинг Е.1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sched.h>
#include <time.h>
#include <errno.h>

int main() {
    pid_t processId = getpid();
    pid_t threadId = syscall(SYS_gettid);

    const int totalIterations = 1000000;
    const int delayInterval = 1000;

    // Засекаем время начала
    clock_t startTime = clock();

    for (int i = 1; i <= totalIterations; i++) {
        if (i % delayInterval == 0) {
            // Задержка 200 мс
            usleep(200 * 1000);

            errno = 0;
            int niceValue = getpriority(PRIO_PROCESS, 0);
            if (errno != 0) {
                perror("Ошибка получения приоритета");
                niceValue = 0;
            }

            int currentCPU = sched_getcpu();
            if (currentCPU == -1) {
                perror("sched_getcpu");
                currentCPU = -1;
            }

            printf("Итерация: %d\n", i);
            printf("  PID: %d\n", processId);
            printf("  TID: %d\n", threadId);
            printf("  Уровень любезности (nice): %d\n", niceValue);
            printf("  Назначенный процессор: %d\n", currentCPU);
            printf("-----------------------------\n");
        }

        // Простая нагрузка
        volatile double calc = 0.0;
        for (int j = 0; j < 50; j++)
            calc += i * 0.001 * j * 0.001; // имитация вычислений
    }

    // Засекаем время окончания
    clock_t endTime = clock();
    double elapsedSeconds = (double)(endTime - startTime) / CLOCKS_PER_SEC;
    int minutes = (int)elapsedSeconds / 60;
    double seconds = elapsedSeconds - minutes * 60;

    printf("Время выполнения: %d мин %.2f сек\n", minutes, seconds);

    return 0;}
```

**Приложение Ж** – Листинг Ж.1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/resource.h>
#include <sched.h>
#include <locale.h>
#include <errno.h>

static pid_t launch_child(unsigned long long affinityMask,
                          int niceValue,
                          const char *childPath)
{
    pid_t pid = fork();
    if (pid == 0) {
        /* Установка nice */
        if (setpriority(PRIO_PROCESS, 0, niceValue) != 0) {
            perror("setpriority");
        }

        /* Установка маски процессоров */
        cpu_set_t mask;
        CPU_ZERO(&mask);
        for (int i = 0; i < 64; i++) {
            if (affinityMask & (1ULL << i)) {
                CPU_SET(i, &mask);
            }
        }

        if (sched_setaffinity(0, sizeof(mask), &mask) != 0) {
            perror("sched_setaffinity");
        }

        /* Запуск Lab-05x */
        execl(childPath, childPath, (char *)NULL);

        perror("execl");
        _exit(1);
    }

    return pid;
}

int main(int argc, char *argv[])
{
    setlocale(LC_ALL, "ru_RU.UTF-8");

    if (argc != 4) {
        fprintf(stderr,
                    "Использование: %s <P1: маска CPU> <P2: nice 1> <P3: nice
2>\n",
                argv[0]);
        return 1;
    }

    /* Разбор аргументов */
    unsigned long long affinityMask = strtoull(argv[1], NULL, 10);
    int nice1 = atoi(argv[2]);
    int nice2 = atoi(argv[3]);
```

```
    printf("Маска процессоров (P1): %llu\n", affinityMask);
    printf("Приоритет дочернего процесса 1 (P2): %d\n", nice1);
    printf("Приоритет дочернего процесса 2 (P3): %d\n", nice2);

    const char *childPath = "./Lab_05x";

    /* Запуск дочерних процессов */
    pid_t pid1 = launch_child(affinityMask, nice1, childPath);
    pid_t pid2 = launch_child(affinityMask, nice2, childPath);

    printf("Процесс 1 запущен: PID = %d\n", pid1);
    printf("Процесс 2 запущен: PID = %d\n", pid2);

    /* Ожидание завершения */
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    printf("Оба дочерних процесса завершены.\n");
    return 0;
}
```

**Приложение 3** – Листинг 3.1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <sys/resource.h>
#include <sched.h>
#include <locale.h>
#include <errno.h>
#include <time.h>

#define MAX_ITERATIONS 1000000

volatile int counter1 = 0;
volatile int counter2 = 0;

typedef struct {
    int threadId;
    int niceValue;
} thread_args_t;

void *thread_function(void *arg)
{
    thread_args_t *args = (thread_args_t *)arg;

    pid_t tid = syscall(SYS_gettid);

    /* Установка nice (фактически процессный, что важно для эксперимента) */
    if (setpriority(PRIO_PROCESS, 0, args->niceValue) != 0) {
        perror("setpriority");
    }

    int niceValue = getpriority(PRIO_PROCESS, 0);

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

        if (i % 1000 == 0) {
            int cpu = sched_getcpu();

            printf("[Поток %d]\n", args->threadId);
            printf("  Итерация: %d\n", i);
            printf("  TID: %d\n", tid);
            printf("  Nice: %d\n", niceValue);
            printf("  CPU: %d\n", cpu);
            printf("-----------------------------\n");

            usleep(200000); /* 200 мс */
        }

        if (args->threadId == 1)
            counter1++;
        else
            counter2++;
    }

    return NULL;
}

int main(int argc, char *argv[])
{
    setlocale(LC_ALL, "ru_RU.UTF-8");
```

```
    if (argc != 4) {
        fprintf(stderr,
                  "Использование: %s <P1: маска CPU> <P2: nice потока 1> <P3:
nice потока 2>\n",
                argv[0]);
        return 1;
    }

    unsigned long long affinityMask = strtoull(argv[1], NULL, 10);
    int nice1 = atoi(argv[2]);
    int nice2 = atoi(argv[3]);

    printf("Маска процессоров (P1): %llu\n", affinityMask);
    printf("Приоритет потока 1 (P2): %d\n", nice1);
    printf("Приоритет потока 2 (P3): %d\n", nice2);

    /* Установка affinity для процесса (общая для всех потоков) */
    cpu_set_t mask;
    CPU_ZERO(&mask);
    for (int i = 0; i < 64; i++) {
        if (affinityMask & (1ULL << i)) {
            CPU_SET(i, &mask);
        }
    }

    if (sched_setaffinity(0, sizeof(mask), &mask) != 0) {
        perror("sched_setaffinity");
    }

    pthread_t t1, t2;
    thread_args_t args1 = {1, nice1};
    thread_args_t args2 = {2, nice2};

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    /* Создание потоков */
    pthread_create(&t1, NULL, thread_function, &args1);
    pthread_create(&t2, NULL, thread_function, &args2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    clock_gettime(CLOCK_MONOTONIC, &end);

    double elapsed =
        (end.tv_sec - start.tv_sec) +
        (end.tv_nsec - start.tv_nsec) / 1e9;

    int minutes = (int)(elapsed / 60);
    double seconds = elapsed - minutes * 60;

    printf("Поток 1 завершён. Итераций: %d\n", counter1);
    printf("Поток 2 завершён. Итераций: %d\n", counter2);
    printf("Общее время выполнения: %d мин %.2f сек\n",
           minutes, seconds);

    return 0;
}
```