Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системы программирования

ОТЧЁТ

к лабораторной работе №5

на тему

**УПРАВЛЕНИЕ ПОТОКАМИ, СРЕДСТВА СИНХРОНИЗАЦИИ**

|  |
| --- |
| Выполнил: студент гр. 253503  Минич С.В. |
| Проверил: ассистент кафедры информатики Гриценко Н.Ю. |

Минск 2025

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc189588542)

[2 Описание работы программы 4](#_Toc189588543)

[2.1 Инициализация параметров 4](#_Toc189588544)

[2.2 Основная работа программы 4](#_Toc189588545)

[2.3 Вывод результатов 4](#_Toc189588546)

[3 Ход выполнения программы 5](#_Toc189588547)

[3.1 Примеры выполнения задания 5](#_Toc189588548)

[Заключение 7](#_Toc189588549)

[Список литературных источников 8](#_Toc189588550)

[Приложение А (обязательно) Исходный код 9](#_Toc189588551)

**1 ЦЕЛЬ РАБОТЫ**

Целью данной лабораторной работы является углубленное изучение и практическое освоение подсистемы потоков (*pthread*) в операционных системах семейства *Unix/Linux*. Особое внимание уделяется пониманию основных механизмов функционирования и управления потоками, а также средствам их синхронизации и взаимодействия в многопоточной среде. В рамках работы предстоит спроектировать, реализовать и отладить программe с параллельными взаимодействующими (конкурирующими) потоками, что позволит на практике освоить концепции многопоточного программирования.

Программная реализация должна демонстрировать эффективное использование потоков для решения вычислительно-интенсивных задач, таких как параллельная сортировка массивов или многопоточная работа с файлами. Предстоит разработать решения, которые разбивают единую задачу на несколько параллельно выполняемых подзадач, обрабатываемых отдельными потоками, с последующей синхронизацией результатов. Особое внимание необходимо уделить корректному использованию средств синхронизации, что позволит избежать типичных проблем многопоточного программирования, таких как гонки данных и взаимные блокировки.

Выполнение данной лабораторной работы направлено на формирование практических навыков многопоточного программирования в среде *Unix/Linux*, понимание внутренних механизмов параллельных вычислений, освоение *API* потоков *POSIX* (*pthread*), развитие умений проектирования и отладки многопоточных приложений, а также подготовку к решению реальных задач в современных многоядерных вычислительных системах, где эффективное использование параллелизма становится ключевым фактором производительности.

**2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ**

* 1. **Инициализация параметров**

Программа реализована на языке *C* и представляет собой стресс-тест для системы с использованием многопоточной обработки. В начале программы подключаются необходимые библиотеки и определяются ключевые константы: *NUM\_THREADS* (количество создаваемых потоков), *FIB\_LIMIT* (чрезвычайно большое число для вычисления числа Фибоначчи) и *NUM\_ITERATIONS* (количество итераций вычислений). Эти параметры обеспечивают длительную нагрузку на процессор для оценки производительности и стабильности системы в условиях интенсивных вычислений [1].

Программный код представлен в приложении А.

**2.2 Основная работа программы**

Основная функциональность программы заключается в создании множества параллельных потоков, каждый из которых выполняет вычислительно-интенсивную задачу. Функция *fibonacci(n)* реализует итеративный алгоритм вычисления числа Фибоначчи для минимизации накладных расходов на стеке вызовов. Каждый запускаемый поток выполняет функцию *perform\_work*, которая принимает номер потока в качестве аргумента и в цикле *NUM\_ITERATIONS* раз вызывает функцию *fibonacci* с параметром *FIB\_LIMIT*.

Главная функция *main* создает массив *pthread\_t* для хранения идентификаторов потоков и массив *thread\_nums* для хранения их порядковых номеров. Затем в цикле создаются *NUM\_THREADS* потоков с использованием *pthread\_create*, каждому передается указатель на его порядковый номер. После создания всех потоков программа ожидает их завершения с помощью *pthread\_join* для каждого потока, обеспечивая корректное завершение программы только после окончания работы всех потоков.

**2.3 Вывод результатов**

Каждый поток по завершении своих вычислений выводит сообщение, позволяя отслеживать прогресс выполнения задачи. После завершения всех потоков главная функция выводит сообщение, сигнализируя об успешном завершении стресс-теста. Программа не сохраняет результаты вычислений в файл, поскольку основная цель – создание вычислительной нагрузки, а не получение конкретных результатов.

**3 ХОД ВЫПОЛНЕНИЯ ПРОГРАММЫ**

**3.1 Примеры выполнения задания**

Пример сборки и запуска теста представлен на рисунке 3.1:

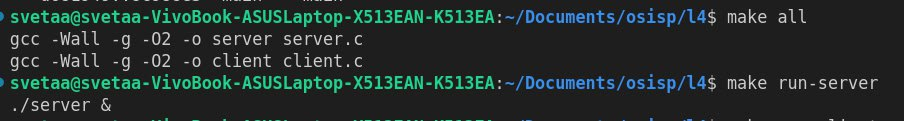


Рисунок 3.1 –Сборка проекта

Результат нагрузки на систему в момент работы данного стресс-теста представлен на рисунке 3.2.

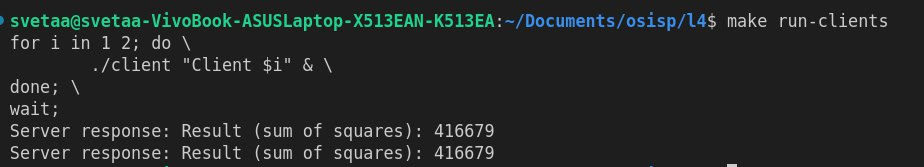


Рисунок 3.2 – Нагрузка на систему

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения лабораторной работы была разработана и реализована многопоточная программа на языке *C*, демонстрирующая механизмы управления потоками и их синхронизации в операционных системах *Linux*. В процессе работы были освоены ключевые аспекты многопоточного программирования: создание и управление потоками через *API pthread*, передача параметров потокам, ожидание завершения потоков, а также организация параллельных вычислений для интенсивной нагрузки процессора.

Программа *stress\_test.c* успешно реализует стресс-тестирование системы с использованием 16 параллельных потоков, каждый из которых выполняет вычислительно-ёмкую задачу – итеративное вычисление числа Фибоначчи для экстремально большого значения в многократном цикле. Используемый итеративный алгоритм Фибоначчи минимизирует накладные расходы, типичные для рекурсивных реализаций, что позволяет сосредоточиться именно на эффективности параллельного выполнения. Каждый поток получает уникальный идентификатор, который выводится при завершении работы, что обеспечивает возможность отслеживания прогресса выполнения задачи.

Реализованный механизм передачи параметров потокам через указатели демонстрирует практические аспекты работы с разделяемыми данными в многопоточной среде. Структура программы с использованием *pthread\_create* для запуска потоков и *pthread\_join* для ожидания их завершения иллюстрирует типичный паттерн параллельного программирования, широко применяемый в высокопроизводительных вычислениях.

Таким образом, работа позволила получить практический опыт в области многопоточного программирования с использованием *POSIX Threads* (*pthread*), организации параллельных вычислений, передачи параметров между потоками и синхронизации их завершения. Полученные навыки являются фундаментальными для разработки эффективных многопоточных приложений в современных многоядерных системах, где параллельная обработка становится ключевым фактором производительности.

**СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ**

[1] Числа Фибоначчи [Электронный ресурс]. – Режим доступа: https://younglinux.info/algorithm/fibonacci.

ПРИЛОЖЕНИЕ А  
(обязательно)  
Исходный код

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdint.h>

#define NUM\_THREADS 16

#define FIB\_LIMIT 100000000000

#define NUM\_ITERATIONS 100000000000

uint64\_t fibonacci(uint64\_t n) {

    if (n <= 1) return n;

    uint64\_t a = 0, b = 1, c;

    for (uint64\_t i = 2; i <= n; ++i) {

        c = a + b;

        a = b;

        b = c;

    }

    return b;

}

void\* perform\_work(void\* arg) {

    int thread\_num = \*((int\*)arg);

    uint64\_t fib\_result;

    for (int i = 0; i < NUM\_ITERATIONS; ++i) {

        fib\_result = fibonacci(FIB\_LIMIT);

    }

    printf("Thread %d finished\n", thread\_num);

    return NULL;

}

int main() {

    pthread\_t threads[NUM\_THREADS];

    int thread\_nums[NUM\_THREADS];

    for (int i = 0; i < NUM\_THREADS; ++i) {

        thread\_nums[i] = i;

        int err = pthread\_create(&threads[i], NULL, perform\_work, &thread\_nums[i]);

        if (err) {

            printf("Error creating thread %d\n", i);

            return 1;

        }

    }

    for (int i = 0; i < NUM\_THREADS; ++i) {

        pthread\_join(threads[i], NULL);

    }

    printf("All threads finished\n");

    return 0;

}