МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

## Факультет информационных технологий и робототехники

Кафедра программного обеспечения информационных систем и технологий

**Отчет по лабораторной работе № 23**

по дисциплине:” Системное программирование”

на тему:***” Многозадачность потоков, автосборка сложных приложений”***

Вариант 8

Выполнил**:** студент группы 10701222 Хасаншин А.В Демянов Л.С

Приняла**:** пр. Давыденко Н.В.

Минск 2024

# Лабораторная работа № #23. Многозадачность потоков, автосборка сложных приложений.

Цель работы: Изучить понятие потоков и их особенности. Создание и завершение потока. Синхронизация потоков. Получение информации о потоках. Обмен данными между потоками.

Изучить встроенный инструментарий для разработки приложений под семейство ОС Linux и фундаментальные основы системного программирования с использованием компиляторов gcc/g++, отладчика gdb и других для проектирования, компиляции, отладки и запуска приложений на языке программирования С/C++.

Научиться эффективно использовать специальные средства для автоматизации процесса компиляции, сборки и запуска многофайловых проектов.

## Задание 1.

Проведите рефакторинг программы, разработанной ранее по заданию No3 лабораторной работы No21 «Концепция «Файл» и операции ввода-вывода, создание программы», руководствуясь принципом Single Responsibility Principle.

Основные дополнительные требования к новой программе:

• Проект программы должен быть многофайловам.

• Каждая сущность приложения должна быть представлено отдельным

исходным файлом.

• Все файлы могут располагаться в одной директории.

• Объявленные функции должны храниться в заголовочных файлах.

• Расчет сложной функции произвести с помощью отдельных несинхронизированных потоков.

## Задание 2.

Проведите рефакторинг программы из 1-го задания так чтобы синхронизировать соответствующие потоки. Соберите и запустите приложение, сравните результаты с результатами предыдущей программы. Дайте объяснения.

## Задание 3.

Проведите компиляцию многофайлового проекта с прохождением всех стадий компиляции. Для ускорения работы примените утилиту make.Запустите и сравните полученный результат с результатом программы из лабораторной работы No21.Исходные файлы программ обязательно должны содержать комментарии. Сборка проекта должна содержать файлы с результатами препроцессинга. Исследуйте фалы препроцессора, найдите в них код своей программы.Определите размеры исходных, препроцессорных, ассемблерных, объектных и исполняемых файлов. С помощью соответствующий консольных команд определите форматы этих файлов.

**Код Программы:**

**1. main.cpp**

#include <iostream>

#include <cmath>

#include <thread>

#include "math\_operations.h"

#include "file\_operations.h"

#include "timer.h"

using namespace std;

int main() {

// Ввод данных от пользователя

double startX, endX, stepX;

cout << "Введите начальное значение x: ";

cin >> startX;

cout << "Введите конечное значение x: ";

cin >> endX;

cout << "Введите шаг: ";

cin >> stepX;

// Запуск таймера для измерения времени выполнения программы

Timer timer;

// Создание потока для выполнения расчетов и записи данных в файл

thread calculationThread(= {

// Вычисление значений функции на заданном интервале

vector<pair<double, double>> results = calculateValues(startX, endX, stepX);

// Запись результатов в файл

writeToFile("function\_data.txt", results);

});

// Ожидание завершения работы потока

calculationThread.join();

// Вывод времени выполнения программы

cout << "Время выполнения программы: " << timer.elapsed() << " секунд" << endl;

// Инструкция для пользователя по визуализации данных в GNUplot

cout << "Используйте следующие команды в GNUplot для построения графика:" << endl;

cout << "set title 'График функции y = sin(5x + 1) - cos(x)'" << endl;

cout << "set xlabel 'x'" << endl;

cout << "set ylabel 'y'" << endl;

cout << "plot 'function\_data.txt' using 1:2 with lines title 'y = sin(5x + 1) - cos(x)'" << endl;

return 0;

}

**2. math\_operations.cpp**

#include "math\_operations.h"

#include <cmath>

#include <vector>

using namespace std;

// Функция для вычисления значения y = sin(5x + 1) - cos(x)

double calculateY(double x) {

return sin(5 \* x + 1) - cos(x);

}

// Функция для вычисления значений функции на интервале [startX, endX] с шагом stepX

vector<pair<double, double>> calculateValues(double startX, double endX, double stepX) {

vector<pair<double, double>> values;

for (double x = startX; x <= endX; x += stepX) {

double y = calculateY(x);

values.push\_back(make\_pair(x, y));

}

return values;

}

**3. math\_operations.h**

#pragma once

#include <vector>

// Объявление функции для вычисления значений функции на интервале

std::vector<std::pair<double, double>> calculateValues(double startX, double endX, double stepX);

// Объявление функции для вычисления значения y

double calculateY(double x);

**4. file\_operations.cpp**

#include "file\_operations.h"

#include <fstream>

#include <mutex>

std::mutex fileMutex; // Мьютекс для синхронизации записи в файл

// Функция для записи значений в файл

void writeToFile(const std::string& filename, const std::vector<std::pair<double, double>>& values) {

std::lock\_guard<std::mutex> guard(fileMutex); // Блокировка мьютекса

std::ofstream outFile(filename);

if (!outFile.is\_open()) {

throw std::runtime\_error("Ошибка открытия файла!");

}

for (const auto& pair : values) {

outFile << pair.first << " " << pair.second << std::endl;

}

outFile.close();

}

**5. file\_operations.h**

#pragma once

#include <vector>

#include <string>

#include <mutex>

// Объявление внешнего мьютекса

extern std::mutex fileMutex;

// Объявление функции для записи значений в файл

void writeToFile(const std::string& filename, const std::vector<std::pair<double, double>>& values);

**6. timer.cpp**

#include "timer.h"

#include <chrono>

using namespace std::chrono;

// Конструктор класса Timer, который сохраняет текущее время

Timer::Timer() {

startTime = high\_resolution\_clock::now();

}

// Функция для вычисления времени, прошедшего с момента создания объекта Timer

double Timer::elapsed() {

auto endTime = high\_resolution\_clock::now();

duration<double> duration = endTime - startTime;

return duration.count();

}

**7. timer.h**

#pragma once

#include <chrono>

// Класс Timer для измерения времени выполнения программы

class Timer {

public:

Timer();

double elapsed();

private:

std::chrono::time\_point<std::chrono::high\_resolution\_clock> startTime;

};

**8. Makefile**

CXX = g++

CXXFLAGS = -std=c++17 -Wall -pthread

SRC = main.cpp math\_operations.cpp file\_operations.cpp timer.cpp

OBJ = $(SRC:.cpp=.o)

**all: program**

**program: $(OBJ)**

$(CXX) $(CXXFLAGS) -o program $(OBJ)

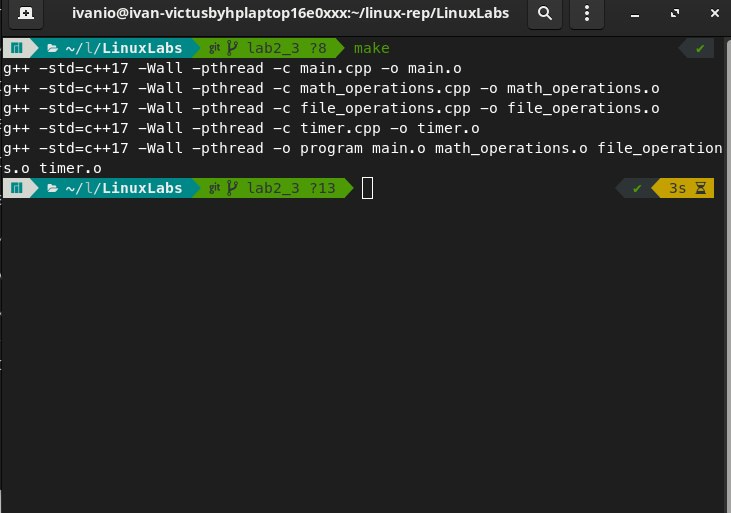
**%.o: %.cpp**

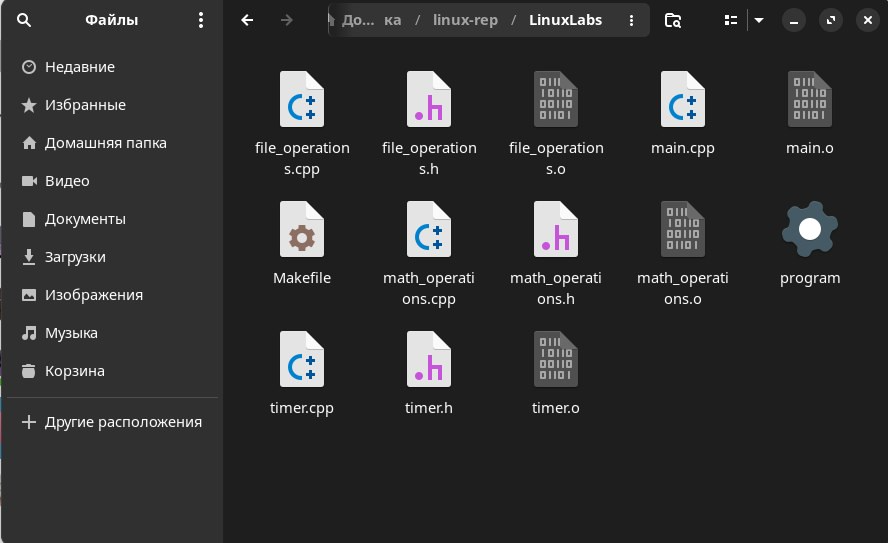
$(CXX) $(CXXFLAGS) -c $< -o $@

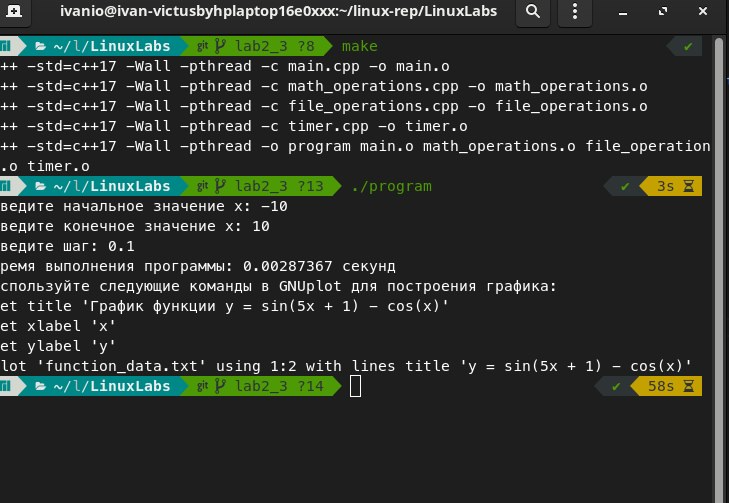
**clean:**

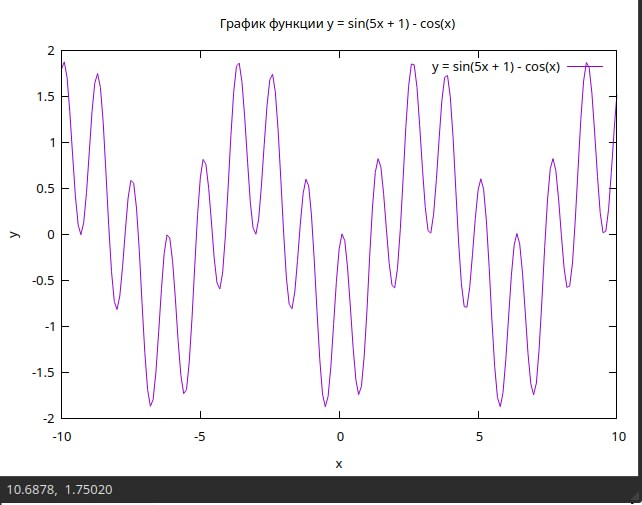
rm -f \*.o program

**Результаты\_работы:**









**Исходные файлы программы**

Исходные файлы программы — это текстовые файлы, содержащие исходный код, написанный на языке программирования. Эти файлы являются основой для создания программного обеспечения и включают в себя инструкции, которые разработчики пишут для выполнения определенных задач.

**Исполняемые файлы программы**

Исполняемые файлы программы — это файлы, которые содержат машинный код, готовый к выполнению процессором компьютера. Эти файлы создаются в результате компиляции исходных файлов и могут быть запущены непосредственно операционной системой.

**Основные стадии компиляции:**

1. **Лексический анализ (Lexical Analysis)**:
   * Разбивает исходный код на лексемы (токены), такие как ключевые слова, идентификаторы, числа и символы.
   * Удаляет пробелы и комментарии.
   * Проверяет корректность лексем.
2. **Синтаксический анализ (Syntax Analysis)**:
   * Строит синтаксическое дерево (дерево разбора) на основе токенов.
   * Проверяет соответствие кода грамматике языка программирования.
3. **Семантический анализ (Semantic Analysis)**:
   * Проверяет смысловую корректность кода.
   * Выполняет проверки типов данных, области видимости переменных и другие семантические правила.
4. **Генерация промежуточного кода (Intermediate Code Generation)**:
   * Преобразует синтаксическое дерево в промежуточное представление, которое проще оптимизировать и преобразовать в машинный код.
5. **Оптимизация кода (Code Optimization)**:
   * Улучшает промежуточный код для повышения производительности и уменьшения размера исполняемого файла.
   * Примеры оптимизаций включают удаление мертвого кода, сворачивание констант и другие.
6. **Генерация кода (Code Generation)**:
   * Преобразует оптимизированный промежуточный код в машинный код, который может быть выполнен процессором.
   * Создает объектные файлы.
7. **Сборка (Linking)**:
   * Объединяет объектные файлы и библиотеки в один исполняемый файл.
   * Разрешает внешние ссылки и адреса функций и переменных.

Вывод: Были изучены многозадачность потоков и автосборка сложных проектов в Ubuntu.

## Контрольные вопросы

1. ***Что такое утилита make?***

Утилита make – это мощный инструмент, широко используемый в программировании, особенно при работе с крупными проектами, состоящими из множества файлов. Она позволяет автоматизировать процесс сборки (компиляции, линковки и других действий) программного продукта.

Основная идея работы make

1)Makefile: В основе работы make лежит специальный файл, называемый Makefile. В этом файле описываются правила сборки проекта, то есть какие файлы необходимо перекомпилировать, в какой последовательности и какие команды для этого использовать.

2)Анализ зависимостей: Make анализирует Makefile и определяет, какие файлы были изменены с момента последней сборки.

3)Выполнение команд: На основе анализа зависимостей make выполняет только те команды, которые необходимы для обновления измененных файлов и получения актуальной версии программы.

Зачем нужна make?

1)Ускорение разработки: За счет инкрементальной сборки, когда перекомпилируются только измененные файлы, значительно сокращается время сборки проекта.

2)Уменьшение ошибок: Автоматизация процесса сборки снижает вероятность возникновения ошибок, связанных с ручным выполнением команд.

3)Улучшение воспроизводимости: Благодаря Makefile процесс сборки становится более предсказуемым и воспроизводимым на разных машинах.

4)Управление сложными проектами: Make позволяет эффективно управлять проектами с большим количеством файлов и сложной структурой зависимостей.

***2. Что такое автосборка? Какие приемущества она дает?***

**Автосборка (или автоматизированная сборка)** — это процесс, при котором создание программного продукта из исходного кода выполняется автоматически с помощью специальных инструментов. Вместо того чтобы вручную компилировать каждый файл, запускать линковщик и выполнять другие рутинные операции, разработчик описывает процесс сборки в специальном конфигурационном файле, а затем запускает автоматизированный инструмент, который выполняет все необходимые действия.

**Преимущества автосборки:**

1. Повышение производительности: Автоматизация рутинных задач позволяет разработчикам сосредоточиться на написании кода, а не на выполнении повторяющихся действий.
2. Уменьшение ошибок: Автоматизация минимизирует риск возникновения человеческих ошибок при выполнении ручных операций.
3. Стандартизация процесса сборки: Все члены команды используют единый подход к сборке проекта, что повышает согласованность и упрощает совместную работу.
4. Ускорение разработки: Автоматическая сборка позволяет быстрее получать готовый продукт, что особенно важно при частых изменениях кода.
5. Улучшение качества кода: Некоторые инструменты автосборки позволяют проводить автоматические проверки кода на наличие ошибок, что повышает его качество.
6. Упрощение развертывания: Автоматизированная сборка может быть интегрирована с системами непрерывной интеграции и развертывания, что позволяет автоматизировать процесс доставки приложения в производственную среду.

**Основные инструменты для автосборки**

1. Make: Один из самых старых и широко используемых инструментов. Он позволяет описать процесс сборки в виде набора правил.
2. Ant: Является более гибким и расширяемым, чем Make. Он использует XML для описания процесса сборки.
3. Maven: Специализируется на сборке проектов на языке Java. Он использует понятие артефактов и репозиториев для управления зависимостями.
4. Gradle: Более современный инструмент, который сочетает в себе гибкость Ant и мощь Maven. Он использует Groovy для описания процесса сборки.
5. npm (Node.js): Используется для управления зависимостями и сборки проектов на JavaScript.
6. Другие инструменты: Bazel, Buck, Pants и многие другие.

***3. В чем разница автосборки скриптами bash и утилитой make?***

**Bash и Make** – это два мощных инструмента, которые используются для автоматизации процесса сборки программного обеспечения. Однако у них есть свои особенности и области применения.

**Bash скрипты**

1. Гибкость: Bash скрипты предоставляют огромную гибкость в управлении процессами сборки. Вы можете выполнять практически любые команды операционной системы, создавать сложные логические конструкции и управлять потоками выполнения.
2. Простота: Для тех, кто знаком с Bash, написание скриптов для сборки может быть довольно интуитивным.
3. Универсальность: Bash скрипты могут использоваться для автоматизации самых разных задач, не только сборки.

**Недостаток:** Отсутствие встроенной системы для отслеживания зависимостей и инкрементальной сборки. Это означает, что при каждом изменении в коде придется пересобирать весь проект, даже если изменилась только небольшая часть.

**Утилита Make**

1. Специализация: Make специально разработан для управления процессом сборки. Он обладает встроенными механизмами для отслеживания зависимостей между файлами и инкрементальной сборки.
2. Эффективность: Благодаря своей специализации, Make обычно быстрее и эффективнее, чем Bash скрипты, особенно для крупных проектов.
3. Язык описания: Make использует свой собственный язык описания, который может быть менее интуитивным для новичков, чем Bash.
4. Сильная сторона: Отслеживание зависимостей. Make автоматически определяет, какие файлы необходимо перекомпилировать после внесения изменений, что значительно ускоряет процесс сборки.