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 № 24**

по дисциплине:” Системное программирование”

на тему:***” Расширенная многозадачность потоков, архитектурный шаблон MVC. Рекурсивная сборка ”***

Вариант 8

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

Приняла**:** пр. Давыденко Н.В.

Минск 2024

# Лабораторная работа № #24. Расширенная многозадачность. Архитектурный шаблон MVC. Рекурсивная сборка.

Цель работы: Изучить дополнительные вопросы, связанные с расширением возможностей многозадачности в Linux. Научиться менять приоритет выполнения процесса, узнать возможности системных вызовов семейства wait(), а также разобраться в том, насколько "страшны" процессы-зомби. 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 грамотно проектировать и разрабатывать многофайловые проекты на языке программирования С/C++ с использованием архитектурного шаблона проектирования и разработки гибких и масштабируемых приложений Model-View-Controller (MVC). Научиться эффективно использовать специальные средства для автоматизации процесса компиляции, сборки и запуска многофайловых проектов.

## Задание 1.

Создайте многофайловый проект, руководствуясь принципом Single Responsibility Principle.Используйте архитектурный шаблон проектирования MVC. Здесь под шаблоном MVC понимается файлы или группа файлов в трех отдельных директориях (папках), а не просто три отдельных файла.Для связывания объектов (файлов и функций) проекта обязательно используйте заголовочные файлы. В заголовочных файлах должны быть комментарии того, что делают функции, которые объявлены в заголовочных файлах. В приложении может быть консольное меню для вызова функций. ВЫХОД ИЗ ПРИЛОЖЕНИЯ ДОЛЖЕН БЫТЬ КОРРЕКТНЫМ. Для этого предусмотрите отдельный параметр в меню.

Разработать многозадачное приложение на основе процессов. В приложении должны быть учтены ранее изученные возможности процессов (получение данных о процессе, ожидание процесса, уступчивость процесса, процессы-зомби).Проведите рекурсивную автосборку проект с прохождением всех стадий компиляции с использованием утилиты make.

**Код Программы:**

**Include**

*Model.h*

#ifndef MODEL\_H

#define MODEL\_H

/\*\*

\* @class Model

\* @brief Класс, представляющий данные приложения.

\*/

class Model {

public:

int getData(); // Получает данные

};

#endif // MODEL\_H

*View.h*

#ifndef VIEW\_H

#define VIEW\_H

#include <iostream>

/\*\*

\* @class View

\* @brief Класс, отвечающий за вывод данных пользователю.

\*/

class View {

public:

void display(int data); // Отображает данные

void showMenu(); // Показывает меню

};

#endif // VIEW\_H

*Controller.h*

#ifndef CONTROLLER\_H

#define CONTROLLER\_H

#include "model.h"

#include "view.h"

/\*\*

\* @class Controller

\* @brief Класс, отвечающий за взаимодействие между моделью и представлением.

\*/

class Controller {

public:

Controller();

void run(); // Запускает приложение

private:

Model model; // Экземпляр модели

View view; // Экземпляр представления

void showMenu(); // Показывает меню

void handleInput(int choice); // Обрабатывает ввод пользователя

};

#endif // CONTROLLER\_H

**Src**

*Model.cpp*

#include "model.h"

int Model::getData() {

return 42; // Пример данных

}

*View.cpp*

#include "view.h"

void View::display(int data) {

std::cout << "Data: " << data << std::endl;

}

void View::showMenu() {

std::cout << "1. Показать данные" << std::endl;

std::cout << "2. Выход" << std::endl;

}

*Controller.cpp*

#include "controller.h"

#include <iostream>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

Controller::Controller() {}

void Controller::showMenu() {

view.showMenu();

}

void Controller::handleInput(int choice) {

switch (choice) {

case 1: {

int data = model.getData();

view.display(data);

break;

}

case 2:

std::cout << "Выход из приложения." << std::endl;

exit(0); // Завершение приложения

default:

std::cout << "Некорректный ввод." << std::endl;

}

}

void Controller::run() {

while (true) {

showMenu();

int choice;

std::cout << "Введите ваш выбор: ";

std::cin >> choice;

handleInput(choice); // Обработка ввода

}

}

*Main.cpp*

#include "controller.h"

int main() {

Controller controller;

controller.run();

return 0;

}

**Makefile**

CC = g++

CFLAGS = -Wall -Wextra -std=c++11

# Директории

SRC\_DIR = src

INCLUDE\_DIR = include

BUILD\_DIR = build

# Исходные файлы и объектные файлы

SRC\_FILES = $(wildcard $(SRC\_DIR)/\*.cpp)

OBJ\_FILES = $(patsubst $(SRC\_DIR)/%.cpp,$(BUILD\_DIR)/%.o,$(SRC\_FILES))

# Цели

all: $(BUILD\_DIR)/my\_project

$(BUILD\_DIR)/my\_project: $(OBJ\_FILES)

mkdir -p $(BUILD\_DIR)

$(CC) $(CFLAGS) -o $@ $^

$(BUILD\_DIR)/%.o: $(SRC\_DIR)/%.cpp

mkdir -p $(BUILD\_DIR)

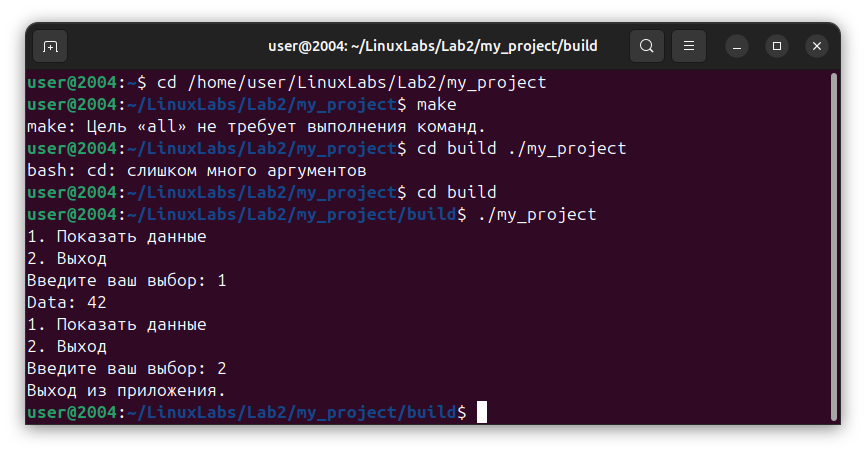
$(CC) $(CFLAGS) -c $< -o $@ -I$(INCLUDE\_DIR)

clean:

rm -rf $(BUILD\_DIR)

.PHONY: all clean

**Результаты\_работы:**



## Задание 2.

Разработайте многопоточное приложение. В приложении должны быть реали-

зованы асинхронность и синхронизация потоков, уступчивость потоков.  
  
2. Сортировка массива слиянием:

• Разделите массив на равные части и создайте поток для каждой части.

• Каждый поток должен отсортировать свою часть массива используя алгоритм сор-

тировки по выбору или вставке.

• Главный поток должен слить отсортированные части в один отсортированный массив.

• Используйте семафоры для синхронизации потоков и управления доступом к данным.

• Можете использовать рекурсию для разделения массива и потоки для сортировки.  
  
Дополнительные советы:

• Используйте системные вызовы pthread\_create(), pthread\_join(),

pthread\_mutex\_init(), pthread\_cond\_init() и другие для работы с потоками.

• Используйте библиотеку stdio.h для вывода информации о работе потоков.

• Не забывайте об обработке ошибок и защите от гонок данных.

• Используйте инструменты отладки (gdb, valgrind) для поиска ошибок в многопоточном коде.

Важные замечания:

• При написании многопоточного кода важно учитывать возможность гонок данных,

т.е. ситуации, когда несколько потоков пытаются одновременно получить доступ к

одному и тому же ресурсу.

• Для предотвращения гонок данных нужно использовать механизмы синхрониза-

ции, такие как mutex, семафоры, условия ожидания.

• Также важно правильно освобождать ресурсы, которые были выделены потокам,

чтобы избежать утечек памяти.

• Не стоит создавать слишком много потоков, так как это может привести к излишней

нагрузке на процессор и снижению производительности.

• Начните с простых задач и постепенно переходите к более сложным, чтобы лучше понять принципы многопоточного программирования.

### Скриншоты результатов

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define ARRAY\_SIZE 16 // Размер массива

#define THREAD\_COUNT 4 // Количество потоков

// Структура для передачи данных в потоки

typedef struct {

int \*array;

int left;

int right;

} ThreadData;

// Семафор для синхронизации слияния

sem\_t merge\_sem;

// Мьютекс для синхронизации вывода

pthread\_mutex\_t print\_mutex;

// Функция сортировки вставками

void insertion\_sort(int \*array, int left, int right) {

for (int i = left + 1; i <= right; i++) {

int key = array[i];

int j = i - 1;

while (j >= left && array[j] > key) {

array[j + 1] = array[j];

j--;

}

array[j + 1] = key;

}

}

// Функция слияния двух отсортированных подмассивов

void merge(int \*array, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int \*left\_arr = malloc(n1 \* sizeof(int));

int \*right\_arr = malloc(n2 \* sizeof(int));

for (int i = 0; i < n1; i++) left\_arr[i] = array[left + i];

for (int i = 0; i < n2; i++) right\_arr[i] = array[mid + 1 + i];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (left\_arr[i] <= right\_arr[j]) array[k++] = left\_arr[i++];

else array[k++] = right\_arr[j++];

}

while (i < n1) array[k++] = left\_arr[i++];

while (j < n2) array[k++] = right\_arr[j++];

free(left\_arr);

free(right\_arr);

}

// Функция сортировки, выполняемая потоком

void\* thread\_sort(void \*arg) {

ThreadData \*data = (ThreadData \*)arg;

int left = data->left;

int right = data->right;

pthread\_mutex\_lock(&print\_mutex);

printf("Thread sorting from %d to %d\n", left, right);

pthread\_mutex\_unlock(&print\_mutex);

insertion\_sort(data->array, left, right);

sem\_post(&merge\_sem);

return NULL;

}

// Функция многопоточной сортировки слиянием

void merge\_sort\_multithreaded(int \*array, int size) {

pthread\_t threads[THREAD\_COUNT];

ThreadData thread\_data[THREAD\_COUNT];

int part\_size = size / THREAD\_COUNT;

// Инициализация семафора и мьютекса

sem\_init(&merge\_sem, 0, 0);

pthread\_mutex\_init(&print\_mutex, NULL);

// Создание потоков для сортировки частей массива

for (int i = 0; i < THREAD\_COUNT; i++) {

thread\_data[i].array = array;

thread\_data[i].left = i \* part\_size;

thread\_data[i].right = (i == THREAD\_COUNT - 1) ? size - 1 : (i + 1) \* part\_size - 1;

pthread\_create(&threads[i], NULL, thread\_sort, &thread\_data[i]);

}

// Ожидание завершения всех потоков

for (int i = 0; i < THREAD\_COUNT; i++) {

sem\_wait(&merge\_sem);

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

}

// Слияние отсортированных частей

for (int i = 0; i < THREAD\_COUNT - 1; i++) {

int mid = (thread\_data[i].right);

int next\_right = thread\_data[i + 1].right;

merge(array, 0, mid, next\_right);

}

// Уничтожение семафора и мьютекса

sem\_destroy(&merge\_sem);

pthread\_mutex\_destroy(&print\_mutex);

}

// Функция для вывода массива

void print\_array(int \*array, int size) {

for (int i = 0; i < size; i++) printf("%d ", array[i]);

printf("\n");

}

int main() {

int array[ARRAY\_SIZE] = {43, 21, 34, 11, 13, 19, 22, 9, 45, 18, 4, 8, 29, 31, 5, 10};

printf("Original array:\n");

print\_array(array, ARRAY\_SIZE);

merge\_sort\_multithreaded(array, ARRAY\_SIZE);

printf("Sorted array:\n");

print\_array(array, ARRAY\_SIZE);

return 0;

}

Makefile

# Компилятор

CC = gcc

# Флаги компиляции

CFLAGS = -Wall -pthread -g

# Имя целевого исполняемого файла

TARGET = merge\_sort\_multithreaded

# Целевая цель по умолчанию

all: $(TARGET)

# Правило для сборки исполняемого файла

$(TARGET): merge\_sort\_multithreaded.c

$(CC) $(CFLAGS) -o $(TARGET) merge\_sort\_multithreaded.c

# Правило для очистки

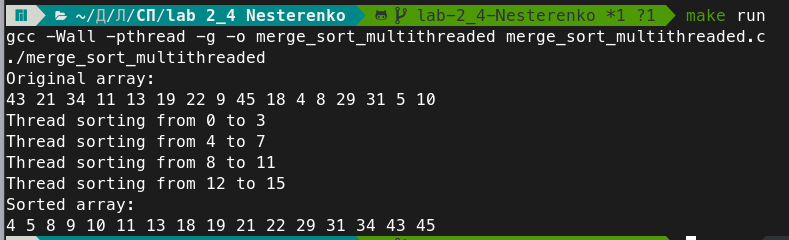
clean:

rm -f $(TARGET)

# Правило для сборки и запуска

run: all

./$(TARGET)



## Задание 3.

Используя материалы заданий No1 и No2 проведите рекурсивную автосборку

проектов с использованием утилиты Cmake.

Общий Cmake file:

cmake

Copy code

cmake\_minimum\_required(VERSION 3.10)

project(MultiProject)

# Устанавливаем общие флаги компиляции для C и C++

set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -Wall -pthread -g")

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall -Wextra -std=c++11")

# Директория для сборки

set(BUILD\_DIR ${CMAKE\_BINARY\_DIR}/build)

file(MAKE\_DIRECTORY ${BUILD\_DIR})

add\_subdirectory("lab 2\_4")

add\_subdirectory("my\_project")

Cmake file1:

В lab 2\_4/CMakeLists.txt:

cmake

Copy code

set(MERGE\_SORT\_TARGET merge\_sort\_multithreaded)

add\_executable(${MERGE\_SORT\_TARGET} merge\_sort\_multithreaded.c)

Cmake file2:

make

Copy code

set(MY\_PROJECT\_TARGET my\_project)

set(SRC\_DIR src)

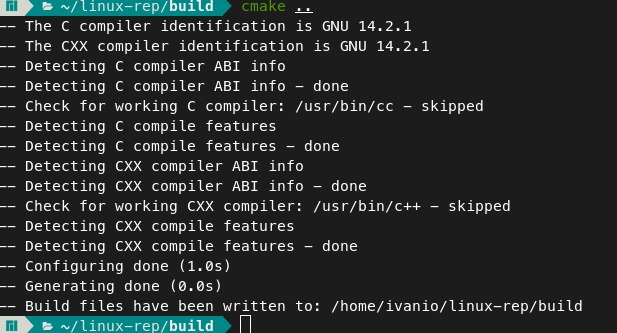
set(INCLUDE\_DIR include)

file(GLOB SRC\_FILES "${SRC\_DIR}/\*.cpp")

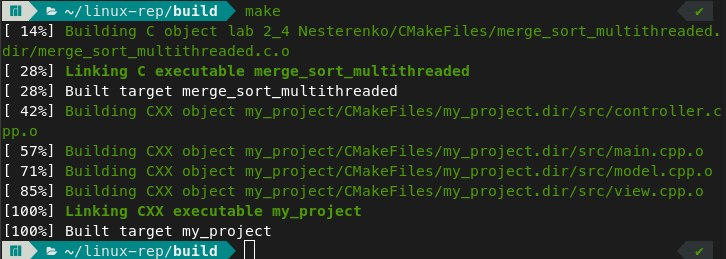
add\_executable(${MY\_PROJECT\_TARGET} ${SRC\_FILES})

include\_directories(${INCLUDE\_DIR})

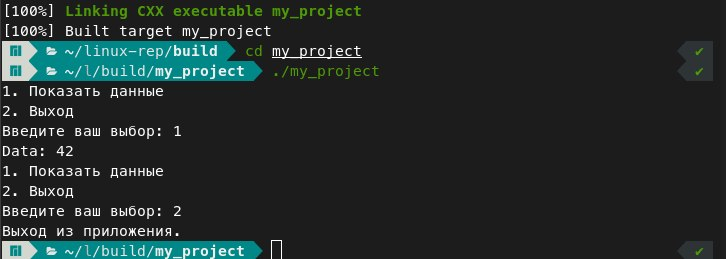
Запускаем Cmake главный и получаем:

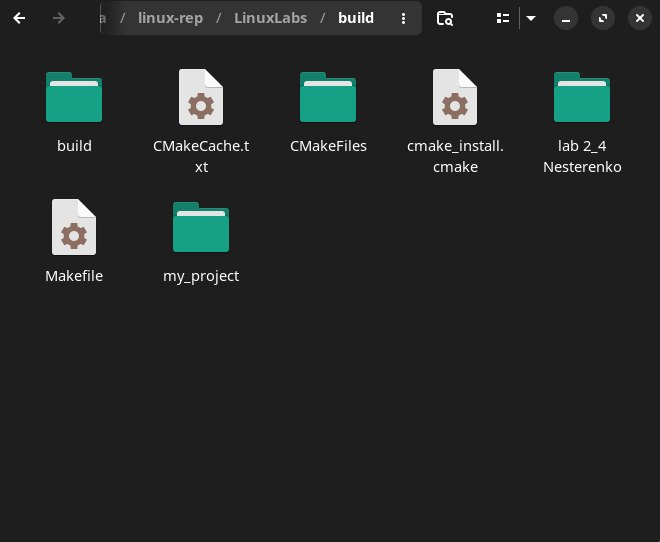


make file создан автоматически, вызываем его:



Проверим работу одного из проектов:





Вывод: Были изучены многозадачность потоков, рекурсивная сборка и шаблон MVC в Linux.

## Контрольные вопросы

В языке C/C++ существуют две формы директивы #include, которые служат для включения заголовочных файлов в исходный код программы: #include <filename> и #include "filename". Разница между ними заключается в способе поиска заголовочных файлов.

1. #include <filename>:
   * Эта форма используется для включения стандартных заголовочных файлов, которые находятся в стандартных библиотеках и установленных системных каталогах.
   * Препроцессор сначала ищет файл в системных каталогах, указанных компилятору по умолчанию (например, stdlib.h, stdio.h).
   * Примеры:

c

#include <stdio.h>

#include <stdlib.h>

1. #include "filename":
   * Эта форма используется для включения пользовательских заголовочных файлов, которые являются частью вашего проекта.
   * Препроцессор сначала ищет файл в текущем каталоге, где находится исходный файл, который выполняет директиву #include.
   * Если файл не найден в текущем каталоге, препроцессор продолжает искать его в стандартных системных каталогах.
   * Примеры:

c

#include "my\_header.h"

Таким образом, использование #include <filename> предназначено для стандартных библиотек, а #include "filename" — для заголовочных файлов, созданных пользователем и находящихся в рабочем каталоге или в других пользовательских каталогах. Это позволяет удобным образом разделять системные и пользовательские заголовочные файлы.

Препроцессор — это утилита, которая обрабатывает исходный код перед его компиляцией. В языках программирования C и C++ препроцессор выполняет ряд текстовых замен и включений, подготавливая код для компилятора. Работа препроцессора заключается в выполнении директив препроцессора, таких как #include, #define, #if, #endif и других.

Вот основные задачи препроцессора и зачем он нужен:

1. **Включение файлов (**#include**)**:
   * Препроцессор вставляет содержимое указанных заголовочных файлов в исходный код. Это позволяет разрабатывать код более модульно и повторно использовать функции и определения из других файлов.
   * Пример:

c

#include <stdio.h>

#include "my\_header.h"

1. **Определение макросов (**#define**)**:
   * Препроцессор позволяет задавать макросы, которые заменяются на соответствующие значения или выражения в исходном коде. Это упрощает управление константами и повторяющимися кодовыми конструкциями.
   * Пример:

c

#define PI 3.14

#define SQUARE(x) ((x) \* (x))

1. **Условная компиляция (**#if**,** #ifdef**,** #ifndef**,** #else**,** #endif**)**:
   * Препроцессор позволяет компилировать различные части кода в зависимости от условий. Это полезно для создания кода, который может быть адаптирован к различным платформам или конфигурациям.
   * Пример:

c

#ifdef DEBUG

#define DEBUG\_PRINT(x) printf("DEBUG: %s\n", x)

#else

#define DEBUG\_PRINT(x)

#endif

1. **Замена строк (**#line**)**:
   * Препроцессор может изменять нумерацию строк и файлов, что может быть полезно для вывода сообщений об ошибках в правильных местах исходного кода.
2. **Удаление комментариев**:
   * Препроцессор удаляет комментарии из исходного кода, чтобы они не мешали компиляции.

Препроцессор облегчает разработку и сопровождение кода, делая его более гибким и удобным для работы. Он выполняет начальную обработку кода, что позволяет компилятору фокусироваться непосредственно на трансляции обработанного кода в машинный код.