МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации



**ЛАБОРАТОРНАЯ РАБОТА №4**

«Многопоточность»

**по дисциплине: «*Программирование*»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АБc-322», «АВТФ»  *Михайлов Егор Сергеевич*  «21» ноября 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  *Ассистент кафедры ЗИ*  *Исаев Глеб Андреевич*  «\_\_\_» \_\_\_\_\_\_ 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2024

**Цели и задачи работы:** изучение принципов реализации многопоточности и практические применение возможностей языков высокого уровня при моделировании бизнес-процессов.

**Задание к работе:** Самостоятельно решить задачи в соответствии с индивидуальным вариантом.

**Ссылка на GitHub:** https://github.com/Discorre/fourLaba

Задание 1 (С++):

#include <iostream>

#include <thread>

#include <mutex>

#include <semaphore>

#include <barrier>

#include <atomic>

#include <condition\_variable>

#include <chrono>

#include <vector>

#include <random>

using namespace std;

// Примитивы синхронизации

mutex mtx;

counting\_semaphore<3> semaphore(3); // Максимум 3 потока могут работать одновременно

atomic\_flag spinLock = ATOMIC\_FLAG\_INIT; // блокировка для Спинлока (изначально False- доступен)

// Функция для генерации случайного символа из ASCII

char generate\_random\_char() {

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<> dis(32, 126); // символы с кодами(которые доступны на клавиатуре)

return static\_cast<char>(dis(gen));

}

// Monitor

class Monitor {

public:

Monitor() : flag(false) {} // конструктор,который создает доступный монитор

// Метод для захвата монитора

void enter() {

unique\_lock<mutex> lock(mtx); //пытаемся захватить монитор

while (flag) { // проверка доступен ли монитор

cond.wait(lock); // если монитор занят,то нахоидся в сотоянии ожидания

}

flag = true; // захватываем монитор

}

// Метод для освобождения монитора

void exit() {

lock\_guard<mutex> lock(mtx); // захватываем мьютексом чтобы поменять флаг

flag = false; // меняем флаг ( монитор свободен)

cond.notify\_one(); // уведомляем другой поток,что монитор свободен

}

private:

bool flag; // Флаг для контроля доступа

mutex mtx;

condition\_variable cond; //переменная для уведомления потоков

};

void test\_monitor(int num\_threads) {

cout << "Тестирование Monitor:" << endl;

Monitor monitor; // создаем монитор

vector<thread> threads; // вектор для потоков

auto start = chrono::high\_resolution\_clock::now(); // таймер начала

// Создаем потоки

for (int i = 0; i < num\_threads; ++i) {

threads.push\_back(thread([i, &monitor]() {

monitor.enter(); // захватываем монитор

{

lock\_guard<mutex> lock(mtx); // защищаем вывод

char c = generate\_random\_char(); // генерируем символ

cout << "Поток " << i << ": " << c << " (Monitor)" << endl;

}

monitor.exit(); // освобождаем монитор

}));

}

for (auto& t : threads) {

t.join();

}

auto end = chrono::high\_resolution\_clock::now(); // таймер конца

chrono::duration<double> duration = end - start;

cout << "Время работы Monitor: " << duration.count() << " секунд." << endl;

cout << "---------------------------------" << endl;

}

// SemaphoreSlim

class SemaphoreSlim {

public:

SemaphoreSlim(int count) : count(count) {}

// Метод для захвата семафора

void wait() {

unique\_lock<mutex> lock(mtx); //пытаемся захватить ресурс

while (count == 0) { // проверка на свободные ресурсы

cond.wait(lock); // Ожидаем освобождения ресурса

}

--count; // Уменьшаем счетчик(захватываем ресурс)

}

// Метод для освобождения семафора

void release() {

lock\_guard<mutex> lock(mtx); // захыватыаем поток,Чтобы изменить счетчик

++count; // Увеличиваем счетчик(освобождая ресурс)

cond.notify\_one(); // Уведомляем один поток,что ресурс свободен

}

private:

int count;// сколько потоков могут использовать ресурс

mutex mtx;

condition\_variable cond;

};

// Тестирование SemaphoreSlim

void test\_semaphore\_slim(int num\_threads) {

cout << "Тестирование SemaphoreSlim:" << endl;

SemaphoreSlim sem(1);

vector<thread> threads; //вектор для потоков

auto start = chrono::high\_resolution\_clock::now(); // таймер начала

// Создаем потоки, которые будут работать с SemaphoreSlim

for (int i = 0; i < num\_threads; ++i) {

threads.push\_back(thread([i, &sem]() {

sem.wait(); // Поток ожидает, пока не освободится ресурс

{

lock\_guard<mutex> lock(mtx); // мьютекс для вывода

char c = generate\_random\_char();

cout << "Поток " << i << ": " << c << " (SemaphoreSlim)" << endl;

}

sem.release(); // Освобождаем ресурс для других потоков

}));

}

// Ожидаем завершения всех потоков

for (auto& t : threads) {

t.join();

}

auto end = chrono::high\_resolution\_clock::now(); // таймер конца

chrono::duration<double> duration = end - start; // длительность

cout << "Время работы SemaphoreSlim: " << duration.count() << " секунд." << endl;

cout << "---------------------------------" << endl;

}

void test\_mutex(int num\_threads) {

cout << "Тестирование мьютекса:" << endl;

vector<thread> threads;

auto start = chrono::high\_resolution\_clock::now();

for (int i = 0; i < num\_threads; ++i) {

threads.push\_back(thread([i]() {

lock\_guard<mutex> lock(mtx);

char c = generate\_random\_char();

cout << "Поток " << i << ": " << c << " (Мьютекс)" << endl;

}));

}

for (auto& t : threads) {

t.join();

}

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

cout << "Время работы мьютекса: " << duration.count() << " секунд." << endl;

cout << "---------------------------------" << endl;

}

void test\_semaphore(int num\_threads) {

cout << "Тестирование Semaphore:" << endl;

vector<thread> threads;

counting\_semaphore<3> sem(3);

auto start = chrono::high\_resolution\_clock::now();

for (int i = 0; i < num\_threads; ++i) {

threads.push\_back(thread([i, &sem]() {

sem.acquire();

{

lock\_guard<mutex> lock(mtx);

char c = generate\_random\_char();

cout << "Поток " << i << ": " << c << " (Semaphore)" << endl;

}

sem.release();

}));

}

for (auto& t : threads) {

t.join();

}

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

cout << "Время работы Semaphore: " << duration.count() << " секунд." << endl;

cout << "---------------------------------" << endl;

}

void test\_spinlock(int num\_threads) {

cout << "Тестирование SpinLock:" << endl;

vector<thread> threads;

auto start = chrono::high\_resolution\_clock::now();

for (int i = 0; i < num\_threads; ++i) {

threads.push\_back(thread([i]() {

while (spinLock.test\_and\_set()) { /\* активно ждём \*/ }

{

lock\_guard<mutex> lock(mtx);

char c = generate\_random\_char();

cout << "Поток " << i << ": " << c << " (SpinLock)" << endl;

}

spinLock.clear();

}));

}

for (auto& t : threads) {

t.join();

}

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

cout << "Время работы SpinLock: " << duration.count() << " секунд." << endl;

cout << "---------------------------------" << endl;

}

void test\_barrier(int num\_threads) {

cout << "Тестирование Barrier:" << endl;

vector<thread> threads;

barrier bar(num\_threads);

auto start = chrono::high\_resolution\_clock::now();

for (int i = 0; i < num\_threads; ++i) {

threads.push\_back(thread([i, &bar]() {

{

lock\_guard<mutex> lock(mtx);

char c = generate\_random\_char();

cout << "Поток " << i << ": " << c << " (до Barrier)" << endl;

}

bar.arrive\_and\_wait();

{

lock\_guard<mutex> lock(mtx);

char c = generate\_random\_char();

cout << "Поток " << i << ": " << c << " (после Barrier)" << endl;

}

}));

}

for (auto& t : threads) {

t.join();

}

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

cout << "Время работы Barrier: " << duration.count() << " секунд." << endl;

cout << "---------------------------------" << endl;

}

void test\_spinwait(int num\_threads) {

cout << "Тестирование SpinWait:" << endl;

vector<thread> threads;

auto start = chrono::high\_resolution\_clock::now();

for (int i = 0; i < num\_threads; ++i) {

threads.push\_back(thread([i]() {

while (spinLock.test\_and\_set()) {

this\_thread::yield(); // Уступаем другим потокам

}

{

lock\_guard<mutex> lock(mtx);

char c = generate\_random\_char();

cout << "Поток " << i << ": " << c << " (SpinWait)" << endl;

}

spinLock.clear();

}));

}

for (auto& t : threads) {

t.join();

}

auto end = chrono::high\_resolution\_clock::now();

chrono::duration<double> duration = end - start;

cout << "Время работы SpinWait: " << duration.count() << " секунд." << endl;

cout << "---------------------------------" << endl;

}

int main() {

//cout << "Запуск потоков, генерирующих случайные символы:" << endl;

const int num\_threads = 8; // количество потоков

vector<thread> threads; // вектор для потоков

//cout << "Завершение работы всех потоков." << endl;

//cout << "---------------------------------" << endl;

// Тестируем примитивы

test\_mutex(num\_threads);

test\_semaphore(num\_threads);

test\_spinlock(num\_threads);

test\_barrier(num\_threads);

test\_spinwait(num\_threads);

test\_monitor(num\_threads);

test\_semaphore\_slim(num\_threads);

return 0;

}

Генерация чеков для Задания 2:

import random

# Данные о товарах

products = [

("Яблоко", 1.0), ("Банан", 0.5), ("Апельсин", 1.5), ("Киви", 2.0),

("Виноград", 3.0), ("Персик", 1.2), ("Лимон", 1.2), ("Груша", 1.1),

("Арбуз", 5.0), ("Дыня", 4.5), ("Слива", 2.0), ("Гранат", 3.0),

("Малина", 3.5), ("Черника", 4.0), ("Клубника", 2.5), ("Грейпфрут", 1.8),

("Манго", 2.8), ("Фейхоа", 3.2), ("Персик", 1.6), ("Апельсин", 2.1)

]

print("Начинается генерация чеков...")

print("Чеки могут загружаться очень долго...")

# Функция для генерации случайных товаров для чека

def generate\_receipt(receipt\_id):

num\_items = random.randint(1, 5) # Чек будет содержать от 1 до 5 товаров

receipt\_data = []

for \_ in range(num\_items):

product, price = random.choice(products) # Выбираем случайный товар

quantity = random.randint(1, 10) # Случайное количество товара

receipt\_data.append((product, price, quantity))

return receipt\_id, receipt\_data

# Генерация 10 чеков

with open('receiptsUltraMini.txt', 'w', encoding='utf-8') as file:

for receipt\_id in range(1, 11):

receipt\_id, items = generate\_receipt(receipt\_id)

receipt\_line = f"{receipt\_id} "

# Формируем строку для одного чека

receipt\_line += ", ".join([f"{product} {price} {quantity}" for product, price, quantity in items])

file.write(receipt\_line + "\n")

# Генерация 100 чеков

with open('receiptsMini.txt', 'w', encoding='utf-8') as file:

for receipt\_id in range(1, 101):

receipt\_id, items = generate\_receipt(receipt\_id)

receipt\_line = f"{receipt\_id} "

# Формируем строку для одного чека

receipt\_line += ", ".join([f"{product} {price} {quantity}" for product, price, quantity in items])

file.write(receipt\_line + "\n")

# Генерация 1000000 чеков

with open('receiptsMacro.txt', 'w', encoding='utf-8') as file:

for receipt\_id in range(1, 1000001):

receipt\_id, items = generate\_receipt(receipt\_id)

receipt\_line = f"{receipt\_id} "

# Формируем строку для одного чека

receipt\_line += ", ".join([f"{product} {price} {quantity}" for product, price, quantity in items])

file.write(receipt\_line + "\n")

# Генерация 5000000 чеков

with open('receiptsUltraMacro.txt', 'w', encoding='utf-8') as file:

for receipt\_id in range(1, 5\_000\_001):

receipt\_id, items = generate\_receipt(receipt\_id)

receipt\_line = f"{receipt\_id} "

# Формируем строку для одного чека

receipt\_line += ", ".join([f"{product} {price} {quantity}" for product, price, quantity in items])

file.write(receipt\_line + "\n")

print("Файлы с чеками успешно созданы!")

Задание 2 (вариант 15 С++):

#include <iostream>

#include <vector>

#include <thread>

#include <chrono>

#include <unordered\_map>

#include <mutex>

#include <fstream>

#include <sstream>

// Структура, описывающая товар

struct Product {

std::string name; // Название товара

double price; // Цена товара

int quantity; // Количество товара

};

// Структура, описывающая чек

struct Receipt {

int id; // Уникальный идентификатор чека

std::vector<Product> products; // Список товаров в чеке

};

// Класс для обработки данных о покупках

class SalesProcessor {

private:

std::vector<Receipt> receipts; // Список всех чеков

std::unordered\_map<std::string, int> totalProductQuantity; // Суммарное количество проданных товаров по названию

std::unordered\_map<std::string, std::vector<std::pair<int, double>>> productReceipts; // Карта товаров и чеков, в которых они присутствуют

std::mutex mtx; // Мьютекс для синхронизации доступа к общим данным в многопоточном режиме

public:

// Конструктор, инициализирующий объект списком чеков

SalesProcessor(const std::vector<Receipt>& recs) : receipts(recs) {}

// Однопоточная обработка данных

void processSingleThread() {

for (const auto& receipt : receipts) {

processReceipt(receipt); // Обрабатываем каждый чек по порядку

}

std::cout << std::endl;

printResults(); // Выводим результаты

}

// Многопоточная обработка данных

void processMultiThread(int numThreads) {

std::vector<std::thread> threads; // Вектор потоков

int chunkSize = receipts.size() / numThreads; // Размер блока чеков, обрабатываемого каждым потоком

// Локальные контейнеры для хранения данных каждого потока

std::vector<std::unordered\_map<std::string, int>> localProductQuantities(numThreads);

std::vector<std::unordered\_map<std::string, std::vector<std::pair<int, double>>>> localProductReceipts(numThreads);

// Создаем потоки

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

int start = i \* chunkSize; // Начальный индекс блока

int end = (i == numThreads - 1) ? receipts.size() : start + chunkSize; // Конечный индекс блока

threads.emplace\_back([this, start, end, &localProductQuantities, &localProductReceipts, i]() {

// Обработка блоков чеков каждым потоком

for (int j = start; j < end; ++j) {

processReceipt(receipts[j], localProductQuantities[i], localProductReceipts[i]);

}

});

}

// Ожидание завершения всех потоков

for (auto& t : threads) {

t.join();

}

// Слияние локальных данных в общий результат

for (const auto& localQuantity : localProductQuantities) {

for (const auto& [product, quantity] : localQuantity) {

totalProductQuantity[product] += quantity; // Суммируем количество проданных товаров

}

}

for (const auto& localReceipts : localProductReceipts) {

for (const auto& [product, receipts] : localReceipts) {

productReceipts[product].insert(productReceipts[product].end(), receipts.begin(), receipts.end()); // Объединяем данные о чеках

}

}

std::cout << std::endl;

printResults();

}

// Функция для обработки одного чека (однопоточная версия)

void processReceipt(const Receipt& receipt) {

for (const auto& product : receipt.products) {

std::lock\_guard<std::mutex> lock(mtx); // Блокируем доступ для синхронизации

totalProductQuantity[product.name] += product.quantity; // Увеличиваем общее количество проданных товаров

productReceipts[product.name].emplace\_back(receipt.id, product.price); // Добавляем информацию о чеке

}

}

// Перегруженная функция для обработки одного чека (многопоточная версия)

void processReceipt(const Receipt& receipt, std::unordered\_map<std::string, int>& localQuantity, std::unordered\_map<std::string, std::vector<std::pair<int, double>>>& localReceipts) {

for (const auto& product : receipt.products) {

localQuantity[product.name] += product.quantity; // Увеличиваем количество проданных товаров в локальном контейнере

localReceipts[product.name].emplace\_back(receipt.id, product.price); // Добавляем информацию о чеке в локальный контейнер

}

}

// Вывод результатов обработки на экран

void printResults() const {

for (const auto& [product, quantity] : totalProductQuantity) {

std::cout << "Товар: " << product << ", Общее количество продано: " << quantity << "\n";

std::cout << "Чеки, содержащие " << product << ":\n";

for (const auto& [receiptId, price] : productReceipts.at(product)) {

std::cout << " - Номер чека: " << receiptId << ", Цена: " << price << "\n";

}

}

}

};

// Функция для загрузки данных о чеках из файла

std::vector<Receipt> loadReceiptsFromFile(const std::string& filename) {

std::ifstream file(filename); // Открываем файл

std::string line;

std::unordered\_map<int, Receipt> receiptsMap; // Используем временную карту для объединения данных по идентификаторам чеков

while (std::getline(file, line)) {

std::istringstream iss(line);

int id;

std::string name;

double price;

int quantity;

// Читаем строку с данными

if (!(iss >> id >> name >> price >> quantity)) {

std::cerr << "Ошибка при чтении строки: " << line << "\n"; // Если ошибка, выводим сообщение и пропускаем строку

continue;

}

// Добавляем данные в соответствующий чек

receiptsMap[id].id = id;

receiptsMap[id].products.push\_back({name, price, quantity});

}

// Переносим данные из карты в вектор

std::vector<Receipt> receipts;

for (auto& [\_, receipt] : receiptsMap) {

receipts.push\_back(std::move(receipt));

}

return receipts;

}

int main() {

// Загружаем данные о чеках из файла

std::vector<Receipt> receipts = loadReceiptsFromFile("receiptsUltraMini.txt");

SalesProcessor processor(receipts);

// Измерение времени для однопоточной обработки

auto start = std::chrono::high\_resolution\_clock::now();

processor.processSingleThread();

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> singleThreadDuration = end - start;

// Вывод времени однопоточной обработки

std::cout << "Время однопоточной обработки: " << singleThreadDuration.count() << " секунд\n";

// Обнуление результатов для многопоточной обработки

SalesProcessor multiThreadProcessor(receipts);

// Измерение времени для многопоточной обработки

int numThreads = 4; // Количество потоков

start = std::chrono::high\_resolution\_clock::now();

multiThreadProcessor.processMultiThread(numThreads);

end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> multiThreadDuration = end - start;

// Вывод времени многопоточной обработки

std::cout << "Время многопоточной обработки: " << multiThreadDuration.count() << " секунд\n";

return 0;

}

Задание 3 (вариант 2):  
#include <iostream>

#include <thread>

#include <mutex>

#include <chrono>

using namespace std;

class MyVector {

public: // можно использовать в любой точке кода

MyVector() : size\_(0), capacity\_(1) { // конструктор - инициализация

data\_ = new thread \* [capacity\_];

}

~MyVector() { //деструктор - освобожденине памяти

delete[] data\_;

}

void push\_back(thread\* t) { //добавление в конец массива

if (size\_ >= capacity\_) {

resize();

}

data\_[size\_++] = t;

}

thread\* operator[](size\_t index) { //доступ к элементу

return data\_[index];

}

size\_t size() const { //узнать размер массива

return size\_;

}

private: // использование только внутри класса

thread\*\* data\_; //указатель на массив указателей (потоки)

size\_t size\_; //размер

size\_t capacity\_;//емкость

void resize() { // метод увеличение емкости массива

capacity\_ \*= 2;

thread\*\* new\_data = new thread \* [capacity\_];

for (size\_t i = 0; i < size\_; ++i) {

new\_data[i] = data\_[i];

}

delete[] data\_;

data\_ = new\_data;

}

};

const int num\_philosophers = 5; // количество философов

mutex forks[num\_philosophers]; // Вилки между философами

mutex output\_mutex; // Мьютекс для синхронизации вывода

void philosopher(int id) {

while (true) {

// Философ размышляет

{

lock\_guard<mutex> lock(output\_mutex); // захватываем мьютексом вывод в консоль

cout << "Философ " << id << " размышляет.\n";

}

this\_thread::sleep\_for(chrono::milliseconds(5000));

// Взятие вилок по порядку

mutex& first\_fork = forks[min(id, (id + 1) % num\_philosophers)]; //сначала берем вилку с меньшим номером

mutex& second\_fork = forks[max(id, (id + 1) % num\_philosophers)]; // потом берем вилку с большим номером

// Захват вилок

lock(first\_fork, second\_fork); // Зафиксируем обе вилки

lock\_guard<mutex> lg1(first\_fork, adopt\_lock); // Зафиксировать первую вилку (передача через параметр adopt\_lock уже ранее захваченного мьютекса)

lock\_guard<mutex> lg2(second\_fork, adopt\_lock); // Зафиксировать вторую вилку

// Философ ест

{

lock\_guard<mutex> lock(output\_mutex); // забираем мьютексом вывод

cout << "Философ " << id << " начал есть.\n"; // Изменено сообщение

}

this\_thread::sleep\_for(chrono::milliseconds(5000));

// Философ освобождает вилки (выйдет из области видимости, и вилки будут освобождены)

{

lock\_guard<mutex> lock(output\_mutex); // забираем мьютексом вывод

cout << "Философ " << id << " освободил вилки.\n"; // Новое сообщение о освобождении вилок

}

}

}

int main() {

MyVector philosophers; // вектор с философами

for (int i = 0; i < num\_philosophers; ++i) {

philosophers.push\_back(new thread(philosopher, i)); // создаем поток(философа) и заплняяем вектор

}

for (size\_t i = 0; i < philosophers.size(); ++i) { //запускаем потоки

philosophers[i]->join();

delete philosophers[i]; // освобождаем память

}

return 0;

}