Министерство цифрового развития, связи и массовых коммуникаций

Российской Федерации Ордена Трудового Красного Знамени

федеральное государственное бюджетное образовательное

учреждение высшего образования

Московский технический университет связи и информатики

Кафедра «Системного программирования»

Лабораторная работа №3

по дисциплине

«Операционные системы»

Выполнила: студентка группы БСТ2104

Первухина А. А.

Проверила: Алексанян Диана Ашотовна

Москва 2023

**Оглавление**

[Цель работы: 3](#_Toc149174580)

[Лабораторное задание: 3](#_Toc149174581)

[Задача 1. Синхронизация потоков с помощью критических секций 4](#_Toc149174582)

[Задача 2. Синхронизация потоков с помощью мьютексов 7](#_Toc149174583)

[Задача 3. Синхронизация потоков с помощью событий 10](#_Toc149174584)

[Задача 4\*. Синхронизация потоков с помощью семафоров 12](#_Toc149174585)

[Вывод 14](#_Toc149174586)

# Цель работы:

Получение практических навыков по использованию API Linux для синхронизации потоков.

# Лабораторное задание:

Исследование на конкретном примере следующих методов синхронизации потоков:

1) критические секции

2) мьтексы

3) события

4) \* семафоры или ждущие таймеры

Задачи для демонстрации применения средств синхронизации придумайте самостоятельно

Задачи для каждого средства синхронизации должны демонстрировать понимание особенностей его использования.

При защите работы необходимо аргументированно обосновать выбор средства синхронизации для каждой из задач.

# Задача 1. Синхронизация потоков с помощью критических секций

Выберем задачу управления инвентарем на складе. У нас есть несколько потоков, представляющих продажи и поставки товаров. Задача состоит в том, чтобы эффективно управлять счетчиком товаров, чтобы избежать проблем параллельного доступа и обеспечить корректное выполнение операций.

Код программы «Критические секции»:

#include <Windows.h>

#include <iostream>

#include <vector>

#include <thread>

// Глобальные переменные

int goodsCount = 0;

int goodsBought = 0;

int goodsSupplied = 0;

CRITICAL\_SECTION goodsCountCriticalSection; // Cоздание критической секции

// Функция для изменения счетчика товаров внутри критической секции

void ChangeGoodsCount(int changeAmount, const char\* action) {

EnterCriticalSection(&goodsCountCriticalSection);

goodsCount += changeAmount;

setlocale(LC\_ALL, "rus");

if (changeAmount > 0) {

goodsSupplied += changeAmount;

}

else {

goodsBought -= changeAmount;

}

std::cout << "Количество товаров изменено. Новое количество: " << goodsCount << " (Продано: " << goodsBought << ", поставлено: " << goodsSupplied << ") Действие: " << action << std::endl;

LeaveCriticalSection(&goodsCountCriticalSection);

// Только один поток может изменять goodsCount в определенный момент времени

}

/\*

Функции, моделирующие поток продаж и поставок.

Они вызывают функцию ChangeGoodsCount для изменения счетчика товаров в критической секции.

\*/

void SalesThread() {

for (int i = 0; i < 5; ++i) {

// Моделируем продажу товаров

Sleep(100); // Имитация времени продажи

ChangeGoodsCount(-1, "Продано (-1)");

}

}

void SupplyThread() {

for (int i = 0; i < 10; ++i) {

// Моделируем поставку товаров

Sleep(200); // Имитация времени поставок

ChangeGoodsCount(2, "Поставлено (+2)");

}

}

int main() {

// Инициализация критической секции

InitializeCriticalSection(&goodsCountCriticalSection);

// Создаем потоки для продаж и поставок

std::thread salesThread(SalesThread);

std::thread supplyThread(SupplyThread);

/\*

Ожидание завершения потоков

Join — это метод синхронизации, который блокирует вызывающий поток (т. е. поток, вызывающий метод), пока не завершится поток, метод которого Join вызывается.

\*/

salesThread.join();

supplyThread.join();

// Уничтожение критической секции после завершения работы потоков.

DeleteCriticalSection(&goodsCountCriticalSection);

return 0;

}

Результат работы программы:

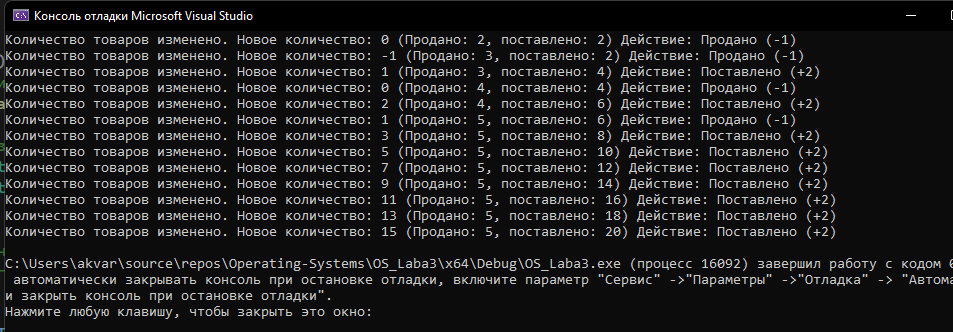


Рисунок 1 – результат работы задания 1

Мы выбрали использование критической секции в WinAPI для синхронизации потоков. Этот выбор обоснован тем, что критическая секция предоставляет простой и эффективный способ защиты критической области кода от параллельного выполнения несколькими потоками. Она автоматически обеспечивает взаимное исключение, что позволяет избежать состояний гонки и обеспечивает атомарное выполнение операций в критической секции.

Выбор критической секции также обоснован тем, что она является простой в использовании и не требует явного создания объекта мьютекса или семафора. Кроме того, она автоматически освобождает ресурсы в случае возникновения исключения в критической секции.

Таким образом, использование критической секции обеспечивает надежную синхронизацию и упрощает код, делая его более читаемым и понятным.

# Задача 2. Синхронизация потоков с помощью мьютексов

Создать многопроцессорную программу для записи данных в файл. Для обеспечения безопасности доступа к файлу и синхронизации процессов необходимо использовать мьютекс. Для данной задачи, где процессы должны записать данные в файл без дополнительной логики, мьютекс является простым и надёжным механизмом синхронизации. Мьютекс гарантирует эксклюзивный доступ, позволяя одному процессу за раз получить доступ к ресурсу (в данном случае – файлу), что исключает возможность конфликтов и пересечения при записи в файл, так как мьютекс обладает жёстким поведением блокировки – пока один процесс владеет мьютексом и выполняет запись в файл, другие процессы будут заблокированы и будут ожидать освобождения мьютекса.

Код программы «Мьютексы»:

#include <iostream>

#include <fstream>

#include <mutex>

#include <thread>

// Создаем мьютекс для синхронизации доступа к файлу

std::mutex file\_mutex;

// Функция для записи в файл с использованием мьютекса

void write\_to\_file(const std::string& filename, const std::string& data) {

std::lock\_guard<std::mutex> lock(file\_mutex);

std::cout << std::this\_thread::get\_id() << " writes to a file '" << filename << "'" << std::endl;

std::ofstream file(filename, std::ios::app); // Открываем файл для записи

if (file.is\_open()) {

file << data;

}

std::cout << std::this\_thread::get\_id() << " finished writing" << std::endl;

}

int main() {

// Создаем файл "example.txt" (перезаписываем его, если он существует)

std::string filename = "C:/Users/akvar/OneDrive/Рабочий стол/lab3OS/example.txt";

std::ofstream init\_file(filename);

init\_file.close();

// Создаем два потока, каждый из которых вызовет функцию write\_to\_file

std::thread thread1(write\_to\_file, filename, "Data from the Thread 1\nqwe\n");

std::thread thread2(write\_to\_file, filename, "Data from the Thread 2\nrty\n");

// Ждем, пока оба потока завершат свою работу

thread1.join();

thread2.join();

std::cout << "All threads have shut down" << std::endl;

return 0;

}

Результат работы программы:

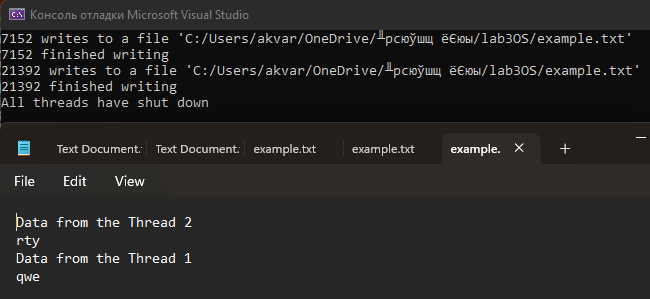


Рисунок 2 – результат работы задания 2

# Задача 3. Синхронизация потоков с помощью событий

Создать многопоточную программу для упорядоченного вывода нечётных и чётных чисел. Для синхронизации потоков и определения порядка вывода чисел использовать события.

В данной задаче нет необходимости в критических секциях или блокировании ресурсов, события просто указывают на то, какой поток, должен выполнить свою работу в данное время, позволяя легко изменять порядок выполнения потоков. События предоставляют два основных состояния – установлено и сброшено, что позволяет эффективно управлять порядком выполнения потоков.

Код программы «События»:

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

// Создаем два события

std::condition\_variable event\_odd, event\_even;

std::mutex mutex;

bool is\_odd\_turn = true;

void odd\_thread() {

for (int i = 1; i < 10; i += 2) {

std::unique\_lock<std::mutex> lock(mutex);

event\_odd.wait(lock, [] { return is\_odd\_turn; });

std::cout << "Odd flow:" << i << std::endl;

is\_odd\_turn = false;

event\_even.notify\_one();

}

}

void even\_thread() {

for (int i = 2; i <= 10; i += 2) {

std::unique\_lock<std::mutex> lock(mutex);

event\_even.wait(lock, [] { return !is\_odd\_turn; });

std::cout << "Even flow:" << i << std::endl;

is\_odd\_turn = true;

event\_odd.notify\_one();

}

}

int main() {

// Создаем потоки

std::thread t1(odd\_thread);

std::thread t2(even\_thread);

// Ждем завершения потоков

t1.join();

t2.join();

return 0;

}

Результат работы программы:

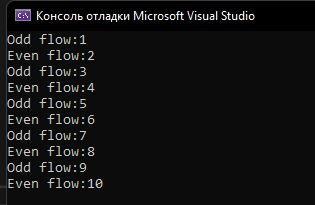


Рисунок 3 – результат работы задания 3

# Задача 4\*. Синхронизация потоков с помощью семафоров

Представим, что у нас есть некоторый ресурс, например, бассейн с ограниченным количеством мест. Каждый поток представляет посетителя, который хочет войти в бассейн. Одновременно в бассейне могут находиться не более определенного количества посетителей.

Код программы «Семафоры»:

#include <iostream>

#include <vector>

#include <Windows.h>

#include <thread>

#include <chrono>

#include <mutex>

const int MAX\_VISITORS = 3;

HANDLE poolSemaphore;

std::mutex consoleMutex;

void Visitor(int visitorID) {

{

setlocale(LC\_ALL, "rus");

std::lock\_guard<std::mutex> lock(consoleMutex);

std::cout << "Посетитель " << visitorID << " готовится плавать." << std::endl;

}

// Ожидание разрешения на посещение бассейна

WaitForSingleObject(poolSemaphore, INFINITE); // поток ожидает, пока семафор не станет доступным (счетчик семафора не равен 0),

// что указывает на наличие свободных мест в бассейне.

{

std::lock\_guard<std::mutex> lock(consoleMutex);

std::cout << "Посетитель " << visitorID << " теперь плавает в бассейне!" << std::endl;

}

// Имитация пребывания в бассейне

std::this\_thread::sleep\_for(std::chrono::seconds(2));

{

std::lock\_guard<std::mutex> lock(consoleMutex);

std::cout << "Посетитель " << visitorID << " покидает бассейн." << std::endl;

}

// Освобождение места в бассейне

ReleaseSemaphore(poolSemaphore, 1, NULL);

}

int main() {

// Создание семафора с начальным значением равным максимальному количеству посетителей

poolSemaphore = CreateSemaphore(NULL, MAX\_VISITORS, MAX\_VISITORS, NULL); // Поток освобождает место в бассейне с помощью функции ReleaseSemaphore.

std::vector<std::thread> visitors;

const int NUM\_VISITORS = 7;

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

visitors.emplace\_back(Visitor, i + 1);

}

for (auto& visitorThread : visitors) {

visitorThread.join();

}

// Закрытие дескриптора семафора

CloseHandle(poolSemaphore);

return 0;

}

Результат работы программы:

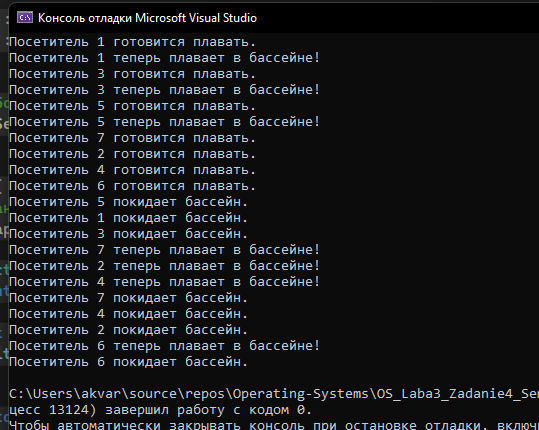


Рисунок 4 – результат работы задания 4

Семафоры прекрасно подходят для управления количеством потоков, которые могут одновременно получить доступ к ресурсу. В данной задаче ресурсом является "место в бассейне", и семафоры позволяют легко управлять этим количеством. Семафоры предоставляют удобный механизм для увеличения счетчика ресурса (в данном случае, мест в бассейне) при освобождении ресурса. Это отражает естественную логику задачи, где посетитель покидает бассейн и освобождает место для следующего.

# Вывод

В данной лабораторной работе мы рассмотрели использование методов синхронизации потоков . Выбор средства синхронизации (семафоры, критические секции, мьютексы, события) зависит от конкретных требований задачи.

Исходя из лабораторной работы, мы можем сделать следующие выводы:

1. Работа с Критическими Секциями:

* Критические секции обеспечивают мутуальное исключение для защиты общих ресурсов.
* Они удобны, но ограничены использованием только внутри одного процесса.

1. Работа с Мьютексами:

* Мьютексы также обеспечивают мутуальное исключение, но могут быть использованы между процессами.
* Они предоставляют более широкие возможности для синхронизации, но требуют явного создания и удаления.

1. Работа с Событиями:

* События позволяют потокам взаимодействовать, сообщая о событиях или ожидая их.
* Они подходят для решения задач с ожиданием определенных условий.

1. Работа с Семафорами:

* Семафоры применяются для управления доступом к ограниченным ресурсам.
* Они предоставляют гибкий механизм для ограничения количества потоков, имеющих доступ к ресурсу.

1. Выбор средства синхронизации зависит от задачи:

* Каждое средство синхронизации имеет свои преимущества и ограничения.
* Выбор средства должен зависеть от конкретных требований задачи.

1. Безопасность при работе с многозадачностью:

* Безопасность выполнения потоков важна, и ее можно обеспечить с использованием средств синхронизации и блокировок.

1. Платформозависимость:

* Некоторые средства синхронизации могут быть зависимы от платформы (например, функции из <Windows.h>).

Необходимо учитывать платформенные особенности при выборе средства.

При работе с многозадачностью важно правильно выбирать средства синхронизации в зависимости от конкретных требований задачи и платформы, а также обеспечивать безопасность выполнения потоков.

Начало формы