Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЁТ

к лабораторной работе №4

на тему

**ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ (ПОТОКОВ): ВЗАИМНОЕ ИСКЛЮЧЕНИЕ И СИНХРОНИЗАЦИЯ**

Выполнил:

студент гр.253505 Таргонский Д.А.

Проверил:

ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc179320137)

[2 Краткие теоретические сведения 4](#_Toc179320138)

[3 Инструментальная языковая среда 5](#_Toc179320139)

[4 Описание программного продукта 6](#_Toc179320140)

[4.1 Описание используемых функций 6](#_Toc179320141)

[4.2 Алгоритм работы программы 7](#_Toc179320142)

[5 Результат выполнения программы 8](#_Toc179320143)

[Заключение 9](#_Toc179320144)

[Список использованных источников 10](#_Toc179320145)

[Приложения А (обязательное) исходный код продукта 11](#_Toc179320146)

# **1 ЦЕЛЬ РАБОТЫ**

Цель данной лабораторной работы заключается в освоении и закреплении навыков работы с методами передачи данных в многопоточной среде, что является важным аспектом системного программирования. В условиях современного программирования, где многопоточность и параллельная обработка данных становятся стандартом, необходимо понимать, как организовать эффективное взаимодействие между потоками для достижения высоких показателей производительности.

В ходе выполнения лабораторной работы будут исследоваться различные подходы к синхронизации доступа к разделяемым ресурсам. Это включает в себя изучение механизмов, таких как мьютексы, семафоры и критические секции, которые позволяют избежать конфликтов при одновременном доступе потоков к общим данным. Также будет рассмотрено, как правильно организовать межпроцессное взаимодействие, что критически важно для обеспечения корректности и эффективности работы приложений.

Задача состоит в реализации программы, которая:

* Организует совместное использование разделяемой памяти для обработки данных. Это позволит потокам обмениваться информацией без необходимости использовать файловую систему, что значительно ускорит процесс.
* Обеспечивает синхронизацию доступа к разделяемым ресурсам, предотвращая коллизии. Важно реализовать механизмы, которые будут контролировать доступ к критическим секциям кода, гарантируя, что только один поток может изменять данные в определенный момент времени.
* Сравнивает эффективность различных методов доступа к данным. В рамках работы будет проведен анализ производительности как подхода, использующего разделяемую память, так и традиционного метода, основанного на взаимодействии через файлы. Это поможет понять преимущества и недостатки каждого из методов в зависимости от условий задачи.
* Измеряет время выполнения операций. Программа будет фиксировать время выполнения операций в обоих подходах, что позволит провести количественный анализ и сделать выводы о том, какой метод более эффективен в различных сценариях.

Таким образом, выполнение данной лабораторной работы не только углубит теоретические знания, но и даст практические навыки, необходимые для разработки многопоточных приложений.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Подсистема межпроцессного взаимодействия (IPC) в операционных системах отвечает за организацию связи между процессами и обмен данными между ними. Существует несколько основных методов IPC: разделяемая память, очереди сообщений, каналы и семафоры. Эти методы позволяют процессам взаимодействовать друг с другом, обеспечивая синхронизацию и согласованный доступ к общим ресурсам.

Разделяемая память позволяет нескольким процессам получить доступ к одной и той же области памяти, что обеспечивает высокую скорость обмена данными. Однако для предотвращения коллизий и обеспечения безопасности данных требуется механизм синхронизации, такой как мьютексы или семафоры.

Очереди сообщений позволяют процессам отправлять и получать сообщения, обеспечивая надежный обмен данными, даже если процессы выполняются асинхронно. Этот метод позволяет избежать проблем, связанных с прямым доступом к памяти, так как данные передаются в виде сообщений.

Каналы представляют собой специальный тип IPC, который обеспечивает потоковый обмен данными между процессами. Каналы могут быть как именованными, так и анонимными, в зависимости от необходимости в идентификации.

Основные функции для работы с IPC включают создание и управление объектами IPC, такими как CreateSemaphore(), CreateMutex(), MapViewOfFile() и SendMessage(). CreateSemaphore() создает семафор, который может использоваться для управления доступом к ресурсам. MapViewOfFile() позволяет отображать разделяемую память в адресное пространство процесса, а SendMessage() используется для отправки сообщений между процессами, обеспечивая асинхронное взаимодействие.

Эти методы и функции обеспечивают эффективное и безопасное межпроцессное взаимодействие, что является ключевым аспектом разработки многозадачных приложений.

# **3 ИНСТРУМЕНТАЛЬНАЯ ЯЗЫКОВАЯ СРЕДА**

Для разработки программы был выбран язык программирования C++. Это компилируемый, высокопроизводительный язык программирования общего назначения, поддерживающий как низкоуровневые, так и высокоуровневые парадигмы программирования. C++ используется для разработки системного программного обеспечения, приложений, драйверов устройств, а также других программ, требующих высокой эффективности и прямого доступа к системным ресурсам.

В качестве интегрированной среды разработки (IDE) был выбран Microsoft Visual Studio, что является важным решением для разработчиков. Интегрированная среда разработки — это программа, которая объединяет все необходимые инструменты для написания кода, отладки и тестирования в одном интерфейсе, что значительно упрощает процесс разработки. Microsoft Visual Studio является одной из самых популярных IDE для разработки на C++. Она предлагает широкий набор функций, включая редактор кода с подсветкой синтаксиса и автозавершением, инструменты для отладки, встроенные средства для юнит-тестирования и возможности для профилирования и анализа производительности кода.

Разработка осуществляется на Microsoft Windows 10. Использование этой операционной системы позволяет эффективно работать с API и системными вызовами Windows, что важно для разработки программ, которые будут функционировать в этой среде. Windows 10 предоставляет разработчикам мощные инструменты для управления процессами и ресурсами, что критично для создания высокоэффективных приложений.

Вся работа ведется на ноутбуке, что обеспечивает мобильность и возможность работать в разных условиях — будь то офис, дом или в пути. Современные ноутбуки могут обеспечить достаточную производительность для разработки и тестирования программ, особенно если они оснащены хорошими процессорами и достаточным объемом оперативной памяти.

Таким образом, выбор Microsoft Visual Studio в качестве IDE, использование операционной системы Windows 10 и работа на ноутбуке создают оптимальные условия для разработки программ на C++. Это позволяет эффективно использовать все доступные инструменты и ресурсы, что в свою очередь способствует созданию качественного и производительного программного продукта.

4 ОПИСАНИЕ ПРОГРАММНОГО ПРОДУКТА

Программа реализует модель «читатель-писатель», демонстрируя взаимодействие потоков с использованием механизмов синхронизации. В данной реализации используются семафоры и мьютексы для управления доступом к общим ресурсам, что позволяет избежать конфликтов при одновременном чтении и записи данных. Рассмотрим ключевые функции и алгоритм работы программы.

**4.1 Описание используемых функций**

Далее представлены функции, используемые для работы с межпроцессным взаимодействием и синхронизацией потоков:

1. CreateSemaphore(): Эта функция создает семафор, который используется для управления доступом читателей к общему ресурсу. Семафор позволяет ограничить количество одновременно работающих читателей, обеспечивая корректное взаимодействие с писателями.
2. CreateMutex(): Создает мьютекс, который используется для синхронизации доступа писателей к общему ресурсу. Мьютекс гарантирует, что в любой момент времени только один писатель может изменять данные.
3. InitializeCriticalSection(): Инициализирует критическую секцию, используемую для защиты счетчика активных читателей. Это необходимо для обеспечения корректного учета количества потоков, работающих с общими данными.
4. EnterCriticalSection(): Эта функция используется для входа в критическую секцию, обеспечивая эксклюзивный доступ к защищаемым данным. Она предотвращает одновременное изменение счетчика читателей.
5. LeaveCriticalSection(): Освобождает критическую секцию, позволяя другим потокам получить доступ к защищаемым данным. Это важный шаг для обеспечения корректности работы программы.
6. WaitForSingleObject(): Ожидает, пока мьютекс станет доступным, обеспечивая эксклюзивный доступ писателя к ресурсу. Эта функция предотвращает одновременную запись данных несколькими потоками.
7. ReleaseMutex(): Освобождает захваченный мьютекс, позволяя другим писателям получить доступ к ресурсу. Это критически важно для поддержания согласованности данных.
8. std::this\_thread::sleep\_for(): Используется для создания паузы в выполнении потока, что позволяет имитировать время, затрачиваемое на чтение или запись данных.
9. CloseHandle(): Закрывает дескрипторы объектов, таких как семафоры и мьютексы, освобождая системные ресурсы. Это необходимо для завершения работы программы и предотвращения утечек памяти.

Таким образом, в программном продукте представлены основные функции работы с механизмами синхронизации, что обеспечивает эффективное и безопасное взаимодействие потоков в модели «читатель-писатель».

# **4.2 Алгоритм работы программы**

Алгоритм работы данной программы включает несколько ключевых этапов, которые обеспечивают эффективное взаимодействие между потоками.

Программа начинается с инициализации необходимых ресурсов. На первом этапе создается семафор с помощью функции CreateSemaphore(), который контролирует доступ читателей к общему ресурсу. Затем создается мьютекс с использованием CreateMutex(), который используется для синхронизации доступа писателей.

После инициализации ресурсов программа создает потоки для читателей и писателей. Каждый читатель пытается получить доступ к разделяемым данным. Если это возможно, он увеличивает счетчик активных читателей, что позволяет ему читать данные. В противном случае, если ждет писатель, читатель временно прекращает свою работу.

Когда читатель завершает чтение, он уменьшает счетчик активных читателей. Если это последний читатель, он освобождает семафор, позволяя писателям получить доступ к ресурсу.

Писатели, в свою очередь, ожидают получения мьютекса для записи данных. Если мьютекс свободен, писатель записывает данные, увеличивая значение общей переменной sharedData, и освобождает мьютекс, позволяя другим писателям получить доступ.

Программа также собирает статистику по успешным и неудачным попыткам чтения и записи, а также измеряет время выполнения операций. После завершения работы всех потоков программа выводит результаты на экран, показывая эффективность работы системы.

Наконец, программа завершает свою работу, освобождая все ресурсы и закрывая дескрипторы объектов, что гарантирует отсутствие утечек памяти и корректное завершение выполнения.

Таким образом, программа демонстрирует эффективный механизм межпроцессного взаимодействия, обеспечивая корректную работу модели «читатель-писатель».

**5 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ**

Программа, основанная на данном коде, обеспечивает обмен данными между потоками через общую переменную. В результате работы приложения пользователи могут функционировать в двух режимах: в качестве читателя или писателя. Программа работает 3 секунды, симулируя 3 читателей и 4 писателей, и выводит результаты на экран.

На рисунке 5.1 представлен фрагмент симуляции.

Изображение выглядит как текст, снимок экрана, схема

Автоматически созданное описание

Рисунок 5.1 – Фрагмент симуляции

На рисунке 5.2 представлен результат симуляции.

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Рисунок 5.2 – Результат работы функции отправки сообщения

Таким образом, программа демонстрирует эффективный механизм межпроцессного взаимодействия модели «читатель-писатель».

# **ЗАКЛЮЧЕНИЕ**

В ходе разработки программы, использующей модель «читатель-писатель» для взаимодействия между потоками, были достигнуты важные результаты. Программа позволяет пользователям функционировать как читатели и писатели, обеспечивая эффективный доступ к разделяемым данным через механизмы синхронизации. Реализованный механизм с использованием мьютексов и семафоров предотвращает конфликты при одновременном чтении и записи, что является критически важным для обеспечения целостности данных в многопоточных приложениях.

В процессе работы над проектом были освоены ключевые аспекты программирования на платформе Windows, включая управление ресурсами через Windows API. Пользователь научился создавать и использовать мьютексы и семафоры для синхронизации потоков, а также работать с критическими секциями для защиты общих данных. Эти знания необходимы для разработки надежных и высокопроизводительных приложений, которые требуют одновременного доступа к ресурсам.

Кроме того, программа демонстрирует принципы работы с многопоточностью, что углубляет понимание взаимодействия между потоками. Пользователь приобрел навыки в написании кода, который обрабатывает одновременные запросы на чтение и запись, а также управляет состоянием потоков, что способствует созданию более сложных и эффективных решений в будущем.

Таким образом, выполненная работа не только улучшила понимание системного программирования и модели «читатель-писатель», но и подготовила к дальнейшему изучению более сложных тем, таких как асинхронное программирование и сетевое взаимодействие. Эти навыки будут полезны в будущих проектах и помогут в разработке приложений, требующих высокой производительности и надежности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Build desktop Windows apps using the Win32 API Microsoft Software Incorp. USA. [Электронный ресурс]. – Режим доступа : https://learn.microsoft.com/en-us/windows/win32/

[2] Основные сообщения ОС Windows (Win32 API). Программирование в ОС Windows Microsoft Software Incorp. Лекция 1. [Электронный ресурс]. – Режим доступа : https://www.youtube.com/watch?v=wTArIolxch0

[3] Разработка с помощью WinAPI. [Электронный ресурс]. – Режим доступа : https://shorturl.at/BDJW8

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <iostream>

#include <windows.h>

#include <thread>

#include <vector>

#include <chrono>

#include <mutex>

#include <atomic>

// Синхронизация потоков

HANDLE semaphoreReaders; // Семафор для читателей

HANDLE mutexWriters; // Мьютекс для писателей

CRITICAL\_SECTION readersCountLock; // Критическая секция для защиты счётчиков

// Общие данные

int readersCount = 0; // Количество активных читателей

int writersWaiting = 0; // Количество ожидающих писателей

int sharedData = 0; // Общая переменная для чтения/записи

// Статистика

std::mutex statsMutex;

int successfulReads = 0;

int successfulWrites = 0;

int failedReads = 0;

int failedWrites = 0;

double totalReadTime = 0.0;

double totalWriteTime = 0.0;

double totalReadBlockTime = 0.0;

double totalWriteBlockTime = 0.0;

// Флаг для завершения потоков

std::atomic<bool> isRunning(true);

// Функция для работы читателя

void reader(int id, int duration) {

while (isRunning) {

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

// Попытка войти в критическую секцию для проверки условий чтения

EnterCriticalSection(&readersCountLock);

if (readersCount == 0 && writersWaiting > 0) {

LeaveCriticalSection(&readersCountLock);

std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

// Увеличиваем счётчик неудачных попыток чтения

{

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

failedReads++;

std::cout << "Читатель " << id << " не смог прочитать (ожидание писателей)." << std::endl;

}

continue;

}

// Увеличиваем количество активных читателей

readersCount++;

LeaveCriticalSection(&readersCountLock);

// Вычисление времени блокировки

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

totalReadBlockTime += std::chrono::duration<double>(blockEndTime - blockStartTime).count();

// Чтение данных

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

std::this\_thread::sleep\_for(std::chrono::milliseconds(duration));

std::cout << "Читатель " << id << " читает данные: " << sharedData << std::endl;

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

// Заканчиваем чтение

EnterCriticalSection(&readersCountLock);

readersCount--;

if (readersCount == 0) {

ReleaseSemaphore(semaphoreReaders, 1, NULL);

}

LeaveCriticalSection(&readersCountLock);

// Обновляем статистику

{

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

successfulReads++;

totalReadTime += std::chrono::duration<double>(endTime - startTime).count();

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

}

}

// Функция для работы писателя

void writer(int id, int duration) {

while (isRunning) {

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

++writersWaiting;

// Ожидание мьютекса для записи

DWORD waitResult = WaitForSingleObject(mutexWriters, 10); // Тайм-аут 10 миллисекунд

if (waitResult == WAIT\_TIMEOUT) {

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

failedWrites++;

--writersWaiting;

std::cout << "Писатель " << id << " не смог дождаться доступа для записи." << std::endl;

continue;

}

// Вычисление времени блокировки

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

totalWriteBlockTime += std::chrono::duration<double>(blockEndTime - blockStartTime).count();

// Запись данных

sharedData++;

std::cout << "Писатель " << id << " записывает данные: " << sharedData << std::endl;

ReleaseMutex(mutexWriters);

--writersWaiting;

// Обновляем статистику

{

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

successfulWrites++;

totalWriteTime += duration / 1000.0;

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

}

}

// Освобождение ресурсов

void cleanup() {

CloseHandle(semaphoreReaders);

CloseHandle(mutexWriters);

DeleteCriticalSection(&readersCountLock);

}

int main() {

setlocale(LC\_ALL, "ru"); // Устанавливаем локаль на русский язык

// Параметры программы

int numReaders = 3; // Количество читателей

int numWriters = 4; // Количество писателей

int readDuration = 200; // Время работы читателя (мс)

int writeDuration = 1000; // Время работы писателя (мс)

// Инициализация синхронизации

semaphoreReaders = CreateSemaphore(NULL, numReaders, numReaders, NULL);

mutexWriters = CreateMutex(NULL, FALSE, NULL);

InitializeCriticalSection(&readersCountLock);

std::vector<std::thread> threads;

// Создаём потоки для читателей

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

threads.emplace\_back(reader, i + 1, readDuration);

}

// Создаём потоки для писателей

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

threads.emplace\_back(writer, i + 1, writeDuration);

}

// Работа программы в течение 3 секунд

std::this\_thread::sleep\_for(std::chrono::seconds(3));

isRunning = false;

// Ожидаем завершения потоков

for (auto& th : threads) {

th.join();

}

// Вывод результатов

std::cout << "\n=== Результаты симуляции ===\n";

std::cout << "Успешные чтения: " << successfulReads << std::endl;

std::cout << "Неудачные чтения: " << failedReads << std::endl;

std::cout << "Успешные записи: " << successfulWrites << std::endl;

std::cout << "Неудачные записи: " << failedWrites << std::endl;

std::cout << "Среднее время чтения: " << (successfulReads ? totalReadTime / successfulReads : 0) << " сек." << std::endl;

std::cout << "Среднее время записи: " << (successfulWrites ? totalWriteTime / successfulWrites : 0) << " сек." << std::endl;

std::cout << "Общее время блокировки чтения: " << totalReadBlockTime << " сек." << std::endl;

std::cout << "Общее время блокировки записи: " << totalWriteBlockTime << " сек." << std::endl;

cleanup(); // Освобождение ресурсов

return 0;

}