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 №3

на тему

**ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ: ОБМЕН ДАННЫМИ**

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 между процессами в многопоточной среде. В ходе выполнения необходимо исследовать подходы к синхронизации доступа к разделяемым ресурсам, использование межпроцессного взаимодействия, а также анализ проблем, возникающих при совместной обработке данных.

Задача состоит в реализации программы, которая:

* организует совместное использование разделяемой памяти для обработки данных;
* обеспечивает синхронизацию доступа к разделяемым ресурсам, предотвращая коллизии;
* сравнивает эффективность различных методов доступа к данным (с использованием разделяемой памяти и традиционного подхода с файлами);
* измеряет время выполнения операций при использовании как разделяемой памяти, так и традиционного способа обработки данных.

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 демонстрирует два подхода к обмену данными между процессами с использованием разделяемой памяти. Первый метод основан на классическом взаимодействии, в то время как второй использует более продвинутый механизм — отображение памяти. Рассмотрим каждый из подходов с точки зрения используемых функций и работы программы.[2]

# **4.1 Описание используемых функций**

Далее представлены функции, используемые для работы с межпроцессным взаимодействием и разделяемой памятью:

1. CreateMutex(): Эта функция создает мьютекс, который используется для синхронизации доступа к разделяемой памяти. Она возвращает дескриптор мьютекса, который необходим для управления доступом к общим ресурсам.
2. CreateFileMapping(): Создает объект сопоставления файла в память, позволяя нескольким процессам обращаться к общему блоку памяти. Это позволяет эффективно обмениваться данными без дополнительных затрат на чтение и запись.
3. MapViewOfFile(): Отображает разделяемую память в адресное пространство процесса, позволяя напрямую работать с ее содержимым. Это ключевой шаг для использования механизма отображения памяти, так как все дальнейшие операции производятся в этом блоке.
4. WaitForSingleObject(): Ожидает, пока мьютекс станет доступным, обеспечивая эксклюзивный доступ к разделяемым ресурсам. Эта функция предотвращает одновременное изменение данных несколькими процессами.
5. ReleaseMutex(): Освобождает захваченный мьютекс, позволяя другим процессам получить доступ к разделяемой памяти. Это важный шаг для поддержания корректности и согласованности данных.
6. memset(): Используется для очистки содержимого разделяемой памяти. Эта функция заполняет указанный блок памяти нулями, что позволяет сбросить данные в буфере.
7. UnmapViewOfFile(): Закрывает отображение памяти, освобождая ресурсы. Это необходимо для завершения работы с общей памятью после завершения всех операций.
8. CloseHandle(): Закрывает дескрипторы объектов, таких как мьютексы и разделяемая память, освобождая системные ресурсы.

Таким образом, в программном продукте представлены основные функции работы с межпроцессным взаимодействием через разделяемую память, что обеспечивает эффективный и безопасный обмен данными между процессами.

# **4.2 Алгоритм работы программы**

Алгоритм работы данной программы включает несколько ключевых этапов, которые обеспечивают эффективное взаимодействие между процессами через разделяемую память.

Программа начинает с инициализации необходимых ресурсов. На первом этапе создается мьютекс с использованием функции CreateMutex(), который играет важную роль в синхронизации доступа к разделяемой памяти. Это гарантирует, что только один процесс или поток может в данный момент изменять данные, что предотвращает возможные коллизии и несоответствия. После этого создается объект сопоставления файла в память с помощью CreateFileMapping(), что позволяет разделять данные между процессами, а затем отображается эта память в адресное пространство текущего процесса с помощью MapViewOfFile().

Когда сообщение введено, программа захватывает мьютекс с помощью WaitForSingleObject(), чтобы обеспечить эксклюзивный доступ к разделяемой памяти. После захвата мьютекса программа проверяет, есть ли свободное место в буфере для хранения сообщения. Если свободное место найдено, сообщение записывается в разделяемую память с использованием функции strncpy\_s(), что позволяет избежать переполнения буфера. После успешной записи сообщения программа выводит на экран уведомление о том, что сообщение отправлено. В случае, если буфер заполнен, программа информирует пользователя о том, что нет доступного места для нового сообщения. После завершения операций записи мьютекс освобождается с помощью ReleaseMutex(), что позволяет другим процессам получить доступ к разделяемой памяти.

Если выбран режим получателя, программа снова захватывает мьютекс, чтобы получить доступ к разделяемой памяти. В этом режиме программа читает и выводит все сообщения, которые находятся в разделе памяти. Каждое сообщение выводится на экран, что позволяет пользователю увидеть все отправленные сообщения. После чтения всех сообщений мьютекс также освобождается, позволяя другим потокам работать с памятью.

Кроме того, программа включает функцию для очистки буфера сообщений. Эта функция, clearMessageBuffer(), позволяет сбрасывать содержимое разделяемой памяти, заменяя все данные нулями.

Когда пользователь завершает работу с программой, все открытые ресурсы должны быть корректно освобождены. Программа завершает свою работу, закрывая отображение памяти с помощью UnmapViewOfFile(), а затем закрывает дескрипторы разделяемой памяти и мьютекса с помощью CloseHandle(). Это гарантирует, что все использованные ресурсы освобождаются.

Таким образом, программа демонстрирует эффективный механизм межпроцессного взаимодействия.

**5 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ**

Программа, реализованная на базе данного кода, обеспечивает обмен сообщениями между процессами через разделяемую память. В результате выполнения приложения пользователи могут отправлять и получать сообщения в двух режимах: как отправитель или как получатель.

На рисунке 5.1 представлено меню приложения.

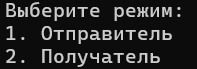


Рисунок 5.1 – Меню приложения

На рисунке 5.2 представлен результат работы функции отправки сообщения.

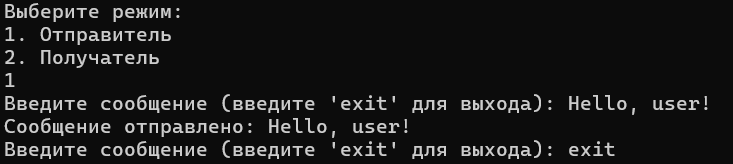


Рисунок 5.2 – Результат работы функции отправки сообщения

На рисунке 5.3 представлен результат работы функции получения сообщения.

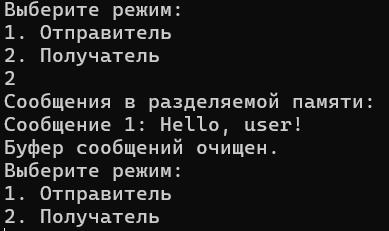


Рисунок 5.3 – Результат работы функции получения сообщения

Таким образом, результат выполнения программы демонстрирует эффективный механизм межпроцессного взаимодействия.

# **ЗАКЛЮЧЕНИЕ**

В ходе разработки программы, использующей разделяемую память для обмена сообщениями между процессами, были достигнуты важные результаты. Программа позволяет пользователям выступать в роли отправителей и получателей, обеспечивая эффективное взаимодействие через общую область памяти. Реализованный механизм синхронизации с использованием мьютексов предотвращает конфликты при доступе к разделяемым данным, что является важным аспектом в многопоточных приложениях.

Во время работы над проектом были освоены ключевые аспекты программирования на платформе Windows, включая управление ресурсами через Windows API. Пользователь научился создавать и использовать объекты сопоставления файлов, а также работать с мьютексами для синхронизации потоков. Это знание критически важно для разработки высокопроизводительных и надежных приложений, которые требуют одновременного доступа к данным.

Кроме того, программа демонстрирует принципы работы с разделяемой памятью и многопоточностью, что углубляет понимание этих концепций. Пользователь приобрел навыки в написании кода, который обрабатывает ввод и вывод сообщений, а также в управлении памятью, что способствует созданию более сложных и эффективных решений в будущем.

Таким образом, выполненная работа не только улучшила понимание системного программирования, но и подготовила к дальнейшему изучению более сложных тем, таких как асинхронный ввод-вывод и сетевое программирование. Эти навыки будут полезны в будущих проектах и помогут в разработке приложений, которые требуют высокой производительности и надежности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Build desktop Windows apps using the Win32 API Microsoft Software Incorp. USA. [Электронный ресурс]. – 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 <windows.h>

#include <iostream>

#include <string>

#define SHARED\_MEM\_SIZE 1024

#define SHARED\_MEM\_NAME L"Local\\MySharedMem"

#define MUTEX\_NAME L"Local\\MyMutex"

#define MAX\_MSG\_COUNT 10

#define SINGLE\_MSG\_SIZE 100

using namespace std;

HANDLE hSharedMem;

LPVOID pSharedMem;

HANDLE hMutex;

char\* msgBuffer;

// Инициализация разделяемой памяти и мьютекса

void initializeSharedMemory() {

setlocale(LC\_ALL, "ru");

hMutex = CreateMutex(NULL, FALSE, MUTEX\_NAME);

if (hMutex == NULL) {

cerr << "Ошибка при создании мьютекса: " << GetLastError() << endl;

exit(EXIT\_FAILURE);

}

hSharedMem = CreateFileMapping(

INVALID\_HANDLE\_VALUE,

NULL,

PAGE\_READWRITE,

0,

SHARED\_MEM\_SIZE,

SHARED\_MEM\_NAME);

if (hSharedMem == NULL) {

cerr << "Не удалось создать объект сопоставления файла: " << GetLastError() << endl;

exit(EXIT\_FAILURE);

}

pSharedMem = MapViewOfFile(hSharedMem, FILE\_MAP\_ALL\_ACCESS, 0, 0, SHARED\_MEM\_SIZE);

if (pSharedMem == NULL) {

cerr << "Не удалось сопоставить представление файла: " << GetLastError() << endl;

CloseHandle(hSharedMem);

exit(EXIT\_FAILURE);

}

msgBuffer = static\_cast<char\*>(pSharedMem);

}

// Запись сообщения в разделяемую память

void sendMessage(const string& msg) {

setlocale(LC\_ALL, "ru");

WaitForSingleObject(hMutex, INFINITE);

int msgCount = 0;

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

if (msgBuffer[i \* SINGLE\_MSG\_SIZE] == '\0') {

strncpy\_s(msgBuffer + (i \* SINGLE\_MSG\_SIZE), SINGLE\_MSG\_SIZE, msg.c\_str(), msg.size());

Sleep(3000);

cout << "Сообщение отправлено: " << msg << endl;

break;

}

msgCount++;

}

if (msgCount == MAX\_MSG\_COUNT) {

cout << "Буфер переполнен!" << endl;

}

ReleaseMutex(hMutex);

}

// Чтение всех сообщений из разделяемой памяти

void receiveMessages() {

setlocale(LC\_ALL, "ru");

WaitForSingleObject(hMutex, INFINITE);

cout << "Сообщения в разделяемой памяти:" << endl;

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

if (msgBuffer[i \* SINGLE\_MSG\_SIZE] != '\0') {

cout << "Сообщение " << (i + 1) << ": " << msgBuffer + (i \* SINGLE\_MSG\_SIZE) << endl;

}

else {

break;

}

}

ReleaseMutex(hMutex);

}

// Очистка буфера

void clearMessageBuffer() {

setlocale(LC\_ALL, "ru");

WaitForSingleObject(hMutex, INFINITE);

memset(msgBuffer, 0, SHARED\_MEM\_SIZE);

cout << "Буфер сообщений очищен." << endl;

ReleaseMutex(hMutex);

}

void messageSender() {

setlocale(LC\_ALL, "ru");

initializeSharedMemory();

string msg;

while (true) {

cout << "Введите сообщение (введите 'exit' для выхода): ";

getline(cin, msg);

if (msg == "exit") {

break;

}

sendMessage(msg);

}

}

void messageReceiver() {

setlocale(LC\_ALL, "ru");

initializeSharedMemory();

receiveMessages();

clearMessageBuffer();

}

int main() {

setlocale(LC\_ALL, "ru");

int option;

while (true) {

cout << "Выберите режим:\n1. Отправитель\n2. Получатель\n";

cin >> option;

cin.ignore();

if (option == 1) {

messageSender();

}

else if (option == 2) {

messageReceiver();

}

else {

break;

}

}

UnmapViewOfFile(pSharedMem);

CloseHandle(hSharedMem);

CloseHandle(hMutex);

return 0;

}