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

**ОТЧЕТ**

к лабораторной работе № 6

на тему «Средства синхронизации и взаимного исключения (Windows). Изучение и использование средств синхронизации и взаимного исключения»

Выполнил:

студент гр. 153504

Князев Н.Д.

Проверил:

Гриценко Н.Ю.

Минск 2023

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc146631498)

[2 Краткие теоретические сведения 4](#_Toc146631499)

[3 Результаты выполнения лабораторной работы 6](#_Toc146631500)

[Выводы 8](#_Toc146631501)

[Список использованных источников 9](#_Toc146631502)

[Приложение А (обязательное) Листинг кода 10](#_Toc146631503)

## 1 ПОСТАНОВКА ЗАДАЧИ

Изучить средства синхронизации и взаимного исключения операционной системы Windows. Изучить средств синхронизации и взаимного исключения. Реализовать многозадачное приложение для моделирования гонки машин, где средства синхронизации используются для синхронизации движения автомобилей.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## Средства синхронизации и взаимного исключения в операционной системе Windows предоставляют механизмы для управления параллельным выполнением процессов и потоков, обеспечивая правильное взаимодействие и координацию между ними. Эти средства позволяют избежать гонок данных и других проблем, связанных с параллельным выполнением. Важно учитывать, что неправильное использование средств синхронизации и взаимного исключения может привести к блокировкам и деградации производительности системы. Для работы с средствами синхронизации и взаимного исключения в Windows разработан ряд функций и объектов. Из наиболее распространенных средств синхронизации следующие: Мьютексы (используются для ограничения доступа к ресурсам только одним потоком или процессом в определенный момент времени. Они широко применяются для синхронизации между процессами.), Семафоры (позволяют контролировать количество потоков, которые могут получить доступ к ресурсу. Это полезное средство для ограничения параллельного доступа к ограниченному количеству ресурсов.), Критические секции (представляют собой участок кода, к которому может получить доступ только один поток в определенный момент времени. Они часто используются для синхронизации внутри одного процесса.), События (позволяют одному потоку сигнализировать другим потокам о наступлении определенного события. Это может быть полезно для ожидания выполнения определенных условий.), Критические ресурсы (может описывать любой ресурс, к которому нужно обеспечить эксклюзивный доступ. Используется с помощью различных механизмов синхронизации).

## 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе выполнения лабораторной работы было реализовано многозадачное приложение для координации потоков, где для синхронизации используются семафоры и мьютексы. Результат работы программы показан на рисунке 3.1.

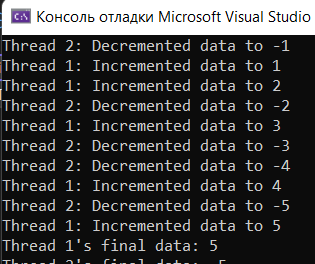


Рисунок 3.1 – Результат работы программы

## ВЫВОДЫ

В ходе выполнения данной лабораторной работы было создано простое многозадачное консольное приложение, которое реализует работу 2ух потоков и синхронизирует их с помощью semaphore и mutex.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Щупак Ю. Win32 API. Разработка приложений для Windows. – СПб: Питер, 2008. – 592 с.: ип.
2. [Электронный ресурс]. – Режим доступа: [https://learn.microsoft.com/ru-ru/windows/win32/procthread/process-and-thread-functions – Дата доступа 01.11.2023](https://learn.microsoft.com/ru-ru/windows/win32/procthread/process-and-thread-functions%20–%20Дата%20доступа%2024.10.2023)
3. [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/8.3.php – Дата доступа: 30.10.2023

**ПРИЛОЖЕНИЕ А**

## (обязательное)

## Листинг кода

**lab6.cpp**

#include <windows.h>

#include <stdio.h>

// Structure to store thread information

struct ThreadData {

int id;

int data;

HANDLE threadHandle;

HANDLE finishedEvent;

};

// Global variables

HANDLE hSemaphore;

HANDLE hMutex;

ThreadData thread1Data;

ThreadData thread2Data;

// Thread function for the first thread

DWORD WINAPI ThreadFunction1(LPVOID lpParam) {

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

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

WaitForSingleObject(hSemaphore, INFINITE);

WaitForSingleObject(hMutex, INFINITE);

data->data++;

printf("Thread %d: Incremented data to %d\n", data->id, data->data);

ReleaseMutex(hMutex);

ReleaseSemaphore(hSemaphore, 1, NULL);

Sleep(1000); // Simulate some work

}

SetEvent(data->finishedEvent); // Signal that the thread has finished

return 0;

}

// Thread function for the second thread

DWORD WINAPI ThreadFunction2(LPVOID lpParam) {

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

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

WaitForSingleObject(hSemaphore, INFINITE);

WaitForSingleObject(hMutex, INFINITE);

data->data--;

printf("Thread %d: Decremented data to %d\n", data->id, data->data);

ReleaseMutex(hMutex);

ReleaseSemaphore(hSemaphore, 1, NULL);

Sleep(1000); // Simulate some work

}

SetEvent(data->finishedEvent); // Signal that the thread has finished

return 0;

}

int main() {

// Initialize the semaphore and mutex

hSemaphore = CreateSemaphore(NULL, 1, 1, NULL);

hMutex = CreateMutex(NULL, FALSE, NULL);

if (hSemaphore == NULL || hMutex == NULL) {

printf("Failed to create synchronization objects\n");

return 1;

}

// Initialize thread data and events

thread1Data.id = 1;

thread2Data.id = 2;

thread1Data.data = 0;

thread2Data.data = 0;

thread1Data.finishedEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

thread2Data.finishedEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

// Create two threads

thread1Data.threadHandle = CreateThread(NULL, 0, ThreadFunction1, &thread1Data, 0, NULL);

thread2Data.threadHandle = CreateThread(NULL, 0, ThreadFunction2, &thread2Data, 0, NULL);

if (thread1Data.threadHandle == NULL || thread2Data.threadHandle == NULL) {

printf("Failed to create threads\n");

return 2;

}

// Wait for threads to finish

WaitForSingleObject(thread1Data.finishedEvent, INFINITE);

WaitForSingleObject(thread2Data.finishedEvent, INFINITE);

// Cleanup

CloseHandle(thread1Data.threadHandle);

CloseHandle(thread2Data.threadHandle);

CloseHandle(thread1Data.finishedEvent);

CloseHandle(thread2Data.finishedEvent);

CloseHandle(hSemaphore);

CloseHandle(hMutex);

printf("Thread 1's final data: %d\n", thread1Data.data);

printf("Thread 2's final data: %d\n", thread2Data.data);

return 0;}