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 Авдошко И.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

[1 Формулировка задачи 3](#_Toc179742261)

[2 Краткие теоритические сведения 3](#_Toc179742262)

[3 Описание функций программы 6](#_Toc179742263)

[Заключение 6](#_Toc179742264)

[Список использованных источников 9](#_Toc179742265)

[Приложение А (обязательное) 10](#_Toc179742266)

# 1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью лабораторной работы является изучение подходов, системных объектов и функций для организации синхронизации и передачи управления между взаимодействующими потоками. В процессе выполнения работы рассматриваются основные модели взаимодействия, такие как «писатели-читатели», и изучаются пути решения типичных проблем, возникающих при одновременном доступе к разделяемым ресурсам, включая предотвращение коллизий, обеспечение согласованности данных и повышение эффективности работы.

Задачей является разработка консольного приложения на языке C++, демонстрирующего корректное управление доступом к разделяемой памяти. Приложение должно моделировать работу нескольких потоков, которые одновременно выполняют различные операции над отдельными блоками памяти, включая:

– Изменение значений в блоке с сохранением целостности данных.

– Вычисление минимального значения в блоке.

– Расчёт среднего значения блока.

Для синхронизации доступа к разделяемым ресурсам должны использоваться мьютексы, предотвращающие одновременную запись и чтение одних и тех же данных разными потоками.

Кроме того, требуется провести серию экспериментов, направленных на оценку эффективности выбранного механизма синхронизации, включая:

1. Измерение времени выполнения операций при увеличении числа потоков.
2. Влияние размера блоков памяти на производительность.

Программа должна предусматривать возможность изменения ключевых параметров системы, таких как общий объём памяти, размер блоков, количество потоков, а также тип выполняемых операций.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Win32 API (Windows API) представляет собой набор функций и интерфейсов, предоставляемых операционной системой Windows для разработки приложений. Этот мощный набор инструментов обеспечивает доступ к различным функциональным возможностям Windows, включая создание и управление окнами, обработку сообщений, работу с файлами и реестром, а также многие другие операции. Win32 API играет ключевую роль в разработке приложений для Windows и обеспечивает высокую степень контроля над поведением приложений. [1]

Объект мьютекса — это объект синхронизации, состояние которого устанавливается в значение Signaled, если он не принадлежит ни одному потоку, и без знака, когда он принадлежит. Только один поток за раз может владеть объектом мьютекса, имя которого происходит из-за того, что он полезен для координации взаимоисключающего доступа к общему ресурсу. Например, чтобы предотвратить запись двух потоков в общую память одновременно, каждый поток ожидает владения объектом мьютекса перед выполнением кода, который обращается к памяти. После записи в общую память поток освобождает объект мьютекса.

Поток использует функцию CreateMutex или CreateMutexEx для создания объекта мьютекса. Создающий поток может запросить немедленное владение объектом мьютекса, а также указать имя для объекта мьютекса. Он также может создать неименованный мьютекс. Дополнительные сведения об именах объектов мьютекса, события, семафора и таймера см. в разделе Синхронизация межпроцессных операций.

Потоки в других процессах могут открывать дескриптор существующего именованного объекта мьютекса, указывая его имя в вызове функции OpenMutex. Чтобы передать дескриптор неименованного мьютекса другому процессу, используйте функцию DuplicateHandle или наследование дескриптора типа "родители-потомки".

Любой поток с дескриптором объекта мьютекса может использовать одну из функций ожидания для запроса владения объектом мьютекса. Если объект мьютекса принадлежит другому потоку, функция ожидания блокирует запрашивающий поток до тех пор, пока поток-владение не освободит объект мьютекса с помощью функции ReleaseMutex. Возвращаемое значение функции wait указывает, была ли функция возвращена по какой-либо причине, кроме состояния мьютекса, для которой задано значение signaled.

Если для мьютекса ожидается несколько потоков, выбирается ожидающий поток. Не предполагайте порядок "первым в систему" (FIFO). Внешние события, такие как APC в режиме ядра, могут изменить порядок ожидания.

После того как поток получает право владения мьютексом, он может указать тот же мьютекс в неоднократных вызовах функций ожидания, не блокируя его выполнение. Это предотвращает взаимоблокировку потока при ожидании мьютекса, которым он уже владеет. Чтобы освободить свое владение в таких обстоятельствах, поток должен вызывать ReleaseMutex один раз для каждого случая, когда мьютекс удовлетворяет условиям функции ожидания. [2]

Функции ожидания позволяют потоку блокировать собственное выполнение. Функции ожидания не возвращаются до тех пор, пока не будут выполнены указанные критерии. Тип функции ожидания определяет набор используемых критериев. При вызове функции ожидания она проверяет, выполнены ли критерии ожидания. Если критерии не выполнены, вызывающий поток переходит в состояние ожидания до тех пор, пока не будут выполнены условия критериев ожидания или не истечёт указанный интервал ожидания.

Функции SignalObjectAndWait, WaitForSingleObject и WaitForSingleObjectEx требуют наличия дескриптора одного объекта синхронизации. Эти функции возвращаются, когда происходит одно из следующих событий:

– Указанный объект находится в сигнальном состоянии.

– Истекает интервал ожидания. Интервал ожидания можно установить на бесконечный, чтобы указать, что ожидание не будет прерываться.

Функция SignalObjectAndWait позволяет вызывающему потоку атомарно установить состояние объекта в «сигнализировано» и дождаться, пока состояние другого объекта не станет «сигнализировано». [3]

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Программа демонстрирует параллельную согласованную работу потоков с использованием разделяемой памяти и механизмов синхронизации, обеспечивая корректный доступ нескольких потоков к разделяемым блокам памяти и исключая возможность коллизий. Для организации доступа к памяти используется синхронизация через мьютексы. Каждый блок памяти защищён своим мьютексом, что позволяет избежать одновременного доступа разных потоков к одному и тому же блоку данных. Мьютексы создаются с помощью функции CreateMutex, которая инициализирует объекты синхронизации для каждого блока данных, обеспечивая тем самым безопасность доступа.

Каждый поток выполняет одну из задач: нахождение минимума в блоке, вычисление среднего значения чисел блока или изменение первого элемента блока. Потоки создаются функцией CreateThread, которая запускает указанные задачи в многозадачной среде. Все потоки работают с различными блоками памяти, что позволяет им параллельно выполнять операции без конфликтов, при этом доступ к каждому блоку данных синхронизируется с помощью мьютекса. Пример выполнения кода программы продемонстрирован на рисунке 3.1.

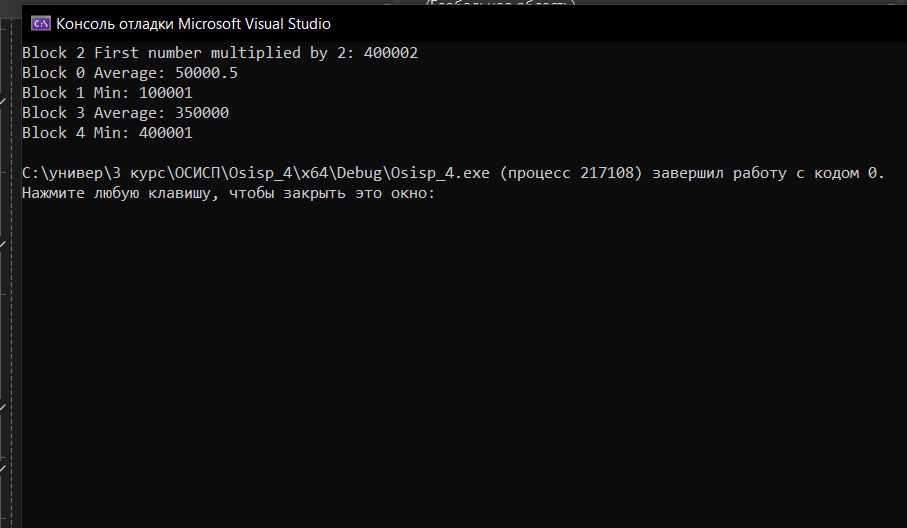


Рисунок 3.1 – Выполнение программы

Для того чтобы поток мог безопасно работать с блоком, ему необходимо захватить соответствующий мьютекс, что выполняется с помощью функции WaitForSingleObject. Эта функция блокирует поток до тех пор, пока мьютекс не будет доступен, что гарантирует, что только один поток будет работать с данным блоком в момент времени. После выполнения необходимой операции, например, вычисления минимального значения или изменения первого элемента блока, поток освобождает мьютекс через функцию ReleaseMutex, что позволяет другим потокам получить доступ к этому блоку. Таким образом, вся работа с блоками памяти строго синхронизирована, и потоки не могут одновременно изменять одни и те же данные.

После того как все потоки завершат выполнение своих задач, используется функция WaitForMultipleObjects, чтобы главный поток мог дождаться завершения работы всех дочерних потоков. Это необходимо для того, чтобы программа корректно завершила свою работу, не прерывая выполнение потоков до их завершения. По завершении работы все потоки и мьютексы закрываются функцией CloseHandle, что освобождает системные ресурсы и предотвращает утечки памяти.

Программа реализует механизм синхронизации с использованием стандартных средств Windows API и эффективно организует параллельное выполнение задач, позволяя безопасно и корректно взаимодействовать нескольким потокам с разделяемыми данными.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была разработана программа, демонстрирующая синхронизированную работу нескольких потоков с разделяемой памятью. Основной целью работы было изучение механизмов синхронизации в многозадачной среде и обеспечение корректного доступа потоков к общим данным с минимизацией коллизий. В процессе реализации были использованы средства синхронизации Windows API, такие как мьютексы, для защиты разделяемых блоков памяти от одновременного доступа со стороны нескольких потоков.

Программа выполняет различные операции с данными в разделяемой памяти, включая вычисление минимального значения, среднего и изменение элементов, с обеспечением синхронизации доступа к этим данным. Для этого были использованы функции CreateThread для создания потоков, CreateMutex для создания мьютексов, WaitForSingleObject для ожидания освобождения мьютекса и ReleaseMutex для освобождения мьютекса после выполнения задачи. Для ожидания завершения всех потоков использована функция WaitForMultipleObjects, а для очистки ресурсов — функция CloseHandle.

Реализованная система позволяет эффективно работать с несколькими потоками, исключая возможность конфликтов при параллельной обработке данных. Также были проведены эксперименты с различными параметрами системы, что позволило наглядно продемонстрировать эффективность синхронизации и предотвращение коллизий при доступе к разделяемой памяти. В итоге, задача организации безопасного и синхронизированного взаимодействия между потоками была успешно решена с использованием функционала Windows API.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Щупак Ю. Win32 API. Разработка приложений для Windows. – СПб: Питер, 2008. – 592 с.

[2] Microsoft. "Объекты синхронизации" [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/sync/synchronization-objects.

[3] Microsoft. "Wait Functions" [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/sync/wait-functions.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <windows.h>

#include <iostream>

#include <vector>

#include <algorithm>

#include <random>

#include <ctime>

#include <numeric>

// Параметры модели

constexpr size\_t MEMORY\_SIZE = 1000000; // Общий объём памяти

constexpr size\_t BLOCK\_SIZE = 100000; // Размер блока

constexpr size\_t THREAD\_COUNT = 5; // Количество потоков

// Глобальные переменные

std::vector<int> sharedData(MEMORY\_SIZE); // Общий ресурс

std::vector<HANDLE> blockMutexes; // Мьютексы для блоков памяти

// Структура для передачи параметров в поток

struct Task {

size\_t blockIndex; // Индекс блока

int taskType; // Тип задачи: 1 - среднее значение, 2 - минимальное значение, 3 - умножение первого числа

};

// Функция для обработки блока памяти

DWORD WINAPI ThreadFunction(LPVOID lpParam) {

Task\* task = static\_cast<Task\*>(lpParam);

size\_t blockIndex = task->blockIndex;

int taskType = task->taskType;

// Синхронизация доступа к блоку

WaitForSingleObject(blockMutexes[blockIndex], INFINITE);

// Получаем указатель на начало блока

auto blockStart = sharedData.begin() + blockIndex \* BLOCK\_SIZE;

auto blockEnd = blockStart + BLOCK\_SIZE;

if (taskType == 1) {

// Вычисление среднего значения

long long sum = std::accumulate(blockStart, blockEnd, 0LL);

double average = static\_cast<double>(sum) / BLOCK\_SIZE;

std::cout << "Block " << blockIndex << " Average: " << average << std::endl;

}

else if (taskType == 2) {

// Нахождение минимального значения

int minVal = \*std::min\_element(blockStart, blockEnd);

std::cout << "Block " << blockIndex << " Min: " << minVal << std::endl;

}

else if (taskType == 3) {

// Умножение первого числа на случайное значение от 1 до 10

std::default\_random\_engine generator(static\_cast<unsigned>(time(NULL)));

std::uniform\_int\_distribution<int> distribution(1, 10);

int randomMultiplier = distribution(generator);

\*blockStart \*= randomMultiplier;

std::cout << "Block " << blockIndex << " First number multiplied by " << randomMultiplier << ": " << \*blockStart << std::endl;

}

// Освобождаем мьютекс блока

ReleaseMutex(blockMutexes[blockIndex]);

return 0;

}

int main() {

// Инициализация общего ресурса

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

sharedData[i] = static\_cast<int>(i + 1);

}

// Создание мьютексов для блоков

size\_t blockCount = MEMORY\_SIZE / BLOCK\_SIZE;

blockMutexes.resize(blockCount);

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

blockMutexes[i] = CreateMutex(NULL, FALSE, NULL);

}

// Создание задач

std::vector<Task> tasks(THREAD\_COUNT);

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

tasks[i].blockIndex = i % blockCount; // Каждому потоку назначается блок

tasks[i].taskType = (i % 3) + 1; // Тип задачи чередуется

}

// Запуск потоков

std::vector<HANDLE> threads(THREAD\_COUNT);

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

threads[i] = CreateThread(NULL, 0, ThreadFunction, &tasks[i], 0, NULL);

}

// Ожидание завершения потоков

WaitForMultipleObjects(THREAD\_COUNT, threads.data(), TRUE, INFINITE);

// Закрытие потоков и мьютексов

for (HANDLE thread : threads) {

CloseHandle(thread);

}

for (HANDLE mutex : blockMutexes) {

CloseHandle(mutex);

}

return 0;

}