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 Павлович В.Ю.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc182342178)

[2 Краткие теоретические сведения 4](#_Toc182342179)

[3 Описание работы программы 5](#_Toc182342180)

[3.1 Писатели 5](#_Toc182342181)

[3.2 Читатели 5](#_Toc182342182)

[3.3 Работа программы 5](#_Toc182342183)

[Заключение 7](#_Toc182342184)

[Список использованных источников 8](#_Toc182342185)

[Приложение А (Обязательное) Исходный код программы 9](#_Toc182342186)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью лабораторной работы является изучение подходов, системных объектов и функций, предназначенных для обеспечения синхронизации и передачи управления между взаимодействующими процессами. Типичных проблем, возникающих при организации взаимодействия, модели для их описания, пути их решения.

В качестве задачи необходимо разработать приложение, демонстрирующее параллельную согласованную работу процессов (потоков) и их взаимодействие. Анализ корректности (отсутствия коллизий). Оценка эффективности механизмов синхронизации (ISO).

Необходимо реализовать модель взаимодействия процессов (потоков) «писатели читатели» с возможностью параметризации и наглядного (не обязательно графического) представления результатов.

Программа должна обеспечивать корректное функционирование, т.е. избегание как «грязного» считывания данных и одновременно минимизацию блокировок.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Объект синхронизации — это объект, дескриптор которого можно указать в одной из функций ожидания для координации выполнения нескольких потоков. Несколько процессов могут иметь дескриптор одного и того же объекта синхронизации, что делает возможной синхронизацию между процессами.

Следующие типы объектов предоставляются исключительно для синхронизации:

– событие – уведомляет один или более ожидающих потоков о том, что произошло событие;

– mutex – может принадлежать только одному потоку за раз, что позволяет потокам координировать взаимоисключающий доступ к общему ресурсу;

– semaphore – поддерживает число от нуля до некоторого максимального значения, ограничивая количество потоков, которые одновременно обращаются к общему ресурсу;

– таймер ожидания – уведомляет один или несколько ожидающих потоков о получении указанного времени.

В некоторых случаях в качестве объекта синхронизации можно также использовать файл, именованный канал или устройство связи. однако их использование для этой цели не рекомендуется. Вместо этого лучше использовать асинхронный ввод-вывод и дожидаться объекта события, заданного в структуре OVERLAPPED. Безопаснее использовать объект события из-за путаницы, которая может возникнуть при выполнении нескольких одновременных перекрывающихся операций в одном файле, именованном канале или устройстве связи. В этом случае невозможно узнать, какая операция вызвала сигнал о состоянии объекта.[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.[2]

Объект семафора — это объект синхронизации, который поддерживает число от нуля до указанного максимального значения. Число уменьшается каждый раз, когда поток завершает ожидание объекта семафора, и увеличивается каждый раз, когда поток освобождает семафор. Когда число достигает нуля, больше потоки не могут ждать, пока состояние объекта семафора станет сигналом. Состояние семафора становится сигнальным, когда это число становится больше нуля, и несигнальным, когда равно нулю.[3]

3 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

В этом разделе будут рассмотрены основные функции программы, которые были задействованы для выполнения работы, а также детали реализации модели читатели-писатели. Для реализации данной модели используются как семафоры, так и мьютексы. Мьютексы используются для безопасного обращения читателей к семафорам, а также для ограничения доступа к файлу для всех потоков, пока писатель записывает в него данные. Семафоры используются для ограничения количества читателей или писателей, одновременно работающих с файлом.

## **3.1 Писатели**

Функция Writer реализует модель писателя, данная функция с помощью мьютекса блокирует обращение других потоков к файлу, после чего происходит запись данных файл. В процессе работы данная функция выводит в консоль информацию о времени ожидания, а также о текущем состоянии самого процесса писателя.

## **3.2 Читатели**

Функция Reader реализует модель читателя. Для ограничения числа одновременно работающих читателей используется семафор, доступ к которому ограничивается мьютексом для обеспечения безопасности работы с ним. После освобождения семафора данная функция считывает данные из файла и выводит информацию о считанных данных и времени ожидания в консоль.

## **3.3 Работа программы**

Для работы используются 2 текстовых файлов, для каждого из которых случайным образом определяется количество читателей и писателей. Далее происходит непосредственно реализация модели писатели-читатели, в которой одновременно доступ к данным может иметь несколько читателей, но при этом несколько писателей не могут одновременно изменять содержимое одного и того же файла. В процессе работы в консоль выводится информация о состоянии каждого конкретного процесса читателя или писателя. Результат работы программы представлен на рисунке 3.1

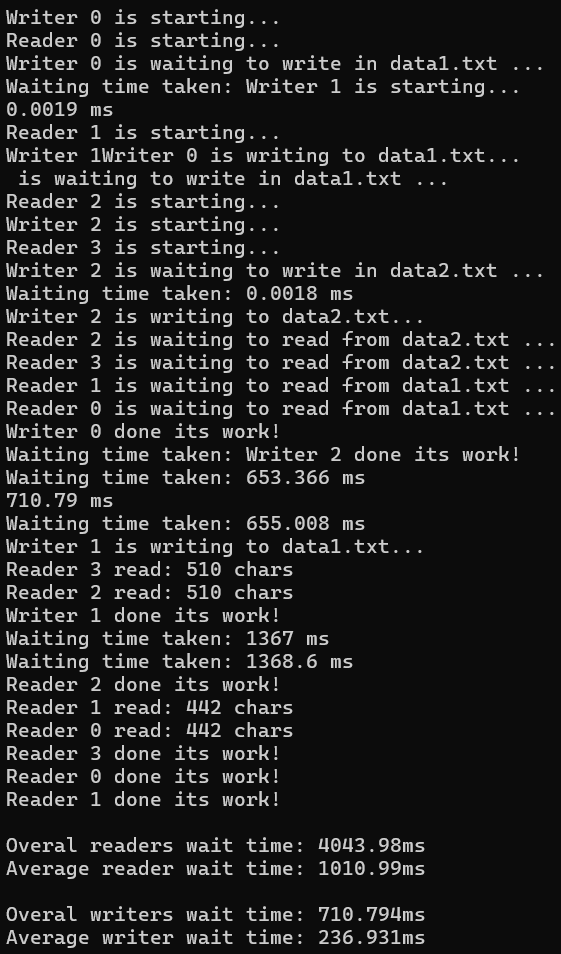


Рисунок 3.1 – Результат работы программы

В выходных данных описывается состояние каждого процесса. В конце подсчитывается общее и среднее время ожидания как для процессов-читателей, так и для процессов-писателей.

# **ЗАКЛЮЧЕНИЕ**

В ходе выполнения лабораторной работы была реализована модель взаимодействия потоков по схеме «Писатели-Читатели», применяя механизмы синхронизации и работу с разделяемыми данными. Спроектированные модули reader и writer гарантируют безопасный доступ к общим ресурса, в данном случае к файлам. Для предотвращения конкурентного доступа используются мьютексы и критические секции, что исключает возможность возникновения гонок и «грязного» чтения данных.

Тестирование продемонстрировало, что система корректно управляет доступом потоков к общим ресурсам, а также отслеживает количество успешных и неудачных операций чтения и записи. В результате была собрана информация о времени ожидания для каждого типа потоков.

Таким образом, лабораторная работа на практике продемонстрировала, как организовать доступ к общим ресурсам в многопоточных приложениях, используя синхронизацию и работу с разделяемой памятью в среде Windows.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Объекты синхронизации – Win32 apps [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/sync/synchron  
ization-objects.

[2] Объекты мьютексов – Win32 apps [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/sync/mutex-objects>.

[3] Объекты семафора – Win32 apps [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/sync/semaphore-objects.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <windows.h>

#include <iostream>

#include <chrono>

#include <cmath>

#define MIN\_BUFFER\_SIZE 2048

int lettersAsync = 0;

int letters = 0;

void CALLBACK ReadCompletionRoutine(DWORD dwErrorCode, DWORD dwNumberOfBytesTransferred, LPOVERLAPPED lpOverlapped) {

if (dwErrorCode == 0) {

lpOverlapped->Offset += dwNumberOfBytesTransferred;

char\* buffer = (char\*)lpOverlapped->hEvent;

buffer[dwNumberOfBytesTransferred] = '\0';

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

if ((buffer[i] >= 65 && buffer[i] <= 90) || (buffer[i] >= 97 && buffer[i] <= 122)) {

lettersAsync++;

}

}

}

else {

if (dwErrorCode == 38) {

return;

}

std::cerr << "Error reading file: " << dwErrorCode << std::endl;

}

}

void processAsync(int bufferSize) {

HANDLE hFile = CreateFile(

L"data.txt",

GENERIC\_READ,

0,

NULL,

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL | FILE\_FLAG\_OVERLAPPED,

NULL

);

if (hFile == INVALID\_HANDLE\_VALUE) {

std::cerr << "Error opening file: " << GetLastError() << std::endl;

return;

}

char\* buffer = new char[bufferSize];

OVERLAPPED overlapped = { 0 };

overlapped.hEvent = (HANDLE)buffer;

while (true) {

DWORD prev\_offset = overlapped.Offset;

if (!ReadFileEx(hFile, buffer, bufferSize - 1, &overlapped, ReadCompletionRoutine)) {

std::cerr << "Error initiating read: " << GetLastError() << std::endl;

CloseHandle(hFile);

return;

}

SleepEx(INFINITE, TRUE);

if (prev\_offset == overlapped.Offset) {

break;

}

}

CloseHandle(hFile);

delete[] buffer;

}

void process() {

HANDLE hFile = CreateFile(

L"data.txt",

GENERIC\_READ,

0,

NULL,

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

if (hFile == INVALID\_HANDLE\_VALUE) {

std::cerr << "Could not open file (Error: " << GetLastError() << ")." << std::endl;

return;

}

DWORD fileSize = GetFileSize(hFile, NULL);

if (fileSize == INVALID\_FILE\_SIZE) {

std::cerr << "Could not get file size (Error: " << GetLastError() << ")." << std::endl;

CloseHandle(hFile);

return;

}

char\* buffer = new char[fileSize + 1];

if (!buffer) {

std::cerr << "Memory allocation failed." << std::endl;

CloseHandle(hFile);

return;

}

DWORD bytesRead;

if (!ReadFile(hFile, buffer, fileSize, &bytesRead, NULL)) {

std::cerr << "Could not read file (Error: " << GetLastError() << ")." << std::endl;

delete[] buffer;

CloseHandle(hFile);

return;

}

buffer[bytesRead] = '\0';

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

if ((buffer[i] >= 65 && buffer[i] <= 90) || (buffer[i] >= 97 && buffer[i] <= 122)) {

letters++;

}

}

delete[] buffer;

CloseHandle(hFile);

}

int main() {

int i = 0;

bool exit = false;

while(!exit) {

int buff\_exp = pow(2, i);

i++;

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

if (MIN\_BUFFER\_SIZE \* buff\_exp >= 1e9) {

exit = true;

}

processAsync(MIN\_BUFFER\_SIZE \* buff\_exp);

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

std::chrono::duration<double> durationAsync = endAsync - startAsync;

std::cout << "Async: Time: " << durationAsync.count() << " " << "Buffer size: " << MIN\_BUFFER\_SIZE \* buff\_exp << " " << "Value: " << lettersAsync << "\n";

lettersAsync = 0;

}

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

process();

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

std::chrono::duration<double> duration = end - start;

std::cout << "Default: Time: " << duration.count() << " " << "Value: " << letters << "\n";

return 0;

}