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**

Студент Севрюков С.И.

Преподаватель Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc184749773)

[2 Краткие теоритические сведения 4](#_Toc184749774)

[3 Описание функций программы 5](#_Toc184749775)

[3.1 Используемые ISO 5](#_Toc184749776)

[3.2 Создание потоков 5](#_Toc184749777)

[3.3 Функция philosopher(LPVOID lpParam) 5](#_Toc184749778)

[Заключение 8](#_Toc184749786)

[Список использованных источников 9](#_Toc184749787)

[Приложение А (обязательное) Исходный код программы 10](#_Toc184749788)

# **1 ФОРМУЛИРОВКА ЗАДАЧИ**

Целью выполнения лабораторной работы является разбор подходов, системных объектов и функций для обеспечения синхронизации и передачи управления между взаимодействующими процессами. Рассмотреть типичные проблемы, возникающие при организации взаимодействия, и пути их решения.

В качестве задачи необходимо реализовать приложение, демонстрирующее параллельную согласованную работу процессов (потоков) и их взаимодействие. Произвести анализ корректности (отсутствия коллизий). Оценить эффективность механизмов синхронизации (*ISO*).

Нужно реализовать модель взаимодействия процессов (потоков) «писатели-читатели» с возможностью параметризации и наглядного представления результатов. Так как речь идет о модели, реальные данные не обязательны, можно ограничиться моделированием обращений к ним (обращение характеризуется параметрами запроса, моментом обращения,длительностью исполнения).

Нужно обеспечить корректное функционирование, т.е. избегание как «грязного» считывания данных и одновременно минимизировать блокировки.

Изменяемые параметры модели: количество «писателей» и «читателей»;интенсивность их обращений к ресурсам, длительность использования ресурса, размер блока данных и т.п. (характеристики случайных величин при моделировании).

В качестве результатов моделирования: соотношение времени активности/блокировки участников; соотношение успешных/неуспешных обращений; общая эффективность (пропускная способность) по «записи» и «чтению».

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Критический ресурс – такой ресурс системы, который не может использоваться одновременно более чем заданным числом пользователей. Часто речь идет о доступности его не более чем одному пользователю.

Критическая секция ‒ это участок кода программы, который запрашивает монопольный доступ к каким-либо общим ресурсам, которые не могут быть использованы в одно время, больше чем одним потоком реализации. Если в критической секции состоит более чем один процесс, тогда обнаруживается состояние так называемой «гонки» [1].

Межпроцессное взаимодействие – обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности *IPC*. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети [2].

Общая идея *IPC*: проверка и модификация некоторого признака (флага) перед доступом к критическому ресурсу. Значение флага отражает свободное или занятое состояние ресурса. Основная проблема: регулятор доступа к критическим ресурсам – тоже критический ресурс, обращения к нему – критическая секция. Рекурсивное замыкание требований.

Семафор – глобальная переменная-счетчик *S*, целочисленные неотрицательные значения, атомарно выполняемые примитивы для доступа *P(S)* и *V(S)*:

1 *P(S)* – условный декремент: если значение достигло 0, то ожидание ненулевого значения.

2 *V(S)* – безусловный инкремент счетчика.

Мьютекс – можно рассматривать как упрощенный двузначный семафор, состояние которого интерпретируют как «свободность» и «занятость», а примитивы доступа – «захват» и «освобождение». Попытка повторного захвата блокирует поток-инициатор до освобождения мьютекса другим потоком.

Барьер (*barrier*) – объект *ISO*, обеспечивающий синхронизацию достигших его процессов (потоков): каждый из них посредством системного вызова запрашивает синхронизацию барьером и переводится в состояние ожидания до тех пор, пока количество таких запросов не достигнет заранее заданного. После этого все получают возможность выполняться дальше, а барьер «освобождается».

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Для реализации параллельной согласованной работы процессов (потоков) и их взаимодействие будем создавать два файла, над которыми будут конкурировать потоки для чтения и записи.

## **3.1 Используемые *ISO***

В качестве ресурсов синхронизации используются следующие объекты:

1 *CRITICAL\_SECTION\* forks* – критические секции для каждой вилки (используется при *CriticalSection*).

2 *HANDLE\* mutex\_forks* – мьютексы для каждой вилки (используется при *Mutex+Semaphore*).

3 *HANDLE semaphore* – семафор, ограничивающий доступ к ресурсам (используется при *Mutex+Semaphore*).

Эти ресурсы синхронизации позволяют корректно организовать взаимодействие философов, избегая взаимоблокировок и обеспечивая равномерное распределение ресурсов.

## **3.2 Создание потоков**

Идет формирование последовательности потоков для выполнения:

**1 Подготовка идентификаторов потоков. Идентификаторы потоков хранятся в массиве *HANDLE*\* *philosopher\_threads*. Каждый элемент массива будет указывать на созданный поток.**

**2 Подготовка данных для каждого потока. Для передачи индивидуального идентификатора философа каждому потоку создаётся массив *ids*. Этот массив содержит индексы философов (0, 1, 2, ...).**

**3 Вызов функции *CreateThread*. Для каждого философа создаётся поток, выполняющий функцию *philosopher*.**

**4 Проверка успешного создания потоков. Если поток не создаётся, программа выводит сообщение об ошибке и завершает работу.**

## **3.3 Функция *philosopher(LPVOID lpParam)***

Функция *philosopher* моделирует поведение философа в задаче "обедающие философы". Каждый философ — это отдельный поток, работающий в бесконечном цикле, где он чередует размышление и попытки взять вилки для еды. Эта функция принимает идентификатор философа, чтобы отличать их друг от друга. Параметры: *lpParam*: указатель на данные, передаваемые потоку. В данной программе это указатель на идентификатор философа (*int*\* *id*). Возвращаемое значение: тип *DWORD*. Возвращаемое значение не используется, так как потоки работают в бесконечном цикле. (рисунок 3.1).

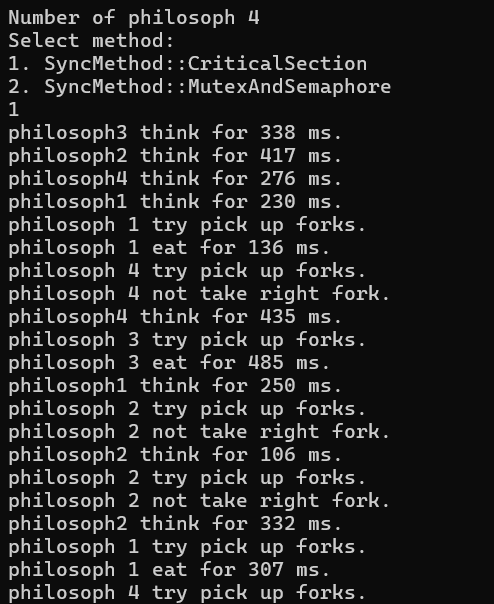


Рисунок 3.1 – Результаты работы философа

**3.4 Функция *CriticalSection* и *MutexAndSemaphore***

*CriticalSection* — механизм синхронизации на уровне процесса. Он обеспечивает эксклюзивный доступ одного потока к ресурсу. Основные функции:

1 *InitializeCriticalSection*. Инициализирует критическую секцию. Вызывается один раз для каждого ресурса (в данном случае для каждой вилки).

2 *EnterCriticalSection*. Позволяет потоку войти в критическую секцию. Если другой поток уже находится внутри, текущий поток блокируется, пока ресурс не освободится.

3 *TryEnterCriticalSection*. Пытается войти в критическую секцию без блокировки. Если ресурс занят, возвращает *false*, иначе позволяет войти.

4 *LeaveCriticalSection*. Поток выходит из критической секции, освобождая ресурс.

5 *DeleteCriticalSection*. Удаляет объект критической секции после завершения работы.

Как работает в коде. Функция *CriticalSection* используется, когда философ хочет взять вилки: левая вилка блокируется через *EnterCriticalSection*, правая вилка проверяется с помощью *TryEnterCriticalSection*. Если правая вилка недоступна, философ освобождает левую вилку и увеличивает счётчик блокировок. Если обе вилки доступны, философ "ест", затем освобождает обе вилки.

*MutexAndSemaphore*: эта комбинация использует мьютексы и семафоры для синхронизации ресурсов.

*Mutex* (мьютуальный исключитель) обеспечивает доступ к ресурсу только одному потоку в любой момент времени. Основные функции:

1 *CreateMutex*. Создаёт объект мьютекса. Используется для каждой вилки.

2 *WaitForSingleObject*. Захватывает мьютекс. Если мьютекс уже занят, поток блокируется до его освобождения.

3 *ReleaseMutex*. Освобождает мьютекс, позволяя другим потокам получить доступ к ресурсу.

*Semaphore*. Семафор ограничивает количество потоков, которые могут одновременно получить доступ к определённому ресурсу. Основные функции:

1 *CreateSemaphore*. Создаёт объект семафора с заданным максимальным и текущим счётчиком.

2 *WaitForSingleObject*. Уменьшает счётчик семафора на 1. Если счётчик равен 0, поток блокируется.

3 *ReleaseSemaphore*. Увеличивает счётчик семафора, позволяя новому потоку войти.

Как работает в коде. Комбинация *Mutex* и *Semaphore* используется для ограничения числа философов, пытающихся одновременно взять вилки. Семафор (*semaphore*) ограничивает доступ, гарантируя, что одновременно может есть не более *n*−1 философов (где *n* — общее число философов). Каждая вилка синхронизируется с помощью мьютекса. Поток захватывает мьютекс для левой вилки, а затем проверяет, доступна ли правая вилка. После завершения работы философ освобождает место в семафоре.

ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы было реализовано приложение, демонстрирующее решение задачи об обедающих философах с использованием механизмов синхронизации потоков. Программа позволяет моделировать взаимодействие нескольких философов, которые поочередно думают и едят, используя общие ресурсы (вилки), и гарантирует корректность доступа к этим ресурсам.

Для предотвращения гонок данных и взаимоблокировок использовались два метода синхронизации:

*CriticalSection*: обеспечивает синхронизацию через критические секции, предотвращая совместный доступ к вилкам со стороны двух философов.

*Mutex* и *Semaphore*: синхронизация достигается за счет использования мьютексов для управления доступом к вилкам и семафоров для ограничения общего числа одновременно едящих философов.

В качестве результата получили корректное функционирование программы с различными параметрами. Статистика по каждому философу включает: количество циклов еды, количество циклов размышлений, количество блокировок (невозможности взять вилки).

При использовании *CriticalSection*, философы редко блокировались, однако возникало время простоя, связанное с попытками взять обе вилки.

При использовании *Mutex* и *Semaphore* удалось добиться более равномерного распределения доступа к ресурсам, что уменьшило вероятность взаимоблокировок. Однако использование дополнительных объектов синхронизации (семафоров) увеличило накладные расходы.

При интенсивной работе потоков время ожидания философов увеличивалось, но мьютексы и семафоры позволяли минимизировать это время.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Требования к Критической секции [Электронный ресурс]. – Электронный ресурс. – Режим доступа: https://scienceforum.ru/

[2] Межпроцессное взаимодействие [Электронный ресурс]. – Электронный ресурс. – Режим доступа: https://ru.wikipedia.org/wiki/

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <windows.h>

#include <fstream>

#include <vector>

#include <sstream>

#include <numeric>

#include <algorithm>

#include <chrono>

#include <cmath>

#include <iostream>

#define BASE\_BUFFER\_SIZE 256

using namespace std;

// Статистические глобальные данные

DWORD asyncExecCount = 0;

DWORD syncExecCount = 0;

int totalAsyncReads = 0;

chrono::duration<double> asyncReadDuration, processingDuration, asyncWriteDuration;

// Подсчёт статистики (среднее, стандартное отклонение, минимум, максимум)

void computeStatistics(const vector<int>& data, double& mean, double& stddev, int& minVal, int& maxVal) {

if (data.empty()) return;

int totalElements = data.size();

mean = accumulate(data.begin(), data.end(), 0.0) / totalElements;

minVal = \*min\_element(data.begin(), data.end());

maxVal = \*max\_element(data.begin(), data.end());

double variance = 0.0;

for (int value : data) {

variance += pow(value - mean, 2);

}

stddev = sqrt(variance / totalElements);

}

// Генерация тестового файла с числами

void createTestData(const wstring& filename, int numElements) {

ofstream outFile(filename);

if (outFile) {

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

outFile << (numElements - i) << " ";

}

outFile.close();

}

}

// Сохранение статистики в файл

void writeStatsToFile(double mean, double stddev, int minVal, int maxVal, const wstring& filename) {

HANDLE fileHandle = CreateFile(

filename.c\_str(),

GENERIC\_WRITE,

FILE\_APPEND\_DATA,

nullptr,

OPEN\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

nullptr

);

if (fileHandle != INVALID\_HANDLE\_VALUE) {

DWORD writtenBytes;

wstring statsContent = L"Average: " + to\_wstring(mean) + L", StdDev: " + to\_wstring(stddev) +

L", Min: " + to\_wstring(minVal) + L", Max: " + to\_wstring(maxVal) + L"\n";

WriteFile(fileHandle, statsContent.c\_str(), statsContent.size() \* sizeof(wchar\_t), &writtenBytes, nullptr);

CloseHandle(fileHandle);

}

}

// Асинхронный колбэк для завершения чтения

void CALLBACK onAsyncReadComplete(DWORD error, DWORD bytesRead, LPOVERLAPPED overlapped) {

if (error == 0) {

overlapped->Offset += bytesRead;

char\* buffer = reinterpret\_cast<char\*>(overlapped->hEvent);

buffer[bytesRead] = '\0';

vector<int> values;

istringstream streamData(buffer);

int tempValue;

while (streamData >> tempValue) {

values.push\_back(tempValue);

}

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

double average, stddev;

int minVal, maxVal;

computeStatistics(values, average, stddev, minVal, maxVal);

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

processingDuration = procEnd - procStart;

procStart = chrono::high\_resolution\_clock::now();

writeStatsToFile(average, stddev, minVal, maxVal, L"async\_results.txt");

procEnd = chrono::high\_resolution\_clock::now();

asyncWriteDuration = procEnd - procStart;

}

}

// Выполнение асинхронного чтения

void performAsyncOperations(int bufferSize) {

HANDLE fileHandle = CreateFile(L"async\_data.txt", GENERIC\_READ, 0, nullptr, OPEN\_EXISTING, FILE\_FLAG\_OVERLAPPED, nullptr);

if (fileHandle == INVALID\_HANDLE\_VALUE) return;

char\* buffer = new char[bufferSize];

OVERLAPPED overlapStruct = { 0 };

overlapStruct.hEvent = reinterpret\_cast<HANDLE>(buffer);

ofstream clearFile(L"async\_results.txt");

clearFile.close();

while (true) {

DWORD oldOffset = overlapStruct.Offset;

if (!ReadFileEx(fileHandle, buffer, bufferSize - 1, &overlapStruct, onAsyncReadComplete)) {

CloseHandle(fileHandle);

delete[] buffer;

return;

}

totalAsyncReads++;

SleepEx(INFINITE, TRUE);

DWORD fileSize = GetFileSize(fileHandle, nullptr);

if (fileSize <= overlapStruct.Offset || oldOffset == overlapStruct.Offset) {

break;

}

}

CloseHandle(fileHandle);

delete[] buffer;

}

// Выполнение синхронного чтения и статистики

DWORD performSyncOperations() {

HANDLE fileHandle = CreateFile(L"sync\_data.txt", GENERIC\_READ, 0, nullptr, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, nullptr);

if (fileHandle == INVALID\_HANDLE\_VALUE) return -1;

DWORD fileSize = GetFileSize(fileHandle, nullptr);

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

DWORD bytesRead;

if (!ReadFile(fileHandle, buffer, fileSize, &bytesRead, nullptr)) {

CloseHandle(fileHandle);

delete[] buffer;

return -1;

}

buffer[bytesRead] = '\0';

vector<int> dataValues;

istringstream inputData(buffer);

int val;

while (inputData >> val) {

dataValues.push\_back(val);

}

double avg, stddev;

int minVal, maxVal;

computeStatistics(dataValues, avg, stddev, minVal, maxVal);

writeStatsToFile(avg, stddev, minVal, maxVal, L"sync\_results.txt");

delete[] buffer;

CloseHandle(fileHandle);

return fileSize;

}

int main() {

createTestData(L"async\_data.txt", 10000);

createTestData(L"sync\_data.txt", 10000);

cout << "Async Processing Results:\n";

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

int bufferMultiplier = static\_cast<int>(pow(2, i));

int bufferSize = BASE\_BUFFER\_SIZE \* bufferMultiplier;

if (bufferSize >= 100000) break;

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

performAsyncOperations(bufferSize);

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

chrono::duration<double> totalDuration = endTime - startTime;

cout << "Buffer: " << bufferSize << " | Time: " << fixed << setprecision(3) << totalDuration.count() << "s\n";

}

cout << "\nSync Processing Results:\n";

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

DWORD syncFileSize = performSyncOperations();

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

chrono::duration<double> syncDuration = syncEnd - syncStart;

cout << "File Size: " << syncFileSize << " | Time: " << syncDuration.count() << "s\n";

return 0;

}