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 Царук В.А.

Преподаватель Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc177388377)

[2 Краткие теоритические сведения 4](#_Toc177388378)

[3 Описание функций программы 5](#_Toc177388379)

[3.1 Используемые *ISO* 5](#_Toc177388380)

3.2 Создание потоков 5

3.3 Функция *Writer* 5

3.4 Функция *Reader* 6

Список использованных источников 9

Приложение А (обязательное) Исходный код программы 10

# **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 – это ресурс системы, который не может одновременно использоваться более чем ограниченным числом пользователей. Чаще всего это означает, что доступ к ресурсу может получить только один пользователь в любой момент времени.

Критическая секция – это часть кода программы, которая требует монопольного доступа к общим ресурсам, которые не могут быть использованы более чем одним потоком одновременно. Если в критической секции одновременно находятся более одного процесса, возникает состояние, называемое «гонкой» [1].

Межпроцессное взаимодействие (IPC) – это обмен данными между потоками, как внутри одного процесса, так и между процессами на разных компьютерах в сети. Оно реализуется с помощью механизмов, предоставляемых операционной системой, или с помощью механизмов, реализующих новые возможности IPC [2].

Основная идея IPC заключается в проверке и модификации некоторого флага (признака) перед доступом к критическому ресурсу. Значение этого флага указывает на состояние ресурса: свободен он или занят. Основная проблема заключается в том, что регулятор доступа к критическому ресурсу сам является критическим ресурсом, и обращения к нему также становятся критической секцией, что вызывает рекурсивное замыкание требований.

Семафор – это глобальная переменная-счетчик (S), принимающая неотрицательные целочисленные значения. Для доступа к семафору используются атомарные операции P(S) и V(S):

1. P(S) – условный декремент: если значение семафора равно 0, поток блокируется до тех пор, пока значение не станет положительным.
2. V(S) – безусловный инкремент счетчика.

Мьютекс можно рассматривать как упрощенную версию двузначного семафора, где состояние интерпретируется как «свободно» или «занято», а операции доступа – как «захват» и «освобождение». Попытка повторного захвата мьютекса тем же потоком блокирует его до тех пор, пока другой поток не освободит мьютекс.

Барьер (barrier) – это объект синхронизации, который координирует выполнение процессов (потоков). Каждый поток запрашивает барьер, и, если количество таких запросов не достигло заранее заданного значения, потоки находятся в состоянии ожидания. Как только количество запросов достигает требуемого, все потоки продолжают выполнение, и барьер «освобождается».

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

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

## **3.1 Используемые ISO**

В качестве ресурсов синхронизации используются следующие объекты:

1 *resourceMutex —* мьютекс для обеспечения взаимного исключения между писателями и читателями, когда они обращаются к общему ресурсу;

2 *readerCountMutex* — мьютекс для защиты счётчика активных читателей, предотвращающий состояние гонки при его изменении;

3*readerCount* — переменная, которая отслеживает текущее количество активных потоков-читателей. Она используется для блокировки писателей, если есть хотя бы один активный читатель;

Мьютексы (*resourceMutex*, *readerCountMutex*) применяются для атомарного изменения состояния критических данных. Например, изменение счётчика читателей (r*eaderCount*) должно выполняться безопасно, чтобы избежать нарушения логики синхронизации.

## **3.2 Создание потоков**

Идет формирование последовательности потоков для выполнения:

Для каждого элемента массива *ids* создается поток с определенной функцией.

Если элемент является писателем, то создается поток с функцией *Writer,* уникальный идентификатор писателя передаётся в поток через массив *ids*.

Если элемент является читателем, то создается поток с функцией *Reader,* аналогично передаётся идентификатор читателя.

## **3.3 Функция *Writer***

В начале работы потока выводится сообщение о том, что писатель ожидает доступа к файлу. Это уведомление сообщает, что поток пытается получить доступ к ресурсу (файлу), чтобы начать запись. Идет измерение времени ожидания:

1 Сохраняется время начала ожидания;

2 Поток блокируется на мьютексе (для безопасного изменения общего ресурса);

3 После получения доступа вычисляется время ожидания которое добавляется к общей статистике;

Далее писатель открывает файл и записывает свой идентификатор (*id*) и текущее время в файл с помощью *GetTickCount64().* После всех действий писатель освобождает мьютекс, позволяя другим потокам использовать общий ресурс.

Результат работы писателя (рисунок 3.1).

Изображение выглядит как текст, Шрифт, снимок экрана, Графика

Автоматически созданное описание

Рисунок 3.1 – Результаты работы писателя

Симулируется процесс записи с задержкой в 300 мс (*simulate\_delay(100, 200)*). Если файл не удалось открыть, то выводится сообщение об ошибке.

**3.4 Функция Reader**

Функция *Reader* представляет поток читателя в многопоточном приложении. Основная задача читателя ‒ безопасно читать данные из одного-двух файлов, используя синхронизацию для координации доступа с писателями и другими читателями.

Если текущий поток ‒ первый читатель, он блокирует семафор писателей, чтобы никакой писатель не начал запись, пока он читает. Вычисляется время, которое читатель ожидал доступа к ресурсу. После ожидания мьютекс освобождается, так как все изменения, связанные с *readCount*, завершены.

Далее идет открытие файла для чтения. Все строки читаются из файла и собираются в одну строку. Если файл открыт: симулируется задержка для имитации чтения. После чтения добавляется 500 мс, чтобы симулировать завершение работы. Снова захватывается мьютекс, чтобы уменьшить счётчик читателей.

Результаты работы читателя (рисунок 3.2).

Изображение выглядит как текст, Шрифт, снимок экрана, черный

Автоматически созданное описание

Рисунок 3.2 – Результаты работы читателя

Чтение блокируется писателями: если есть активные читатели, писатели не могут записывать данные. Это достигается с помощью механизма блокировки первого читателя и освобождения последнего. Несколько читателей могут читать файл одновременно, пока писатели ждут. Время ожидания каждого читателя учитывается, чтобы анализировать производительность.

Ожидание доступа происходит для получения доступа к мьютексу, чтобы безопасно изменять счётчик читателей (*readCount*). Также происходит контроль общего числа одновременно читающих потоков и увеличивается счётчик активных читателей для данного файла.

ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы было разработано приложение, демонстрирующее параллельное взаимодействие процессов (потоков) с обеспечением их согласованной работы. Реализованы функции для потоков, которые выполняют роли писателей или читателей. Для обеспечения параллельной согласованной работы потоков использованы объекты синхронизации: мьютекс, семафоры для читателей и писателей. Мьютекс используется для обеспечения потокобезопасного изменения счетчика читателей, что исключает возможность «грязного» чтения или записи, а также доступ к ресурсу, который в данный момент не подготовлен для использования.

В результате работы программы обеспечена корректная обработка файлов потоками, а объекты синхронизации гарантируют их правильную работу. Однако, несмотря на это, присутствует время простоя потоков, связанное с необходимостью ожидания доступа к ресурсу. Потоки читатели могли ожидать в среднем 2063 мс. Суммарное ожидание составляет 20630 мс. Потоки писатели могли ожидать в среднем 516 мс. Суммарное ожидание составляет 1032 мс.

Типичная проблема заключается в том, что «читатели» могут захватывать доступ ко всем ресурсам, так как их количество обычно больше, а интенсивность чтения выше. Для решения этой проблемы использовались семафоры и мьютексы. Это позволяет сохранять текущие активные читатели, но блокирует появление новых. Запросы на запись имеют приоритет перед запросами на чтение, но не мешают самим процессам чтения.

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 <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_READERS 10

#define MAX\_WRITERS 5

#define RESOURCE\_USE\_TIME 2000 // Время использования ресурса (мс)

#define DELAY\_TIME 1000 // Задержка между операциями (мс)

const char\* FILENAME = "shared\_resource.txt"; // Имя файла для чтения/записи

// Глобальные переменные

HANDLE resourceMutex; // Для контроля доступа писателей

HANDLE readerCountMutex; // Для синхронизации счётчика читателей

int readerCount = 0; // Текущее количество активных читателей

double totalReaderWaitTime = 0.0;

double totalWriterWaitTime = 0.0;

int readerOps = 0;

int writerOps = 0;

// Логирование событий

CRITICAL\_SECTION logLock;

void log\_message(const char\* role, int id, const char\* action) {

EnterCriticalSection(&logLock);

printf("[%s %d] %s\n", role, id, action);

LeaveCriticalSection(&logLock);

}

// Симуляция задержки

void simulate\_delay(int minTime, int maxTime) {

Sleep((rand() % (maxTime - minTime + 1)) + minTime);

}

// Чтение из файла

void read\_file(int readerId) {

FILE\* file = nullptr;

// Открываем файл для чтения

if (fopen\_s(&file, FILENAME, "r") == 0 && file != nullptr) {

char buffer[256];

size\_t totalChars = 0;

// Читаем содержимое файла

while (fgets(buffer, sizeof(buffer), file)) {

totalChars += strlen(buffer);

simulate\_delay(100, 200); // Симуляция задержки при чтении

}

fclose(file);

log\_message("Reader", readerId, "read from file");

}

else {

log\_message("Reader", readerId, "failed to open file for reading");

printf("[Reader %d] Error opening file for reading\n", readerId);

}

}

// Запись в файл

void write\_file(int writerId) {

FILE\* file = nullptr;

// Открываем файл для записи (добавление в конец)

if (fopen\_s(&file, FILENAME, "a") == 0 && file != nullptr) {

// Генерация данных для записи

char buffer[128];

snprintf(buffer, sizeof(buffer), "[Writer %d] wrote data at %llu\n", writerId, GetTickCount64());

// Пишем данные в файл

if (fputs(buffer, file) != EOF) {

log\_message("Writer", writerId, "wrote to file");

printf("[Writer %d] Wrote data to file\n", writerId);

}

else {

log\_message("Writer", writerId, "failed to write to file");

printf("[Writer %d] Error writing to file\n", writerId);

}

fclose(file);

}

else {

log\_message("Writer", writerId, "failed to open file for writing");

printf("[Writer %d] Error opening file for writing\n", writerId);

}

// Симуляция времени работы с ресурсом

simulate\_delay(100, 200);

}

// Читатель

DWORD WINAPI reader(LPVOID param) {

int id = \*(int\*)param;

while (1) {

simulate\_delay(500, DELAY\_TIME);

DWORD startWait = GetTickCount64();

// Запрашиваем доступ к счётчику читателей

WaitForSingleObject(readerCountMutex, INFINITE);

if (readerCount == 0) {

WaitForSingleObject(resourceMutex, INFINITE); // Если первый читатель, блокируем писателей

}

readerCount++;

ReleaseMutex(readerCountMutex);

DWORD endWait = GetTickCount64();

totalReaderWaitTime += (endWait - startWait);

readerOps++;

// Чтение

log\_message("Reader", id, "started reading");

read\_file(id);

simulate\_delay(500, RESOURCE\_USE\_TIME);

log\_message("Reader", id, "finished reading");

// Освобождаем ресурс

WaitForSingleObject(readerCountMutex, INFINITE);

readerCount--;

if (readerCount == 0) {

ReleaseMutex(resourceMutex); // Если последний читатель, освобождаем писателей

}

ReleaseMutex(readerCountMutex);

}

return 0;

}

// Писатель

DWORD WINAPI writer(LPVOID param) {

int id = \*(int\*)param;

while (1) {

simulate\_delay(500, DELAY\_TIME);

DWORD startWait = GetTickCount64();

// Запрашиваем доступ к ресурсу

WaitForSingleObject(resourceMutex, INFINITE);

DWORD endWait = GetTickCount64();

totalWriterWaitTime += (endWait - startWait);

writerOps++;

// Запись

log\_message("Writer", id, "started writing");

write\_file(id);

simulate\_delay(500, RESOURCE\_USE\_TIME);

log\_message("Writer", id, "finished writing");

// Освобождаем ресурс

ReleaseMutex(resourceMutex);

}

return 0;

}

int main() {

srand((unsigned int)time(NULL));

// Инициализация

InitializeCriticalSection(&logLock);

resourceMutex = CreateMutex(NULL, FALSE, NULL);

readerCountMutex = CreateMutex(NULL, FALSE, NULL);

HANDLE threads[MAX\_READERS + MAX\_WRITERS];

int ids[MAX\_READERS + MAX\_WRITERS];

// Создаём потоки читателей

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

ids[i] = i + 1;

threads[i] = CreateThread(NULL, 0, reader, &ids[i], 0, NULL);

}

// Создаём потоки писателей

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

ids[MAX\_READERS + i] = i + 1;

threads[MAX\_READERS + i] = CreateThread(NULL, 0, writer, &ids[MAX\_READERS + i], 0, NULL);

}

// Работаем заданное время, например, 30 секунд

Sleep(10000);

// Завершаем потоки (в данной модели потоки бесконечные, поэтому принудительно)

for (int i = 0; i < MAX\_READERS + MAX\_WRITERS; i++) {

TerminateThread(threads[i], 0);

CloseHandle(threads[i]);

}

// Вывод статистики

printf("\n--- Statistics ---\n");

printf("Total Reader Wait Time: %.2f ms\n", totalReaderWaitTime);

printf("Average Reader Wait Time: %.2f ms\n", readerOps > 0 ? totalReaderWaitTime / readerOps : 0.0);

printf("Total Writer Wait Time: %.2f ms\n", totalWriterWaitTime);

printf("Average Writer Wait Time: %.2f ms\n", writerOps > 0 ? totalWriterWaitTime / writerOps : 0.0);

// Освобождение ресурсов

DeleteCriticalSection(&logLock);

CloseHandle(resourceMutex);

CloseHandle(readerCountMutex);

return 0;

}