МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЕТ

по лабораторной работе № 4.1

по дисциплине «Операционные системы»

ТЕМА: «Межпроцессорное взаимодействие»

Студент гр. 3311 Баймухамедов Р. Р.

Преподаватель Тимофеев А. В.

Санкт-Петербург

2025

Цель работы

Исследовать инструменты и механизмы взаимодействия процессов в Windows

Задание

Реализация решения задачи о читателях-писателях

1. Выполнить решение задачи о читателях-писателях, для чего необходимо разработать консольные приложения “Читатель” и “Писатель”

    - Одновременно запущенные экземпляры процессоров-читателей и процессов-писателей должны совместно работать с буферной памятью в виде проецируемого файла
        - Размер страницы буферной памяти равен размеру физической страницы оперативной памятью
        - Число страниц буферной памяти равно сумме цифр в номере студенческого билета без учета первой цифры
    - Страницы буферной памяти должны быть заблокированы в оперативной памяти (функция VirtualLock)
    - Длительность выполнения процессами операций “чтения” и “записи” задается случайным образом в диапазоне от 0,5 до 1,5 секунд
    - Для синхронизации работы процессов необходимо использовать объекты синхронизации типа “семафор” или “мьютекс”
    - Процессы-читатели и процессы-писатели ведут свои журнальные файлы, в которые регистрируют переходы из одного “состояния” в другое (начало ожидания, запись или чтение, переход к освобождению) с указанием кода времени (функция TimeGetTime). Для состояний “запись” и “чтение” необходимо также запротоколировать номер рабочей страницы.
2. Запустите приложения читателей и писателей, суммарное количество одновременно работающих читателей и писателей должно быть не менее числа страниц буферной памяти. Проверьте функционирование приложений, проанализируйте журнальные файлы, процессов, постройте сводные графики смены “состояний” для не менее 5 процессов-читателей и 5 процессов-писателей, дайте свои комментарии относительно переходов процессов из одного состояния в другое
3. Постройте графики в Excel/Python и дайте свои комментарии:

    - Смена состояний процессов (пример: читатель 1 -> ожидание -> чтение -> освобождение
    - Занятость страниц по времени (визуализация через heatmap)
4. Подготовьте итоговый отчет с развернутыми выводами по заданию

Поведение процессов

Писатель (writer.exe):

Инициализирует разделяемую память, мьютекс и семафор.

Периодически захватывает мьютекс, записывает строку формата hello_from_<id> в случайную страницу и освобождает мьютекс.

В каждой итерации вызывает SleepEx(..., TRUE) для имитации работы и перехода в alertable state.

Читатель (reader.exe):

Подключается к существующим объектам памяти и синхронизации, созданным писателем.

При чтении:

Захватывает семафор h_readers_semaphore.

Выполняет InterlockedIncrement(readers_count).

Если значение стало 1, захватывает мьютекс h_writers_mutex.

Освобождает h_readers_semaphore.

Выбирает для чтения страницу, предварительно проверяя есть ли в ней запись писателя

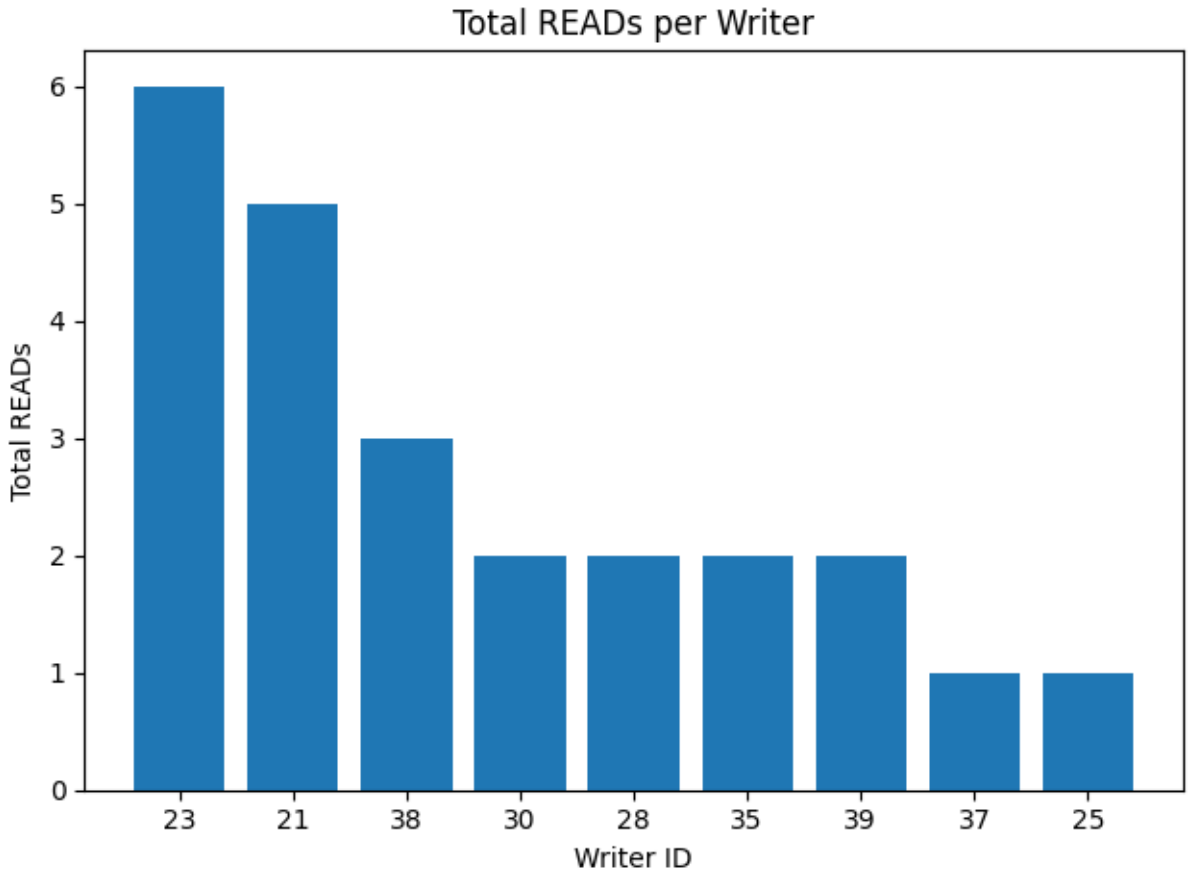
Переходит в alertable state через SleepEx(...).

Повторно захватывает семафор, выполняет InterlockedDecrement(readers_count).

Если значение стало 0, освобождает мьютекс.

Освобождает семафор.

График



Заключение

Был рассмотрен один из способов реализации межпроцессного взаимодействия (IPC) — через использование разделяемой памяти с проецированием. В рамках данной реализации писатели и читатели отображают в своё виртуальное адресное пространство область общей памяти, состоящую из восьми страниц по 4096 байт и дополнительных 4 байт, предназначенных для хранения счётчика активных читателей.

Код программы

writer.cpp

```
#include <windows.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <iostream>

#define SHM_NAME L"SharedBuffer"
#define MUTEX_NAME L"WritersMutex"
#define SEM_NAME L"ReadersSemaphore"
// 331103 = 3+1+1+3
#define PAGE_COUNT 8
#define PAGE_SIZE 4096
#define TOTAL_SIZE (4 + PAGE_COUNT * PAGE_SIZE)
#define LOG_FMT "logs_writers/writer_%02d.log"

#define COUNTER_OFFSET 0
#define BUFFER_OFFSET 4

using namespace std;

BYTE* page_base = nullptr;
FILE* log_file = nullptr;

HANDLE h_map = nullptr;
HANDLE h_mutex = nullptr;
HANDLE h_reader_semaphore = nullptr;

int writer_id = -1;

void endup() {
   if (log_file) fclose(log_file);
   if (page_base) UnmapViewOfFile(page_base - BUFFER_OFFSET);
   if (h_map) CloseHandle(h_map);
   if (h_mutex) CloseHandle(h_mutex);
   if (h_reader_semaphore) CloseHandle(h_reader_semaphore);
}

void sync_objects_init() {
   h_mutex = CreateMutexW(NULL, FALSE, MUTEX_NAME);
   if (!h_mutex) {
      cout << "CreateMutex failed";
      exit(1);
   }

   h_reader_semaphore = CreateSemaphoreW(NULL, 1, 1, SEM_NAME);

   if (!h_reader_semaphore) {
      cout << "CreateSemaphore failed";
      exit(1);
   }
}

void shared_memory_init() {
   h_map = CreateFileMappingW(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, TOTAL_SIZE, SHM_NAME);

   if (!h_map) {
      cout << "CreateFileMapping failed";
      exit(1);
   }

   BYTE* base = (BYTE*)MapViewOfFile(h_map, FILE_MAP_ALL_ACCESS, 0, 0, TOTAL_SIZE);
   if (!base) {
      cout << "MapViewOfFile failed";
      exit(1);
   }

   if (GetLastError() != ERROR_ALREADY_EXISTS) {
      memset(base, 0, TOTAL_SIZE);
   }
```

```
    page_base = base + BUFFER_OFFSET;
}

void init_log(int id) {
    char filename[64];
    snprintf(filename, sizeof(filename), LOG_FMT, id);

    log_file = fopen(filename, "w");
    if (!log_file) {
        cout << "Failed to init log file";
        exit(1);
    }
}

void log_event(const char* state, int page_index, DWORD timestamp) {
    if (!log_file) return;

    if (page_index >= 0)
        fprintf(log_file, "WRITER_%02d %s PAGE_%d %lu\n", writer_id, state, page_index, timestamp);
    else
        fprintf(log_file, "WRITER_%02d %s %lu\n", writer_id, state, timestamp);

    fflush(log_file);
}

void write_operation() {
    int page_index = -1;

    for (int attempt = 0; attempt < PAGE_COUNT * 2; ++attempt) {
        int candidate = rand() % PAGE_COUNT;
        BYTE* page = page_base + candidate * PAGE_SIZE;

        if (page[0] == 0) {
            page_index = candidate;
            break;
        }
    }

    if (page_index == -1) {
        SleepEx(10, FALSE);
        return;
    }

    DWORD t_wait = GetTickCount();
    log_event("WAIT_WRITE", -1, t_wait);

    WaitForSingleObject(h_mutex, INFINITE);

    VirtualLock(page_base + page_index * PAGE_SIZE, PAGE_SIZE);

    DWORD t_write = GetTickCount();
    log_event("WRITE", page_index, t_write);

    BYTE* target = page_base + page_index * PAGE_SIZE;
    target[0] = 1;

    char content[16] = {0};
    sprintf(content, "hello %02d", writer_id);
    memcpy(target + 1, content, strlen(content) + 1);

    VirtualUnlock(h_map, PAGE_SIZE * page_index);
    SleepEx(5 + rand() % 10, TRUE);

    DWORD t_release = GetTickCount();
    log_event("RELEASE", -1, t_release);

    ReleaseMutex(h_mutex);
}
```

```
int main(int argc, char* argv[]) {
    if (argc < 2) {
        cout << "count of args is not correct";
        return 1;
    }

    writer_id = atoi(argv[1]);
    srand((unsigned int)(time(NULL) + writer_id));

    init_log(writer_id);
    shared_memory_init();
    sync_objects_init();

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

    endup();
    printf("writer %d finished\n", writer_id);
    return 0;
}
```

reader.cpp

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mmsystem.h>
#include <iostream>

#define SHM_NAME L"SharedBuffer"
#define MUTEX_NAME L"WritersMutex"
#define SEM_NAME L"ReadersSemaphore"
// 331103 = 3+1+1+3
#define PAGE_COUNT 8
#define PAGE_SIZE 4096
#define LOG_NAME_FMT "logs_readers/reader_%02d.log"

#define READER_COUNTER_OFFSET 0
#define BUFFER_OFFSET 4

using namespace std;

volatile LONG* readers_count = nullptr;
BYTE* page_base = nullptr;

HANDLE h_writers_mutex = nullptr;
HANDLE h_readers_semaphore = nullptr;
HANDLE h_map = nullptr;
FILE* log_file = nullptr;

int reader_id = -1;

void endup() {
    if (log_file) fclose(log_file);
    if (page_base) UnmapViewOfFile(page_base - BUFFER_OFFSET);
    if (h_map) CloseHandle(h_map);
    if (h_writers_mutex) CloseHandle(h_writers_mutex);
    if (h_readers_semaphore) CloseHandle(h_readers_semaphore);
}

void shared_objects_init() {
    h_map = OpenFileMappingW(FILE_MAP_ALL_ACCESS, FALSE, SHM_NAME);
    if (!h_map) {
        cout << "OpenFileMapping failed";
        exit(1);
```

```
    }

    BYTE* base = (BYTE*)MapViewOfFile(h_map, FILE_MAP_ALL_ACCESS, 0, 0, 0);
    if (!base) {
        cout << "MapViewOfFile failed";
        exit(1);
    }

    readers_count = (volatile LONG*)(base + READER_COUNTER_OFFSET);
    page_base = base + BUFFER_OFFSET;

    h_writers_mutex = OpenMutexW(SYNCHRONIZE, FALSE, MUTEX_NAME);
    if (!h_writers_mutex) {
        cout << "OpenMutex failed";
        exit(1);
    }

    h_readers_semaphore = OpenSemaphoreW(SYNCHRONIZE | SEMAPHORE_MODIFY_STATE, FALSE, SEM_NAME);
    if (!h_readers_semaphore) {
        cout << "OpenSemaphore failed";
        exit(1);
    }
}

void init_log(int id) {
    char filename[64];
    snprintf(filename, sizeof(filename), LOG_NAME_FMT, id);

    log_file = fopen(filename, "w");
    if (!log_file) {
        cout << "Failed to init log file";
        exit(1);
    }
}

void log_event(const char* state, int page_index, DWORD timestamp) {
    if (!log_file) return;

    if (page_index >= 0)
        fprintf(log_file, "READER_%02d %s PAGE_%d %lu\n", reader_id, state, page_index, timestamp);
    else
        fprintf(log_file, "READER_%02d %s %lu\n", reader_id, state, timestamp);

    fflush(log_file);
}

void read_operation() {
    int page_index = -1;
    char buffer[16] = {};

    for (int attempt = 0; attempt < PAGE_COUNT * 2; ++attempt) {
        int candidate = rand() % PAGE_COUNT;
        BYTE* page = page_base + candidate * PAGE_SIZE;
        memcpy(buffer, page, 9);

        if (buffer[0] == 1) {
            page_index = candidate;
            break;
        }
    }

    if (page_index == -1) {
        SleepEx(10, FALSE);
        return;
    }

    DWORD time_start = timeGetTime();
    log_event("WAIT_READ", -1, time_start);
```

```
    WaitForSingleObject(h_readers_semaphore, INFINITE);

    if (InterlockedIncrement(readers_count) == 1)
        WaitForSingleObject(h_writers_mutex, INFINITE);

    ReleaseSemaphore(h_readers_semaphore, 1, NULL);

    DWORD t_read = timeGetTime();
    log_event("READ", page_index, t_read);

    BYTE* page = page_base + page_index * PAGE_SIZE;
    memset(buffer, 0, sizeof(buffer));
    memcpy(buffer, page, sizeof(buffer) - 1);

    fprintf(log_file, "READER_%02d READ_CONTENT PAGE_%d \"%s\"\n", reader_id, page_index, buffer);
    fflush(log_file);

    SleepEx(5 + rand() % 10, TRUE);

    DWORD t_release = timeGetTime();
    log_event("RELEASE", -1, t_release);

    WaitForSingleObject(h_readers_semaphore, INFINITE);

    if (InterlockedDecrement(readers_count) == 0)
        ReleaseMutex(h_writers_mutex);

    ReleaseSemaphore(h_readers_semaphore, 1, NULL);
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        cout << "count of args is not correct";
        return 1;
    }

    reader_id = atoi(argv[1]);
    srand((unsigned int)(time(NULL) + reader_id));

    init_log(reader_id);
    shared_objects_init();

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

    endup();
    printf("reader %d finished\n", reader_id);
    return 0;
}
```