**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.

1. 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
2. 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)

1. 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

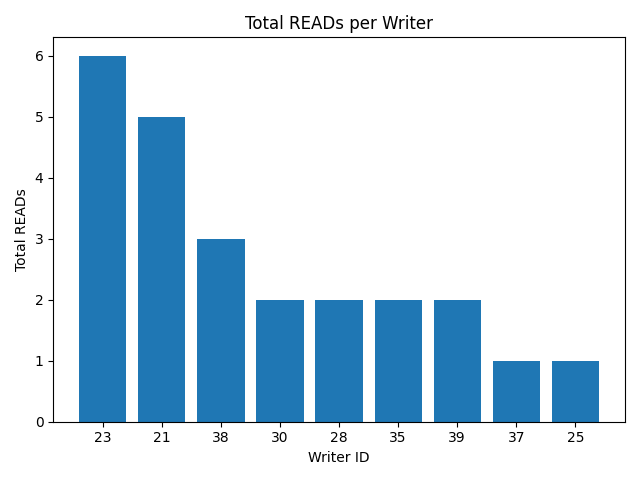
Переходит в alertable state через SleepEx(...).

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;

}