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

ОТЧЕТ

к лабораторной работе №6

на тему

Средства синхронизации и взаимного исключения (Windows). Изучение и использование средств синхронизации и взаимного исключения.

Студент И. А. Тиханёнок

Преподаватель Н. Ю. Гриценко

Минск 2023

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146752068)

[2 Теоретические сведения 4](#_Toc146752069)

[3 Результат выполнения 5](#_Toc146752070)

[Заключение 5](#_Toc146752071)

[Список использованных источников 7](#_Toc146752072)

[Приложение А (обязательное) Листинг кода 8](#_Toc146752073)

1. **ЦЕЛЬ РАБОТЫ**

Цель работы изучить и применить на практике знания об средствах синхронизации и взаимного исключения *Windows*, научиться использовать средства синхронизации и взаимного исключения. Для достижения цели будет создано приложение в виде компьютерной игры змейка.

1. **ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Средства синхронизации и взаимного исключения в операционной системе *Windows* играют ключевую роль в обеспечении правильного и безопасного взаимодействия множества потоков в многозадачных приложениях. Они помогают предотвратить гонки данных и другие проблемы, возникающие при одновременном доступе к общим ресурсам. Вот некоторые важные теоретические сведения на эту тему:

Многозадачные приложения, работающие с несколькими потоками, требуют синхронизации для контроля над порядком выполнения потоков. Синхронизация позволяет управлять доступом к разделяемым ресурсам, таким как переменные, структуры данных и файлы.

Критическая секция - это участок кода, который должен выполняться только одним потоком в определенный момент времени. В *Windows*, для создания критических секций используется структура данных *CRITICAL\_SECTION*. Она обеспечивает монопольный доступ к коду внутри секции.

Мьютексы - это объекты, которые используются для координации между потоками. Они могут быть использованы для взаимного исключения, что означает, что только один поток может войти в критическую секцию, защищенную мьютексом, в любой момент времени. В Windows, мьютексы создаются с помощью функции `*CreateMutex*`.

Семафоры - это средство синхронизации, которое позволяет контролировать доступ к определенному количеству ресурсов. Они полезны, когда есть несколько экземпляров общего ресурса. В *Windows*, семафоры могут быть созданы с помощью функции `*CreateSemaphore*`.

События позволяют потокам ожидать события и продолжать выполнение, когда событие произойдет. Они могут использоваться для оповещения о завершении задачи или для синхронизации между несколькими потоками. В *Windows*, события могут быть созданы с помощью функции `*CreateEvent*`.

В некоторых случаях, когда нужно выполнить операцию над разделяемыми данными, можно использовать атомарные операции. Они гарантируют, что операция будет выполнена без переключения контекста и вмешательства других потоков.

Концепция взаимного исключения гарантирует, что только один поток может выполнять критическую секцию кода в заданный момент времени. Это предотвращает состояние гонки данных и обеспечивает целостность общих ресурсов.

Изучение и правильное использование средств синхронизации и взаимного исключения важны для разработчиков многозадачных приложений в *Windows*. Они обеспечивают надежность и стабильность работы приложений, предотвращая возможные конфликты и проблемы, связанные с параллельным выполнением кода разными потоками.

1. **РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ**

В результате выполнения лабораторной работы была дополнена игра змейка при помощи *mutex* для предотвращения гонки данных, были использованы условные переменные в сочетании с ним для реализации ожидания определенных условий, были использованы критические секции для синхронизации доступа к разделяемой памяти и обеспечения безопасности многопоточного доступа к данным, хранящимся в этой области памяти; также были применены функции для создания и управлениями потоками в игре, в частности, ввод пользователя и обновления игры и перерисовка (Рисунок 1).

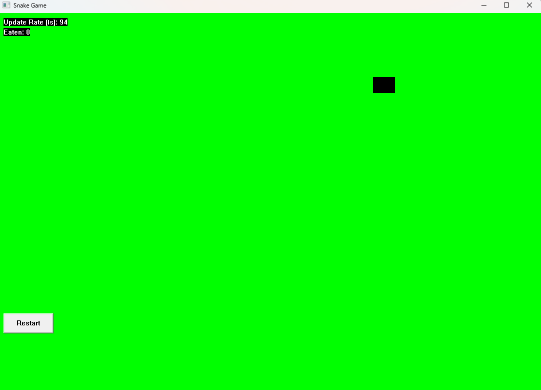


Рисунок 1 – Рабочее окно приложения

## ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы были изучены и применены важные аспекты управления процессами и потоками в операционной системе *Windows* с использованием средств синхронизации и взаимного исключения. Работа с многозадачностью и параллельными процессами требует точной синхронизации и координации для предотвращения гонок данных и других проблем, связанных с одновременным доступом к общим ресурсам.

Был изучен принцип работы мьютексов и критических секций. Они позволяют обеспечить взаимное исключение, гарантируя, что только один поток может одновременно выполнять критический участок кода. Это средство предотвращает конфликты и гарантирует целостность общих данных.

Было рассмотрено применение семафоров для контроля доступа к определенному количеству ресурсов. Это полезно, когда существует ограниченное количество экземпляров общего ресурса, и нужно управлять их распределением между потоками.

Исследовались события как средство для оповещения потоков о важных событиях. События позволяют потокам ожидать определенных условий и продолжать выполнение после их наступления.

Рассмотрены атомарные операции, которые обеспечивают выполнение операции как одно целое, без прерывания другими потоками. Это полезно при работе с общими данными, когда требуется атомарность операций.

Лабораторная работа также предоставила практический опыт в использовании указанных средств синхронизации. На примере создания игрового приложения было продемонстрировано, как обеспечить корректное взаимодействие между потоками, гарантируя, что общие данные остаются валидными и целостными.

Полученные знания о средствах синхронизации и взаимного исключения позволят разрабатывать более надежные и эффективные многозадачные приложения под операционной системой *Windows*. Эти инструменты и концепции синхронизации имеют важное значение для обеспечения стабильной работы и предоставления позитивного опыта пользователя.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Щупак Ю. *Win32 API*. Разработка приложений для *Windows*. ─ СПБ: Питер, 2008. ─ 592 с.: ип.
2. Создание классических приложений для *Windows* с использованием *API Win32* [Электронный ресурс]. ─ Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – Файл Lab6.cpp

#include <windows.h>

#include <deque>

#include <fstream>

#include <thread>

#include <mutex>

#include <string>

#include <chrono>

#include <fstream>

#include "resource.h"

using namespace std;

const int gridSize = 25;

const int screenWidth = 1100;

const int screenHeight = 800;

const wchar\_t\* saveFileName = L"D:\\Study\\OSiSP\\LR1\\LR1\\snake\_save.bin";

const wchar\_t\* sharedMemoryName = L"MySharedMemory";

std::thread inputThread; // Новый поток для обработки пользовательского ввода

std::thread updateGameThread; // Новый поток для функции обновления игры

HWND hWnd;

HWND restartButton;

HHOOK g\_hook = NULL;

struct SnakeSegment {

int x, y;

SnakeSegment() : x(0), y(0) {}

SnakeSegment(int \_x, int \_y) : x(\_x), y(\_y) {}

};

LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam);

deque<SnakeSegment> snake;

int foodX, foodY;

bool gameOver = false;

int direction = 1;

int foodEaten = 0;

mutex snakeMutex;

HANDLE hMapFile;

LPCTSTR pBuf;

CONDITION\_VARIABLE conditionVariable;

CRITICAL\_SECTION cs;

bool condition = false;

// Функция для установки приоритета процесса

void SetProcessPriority() {

HANDLE hProcess = GetCurrentProcess();

BOOL success = SetPriorityClass(hProcess, HIGH\_PRIORITY\_CLASS);

if (success) {

MessageBox(hWnd, L"Process priority set to HIGH\_PRIORITY\_CLASS.", L"Priority Set", MB\_OK | MB\_ICONINFORMATION);

}

else {

MessageBox(hWnd, L"Failed to set process priority.", L"Priority Set Failed", MB\_OK | MB\_ICONERROR);

}

}

// Создаём разделяемую память, которая будет использоваться для обмена данными между процессами (для хранения координат головы змейки)

void CreateMemoryMapping() {

// создаем область разделяемой памяти

hMapFile = CreateFileMapping(

INVALID\_HANDLE\_VALUE, // дескриптор файла

NULL, // атрибут для области разделяемой памяти

PAGE\_READWRITE, // определение защиты - чтение + запись

0, // - старшее слово максимального размера файла-карты, у нас без привязки -> 0

sizeof(SnakeSegment), // младшее слово максимального размера файла-карты

sharedMemoryName); // создание идентификатора области разделяемой памяти - для доступа к данной области разделяемой памяти

if (hMapFile == NULL) {

MessageBox(hWnd, L"Failed to create file-mapping object", L"Error", MB\_ICONERROR | MB\_OK);

return;

}

// создаем представление разделяемой памяти (через которое будем получать доступ к данным в области разделяемой памяти)

pBuf = (LPCTSTR)MapViewOfFile(hMapFile, FILE\_MAP\_ALL\_ACCESS, 0, 0, sizeof(SnakeSegment));

// 0 + 0 - без ограничений - sizeof.. - размер части области разделяемой памяти

if (pBuf == NULL) {

MessageBox(hWnd, L"Failed to create file view", L"Error", MB\_ICONERROR | MB\_OK);

CloseHandle(hMapFile);

return;

}

}

// Закрываем и освобождаем связанные с разделяемой памятью ресурсы

void CloseMemoryMapping() {

UnmapViewOfFile(pBuf);

CloseHandle(hMapFile);

}

// Сохраняем текущее состояние игры - положение змейки в бинарный файл

void SaveGame() {

lock\_guard<mutex> lock(snakeMutex);

ofstream file(saveFileName, ios::binary);

if (file.is\_open()) {

int size = static\_cast<int>(snake.size());

file.write(reinterpret\_cast<char\*>(&size), sizeof(int));

for (const SnakeSegment& segment : snake) {

file.write(reinterpret\_cast<const char\*>(&segment), sizeof(SnakeSegment));

}

file.close();

}

SnakeSegment segment;

if (!snake.empty()) {

segment = snake.front();

}

memcpy((LPVOID)pBuf, &segment, sizeof(SnakeSegment));

}

// Загружаем сохраненное состояние игры из файла

void LoadGame() {

ifstream file(saveFileName, ios::binary);

if (file.is\_open()) {

int size;

file.read(reinterpret\_cast<char\*>(&size), sizeof(int));

deque<SnakeSegment> newSnake;

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

SnakeSegment segment;

file.read(reinterpret\_cast<char\*>(&segment), sizeof(SnakeSegment));

newSnake.push\_back(segment);

}

file.close();

SnakeSegment segment;

memcpy(&segment, pBuf, sizeof(SnakeSegment));

{

lock\_guard<mutex> lock(snakeMutex);

snake = newSnake;

snake.push\_front(segment);

}

}

}

// Рисуем одну ячейку игрового поля - 1 - контекст устройства, координаты x + y - цвет

void DrawCell(HDC hdc, int x, int y, COLORREF color) {

int cellWidth = screenWidth / gridSize;

int cellHeight = screenHeight / gridSize;

HBRUSH brush = CreateSolidBrush(color);

HPEN pen = CreatePen(PS\_SOLID, 1, color);

HBRUSH oldBrush = (HBRUSH)SelectObject(hdc, brush);

HPEN oldPen = (HPEN)SelectObject(hdc, pen);

Rectangle(hdc, x \* cellWidth, y \* cellHeight, (x + 1) \* cellWidth, (y + 1) \* cellHeight);

SelectObject(hdc, oldBrush);

SelectObject(hdc, oldPen);

DeleteObject(brush);

DeleteObject(pen);

}

// Принимаем код клавиши - обрабатываем пользовательский ввод (управление и действия: сохранить и загрузить)

void HandleInput(WPARAM wParam) {

switch (wParam) {

case VK\_LEFT:

if (direction != 2) direction = 0;

break;

case VK\_UP:

if (direction != 3) direction = 1;

break;

case VK\_RIGHT:

if (direction != 0) direction = 2;

break;

case VK\_DOWN:

if (direction != 1) direction = 3;

break;

case 'S':

SaveGame();

break;

case 'L':

LoadGame();

break;

}

}

// Обновление состояния игры (движение змейки + проверка столкновений) - принимаем дескриптор окна

void UpdateGame(HWND hWnd) {

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

if (snake.empty()) {

return;

}

int headX = snake.front().x;

int headY = snake.front().y;

switch (direction) {

case 0:

headX--;

break;

case 1:

headY--;

break;

case 2:

headX++;

break;

case 3:

headY++;

break;

}

if (headX < 0 || headX >= screenWidth / gridSize || headY < 0 || headY >= screenHeight / gridSize) {

gameOver = true;

return;

}

lock\_guard<mutex> lock(snakeMutex);

for (const SnakeSegment& segment : snake) {

if (segment.x == headX and segment.y == headY) {

gameOver = true;

return;

}

}

if (headX == foodX and headY == foodY) {

foodX = rand() % (screenWidth / gridSize);

foodY = rand() % (screenHeight / gridSize);

foodEaten++;

if (foodEaten % 5 == 0) {

MessageBox(hWnd, L"You have eaten 5 apples!", L"Notification", MB\_OK | MB\_ICONINFORMATION);

}

}

else {

snake.pop\_back();

}

snake.push\_front(SnakeSegment(headX, headY));

InvalidateRect(hWnd, NULL, TRUE);

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

auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

wstring updateRateText = L"Update Rate (μs): " + to\_wstring(duration.count());

TextOut(GetDC(hWnd), 10, 10, updateRateText.c\_str(), static\_cast<int>(updateRateText.length()));

}

// Функция для обновления игры в отдельном потоке - периодически обновляет состояние игры

DWORD WINAPI GameUpdateThreadWrapper(LPVOID lpParam) {

HWND hWnd = static\_cast<HWND>(lpParam);

while (true) {

if (!gameOver) {

UpdateGame(hWnd);

InvalidateRect(hWnd, NULL, TRUE);

}

Sleep(100);

}

return 0;

}

// Используется для перезапуска игры + начальной инициализации игровых параметров

void RestartGame() {

lock\_guard<mutex> lock(snakeMutex);

snake.clear();

snake.push\_back(SnakeSegment(5, 5));

foodX = rand() % (screenWidth / gridSize);

foodY = rand() % (screenHeight / gridSize);

direction = 1;

gameOver = false;

foodEaten = 0;

InvalidateRect(hWnd, NULL, TRUE);

}

// Новая функция для обработки пользовательского ввода в отдельном потоке - сейчас ожидает клавишу "R", но вы можете добавить другие клавиши управления

DWORD WINAPI InputThreadWrapper(LPVOID lpParam) {

while (true) {

if (GetAsyncKeyState('R') & 0x8000) {

RestartGame();

}

Sleep(100);

}

return 0;

}

// Отрисовка всего игрового поля и игровых элементов (змейка + еда)

void PaintGame(HDC hdc) {

RECT clientRect;

GetClientRect(hWnd, &clientRect);

HBRUSH greenBrush = CreateSolidBrush(RGB(0, 255, 0));

FillRect(hdc, &clientRect, greenBrush);

DeleteObject(greenBrush);

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

wstring foodEatenText = L"Eaten: " + to\_wstring(foodEaten);

COLORREF backgroundColor = RGB(0, 0, 0);

COLORREF textColor = RGB(255, 255, 255);

SetBkColor(hdc, backgroundColor);

SetTextColor(hdc, textColor);

TextOut(hdc, 10, 30, foodEatenText.c\_str(), static\_cast<int>(foodEatenText.length()));

DrawCell(hdc, foodX, foodY, RGB(255, 0, 0)); // для яблока

lock\_guard<mutex> lock(snakeMutex);

for (const SnakeSegment& segment : snake) {

DrawCell(hdc, segment.x, segment.y, RGB(0, 0, 0));

}

if (gameOver) {

TextOut(hdc, 10, 30, L"Game Over", 9);

}

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

auto duration = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

wstring updateRateText = L"Update Rate (μs): " + to\_wstring(duration.count());

TextOut(hdc, 10, 10, updateRateText.c\_str(), static\_cast<int>(updateRateText.length()));

}

// Дескриптор окна (какое окно должно быть обработано) + код сообщения (определяет, какая часть кода будет выполнена в ответ на сообщение)

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {

switch (message) {

case WM\_PAINT:

PAINTSTRUCT ps;

HDC hdc;

hdc = BeginPaint(hWnd, &ps);

PaintGame(hdc);

EndPaint(hWnd, &ps);

break;

case WM\_COMMAND:

if ((HWND)lParam == restartButton) {

RestartGame();

}

break;

case WM\_CLOSE:

DestroyWindow(hWnd);

break;

case WM\_DESTROY:

PostQuitMessage(0);

break;

case WM\_KEYDOWN:

HandleInput(wParam);

break;

default:

return DefWindowProc(hWnd, message, wParam, lParam);

break;

}

return 0;

}

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {

WNDCLASSEX wcex;

ZeroMemory(&wcex, sizeof(wcex));

wcex.cbSize = sizeof(wcex);

wcex.style = CS\_HREDRAW | CS\_VREDRAW;

wcex.lpfnWndProc = WndProc;

wcex.hInstance = hInstance;

wcex.hCursor = LoadCursor(nullptr, IDC\_ARROW);

wcex.lpszClassName = L"SnakeGame";

RegisterClassEx(&wcex);

hWnd = CreateWindow(L"SnakeGame", L"Snake Game", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, 0, screenWidth, screenHeight, nullptr, nullptr, hInstance, nullptr);

if (!hWnd) {

MessageBox(nullptr, L"CreateWindow failed", L"Error", MB\_ICONERROR | MB\_OK);

return 1;

}

restartButton = CreateWindow(L"BUTTON", L"Restart", WS\_CHILD | WS\_VISIBLE | BS\_PUSHBUTTON, 10, 600, 100, 40, hWnd, (HMENU)1, hInstance, 0);

if (!restartButton) {

MessageBox(nullptr, L"Create restart button failed", L"Error", MB\_ICONERROR | MB\_OK);

return 2;

}

SetProcessPriority();

// Создаем секцию синхронизации

InitializeCriticalSection(&cs);

// Создаем условную переменную

InitializeConditionVariable(&conditionVariable);

CreateMemoryMapping();

ShowWindow(hWnd, nCmdShow);

UpdateWindow(hWnd);

RestartGame();

inputThread = std::thread(InputThreadWrapper, hWnd);

inputThread.detach();

updateGameThread = std::thread(GameUpdateThreadWrapper, hWnd);

updateGameThread.detach();

MSG msg;

while (GetMessage(&msg, nullptr, 0, 0)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

DeleteCriticalSection(&cs);

CloseMemoryMapping();

return (int)msg.wParam;

}