

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 №1  
на тему  
**«УПРАВЛЕНИЕ ПРОЦЕССАМИ, ПОТОКАМИ, НИТЯМИ»**  
БГУИР 6-05-0612-02 23

Выполнил студент группы 353503  
СЕБЕЛЕВ Дмитрий Юрьевич

---

(дата, подпись студента)

Проверил ассистент каф. информатики  
ГРИЦЕНКО Никита Юрьевич

---

(дата, подпись преподавателя)

Минск 2025

# **СОДЕРЖАНИЕ**

1 Постановка задачи .....	3
2 Описание работы программы .....	4
2.1 Инициализация и загрузка состояния .....	4
2.2 Регистрация класса окна и создание интерфейса .....	4
2.3 Обработка сообщений и отображение состояния .....	4
2.4 Сохранение состояния приложения .....	4
2.5 Создание нового экземпляра процесса .....	5
3 Ход выполнения программы .....	6
3.1 Пример выполнения задания .....	6
Вывод .....	7
Список использованных источников .....	8
Приложение А (справочное) Исходный код программы .....	9

## **1 ПОСТАНОВКА ЗАДАЧИ**

Цель данной лабораторной работы заключается в возобновлении, закреплении и развитии навыков программирования приложений Windows через изучение концепций вычислительных процессов, потоков и нитей, освоение основных этапов их жизненного цикла включая порождение, завершение, получение и изменение состояния, а также работу с программным интерфейсом API для управления процессами.

Индивидуальное задание предполагает разработку приложения, способного возобновлять свою работу после завершения сообщением *WM\_CLOSE*. При получении команды завершения процесс штатно завершается, но перед этим порождает свою копию из того же исполняемого файла, которая продолжает работу вместо родительского процесса. Для обеспечения непрерывной обработки данных текущее состояние приложения записывается в файл, а процесс-наследник читает данные из файла для продолжения работы.

## 2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

### 2.1 Инициализация и загрузка состояния

При запуске приложения вызывается функция *LoadState()*, которая пытается открыть файл состояния «self\_healing\_state.dat» с помощью *CreateFileW()* с флагом *OPEN\_EXISTING* [1]. Если файл существует, его содержимое считывается функцией *ReadFile()* в переменную *g\_counter*, восстановливая тем самым предыдущее состояние счетчика. После успешного чтения файл удаляется функцией *DeleteFileW()*, чтобы при следующем «чистом» запуске приложения состояние не загружалось автоматически. Если файл состояния отсутствует, счетчик остается равным нулю, что соответствует первому запуску приложения.

### 2.2 Регистрация класса окна и создание интерфейса

Приложение регистрирует класс окна с именем «*SelfHealingProcessClass*» с помощью структуры *WNDCLASSW* и функции *RegisterClassW()*, указывая в качестве оконной процедуры функцию *WindowProc*. После успешной регистрации создается главное окно приложения функцией *CreateWindowExW()* с заголовком «Самовосстанавливающийся процесс» и стандартными параметрами оконного интерфейса [2]. Окно отображается на экране функциями *ShowWindow()* и *UpdateWindow()*, после чего устанавливается таймер *SetTimer()* с интервалом 1000 миллисекунд для периодического обновления счетчика.

### 2.3 Обработка сообщений и отображение состояния

Основной цикл сообщений *GetMessageW()* обрабатывает все поступающие сообщения через функцию *WindowProc()* [3]. При получении сообщения *WM\_PAINT* окно перерисовывается, отображая текущее значение счетчика с помощью функций *BeginPaint()*, *DrawTextW()* и *EndPaint()*. Сообщение *WM\_TIMER* обрабатывается увеличением значения *g\_counter* на единицу и вызовом *InvalidateRect()* для принудительной перерисовки окна, что обеспечивает визуальное отображение работающего счетчика, увеличивающегося каждую секунду.

### 2.4 Сохранение состояния приложения

При получении сообщения *WM\_CLOSE* приложение вызывает функцию *SaveState()*, которая создает файл «self\_healing\_state.dat» с помощью функции *CreateFileW()* с флагом *CREATE\_ALWAYS* для перезаписи существующего файла. В этот файл записывается текущее значение счетчика *g\_counter* с использованием функции *WriteFile()*, после чего файл закрывается функцией *CloseHandle()*. Такой подход обеспечивает сохранение текущего состояния

приложения перед его завершением, что позволяет новому экземпляру процесса продолжить работу с того же значения счетчика.

## 2.5 Создание нового экземпляра процесса

После сохранения состояния приложение вызывает функцию *CreateNewInstance()*, которая получает полный путь к исполняемому файлу текущего процесса с помощью *GetModuleFileNameW()* и сохраняет его в буфер *szPath*. Затем инициализируются структуры *STARTUPINFO* и *PROCESS\_INFORMATION*, необходимые для создания нового процесса. Функция *CreateProcessW()* запускает новый экземпляр того же приложения, передавая ему путь к исполняемому файлу, после чего закрываются дескрипторы процесса и потока с помощью *CloseHandle()*, поскольку родительский процесс не нуждается в контроле над дочерним процессом [4]. Новый экземпляр при запуске загрузит сохраненное состояние и продолжит работу с того же значения счетчика, обеспечивая таким образом непрерывность функционирования приложения.

### **3 ХОД ВЫПОЛНЕНИЯ ПРОГРАММЫ**

#### **3.1 Пример выполнения задания**

На рисунке 3.1 отображено главное окно работающего приложения. В нем выводится текущее значение счетчика, которое инкрементируется каждую секунду, а также собственный идентификатор процесса (*PID*).

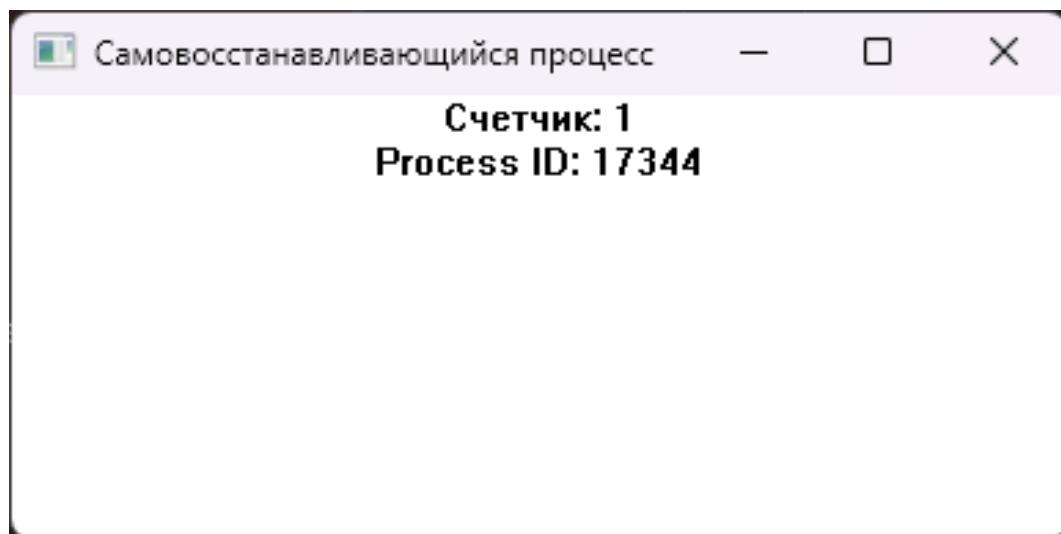


Рисунок 3.1 – Результат работы программы

На рисунке 3.2 показан результат работы механизма самовосстановления после закрытия предыдущего окна. Был запущен новый экземпляр приложения с новым идентификатором процесса (*PID*). При этом счетчик продолжил работу с сохраненного значения, демонстрируя успешное восстановление состояния.

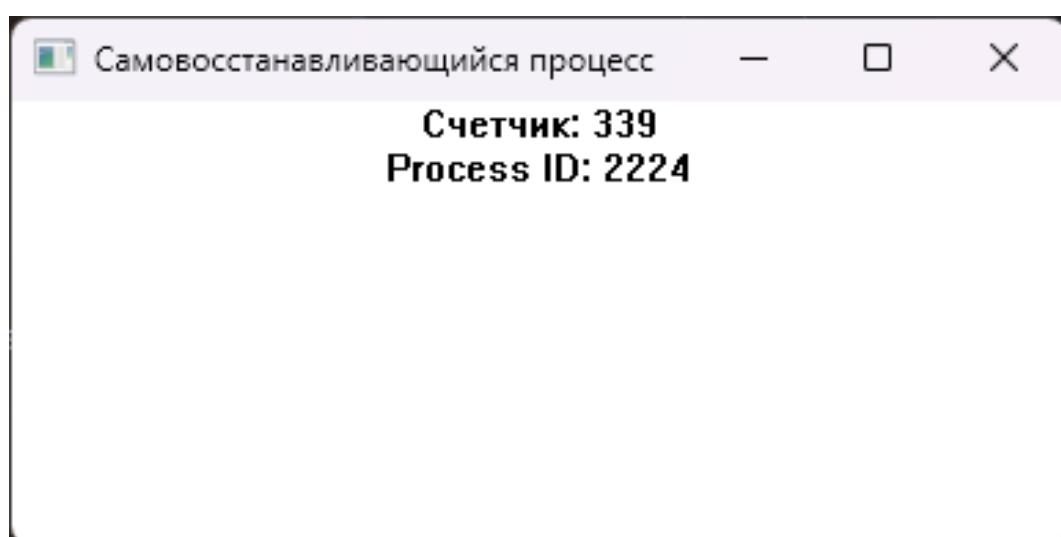


Рисунок 3.2 – Результат работы программы после закрытия

## **ВЫВОД**

В ходе выполнения лабораторной работы был разработан прототип самовосстанавливающегося приложения *Windows*, демонстрирующий основные концепции управления процессами и сохранения состояния приложения.

Основным достижением является успешная реализация механизма сохранения и загрузки состояния приложения. Приложение сохраняет текущее значение счетчика в файл перед завершением и восстанавливает его при запуске нового экземпляра, обеспечивая непрерывность функционирования. Кроме того, реализована функция создания нового процесса через *CreateProcessW()*, которая автоматически запускает новый экземпляр приложения при получении сигнала завершения.

Пользовательский интерфейс выводит текущее значение счетчика, идентификатор текущего процесса (*PID*) и идентификатор родительского процесса, позволяя наглядно видеть переход управления между процессами. Приложение корректно обрабатывает сообщения *WM\_PAINT*, *WM\_TIMER* и *WM\_CLOSE*, демонстрируя правильную работу основного цикла сообщений *Windows*.

Полученные навыки управления процессами и потоками, работа с *API* функциями *Windows* и понимание жизненного цикла процессов являются фундаментальными для системного программирования на платформе *Windows*.

## **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] WinAPI: Window Messages [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/fileio/creating-and-opening-files>. – Дата доступа: 14.09.2025.

[2] WinAPI: Using Window Procedures [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/winmsg/using-window-procedures>. – Дата доступа: 14.09.2025.

[3] WinAPI: Using Messages [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/winmsg/using-messages-and-message-queues>. – Дата доступа: 15.09.2025.

[4] WinAPI: Process and Thread Functions [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/procthread/process-and-thread-functions>. – Дата доступа: 15.09.2025.

# ПРИЛОЖЕНИЕ А

## (справочное)

### Исходный код программы

```
#define UNICODE
#define _UNICODE

#include <windows.h>
#include <tchar.h>
#include <string>

const wchar_t* STATE_FILE_NAME = L"self_healing_state.dat";
const wchar_t* CLASS_NAME = L"SelfHealingProcessClass";

int g_counter = 0;
HWND g_hwnd;

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
void SaveState();
void LoadState();
void CreateNewInstance();

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    LoadState();

    WNDCLASSW wc = { 0 };
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstance;
    wc.lpszClassName = CLASS_NAME;
    wc.hbrBackground = reinterpret_cast<HBRUSH>(COLOR_WINDOW + 1);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);

    if (!RegisterClassW(&wc)) {
        MessageBoxW(NULL, L"Window Registration Failed!", L"Error",
        MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    g_hwnd = CreateWindowExW(
        0,
        CLASS_NAME,
        L"Самовосстанавливающийся процесс",
        WS_OVERLAPPEDWINDOW,
        200, 200, 400, 200,
        NULL, NULL, hInstance, NULL
    );

    if (g_hwnd == NULL) {
        MessageBoxW(NULL, L"Window Creation Failed!", L"Error",
        MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    ShowWindow(g_hwnd, nCmdShow);
    UpdateWindow(g_hwnd);

    SetTimer(g_hwnd, 1, 1000, NULL);

    MSG msg = { 0 };
    while (GetMessageW(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessageW(&msg);
    }
}

return static_cast<int>msg.wParam;
```

```

}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    switch (uMsg) {
        case WM_PAINT: {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            wchar_t buffer[256];
            swprintf_s(buffer, L"Текущий счетчик: %d", g_counter);

            RECT rect;
            GetClientRect(hwnd, &rect);
            DrawTextW(hdc, buffer, -1, &rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER);

            EndPaint(hwnd, &ps);
            return 0;
        }

        case WM_TIMER: {
            g_counter++;
            InvalidateRect(hwnd, NULL, TRUE);
            return 0;
        }

        case WM_CLOSE: {
            SaveState();

            CreateNewInstance();

            DestroyWindow(hwnd);
            return 0;
        }

        case WM_DESTROY: {
            PostQuitMessage(0);
            return 0;
        }
    }
    return DefWindowProcW(hwnd, uMsg, wParam, lParam);
}

void SaveState() {
    HANDLE hFile = CreateFileW(STATE_FILE_NAME,
        GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        DWORD bytesWritten;
        WriteFile(hFile, &g_counter, sizeof(g_counter), &bytesWritten,
        NULL);
        CloseHandle(hFile);
    }
}

void LoadState() {
    HANDLE hFile = CreateFileW(STATE_FILE_NAME,
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL);

    if (hFile != INVALID_HANDLE_VALUE) {

```

```

        DWORD bytesRead;
        ReadFile(hFile, &g_counter, sizeof(g_counter), &bytesRead, NULL);
        CloseHandle(hFile);

        DeleteFileW(STATE_FILE_NAME);
    }

void CreateNewInstance() {
    wchar_t szPath[MAX_PATH];
    GetModuleFileNameW(NULL, szPath, MAX_PATH);

    STARTUPINFOW si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    if (!CreateProcessW(szPath, NULL, NULL, NULL, FALSE, 0, NULL, NULL,
&si, &pi)) {
        MessageBoxW(NULL, L"Failed to create new instance!", L"Error",
MB_OK);
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}

```