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 №2

на тему

Управление процессами

Студент П. С. Павлюткин

Преподаватель С. И. Сиротко

Минск 2023

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146752068)

[2 Теоретические сведения 4](#_Toc146752069)

[3 Результат выполнения 5](#_Toc146752070)

[Заключение 6](#_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 управлении процессами *Windows*, научиться создавать, завершать, изменять приоритеты процессов и потоков, а также изучить организацию и контроль исследование эффективности.Для достижения цели будет создано приложение демонстрирующее самовосстанавливающийся процесс.

1. **ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Управление процессами и потоками в операционной системе *Windows* является важной частью организации работы приложений и ресурсов компьютера.

Вычислительные процессы. Вычислительный процесс как системный объект, объект *Process* в *Windows*. Атрибуты процесса. Образ процесса. Адресное пространство и пространство дескрипторов. Состояния процесса. Создание (порождение) процессов. Иерархия процессов: родительские (*parent*) и дочерние (*child*) процессы. Завершение процессов. Код завершения. Программный интерфейс (*API*) для «базового» управления процессами.

Процесс в операционной системе *Windows* представляет собой независимую программу, выполняющуюся в собственном адресном пространстве памяти. Процесс обладает собственными системными ресурсами, такими как память и дескрипторы файлов. Поток легковесный исполняемый агент внутри процесса. Потоки внутри одного процесса могут совместно использовать ресурсы этого процесса, включая память и файлы. Один процесс может содержать несколько потоков, которые выполняются параллельно.

Для порождения новых процессов в *Windows* можно использовать функции, такие как *CreateProcess*. Это позволяет запускать новые исполняемые файлы в отдельных процессах. Для порождения новых потоков внутри процесса можно использовать функции, такие как *CreateThread*. Это позволяет создавать параллельные задачи внутри процесса. Завершение процессов и потоков может осуществляться с помощью функций *ExitProcess* и *ExitThread*. Эти функции завершают выполнение соответствующих процессов и потоков.

В *Windows* можно изменять приоритеты процессов и потоков для управления их доступом к ресурсам *CPU*. Высший приоритет обеспечивает больше времени *CPU* для выполнения задачи. Приоритеты процессов можно устанавливать с помощью функции *SetPriorityClass*, а приоритеты потоков - с помощью функции *SetThreadPriority*.

Для оценки эффективности работы процессов и потоков в *Windows* можно использовать различные инструменты мониторинга и профилирования, такие как *Windows* *Performance* *Monitor* и *Visual* *Studio* *Profiler*. Эти инструменты позволяют анализировать использование *CPU*, памяти и других ресурсов, а также выявлять узкие места в процессах и потоках для оптимизации производительности.

1. **РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ**

В результате выполнения лабораторной работы было создано приложение, демонстрирующее самовосстанавливающийся процесс (Рисунок 1).

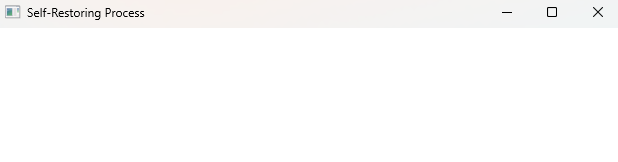


Рисунок 1 – Начальное окно приложения

После закрытия окна приложение вызывает MessageBox, в котором показываем рабочие данные, которые были созданы для теста нашей программы (Рисунок 2).

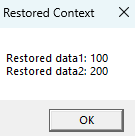


Рисунок 2– Вывод рабочих (восстановленных) данных

После нажатия на кнопку OK, окно приложения закрывается, и программа завершается.

# ЗАКЛЮЧЕНИЕ

В данной лабораторной работе изучены и применены следующие концепции и технологии, связанные с управлением процессами и оконными приложениями в среде *Windows*. Изучение создания и управления оконными приложениями с использованием *Windows API*. Создание окон - важный аспект управления процессами в графических приложениях. Демонстрация способности *Windows* создавать новые процессы при необходимости. Создание процессов - важный элемент управления процессами в многозадачной среде. Разработка механизма сохранения и восстановления контекста приложения. Контекст - это набор данных, в данном случае, *data1* и *data2*. Управление контекстом важно при переключении между процессами и сохранении состояния. Использование операторов `*new*` и `*delete*` для динамического выделения и освобождения памяти. Это важно для управления ресурсами процесса и избегания утечек памяти. Создание и настройка оконных процедур (обработчиков сообщений), таких как `*WndProc*`. Обработка оконных сообщений позволяет определять поведение приложения в ответ на события и действия пользователя. Главный цикл `*WinMain*`, который обрабатывает оконные сообщения, что обеспечивает отзывчивость и интерактивность приложения. Демонстрация работы с несколькими процессами в оконном приложении. Сохранение и восстановление контекста при переключении между процессами.

Таким образом, в данной лабораторной работе мы изучили и применили ключевые аспекты управления процессами и разработки оконных приложений в среде Windows.

# 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 Lab2.cpp

#include <windows.h>

#include <tchar.h>

#include <iostream>

#include <string>

/\*

Этот код представляет собой Windows-приложение, которое демонстрирует создание самовосстанавливающегося процесса. Вот пошаговое объяснение, что делает данный код:

1. Сначала определена структура `AppContext`, которая используется для хранения "рабочих" данных. В данном случае, у нас есть два целочисленных поля: `data1` и `data2`.

2. Создается указатель `g\_Context` для хранения экземпляра структуры `AppContext`. Этот указатель будет использоваться для сохранения и восстановления контекста приложения.

3. `SaveContext()` - это функция, которая сохраняет текущие рабочие данные в экземпляре `AppContext`. В данном случае, устанавливаются значения `data1` и `data2` равными 100 и 200.

4. `RestoreContext()` - это функция для восстановления ранее сохраненных данных и выполнения действий для продолжения работы. Если контекст был сохранен (т.е., `g\_Context` не равен `nullptr`), то данные восстанавливаются, и сообщение `MessageBox` выводит восстановленные значения `data1` и `data2`. После этого указатель `g\_Context` освобождается (удаляется).

5. В функции `WndProc` (обработчик оконных сообщений) обрабатывается сообщение `WM\_CLOSE`, которое генерируется при закрытии окна. В этом случае, вызывается `SaveContext()` для сохранения текущего контекста, а затем создается новое окно с тем же оконным классом (`SelfRestoringProcess`) и названием "Self-Restoring Process". После создания нового окна, вызывается `RestoreContext()` для восстановления контекста, а текущее окно завершает свою работу с помощью `PostMessage(hwnd, WM\_QUIT, 0, 0)`.

6. В функции `WinMain` создается окно с определенным оконным классом, и запускается цикл обработки сообщений с помощью `GetMessage` и `DispatchMessage`.

Таким образом, приложение позволяет сохранить контекст, создать новое окно и восстановить контекст в новом окне при закрытии старого окна.

\*/

// Структура для хранения полей наших "рабочих" данных

struct AppContext {

int data1;

int data2;

};

AppContext\* g\_Context = nullptr;

// Функция для сохранения контекста

void SaveContext() {

g\_Context = new AppContext;

// Здесь сохраняем рабочие данные

g\_Context->data1 = 100;

g\_Context->data2 = 200;

}

// Функция для восстановления контекста

void RestoreContext() {

// Восстанавливайте свои рабочие данные и выполняйте необходимые действия

if (g\_Context) {

std::string message = "Restored data1: " + std::to\_string(g\_Context->data1) + "\n";

message += "Restored data2: " + std::to\_string(g\_Context->data2);

MessageBoxA(NULL, message.c\_str(), "Restored Context", MB\_OK);

delete g\_Context;

g\_Context = nullptr;

}

}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {

switch (msg) {

case WM\_CLOSE:

{

SaveContext();

// Создайте новое окно для продолжения работы

WNDCLASS wc = { 0 };

wc.lpszClassName = \_T("SelfRestoringProcess");

wc.hInstance = GetModuleHandle(NULL);

wc.lpfnWndProc = WndProc;

RegisterClass(&wc);

HWND newHwnd = CreateWindow(wc.lpszClassName, \_T("Self-Restoring Process"), WS\_OVERLAPPEDWINDOW, 0, 0, 640, 480, 0, 0, GetModuleHandle(NULL), 0);

ShowWindow(newHwnd, SW\_SHOW);

UpdateWindow(newHwnd);

// Восстановите контекст в новом окне

RestoreContext();

// Завершите текущее окно

PostMessage(hwnd, WM\_QUIT, 0, 0);

}

break;

case WM\_QUIT:

PostQuitMessage(0);

break;

default:

return DefWindowProc(hwnd, msg, wParam, lParam);

}

return 0;

}

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

WNDCLASSEX wc = { 0 };

wc.cbSize = sizeof(WNDCLASSEX);

wc.style = 0;

wc.lpfnWndProc = WndProc;

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hInstance = hInstance;

wc.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

wc.hbrBackground = (HBRUSH)(COLOR\_WINDOW + 1);

wc.lpszMenuName = NULL;

wc.lpszClassName = \_T("SelfRestoringProcess");

wc.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

if (!RegisterClassEx(&wc)) {

MessageBox(NULL, \_T("Failed to register the window class"), \_T("Error"), MB\_ICONERROR);

return -1;

}

HWND hwnd = CreateWindow(\_T("SelfRestoringProcess"), \_T("Self-Restoring Process"), WS\_OVERLAPPEDWINDOW, 0, 0, 640, 480, NULL, NULL, hInstance, NULL);

if (!hwnd) {

MessageBox(NULL, \_T("Failed to create the window"), \_T("Error"), MB\_ICONERROR);

return -1;

}

ShowWindow(hwnd, nCmdShow);

UpdateWindow(hwnd);

MSG msg;

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

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return (int)msg.wParam;

}