

Міністерство освіти і науки України
Національний університет «Запорізька політехніка»

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт з дисципліни
“ Системне програмування ”
Частина 2
для студентів спеціальності
123 "Комп'ютерна інженерія"
всіх форм навчання

Методичні вказівки до виконання лабораторних робіт з дисципліни “Системне програмування” Частина 2 для студентів спеціальності 123 "Комп'ютерна інженерія" всіх форм навчання. /Укл. М.М.Хохлов, Т.С. Дьячук. - Запоріжжя: НУ «Запорізька політехніка», 2021.- 43 с.

Укладачі:

М.М.Хохлов, ст.викл.,
Т.С. Дьячук, асистент

Рецензент:

Р.К. Кудерметов, доцент, к.т.н.

Відповідальний за випуск:

Т.С. Дьячук, асистент

Затверджено
на засіданні кафедри КСМ
Протокол № 6 від 05.02.2021

Рекомендовано до видання
на засіданні НМК факультету КНТ
Протокол № 7 від 26.02.2021

ЗМІСТ

1 ЛАБОРАТОРНА РОБОТА № 3 ОБРОБКА ПОВІДОМЛЕНЬ ТАЙМЕРА.....	4
1.1 Загальні відомості.....	4
1.2 Завдання до лабораторної роботи № 3	10
1.3 Зміст письмового звіту.....	20
1.4 Контрольні запитання	20
2 ЛАБОРАТОРНА РОБОТА № 4 ОБМІН ІНФОРМАЦІЄЮ МІЖ ПРОЦЕСАМИ.....	21
2.1 Загальні відомості.....	21
2.2 Завдання до лабораторної роботи № 4	25
2.3 Зміст письмового звіту.....	33
2.4 Контрольні запитання	33
3 ЛАБОРАТОРНА РОБОТА № 5 ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ. СТРУКТУРОВАНА ОБРОБКА ВИНЯТКІВ.....	35
3.1 Загальні відомості.....	35
3.2 Завдання до лабораторної роботи № 5.....	40
3.3 Зміст письмового звіту	42
3.4 Контрольні запитання	42
РЕКОМЕНДОВАНА ЛІТЕРАТУРА.....	43

1 ЛАБОРАТОРНА РОБОТА № 3 ОБРОБКА ПОВІДОМЛЕНЬ ТАЙМЕРА

Мета роботи - познайомитися зі структурою віконного застосунку та з принципами побудови програм обробки повідомлень від таймера WM_TIMER.

1.1 Загальні відомості

Віконні застосунки будуються за принципами подієво-керованого програмування (event-driven programming) - стилю програмування, при якому поведінка компонента системи визначається набором можливих зовнішніх подій і відповідних реакцій компонента на них. Такими компонентами в Windows є вікна. З кожним вікном в Windows пов'язана певна функція обробки подій - віконна функція. Події для вікон називаються повідомленнями. Повідомлення відноситься до того чи іншого типу, ідентифікованому певним кодом (32-бітовим цілим числом), і супроводжується парою 32-бітних параметрів (wParam і lParam), інтерпретація яких залежить від типу повідомлення.

Завданням будь-якої віконної програми - створити головне вікно і повідомити Windows функцію обробки подій для цього вікна. Все найцікавіше для застосунка буде відбуватися саме в функції обробки подій головного вікна. У Windows програма пасивна. Після запуску вона чекає, коли їй приділить увагу операційна система. Операційна система робить це посилкою повідомлень. Повідомлення можуть бути різного типу, вони функціонують в системі досить хаотично, і додаток не знає, якого типу повідомлення прийде наступним. Логіка побудови Windows-застосунка повинна забезпечувати коректну і передбачувану роботу при надходженні повідомлень будь-якого типу.

Класичний віконний застосунок, як правило, складається принаймні з двох функцій:

- стартова функція, що створює головне вікно WinMain();
- функція обробки повідомлень вікна (віконна функція).

Точкою входу програми для Windows є функція WinMain(). Ця функція використовує послідовність викликів API і при завершенні повертає операційній системі ціле число.

```
int WINAPI WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    PSTR szCmdLine,
    int iCmdShow) {...}
```

Аргументи функції:

hInstance - дескриптор процесу (instance handle) - число, яке ідентифікує програму, коли вона працює під Windows. Якщо одночасно працюють кілька копій однієї програми, кожна копія має своє значення hInstance.

hPrevInstance - попередній дескриптор процесу (previous instance) - в даний час застарів, завжди дорівнює NULL.

szCmdLine - покажчик на рядок, що закінчується нулем, в якому містяться параметри, передані в програму з командного рядка. Можна запустити програму з параметром командного рядка, вставивши цей параметр після імені програми в командному рядку.

iCmdShow - ціле константне значення, що показує, яким має бути виведено на екран вікно в початковий момент. Задається при запуску програми іншою програмою. У більшості випадків число дорівнює 1 (SW_SHOWNORMAL).

Створення вікна здійснюється функцією

```
HWND WINAPI CreateWindow(...);
```

Значення, що повертається - дескриптор створюваного вікна. У разі неможливості створити вікно повертається NULL.

Відображення вікна здійснюється функцією

```
BOOL WINAPI ShowWindow(
    _In_ HWND hWnd,
    _In_ int nCmdShow);
```

Значення, що повертається: 1 - успішне відображення вікна, 0 - помилка.

Аргументи функції:

hWnd - дескриптор відображуваного вікна.

nCmdShow - константа, яка визначає, як буде відображатися вікно згідно з таблицею.

Перемальовування вікна здійснюється функцією

BOOL UpdateWindow(_In_ HWND hWnd);

Значення, що повертається: 1 - успішна перерисовка вікна, 0 - помилка. Аргумент функції hWnd - дескриптор вікна.

Прототипи функцій знаходяться в файлі бібліотеки user32.dll.

Після виклику функції UpdateWindow, вікно остаточно виведено на екран. Тепер програма повинна підготувати себе для отримання інформації від користувача через клавіатуру і мишу. Windows підтримує «чергу повідомлень» (message queue) для кожної програми, що працює в даний момент в системі Windows. Коли відбувається введення інформації, Windows перетворює її в «повідомлення», яке міститься в чергу повідомлень програми. Програма витягує повідомлення з черги повідомлень, виконуючи блок команд, відомий як «цикл обробки повідомлень» (message loop):

```
while(GetMessage(&msg,NULL,0,0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Для отримання повідомлення з черги використовується функція:

```
BOOL WINAPI GetMessage(
    _Out_ LPMMSG lpMsg, // покажчик на структуру повідомлення
    _In_opt_ HWND hWnd, // дескриптор вікна
    _In_ UINT wMsgFilterMin, // нижня межа фільтра
    // ідентифікаторів повідомлень
    _In_ UINT wMsgFilterMax); // верхня межа фільтра
    // ідентифікаторів повідомлень
```

У разі отримання з черги повідомлення, відмінного від WM_QUIT, повертає ненульове значення.

Функція

```
BOOL WINAPI TranslateMessage(_In_ const MSG *lpMsg);
```

передає аргумент - структуру msg назад в Windows для перетворення будь-якого повідомлення з клавіатури. Повертає ненульове значення в разі успішної розшифровки повідомлення, 0 - помилка.

Функція

`LRESULT WINAPI DispatchMessage(_In_ const MSG *lpmsg );`

передає аргумент - структуру msg назад в Windows. Windows відправляє повідомлення для його обробки відповідної віконної процедури - таким чином, Windows викликає відповідну віконну функцію, зазначену при реєстрації класу вікна.

Після того, як віконна функція обробить повідомлення, воно повертається в Windows, яка все ще обслуговує виклик функції DispatchMessage. Коли Windows повертає управління в стартову функцію WinMain () до наступного за викликом DispatchMessage коду, цикл обробки повідомлень в черговий раз відновлює роботу, викликаючи GetMessage.

Повертає значення, яке визначається віконної функцією, яке найчастіше ігнорується.

Віконна функція призначена для обробки повідомлень вікна. Функція обробки повідомлень вікна організована за принципом розгалуження, що складається з послідовної перевірки типу повідомлення. При збігу типу повідомлення, переданого в структурі Message з відповідною гілкою, здійснюється його обробка. Мінімальний вид віконної функції представлений нижче.

```
LONG WINAPI WndProc(HWND hwnd, UINT Message,
WPARAM wparam, LPARAM lparam)
{
    switch (Message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, Message, wparam, lparam);
    }
}
```

```
return 0;
}
```

Чотири аргументи віконної функції ідентичні першим чотирьом полям структури повідомлення MSG. У прикладі обробляється тільки один тип повідомлення WM_DESTROY, яке передається віконній функції при закритті вікна.

Виклик функції DefWindowProc() обробляє за замовчуванням всі повідомлення, які не виконує віконна процедура.

Функція PostQuitMessage() повідомляє Windows, що даний потік запитує завершення. Аргументом є ціле число, яке функція поверне операційній системі.

Обробка повідомлень від таймера - WM_TIMER.

Повідомлення WM_TIMER відправляється в чергу повідомлень потоку, що встановив таймер, або відповідній функції зворотного виклику TimerProc після закінчення інтервалу, заданого при установці таймера функцією SetTimer.

```
WM_TIMER
wTimerID = wParam; // ідентифікатор таймера
tmprc = (TIMERPROC *) lParam; // адреса функції зворотного
//виклику таймера
```

Параметри

wTimerID – значення wParam. Визначає ідентифікатор таймера.

tmprc – значення lParam. Вказує на функцію зворотного виклику, що визначається додатком, адреса якої була передана при установці таймера функцією SetTimer. Якщо значення параметра tmprc не рівне NULL, Windows передає повідомлення WM_TIMER вказаній функції зворотного виклику замість приміщення повідомлення в чергу повідомлень потоку.

Значення, що повертаються

Додаток повинен повертати нуль, якщо воно обробляє це повідомлення.

Функція SetTimer створює таймер з вказаним інтервалом спрацювання.

```
UINT SetTimer(
```

```

HWND hWnd, // дескриптор вікна для повідомлень таймера
UINT nIDEvent, // ідентифікатор таймера
UINT uElapse, // інтервал спрацьовування таймера
TIMERPROC lpTimerFunc // адреса процедури таймера
);

```

Параметри

hWnd – ідентифікує вікно, пов'язане з таймером. Вікном повинен володіти викликаємий потік. Якщо значення цього параметра дорівнює **NULL**, з таймером не зв'язується ніяке вікно і параметр **nIDEvent** ігнорується.

nIDEvent – визначає ненульовий ідентифікатор таймера. Якщо значення параметра **hWnd** дорівнює **NULL**, цей параметр ігнорується.

uElapse – визначає інтервал спрацьовування в мілісекундах.

lpTimerFunc – вказує на функцію, що повідомляється про закінчення заданого інтервалу.

Якщо значення **lpTimerFunc** дорівнює **NULL**, система відправляє повідомлення **WM_TIMER** в чергу повідомлень додатку. Значення члена **hwnd** структури повідомлення типу **MSG** містить значення параметра **hWnd**.

Значення, що повертаються

У разі успіху повертається цілочисельний ідентифікатор нового таймера. Додаток може передавати цей ідентифікатор у функцію **KillTimer** для руйнування таймера.

Якщо не вдалося створити новий таймер, повертається нуль.

Додаток може обробляти повідомлення **WM_TIMER** у віконній процедурі додатку, з відповідним включенням в її початковий код оператора **case**, або указуючи при створенні таймера функцію зворотного виклику **TimerProc**. Коли ви вкажете функцію зворотного виклику, функція **DispatchMessage** просто викликає її замість віконної процедури.

Значення параметра **wParam** повідомлення **WM_TIMER** містить значення параметра **nIDEvent**

Наприклад:

case IDM_M_START:

```

SetTimer (hWnd, // дескриптор вікна для повідомлень таймера

```

```

1, // дескриптор вікна для повідомлень таймера
1000, // інтервал спрацьовування таймера
(TIMERPROC)NULL); /*вказує на функцію, що повідомляється
про закінчення заданого інтервалу. Якщо значення дорівнює NULL,
система відправляє повідомлення WM_TIMER в чергу повідомлень.*/
    counter=0;
    break;

```

1.2 Завдання до лабораторної роботи № 3

Програма 1 - Обробка повідомлень таймера.

```

#include "MOVIES.h"
#include <conio.h>
#include <stdio.h>
#include <math.h>

TCHAR string[]="Привіт всім!";

// Головна функція:
int WINAPI WinMain(HINSTANCE hInstance
                  HINSTANCE hPrevInstance
                  LPSTR lpszCmdLine, int nCmdShow)
{
    MSG msg;// структура для роботи з повідомленнями;

    int nRc;

    strcpy(szAppName, "MOVIES");
    hInst = hInstance;

    if((nRc = nCwRegisterClasses()) == -1)
    { // Відладка !
        MessageBox(NULL, "Not Register Class", "Main", MB_OK);
        return FALSE;
    }
    else
        MessageBox(NULL, "Register Class", "Main", MB_OK);

```

```

// Створити вікно:
hWndMain = CreateWindow(
    szAppName, //Назва зареєстрованого класу вікна;
    "Movies", // Заголовок вікна;
    WS_OVERLAPPEDWINDOW, //Стиль вікна;
    //Задати розміри й розташування вікна, прийняті за замовчуванням:
    CW_USEDEFAULT, 0, // координати x, y;
    CW_USEDEFAULT, 0, // ширина та висота вікна;
    0, // дескриптор батьківського вікна;
    0, // дескриптор меню (меню немає);
    hInst, //ідентифікує екземпляр модуля, який буде
    //пов'язаний з вікном. |
    0); // вказує на значення, передане вікну при створенні.
// Відладка !
MessageBox(NULL, "Create Widow", "Main", MB_OK);
if (hWndMain ==NULL) {
    MessageBox(NULL, "hWnd==NULL", "Main", MB_OK);
    return FALSE;}

ShowWindow(hWndMain, nCmdShow);

//Запускаємо цикл обробки повідомлень:
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
CwUnRegisterClasses();
return msg.wParam;
}

LRESULT CALLBACK WindowProc(
    HWND hWnd, UINT Message
    WPARAM wParam, LONG lParam)
{
    HMENU hMenu = 0;
    HBITMAP hBitmap = 0;

```

```

HDC hDC;
PAINTSTRUCT ps;
Int nRc=0;
Float PI = 3.14159265;
Static int counter = 0;
static int x,y;
HPEN hPen, hWhitePen;
TCHAR str[255]= {0};
SYSTEMTIME SystemTime;
switch (Message)
{
case WM_COMMAND:
    switch (wParam)
    {
case IDM_M_START:
SetTimer (hWnd, 1, 1000, (TIMERPROC) NULL);
        counter=0;
break;
case IDM_M_STOP:
        KillTimer (hWnd, 1);
break;
        default:
return DefWindowProc(hWnd, Message, wParam, lParam);
    }
break;
case WM_TIMER:
hPen = CreatePen (PS_SOLID, 3, RGB(255,0,0)); //Колір стрілки
hWhitePen = CreatePen (PS_SOLID, 5, RGB(255,255,255));
hDC = GETDC (hWnd);
case IDM_M_START:
SetTimer (hWnd, 1, 1000, (TIMERPROC) NULL);
        counter=0;
break;
        SelectObject(hDC, hWhitePen);
//Малюємо лінію з поточної позиції до вказаної точки
x=175+(int) 100.0*sin((float)counter*PI/6.0);
y=150-(int) 100.0*cos((float)counter*PI/6.0);
LineTo(hDC, // дескриптор контексту пристрою

```

```

x,// x-координата точки, що завершує лінію
y,// y- координата точки, що завершує лінію
);
counter%=12;
counter++;
MoveToEx(hDC, 175, 150, NULL);
x = 175 + (int)100*sin(counter*PI/6.0);
y = 150 - (int) 100*cos(counter*PI/6.0);
SelectObject(hDC, hPen);
LineTo(hDC,x,y);
GetLocalTime(&SystemTime);

```

```

sprintf(str,"%02d:%02d:%02d",SystemTime.wHour,SystemTime.w
Minute,SystemTime.wSecond);
TextOut(hDC, 0,0,str,strlen(str));

```

```

TextOut(hDC
175 + (int)120*sin(counter*PI/6.0),
145 - (int)120*cos(counter*PI/6.0), string+counter-1,1);
RELEASEDC (hWnd, hDC);
DeleteObject (hPen);
DeleteObject (hWhitePen);

```

break;

// Обробка запиту на перемальовування вікна

case WM_PAINT:

```

hDC=BeginPaint (hWnd &ps);//Отримати контекст пристрою
EndPaint (hWnd &ps);// Звільнити контекст пристрою

```

break;

// Закрити вікно

case WM_CLOSE:

```

MessageBox крапки "Вікно закривається", "Movie", MB_OK);
KillTimer ( hWnd, 1);
DestroyWindow(hWnd);
if (hWnd == hWndMain)
PostQuitMessage(0);

```

break;

default:

```

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

```

```

    }
    return 0L;
}
int nCwRegisterClasses(void)
{
    WNDCLASS wndclass;
    memset(&wndclass, 0x00, sizeof(WNDCLASS));
    wndclass.style = CS_HREDRAW | CS_VREDRAW |
CS_BYTEALIGNWINDOW;
    wndclass.lpfnWndProc = WindowProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInst;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = GetStockObject(GRAY_BRUSH);
    wndclass.lpszMenuName = szAppName;
    wndclass.lpszClassName = szAppName;
    if(!RegisterClass(&wndclass)) return -1;
    return(0);
}
void CwUnRegisterClasses(void)
{
    WNDCLASS wndclass;
    memset(&wndclass, 0x00, sizeof(WNDCLASS));
    UnregisterClass(szAppName, hInst);
}

```

Файл ресурсів ***.rc**

```
#include "MOVIES.h"
```

```
MOVIES MENU
```

```
{
```

```
POPUP "&Rotation"
```

```
{
```

```
    MENUITEM "&Start", IDM_M_START
```

```
    MENUITEM "S&top", IDM_M_STOP
```

```
    MENUITEM SEPARATOR
```

```
    MENUITEM "E&xit", IDM_M_EXIT
```

```

    }
    POPUP "&Help"
{
    MENUITEM "&About", IDM_H_ABOUT
    }
}

```

Файл заголовків **movies.h**

```

#include <windows.h>
#include <string.h>
#define IDM_ROTATION      1000
#define IDM_M_START      1050
#define IDM_M_STOP       1100
#define IDM_M_EXIT       1250
#define IDM_HELP         2000
#define IDM_H_ABOUT      2050
char szString[128];
char szAppName[20];
HINSTANCE hInst;
HWND hWndMain;
LRESULT CALLBACK WindowProc (HWND, WORD, WORD,
LONG);
int nCwRegisterClasses(void);
void CwUnRegisterClasses(void);

```

Завдання до програми 1

1. Замініть заголовок створеного вікна на заголовок, що містить наступні дані: номер групи, Ваше прізвище і номер варіанту.
2. Модифікуйте програму так, щоб була можливість увімкнути і вимкнути таймер способом, вказаним у варіанті. Після ввімкнення / вимикання таймера необхідно вивести повідомлення.
3. Обробіть натиснення літери(знак) згідно варіанту.
4. Перемістіть центр годинника до координати згідно варіанту.

Примітка:

Прямокутники повинні виводитися у випадковому місці, але не за лінією вікна. Використовуйте функцію Random.

Таблиця 1.1 - Варіанти завдань

<i>№</i>	<i>Спосіб ввімкнення і вимкнення таймера</i>	<i>Знак</i>	<i>Дія при натисненні і клавіші</i>	<i>Обробник переривань таймера</i>	<i>Подія при спрацьовуванні таймера</i>	<i>Коор дина ти</i>
1	Головне меню	G	Прискор юється рахунок таймера удвічі	Окремою процедурою	Вивести повідомлення з кількістю спрацьовування таймера	(10, 100)
2	Лівий клік миші вмикає, правий вимикає	H	Прискор юється рахунок таймера удвічі	У віконній процедурі	Вивести у вікно кількість спрацьовування таймера	(200, 20)
3	Подвійний лівий клік миші вмикає, подвійний правий вимикає	J	Прискор юється рахунок таймера удвічі	У віконній процедурі	При парній кількості разів виводити прямокутник довільного кольору	(30, 300)
4	Лівий клік миші в лівій частині вікна вмикає, в правій – вимикає	U	Прискор юється рахунок таймера удвічі	У віконній процедурі	При не парній кількості разів виводити прямокутник довільного кольору	(400, 40)
5	Головне меню	O	Сповільнює ться рахунок таймера удвічі	У віконній процедурі	Вивести довільне повідомлення, коли значення лічильника буде рівне номеру варіанту	(50, 500)

Продовження таблиці 1.1

6	Лівий клік миші вмикає, правий вимикає	P	Сповільнюється рахунок таймера удвічі	Окремою процедурою	Вивести повідомлення з кількістю спрацьовування таймера	(600, 60)
7	Подвійний лівий клік миші вмикає, подвійний правий вимикає	Y	Сповільнюється рахунок таймера удвічі	Окремою процедурою	Вивести у вікно кількість спрацьовування таймера	(70, 550)
8	Лівий клік миші в лівій частині вікна вмикає, в правій— вимикає	L	Сповільнюється рахунок таймера удвічі	Окремою процедурою	При парній кількості разів виводити прямокутник довільного кольору	(800, 80)
9	Головне меню	K	Прискорюється рахунок таймера удвічі	Окремою процедурою	При не парній кількості разів виводити прямокутник довільного кольору	(100, 100)
10	Лівий клік миші вмикає, правий вимикає	T	Прискорюється рахунок таймера удвічі	У віконній процедурі	Вивести довільне повідомлення, коли значення лічильника буде рівне варіанту	(550, 650)
11	Подвійний лівий клік миші вмикає, подвійний вимикає	R	Прискорюється рахунок таймера удвічі	У віконній процедурі	Вивести повідомлення з кількістю спрацьовування таймера	(200, 200)

Продовження таблиці 1.1

12	Лівий клік миші вмикає, правий вимикає	E	Прискорюється рахунок таймера удвічі	У віконній процедурі	Вивести у вікно кількість спрацьовування таймера	(750, 150)
13	Головне меню	F	Сповільнюється рахунок таймера удвічі	У віконній процедурі	При парній кількості разів виводити прямокутник довільного кольору	(300, 300)
14	Лівий клік миші вмикає, правий вимикає	D	Сповільнюється рахунок таймера удвічі	Окремою процедурою	При не парній кількості разів виводити прямокутник довільного кольору	(600, 600)
15	Подвійний лівий клік миші вмикає, правий вимикає	W	Сповільнюється рахунок таймера удвічі	Окремою процедурою	Вивести довільне повідомлення, коли значення лічильника буде рівне номеру варіанту	(500, 0)
16	Клік миші в лівій частині вікна вмикає, в правій – вимикає	Q	Сповільнюється рахунок таймера удвічі	Окремою процедурою	Вивести повідомлення з кількістю спрацьовування таймера	(0, 650)
17	Головне меню	S	Прискорюється рахунок таймера удвічі	Окремою процедурою	Вивести у вікно кількість спрацьовування таймера	(250, 0)

Продовження таблиці 1.1

18	Лівий клік миші вмикає, правий вимикає	A	Прискорюється рахунок таймера удвічі	У віконній процедурі	При парній кількості разів виводити прямокутник довільного кольору	(0, 600)
19	Подвійний лівий клік миші вмикає, подвійний правий вимикає	N	Прискорюється рахунок таймера удвічі	У віконній процедурі	При не парній кількості разів виводити прямокутник довільного кольору	(800, 90)
20	Лівий клік миші в лівій частині вікна вмикає, в правій – вимикає	C	Прискорюється рахунок таймера удвічі	У віконній процедурі	Вивести довільне повідомлення, коли значення лічильника буде рівне номеру варіанту	(750, 0)
21	Головне меню	M	Сповільнюється рахунок таймера удвічі	У віконній процедурі	Вивести повідомлення з кількістю спрацьовування таймера	(10, 100)
22	Лівий клік миші вмикає, правий вимикає	Z	Сповільнюється рахунок таймера удвічі	Окремою процедурою	Вивести у вікно кількість спрацьовування таймера	(10, 100)
23	Подвійний лівий клік миші вмикає, подвійний правий вимикає	X	Сповільнюється рахунок таймера удвічі	Окремою процедурою	При парній кількості разів виводити прямокутник довільного кольору	(10, 100)

Продовження таблиці 1.1

24	Лівий клік миші в лівій частині вікна вмикає, в правій – вимикає	V	Сповільнюється рахунок таймера удвічі	Окремою процедурою	При не парній кількості разів виводити прямокутник довільного кольору	(10, 100)
25	Головне меню	I	Прискорюється рахунок таймера удвічі	У віконній процедурі	Вивести довільне повідомлення, коли значення лічильника буде рівне номеру варіанту	(10, 100)

1.3 Зміст письмового звіту

1. Тексти програм з вказаними змінами.
2. Короткі відповіді на контрольні запитання.

1.4 Контрольні запитання

1. У яких випадках відправляється повідомлення WM_TIMER? Напишіть фрагмент програми, що оброблює це повідомлення.
2. Опишіть параметри і призначення функції SetTimer. Що повертає функція у разі успішного виконання?
3. Опишіть параметри і призначення функції KillTimer.
4. Напишіть програму календаря - годинника, яка виводить в маленькому вікні поточну дату та час і періодично, кожну секунду, оновлює цю інформацію.
5. *Напишіть програму, що використовує таймер для визначення заданого нею інтервалу часу і повідомлення користувачу про закінчення цього інтервалу виводом на екран застереження.

2 ЛАБОРАТОРНА РОБОТА № 4 ОБМІН ІНФОРМАЦІЄЮ МІЖ ПРОЦЕСАМИ

Мета роботи - познайомитися з створення дочірнього процесу та із засобами обміну інформацією між процесами.

2.1 Загальні відомості

Пам'ять

Організація пам'яті під Win32 з погляду додатку достатньо проста. Модель пам'яті, що використовується, називається плоскою моделлю пам'яті. У цій моделі всі сегментні регістри (або селектори) указують на одну і ту ж 32-бітну стартову адресу і зсув, так що додаток може звернутися до будь-якої точки пам'яті свого адресного простору. Більше немає "дальніх" і "ближніх" покажчиків. Для виділення блоку пам'яті використовуються функції GlobalAlloc або GlobalLock. Ці функції повертають дескриптор на блок пам'яті. «Відкрити» блок пам'яті можна за допомогою виклику функції GlobalUnlock. Ця функція повертає покажчик на блок пам'яті. Звільнити блок пам'яті за допомогою GlobalFree.

Алгоритм програми:

1. Відкрийте або створіть файл функцією CreateFile. У разі успіху вона повертає дескриптор файлу або пристрою.
2. Перемістите файловий покажчик в бажане місцезположення функцією SetFilePointer.
3. Проведіть операцію читання або запису за допомогою виклику ReadFile або WriteFile. перед цим ви повинні зарезервувати достатньо великий блок пам'яті даних.
4. Закрийте файл за допомогою CloseHandle. Ця функції приймає дескриптор файлу.

Якщо файл більше, ніж зарезервований блок пам'яті? Або якщо рядок буде обрізано посередині, тому що кінчився блок пам'яті? Традиційна відповідь на перше питання - послідовно читати дані з файлу, поки він не кінчиться. Відповіддю на друге питання є те, що ви повинні обробляти подібну можливість. Це називається проблемою граничного значення.

File mapping (FM) - це порятунок. Використовуючи його, ви можете вважати весь файл вже завантаженим в пам'ять і

використовувати покажчик на пам'ять, щоб читати або писати дані у файл. Немає необхідності використовувати API пам'яті і файлові API одночасно, в FM це одне і те ж. FM також використовується для обміну даними між процесами. При використанні FM таким чином, реально не використовується ніякий файл. Це більше схоже на блок пам'яті, який можуть бачити всі процеси. Але обмін даними між процесами - вельми делікатний предмет. Ви повинні забезпечити синхронізацію між процесами та потоками.

Алгоритм програми:

1. Виклик CreateFile для відкриття файлу.
2. Виклик CreateFileMapping, якою передається дескриптор файлу, повернений CreateFile. Ця функція створює FM-об'єкт з файлу, створеного CreateFile'ом.
3. Виклик MapViewOfFile, щоб завантажити весь файл в пам'ять. Ця функція повертає покажчик на перший байт файлу.
4. Використовуйте покажчик, щоб писати або читати з файлу.
5. Викличте UnmapViewOfFile, щоб вивантажити файл.
6. Виклик CloseHandle знову, передавши йому на цей раз дескриптор файлу, повернений CreateFile, щоб закрити сам файл

Процеси і потоки

"Процес - це додаток, що виконується, він складається з особистого віртуального адресного простору, коду, даних і інших ресурсів операційної системи, таких як файли, пайпи і об'єкти синхронізації, видимі для процесу."

Примітка. Пайп(pipe) - канал або дорога з двома кінцями. Ви можете використовувати пайп, щоб обмінюватися даними між двома різними процесами або всередині одного процесу.

Як ви можете бачити з вищенаведеного визначення, у процесу є декілька об'єктів: адресний простір, виконуваний модуль (модулі) і все, що ці модулі створюють або відкривають. Як мінімум, процес повинен складатися з модуля, що виконується, особистого адресного простору і потоку. Кожен екземпляр програми відповідає тільки одному процесу, і кожен процес автоматично має в своєму розпорядженні потік, який позначає те, що власне підлягає виконанню ланцюжка дій. Що таке потік? Фактично, потік - це черга повідомлень, що виконується. Коли Windows вперше створює процес, вона робить тільки один потік на процес. Цей потік зазвичай починає виконання з першої інструкції в модулі. Коли Windows отримує команду для

створення процесу, вона створює особистий адресний простір для процесу, а потім вона завантажує виконуваний файл в простір. Після цього вона створює основний потік для процесу.

У Win 32 повідомлення посилаються потокам, і потоки ділять між собою процесорний час. Потоки можуть створювати нові потоки, причому створений і той, що створює потоки належать одному і тому ж процесу. Під Win32 ви також можете створити процеси зі своїх програм за допомогою функції `CreateProcess`. Параметри цієї функції:

1. `lpApplicationName` - ім'я виконуваного файлу з або без шляху, який ви хочете запустити. Якщо параметр рівний нулю, ви повинні надати ім'я виконуваного файлу в параметрі `lpCommandLine`.

2. `lpCommandLine` - аргументи командного рядка до програми, яку вам потрібно запустити. Якщо `lpApplicationName` дорівнює нулю, цей параметр повинен містити також ім'я виконуваного файлу. Наприклад так: `"notepad.exe readme.txt"`.

3. `lpProcessAttributes` і `lpThreadAttributes` - вкажіть атрибути безпеки для процесу та основного потоку. Якщо вони рівні `NULL`, то використовуються атрибути безпеки за замовченням.

4. `bInheritHandles` - ознака, яка вказує, чи хочете ви, щоб новий процес успадковував всі відкриті дескриптори з вашого процесу.

5. `dwCreationFlags` - декілька ознак, які визначають поведінку процесу, який ви хочете створити, наприклад, чи хочете ви, щоб процес був створений, але тут же припинений, щоб ви могли перевірити його або змінити, перш, ніж він запуститься. Ви також можете вказати клас пріоритету потоку в новому процесі. Цей клас пріоритету використовується, щоб визначити планований пріоритет гілок всередині процесу. Зазвичай ми використовуємо ознаку `NORMAL_PRIORITY_CLASS`.

6. `lpEnvironment` - покажчик на блок пам'яті, який містить декілька змінних оточення для нового процесу. Якщо цей параметр дорівнює `NULL`, новий процес успадковує їх від батьківського процесу.

7. `lpCurrentDirectory` - покажчик на рядок, який вказує поточний диск і директорію для дочірнього процесу. `NULL` - якщо ви хочете, щоб дочірній процес успадкував їх від батьківського процесу.

8. `lpStartupInfo` - вказує на структуру `STARTUPINFO`, яка визначає, як повинне з'явитися основне вікно нового процесу. Ця структура містить багато членів, які визначають появу головного вікна

дочірнього процесу. Якщо ви не хочете нічого особливого, ви можете заповнити дану структуру значеннями батьківського процесу, викликавши функцію `GetStartupInfo`.

9. `lpProcessInformation` - вказує на структуру `PROCESS_INFORMATION`, яка отримує ідентифікаційну інформацію про новий процес.

Структура `PROCESS_INFORMATION` має наступні параметри:

`PROCESS_INFORMATION` STRUCT

`hProcess` HANDLE; дескриптор дочірнього процесу

`hThread` HANDLE; дескриптор основного потоку дочірнього процесу

`dwProcessId` DWORD; ID дочірнього процесу

`dwThreadId` DWORD; ID основного потоку

Дескриптор процесу і ID процесу - це дві різні речі. ID процесу - це унікальний ідентифікатор процесу в системі. Дескриптор процесу - це значення, Windows, що повертається, для використання іншими API-функціями, пов'язаними з процесами. Дескриптор процесу не може використовуватися для ідентифікації процесу, оскільки він не унікальний.

Потоки.

Якщо говорити в термінах безпосередньої реалізації, потік - це функція, яка виконується паралельно з основною програмою. Ви можете запустити декілька екземплярів однієї і тієї ж функції або ви можете запустити декілька функцій одночасно, залежно від ваших вимог. Мультипоточність властива Win32, під Win16 аналогів не існує. Ви можете створювати додаткові потоки у вашій програмі. Можна рахувати Мультипоточність як багатозадачність всередині однієї програми. Треди виконуються в тому ж процесі, тому вони мають доступ до всіх ресурсів процесу: глобальним змінним, дескрипторам і так далі. Створити тред можна за допомогою виклику функції `CreateThread`. Функція `CreateThread` схожа на `CreateProcess`.

`lpThreadAttributes` - Ви можете використовувати `NULL`, якщо хочете, щоб у треда були установки безпеки за замовченням.

`dwStackSize` - вкажіть розмір стека треда. Якщо ви хочете, щоб тред мав такий же розмір стека, як і у основного, використовуйте `NULL` як параметр.

`lpStartAddress` - адреса функції треда. Ця функція виконуватиме

призначену для треда роботу. Вона повинна отримувати один і лише один 32-бітний параметр і повертати 32-бітове значення.

lpParametr - параметр, який ви хочете передати функції треда.

dwCreationFlags - 0 означає, що тред починає виконуватися відразу ж після його створення. Для зворотного можна використовувати ознаку **CREATE_SUSPEND**.

lpThreadId - **CreateThread** помістить сюди ID створеного треда.

Якщо виклик **CreateThread** пройшов успішно, вона повертає дескриптор створеного треда, інакше вона повертає **NULL**.

Отримати код виходу треда можна за допомогою функції **GetExitCodeThread**. Якщо необхідно перервати тред з іншого треда, то викликається функція **TerminateThread**. Треди розділяють ресурси процесу, включаючи глобальні змінні, тому треди можуть використовувати їх для того, щоб взаємодіяти один з одним, при цьому необхідно спланувати синхронізацію.

Розробити програмний комплекс, що складається з двох застосувань. Перше застосування буде процес - відправник, другий, – процес - приймач. У процесі - приймачеві за допомогою меню можна відкрити модальний діалог з інформацією про програму; у процесі - відправнику передбачена можливість виконання наступних дій:

Створення дочірнього процесу - приймача і визначення дескриптора його головного вікна;

Посилка в процес - приймач повідомлення **WM_SETTEXT**, що змінює заголовок його головного вікна;

Посилка в процес - приймач повідомлення **WM_COMMAND**, що імітує вибір в цьому застосуванні певного пункту меню;

Посилка в процес - приймач повідомлення **WM_DESTROY** “насильницького” завершення цього застосування.

2.2 Завдання до лабораторної роботи № 4

Програма 1.2а

Процес - відправник

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "1.h"
```

```

char szClassName[]="WinSnd";//Ім'я класу головного вікна
char  szTitle[]="Додаток_1-Відправник";//Заголовок  головного
вікна
HWND  hwndRcv;//Дескриптор  головного  вікна  процесу  -
приймача
/*Головна функція WinMain*/
int  WINAPI  WinMain(HINSTANCE  hInst,HINSTANCE,LPSTR
,int)
{
    MSG Msg;
    WNDCLASS wc;
    ZeroMemory(&wc,sizeof(wc));
    wc.lpfnWndProc=WndProc;
    wc.hInstance=hInst;
    wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
    wc.hCursor=LoadCursor(NULL,IDC_ARROW);
    wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
    wc.lpszMenuName="Main";
    wc.lpszClassName=szClassName;
    RegisterClass(&wc);
    //Створимо головне вікно
    HWND hwnd=CreateWindow(szClassName,szTitle
        WS_OVERLAPPEDWINDOW,20,20,300,200
        HWND_DESKTOP,NULL,hInst,NULL);
    //Показуємо вікно
    ShowWindow(hwnd,SW_NORMAL);
    //Цикл обробки повідомлень
    while(GetMessage(&Msg,NULL,0,0))
        DispatchMessage(&Msg);
    return 0;
}
//Виконна функція головного вікна
LRESULT  CALLBACK  WndProc(HWND  hwnd,UINT
msg,WPARAM wParam,LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hwnd,WM_PAINT,OnPaint);

```

```

HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
default:
return(DefWindowProc(hwnd,msg,wParam,lParam));
}
}
/*Функція обробки повідомлень WM_COMMAND від пунктів
меню*/
void OnCommand(HWND hwnd,int id,HWND,UINT)
{
switch(id)
{
case MI_NEW://Запускаем процес приймач
{
STARTUPINFO si;
PROCESS_INFORMATION pi;
ZeroMemory(&si,sizeof(si));
si.cb=sizeof(si);
hwndRcv=FindWindow("WinRcv",NULL);
if(!hwndRcv)
{
CreateProcess(NULL,"2.exe",NULL,NULL,FALSE,0,NULL,NUL
L,&si,&pi);
while(!hwndRcv) //Дождемося запуску процесу
//Отримаємо дескриптор вікна
hwndRcv=FindWindow("WinRcv",NULL);
}
else
MessageBox(NULL,"Процесс запущен","Info",MB_OK);
break;
} //Кінець case MI_NEW
case MI_LINK1: //Змінємо заголовок процесу - приймачу
SendMessage(hwndRcv,WM_SETTEXT,0,(LPARAM)
"Повідомлення отримане: Новий заголовок");
break;
case MI_LINK2: //Виберемо пункт меню процесу - приймача
SendMessage(hwndRcv,WM_COMMAND,(LPARAM)
MI_ABOUT,(LPARAM) 0);

```

```

    break;
case MI_LINK3: //Завершимо процес - приймач
    SendMessage(hwndRcv,WM_DESTROY,0,0);
    break;
case MI_EXIT://Завершимо процес - відправник
    DestroyWindow(hwnd);
}
}
/*Функція обробки повідомлень WM_PAINT*/
void OnPaint(HWND hwnd)
{
    char szT[]="Процес- відправник ";
    PAINTSTRUCT ps;
    HDC hdc=BeginPaint(hwnd &ps);
    TextOut(hdc,5,10,szT,strlen(szT));
    EndPaint(hwnd,&ps);
}
/*Функція обробки повідомлень WM_DESTROY*/
void OnDestroy(HWND)
{
    PostQuitMessage(0);
}

```

Файл 12a.rc

```

#include "1.h"
/*Сценарій меню*/
Main MENU{
    POPUP "Файл" {
        MENUITEM "Запуск процеса",MI_NEW
        MENUITEM "Зміна заголовка",MI_LINK1
        MENUITEM "Вибір пункту меню",MI_LINK2
        MENUITEM "Завершення процеса",MI_LINK3
        MENUITEM SEPARATOR
        MENUITEM "Вихід",mi_exit
    }
}

```

Файл 1.h

```

#include <windows.h>

```

```

#define MI_ABOUT    100    //Ідентифікатори пункту меню
процеса-приймача
#define MI_NEW      100
#define MI_LINK1    101    // Ідентифікатори пунктів меню
даного процесу
#define MI_LINK2    102
#define MI_LINK3    103
#define MI_LINK5    1002
#define MI_EXIT     104
LRESULT CALLBACK WndProc(HWND,UINT
WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnPaint(HWND);
void OnCopyData(HWND,HWND,COPYDATASTRUCT*);

```

Програма 1.2b

Процес-приймач

```

#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "2.h"
char szClassName[]="WinRcv"; //Ім'я класу вікна-приймача
char szTitle[]="Додаток_2-Приймач"; //Заголовок вікна-приймача
HINSTANCE hInstance; //Дескриптор застосування
/*Головна функція WinMain*/
int WINAPI WinMain(HINSTANCE
hInst,HINSTANCE,LPSTR,int)
{
    MSG Msg;
    WNDCLASS wc;
    ZeroMemory(&wc,sizeof(WNDCLASS));
    wc.lpfnWndProc=WndProc;
    wc.hInstance=hInst;
    wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);

```

```

wc.hCursor=LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
wc.lpszMenuName="Main";
wc.lpszClassName=szClassName;
RegisterClass(&wc);
//Створимо вікно
HWND hwnd=CreateWindow(szClassName,szTitle
    WS_OVERLAPPEDWINDOW,310,50,300,200
    HWND_DESKTOP,NULL,hInst,NULL);
//Покажемо вікно
ShowWindow(hwnd,SW_NORMAL);
//Цикл обробки повідомлень
while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
return 0;
}
/*Віконна функція головного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg
    WPARAM wParam,LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
        HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
        HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
        default:
            return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
}
/*Функція обробки повідомлень WM_PAINT*/
void OnPaint(HWND hwnd)
{
    PAINTSTRUCT ps;
    char szText[]="Процес-приймач";
    HDC hdc=BeginPaint(hwnd &ps);
    TextOut(hdc,5,10,szText,strlen(szText));
    EndPaint(hwnd,&ps);
}

```

```

/*Функція обробки повідомлень WM_COMMAND від пунктів
меню*/
void OnCommand(HWND hwnd,int id,HWND,UINT)
{
    switch(id) //Ідентифікатор пункту меню
    {
        case MI_ABOUT: //Відкрити діалог
            DialogBox(hInstance,"About",hwnd,DlgProc);
        break;
        case MI_EXIT: //Завершення процесу
            DestroyWindow(hwnd);
        }
    }
/*Функція обробки повідомлень WM_DESTROY*/
void OnDestroy(HWND)
{
    PostQuitMessage(0);
}
/*Віконна функція діалога*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg
LPARAM wParam,LPARAM lParam)
{
    switch(msg)
    {
        {
            HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
            HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
            default:
                return FALSE;
        }
    }
}
/*Функція обробки повідомлення WM_INITDIALOG для
діалогового вікна*/
BOOL DlgOnInitDialog(HWND,HWND,LPARAM)
{
    return TRUE;
}
/*Функція обробки повідомлень WM_COMMAND для
діалогового вікна*/

```

```

voidDlgOnCommand(HWND hwnd,int id,HWND,UINT)
{
    switch(id) //Код елемента управління (кнопки)
    {
        case IDOK: //Натиснута кнопка "Закрити"
        case IDCANCEL: //Надана команда закрити діалог через його
системне меню
            EndDialog(hwnd,0); //Закриття діалогового вікна
    }
}

```

Файл 126.rc

```

#include "2.h"
/*Сценарій меню*/
Main MENU{
    POPUP "Файл" {
        MENUITEM "Про програму", MI_ABOUT
        MENUITEM "Вихід", mi_exit
    }
}
/*Сценарій діалога*/
About DIALOG 65,0,93,40
STYLE
WS_POPUP|WS_CAPTION|WS_SYSMENU|DS_MODALFRAME
CAPTION "Про програму" {
    TEXT "Демонстрація зв'язку процесів"-1, 12, 4, 72, 25
    PUSHBUTTON "Закрити",idok,27,24,43,12
}

```

Файл 2.h

```

#include <windows.h>
#define MI_ABOUT    100 //Ідентифікатори пунктів меню
#define MI_EXIT    101
LRESULT CALLBACK WndProc(HWND,UINT
WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnPaint(HWND);
void OnDestroy(HWND);

```

```

BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
BOOL OnCopyData(HWND,HWND,COPYDATASTRUCT*);
void DlgOnCommand(HWND,int,HWND,UINT);

```

Завдання до програм 1.2а та 1.2б

Загальне. Додати в елемент меню About додатка-приймальника повідомлення: «Демонстрація зв'язку процесів. Дата і прізвище.»

У додатку-приймальника за допомогою меню відкрити модальний діалог з інформацією про програму джерела.

Перший варіант

Розробіть програмний комплекс, в якому батьківський процес-відправник готує масив даних і передає його запущеному процесу-приймальнику, який запише масив на диск. Для передачі даних скористайтеся повідомленням WM_COPYDATA.

Другий варіант

Розробіть програму управління деяким вимірювальним устаткуванням, наприклад медичного характеру. Для цього створити в рамках процесу, крім первинного потоку, ще два породжених, закріпивши за цими потоками наступні функції:

- первинний потік – вивід в головне вікно інформації про поточний час.
- породжений потік 1 – подача звукових сигналів кожні 5 с.
- породжений потік 2 – перегляд файлів даних, що відносяться до вимірювань, що проводяться, і вивід на екран у формі графіка.

2.3 Зміст письмового звіту

1. Текст віконної функції після модернізації програми.
2. Тексти файлу ресурсів .rc і заголовного файлу .h
3. Короткі відповіді на контрольні запитання.

2.4 Контрольні запитання

1. Дайте визначення процесу і потоку.
2. Напишіть алгоритм програми, яка запускає іншу програму як дочірній процес. Перерахуйте функції, за допомогою яких можливо запустити дочірні процеси.
3. Опишіть параметри і призначення функції CreateProcess().

4. Напишіть програму, яка формує масив даних, наприклад масив, що складається з букв латинського алфавіту, і створює процес (використовувати функцію `CreateProcess`) з ім'ям здійснюваного файлу `notepad.exe`. Як параметр командного рядка йому передається специфікація файлу `1.txt`.

5. Опишіть параметри і призначення функції `CreateThread` ().

6. Напишіть алгоритм створення у рамках процесу деякої кількості однорідних потоків. Створювані потоки виводять у власному вікні свій номер і час свого запуску.

7. Яка функція здійснює пересилку повідомлення в інший процес? Опишіть параметри цієї функції.

8. Як визначити дескриптори вікон інших процесів?

9. Зіставте параметри повідомлення `WM_COMMAND` з параметрами `Send| Message|` і поясніть обробку команди «Вибір пункту меню» (`case| MI_LINK2|`).

3 ЛАБОРАТОРНА РОБОТА № 5

ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ. СТРУКТУРОВАНА ОБРОБКА ВИНЯТКІВ.

Мета роботи - познайомитися з принципами обробки виняткових ситуацій та з структурованою обробкою винятків.

3.1 Загальні відомості

Виняткові ситуації (exception handling) являють собою події при виконанні програми, які призводять до її ненормального або неправильного поводження, помилок часу виконання. Існує два види винятків: апаратні, які генеруються процесором, і програмні, які генеруються операційною системою і прикладними програмами.

У мові C++ передбачено три ключових слова для обробки виняткових ситуацій: `try` (контролювати), `catch` (ловити) і `throw` (генерувати, породжувати, кидати, посилати, формувати). Код, здатний згенерувати виключення, повинен виконуватись всередині фігурних дужок, які мають стояти за ключовим словом `try`.

Якщо блок `try` виявляє виключення усередині цього блоку коду, відбувається програмне переривання та виконується зазначена нижче послідовність дій:

- програма шукає відповідний оброблювач виключення;
- якщо оброблювач знайдено, стек очищується і керування передається оброблювачеві виключення;
- якщо оброблювач не знайдений, викликається функція `terminate()` для завершення програми.

Виняткова ситуація обробляється всередині блоку `catch`, що повинен стояти безпосередньо за блоком `try`. Для кожного виключення необхідно передбачити свій оброблювач, інакше може виникнути ненормальне завершення програми.

Блоки `try` і `catch` мають наступні формати:

```
try{  
    // Код, що генерує виключення  
}  
catch (Type X){
```

```

// Оброблювач виключення X типу Type, що могло бути
// раніше згенеровано всередині попереднього блоку try
}
// Оброблювачі інших виключень попереднього блоку try
catch (...){
// Оброблювач будь-якого виключення попереднього блоку try
}

```

throw <виключення>;

Виконання цієї інструкції повинне відбуватися або всередині блоку try, або в функції, яка викликана з блоку try.

Приклад 1:

```

int main()
{
try
{
int n;
cin >> n;
int* myarray = new int[n*1000];
delete[] myarray;
}
catch (exception& e) //Takes a reference to an 'exception' object
{
cout << "Error allocating memory: " << e.what() << endl;
}
return 0;
}

```

Структурована обробка винятків (англ. SEH - Structured Exception Handling) - механізм обробки програмних і апаратних винятків в операційній системі Microsoft Windows, що дозволяє програмістам контролювати обробку винятків, а також є налагоджувальний засобом. Механізм структурної обробки виключень дозволяє однотипно обробляти як програмні, так і апаратні виключення.

Механізм підтримується Microsoft тільки на рівні компілятора за допомогою реалізації нестандартних синтаксичних конструкцій `__try`, `__except` і `__finally`. Ключове слово `__try` використовується для виділення ділянки коду, в якому генерація виключення буде оброблена одним або декількома блоками `__except`. Код, що знаходиться в блоці `__finally`, виконається завжди і незалежно від інших блоків `__try` і `__except`.

Приклад 2, використання в мовах C і C++:

```
__try {
    // захищений код,
    // який поміщається в SEH-фрейм
}
__except (фільтр виключень) {
    // обробник виключень
}
__finally {
    // код, що виконується в будь-якому випадку
}
```

В якості фільтра винятків можуть виступати звичайні функції, які повертають три константні вирази:

– **EXCEPTION_EXECUTE_HANDLER** - вказує на можливість даного обробника обробити виняток. При отриманні такого значення операційна система припиняє пошук релевантних оброблювачів виключення і, виконавши розкрутку стека, передає управління першому, який повернув значення **EXCEPTION_EXECUTE_HANDLER**;

– **EXCEPTION_CONTINUE_EXECUTION** - вказує на виправлення помилки. Система знову передасть управління на інструкцію, яка викликала виняток, оскільки передбачається, що в цей раз вона не викличе виключення;

– **EXCEPTION_CONTINUE_SEARCH** - вказує, що відповідний обробник може бути знайдений вище по стеку. У той же час повернення цього значення може бути свідченням того, що помилка не оброблена.

Кожен потік будь-якого процесу використовує регістр (16-розрядний селектор) `fs` для зберігання покажчика на структуру даних Thread Information Block, що містить інформацію про цей потік. У цій структурі зберігається покажчик на останню з пов'язаного списку зареєстровану структуру `_EXCEPTION_REGISTRATION_RECORD`, що включає покажчик на обробник виключення і покажчик на попередній запис `_EXCEPTION_REGISTRATION_RECORD`. При створенні потоку операційна система додає обробник виключення за замовчуванням, що викликається функцією `kernel32! UnhandledExceptionFilter`.

Прототип callback функції-обробника наступний:

```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
);
```

Кожен раз, коли програміст використовує конструкцію `__try`, відбувається додавання нового екземпляра структури `_EXCEPTION_REGISTRATION_RECORD`, що вказує на функцію `_except_handler3` бібліотеки `msvcrt.dll`, в стек потоку. Код, що полягає в блоках `__except` і `__finally` викликається з `_except_handler3`. В кінці блоку `__try` компілятор додає код, який видаляє поточний запис `_EXCEPTION_REGISTRATION_RECORD` і відновлює значення покажчика `fs: 0` на попередній запис.

Коли відбувається виключення, система послідовно перебирає весь ланцюжок обробників переривань. Кожен обробник повертає значення, яке вказує на те, чи може він обробити цей виняток чи ні. Покажчиком кінця списку доступних обробників виключення є значення `FFFFFFFF`, яка розташовується в стеці за останнім оброблювачем. Якщо система знаходить потрібний обробник, то управління передається йому. При цьому, після знаходження релевантного обробника виключення, операційна система не відразу передає йому управління, а ще раз послідовно викликає все обробники по ланцюжку з ознакою `EH_UNWINDING` для проведення очищення

(виклику деструктора). Якщо жоден з встановлених програмістом фільтрів оброблювачів винятків не повернув `EXCEPTION_EXECUTE_HANDLER` або `EXCEPTION_CONTINUE_EXECUTION`, то відбувається виконання `UnhandledExceptionFilter` - фільтра обробника винятків за замовчуванням, який реєструється при підготовці потоку до запуску.

При виникненні виключення операційна система не викликає безпосередньо фільтр виключень (який відповідає за те, чи буде конкретний обробник обробляти це виключення чи ні), а передає його адресу функції `_except_handler3`, звідки і викликається функція-фільтр. Вона використовує наступну структуру даних:

```
struct _EXCEPTION_REGISTRATION{
    struct _EXCEPTION_REGISTRATION *prev;
    void (*handler)(PEXCEPTION_RECORD,
                    PEXCEPTION_REGISTRATION,
                    PCONTEXT,
                    PEXCEPTION_RECORD);
    struct scopetable_entry *scopetable;
    int trylevel;
    int _ebp;
    PEXCEPTION_POINTERS xpointers;
};
```

Поле `* scopetable` вказує на адресу масиву структур `scopetable_entry`, а цілочисленне поле `trylevel` - індекс в цьому масиві. Поле `_ebp` містить значення покажчика кадру стека, яке існувало до створення структури `EXCEPTION_REGISTRATION`. Функція `_except_handler3` викликає потрібний фільтр і до виклику обробника робить розкруктку (очищення) стека функцією `ntdll.dll! RtlUnwind`.

Якщо жоден з встановлених програмістом оброблювачів не погодився обробити виняток, то викликається функція `UnhandledExceptionFilter`, яка перевіряє, чи запущений процес під відладником, і інформує його, якщо він доступний. Після цього функція викликає фільтр замовчуваного обробника (який встановлюється функцією `SetUnhandledExceptionFilter` і який завжди повертає `EXCEPTION_EXECUTE_HANDLER`). Потім, в залежності від налаштувань операційної системи, викликається або відладчик,

або функція `NtRaiseHardError`, яка відображає повідомлення про помилку.

3.2 Завдання до лабораторної роботи № 5

Програма 3.1 - Обробка виключення в разі ділення на 0 та взяття кореня з негативного числа

```
#include <math.h>
#include <iostream>
using namespace std;

int main()
{
    // обробка виразу sqrt(a)/sqrt(b)
    double a, b, c;
    cout << "a = ";
    cin >> a;

    cout << "b = ";
    cin >> b;

    try { // початок блока try
        if (b == 0)        throw 1;
        if (b < 0)         throw 2;
        if (a < 0)         throw 2;

        c = sqrt(a) / sqrt(b);
        cout << "c = " << c << endl;
    }
    catch (int e) // перехоплення помилки
    {
        if (e == 1)
            cout << "Division by 0." << endl;
        if (e == 2)
            cout << "Negative root." << endl;
    }
    return 0;
}
```

Програма 3.2 - Обробка виключення у разі виходу за межі змінної типу string

```
#include <iostream>
using namespace std;

int main()
{
    int length;
    cout << "length = ";
    cin >> length;

    try
    {
        string s = "Success!\n";
        for (int i=0; i<length; ++i)
            cout << s.at(i);
        cout << endl;
        return 0;
    }
    catch (...)
    {
        cerr << "Unknown exception!\n";
        return 1;
    }
}
```

Завдання 3.1

Реалізувати програми 3.1 та 3.2, пояснити результати їх роботи при різних вхідних даних. В чому відмінність обробки помилок у цих програмах.

Завдання 3.2.

Додайте до своєї модифікованої програми 1 з лабораторної роботи №3 обробку помилки після натиснення будь-якої літери (знаку) крім літери згідно варіанту.

Завдання 3.3.

Напишіть програму з обробкою виключення при зверненні в неіснуючу пам'ять за межі масиву по некоректному індексу. Розмір масиву відповідає номеру робочого місця. Індекс для звернення номер робочого місця + 10.

3.3 Зміст письмового звіту

1. Тексти програм з вказаними змінами.
2. Результати роботи програм.
3. Короткі відповіді на контрольні запитання.

3.4 Контрольні запитання

1. Дайте визначення винятковій ситуації.
2. Опишіть блоки try, catch та throw.
3. Дайте визначення структурованої обробки винятків.
4. Поясніть програмні та апаратні виключення.
5. Поясніть синтаксичні конструкції __try, __except і __finally.
6. Що виступає у якості фільтра винятків, та що при цьому може повертатися?
7. Коли відбувається виконання UnhandledExceptionFilter?
8. Для чого потрібна функція SetUnhandledExceptionFilter?

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Мартынов Н.Н. Программирование для Windows на C/C++. В 2-х томах. Том 1. 2-е издание – М.: БИНОМ, 2008.-478с.
2. Мартынов Н.Н. Программирование для Windows на C/C++. В 2-х томах. Том 2. – М.: БИНОМ, 2016.-480с.
3. Щупак Ю.А. Win32 API. Разработка приложений для Windows. – Санкт-Петербург: Питер, 2010.-592с.
4. Побегайло А. П. Системное программирование в Windows. — СПб.: БХВ-Петербург, 2006. — 1056 с.: ил. ISBN 5-94157-792-3
5. Верма Р.Д. Справочник по функциям Win32 API.— М.:Горячая линия – Телеком, 2002.-488с.
6. Финогенов К.Г. Win32. Основы программирования.- М.:ДИАЛОГ-МИФИ. 2002
7. errhandlingapi.h header – [Электронный ресурс] Режим доступа: <https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/>