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 №3

на тему

**УПРАВЛЕНИЕ ПАМЯТЬЮ И ВВОДОМ-ВЫВОДОМ, РАСШИРЕННЫЕ ВОЗМОЖНОСТИ ВВОДА-ВЫВОДА WINDOWS. ФУНКЦИИ API ПОДСИСТЕМЫ ПАМЯТИ WIN 32. ОРГАНИЗАЦИЯ И КОНТРОЛЬ АСИНХРОННЫХ ОПЕРАЦИЙ ВВОДА-ВЫВОДА**

Студент И. В. Бобко

Преподаватель Н. Ю. Гриценко

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 данной работы заключается в исследовании и анализе управления памятью и вводом-выводом в операционной системе *Windows*, а также в рассмотрении расширенных возможностей ввода-вывода. Основное внимание уделяется функциям *API* подсистемы памяти *Win32*, их структуре и принципам работы. Кроме того, цель работы включает в себя рассмотрение организации и контроля асинхронных операций ввода-вывода, что является важной частью эффективной работы приложений в *Windows*. Важным аспектом работы также является анализ отображения файлов в память, что позволяет улучшить производительность и эффективность работы с данными в приложениях под управлением *Windows*.

1. **ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Управление памятью и вводом-выводом (*I/O*) являются критическими аспектами операционных систем, таких как *Windows*. Оптимальное управление памятью позволяет эффективно использовать ресурсы компьютера и обеспечивать быстрый доступ к данным. Ввод-вывод, с другой стороны, отвечает за взаимодействие между приложениями и внешними устройствами, такими как диски, сетевые устройства и многие другие.

Представление файла — это часть виртуального адресного пространства, которую процесс использует для доступа к содержимому файла. Сопоставление файлов позволяет процессу использовать как случайные входные и выходные *(I/O),* так и последовательные операции ввода-вывода. Это также позволяет процессу эффективно работать с большим файлом данных, например с базой данных, без необходимости сопоставлять весь файл с памятью. Несколько процессов также могут использовать сопоставленные с памятью файлы для совместного использования данных. [1]

Обрабатывает чтение из представления файлов и запись в нее с помощью указателей так же, как и с динамически выделенной памятью. Использование сопоставления файлов повышает эффективность, так как файл находится на диске, а представление файла находится в памяти.

Асинхронность означает выполнение задачи или операции, которая не блокирует основной поток выполнения программы. Вместо того, чтобы ждать завершения долгой операции, программа может продолжать выполнение других задач или операций. Асинхронность позволяет увеличить производительность и отзывчивость приложений, так как не блокирует пользовательский интерфейс или другие потоки исполнения во время выполнения задач. [2]

Использование асинхронности с помощью потоков *(thread)* и мьютексов *(mutex)* позволяет эффективно управлять параллельным выполнением задач и предотвращать гонки данных в многопоточных приложениях.

Потоки представляют собой легковесные подпроцессы внутри основного процесса, которые могут выполняться параллельно и многозадачно. Они используются для выполнения асинхронных задач, таких как обработка данных, выполнение вычислений, ввод-вывод и другие операции, которые могут выполняться параллельно.

1. **РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ**

Основой является приложение вывода всех процессов и используемая ими память (рисунок 1)*.*

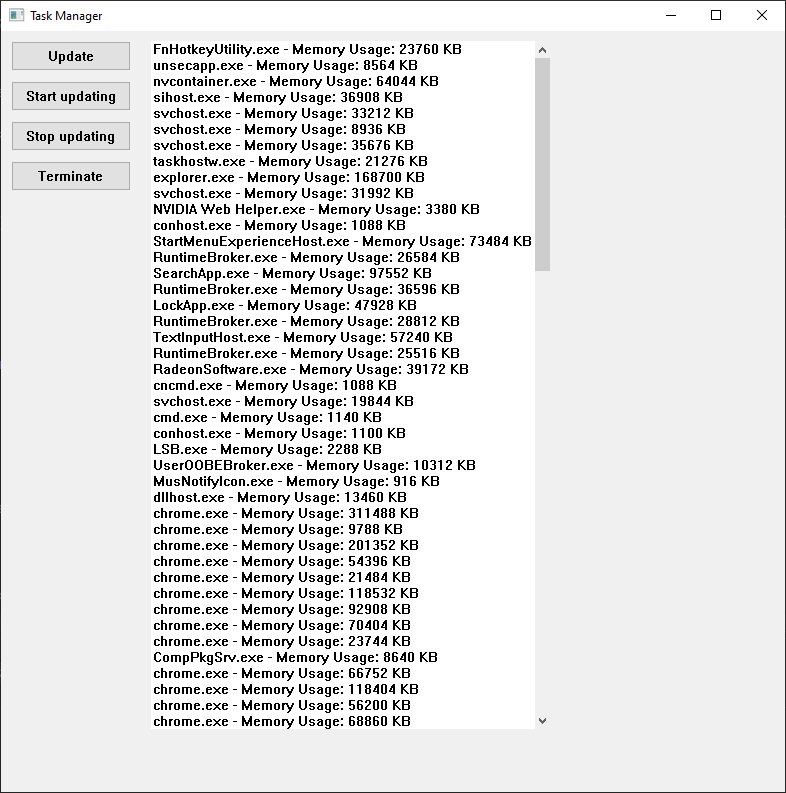


Рисунок 1 – Окно приложения

Была реализована возможность автоматического обновления списка процессов в другом потоке, а также возможность остановки автоматического обновления и включения. Имеется и кнопка ручного обновления.

# ЗАКЛЮЧЕНИЕ

В результате лабораторной работы были изучены основные принципы работы с вводом-выводом: управление памятью, контроль асинхронных операций, отображение файлов в память. Для приложения была добавлена возможность мониторинга всех процессов, а также используемой ими памятью.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Основы программирования для Win32 API [Электронный ресурс]. – Режим доступа: <https://dims.karelia.ru/win32/>.
2. Сопоставление файлов [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/memory/file-mapping>.

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – Файл *Source.cpp*

#pragma comment(linker,"\"/manifestdependency:type='win32' \

name='Microsoft.Windows.Common-Controls' version='6.0.0.0' \

processorArchitecture='\*' publicKeyToken='6595b64144ccf1df' language='\*'\"")

#include <windows.h>

#include <tchar.h>

#include <vector>

#include <string>

#include <tlhelp32.h>

#include <iostream>

#include <windows.h>

#include <tchar.h>

#include <process.h>

#include <Psapi.h>

#define ID\_TERMINATE 1001

#define ID\_UPDATE 1002

#define ID\_STOP\_UPDATING 1003

#define ID\_START\_UPDATING 1004

#define ID\_PROCESS\_LIST 1005

#define UPDATE\_TIME 1000

HANDLE g\_hThread = NULL;

HANDLE g\_hThreadMutex = NULL;

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

void UpdateProcessList(HWND hwnd);

void TerminateProcces(HWND hwnd);

void StartUpdateThread(HWND hwnd);

void StopUpdateThread();

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

MSG msg;

HWND hwnd;

WNDCLASSEX wc;

wc.cbSize = sizeof(WNDCLASSEX);

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

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

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

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

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

wc.hInstance = hInstance;

wc.lpfnWndProc = WndProc;

wc.lpszClassName = L"TaskManager";

wc.lpszMenuName = NULL;

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

if (!RegisterClassEx(&wc))

{

MessageBox(NULL, L"Failed to register window class.", L"Error", MB\_ICONERROR | MB\_OK);

return EXIT\_FAILURE;

}

hwnd = CreateWindow(wc.lpszClassName, L"Task Manager", WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT, CW\_USEDEFAULT, 800, 800, NULL, NULL, wc.hInstance, NULL);

if (hwnd == INVALID\_HANDLE\_VALUE)

{

MessageBox(NULL, L"Failed to create a window.", L"Error", MB\_ICONERROR);

return EXIT\_FAILURE;

}

ShowWindow(hwnd, nCmdShow);

UpdateWindow(hwnd);

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

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return (int)msg.wParam;

}

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

switch (msg)

{

case WM\_CREATE:

{

HWND hUpdateButton = CreateWindow(L"BUTTON", L"Update", WS\_VISIBLE | WS\_CHILD | BS\_PUSHBUTTON, 10, 10, 120, 30, hwnd, (HMENU)ID\_UPDATE, NULL, NULL);

HWND hStartUpdating = CreateWindow(L"BUTTON", L"Start updating", WS\_VISIBLE | WS\_CHILD, 10, 50, 120, 30, hwnd, (HMENU)ID\_START\_UPDATING, NULL, NULL);

HWND hStopUpdating = CreateWindow(L"BUTTON", L"Stop updating", WS\_VISIBLE | WS\_CHILD, 10, 90, 120, 30, hwnd, (HMENU)ID\_STOP\_UPDATING, NULL, NULL);

HWND hTerminateButton = CreateWindow(L"BUTTON", L"Terminate", WS\_VISIBLE | WS\_CHILD, 10, 130, 120, 30, hwnd, (HMENU)ID\_TERMINATE, NULL, NULL);

HWND hProcessList = CreateWindow(L"LISTBOX", L"", WS\_CHILD | WS\_VISIBLE | WS\_VSCROLL | LBS\_DISABLENOSCROLL | LBS\_NOTIFY, 150, 10, 400, 700, hwnd, (HMENU)ID\_PROCESS\_LIST, NULL, NULL);

StartUpdateThread(hwnd);

g\_hThreadMutex = CreateMutex(NULL, FALSE, NULL);

break;

}

case WM\_COMMAND:

{

switch (LOWORD(wParam))

{

case ID\_STOP\_UPDATING:

{

StopUpdateThread();

break;

}

case ID\_START\_UPDATING:

{

StartUpdateThread(hwnd);

break;

}

case ID\_UPDATE:

{

UpdateProcessList(hwnd);

break;

}

case ID\_TERMINATE:

{

TerminateProcces(hwnd);

break;

}

}

break;

}

case WM\_DESTROY:

{

PostQuitMessage(0);

break;

}

default:

{

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

}

}

return 0;

}

DWORD WINAPI UpdateProcessListThread(LPVOID lpParam) {

while (true)

{

UpdateProcessList((HWND)lpParam);

Sleep(UPDATE\_TIME);

}

return 0;

}

void StartUpdateThread(HWND hwnd) {

WaitForSingleObject(g\_hThreadMutex, INFINITE);

if (g\_hThread == NULL)

{

g\_hThread = CreateThread(NULL, 0, UpdateProcessListThread, (LPVOID)hwnd, 0, NULL);

}

ReleaseMutex(g\_hThreadMutex);

}

void StopUpdateThread() {

WaitForSingleObject(g\_hThreadMutex, INFINITE);

if (g\_hThread != NULL) {

TerminateThread(g\_hThread, 0);

CloseHandle(g\_hThread);

g\_hThread = NULL;

}

ReleaseMutex(g\_hThreadMutex);

}

void TerminateProcces(HWND hwnd) {

HWND listBox = GetDlgItem(hwnd, ID\_PROCESS\_LIST);

int selectedIndex = SendMessage(listBox, LB\_GETCURSEL, 0, 0);

if (selectedIndex == LB\_ERR) {

MessageBox(hwnd, L"No process selected.", L"Error", MB\_ICONERROR);

return;

}

TCHAR processName[MAX\_PATH];

SendMessage(listBox, LB\_GETTEXT, selectedIndex, (LPARAM)processName);

TCHAR\* spacePos = \_tcschr(processName, ' ');

if (spacePos != nullptr) {

\*spacePos = '\0';

}

HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (hProcessSnap == INVALID\_HANDLE\_VALUE) {

MessageBox(hwnd, L"Error creating process snapshot.", L"Error", MB\_ICONERROR);

return;

}

PROCESSENTRY32 pe32;

pe32.dwSize = sizeof(PROCESSENTRY32);

DWORD targetPID = 0;

bool processFound = false;

if (Process32First(hProcessSnap, &pe32)) {

do {

if (\_tcscmp(pe32.szExeFile, processName) == 0) {

targetPID = pe32.th32ProcessID;

processFound = true;

break;

}

} while (Process32Next(hProcessSnap, &pe32));

}

CloseHandle(hProcessSnap);

if (!processFound) {

MessageBox(hwnd, L"Process not found.", L"Error", MB\_ICONERROR);

return;

}

HANDLE hProcess = OpenProcess(PROCESS\_TERMINATE, FALSE, targetPID);

if (hProcess == NULL) {

MessageBox(hwnd, L"Error opening process.", L"Error", MB\_ICONERROR);

return;

}

if (TerminateProcess(hProcess, 0)) {

MessageBox(hwnd, L"Process terminated successfully.", L"Success", MB\_ICONINFORMATION);

}

else {

MessageBox(hwnd, L"Error terminating process.", L"Error", MB\_ICONERROR);

}

CloseHandle(hProcess);

}

void UpdateProcessList(HWND hwnd) {

// Очистка listBox

HWND listBox = GetDlgItem(hwnd, ID\_PROCESS\_LIST);

SendMessage(listBox, LB\_RESETCONTENT, 0, 0);

// Создание снимка (TH32CS\_SNAPPROCESS включает в снимок все процессы в системе)

HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);

if (hProcessSnap == INVALID\_HANDLE\_VALUE) {

MessageBox(hwnd, L"Error creating process snapshot.", L"Error", MB\_ICONERROR);

return;

}

PROCESSENTRY32 pe32;

pe32.dwSize = sizeof(PROCESSENTRY32);

if (Process32First(hProcessSnap, &pe32)) {

do {

// Открытие процесса

HANDLE hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, pe32.th32ProcessID);

if (hProcess != NULL) {

PROCESS\_MEMORY\_COUNTERS pmc;

if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc))) {

TCHAR buffer[512];

\_stprintf\_s(buffer, \_T("%s - Memory Usage: %I64u KB"), pe32.szExeFile, pmc.WorkingSetSize / (1024));

SendMessage(listBox, LB\_ADDSTRING, 0, (LPARAM)buffer);

//InvalidateRect(listBox, NULL, TRUE);

}

CloseHandle(hProcess);

}

} while (Process32Next(hProcessSnap, &pe32));

}

CloseHandle(hProcessSnap);

}