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 №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. Отображение файлов в память.

Студент: гр.153502

Сачивко В.Г.

Проверил: Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

1 Цель работы 3

2 Теоретические сведения 4  
3 Результат выполнения программы 5

[Список использованных источников 6](#_TOC_250001)

[Приложение А 7](#_TOC_250000)

**1 ЦЕЛЬ РАБОТЫ**

Целью лабораторной работы является создание приложения для мониторинга и управления системной памятью, отображающее текущее потребление памяти различными процессами.

**2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Управление памятью и вводом-выводом, а также расширенные возможности ввода-вывода в *Windows* предоставляют программистам широкий набор инструментов для эффективного управления ресурсами и выполнения асинхронных операций. В рамках *Win32 API* существует ряд функций и концепций, которые обеспечивают управление памятью и обеспечивают эффективное взаимодействие с вводом-выводом. [1]

Функции *API* подсистемы памяти в *Win32* *API* предоставляют возможности выделения и освобождения памяти. Например, функции *GlobalAlloc* и *LocalAlloc* используются для выделения глобальной и локальной памяти соответственно, а функции *GlobalFree* и *LocalFree* освобождают выделенную ранее память. Управление виртуальной памятью, включая отображение файлов в память с использованием *CreateFileMapping* и *MapViewOfFile*, также является важной частью работы с памятью в *Win32*.

Асинхронные операции ввода-вывода позволяют программам эффективно обрабатывать операции чтения и записи без блокировки основного потока. В *Win32* *API* для этого предоставляются функции *ReadFileEx* и *WriteFileEx*, а также использование структуры *OVERLAPPED* для организации асинхронных операций. Дополнительно, можно использовать *I/O Completion Ports* для управления множеством асинхронных операций эффективным образом. Функции *CreateIoCompletionPort*, *GetQueuedCompletionStatus*, и *PostQueuedCompletionStatus*` предоставляют средства для работы с *I/O Completion Ports*.

Важным аспектом является обработка ошибок ввода-вывода, и для этого используется функция GetLastError для проверки ошибок после завершения асинхронной операции. Многозадачность также может быть обеспечена с использованием I/O Completion Ports, что особенно важно в приложениях с высокой производительностью.

Все эти концепции и функции предоставляют разработчикам инструменты для создания эффективных и отзывчивых приложений под Windows, обеспечивая при этом эффективное управление ресурсами, асинхронные операции ввода-вывода и возможности работы с памятью.

**3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ**

При нажатии на кнопку «Обновить» появляется список процессов в виде таблицы, в которой указаны PID процесса, занимаемая память и название процесса (рисунок 1).

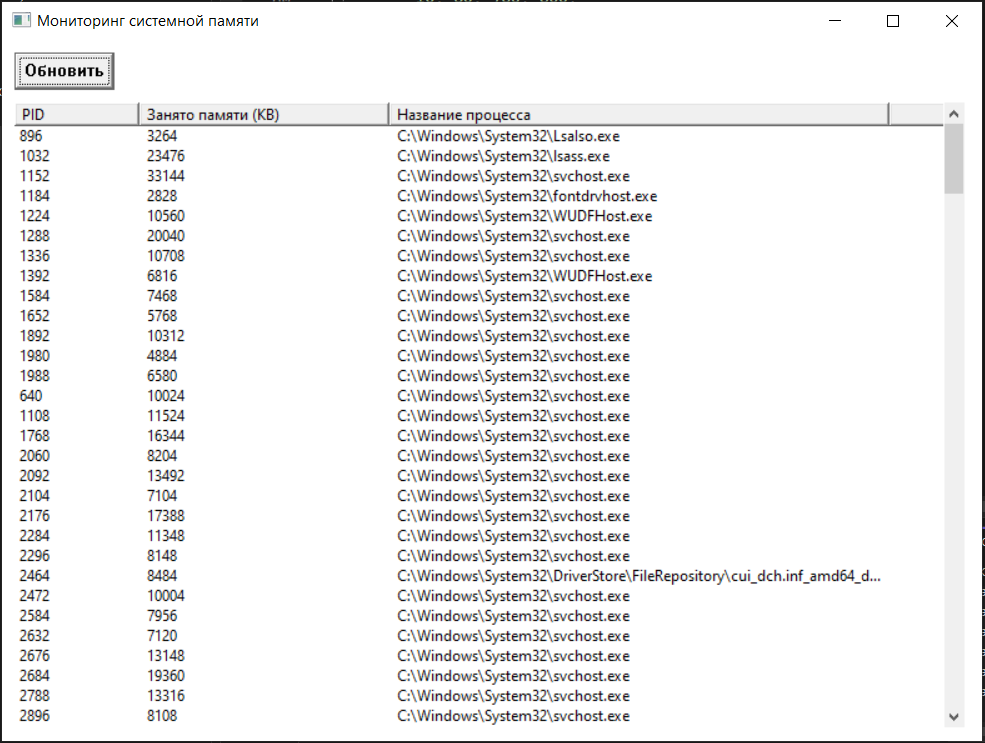


Рисунок 1 – Список процессов

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Функции управления памятью [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/memory/memory-management-functions>

**ПРИЛОЖЕНИЕ А**

**Исходный код программы**

**Файл Lab3.cpp**

#include <windows.h>

#include <psapi.h>

#include <commctrl.h>

#include <iostream>

#include <string>

#include <vector>

#pragma comment(lib, "comctl32.lib")

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

void ListProcesses(HWND hwnd);

void TerminateSelectedProcess(DWORD processId);

std::vector<std::wstring> processListTextLines;

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int nCmdShow) {

INITCOMMONCONTROLSEX iccex;

iccex.dwSize = sizeof(INITCOMMONCONTROLSEX);

iccex.dwICC = ICC\_LISTVIEW\_CLASSES;

InitCommonControlsEx(&iccex);

WNDCLASS wc = {};

wc.lpfnWndProc = WindowProc;

wc.hInstance = hInstance;

wc.lpszClassName = L"MemoryMonitorApp";

RegisterClass(&wc);

HWND hwnd = CreateWindowEx(

0,

L"MemoryMonitorApp",

L"Мониторинг системной памяти",

WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, CW\_USEDEFAULT, 800, 600,

NULL, NULL, hInstance, NULL);

if (hwnd == NULL) {

return 0;

}

CreateWindow(

L"BUTTON",

L"Обновить",

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

10, 10, 80, 30,

hwnd, (HMENU)1, hInstance, NULL);

/\*CreateWindow(

L"BUTTON",

L"Завершить процесс",

WS\_TABSTOP | WS\_VISIBLE | WS\_CHILD | BS\_DEFPUSHBUTTON,

100, 10, 150, 30,

hwnd, (HMENU)3, hInstance, NULL);\*/

HWND hList = CreateWindowEx(

0,

WC\_LISTVIEW,

L"",

WS\_VISIBLE | WS\_CHILD | LVS\_REPORT | LVS\_EDITLABELS,

10, 50, 760, 500,

hwnd,

(HMENU)2,

GetModuleHandle(NULL),

NULL);

DWORD dwStyle = GetWindowLong(hList, GWL\_STYLE);

dwStyle |= LVS\_REPORT;

SetWindowLong(hList, GWL\_STYLE, dwStyle);

LVCOLUMN lvc;

lvc.mask = LVCF\_WIDTH | LVCF\_TEXT | LVCF\_SUBITEM;

lvc.iSubItem = 0;

wchar\_t pidText[] = L"PID";

lvc.pszText = pidText;

lvc.cx = 100;

ListView\_InsertColumn(hList, 0, &lvc);

lvc.iSubItem = 1;

wchar\_t memoryText[] = L"Занято памяти (KB)";

lvc.pszText = memoryText;

lvc.cx = 200;

ListView\_InsertColumn(hList, 1, &lvc);

lvc.iSubItem = 2;

wchar\_t processText[] = L"Название процесса";

lvc.pszText = processText;

lvc.cx = 400;

ListView\_InsertColumn(hList, 2, &lvc);

ShowWindow(hwnd, nCmdShow);

MSG msg;

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

TranslateMessage(&msg);

DispatchMessage(&msg);

}

processListTextLines.clear();

return 0;

}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {

case WM\_CREATE: {

break;

}

case WM\_COMMAND:

if (LOWORD(wParam) == 1) {

ListProcesses(hwnd);

}

else if (LOWORD(wParam) == 3) {

HWND hList = GetDlgItem(hwnd, 2);

int selectedIndex = ListView\_GetNextItem(hList, -1, LVNI\_SELECTED);

if (selectedIndex != -1) {

DWORD processId = std::stoi(processListTextLines[selectedIndex \* 3]);

TerminateSelectedProcess(processId);

//Sleep(1000);

// ListProcesses(hwnd); // Обновить список после завершения процесса

}

}

break;

case WM\_DESTROY:

processListTextLines.clear();

PostQuitMessage(0);

break;

default:

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

}

return 0;

}

void ListProcesses(HWND hwnd) {

HWND hList = GetDlgItem(hwnd, 2);

// Очищаем предыдущий список процессов

ListView\_DeleteAllItems(hList);

std::vector<DWORD> processes(1024);

DWORD bytesNeeded;

if (!EnumProcesses(processes.data(), processes.size() \* sizeof(DWORD), &bytesNeeded)) {

DWORD error = GetLastError();

std::cerr << "EnumProcesses не удалось выполнить с ошибкой: " << error << "\n";

std::cerr << "Требуемый размер буфера: " << bytesNeeded << " байт\n";

return;

}

DWORD processCount = bytesNeeded / sizeof(DWORD);

processes.resize(processCount);

// Очищаем и заново выделяем память для строк

processListTextLines.clear();

size\_t totalStringLength = processCount \* 3; // 3 строки на каждый процесс

processListTextLines.resize(totalStringLength);

LVITEM lvi = { 0 };

lvi.mask = LVIF\_TEXT;

for (DWORD i = 0; i < processCount; ++i) {

HANDLE process = OpenProcess(PROCESS\_QUERY\_LIMITED\_INFORMATION | PROCESS\_VM\_READ, FALSE, processes[i]);

if (process != NULL) {

PROCESS\_MEMORY\_COUNTERS\_EX pmc = {};

if (GetProcessMemoryInfo(process, (PROCESS\_MEMORY\_COUNTERS\*)&pmc, sizeof(pmc))) {

TCHAR szProcessName[MAX\_PATH] = TEXT("<unknown>");

DWORD dwSize = sizeof(szProcessName) / sizeof(TCHAR);

if (QueryFullProcessImageName(process, 0, szProcessName, &dwSize)) {

processListTextLines[i \* 3] = std::to\_wstring(processes[i]);

processListTextLines[i \* 3 + 1] = std::to\_wstring(pmc.WorkingSetSize / 1024);

processListTextLines[i \* 3 + 2] = szProcessName;

lvi.iItem = i;

lvi.iSubItem = 0;

lvi.pszText = const\_cast<LPWSTR>(processListTextLines[i \* 3].c\_str());

int itemIndex = ListView\_InsertItem(hList, &lvi);

lvi.iSubItem = 1;

lvi.pszText = const\_cast<LPWSTR>(processListTextLines[i \* 3 + 1].c\_str());

ListView\_SetItemText(hList, itemIndex, 1, lvi.pszText);

lvi.iSubItem = 2;

lvi.pszText = const\_cast<LPWSTR>(processListTextLines[i \* 3 + 2].c\_str());

ListView\_SetItemText(hList, itemIndex, 2, lvi.pszText);

UpdateWindow(hList);

}

else {

DWORD error = GetLastError();

std::cerr << "QueryFullProcessImageName не удалось выполнить с ошибкой: " << error << "\n";

std::cerr << "ID процесса: " << processes[i] << "\n";

}

}

else {

DWORD error = GetLastError();

std::cerr << "GetProcessMemoryInfo не удалось выполнить с ошибкой: " << error << "\n";

std::cerr << "ID процесса: " << processes[i] << "\n";

}

CloseHandle(process);

}

else {

DWORD error = GetLastError();

std::cerr << "OpenProcess не удалось выполнить с ошибкой: " << error << "\n";

std::cerr << "ID процесса: " << processes[i] << "\n";

}

// Обновим lvi перед следующей вставкой

ZeroMemory(&lvi, sizeof(lvi));

lvi.mask = LVIF\_TEXT;

}

}

void TerminateSelectedProcess(DWORD processId) {

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

if (hProcess != NULL) {

if (TerminateProcess(hProcess, 0)) {

std::wcout << L"Процесс успешно завершен." << std::endl;

}

else {

DWORD error = GetLastError();

std::cerr << "TerminateProcess не удалось выполнить с ошибкой: " << error << std::endl;

}

CloseHandle(hProcess);

}

else {

DWORD error = GetLastError();

std::cerr << "OpenProcess не удалось выполнить с ошибкой: " << error << std::endl;

}

}