Лабораторная работа №3

студента группы ИТ – 32

Курбатовой Софьи Андреевны

Выполнение:_____ Защита _____

ПРОЦЕССЫ И ПОТОКИ В WINDOWS

Цель работы: знакомство с основами управления процессами и потоками, а также изучение механизмов синхронизации потоков. Получение практических навыков использования объектов ядра Windows.

Содержание работы

Вариант 11

1. Разработать в Visual C++ оконное приложение Win32, которое:

– должно выводить в своем окне список всех процессов;

– должно выводить список модулей, загружаемых выбранным процессом;

– должно приостанавливать свою работу, до завершения работы выбранного процесса или до истечения задаваемого времени;

– должно иметь возможность создавать несколько процессов и группировать их в задание;

– должно проверять включено ли оно в какое-либо задание, если да, то выводить список всех процессов, включенных в тоже задание;

– должно иметь возможность изменять свой класс приоритета и относительный приоритет своего главного потока;

2. Разработать в Visual C++ приложение Win32, которое будет формировать матрицу, заполненную случайными числами от 0 до 999, и определять в ней количество двухзначных чисел, а также максимальное и минимальное значения. При этом для каждой строки матрицы следует использовать отдельные потоки. Синхронизация потоков должна выполняться с помощью механизма указанного на рисунке 3.1.

3. Разработать в Visual C++ приложение, которое будет порождать несколько своих экземпляров, а затем ожидать завершения их работы. Все порожденные экземпляры разработанного приложения должны по очереди выполнять счет до 10, после чего начинать считать заново. Работа приложения завершается после того, как оно три раза сосчитает до 10. Синхронизация должна выполняться с помощью объекта ядра, указанного на рисунке 3.1.

4. Протестировать работу разработанных приложений на компьютере под управлением Windows. Результаты отразить в отчете.

5. Исследовать работу разработанных приложений с помощью утилиты Process Explorer. Результаты исследования отразить в отчете. Обязательно необходимо убедиться в том, что отсутствует утечка дескрипторов объектов ядра.

6. Включить в отчет исходный программный код и выводы о проделанной работе.

11,26	Критические секции	Семафор (*дублирование*)

Рис. 3.1. задание для варианта 11

Ход работы

1. Все наименования запущенных процессов и модулей отображаются в виде списка при старте программы (см. Рис 3.2). Используя меню можно выполнить такие действия как: обновление списка процессов, изменение приоритета, ожидание процесса (приложение приостановит свою работу, пока не будет завершен запущенный процесс), завершение процесса. Для того, чтобы выполнить эти действия необходимо выделить строку в списке процессов.

Используя пункт «Задания» можно создать новый процесс. Например, на Рис. 3.6 продемонстрирован запуск файла с расширением .exe. Добавляемые таким образом процессы группируются с отдельное задание и через пункт «Сгруппированные в задание процессы» можно отобразить их в списке. Это продемонстрировано на Рис. 3.7. Строка меню «Процессы в текущем задании» отображает только информацию о запущенном приложении Process.

Приоритет потока устанавливается относительно класса приоритета процесса. Исходя из класса приоритета процесса и относительного приоритета потока операционная система формирует уровень приоритета потока. Ядро Windows всегда запускает тот из потоков, готовых к выполнению, который обладает наивысшим приоритетом. Поток не является готовым к выполнению, если он находится в состоянии ожидания, приостановлен или блокирован по той или иной причине. На случай изменения класса приоритета и относительного приоритета главного потока в приложении предусмотрена соответствующая функция. Для этого необходимо нажать на пункт меню «Изменить приоритет». Пример продемонстрирован на Рис. 3.9.

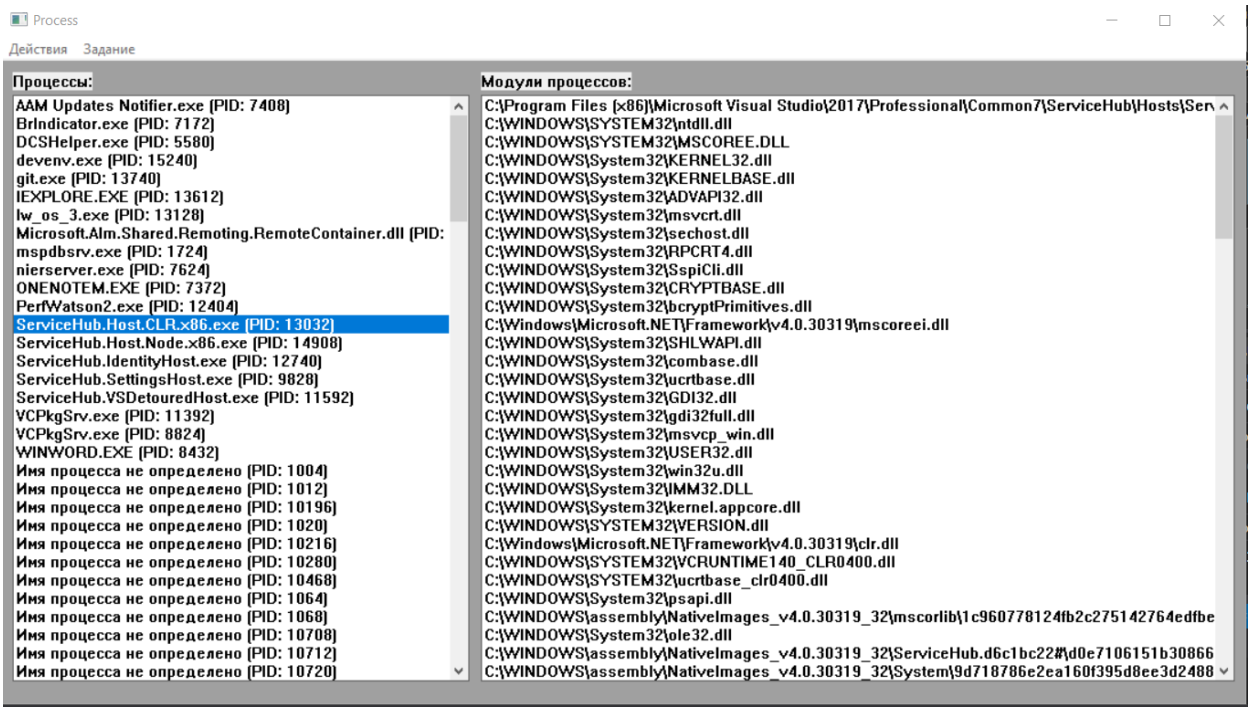

Рис. 3.2. Общий вид приложения Process

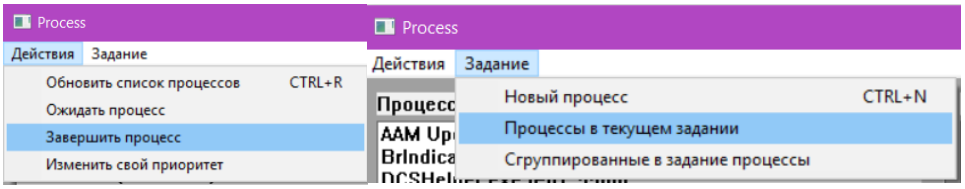

Рис. 3.3. Меню

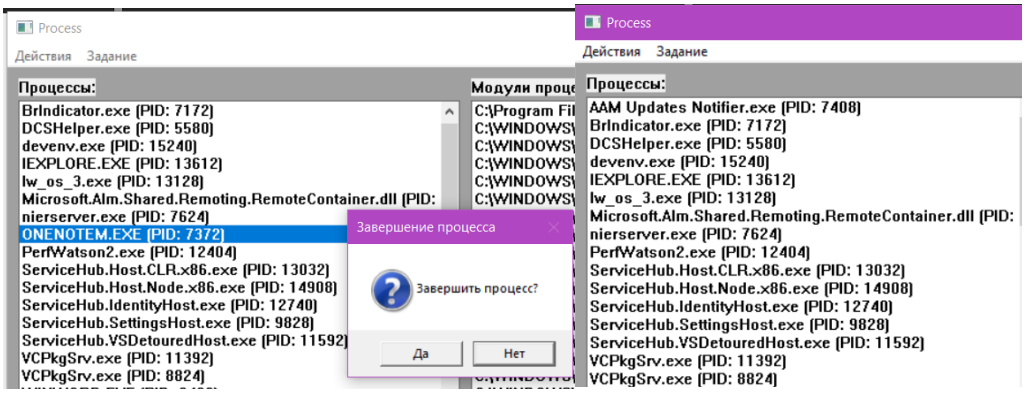

Рис. 3.4. Завершение процесса ONENOTE.EXE

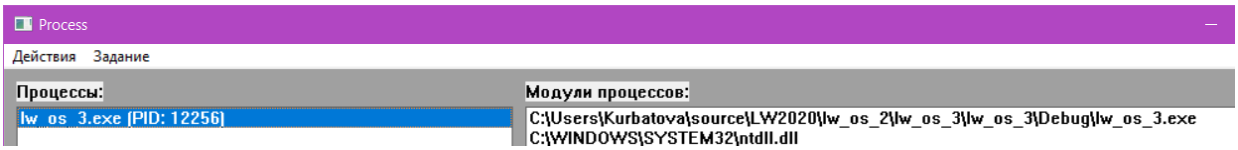

Рис. 3.5. Процессы в текущем задании

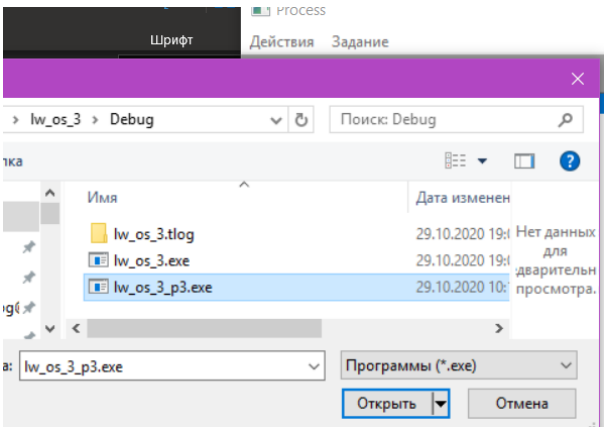

Рис. 3.6. Добавление нового процесса

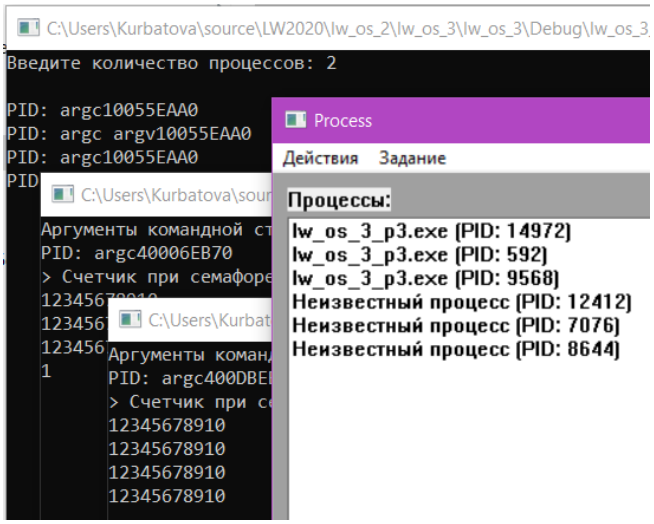

Рис. 3.7. Сгруппированные процессы в новом задании

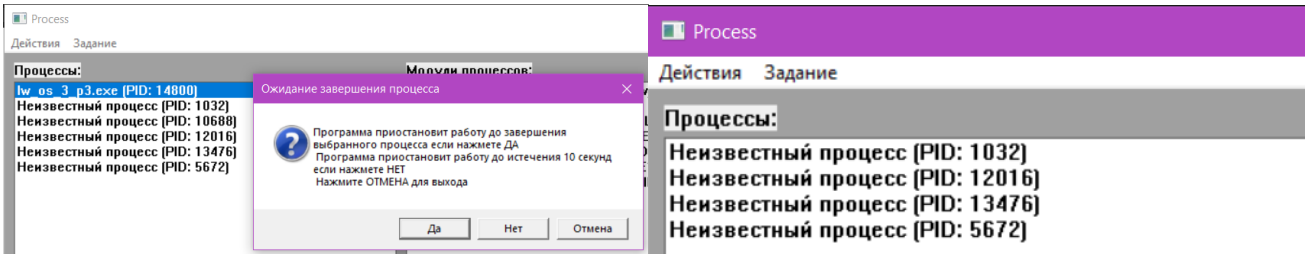

Рис. 3.8. Диалоговое окно ожидания завершения процесса

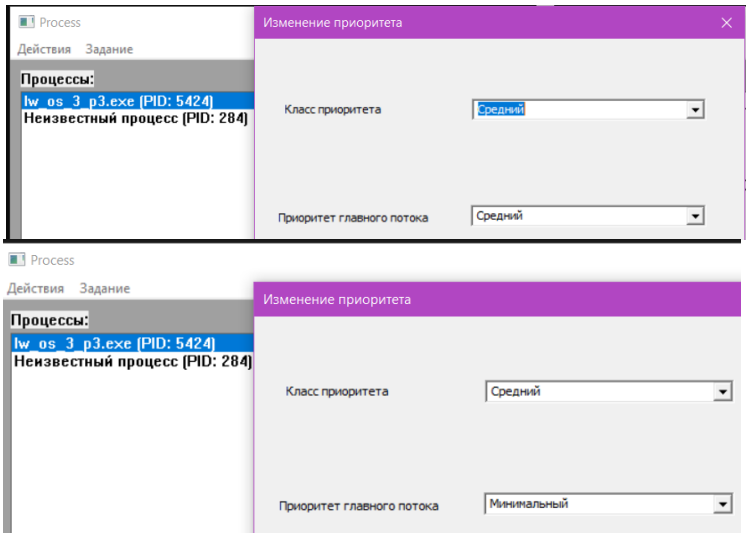

Рис. 3.9. Изменение класса приоритета и относительного приоритета

Листинг 1. Заголовки функций

```
#include <Windows.h>
#include <WindowsX.h>
#include <tchar.h>
#include <strsafe.h>
#include “DialogWork.h”
#include “resource.h”
#include <string>
#include <Psapi.h> // для GetModuleBaseName
#define IDC_LB_PROCESSES        2001
#define IDC_LB_MODULES          2002

HANDLE hJob = NULL; // дескриптор задания
LRESULT CALLBACK MainWindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
/*Обработчики сообщений WM_CREATE WM_DESTROY WM_SIZE WM_COMMAND */
BOOL OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct);
void OnDestroy(HWND hwnd);
void OnSize(HWND hwnd, UINT state, int cx, int cy);
void OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
/*Для ожидания завершения процесса*/
BOOL WaitProcessById(DWORD PID, DWORD WAITTIME, LPDWORD lpExitCode);
/*Диалоговое окно для изменения приоритета процесса*/
INT_PTR CALLBACK DialProc(HWND hwndDlg, UINT uMsg, WPARAM wParam, LPARAM lParam);// процедура
диалогового окна
BOOL Dialog_InitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam);//инициализация диалогового окна
void Dialog_Close(HWND hwnd);//закрытие диалогового окна
void Dialog_Command(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);//если изменить приоритет
/*Загрузка процессов и модулей этих процессов в LISTBOX*/
void ToLB_LoadProcesses(HWND hwndCtl);//процессы в LB
void ToLB_LoadModules(HWND hwndCtl, DWORD PID);//перечисление модулей в LB
void ToLB_LoadProcessesInJob(HWND hwndCtl, HANDLE hJob = NULL);//  процессы в задании
// функция, запускающая группу процессов в одном задании
BOOL StartGroupProcessesAsJob(HANDLE hJob, LPCTSTR lpszCmdLine[], DWORD nCount, BOOL
bInheritHandles, DWORD CreationFlags);
// функция, которая возвращает список идентификаторов включенных в задание процессов
```

```
BOOL EnumProcessesInJob(HANDLE hJob, DWORD* lpPID, DWORD cb, LPDWORD lpcbNeeded);
```

Листинг 2. Работа с главным окном

```
int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE, LPTSTR lpszCmdLine, int nCmdShow)
{
	LoadLibrary(TEXT(«ComCtl32.dll»));//для элементов общего пользования


	WNDCLASSEX wcex = { sizeof(WNDCLASSEX) };

	wcex.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
	wcex.lpfnWndProc = MainWindowProc; // оконная процедура
	wcex.hInstance = hInstance;
	wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION);
	wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
	wcex.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 2);
	wcex.lpszMenuName = MAKEINTRESOURCE(IDR_MENU1);
	wcex.lpszClassName = TEXT(“MainWindowClass”); // имя класса
	wcex.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

	if (0 == RegisterClassEx(&wcex)) // регистрируем класс
	{	return -1; } // завершаем работу приложения

	HACCEL hAccel = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDR_ACCELERATOR1));
	// создаем главное окно на основе нового оконного класса
	HWND hWnd = CreateWindowEx(0, TEXT(“MainWindowClass”), TEXT(“Process”),
WS_OVERLAPPEDWINDOW,
		CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

	if (NULL == hWnd)
	{	return -1; } // завершаем работу приложения

	hJob = CreateJobObject(NULL, TEXT(“FirstJob”));// создаем задание

	ShowWindow(hWnd, nCmdShow); // отображаем главное окно

	MSG  msg;
	BOOL bRet;

	while ((bRet = GetMessage(&msg, NULL, 0, 0)) != FALSE)
	{
		 if (!TranslateAccelerator(hWnd, hAccel, &msg))
		{
			TranslateMessage(&msg);
			DispatchMessage(&msg);
		}
	}

	CloseHandle(hJob);// закрываем дескриптор задания
	return (int)msg.wParam;
}
LRESULT CALLBACK MainWindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
	switch (uMsg)
	{
		HANDLE_MSG(hWnd, WM_CREATE, OnCreate);
		HANDLE_MSG(hWnd, WM_DESTROY, OnDestroy);
		HANDLE_MSG(hWnd, WM_SIZE, OnSize);
		HANDLE_MSG(hWnd, WM_COMMAND, OnCommand);
	}
	return DefWindowProc(hWnd, uMsg, wParam, lParam);	// передача необработанного
сообщения
}
```

Листинг 3. Работа с оконным приложением

```
/*Создание оконного приложения с 2-мя listbox и меню*/
BOOL OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct)
{
	CreateWindowEx(0, TEXT("Static"), TEXT("Процессы:"), WS_CHILD | WS_VISIBLE | SS_SIMPLE,
		10, 10, 400, 20, hwnd, NULL, lpCreateStruct->hInstance, NULL);

	CreateWindowEx(0, TEXT("Static"), TEXT("Модули процессов:"), WS_CHILD | WS_VISIBLE |
SS_SIMPLE,
		420, 10, 400, 20, hwnd, NULL, lpCreateStruct->hInstance, NULL);

	// создаем список для перечисления процессов
	HWND hwndCtl = CreateWindowEx(0, TEXT("ListBox"), TEXT(""), WS_CHILD | WS_VISIBLE |
LBS_STANDARD,
		10, 30, 400, 400, hwnd, (HMENU)IDC_LB_PROCESSES, lpCreateStruct->hInstance, NULL);

	// получаем список процессов активных сейчас
	ToLB_LoadProcesses(hwndCtl);

	// создаем список для перечисления загруженных модулей
	CreateWindowEx(0, TEXT("ListBox"), TEXT(""), WS_CHILD | WS_VISIBLE | WS_HSCROLL |
WS_VSCROLL | WS_BORDER,
		420, 30, 400, 400, hwnd, (HMENU)IDC_LB_MODULES, lpCreateStruct->hInstance, NULL);

	return TRUE;}
/*При завершении работы с приложением*/
void OnDestroy(HWND hwnd)
{
	PostQuitMessage(0); // отправляем сообщение WM_QUIT
}

/*Список меняет ширину и высоту*/
void OnSize(HWND hwnd, UINT state, int cx, int cy)
{
	if (state != SIZE_MINIMIZED)
	{
		// изменяем высоту списка для перечисления процессов
		MoveWindow(GetDlgItem(hwnd, IDC_LB_PROCESSES), 10, 30, 400, cy - 40, TRUE);

		// изменяем высоту списка для перечисления загруженных модулей
		MoveWindow(GetDlgItem(hwnd, IDC_LB_MODULES), 420, 30, cx - 430, cy - 40, TRUE);
	} // if
} // OnSize

void OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)
{
	switch (id)
	{
	case IDC_LB_PROCESSES:
	{
		/*При выборе строки из списка процессов, для процесса определяются модули*/
		if (LBN_SELCHANGE == codeNotify) // выбран другой элемент в списке процессов
		{
			int iItem = ListBox_GetCurSel(hwndCtl);//выделенная строка

		if (iItem != -1)
			{
			UINT PID = (UINT)ListBox_GetItemData(hwndCtl, iItem);// определение ID
процесса
			ToLB_LoadModules(GetDlgItem(hwnd, IDC_LB_MODULES), PID);// получаем список
загруженных модулей
			}
		}
	}
		break;
```

```
	case ID_RELOADPROCESS: // Обновление списка процессов
	{
		HWND hwndList = GetDlgItem(hwnd, IDC_LB_PROCESSES);

		ToLB_LoadProcesses(hwndList);// получаем список процессов

		ListBox_ResetContent(GetDlgItem(hwnd, IDC_LB_MODULES));// очистим список модулей
	}
	break;
	/*Программа приостановит свою работу пока не завершит процесс.
	Если завершение не произойдет по истечению заданного времени, то вернет сообщение об ошибке*/
	case ID_WAITPROCCESS: // Ожидание процесса
	{
		UINT Milliseconds = INFINITE;

		HWND hwndList = GetDlgItem(hwnd, IDC_LB_PROCESSES);

		// определяем индекс выбранного элемента в списке процессов
		int iItem = ListBox_GetCurSel(hwndList);

		if (iItem != -1)
		{
			TCHAR Text[] = TEXT("Программа приостановит работу до завершения выбранного процесса если нажмете ДА\n Программа приостановит работу до истечения 10 секунд если нажмете НЕТ\n Нажмите ОТМЕНА для выхода");

			int mes = MessageBox(hwnd, Text, TEXT("Ожидание завершения процесса"), MB_ICONQUESTION | MB_YESNOCANCEL | MB_DEFBUTTON1);

			switch (mes)
			{
			case IDYES:
				Milliseconds = INFINITE;
				break;

			case IDNO:
				Milliseconds = 10000;
				break;

			case IDCANCEL:
				iItem = -1;
				break;
			}
		}

		if (iItem != -1)
		{
			UINT PID = (DWORD)ListBox_GetItemData(hwndList, iItem); // определяем идентификатор процесса

			DWORD ExitCode;// ожидаем завершения работы процесса
			BOOL bRet = WaitProcessById(PID, Milliseconds, &ExitCode);// ожидаем завершения работы процесса

			if ((FALSE != bRet) && (STILL_ACTIVE != ExitCode)) // если процесс был завершен
			{
				MessageBox(hwnd, TEXT("Процесс завершен"), TEXT("Ожидание завершения процесса"), MB_ICONINFORMATION | MB_OK);

				/*После заврешения процесса удалить его и его модули из списка*/
				ListBox_DeleteString(hwndList, iItem);
				ListBox_ResetContent(GetDlgItem(hwnd, IDC_LB_MODULES));
			}
			else if (FALSE != bRet) // если истекло время ожидания
```

```
                    {
                        MessageBox(hwnd, TEXT("Истекло время ожидания"), TEXT("Ожидание
завершения процесса"), MB_ICONWARNING | MB_OK);
                    }
                    else
                    {
                        MessageBox(hwnd, TEXT("Возникла ошибка"), NULL, MB_ICONERROR | MB_OK);
                    }
                }
            }
            break;
            /*Завершение без приостановки работы*/
            case ID_ENDPROCESS: // Завершение процесса
            {
                HWND hwndLB = GetDlgItem(hwnd, IDC_LB_PROCESSES);//дескпритор списка с процессами

                int SelItem = ListBox_GetCurSel(hwndLB);//  индекс выбранного элемента

                if (SelItem != -1)
                {
                    int mess = MessageBox(hwnd, TEXT("Завершить процесс?"), TEXT("Завершение
процесса"), MB_ICONQUESTION | MB_YESNO | MB_DEFBUTTON2);
                    if (IDYES != mess) SelItem = -1;
                }

                if (SelItem != -1)
                {

                    UINT PID = (DWORD)ListBox_GetItemData(hwndLB, SelItem);// определяем
идентификатор процесса
                    HANDLE hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, PID);
                    BOOL bRet = FALSE;

                    if (NULL != hProcess)
                    {
                        bRet = TerminateProcess(hProcess, 0); // завершаем процесс
                        CloseHandle(hProcess);
                    }

                    if (FALSE != bRet)
                    {
                        MessageBox(hwnd, TEXT("Процесс завершен"), TEXT("Завершение
процесса"), MB_ICONINFORMATION | MB_OK);
                        /*После завершения процесса удалить его и его модули из списка*/
                        ListBox_DeleteString(hwndLB, SelItem);
                        ListBox_ResetContent(GetDlgItem(hwnd, IDC_LB_MODULES));
                    }
                    else
                    {
                        MessageBox(hwnd, TEXT("Неудалось завершить процесс"), TEXT("Завершение
процесса"), MB_ICONWARNING | MB_OK);
                    }
                }
            }
            break;

            case ID_NEWPROCESS: // Создание новых процессов в задании
            {
                TCHAR szFileName[1024] = TEXT("");
                OPENFILENAME ofn = { sizeof(OPENFILENAME) };

                ofn.hwndOwner = hwnd;
                ofn.hInstance = GetWindowInstance(hwnd);
                ofn.lpstrFilter = TEXT("Программы (*.exe)\0*.exe\0");
                ofn.lpstrFile = szFileName;
                ofn.nMaxFile = _countof(szFileName);
                ofn.lpstrTitle = TEXT("Запустить программу");
```

```
		ofn.Flags = OFN_EXPLORER | OFN_ENABLESIZING | OFN_FILEMUSTEXIST |
OFN_ALLOWMULTISELECT;
		ofn.lpstrDefExt = TEXT("exe");

		if (GetOpenFileName(&ofn) == TRUE)
		{
			BOOL bRet = FALSE;

			//(?Кажется не работает)
			UINT FileCount = 0;// определяем количество выбранных файлов
			LPCTSTR filename = ofn.lpstrFile;
			while ((*filename)!= 0)
			{
				filename += _tcslen(filename) + 1;
				++FileCount;
			}

			if (FileCount-- > 1) // если выбрано несколько файлов
			{
				LPCTSTR lpszName = szFileName + _tcslen(szFileName) + 1;
				LPTSTR *aCmdLine = new LPTSTR[FileCount];// создаём массив строк для
нескольких файлов

				for (UINT i = 0; i < FileCount; ++i)
				{

					aCmdLine[i] = new TCHAR[MAX_PATH];// выделяем память для
командной строки

					StringCchPrintf(aCmdLine[i], MAX_PATH, TEXT("%s\\%s"),
szFileName, lpszName);// формируем командную строку

					lpszName += _tcslen(lpszName) + 1;
				} // for


				bRet = StartGroupProcessesAsJob(hJob, (LPCTSTR *)aCmdLine, FileCount,
FALSE, 0);// создаём группу процессов в одном задании

				/* освобождаем выделенную память*/
				for (UINT i = 0; i < FileCount; ++i)
					delete[] aCmdLine[i];
				delete[] aCmdLine;
			}
			else
			{
				LPCTSTR aCmdLine[1] = { szFileName };

				bRet = StartGroupProcessesAsJob(hJob, aCmdLine, 1, FALSE, 0);//
создаём процессы в одном задании
			}

			if ( FALSE != bRet)
			{
				HWND hwndList = GetDlgItem(hwnd, IDC_LB_PROCESSES);

				ToLB_LoadProcessesInJob(hwndList, hJob);// получаем список процессов в
созданном задании

				ListBox_ResetContent(GetDlgItem(hwnd, IDC_LB_MODULES));// очистим
список модулей
			}
			else
			{
				MessageBox(hwnd, TEXT("Ошибка"), NULL, MB_ICONERROR | MB_OK);
			}
		}
```

```
	}
	break;

	case ID_CURRENT_WORKING_PROCESS : // Процессы в текущем задании
	{
		HWND hwndList = GetDlgItem(hwnd, IDC_LB_PROCESSES);

		ToLB_LoadProcessesInJob(hwndList);// получаем список процессов в текущем задании

		ListBox_ResetContent(GetDlgItem(hwnd, IDC_LB_MODULES));
	}
	break;


	case ID_GROUP_PROCESS_IN_TASK: // Процессы, сгрупированные в задание
	{
		HWND hwndList = GetDlgItem(hwnd, IDC_LB_PROCESSES);


		ToLB_LoadProcessesInJob(hwndList, hJob); // получаем список процессов в созданном
задании

		ListBox_ResetContent(GetDlgItem(hwnd, IDC_LB_MODULES));	// очистим список
модулей
	}
	break;

	case ID_PRIORCHANGE: // Изменение приоритета
	{
		/*Создание диалогового окна для изменения приоритета с использование HANDLE от
приложения*/
		HINSTANCE hInstance = GetWindowInstance(hwnd);// получим дескриптор экземпляра
приложения

		DialogBox(hInstance, MAKEINTRESOURCE(IDD_DIALOG1), hwnd, DialProc);
	}
	break;
	}
}
```

Листинг 4. Диалоговые окна

```
BOOL Dialog_InitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam)
{
	HWND hwndCtl;
	hwndCtl = GetDlgItem(hwnd, IDC_COMBO_PRIOR_CLASS);

	/*Классы приоритета процесса*/
	constexpr LPCTSTR PriorityClassesArr[6] = {
		TEXT("Реального времени"),
		TEXT("Высокий"),
		TEXT("Выше среднего"),
		TEXT("Средний"),
		TEXT("Ниже среднего"),
		TEXT("Низкий")
	};

	constexpr UINT PriorityClasses[6] = {
		REALTIME_PRIORITY_CLASS,
		HIGH_PRIORITY_CLASS,
		ABOVE_NORMAL_PRIORITY_CLASS,
		NORMAL_PRIORITY_CLASS,
		BELOW_NORMAL_PRIORITY_CLASS,
		IDLE_PRIORITY_CLASS
	};

	UINT PriorityClass = GetPriorityClass(GetCurrentProcess());
```

```
	for (int i = 0; i < _countof(PriorityClasses); ++i)
	{
		int iItem = ComboBox_AddString(hwndCtl, PriorityClassesArr[i]);
		ComboBox_SetItemData(hwndCtl, iItem, PriorityClasses[i]);

		if (PriorityClasses[i] == PriorityClass)
		{
			ComboBox_SetCurSel(hwndCtl, iItem);
		}
	}

	hwndCtl = GetDlgItem(hwnd, IDC_COMBO_PRIOR);
	/*Относительные приоритеты потока*/
	constexpr LPCTSTR PrioritiesArr[7] = {
		TEXT("Критичный по времени"),
		TEXT("Максимальный"),
		TEXT("Выше среднего"),
		TEXT("Средний"),
		TEXT("Ниже среднего"),
		TEXT("Минимальный"),
		TEXT("Простаивающий")
	};

	constexpr UINT Priorities[7] = {
		THREAD_PRIORITY_TIME_CRITICAL,
		THREAD_PRIORITY_HIGHEST,
		THREAD_PRIORITY_ABOVE_NORMAL,
		THREAD_PRIORITY_NORMAL,
		THREAD_PRIORITY_BELOW_NORMAL,
		THREAD_PRIORITY_LOWEST,
		THREAD_PRIORITY_IDLE
	};

	UINT Priority = GetThreadPriority(GetCurrentThread());

	for (int i = 0; i < _countof(Priorities); ++i)
	{
		int iItem = ComboBox_AddString(hwndCtl, PrioritiesArr[i]);
		ComboBox_SetItemData(hwndCtl, iItem, Priorities[i]);

		if (Priorities[i] == Priority)
		{
			ComboBox_SetCurSel(hwndCtl, iItem);
		}
	}

	return TRUE;
}


void Dialog_Close(HWND hwnd)
{
	EndDialog(hwnd, IDCLOSE);
}

void Dialog_Command(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)
{
	switch (id)
	{
	case IDOK:
	{
		HWND hwndCtl;
		int SelItem;

		hwndCtl = GetDlgItem(hwnd, IDC_COMBO_PRIOR_CLASS);
```

```
		/*ПОЛУЧЕНИЕ КЛАССА ПРИОРИТЕТА*/

		SelItem = ComboBox_GetCurSel(hwndCtl);
		UINT Priority;
		if (SelItem != -1)
			Priority = (UINT)ComboBox_GetItemData(hwndCtl, SelItem);
		else Priority = THREAD_PRIORITY_NORMAL;
		/*ЗАДАНИЕ КЛАССА ПРИОРИТЕТА*/

		SetPriorityClass(GetCurrentProcess(), PriorityClass);

		/*ИЗМЕНЕНИЕ ОТНОСИТЕЛЬНОГО ПРИОРИТЕТА ДЛЯ ГЛАВНОГО ПОТОКА */

		hwndCtl = GetDlgItem(hwnd, IDC_COMBO_PRIOR);

		/*ПОЛУЧЕНИЕ ОТНОСИТЕЛЬНОГО ПРИОРИТЕТА ДЛЯ ГЛАВНОГО ПОТОКА*/
		iItem = ComboBox_GetCurSel(hwndCtl);
		UINT Priority = (SelItem!= -1) ? (UINT)ComboBox_GetItemData(hwndCtl, SelItem) :
THREAD_PRIORITY_NORMAL;

		/*ЗАДАНИЕ ОТНОСИТЕЛЬНОГО ПРИОРИТЕТА*/
		SetThreadPriority(GetCurrentThread(), Priority);

		EndDialog(hwnd, IDOK);
	}
	break;

	case IDCANCEL:
		EndDialog(hwnd, IDCANCEL);
		break;
	}
}
```

Листинг 5. Функции загрузки в список

```
void ToLB_LoadProcesses(HWND hwndCtl)
{
	ListBox_ResetContent(hwndCtl);//очистка списка

	DWORD PIDarray[1024], cbNeeded = 0;//массив для ID созданных процессов
	BOOL bRet = EnumProcesses(PIDarray, sizeof(PIDarray), &cbNeeded);//получение списка ID
созданных процессоров

	if (FALSE != bRet)
	{
		TCHAR ProcessName[MAX_PATH], szBuffer[300];

		for (DWORD i = 0,
			n = cbNeeded / sizeof(DWORD); i < n; ++i)
		{
			DWORD PID = PIDarray[i], cch = 0;
			if (0 == PID) continue;


			HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, PID);//получение дескриптора процесса по его ID

			if (NULL != hProcess)
			{
				cch = GetModuleBaseName(hProcess, NULL, ProcessName,
_countof(ProcessName));// получаем имя главного модуля процесса

				CloseHandle(hProcess); // закрываем объект ядра
			}

			if (0 == cch)
```

```
                        StringCchCopy(ProcessName, MAX_PATH, TEXT("Имя процесса не
определено"));

                    StringCchPrintf(szBuffer, _countof(szBuffer), TEXT("%s (PID: %u)"),
ProcessName, PID);


                    int iItem = ListBox_AddString(hwndCtl, szBuffer);

                    ListBox_SetItemData(hwndCtl, iItem, PID);//запись в ListBox имени процесса
            }
        }
}

void ToLB_LoadModules(HWND hwndCtl, DWORD PID)
{
        ListBox_ResetContent(hwndCtl);

        HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE, PID);

        if (NULL != hProcess)
        {
                /*определение размера списка модулей*/
                DWORD cMod = 0;
                EnumProcessModulesEx(hProcess, NULL, 0, &cMod, LIST_MODULES_ALL);

                DWORD NcMod = cMod / sizeof(HMODULE);//столько будет модулей

                HMODULE *hModule = new HMODULE[NcMod];//столько потребуется памяти

                // получаем список модулей
                cMod = NcMod * sizeof(HMODULE);
                BOOL bRet = EnumProcessModulesEx(hProcess, hModule, cMod, &cMod, LIST_MODULES_ALL);

                if (FALSE != bRet)
                {
                        TCHAR ModuleName[MAX_PATH];

                        for (DWORD i = 0; i < NcMod; ++i)
                        {
                                bRet = GetModuleFileNameEx(hProcess, hModule[i], ModuleName,
MAX_PATH);
                                if (FALSE != bRet) ListBox_AddString(hwndCtl, ModuleName); //
добавляем в список новую строку
                        }
                }

                /*Освобождение ресурсов*/
                delete[]hModule;
                CloseHandle(hProcess);
        }
}

void ToLB_LoadProcessesInJob(HWND hwndCtl, HANDLE hJob)
{

        ListBox_ResetContent(hwndCtl);

        DWORD PIDarray[1024]; //массив для всех идентификаторов процессов
        DWORD cbNeeded = 0;//размер блока памяти с идентификаторами
        BOOL bRet = EnumProcessesInJob(hJob, PIDarray, sizeof(PIDarray), &cbNeeded);

        if (FALSE != bRet)
        {
                TCHAR ProcessName[MAX_PATH];//имя процесса
                TCHAR szBuffer[300];//строка в которую запишем имя процесса и его номер
```

```
		for (DWORD i = 0,
			n = cbNeeded / sizeof(DWORD); i < n; ++i)
		{
			DWORD PID = PIDarray[i], cch = 0;

			HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, PID);

			if (NULL != hProcess)
			{

				cch = GetModuleBaseName(hProcess, NULL, ProcessName,
_countof(ProcessName));// получаем имя главного модуля процесса
				CloseHandle(hProcess); // закрываем объект ядра
			} // if

			if (0 == cch)
				StringCchCopy(ProcessName, MAX_PATH, TEXT("Неизвестный процесс"));

			StringCchPrintf(szBuffer, _countof(szBuffer), TEXT("%s (PID: %u)"),
ProcessName, PID);// формируем строку для списка

			int iItem = ListBox_AddString(hwndCtl, szBuffer);

			ListBox_SetItemData(hwndCtl, iItem, PID);// сохраняем в новой строке
идентификатор процесса
		}
	}
}
```

Листинг 6. Функции подсчета процессов, группировки и ожидания

```
BOOL EnumProcessesInJob(HANDLE hJob, DWORD* lpPID, DWORD cb, LPDWORD lpcbNeeded)
{

	DWORD nCount = cb / sizeof(ULONG_PTR);//столько max процессов будет в буфере

	if (NULL != lpPID && nCount > 0)
	{
		DWORD jobProcessStructSize = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
sizeof(ULONG_PTR) * 1024;//блок памяти для структуры
		JOBOBJECT_BASIC_PROCESS_ID_LIST* jobProcessIdList =
static_cast<JOBOBJECT_BASIC_PROCESS_ID_LIST*>(malloc(jobProcessStructSize));

		if (NULL != jobProcessIdList)
		{
			jobProcessIdList->NumberOfAssignedProcesses = nCount;// MAX число процессов,
на которое рассчитана выделенная память
			// запрашиваем список идентификаторов процессов
			BOOL bRet = QueryInformationJobObject(hJob, JobObjectBasicProcessIdList,
jobProcessIdList, jobProcessStructSize, NULL);

			if (FALSE != bRet)
			{
				nCount = jobProcessIdList->NumberOfProcessIdsInList;// определяем
количество идетификаторов
				CopyMemory(lpPID, jobProcessIdList->ProcessIdList, nCount *
sizeof(ULONG_PTR));//copies a block of memory from one location to another

				if (NULL != lpcbNeeded)
					*lpcbNeeded = nCount * sizeof(ULONG_PTR);//размер блока памяти
с идентификаторами
			}

			free(jobProcessIdList); // освобождаем память
			return bRet;
		}
```

```
	}

	return FALSE;
}

BOOL StartGroupProcessesAsJob(HANDLE hjob, LPCTSTR lpszCmdLine[], DWORD nCount, BOOL
bInheritHandles, DWORD CreationFlags)
{

	BOOL bInJob = FALSE;
	IsProcessInJob(GetCurrentProcess(), NULL, &bInJob);// определим, включен ли вызывающий
процесс в задание

		JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };//базовые ограничения

		QueryInformationJobObject(NULL, JobObjectBasicLimitInformation, &jobli,
sizeof(jobli), NULL);
		JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
		//процессу запрещено чтение из буфера обмена
		jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_READCLIPBOARD;

		SetInformationJobObject(hjob, JobObjectBasicUIRestrictions, &jobuir,
sizeof(jobuir));

		//ограничения если вызывающий процесс связан с заданием
		//первый флаг - сообщаем что новый процесс может выполнятся вне задания
		//второй флаг - сообщаем, что новый процесс не является частью задания

		DWORD dwLimitMask = JOB_OBJECT_LIMIT_BREAKAWAY_OK |
JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK;

		if ((jobli.LimitFlags & dwLimitMask) == 0)
		{	/* все порожденные процессы   автоматически включаются в задание */
			return FALSE;
		}

	// порождаем процессы...
	STARTUPINFO si = { sizeof(STARTUPINFO) };
	PROCESS_INFORMATION pi = { 0 };
	TCHAR CmdLine[MAX_PATH];

	for (DWORD i = 0; i < nCount; ++i)
	{
		StringCchCopy(CmdLine, MAX_PATH, lpszCmdLine[i]);
		// порождаем новый процесс,
		// приостанавливая работу его главного потока
		BOOL bRet = CreateProcess(NULL, CmdLine, NULL, NULL,
			bInheritHandles, CreationFlags | CREATE_SUSPENDED |
CREATE_BREAKAWAY_FROM_JOB, NULL, NULL, &si, &pi);

		if (FALSE != bRet)
		{
			//Дочерние процессы, порождаемые этим процессом станут частью этого задания
автоматически
			AssignProcessToJobObject(hjob, pi.hProcess);// добавляем новый процесс в
задание

			ResumeThread(pi.hThread);// возобновляем работу потока нового процесса

			CloseHandle(pi.hThread);// закрывает дескриптор потока нового процесса

			CloseHandle(pi.hProcess);// закрывает дескриптор нового процесса
		}
	}
	return TRUE;
}
```

```
BOOL WaitProcessById(DWORD PID, DWORD WAITTIME, LPDWORD lpExitCode)
{	HANDLE hProcess = OpenProcess(SYNCHRONIZE | PROCESS_QUERY_INFORMATION, FALSE, PID);//
открываем процесс

	if (NULL == hProcess)
	{
		return FALSE;
	}

	WaitForSingleObject(hProcess, WAITTIME);// ожидаем завершения процесса

	if (NULL != lpExitCode)
	{	GetExitCodeProcess(hProcess, lpExitCode);}// получим код завершения процесса

	CloseHandle(hProcess);// закрываем дескриптор процесса

	return TRUE;}
```

2. Разработанное консольное приложение формирует матрицу с размерами, определяемыми пользователем при старте программы. При этом минимальное количество столбцов 2 и строк 2. Соответственно если данное условие не будет выполняться, пользователю будет выведено сообщение об этом. Назначение приложения состоит в том, чтобы заполнить матрицу случайными числами от 0 до 999 и определить сколько в ней 2-значных чисел, а также минимальное и максимальное значение. Так как для каждой строки матрицы используются отдельные потоки, то их необходимо синхронизировать. Определенный в содержании работы механизм: Критические секции.

Этот механизм гарантирует, что в один момент времени монопольный доступ к ресурсу у одного потока, и пока он не выйдет за границы секции, остальные потоки будут ждать.

Когда в строке от ее начала и до конца будет найден индекс минимального и максимального значения, то программа перейдет в область критической секции, где будет осуществлено сравнение найденных значений с установленными пределами, т.е. с 0 и 999. После этого программа продолжит выполнение и перейдет к проверке следующей строки. Пример работы программы показан на Рис. 3.10

```
Для завершения работы программы нажмите 0 иначе 1
1
Введите количество строк: 10
Введите количество столбцов : 4
929     521     254     103
146     55      7       273
481     106     987     946
87      257     461     523
833     453     886     189
668     355     677     922
533     480     836     564
763     734     573     805
997     663     381     230
773     479     760     372

Количество двузначных чисел: 2
Максимальный элемент: 997
Минимальный элемент: 7
```

Рис. 3.10. Демонстрация работы программы

Листинг 7. Функция формирующая матрицу

```
void matrix()
{
	do
	{
		std::cout << "Введите количество строк: ";
		std::cin >> rows;
		if (rows < 2)
		{
			std::cout << "Слишком мало!" << "\n";
		}
	} while (rows < 2);
	do
	{
		std::cout << "Введите количество столбцов : ";
		std::cin >> cols;
		if (cols < 2)
		{
			std::cout << "Слишком мало!" << "\n";
		}
	} while (cols < 2);

	/*Создание матрицы с числами от 0 до 999*/
	srand(time(NULL)); // Инициализируем генератор случайных чисел.
	int** array2D = new int*[rows];
	for (int i = 0; i < rows; ++i)
	{
		array2D[i] = new int[cols];
		for (int j = 0; j < cols; ++j)
		{
			array2D[i][j] = rand() % 1000; // генерируем случайное число от 0 до 999;
			std::cout << array2D[i][j] << "\t";
		}
		std::cout << std::endl;
	}
	std::cout << std::endl;

	// создаём массив потоков для строк матрицы
	HANDLE* threads = new HANDLE[rows];
	InitializeCriticalSection(&cs); // инициализируем критическую секцию
	for (int i = 0; i < rows; ++i)
	{
		//_beginthreadex потому что CreateThread and ExitThread не учитывают возникновение
некоторых проблем
		threads[i] = (HANDLE)_beginthreadex(NULL, 0, ThreadFuncWithCriticalSection, (void
*)array2D[i], 0, NULL);
	}

	WaitForMultipleObjects(rows, threads, TRUE, INFINITE);// ожидание завершения всех
созданных потоков

	std::cout << "Количество двузначных чисел: " << total << std::endl
		<< "Максимальный элемент: " << max << std::endl
		<< "Минимальный элемент: " << min << std::endl;

	for (int i = 0; i < rows; ++i)
	{
		delete[] array2D[i];			// удаляем строку матрицы
		CloseHandle(threads[i]);// закрываем дискриптор созданного потока
	}
	delete[] threads;// удаляем массив потоков
	delete[] array2D;// удаляем матрицу
	DeleteCriticalSection(&cs); // удаляем критическую секцию
}
```

Листинг 8. Функция поиска MAX, MIN, 2-х чисел с помощью потоков

```
unsigned __stdcall ThreadFuncWithCriticalSection(void *lpParameter)
{
	const int *row = (const int *)lpParameter; // указатель на строку матрицы

	unsigned int count = 0; // количество двузначных чисел в строке матрицы
	int Max = 0, Min = 0; // индексы максимального и минимального элемента в строке матрицы

	for (int j = 0; j < cols; ++j)
	{
		if (row[j] > row[Max])
		{	Max = j;	}// запомним индекс максимального значения
		else
			if (row[j] < row[Min])
			{	Min = j;	} // запомним индекс минимального значения
		if (row[j] > 9 && row[j] < 100) // двузначное число
		{
			++count;// увеличим на единицу количество двузначных чисел
		}
	}
		EnterCriticalSection(&cs);
		total += count; // увеличиваем количество двузначных чисел
		if (row[Max] > max) max = row[Max]; // определяем максимальный элемент
		if (row[Min] < min) min = row[Min]; // определяем минимальный элемент
		LeaveCriticalSection(&cs); // покидаем критическую секцию
	return 0;
}
```

Листинг 9. Точка входа в программу и глобальные параметры

```
#include <Windows.h>
#include <tchar.h>
#include <stdio.h>
#include <conio.h>
#include <locale.h>
#include <time.h>
#include <process.h>
#include <iostream>
// функция потока для строки матрицы (используются критические секции)
unsigned __stdcall ThreadFuncWithCriticalSection(void *lpParameter);

int rows, cols, value; // строки, строки, значение

long total = 0; // количество двузначных чисел
int max = 0, min = 999; // максимальное и минимальное значения

CRITICAL_SECTION cs; // критическая секция
int _tmain()
{
	int answer;
	setlocale(LC_ALL, "");
	do
	{
		std::cout << "Для завершения работы программы нажмите 0 иначе 1\n";
		std::cin >> answer;
		if (answer == 1)
			matrix();
	} while (answer != 0);
return 0;
}
```

3. Разработанное консольное приложение, показанное на Рис. 3.11, запрашивает у пользователя количество процессов, которое необходимо создать. Созданными процессами являются копии исполняемого приложения, которые выполняют счет до 10, после чего начинают считать заново. Если счет выполнился 3

раза подряд, то экземпляр завершает свою работу. Работа экземпляра, из которого был осуществлен запуск программы завершиться только после того, как завершатся все остальные. Объект ядра для синхронизации процессов: семафор (дублирование).

Семафор предназначен для того, чтобы ограничить максимальное число потоков, одновременно работающих с неким разделяемым ресурсом. Для создания объекта ядра используется функция CreateSemaphore(LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName). Первый передаваемый параметр в этой функции позволяет определить параметры безопасности и наследования, второй - задает начальное, третий - максимальное количество обращений к семафору, четвертый параметр указывает на имя создаваемого объекта ядра. Увеличение значение счетчика семафора с помощью функции ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount, LPLONG lpPreviousCount).

Механизм использования – Дублирование, требует такой функции как BOOL DuplicateHandle(HANDLE hSourceProcessHandle, HANDLE hSourceHandle, HANDLE hTargetProcessHandle, PHANDLE phTargetHandle, DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwOptions). Эта функция берет запись в таблице описателей одного процесса и создает ее копию в таблице другого

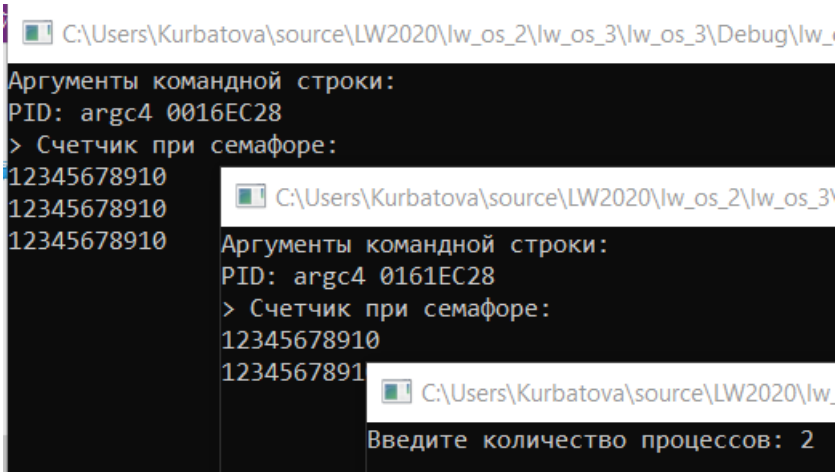

Рис. 3.11. Демонстрация работы приложения

Листинг 10. Исходный код программы для программы создающей саму себя

```
#include <Windows.h>
#include <tchar.h>
#include <strsafe.h>
#include <stdio.h>
#include <conio.h>
#include <locale.h>
#include <iostream>


int _tmain(int argc, LPCTSTR argv[])
{
	setlocale(LC_ALL, "");

	if (1 == argc) // (!) ветка родительского процесса
	{
		int n_processes;
		std::cout << "Введите количество процессов: ";
		std::cin >> n_processes;
		std::cout << std::endl;
```

```
		HANDLE hObject = NULL;

		TCHAR commandline[MAX_PATH + 40];
		LPCTSTR  sinchtype = TEXT("%s semaphore-duplicate %d %p");
		/*дескриптор объекта ядра и идентификатор родительского процесса перадются в
дочерний процесс как аргументы в командной строке*/

		hObject = CreateSemaphore(NULL, 1, 2, TEXT("Semaphor"));

		if (hObject != NULL)
		{
			/*формируем командную строку для создания дочерних процессов*/
			//std::wstring commandline = sinchtype + *argv[0] +
(int)GetCurrentProcessId() + hObject;
			StringCchPrintf(commandline, _countof(commandline), sinchtype, argv[0],
(int)GetCurrentProcessId(), hObject);
		}

		if (hObject != NULL)
		{
			// создаем массив процессов
			HANDLE* ProcArray = new HANDLE[n_processes];

			STARTUPINFO si = { sizeof(STARTUPINFO) };
			PROCESS_INFORMATION procinfo = { 0 };

			for (int i = 0; i < n_processes; ++i)
			{
				// порождаем новый процесс
				BOOL bRet = CreateProcess(NULL, commandline, NULL, NULL, TRUE,
CREATE_NEW_CONSOLE, NULL, NULL, &si, &procinfo);
				std::cout << "PID: argc"<<argc << argv << std::endl;
				if (FALSE != bRet)
				{
					ProcArray[i] = procinfo.hProcess;/*записать дескриптор в массив
и закрыть дескриптор потока*/
					std::cout << "PID: argc argv" << argc << argv << std::endl;
					CloseHandle(procinfo.hThread);
				}
				else
				{
					ProcArray[i] = NULL;
				}
			}

			WaitForMultipleObjects(n_processes, ProcArray, TRUE, INFINITE);// ожидаем
завершения всех дочерних процессов

			for (int i = 0; i < n_processes; ++i)
				CloseHandle(ProcArray[i]);

			delete[] ProcArray;

			CloseHandle(hObject);// закрываем дескриптор
		}
	}
	else if (argc > 1) // ветка дочернего процесса
	{
		std::cout << "Аргументы командной строки:" << std::endl;
		std::cout << "PID: argc" << argc <<" "<< argv[1] << std::endl;
		HANDLE hSemaphore = NULL; // дескриптор семафора
		if ((4 == argc) && (lstrcmpi(argv[1], TEXT("semaphore-duplicate")) == 0))
			{
				UINT parentPID = (UINT)_ttoi(argv[2]);

				HANDLE hObject = (HANDLE)_tcstoui64(argv[3], NULL, 16);//значение
дескриптора семафора в 16-м формате
```

```
                        HANDLE hProcessChild = OpenProcess(PROCESS_DUP_HANDLE, FALSE,
parentPID);// открываем процесс
                        if (NULL != hProcessChild)
                        {
                                DuplicateHandle(hProcessChild, hObject, GetCurrentProcess(),
&hSemaphore, SEMAPHORE_ALL_ACCESS, FALSE, 0);// дублируем полученный дескриптор
                                CloseHandle(hProcessChild);// закрываем дескриптор процесса
                        }
                }

                if (NULL != hSemaphore)
                {
                        std::cout<<"> Счетчик при семафоре:"<<std::endl;

                        for (int i = 0; i < 3; ++i)
                        {
                                WaitForSingleObject(hSemaphore, INFINITE);//жждать пока
освободится семафор
                                int j;
                                for (j = 0; j < 10; ++j)
                                {
                                        std::cout << j + 1;
                                        Sleep(500);   //задержка вывода цифр
                                }
                                if (j == 10) std::cout << std::endl;
                                ReleaseSemaphore(hSemaphore, 1, NULL);// Поток увеличивает
значение счетчика текущего числа ресурсов
                        }

                        CloseHandle(hSemaphore);// закрываем дескриптор семафора
                }
        }
}
```

4. Получение списка процессов и дескрипторов в Windows с помощью утилиты Process Explorer:

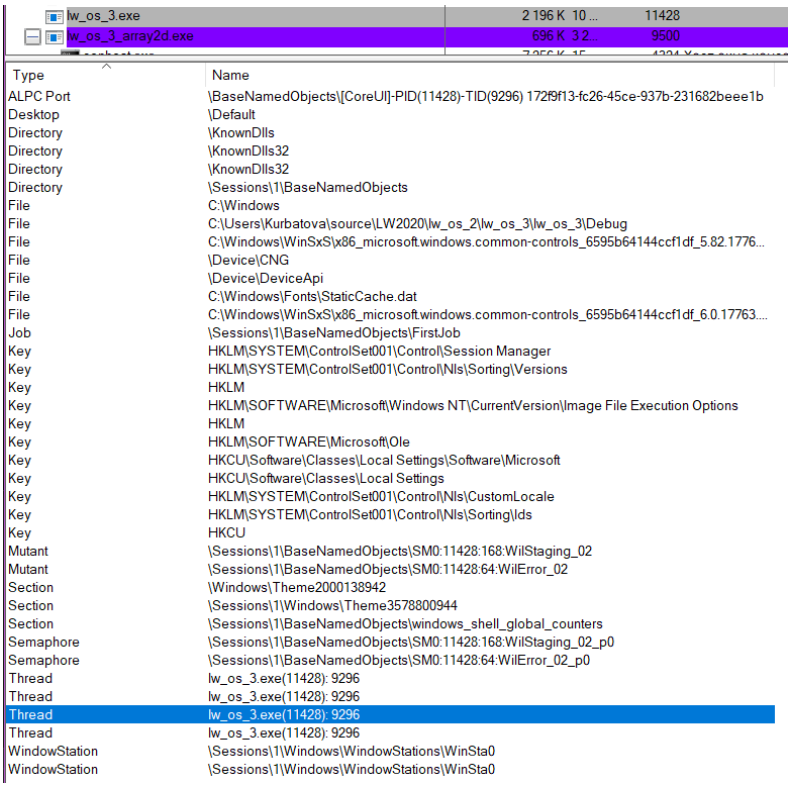

Рис. 3.12. Дескрипторы 1-го приложения

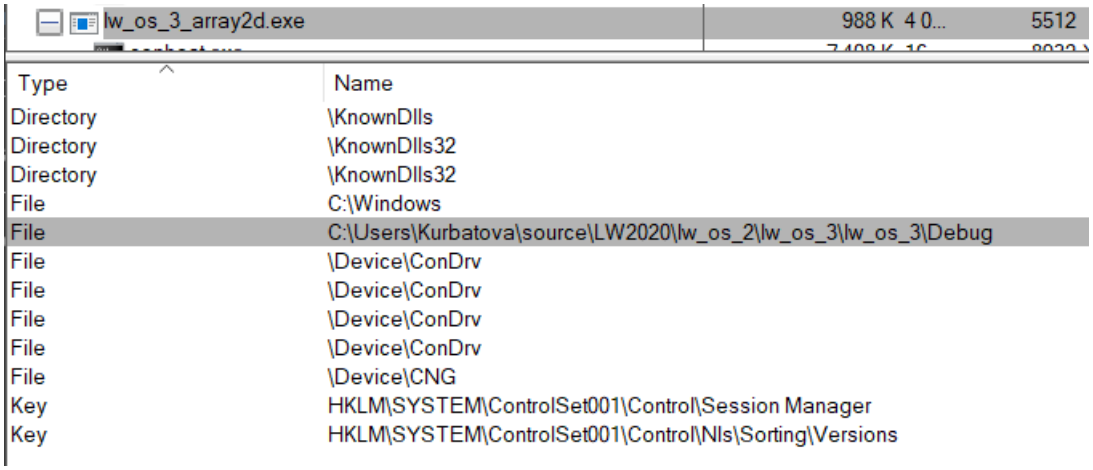

Рис. 3.13. Дескрипторы 2-го приложения

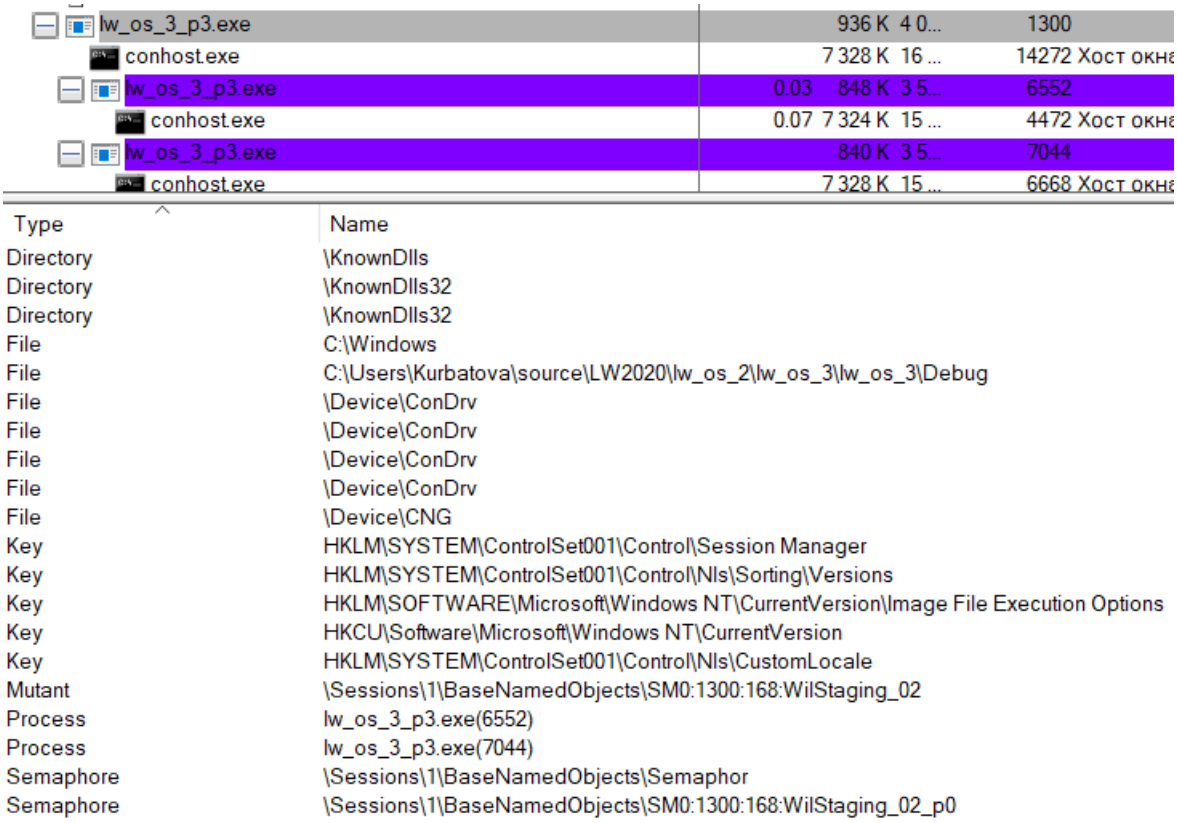

Рис. 3.14. Дескрипторы 3-го приложения в процессе работы

Вывод: Таким образом, в ходе выполнения лабораторной работы было осуществлено создание 3 приложений для получения практических навыков использования объектов ядра Windows. В процессе создания приложений было осуществлено знакомство с основами управления процессами и потоками, а также изучен механизм синхронизации потоков с помощью критических секций и с использованием объектов ядра Семафор.