Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №2 по теме: «Архитектура Windows»

Выполнил:  
студент группы ПВ-31  
Адаменко И. И.

Проверил:  
к. т. н., доцент  
Михелёв В. М.

Белгород  
2014

Цель работы: изучение архитектуры операционной системы Windows.

## Задание

1. Провести исследование ОС с использованием системного монитора.
   1. Определить количество процессов, потоков и дескрипторов в ОС; изменить их число, запуская на выполнение новые приложения.
   2. Определить процент работы в пользовательском режиме, процент работы в привилегированном режиме и процент времени бездействия при выполнении, связанными с интенсивными графическими операциями.
   3. Включить в отчёт полученные графики и привести их объяснение.
2. Составить три программы, которые:
   1. Принимая дескриптор, имя или полное имя модуля, возвращает другие два элемента в своих выходных параметрах. Возможны четыре варианта работы этой функции.
   2. Будет выполнять по шагам следующее:
      1. Используя функцию GetCurrentProcessId определит идентификатор текущего процесса.
      2. Используя функцию GetCurrentProcess определит псевдодескриптор текущего процесса.
      3. Используя функцию DuplicateHandle и значение псевдодескриптора определить дескриптор текущего процесса.
      4. Используя функцию OpenProcess определит копию дескриптора текущего процесса.
      5. Закроет дескриптор, полученный функцией DuplicateHandle.
      6. Закроет дескриптор, полученный функцией OpenProcess.
   3. Выдаёт список перечисления всех процессов, потоков, модулей и их свойства в системе:
      1. Для Windows NT (и список загруженных драйверов устройств).
      2. Для Windows 9x.

## Отчёт

Приложение Windows — это совокупность исполняемых программ (файлов с кодом) и вспомогательных файлов. Процессом называется исполняемый экземпляр приложения и те ресурсы, которые ему отвелись. Чаще всего пользователь может запускать несколько экземпляров одного приложения (например, можно запустить несколько окон браузера). Каждый исполняемый экземпляр в таком случае — это отдельный процесс со своей собственной областью памяти.

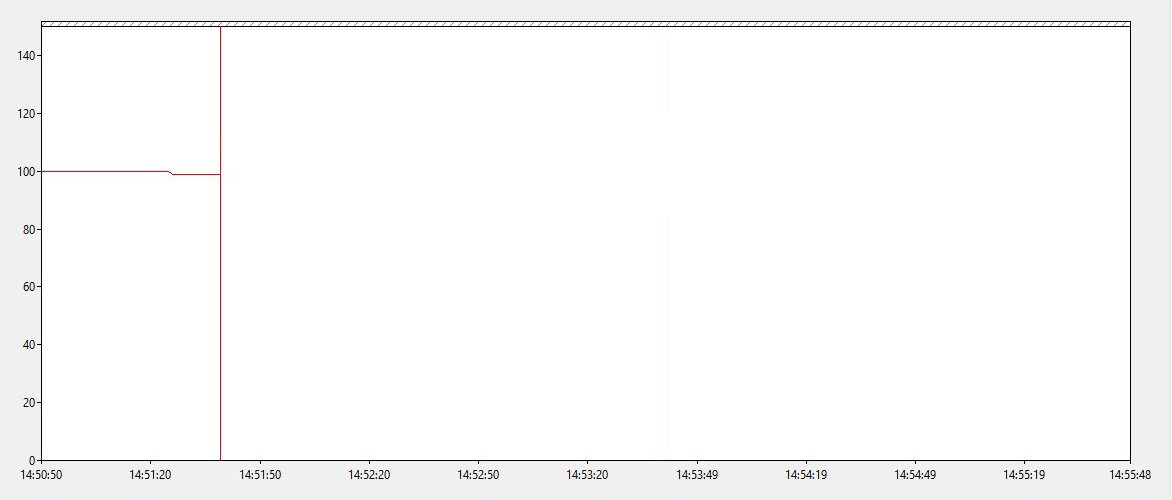
Сам процесс не исполняет код. Этим занимаются потоки — внутренние составляющие процесса, которым операционная система выделяет процессорное время. Потому у каждого процесса есть как минимум один поток.

Поток может выполняться в двух режимах: в пользовательском и в режиме ядра. Однако он не может работать постоянно в режиме ядра. Поток переходит в этот режим только в том случае, когда использует специальные API-функции. Это связано с тем, что режим ядра — это наиболее привилегированный режим, с полным доступом ко всей памяти и командам процессора.

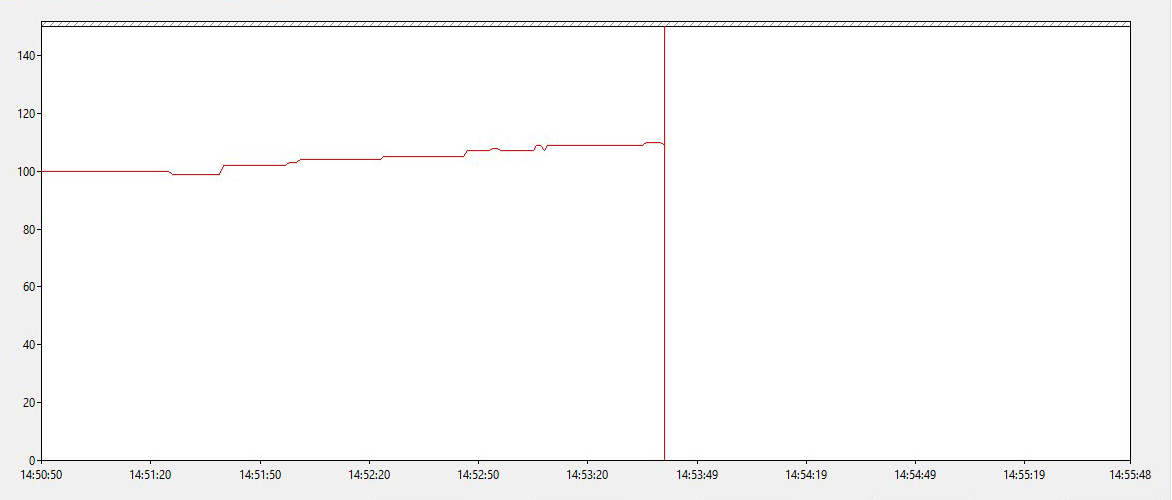
## Исследование ОС

В момент исследования в ОС запущено 100 процессов, 1198 потоков и 35 192 дескриптора.

График, отображающий количество процессов на момент начала исследования:



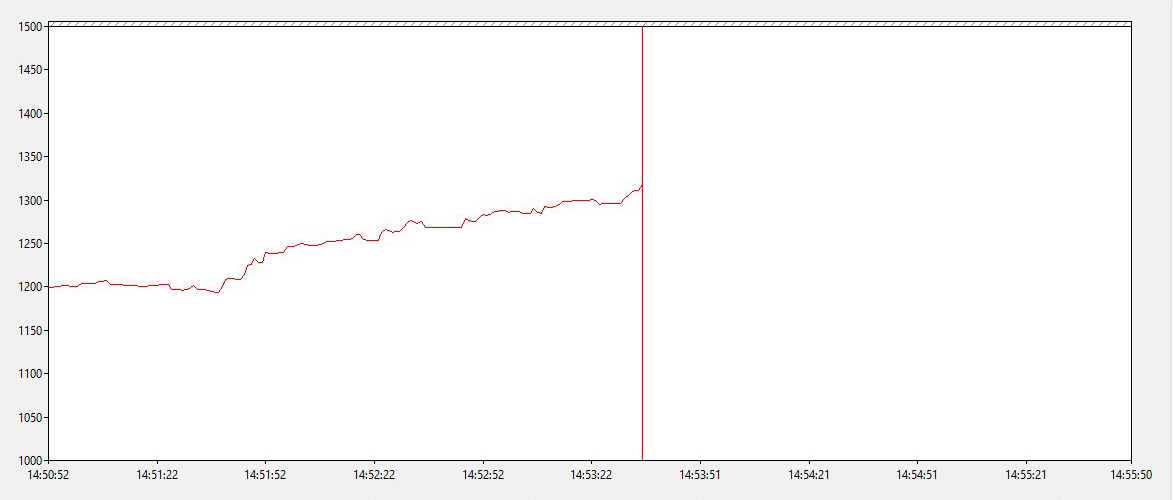
После запуска нескольких приложений явно заметно увеличения количества процессов:



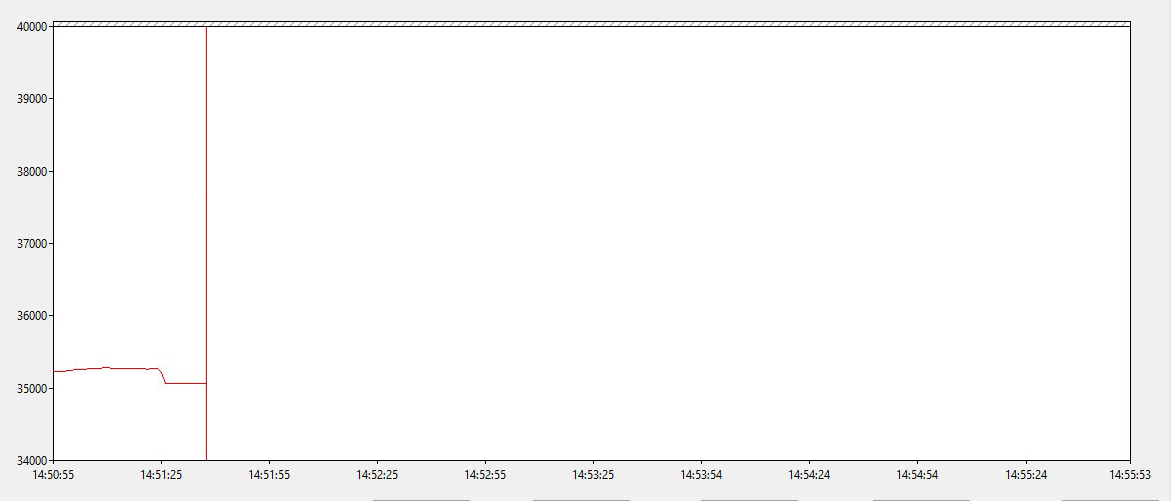
График, отображающий количество потоков на момент начала исследования:



После запуска приложений:



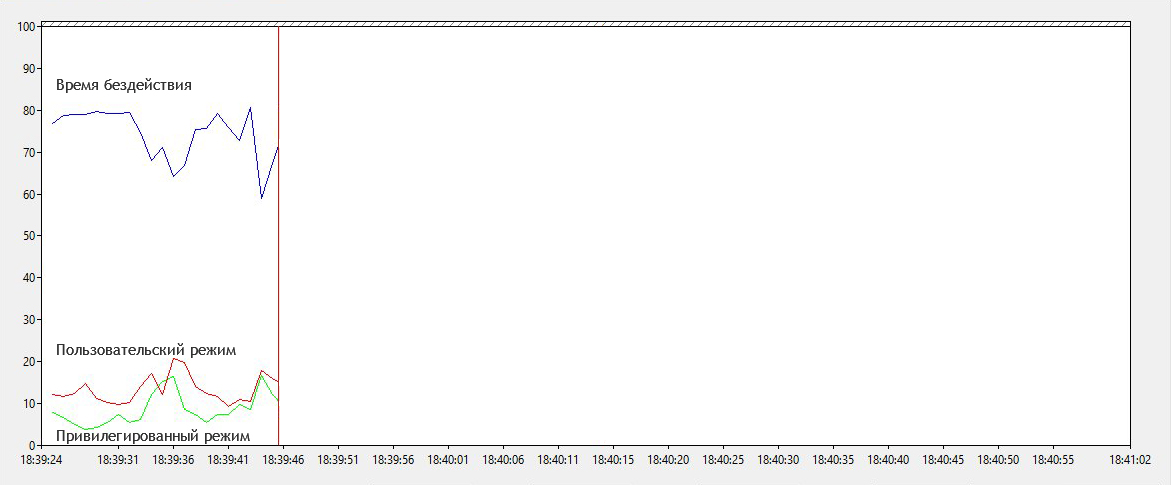
График, отображающий количество дескрипторов на момент начала исследования:



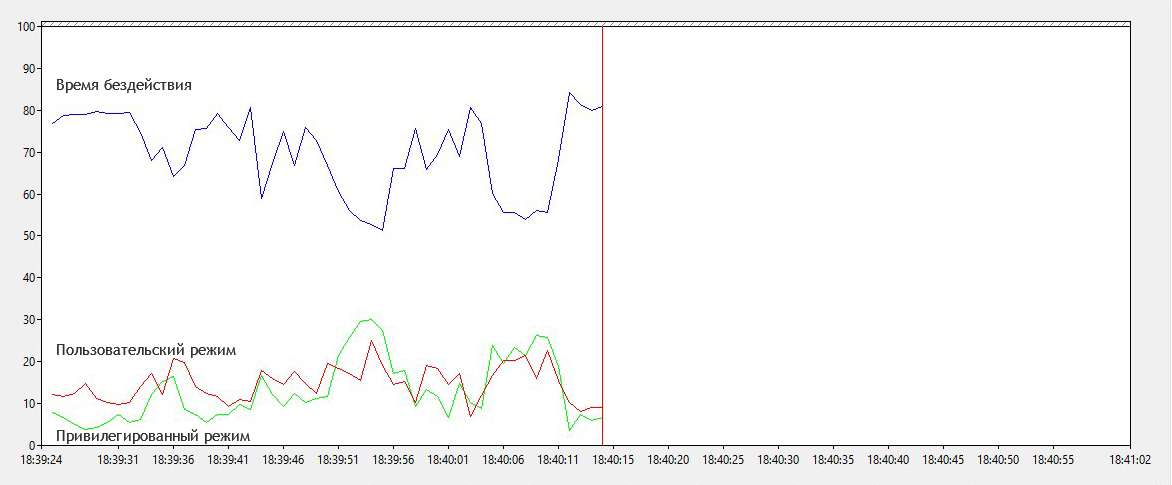
После запуска приложений:



Определим процент работы в пользовательском и привилегированном режиме, а также время бездействия системы:



На графике видно, что операционная система в большей степени работает именно в пользовательском режиме, а также в основном бездействует. Однако, если мы запустим приложение, работающее с графическими средствами (в данном случае это Adobe Photoshop CS6), то график изменится:



Виды пики в некоторые моменты времени, когда процент работы в привилегированном режиме превышает процент работы в пользовательском. Это те самые моменты, когда выполнялись манипуляции с графическими средствами (кисть, карандаш) в программе. Так же время бездействия значительно сокращается.

## Код программ

### DllImport\Kernel32.cs

Файл, отвечающий за подключение API-функций из библиотеки kernel32.dll:

|  |  |
| --- | --- |
| 001 | class Kernel32 { |
| 002 | [DllImport("Kernel32", SetLastError = true)] |
| 003 | [PreserveSig] |
| 004 | public static extern uint GetModuleFileName([In] IntPtr hModule,  [Out] StringBuilder lpFilename, |
| 005 | [In] [MarshalAs(UnmanagedType.U4)]   int nSize); |
| 006 |  |
| 007 | [DllImport("Kernel32", CharSet = CharSet.Auto)] |
| 008 | public static extern IntPtr GetModuleHandle(string lpModuleName); |
| 009 |  |
| 010 | [DllImport("Kernel32")] |
| 011 | public static extern uint GetCurrentProcessId(); |
| 012 |  |
| 013 | [DllImport("Kernel32")] |
| 014 | public static extern IntPtr GetCurrentProcess(); |
| 015 |  |
| 016 | [DllImport("Kernel32", SetLastError = true)] |
| 017 | [return: MarshalAs(UnmanagedType.Bool)] |
| 018 | public static extern bool DuplicateHandle(IntPtr hSourceProcessHandle,  IntPtr hSourceHandle, |
| 019 | IntPtr hTargetProcessHandle,  out IntPtr lpTargetHandle, |
| 020 | uint dwDesiredAccess, |
| 021 | [MarshalAs(UnmanagedType.Bool)]  bool bInheritHandle, uint dwOptions); |
| 022 |  |
| 023 | [DllImport("Kernel32")] |
| 024 | public static extern IntPtr OpenProcess(int dwDesiredAccess,  bool bInheritHandle, int dwProcessId); |
| 025 |  |
| 026 | [Flags] |
| 027 | public enum ProcessAccessFlags : uint { |
| 028 | All = 0x001F0FFF, |
| 029 | Terminate = 0x00000001, |
| 030 | CreateThread = 0x00000002, |
| 031 | VMOperation = 0x00000008, |
| 032 | VMRead = 0x00000010, |
| 033 | VMWrite = 0x00000020, |
| 034 | DupHandle = 0x00000040, |
| 035 | SetInformation = 0x00000200, |
| 036 | QueryInformation = 0x00000400, |
| 037 | Synchronize = 0x00100000 |
| 038 | } |
| 039 |  |
| 040 | [DllImport("Kernel32", SetLastError = true)] |
| 041 | [return: MarshalAs(UnmanagedType.Bool)] |
| 042 | public static extern bool CloseHandle(IntPtr hObject); |
| 043 |  |
| 044 | [DllImport("Kernel32", SetLastError = true,  CharSet = System.Runtime.InteropServices.CharSet.Auto)] |
| 045 | public static extern IntPtr CreateToolhelp32Snapshot([In] UInt32 dwFlags,  [In] UInt32 th32ProcessID); |
| 046 |  |
| 047 | [Flags] |
| 048 | public enum SnapshotFlags : uint { |
| 049 | HeapList = 0x00000001, |
| 050 | Process = 0x00000002, |
| 051 | Thread = 0x00000004, |
| 052 | Module = 0x00000008, |
| 053 | Module32 = 0x00000010, |
| 054 | Inherit = 0x80000000, |
| 055 | All = 0x0000001F, |
| 056 | NoHeaps = 0x40000000 |
| 057 | } |
| 058 |  |
| 059 | [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)] |
| 060 | public struct PROCESSENTRY32 { |
| 061 | internal UInt32 dwSize; |
| 062 | internal UInt32 cntUsage; |
| 063 | internal UInt32 th32ProcessID; |
| 064 | internal IntPtr th32DefaultHeapID; |
| 065 | internal UInt32 th32ModuleID; |
| 066 | internal UInt32 cntThreads; |
| 067 | internal UInt32 th32ParentProcessID; |
| 068 | internal Int32 pcPriClassBase; |
| 069 | internal UInt32 dwFlags; |
| 070 | [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)] |
| 071 | internal string szExeFile; |
| 072 | } |
| 073 |  |
| 074 | [DllImport("Kernel32", SetLastError = true,  CharSet = System.Runtime.InteropServices.CharSet.Auto)] |
| 075 | public static extern bool Process32First([In] IntPtr hSnapshot,  ref PROCESSENTRY32 lppe); |
| 076 |  |
| 077 | [DllImport("Kernel32", SetLastError = true,  CharSet = System.Runtime.InteropServices.CharSet.Auto)] |
| 078 | public static extern bool Process32Next([In] IntPtr hSnapshot,  ref PROCESSENTRY32 lppe); |
| 079 |  |
| 080 | [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)] |
| 081 | public struct THREADENTRY32 { |
| 082 | internal UInt32 dwSize; |
| 083 | internal UInt32 cntUsage; |
| 084 | internal UInt32 th32ThreadID; |
| 085 | internal UInt32 th32OwnerProcessID; |
| 086 | internal UInt32 tpBasePri; |
| 087 | internal Int32 tpDeltaPri; |
| 088 | internal UInt32 dwFlags; |
| 089 | } |
| 090 |  |
| 091 | [DllImport("Kernel32")] |
| 092 | public static extern bool Thread32First(IntPtr hSnapshot,  ref THREADENTRY32 lpte); |
| 093 |  |
| 094 | [DllImport("Kernel32")] |
| 095 | public static extern bool Thread32Next(IntPtr hSnapshot,  ref THREADENTRY32 lpte); |
| 096 |  |
| 097 | [StructLayoutAttribute(LayoutKind.Sequential)] |
| 098 | public struct MODULEENTRY32 { |
| 099 | internal uint dwSize; |
| 100 | internal uint th32ModuleID; |
| 101 | internal uint th32ProcessID; |
| 102 | internal uint GlblcntUsage; |
| 103 | internal uint ProccntUsage; |
| 104 | internal IntPtr modBaseAddr; |
| 105 | internal uint modBaseSize; |
| 106 | internal IntPtr hModule; |
| 107 | [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)] |
| 108 | internal string szModule; |
| 109 | [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)] |
| 110 | internal string szExePath; |
| 111 | } |
| 112 |  |
| 113 | [DllImport("Kernel32")] |
| 114 | public static extern bool Module32First(IntPtr hSnapshot,  ref MODULEENTRY32 lpme); |
| 115 |  |
| 116 | [DllImport("Kernel32")] |
| 117 | public static extern bool Module32Next(IntPtr hSnapshot,  ref MODULEENTRY32 lpme); |
| 118 | } |

### DllImport\Psapi.cs

Файл, отвечающий за подключение API-функций из библиотеки psapi.dll:

|  |  |
| --- | --- |
| 01 | class Psapi **{** |
| 02 | [DllImport("Psapi", SetLastError = true)] |
| 03 | public static extern bool EnumProcesses([MarshalAs(UnmanagedType.LPArray,   ArraySubType = UnmanagedType.U4)] |
| 04 | [In][Out] UInt32[] processIds, UInt32 arraySizeBytes, |
| 05 | [MarshalAs(UnmanagedType.U4)] out UInt32 bytesCopied); |
| 06 |  |
| 07 | [DllImport("Psapi", SetLastError = true)] |
| 08 | public static extern bool EnumProcessModules(IntPtr hProcess, |
| 09 | [MarshalAs(UnmanagedType.LPArray,   ArraySubType = UnmanagedType.U4)] |
| 10 | [In][Out] uint[] lphModule, uint cb, |
| 11 | [MarshalAs(UnmanagedType.U4)]   out uint lpcbNeeded); |
| 12 |  |
| 13 | [DllImport("Psapi")] |
| 14 | public static extern uint GetModuleBaseName(IntPtr hProcess, IntPtr hModule, |
| 15 | StringBuilder lpBaseName, uint nSize); |
| 16 |  |
| 17 | [DllImport("Psapi")] |
| 18 | public static extern uint GetModuleFileNameEx(IntPtr hProcess, IntPtr hModule,   [Out] StringBuilder lpBaseName, |
| 19 | [In] [MarshalAs(UnmanagedType.U4)]   int nSize); |
| 20 |  |
| 21 | [StructLayout(LayoutKind.Sequential)] |
| 22 | public struct MODULEINFO { |
| 23 | public IntPtr lpBaseOfDll; |
| 24 | public uint SizeOfImage; |
| 25 | public IntPtr EntryPoint; |
| 26 | } |
| 27 |  |
| 28 | [DllImport("Psapi", SetLastError = true)] |
| 29 | public static extern bool GetModuleInformation(IntPtr hProcess,  IntPtr hModule,  out MODULEINFO lpmodinfo, |
| 30 | uint cb); |
| 31 |  |
| 32 | [DllImport("Psapi")] |
| 33 | public static extern bool EnumDeviceDrivers([MarshalAs(UnmanagedType.LPArray,   ArraySubType = UnmanagedType.U4)] |
| 34 | [In] [Out] UInt32[] ddAddresses,   UInt32 arraySizeBytes, |
| 35 | [MarshalAs(UnmanagedType.U4)] out   UInt32 bytesNeeded); |
| 36 |  |
| 37 | [DllImport("Psapi")] |
| 38 | public static extern int GetDeviceDriverBaseName(UInt32 ddAddress,   StringBuilder ddBaseName, |
| 39 | int baseNameStringSizeChars); |
| 40 | } |

### MainWindow.xaml.cs

Файл, отвечающий за основную логику работы программы:

|  |  |
| --- | --- |
| 001 | public partial class MainWindow : Window { |
| 002 | private DispatcherTimer timer = null; |
| 003 |  |
| 004 | public MainWindow() { |
| 005 | InitializeComponent(); |
| 006 |  |
| 007 | uint pid = Kernel32.GetCurrentProcessId(); |
| 008 | LabelCurrentPID.Content = pid; |
| 009 |  |
| 010 | IntPtr pHandle = Kernel32.GetCurrentProcess(); |
| 011 | LabelPHandle.Content = String.Format("0x{0:X8}", (uint)pHandle); |
| 012 |  |
| 013 | IntPtr pHandleDuplicate; |
| 014 | Kernel32.DuplicateHandle(pHandle, pHandle, pHandle, out pHandleDuplicate, |
| 015 | 0, false, 0x00000001 | 0x00000002); |
| 016 | LabelPHandleDuplicate.Content = String.Format("0x{0:X8}",   (uint)pHandleDuplicate); |
| 017 |  |
| 018 | IntPtr pHandleOpenProcess = Kernel32.OpenProcess(0x00000040, false,   Convert.ToInt32(pid)); |
| 019 | LabelPHandleOpenProcess.Content = String.Format("0x{0:X8}",   (uint)pHandleOpenProcess); |
| 020 |  |
| 021 | LabelPHandleDuplicateClose.Content =   (Kernel32.CloseHandle(pHandleDuplicate)) |
| 022 | ? "Закрылся " + LabelPHandleDuplicateClose.Content |
| 023 | : "Не закрылся " + LabelPHandleDuplicateClose.Content; |
| 024 | LabelPHandleOpenProcessClose.Content =   (Kernel32.CloseHandle(pHandleOpenProcess)) |
| 025 | ? "Закрылся " + LabelPHandleOpenProcessClose.Content |
| 026 | : "Не закрылся " + LabelPHandleOpenProcessClose.Content; |
| 027 |  |
| 028 | timer = new DispatcherTimer(); |
| 029 | timer.Tick += new EventHandler(timerTick); |
| 030 | timer.Interval = new TimeSpan(0, 0, 1); |
| 031 | timer.Start(); |
| 032 |  |
| 033 | UpdateModulesGrid(); |
| 034 | UpdateDriversGrid(); |
| 035 | UpdateThreadsGrid9x(); |
| 036 | UpdateModulesGrid9x(); |
| 037 | } |
| 038 |  |
| 039 | private void timerTick(object sender, EventArgs e) { |
| 040 | UpdateProcessesGrid(); |
| 041 | UpdateProcessesGrid9x(); |
| 042 | } |
| 043 |  |
| 044 | public static void GetHandle(ref string handle, ref string name,  ref string fullname) { |
| 045 | if (handle != null && handle.Length > 0) { |
| 046 | fullname = GetFullName(new IntPtr(Int32.Parse(handle))); |
| 047 | name = GetName(fullname); |
| 048 |  |
| 049 | return; |
| 050 | } |
| 051 |  |
| 052 | if (fullname != null && fullname.Length > 0) { |
| 053 | name = GetName(fullname); |
| 054 | handle = Kernel32.GetModuleHandle(name).ToInt32().ToString(); |
| 055 |  |
| 056 | return; |
| 057 | } |
| 058 |  |
| 059 | if (name != null && name.Length > 0) { |
| 060 | IntPtr h = Kernel32.GetModuleHandle(name); |
| 061 | handle = h.ToInt32().ToString(); |
| 062 | fullname = GetFullName(h); |
| 063 |  |
| 064 | return; |
| 065 | } |
| 066 |  |
| 067 | fullname = GetFullName(IntPtr.Zero); |
| 068 | name = GetName(fullname); |
| 069 | handle = Kernel32.GetModuleHandle(name).ToInt32().ToString(); |
| 070 | } |
| 071 |  |
| 072 | public static string GetFullName(IntPtr handle) { |
| 073 | StringBuilder fullname = new StringBuilder(255); |
| 074 | Kernel32.GetModuleFileName(handle, fullname, 255); |
| 075 |  |
| 076 | return fullname.ToString(); |
| 077 | } |
| 078 |  |
| 079 | public static string GetName(string fullname) { |
| 080 | string[] splited = fullname.Split('\\'); |
| 081 |  |
| 082 | return splited[splited.Length - 1]; |
| 083 | } |
| 084 |  |
| 085 | private void Button\_Click(object sender, RoutedEventArgs e) { |
| 086 | string name = ProcessName.Text; |
| 087 | string path = ProcessPath.Text; |
| 088 | string handle = ProcessHandle.Text.Split('x').Last(); |
| 089 | if (handle.Length != 0) { |
| 090 | handle = System.Int32.Parse(handle,   NumberStyles.AllowHexSpecifier).ToString(); |
| 091 | } |
| 092 |  |
| 093 | GetHandle(ref handle, ref name, ref path); |
| 094 |  |
| 095 | ProcessName.Text = name; |
| 096 | ProcessPath.Text = path; |
| 097 | ProcessHandle.Text = String.Format("0x{0:X8}", Convert.ToUInt32(handle)); |
| 098 | } |
| 099 |  |
| 100 | private void Button\_Click\_1(object sender, RoutedEventArgs e) { |
| 101 | ProcessName.Text = ""; |
| 102 | ProcessPath.Text = ""; |
| 103 | ProcessHandle.Text = ""; |
| 104 | } |
| 105 |  |
| 106 | public static uint[] GetPIDs() { |
| 107 | UInt32 size = 120, |
| 108 | bytes = size \* sizeof(UInt32), |
| 109 | copy; |
| 110 | var PIDs = new UInt32[size]; |
| 111 |  |
| 112 | bool success = Psapi.EnumProcesses(PIDs, bytes, out copy); |
| 113 |  |
| 114 | if (success && copy > 0) { |
| 115 | copy >>= 2; |
| 116 | var result = new UInt32[copy]; |
| 117 |  |
| 118 | for (int i = 0; i < copy; i++) { |
| 119 | result[i] = PIDs[i]; |
| 120 | } |
| 121 |  |
| 122 | return result; |
| 123 | } |
| 124 | else { |
| 125 | return new UInt32[0]; |
| 126 | } |
| 127 | } |
| 128 |  |
| 129 | public static IntPtr[] GetProcessHandles(uint[] PIDs) { |
| 130 | var result = new IntPtr[PIDs.Length]; |
| 131 |  |
| 132 | for (int i = 0; i < PIDs.Length; i++) { |
| 133 | result[i] =  Kernel32.OpenProcess((int)Kernel32.ProcessAccessFlags.QueryInformation |
| 134 | | (int)Kernel32.ProcessAccessFlags.VMRead, false, (int)PIDs[i]); |
| 135 | } |
| 136 |  |
| 137 | return result; |
| 138 | } |
| 139 |  |
| 140 | public static string[] GetProcessNames(IntPtr[] handles) { |
| 141 | var result = new string[handles.Length]; |
| 142 | var tmp = GetProcessModules(handles); |
| 143 |  |
| 144 | for (int i = 0; i < handles.Length; i++) { |
| 145 | if (tmp[i].Length == 0) { |
| 146 | result[i] = ""; |
| 147 | continue; |
| 148 | } |
| 149 |  |
| 150 | var sb = new StringBuilder(255); |
| 151 | uint size = Psapi.GetModuleBaseName(handles[i], new IntPtr(tmp[i][0]),   sb, 255); |
| 152 | result[i] = sb.ToString().Substring(0, (int)size); |
| 153 | } |
| 154 |  |
| 155 | return result; |
| 156 | } |
| 157 |  |
| 158 | public static uint[][] GetProcessModules(IntPtr[] handles) { |
| 159 | uint size = 1000, |
| 160 | bytes = size \* sizeof(uint), |
| 161 | copy; |
| 162 |  |
| 163 | var result = new uint[size]; |
| 164 | var full = new uint[handles.Length][]; |
| 165 |  |
| 166 | for (int i = 0; i < handles.Length; i++) { |
| 167 | Psapi.EnumProcessModules(handles[i], result, bytes, out copy); |
| 168 | copy >>= 2; |
| 169 | var tmp = new uint[copy]; |
| 170 | for (int j = 0; j < copy; j++) { |
| 171 | tmp[j] = result[j]; |
| 172 | } |
| 173 |  |
| 174 | full[i] = tmp; |
| 175 | } |
| 176 |  |
| 177 | return full; |
| 178 | } |
| 179 |  |
| 180 | private static Psapi.MODULEINFO[] GetModuleInfo(IntPtr handle, uint[] modules)   { |
| 181 | var infos = new Psapi.MODULEINFO[modules.Length]; |
| 182 | Psapi.MODULEINFO info; |
| 183 |  |
| 184 | for (int i = 0; i < modules.Length; i++) { |
| 185 | if (Psapi.GetModuleInformation(handle, new IntPtr(modules[i]), out info, |
| 186 | (uint)Marshal.SizeOf(new Psapi.MODULEINFO()))) { |
| 187 | infos[i] = info; |
| 188 | } |
| 189 | else { |
| 190 | infos[i] = new Psapi.MODULEINFO(); |
| 191 | infos[i].EntryPoint = IntPtr.Zero; |
| 192 | infos[i].lpBaseOfDll = IntPtr.Zero; |
| 193 | infos[i].SizeOfImage = 0; |
| 194 | } |
| 195 | } |
| 196 |  |
| 197 | return infos; |
| 198 | } |
| 199 |  |
| 200 | public static string[] GetModulesInfo(IntPtr[] handles) { |
| 201 | var result = new string[handles.Length]; |
| 202 | var tmp = GetProcessModules(handles); |
| 203 |  |
| 204 | for (int i = 0; i < handles.Length; i++) { |
| 205 | if (tmp[i].Length == 0) { |
| 206 | result[i] = ""; |
| 207 | continue; |
| 208 | } |
| 209 |  |
| 210 | Psapi.MODULEINFO info; |
| 211 |  |
| 212 | if (Psapi.GetModuleInformation(handles[i], new IntPtr(tmp[i][0]),  out info, (uint)Marshal.SizeOf(new Psapi.MODULEINFO()))) { |
| 213 | result[i] = info.SizeOfImage + " б"; |
| 214 | while (result[i].Length < 30) { |
| 215 | result[i] += " "; |
| 216 | } |
| 217 | } |
| 218 | } |
| 219 |  |
| 220 | return result; |
| 221 | } |
| 222 |  |
| 223 | public static string[][] GetModulesFilenames(uint[] PIDs, string[] names,   IntPtr[] handles) { |
| 224 | var result = new List<string[]>(); |
| 225 | var tmp = GetProcessModules(handles); |
| 226 |  |
| 227 | for (int i = 0; i < tmp.GetLength(0); i++) { |
| 228 | if (tmp[i].Length == 0) { |
| 229 | continue; |
| 230 | } |
| 231 |  |
| 232 | Psapi.MODULEINFO[] info = GetModuleInfo(handles[i], tmp[i]); |
| 233 |  |
| 234 | for (int j = tmp[i].Length - 1; j >= 0; j--) { |
| 235 | var sb = new StringBuilder(255); |
| 236 | var size = Psapi.GetModuleFileNameEx(handles[i],  new IntPtr(tmp[i][j]), sb, 255); |
| 237 | result.Add(new string[] { PIDs[i].ToString(), names[i], |
| 238 | info[j].lpBaseOfDll.ToString(), |
| 239 | info[j].EntryPoint.ToString(), |
| 240 | info[j].SizeOfImage.ToString() + " б", |
| 241 | sb.ToString() }); |
| 242 | } |
| 243 | } |
| 244 |  |
| 245 | return result.ToArray(); |
| 246 | } |
| 247 |  |
| 248 | public static List<string[]> GetDriversInfo() { |
| 249 | var result = new List<string[]>(); |
| 250 | uint size, bytes, needed; |
| 251 | uint[] addresses; |
| 252 |  |
| 253 | bool success = Psapi.EnumDeviceDrivers(null, 0, out needed); |
| 254 |  |
| 255 | if (!success || needed == 0) { |
| 256 | return null; |
| 257 | } |
| 258 |  |
| 259 | size = needed >> 2; |
| 260 | bytes = needed; |
| 261 | addresses = new uint[size]; |
| 262 |  |
| 263 | success = Psapi.EnumDeviceDrivers(addresses, bytes, out needed); |
| 264 |  |
| 265 | if (!success) { |
| 266 | return null; |
| 267 | } |
| 268 |  |
| 269 | for (int i = 0; i < size; i++) { |
| 270 | var sb = new StringBuilder(1000); |
| 271 |  |
| 272 | int res = Psapi.GetDeviceDriverBaseName(addresses[i], sb, sb.Capacity); |
| 273 |  |
| 274 | result.Add(new string[] { addresses[i].ToString(), sb.ToString() }); |
| 275 | } |
| 276 |  |
| 277 | return result; |
| 278 | } |
| 279 |  |
| 280 | public static List<string[]> GetProcessesInfo() { |
| 281 | var result = new List<string[]>(); |
| 282 | var handleToSnapshot = IntPtr.Zero; |
| 283 |  |
| 284 | try { |
| 285 | var procEntry = new Kernel32.PROCESSENTRY32(); |
| 286 | procEntry.dwSize =   (uint)Marshal.SizeOf(typeof(Kernel32.PROCESSENTRY32)); |
| 287 | handleToSnapshot = Kernel32.CreateToolhelp32Snapshot(   (uint)Kernel32.SnapshotFlags.Process, 0); |
| 288 |  |
| 289 | if (Kernel32.Process32First(handleToSnapshot, ref procEntry)) { |
| 290 | do { |
| 291 | result.Add(new string[] { procEntry.cntUsage.ToString(),   procEntry.th32ProcessID.ToString(), |
| 292 | procEntry.th32DefaultHeapID.ToString(), |
| 293 | procEntry.th32ModuleID.ToString(), |
| 294 | procEntry.pcPriClassBase.ToString(), |
| 295 | procEntry.szExeFile.ToString() |
| 296 | }); |
| 297 | } while (Kernel32.Process32Next(handleToSnapshot, ref procEntry)); |
| 298 | } |
| 299 | else { |
| 300 | throw new ApplicationException(  string.Format("Failed with win32 error code {0}",   Marshal.GetLastWin32Error())); |
| 301 | } |
| 302 | } |
| 303 | catch {} |
| 304 | finally { |
| 305 | Kernel32.CloseHandle(handleToSnapshot); |
| 306 | } |
| 307 |  |
| 308 | return result; |
| 309 | } |
| 310 |  |
| 311 | public static List<string[]> GetThreadsInfo() { |
| 312 | var result = new List<string[]>(); |
| 313 | var handleToSnapshot = IntPtr.Zero; |
| 314 |  |
| 315 | try { |
| 316 | var threadEntry = new Kernel32.THREADENTRY32(); |
| 317 | threadEntry.dwSize =   (uint)Marshal.SizeOf(typeof(Kernel32.THREADENTRY32)); |
| 318 |  |
| 319 | if (Kernel32.Thread32First(handleToSnapshot, ref threadEntry)) { |
| 320 | do { |
| 321 | result.Add(new string[] { threadEntry.cntUsage.ToString(),   threadEntry.th32ThreadID.ToString(), |
| 322 | threadEntry.th32OwnerProcessID.ToString(), |
| 323 | threadEntry.tpBasePri.ToString(),   threadEntry.tpDeltaPri.ToString() |
| 324 | }); |
| 325 | } while (Kernel32.Thread32Next(handleToSnapshot, ref threadEntry)); |
| 326 | } |
| 327 | else { |
| 328 | throw new ApplicationException(  string.Format("Failed with win32 error code {0}",   Marshal.GetLastWin32Error())); |
| 329 | } |
| 330 | } |
| 331 | catch { } |
| 332 | finally { |
| 333 | Kernel32.CloseHandle(handleToSnapshot); |
| 334 | } |
| 335 |  |
| 336 | return result; |
| 337 | } |
| 338 |  |
| 339 | public static List<string[]> GetModulesInfo() { |
| 340 | var result = new List<string[]>(); |
| 341 | var handleToSnapshot = IntPtr.Zero; |
| 342 |  |
| 343 | try { |
| 344 | var modEntry = new Kernel32.MODULEENTRY32(); |
| 345 | modEntry.dwSize = (uint)Marshal.SizeOf(typeof(Kernel32.MODULEENTRY32)); |
| 346 | handleToSnapshot =   Kernel32.CreateToolhelp32Snapshot((uint)Kernel32.SnapshotFlags.All,   0); |
| 347 |  |
| 348 | if (Kernel32.Module32First(handleToSnapshot, ref modEntry)) { |
| 349 | do { |
| 350 | result.Add(new string[] { modEntry.th32ModuleID.ToString(),   modEntry.th32ProcessID.ToString(), |
| 351 | modEntry.GlblcntUsage.ToString(),   modEntry.ProccntUsage.ToString(), |
| 352 | modEntry.modBaseAddr.ToString(),   modEntry.modBaseSize.ToString(), |
| 353 | String.Format("0x{0:X8}",   (uint)modEntry.hModule), |
| 354 | modEntry.szModule.ToString(),  modEntry.szExePath.ToString() |
| 355 | }); |
| 356 | } while (Kernel32.Module32Next(handleToSnapshot, ref modEntry)); |
| 357 | } |
| 358 | else { |
| 359 | throw new ApplicationException(  string.Format("Failed with win32 error code {0}",   Marshal.GetLastWin32Error())); |
| 360 | } |
| 361 | } |
| 362 | finally { |
| 363 | Kernel32.CloseHandle(handleToSnapshot); |
| 364 | } |
| 365 |  |
| 366 | return result; |
| 367 | } |
| 368 |  |
| 369 | private void UpdateProcessesGrid() { |
| 370 | var PIDs = GetPIDs(); |
| 371 | var tmp = ProcessesCounter.Content; |
| 372 | ProcessesCounter.Content = "Обновляю.."; |
| 373 |  |
| 374 | if (ProcessesGrid.Items.Count == 0  || ProcessesGrid.Items.Count != PIDs.Length) { |
| 375 | var handles = GetProcessHandles(PIDs); |
| 376 | var names = GetProcessNames(handles); |
| 377 | var modules = GetModulesInfo(handles); |
| 378 | var ps = Processes.Fill(PIDs, handles, names, modules); |
| 379 |  |
| 380 | ProcessesCounter.Content = "Всего процессов: " + ps.Count.ToString(); |
| 381 | ProcessesGrid.ItemsSource = ps.OrderByDescending(o => o.Name).ToList(); |
| 382 |  |
| 383 | ProcessesGrid.Items.Refresh(); |
| 384 |  |
| 385 | for (int i = 0; i < handles.Length; i++) { |
| 386 | Kernel32.CloseHandle(handles[i]); |
| 387 | } |
| 388 | } |
| 389 | else |
| 390 | ProcessesCounter.Content = tmp; |
| 391 | } |
| 392 |  |
| 393 | private void UpdateModulesGrid() { |
| 394 | var PIDs = GetPIDs(); |
| 395 | var handles = GetProcessHandles(PIDs); |
| 396 | var names = GetProcessNames(handles); |
| 397 | var filenames = GetModulesFilenames(PIDs, names, handles); |
| 398 | var ms = Modules.Fill(filenames); |
| 399 |  |
| 400 | ModulesCounter.Content = "Модулей: " + ms.Count.ToString(); |
| 401 | ModulesGrid.ItemsSource = ms; |
| 402 |  |
| 403 | for (int i = 0; i < handles.Length; i++) |
| 404 | Kernel32.CloseHandle(handles[i]); |
| 405 | } |
| 406 |  |
| 407 | private void UpdateDriversGrid() { |
| 408 | var drivers = GetDriversInfo(); |
| 409 | var ds = Drivers.Fill(drivers); |
| 410 |  |
| 411 | DriversCounter.Content = "Драйверов: " + ds.Count.ToString(); |
| 412 | DriversGrid.ItemsSource = ds; |
| 413 | } |
| 414 |  |
| 415 | private void UpdateProcessesGrid9x() { |
| 416 | var processes = GetProcessesInfo(); |
| 417 | var tmp = ProcessesCounter9x.Content; |
| 418 | ProcessesCounter9x.Content = "Обновляю.."; |
| 419 |  |
| 420 | if (ProcessesGrid9x.Items.Count == 0  || ProcessesGrid9x.Items.Count != processes.Count) { |
| 421 | var ps = Processes9x.Fill(processes); |
| 422 |  |
| 423 | ProcessesCounter9x.Content = "Процессов: " + ps.Count.ToString(); |
| 424 | ProcessesGrid9x.ItemsSource = ps; |
| 425 | } |
| 426 | else { |
| 427 | ProcessesCounter9x.Content = tmp; |
| 428 | } |
| 429 | } |
| 430 |  |
| 431 | private void UpdateThreadsGrid9x() { |
| 432 | var threads = GetThreadsInfo(); |
| 433 | var tmp = ThreadsCounter9x.Content; |
| 434 | ThreadsCounter9x.Content = "Обновляю..."; |
| 435 |  |
| 436 | if (ThreadsGrid9x.Items.Count == 0  || ThreadsGrid9x.Items.Count != threads.Count) { |
| 437 | var ts = Threads9x.Fill(threads); |
| 438 |  |
| 439 | ThreadsCounter9x.Content = "Потоков: " + ts.Count.ToString(); |
| 440 | ThreadsGrid9x.ItemsSource = ts; |
| 441 | } |
| 442 | else { ThreadsCounter9x.Content = tmp; } |
| 443 | } |
| 444 |  |
| 445 | private void UpdateModulesGrid9x() { |
| 446 | var modules = GetModulesInfo(); |
| 447 | var tmp = ModulesCounter9x.Content; |
| 448 | ModulesCounter9x.Content = "Обновляю..."; |
| 449 |  |
| 450 | if (ModulesGrid9x.Items.Count == 0  || ModulesGrid9x.Items.Count != modules.Count) { |
| 451 | var ms = Modules9x.Fill(modules); |
| 452 |  |
| 453 | ModulesCounter9x.Content = "Модулей: " + ms.Count.ToString(); |
| 454 | ModulesGrid9x.ItemsSource = ms; |
| 455 | } |
| 456 | else { ModulesCounter9x.Content = tmp; } |
| 457 | } |
| 458 |  |
| 459 | public static string PriorityToString(string pri) { |
| 460 | switch (pri) { |
| 461 | case "4": return "Ожидающий"; |
| 462 | case "8": return "Нормальный"; |
| 463 | case "13": return "Высокий"; |
| 464 | case "24": return "Реального времени"; |
| 465 | default: return "Неизвестен"; |
| 466 | } |
| 467 | } |
| 468 | } |