Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Лабораторная работа

Дисциплина: Проектирование ОС и компонентов

Тема: Создание инсталятора и службы в Windows и демона в Linux

Выполнил студент гр. 13541/4 \_\_\_\_\_\_\_\_\_\_ Зорин А.Г.

(подпись)

Руководитель \_\_\_\_\_\_\_\_\_\_ Душутина Е.В.

(подпись)

«\_\_»\_\_\_\_\_\_\_\_ 2017 г.

Санкт –Петербург

2017

**Цель работы:**

В данной работе необходимо выполнить:

* Создание инсталятора Windows
* Создание службы Windows
* Создание демона в Linux

Все эти пункты необходимо выполнить по отношению к утилите, которая была написана ранее. Суть этой утилиты заключается в отслеживании изменений файлов и директорий. Эти пункты выполняются в ОС двух семейств.

**Выполнение работы:**

1. Ubuntu

Информация о системе:

|  |
| --- |
| ➜ ~ uname -a  Linux Arseny-HP-Pavilion-dv6-Notebook-PC 4.4.0-66-generic #87-Ubuntu SMP Fri Mar 3 15:29:05 UTC 2017 x86\_64 x86\_64 x86\_64 GNU/Linux |

Демоном в UNIX системах называют программу, которая работает в «фоновом режиме», не требуя управления с терминала, тем самым, предоставляет возможность выполнять другую работу «на переднем фоне». Демон может быть запущен либо другим процессом, например, одним из системных стартовых скриптов вообще без обращения к какому-либо управляющему терминалу, либо пользователем с какого-нибудь терминала, но и в этом случае демон "не захватывает" терминал на время своей работы. В данной работе будет использоваться второй вариант запуска демона.

Для того, чтобы сделать из консольной утилиты демон, необходимо отвязаться от консоли. Для этого был использован системный вызов fork(), который позволяет создать новый процесс. При вызове fork() возникают два полностью идентичных процесса. Весь код выполняется дважды, как в процессе-потомке, так и в процессе-родителе. Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родитель получает идентификатор (PID) потомка. Если это значение будет отрицательным, следовательно при порождении процесса произошла ошибка. Процесс-потомок получает в качестве кода возврата значение 0, если вызов fork() оказался успешным.

Использование fork() в утилите показано ниже

|  |
| --- |
| switch (fork()){  case -1:  writeLog(strcat("Daemon start failed ", strerror(errno)));  return -1;  case 0:  writeLog("Daemon is running");  umask(0);  //Cоздание нового сеанса, чтобы не зависеть от родителя  setsid();  //Переход в корень диска,  //чтобы потом не было непредвиденных проблем  chdir("/");  //Закрытие дискрипторов вывода/ввода/ошибок  close(STDIN\_FILENO);  close(STDOUT\_FILENO);  close(STDERR\_FILENO);  watch\_dir = argv[1];  return do\_watch(max\_watches);  default:  return 0;  } |

Закрытие дескрипторов необходимо по той причине, что printf, scanf и прочие команды работы с консолью не будут использоваться в дальнейшем.

Так как написанная ранее утилита отслеживала изменение директорий, то такие изменения она должна куда-то выводить. Поскольку терминал не будет использоваться, все изменения будут записываться в файл .log. Функция, выполняющая такие записи показана ниже:

|  |
| --- |
| }char\* getTime() {  time\_t now;  struct tm \*ptr;  static char tbuf[64];  bzero(tbuf,64);  time(&now);  ptr = localtime(&now);  strftime(tbuf,64, "%Y-%m-%e %H:%M:%S", ptr);  return tbuf;  }  int writeLog(char msg[256]){  FILE\* pLog = fopen("/home/arseny/daemon.log", "a");  if(pLog == NULL)  return -1;  char str[312];  bzero(str,312);  strcpy(str, getTime());  strcat(str, "\n====================\n");  strcat(str,msg);  strcat(str, "\n\n");  fputs(str, pLog);  fclose(pLog);  return 0;  } |

После сборки проекта, запускаем демон также, как и обычную консольную утилиту. И проверим, что демон запустился через ps aux

|  |
| --- |
| ➜ inot ./inot /home/arseny/DirForWatching  ➜ inot ps aux | grep inot  arseny 22660 0.0 0.0 22816 96 ? Ss 11:50 0:00 ./inot /home/arseny/DirForWatching  arseny 22666 0.0 0.0 14164 1092 pts/8 S+ 11:50 0:00 grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn inot |

Для проверки корректной работы демона, были проведены изменения в отслеживаемой директории. Ниже показано содержание .log файла

|  |
| --- |
| 2017-04-18 11:53:22  ====================  Daemon is running  2017-04-18 11:53:22  ====================  Doing initial watches setup  2017-04-18 11:53:33  ====================  Path: /home/arseny/DirForWatching/  File: Untitled Document  Event: FILE WAS MOVED OUTSIDE WATCHING DIR  2017-04-18 11:53:33  ====================  Path: /home/arseny/DirForWatching/  File: fff  Event: FILE WAS MOVED INSIDE WATCHING DIR  2017-04-18 11:53:43  ====================  Path: /home/arseny/DirForWatching/  File: Untitled Folder  Event: OBJECT IS A DIR  2017-04-18 11:53:51  ====================  Path: /home/arseny/DirForWatching/  File: fff  Event: FILE WAS MOVED OUTSIDE WATCHING DIR  2017-04-18 11:53:51  ====================  Path: /home/arseny/DirForWatching/Untitled Folder/  File: fff  Event: FILE WAS MOVED INSIDE WATCHING DIR  2017-04-18 11:54:02  ====================  Path: /home/arseny/DirForWatching/  File: Untitled Folder  Event: FILE WAS MOVED OUTSIDE WATCHING DIR  2017-04-18 11:54:02  ====================  Path: /home/arseny/DirForWatching/Untitled Folder/  File: fff  Event: WATCHING DIR WAS MOVED |

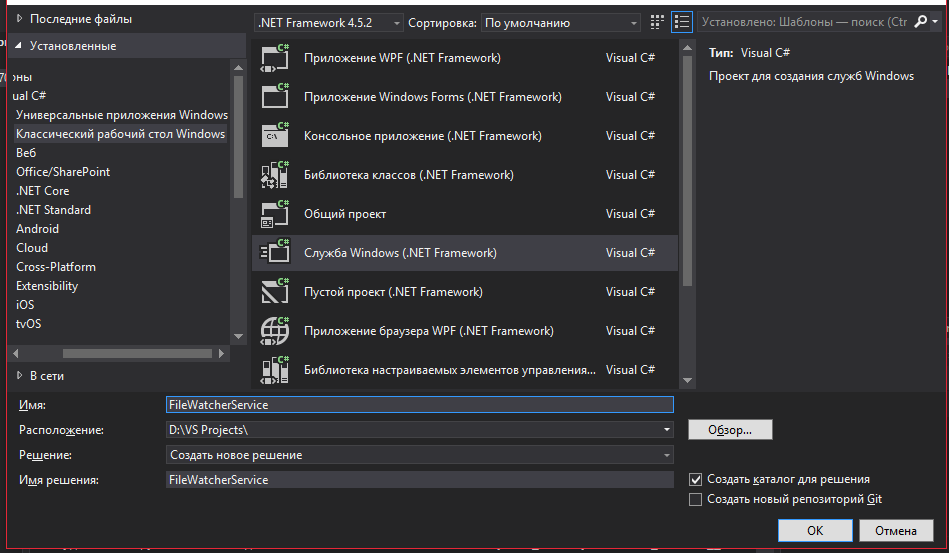
2. Windows 10.

2.1. Создание службы

|  |
| --- |
| C:\Users\zorin>systeminfo  Имя узла: DESKTOP-1LD1IT8  Название ОС: Майкрософт Windows 10 Pro (Registered Trademark)  Версия ОС: 10.0.14393 Н/Д построение 14393  Изготовитель ОС: Microsoft Corporation  Параметры ОС: Изолированная рабочая станция  Построение ОС: Multiprocessor Free  Зарегистрированный владелец: Пользователь Windows |

Для создания службы на Windows использовался тот же язык программирования, что и для создания утилиты – C#, с использованием IDE Visual Studio 2017.

Создание службы заключается в выборе службы Windows (.NET Framework) при создании проекта.



Служба представляет обычное приложение, но она не запускается сама по себе. Все вызовы и обращения к ней проходят через менеджер управления службами (Service Control Manager или SCM). Когда служба запускается автоматически при старте системы или вручную, то SCM обращается к методу Main в классе Program.

Метод Main по умолчанию определен таким образом, чтобы запускать сразу несколько служб, которые определены в массиве ServicesToRun. Однако, по умолчанию, проект содержит только одну службу Service1. Сам запуск производится с помощью метода Run: ServiceBase.Run(ServicesToRun).

Сама запускаемая служба представлена узлом Service1.cs. Однако на самом деле это не простой файл кода. Если мы откроем этот узел, то увидим в нем файл дизайнера службы Service1.Designer.cs и класс Service1.

Класс Service1 представляет службу.

Код службы, после его изменения выглядит следующим образом:

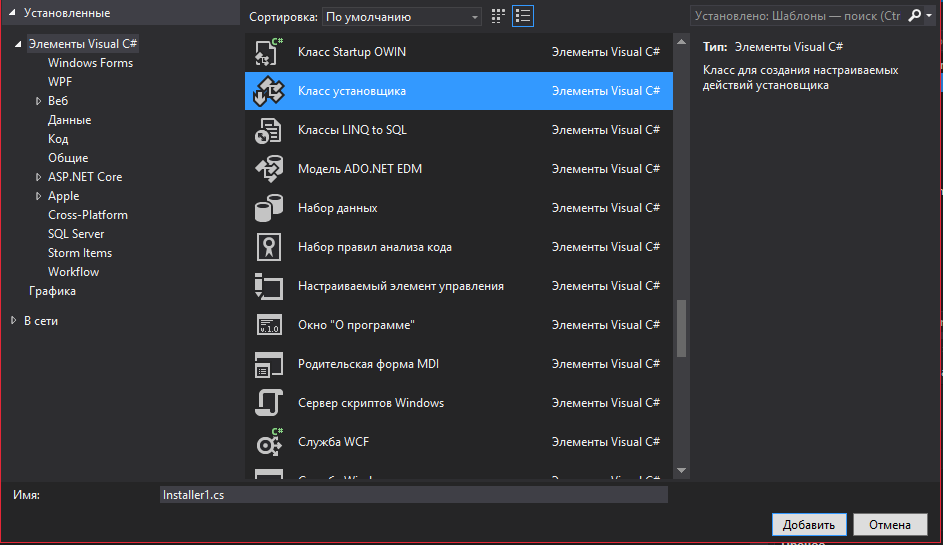
|  |
| --- |
| using System;  using System.Security.Permissions;  using System.ServiceProcess;  using DynamicLibraryFSW;  using StaticLibraryFSW;  using System.Threading;  namespace FileWatcherService  {  public partial class Service1 : ServiceBase  {  Watch watch;  public Service1()  {  InitializeComponent();  this.CanStop = true;  this.CanPauseAndContinue = true;  this.AutoLog = true;  }  protected override void OnStart(string[] args)  {  watch = new Watch();  Thread watchThread = new Thread(new ThreadStart(watch.Start));  watchThread.Start();  }  protected override void OnStop()  {  watch.Stop();  Thread.Sleep(1000);  }  }  class Watch  {  bool enabled = true;  [PermissionSet(SecurityAction.Demand, Name = "FullTrust")]  /\*private static void Main()  {  while (true)  {  //Ввод отслеживаемой директории  Console.WriteLine("Enter directory name for monitoring:");  var directory = Console.ReadLine();  //Проверка введенной директории на корректность  try  {  if ("".Equals(directory)) throw new FileNotFoundException();  if ("q".Equals(directory))  {  Console.WriteLine("Exit from util");  return;  }  }  catch (FileNotFoundException)  {  Console.WriteLine("Error: Incorrect directory name");  continue;  }  catch (Exception e)  {  Console.WriteLine(e.Message);  continue;  }  //Выбор дальнейшей работы утилиты  //Будут отслеживаться изменения всей директории или одного конкретного файла  Console.WriteLine("Press 'f' for selecting file for monitoring changes");  Console.WriteLine("Press any key for monitoring changes of all files in selected direcory");  var typedValue = Console.ReadLine();  Console.WriteLine();  //Ключ утилиты соответствующий отслеживанию конкретного файла  if ("f".Equals(typedValue.ToLower()))  {  //Метод отслеживания конкретного файла  if (!OneFile(directory)) return;  continue;  }  //Отслеживание всей директории  if (!AllFiles(directory)) return;  }  }\*/  public Watch()  {  AllFiles("D:\\DirForWatching");  }  public void Start()  {  while (enabled)  {  Thread.Sleep(1000);  }  }  public void Stop()  {  enabled = false;  }  private static bool AllFiles(string path)  {  while (true)  {  //Ключи утилиты для выхода и для ввода новой директории  Console.WriteLine("There will be changes of all files in the directory you entered");  Console.WriteLine();  Console.WriteLine("Press 'q' to quit this util");  Console.WriteLine("Press 'd' to enter new directory");  Console.WriteLine();  //Метод задания всех свойств объекта класса  //+Метод из динамической библиотеки  WatcherEven.WatcherEvents(Watcher.GetFolderWatcher(path));  var typedValue = Console.ReadLine();  Console.WriteLine();  //Выход из метода  if ("q".Equals(typedValue.ToLower())) return false;  if ("d".Equals(typedValue.ToLower())) return true;  }  }  private static bool OneFile(string path)  {  while (true)  {  //Ввод имени файла  //Console.WriteLine("Enter file name for monitoring:");  Console.WriteLine();  Console.WriteLine("Press 'q' to quit this util");  Console.WriteLine("Press 'd' to enter new directory");  //Метод задания всех свойств объекта класса  WatcherEven.WatcherEvents(Watcher.GetFileWatcher(path));  var typedValue = Console.ReadLine();  Console.WriteLine();  //Выход из метода  if ("q".Equals(typedValue.ToLower())) return false;  if ("d".Equals(typedValue.ToLower())) return true;  }  }  }  } |

Функция логгирования была добавлена в статическую библиотеку, так как события находятся там. Таким образом, код статической библиотеки выглядит следующим образом:

|  |
| --- |
| // StaticLibraryFSW.h  #pragma once  #using <System.dll>  using namespace System;  using namespace System::IO;  using namespace System::Security::Permissions;  using namespace System::Threading;  namespace StaticLibraryFSW {  public ref class Lock {  Object^ m\_pObject;  public:  Lock(Object ^ pObject) : m\_pObject(pObject) {  Monitor::Enter(m\_pObject);  }  ~Lock() {  Monitor::Exit(m\_pObject);  }  };  public ref class WatcherEven  {  public:  static void WatcherEvents(FileSystemWatcher^ fsWatcher)  {  //Задание отслеживаемых изменений  fsWatcher -> NotifyFilter = static\_cast<NotifyFilters>(NotifyFilters::LastAccess |  NotifyFilters::LastWrite | NotifyFilters::FileName | NotifyFilters::DirectoryName  | NotifyFilters::Attributes | NotifyFilters::CreationTime | NotifyFilters::Security  | NotifyFilters::Size);      //Подпись на события изменений  fsWatcher->Created += gcnew FileSystemEventHandler(WatcherEven::OnCreated);  fsWatcher->Changed += gcnew FileSystemEventHandler(WatcherEven::OnChanged);  fsWatcher->Deleted += gcnew FileSystemEventHandler(WatcherEven::OnDeleted);  fsWatcher->Renamed += gcnew RenamedEventHandler(WatcherEven::OnRenamed);  fsWatcher->Error += gcnew ErrorEventHandler(WatcherEven::OnError);  //Обеспечение отслеживания изменений  fsWatcher->EnableRaisingEvents = true;  }  private:  static void OnCreated(Object^ source, FileSystemEventArgs^ e)  {  Console::WriteLine("{0}\nFile {1} was created",  DateTime::Now, e->FullPath, e->ChangeType);  RecordEntry(source, e->ChangeType.ToString(), e->FullPath);  }    static void OnChanged(Object^ source, FileSystemEventArgs^ e)  {  Console::WriteLine("{0}\nFile {1}\tChange type: {2}",  DateTime::Now, e->FullPath, e->ChangeType);  RecordEntry(source, e->ChangeType.ToString(), e->FullPath);  }  static void OnDeleted(Object^ source, FileSystemEventArgs^ e)  {  Console::WriteLine("{0}\nFile {1} was deleted",  DateTime::Now, e->FullPath);  RecordEntry(source, e->ChangeType.ToString(), e->FullPath);  }  static void OnRenamed(Object^ source, RenamedEventArgs^ e)  {  Console::WriteLine("{0}\nFile {1} was renamed to {2}",  DateTime::Now, e->OldFullPath, e->FullPath);  RecordEntry(source, e->ChangeType.ToString(), e->FullPath);  }  static void OnError(Object^ source, ErrorEventArgs^ e)  {  Console::WriteLine("There was an error {0}", e);  }  static void RecordEntry(Object^ source, String^ fileEvent, String^ filePath)  {  Lock lock(source);  StreamWriter^ writer = gcnew StreamWriter("D:\\DirForWatching\\templog.txt", true);  DateTime^ dt = gcnew DateTime();  String^ a = gcnew String(dt->Now.ToString("dd/MM/yyyy hh:mm:ss") + ": file " + filePath + "was " + fileEvent );  writer -> Write(a);  writer->Flush();  }  };  } |

После реализации класса службы, необходимо реализовать установщик этой службы, чтобы ее возможно было запустить.

Для установки службы необходимо добавить класс установщика в текущий проект.



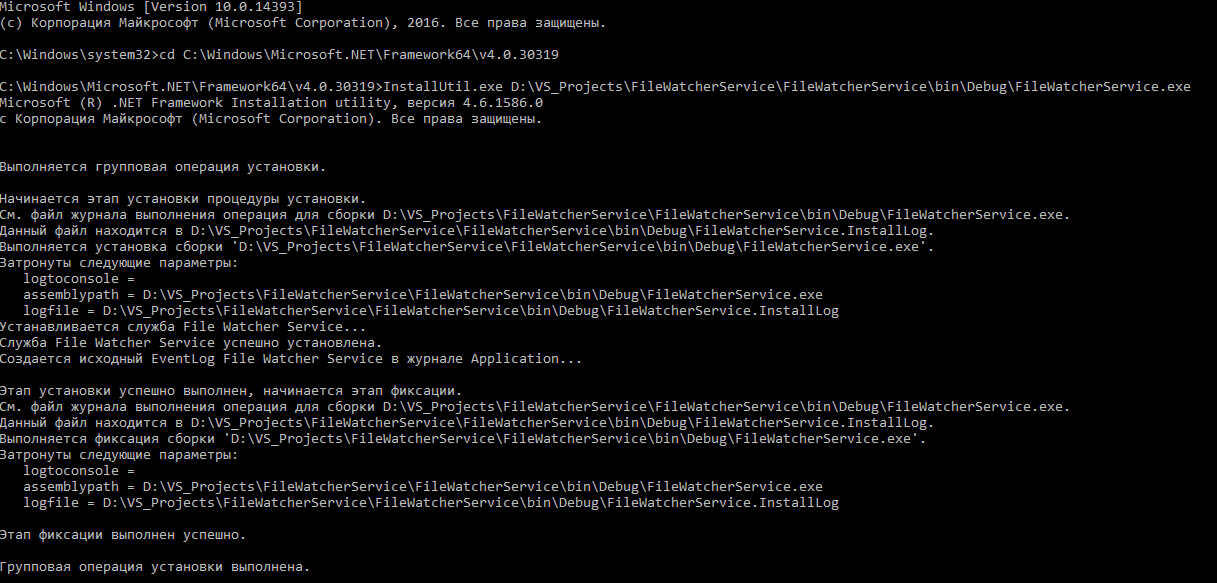
Класс установщика выглядит следующим образом:

|  |
| --- |
| using System;  using System.Collections;  using System.Collections.Generic;  using System.ComponentModel;  using System.Configuration.Install;  using System.Linq;  using System.ServiceProcess;  using System.Threading.Tasks;  namespace FileWatcherService  {  [RunInstaller(true)]  public partial class Installer1 : System.Configuration.Install.Installer  {  ServiceInstaller serviceInstaller;  ServiceProcessInstaller processInstaller;  public Installer1()  {  InitializeComponent();  serviceInstaller = new ServiceInstaller();  processInstaller = new ServiceProcessInstaller();  processInstaller.Account = ServiceAccount.LocalSystem;  serviceInstaller.StartType = ServiceStartMode.Manual;  serviceInstaller.ServiceName = "File Watcher Service";  Installers.Add(processInstaller);  Installers.Add(serviceInstaller);  }  }  } |

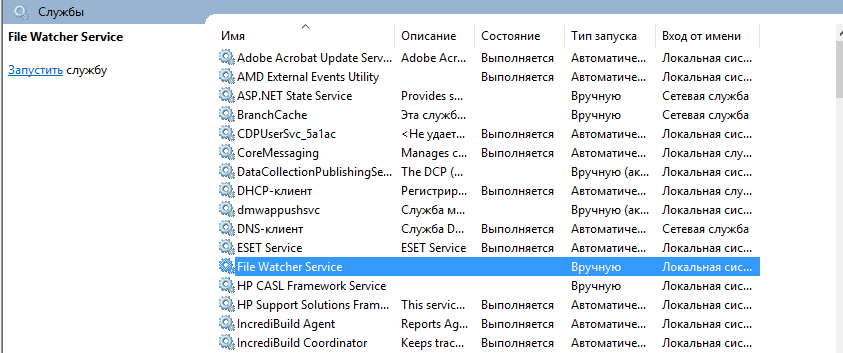
Для установки службы нам также понадобится классы ServiceInstaller и ServiceProcessInstaller. ServiceProcessInstaller управляет настройкой значений для всех запускаемых служб внутри одного процесса. Класс ServiceInstaller предназначен для настройки значений для каждой из запускаемых служб.

Следующим этапом необходимо скомпилировать проект, после этого, в папке bin/Debug появится исполняемый файл службы.

Запустить этот файл можно с помощью утилиты InstallUtil.exe, передав ей имя файла службы.



После установки службы, проверим, что служба успешно установлена. Это можно проверить в «Компьютер» -> «Управление» -> «Службы».



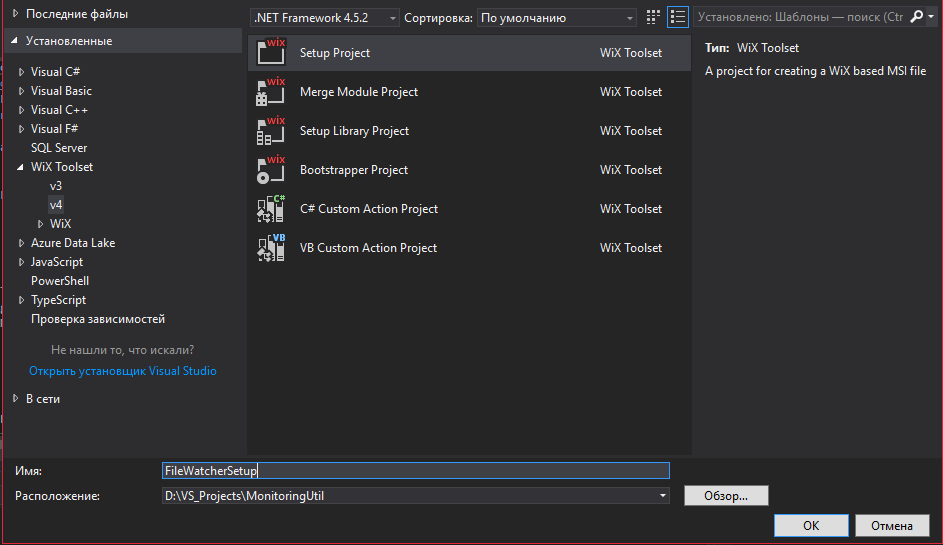
2.2. Создание инсталятора.

В первую очередь, для создания инсталятора необходимо выбрать средство для создания оного. Выбор средств очень широк. Небольшой список приведен ниже:

1. [InstallShield](http://www.installshield.com/) — классика жанра, достаточно солидный проприетарный продукт.
2. [Adnvanced Installer](http://www.advancedinstaller.com/) — проприетарный инструмент с широкими возможностями кастомизации через GUI. На сайте сказано, что если Вы блоггер и будете писать об этом продукте много хороших слов, то у Вас есть возможность получить Free License.
3. [WiX](http://wixtoolset.org/) — открытый бесплатный продукт, основанный на XML-скриптах. Мощная, хорошо документированная штука. К нему есть плагины для Visual Studio, что несомненный плюс.
4. [Inno Setup](http://www.jrsoftware.org/isdl.php) — опенсорсный проект, [код которого доступен на гитхабе](https://github.com/jrsoftware/issrc).

В данной работе был выбран WiX (Windows Installer XML), который представляет собой набор инструментов и спецификаций, упрощающих процесс создания дистрибутивов на базе MSI (Microsoft Installer). В создаваемом установщике определим следующие элементы: Окно приветствия, окно лицензионного соглашения, выбор директории и начало установки.

На сайте WiX можно скачать плагин для Visual Studio (в данном случае для версии 2017). Следующим шагом, добавляем в решение новый проект из раздела WiX Toolset и шаблон Setup Project.



Проект будет состоять из одного файла Product.wxs, с которым мы будем работать. Изначально файл выглядит следующим образом:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Wix xmlns="http://wixtoolset.org/schemas/v4/wxs">  <Product Id="\*" Name="FileWatcherSetup" Language="1033" Version="1.0.0.0" Manufacturer="" UpgradeCode="2d10a4fc-9308-409a-8d14-c65dee928117">  <Package InstallerVersion="200" Compressed="yes" InstallScope="perMachine" />  <MajorUpgrade DowngradeErrorMessage="A newer version of [ProductName] is already installed." />  <MediaTemplate />  <Feature Id="ProductFeature" Title="FileWatcherSetup" Level="1">  <ComponentGroupRef Id="ProductComponents" />  </Feature>  </Product>  <Fragment>  <Directory Id="TARGETDIR" Name="SourceDir">  <Directory Id="ProgramFilesFolder">  <Directory Id="INSTALLFOLDER" Name="FileWatcherSetup" />  </Directory>  </Directory>  </Fragment>  <Fragment>  <ComponentGroup Id="ProductComponents" Directory="INSTALLFOLDER">  </ComponentGroup>  </Fragment>  </Wix> |

Приступая к изменениям, начнем с настройки внешнего вида и добавления русского языка.

* Для добавления русского языка, в блоке <Product>, в свойстве Language, меняем значение с 1033 на 1049.
* В свойствах проекта, на закладке Build, в графе Cultures to build добавляем ru-RU.
* В проект добавляем языковой файл, скачанный заранее.

В созданном проекте нет ни одного диалогового окна. Такое окно можно создать или использовать уже готовые наборы. Воспользуемся готовым окном. Для этого, в конце блока <Product> добавляем следующие строчки:

|  |
| --- |
| <Property Id="WIXUI\_INSTALLDIR" Value="INSTALLLOCATION" ></Property>  <WixVariable Id="WixUILicenseRtf" Overridable="yes" Value="License.rtf"/>  <UIRef Id="WixUI\_InstallDir"/> |

* WixVariable Id=«WixUILicenseRtf» — указывает на путь к файлу лицензии.
* WixUI\_InstallDir — готовый набор диалоговых окон. Данный набор включает все необходимые нам диалоги. Помимо него так же существуют наборы WixUI\_Advanced, WixUI\_Mondo, WixUI\_FeatureTree, WixUI\_InstallDir, WixUI\_Minimal.

Так как некоторые свойства, например, название продукта, будут встречаться ни один раз, логичнее будет объявить переменные для таких свойств. Для этого добавим следующий код перед блоком <Product> и заменим свойства этого блока на переменные:

|  |
| --- |
| <?define ProductName="FileWatcherSetup" ?>  <?define ProductVersion="1.0.0.0" ?>  <?define ProductCode="b7bc7c6f-9a4e-4973-be84-eca8e3427c97"?>  <?define UpgradeCode="06a81104-1e30-463d-87e1-e8a79b4c682a"?>  <?define Manufacturer="ArsenyZorin"?>  <Product Id="$(var.ProductCode)" Name="$(var.ProductName)" Language="1049" Version="$(var.ProductVersion)" Manufacturer="$(var.Manufacturer)" UpgradeCode="$(var.UpgradeCode)"> |

Следующим шагом определим директорию, в которую будет устанавливаться утилита.

Ключ Directory — определяет путь для установки.

Directory Id=«TARGETDIR» корневой элемент для всех папок, которые будут использоваться для установки проекта.

Directory Id=«ProgramFilesFolder» папка Program Files (на что указывает Id=«ProgramFilesFolder»).

Directory Id=«INSTALLLOCATION» папка с именем FileWatcherSetup в папке Program Files. Заменим сразу Name=«FileWatcherSetup» на Name="$(var.ProductName)"

Добавим файлы в пакет установки. Для этого сначала добавим устанавливаемые компоненты в тэг <Component>.

|  |
| --- |
| <Component Id="ProductComponent">  <File Id='Watcher' DiskId='1' Source='D:\VS\_Projects\MonitoringUtil\MonitoringUtil\bin\Debug\MonitoringUtil.exe'/>  </Component> |

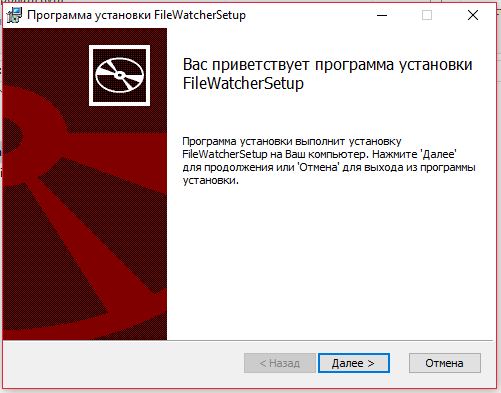
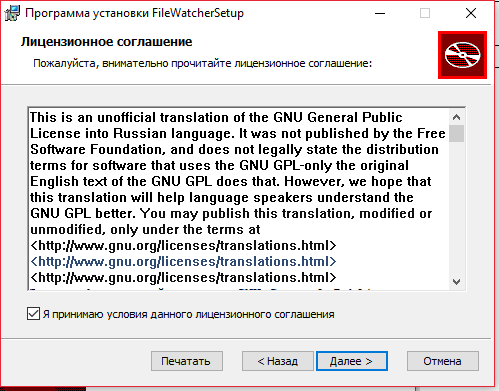
Далее добавляем ярлык в меню Пуск. Сначала указываем, что мы собираемся работать с папкой Пуск и хотим там создать папку с именем нашей утилиты, которая будет содержать ярлык.

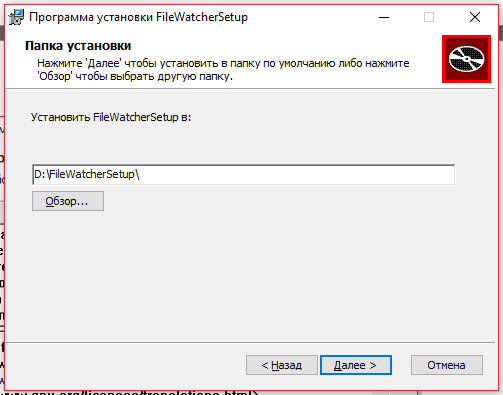
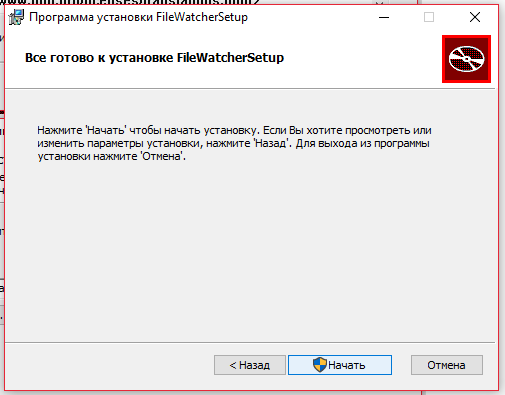
|  |
| --- |
| <Directory Id="ProgramMenuFolder">  <Directory Id="ApplicationProgramsFolder" Name="$(var.ProductName)">  <Component Id="ApplicationShortcutWatcher">  <Shortcut Id="ShortcutWatcher"  Name="Watcher"  Description="$(var.ProductName)"  Target="[INSTALLLOCATION]MonitoringUtil.exe"  WorkingDirectory="INSTALLLOCATION"/>  <RemoveFolder Id="ApplicationProgramsFolder" On="uninstall"/>  <RegistryValue Root="HKCU" Key="Software\$(var.Manufacturer)\$(var.ProductName)" Name="installed" Type="integer" Value="1" KeyPath="yes"/>  </Component>  </Directory>  </Directory> |

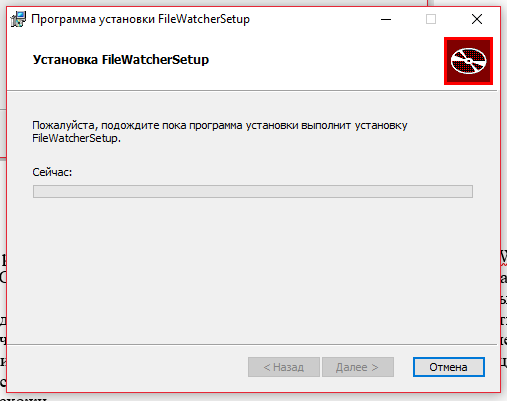
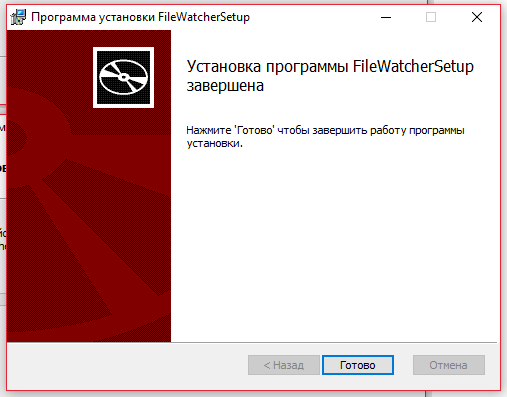
После всех изменений код выглядит следующим образом:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">  <?define ProductName="FileWatcherSetup" ?>  <?define ProductVersion="1.0.0.0" ?>  <?define ProductCode="b7bc7c6f-9a4e-4973-be84-eca8e3427c97"?>  <?define UpgradeCode="06a81104-1e30-463d-87e1-e8a79b4c682a"?>  <?define Manufacturer="ArsenyZorin"?>  <Product Id="$(var.ProductCode)" Name="$(var.ProductName)" Language="1049" Version="$(var.ProductVersion)" Manufacturer="$(var.Manufacturer)" UpgradeCode="$(var.UpgradeCode)">  <Package InstallerVersion="200" Compressed="yes" InstallScope="perMachine" />  <MajorUpgrade DowngradeErrorMessage="A newer version of [ProductName] is already installed." />  <MediaTemplate />  <Feature Id="ProductFeature" Title="FileWatcherSetup" Level="1">  <ComponentRef Id="ProductComponent" />  <ComponentRef Id="ApplicationShortcutWatcher"/>  </Feature>  <Property Id="WIXUI\_INSTALLDIR" Value="INSTALLLOCATION" ></Property>  <WixVariable Id="WixUILicenseRtf" Overridable="yes" Value="D:\VS\_Projects\FileWatcherSetup\gplru.rtf"/>  <UIRef Id="WixUI\_InstallDir"/>    <Directory Id="TARGETDIR" Name="SourceDir">  <Directory Id="ProgramFilesFolder">  <Directory Id="INSTALLLOCATION" Name="$(var.ProductName)">  <Component Id="ProductComponent">  <File Id='Watcher' DiskId='1' Source='D:\VS\_Projects\MonitoringUtil\MonitoringUtil\bin\Debug\MonitoringUtil.exe'/>  </Component>  </Directory>  </Directory>  <Directory Id="ProgramMenuFolder">  <Directory Id="ApplicationProgramsFolder" Name="$(var.ProductName)">  <Component Id="ApplicationShortcutWatcher">  <Shortcut Id="ShortcutWatcher"  Name="Watcher"  Description="$(var.ProductName)"  Target="[INSTALLLOCATION]MonitoringUtil.exe"  WorkingDirectory="INSTALLLOCATION"/>  <RemoveFolder Id="ApplicationProgramsFolder" On="uninstall"/>  <RegistryValue Root="HKCU" Key="Software\$(var.Manufacturer)\$(var.ProductName)" Name="installed" Type="integer" Value="1" KeyPath="yes"/>  </Component>  </Directory>  </Directory>  </Directory>  </Product>  </Wix> |

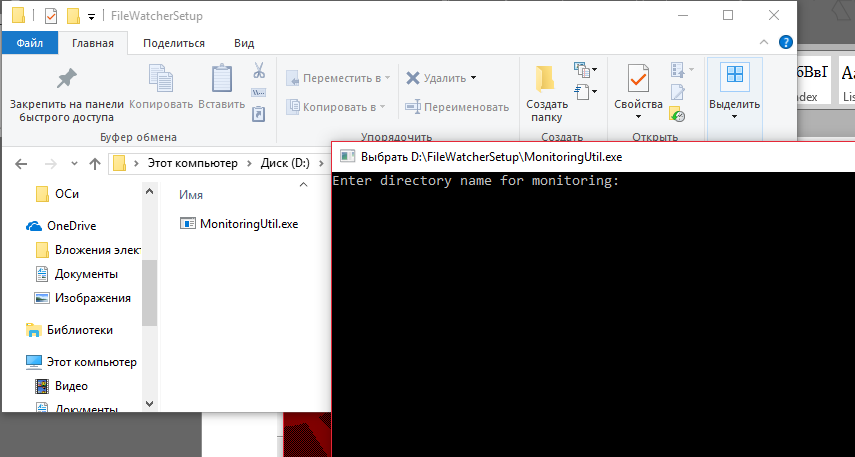
Теперь собираем проект и запускаем исполняемый (.exe) файл. Ниже показаны все этапы установки.

После того, как завершены все этапы установки, в выбранной директории появится исполняемый файл написанной ранее утилиты.



**Вывод:**

В ходе данной лабораторной работы были изучены методы создания и запуска демона на UNIX подобных системах. Также, были изучены методы создания и установки служб и инсталяторов в Windows.

Помимо изучения теоретической части, была произведена практическая реализация. То есть, был создан демон, который работает как консольная утилита, но без захвата консоли. В системе Windows была создана служба, которая, пока она запущена, записывала в файл изменения наблюдаемой директории. И, в дополнение ко всему, был создан инсталятор, который в выбранную директорию устанавливает утилиту.

**Список литературы:**

1. WiX tools http://wixtoolset.org/releases/v3-11-0-1528/
2. Лицензия GPL v3 <http://www.gnu.org/licenses/gpl>
3. Создание инсталятора <https://habrahabr.ru/post/255807/>
4. Создание демона <http://www.opennet.ru/base/dev/daemon_example.txt.html>
5. fork http://www.opennet.ru/docs/RUS/linux\_parallel/node7.html