**Тема**: Создание инсталлируемой версии утилиты в ОС Windows и Linux

**Цель работы**

Ранее для ОС Windows и Linux было реализовано приложение, осуществляющее логирование запуска новых приложений в системе (выводит в файл информацию о времени запуска приложения, пользователе, приоритете процесса и т.д.). Необходимо:

* Создать инсталлируемую версию утилиты для ОС Windows
* Создать инсталлируемую версию утилиты для ОС Linux
* Создать версию утилиты в виде службы Windows
* Создать версию утилиты в виде демона Linux

**Выполнение работы**

**Windows**

**Создание инсталлируемой версии**

Создание инсталлятора Windows выполняется с помощью пакета InstallShield в VisualStuio. Поэтому, первым делом необходимо установить данное расширение среды. Далее, создается новый проект типа InstallShield (рисунок 1).

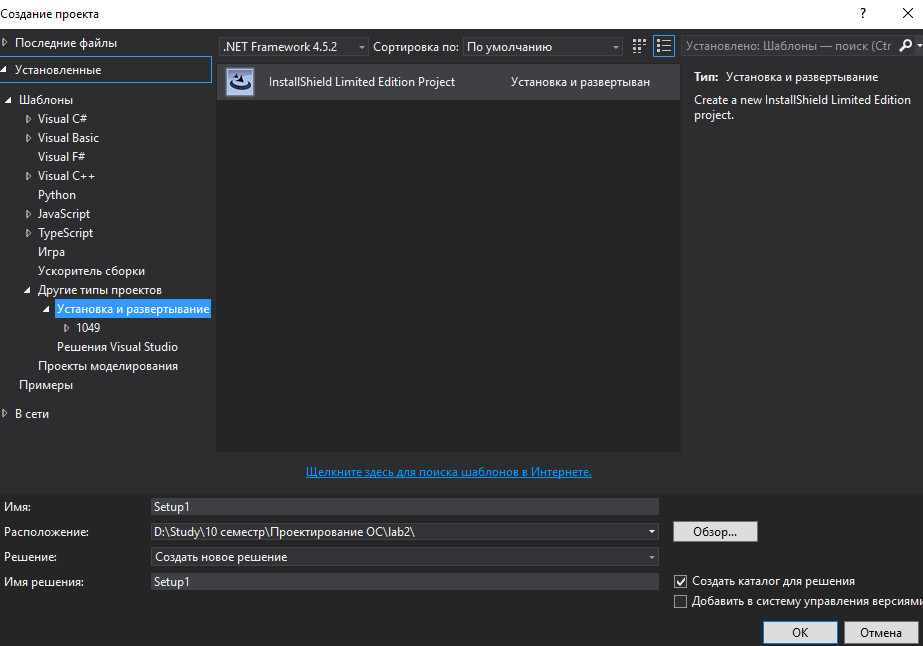


Рис. 1. Создание проекта установщика

После создания проекта появится окно, которое поможет ввести всю необходимую информацию для создания установщика:

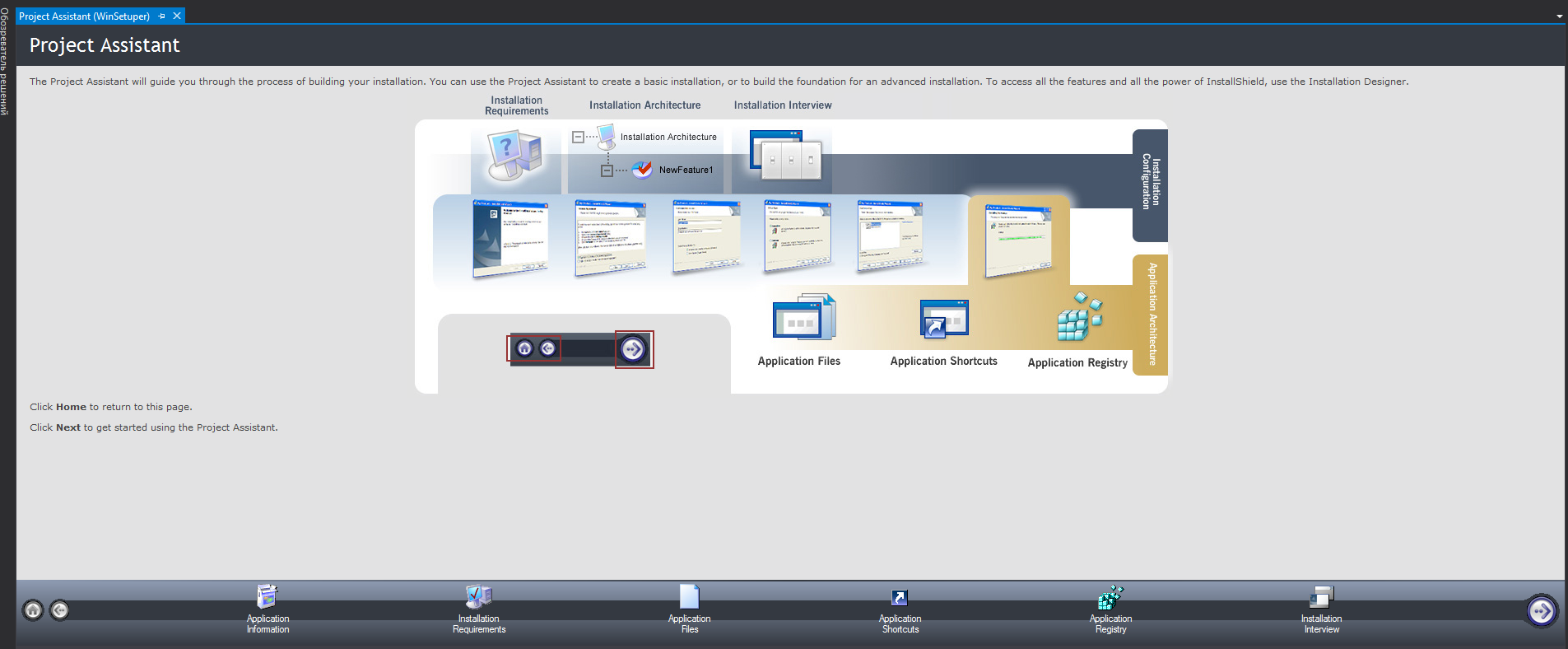


Рис. 2. Проект установщика

Пройдя по очереди по всем пунктам данного проекта вводится вся необходимая информация. Сначала вводится общая информация о приложении:

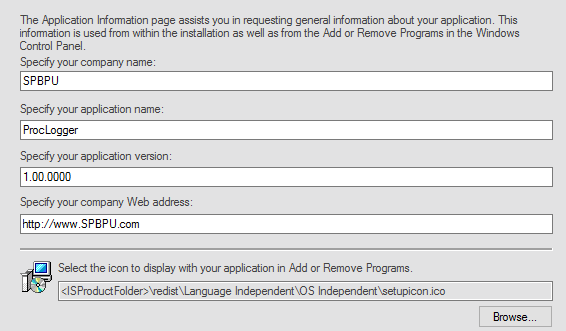


Рис. 3. Окно ввода общей информации о приложении

В следующем окне указываются зависимости устанавливаемого приложения (в данном случае зависимостей нет). В предлагаемом окне указаны только стандартные зависимости, можно добавить свои условия с помощью System Search Wizard.

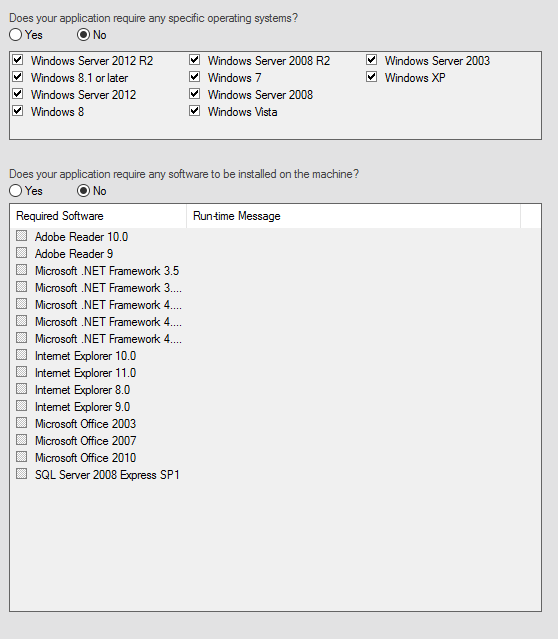


Рис. 4. Определение зависимостей приложения

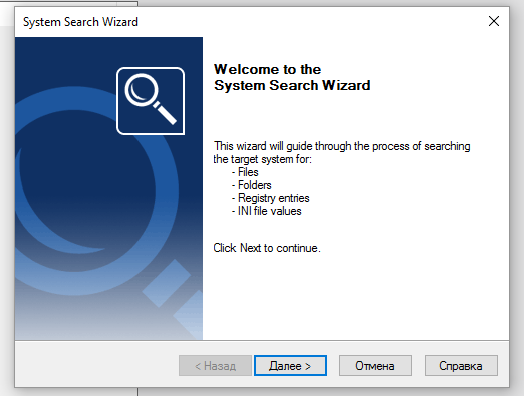


Рис. 5. Добавление зависимостей приложения с помощью System Search Wizard

В следующем окне указываются устанавливаемые файлы приложения (в данном случае это только исполняемый файл).

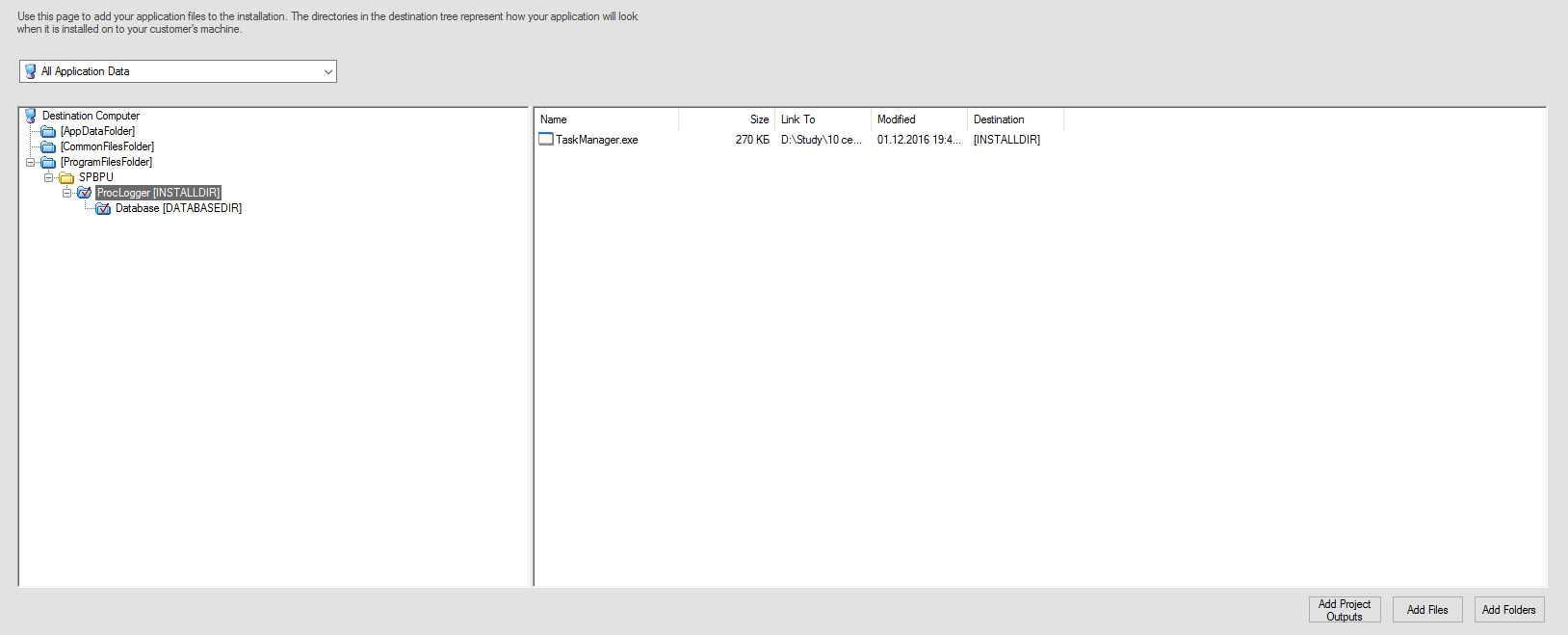


Рис. 5. Добавление файлов приложения

Далее создаются ярлыки приложения. В данном случае было создано два ярлыка: для запуска приложения и для его удаления.

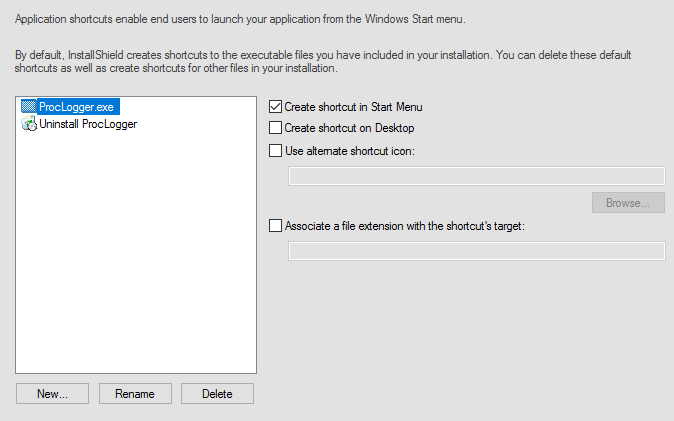


Рис. 6. Создание ярлыков

Далее открывается окно в котором можно указать зависимости приложения от определенных записей в реестре Windows (в данном случае зависимостей нет).

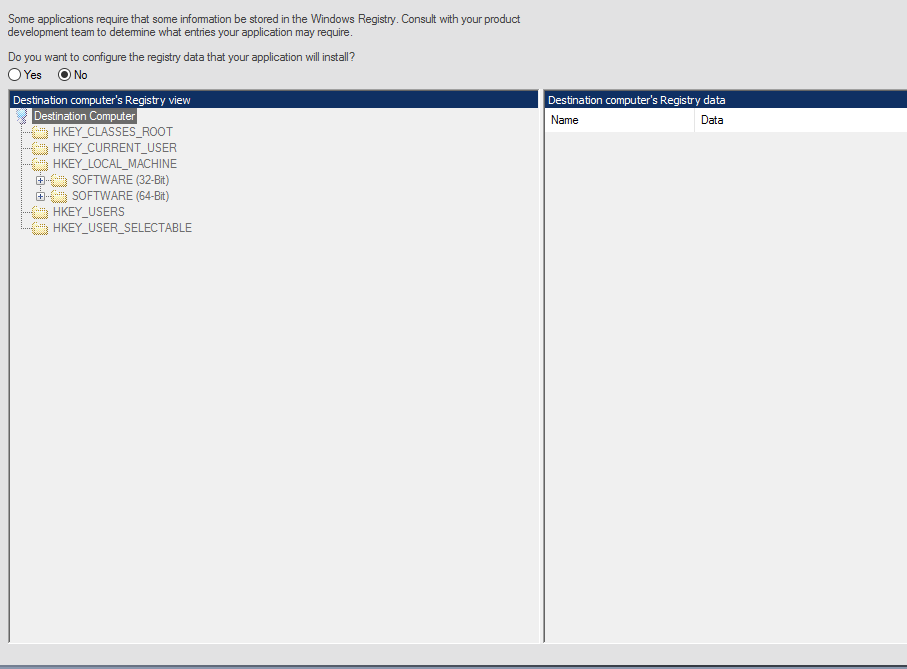


Рис. 7. Окно редактора реестра

В последнем окне можно настроить различные параметры инсталлятора: необходимо ли показывать лицензию, должен ли пользователь указывать свои личные данные, может ли пользователь изменить папку установки, добавить ли в установщик возможность сразу запустить приложение.

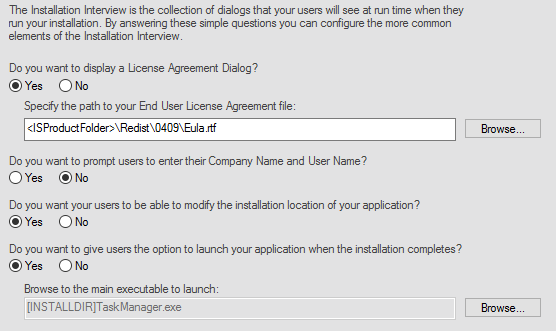


Рис. 8. Настройка инсталлятора

После указания всех этих действий, можно собрать проект установщика. Перед сборкой можно выбрать тип результирующего образа инсталлятора: CD\_ROM, DVD5 или Single Image. В зависимости от типа образа в соответствующей папке появится инсталлятор Setup.exe. Процесс установки утилиты представлен на рисунках 9-13.

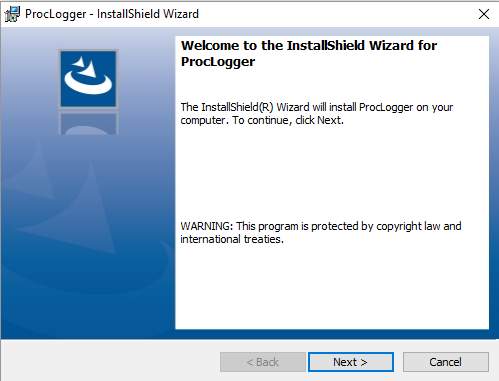


Рис. 9. Окно инсталлятора

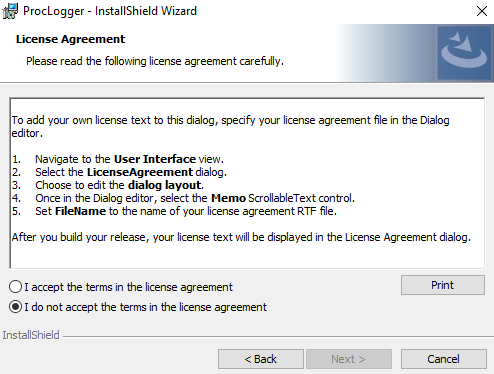


Рис. 10. Окно инсталлятора

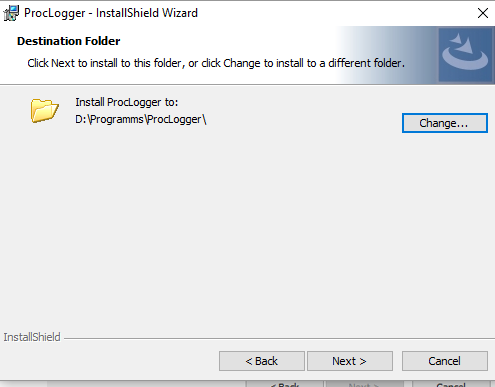


Рис. 11. Окно инсталлятора

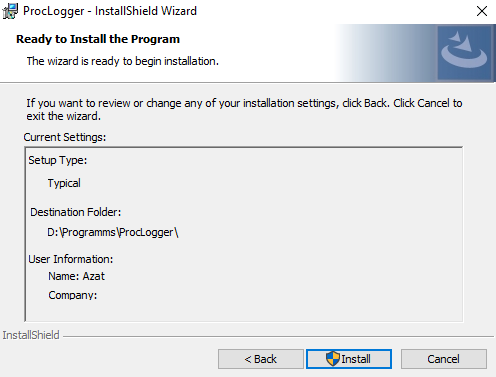


Рис. 12. Окно инсталлятора

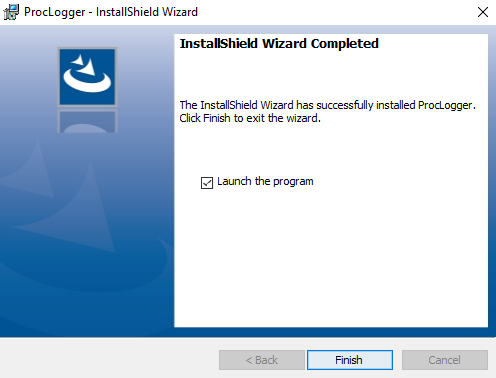


Рис. 13. Окно инсталлятора

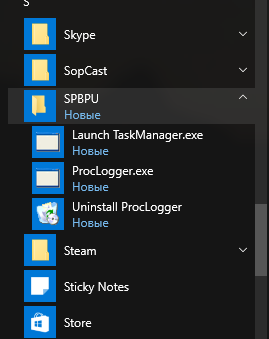


Рис. 14. Установленное приложение

Необходимо упомянуть о том, что описанный инструмент – не единственный способ создания своего инсталлятора. Существует так же множество других способов, как от Microsoft, так и от сторонних производителей.

Инсталляторы от Microsoft можно объединить под одной технологией – Windows Installer. Примеры данных технологий установщиков:

* Описанный InstallShield
* WiX (Windows installer XML) - набор инструментов, позволяющих создавать инсталляционные пакеты Windows Installer (.MSI и .MSM) на основе XML-описаний. Программы из набора WiX предназначены для работы в командной строке, однако существует плагин Votive, который позволяет работать с исходными кодами инсталлятора как с проектом Visual Studio (проектный файл имеет расширение .wixproj). При этом поддерживаются обычные возможности проектов: подсветка синтаксиса, построение из командной строки, IntelliSense и т. д.
* Wise InstallMaster 8.1 – инструмент от компании Wise Solutions, обладает такой же функциональностью, как и InstallShield
* Inno Setup — система создания инсталляторов для Windows программ с открытым исходным кодом. Впервые выпущенный в 1997 году, Inno Setup сегодня конкурирует и даже превосходит многие коммерческие установщики по функциональности и стабильности
* Smart Install Maker – это мощное и удобное средство для создания инсталляторов. Программа обладает удобным и интуитивно понятным интерфейсом, а также полным набором необходимых функций для создания профессиональных инсталляторов с минимальным размером, высокой степенью сжатия файлов и приятным интерфейсом

**Создание службы Windows**

Службы ОС Windows — приложения, автоматически (если настроено) запускаемые системой при запуске Windows и выполняющиеся вне зависимости от статуса пользователя. Имеет общие черты с концепцией демонов в Unix.

Для создания службы необходимо в VisualStudio создать новый проект и выбрать шаблон Visual C# -> Windows -> Служба Windows. Visual Studio не имеет шаблона для создания службы на C++, только C# или Basic.

Создав проект, откроем его основной файл. Добавим в него необходимые нам действия: запуск нашей утилиты на старте и ее уничтожение при остановке:

namespace ProcService

{

public partial class ProcService : ServiceBase

{

string name = "TaskManager.exe";

string path = "D:\\Study\\10 семестр\\Проектирование ОС\\WinUtil\\TaskManager\\TaskManager\\";

string log = "C:\\log.txt";

public ProcService()

{

InitializeComponent();

}

protected override void OnStart(string[] args)

{

Process.Start(path + name, log);

}

protected override void OnStop()

{

System.Diagnostics.Process[] etc = System.Diagnostics.Process.GetProcesses();//получим процессы

foreach (System.Diagnostics.Process anti in etc)//обойдем каждый процесс

if (anti.ProcessName.ToLower().Contains(name.ToLower())) anti.Kill();

}

}

}

Далее, добавим установщик в службу:

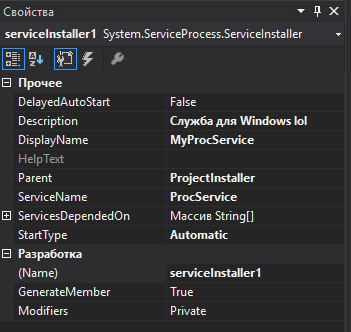


Рис. 15. Установщик службы

В установщике мы задали необходимые параметры: имя службы, ее описание, тип запуска и т.д. Далее в свойствах проекта мы выбираем автоматически запускаемый объект (созданный ранее класс):

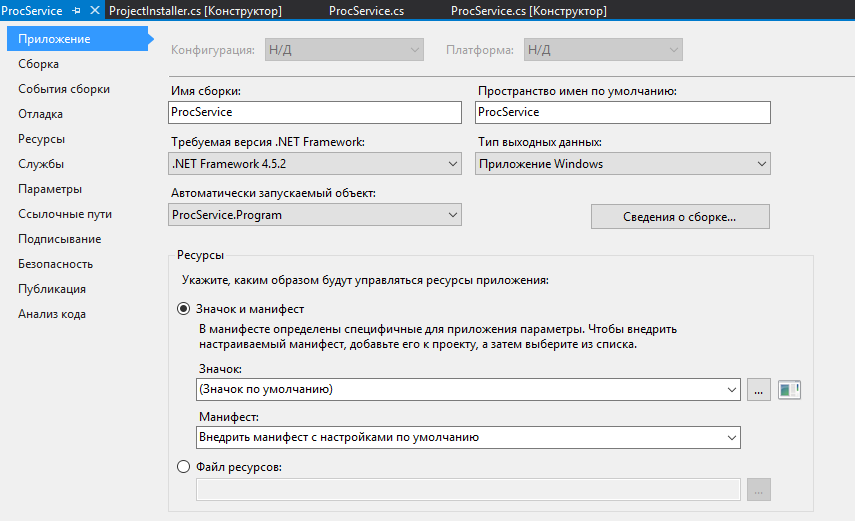


Рис. 16. Свойства проекта

Для установки утилиты необходимо запустить командную строку разработчика от имени администратора, перейти в папку с проектом службы и выполнить команду: installutil.exe ServiceName, где ServiceName – это имя исполняемого файла созданной службы.

D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug>installutil.exe ProcService.exe

Microsoft (R) .NET Framework Installation utility, версия 4.6.1586.0

c Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

Выполняется групповая операция установки.

Начинается этап установки процедуры установки.

См. файл журнала выполнения операция для сборки D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe.

Данный файл находится в D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.InstallLog.

Выполняется установка сборки 'D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe'.

Затронуты следующие параметры:

logtoconsole =

logfile = D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.InstallLog

assemblypath = D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe

Устанавливается служба ProcService...

Служба ProcService успешно установлена.

Создается исходный EventLog ProcService в журнале Application...

Этап установки успешно выполнен, начинается этап фиксации.

См. файл журнала выполнения операция для сборки D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe.

Данный файл находится в D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.InstallLog.

Выполняется фиксация сборки 'D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe'.

Затронуты следующие параметры:

logtoconsole =

logfile = D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.InstallLog

assemblypath = D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe

Этап фиксации выполнен успешно.

Групповая операция установки выполнена.

Далее, открыв менеджер служб Windows, мы можем увидеть в нем созданную службу.

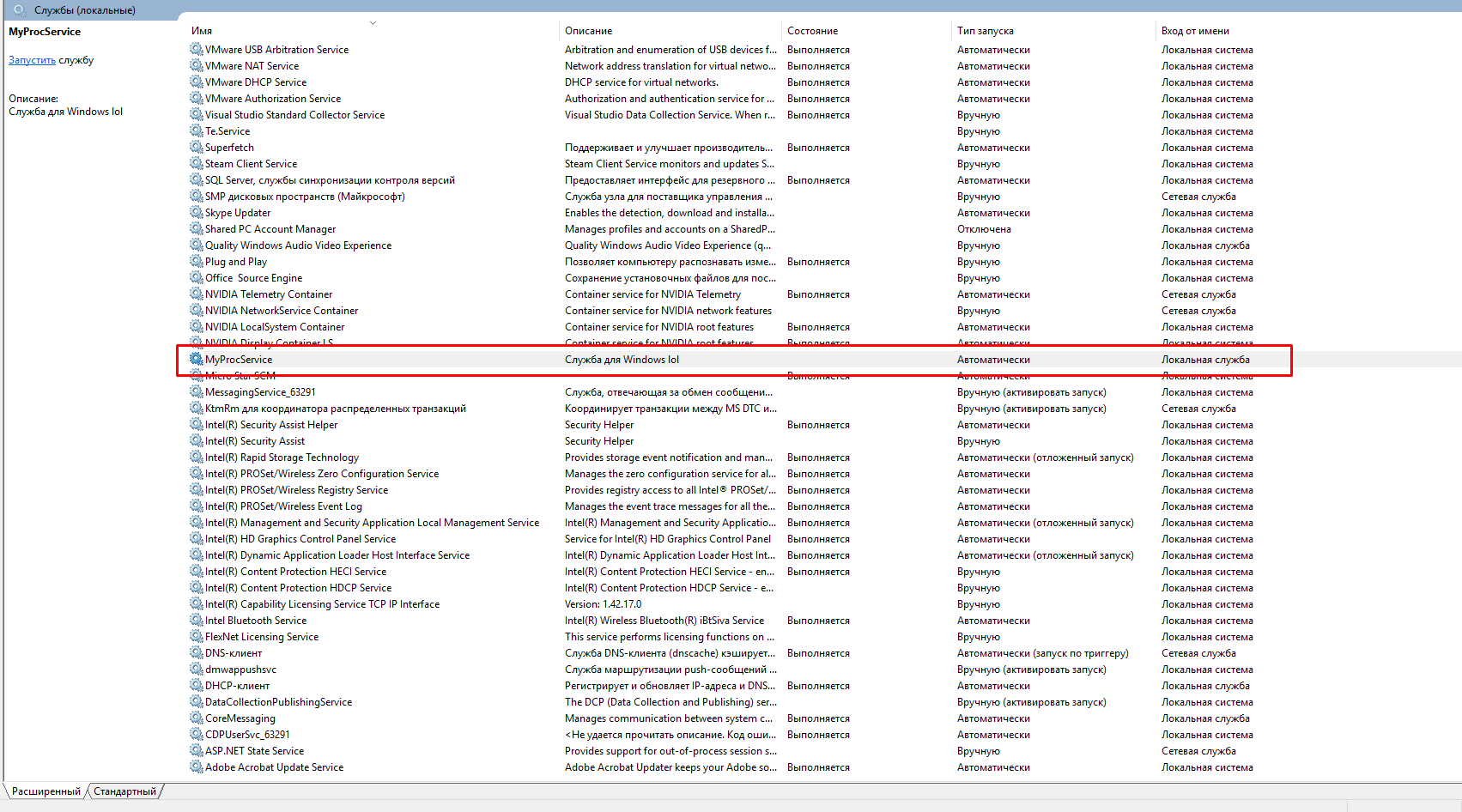


Рис. 17. Запущенная служба

Для удаления службы можно так же воспользоваться утилитой installutil.exe.

D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug>installutil.exe /u ProcService.exe

Microsoft (R) .NET Framework Installation utility, версия 4.6.1586.0

c Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

Начинается установка.

См. файл журнала выполнения операция для сборки D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe.

Данный файл находится в D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.InstallLog.

Выполняется удаление сборки 'D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe'.

Затронуты следующие параметры:

logtoconsole =

logfile = D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.InstallLog

assemblypath = D:\Study\10 семестр\Проектирование ОС\lab2\ProcService\ProcService\bin\Debug\ProcService.exe

Выполняется удаление источника EventLog ProcService.

Выполняется удаление службы ProcService из системы...

Служба ProcService успешно удалена из системы.

Удаление из системы выполнено.

Для создания сервиса на С++ в Visual Studio нет отдельного типа проекта, необходимо создавать все с нуля в качестве обычного консольного приложения используя WinAPI.

Основной структурой при создании сервиса является:

typedef struct \_SERVICE\_TABLE\_ENTRY {

LPTSTR                  lpServiceName;

LPSERVICE\_MAIN\_FUNCTION lpServiceProc;

} SERVICE\_TABLE\_ENTRY, \*LPSERVICE\_TABLE\_ENTRY;

Она содержит имя сервиса и указатель на main-функцию сервиса. Для регистрации нового сервиса необходимо создать данную структуру и передать ее в качестве аргумента функции StartServiceCtrlDispatcher, которая связывает наш сервис с Service Control Manager. Для контроля состояния сервиса используется структура:

typedef struct \_SERVICE\_STATUS {

DWORD dwServiceType;

DWORD dwCurrentState;

DWORD dwControlsAccepted;

DWORD dwWin32ExitCode;

DWORD dwServiceSpecificExitCode;

DWORD dwCheckPoint;

DWORD dwWaitHint;

} SERVICE\_STATUS, \*LPSERVICE\_STATUS;

Исходный код main-функции сервиса приведен ниже:

void ServiceMain(int argc, char\*\* argv) {

int error;

int i = 0;

serviceStatus.dwServiceType = SERVICE\_WIN32\_OWN\_PROCESS;

serviceStatus.dwCurrentState = SERVICE\_START\_PENDING;

serviceStatus.dwControlsAccepted = SERVICE\_ACCEPT\_STOP | SERVICE\_ACCEPT\_SHUTDOWN;

serviceStatus.dwWin32ExitCode = 0;

serviceStatus.dwServiceSpecificExitCode = 0;

serviceStatus.dwCheckPoint = 0;

serviceStatus.dwWaitHint = 0;

serviceStatusHandle = RegisterServiceCtrlHandler(serviceName, (LPHANDLER\_FUNCTION)ControlHandler);

if (serviceStatusHandle == (SERVICE\_STATUS\_HANDLE)0) {

return;

}

error = 0;

if (error) {

serviceStatus.dwCurrentState = SERVICE\_STOPPED;

serviceStatus.dwWin32ExitCode = -1;

SetServiceStatus(serviceStatusHandle, &serviceStatus);

return;

}

serviceStatus.dwCurrentState = SERVICE\_RUNNING;

SetServiceStatus(serviceStatusHandle, &serviceStatus);

while (serviceStatus.dwCurrentState == SERVICE\_RUNNING)

{

//Свой код

char buffer[255];

sprintf\_s(buffer, "%u", i);

int result = addLogMessage(buffer);

if (result) {

serviceStatus.dwCurrentState = SERVICE\_STOPPED;

serviceStatus.dwWin32ExitCode = -1;

SetServiceStatus(serviceStatusHandle, &serviceStatus);

return;

}

i++;

}

return;

}

Данная функция достаточно проста: сначала заполняются поля структуры SERVICE\_STATUS, далее регистрируется функция, которая будет обрабатывать управляющие запросы от SCM (в данном случае ControlHandler). Регистрация производится с помощью функции RegisterServiceCtrlHandler. Далее в функцию можно вставить свой код. В данном случае программа проверяет статус сервиса, и если статус «RUNNING», то выводит в файл значение счетчика. Функция ControlHandler:

void ControlHandler(DWORD request) {

switch (request)

{

case SERVICE\_CONTROL\_STOP:

addLogMessage("Stopped.");

serviceStatus.dwWin32ExitCode = 0;

serviceStatus.dwCurrentState = SERVICE\_STOPPED;

SetServiceStatus(serviceStatusHandle, &serviceStatus);

return;

case SERVICE\_CONTROL\_SHUTDOWN:

addLogMessage("Shutdown.");

serviceStatus.dwWin32ExitCode = 0;

serviceStatus.dwCurrentState = SERVICE\_STOPPED;

SetServiceStatus(serviceStatusHandle, &serviceStatus);

return;

default:

break;

}

SetServiceStatus(serviceStatusHandle, &serviceStatus);

return;

}

Установку, запуск и удаление сервиса можно так же выполнять программу. Ниже приведены функции, которые выполняют это:

int InstallService() {

SC\_HANDLE hSCManager = OpenSCManager(NULL, NULL, SC\_MANAGER\_CREATE\_SERVICE);

if (!hSCManager) {

addLogMessage("Error: Can't open Service Control Manager");

return -1;

}

SC\_HANDLE hService = CreateService(

hSCManager,

serviceName,

serviceName,

SERVICE\_ALL\_ACCESS,

SERVICE\_WIN32\_OWN\_PROCESS,

SERVICE\_DEMAND\_START,

SERVICE\_ERROR\_NORMAL,

servicePath,

NULL, NULL, NULL, NULL, NULL

);

if (!hService) {

int err = GetLastError();

switch (err) {

case ERROR\_ACCESS\_DENIED:

addLogMessage("Error: ERROR\_ACCESS\_DENIED");

break;

case ERROR\_CIRCULAR\_DEPENDENCY:

addLogMessage("Error: ERROR\_CIRCULAR\_DEPENDENCY");

break;

case ERROR\_DUPLICATE\_SERVICE\_NAME:

addLogMessage("Error: ERROR\_DUPLICATE\_SERVICE\_NAME");

break;

case ERROR\_INVALID\_HANDLE:

addLogMessage("Error: ERROR\_INVALID\_HANDLE");

break;

case ERROR\_INVALID\_NAME:

addLogMessage("Error: ERROR\_INVALID\_NAME");

break;

case ERROR\_INVALID\_PARAMETER:

addLogMessage("Error: ERROR\_INVALID\_PARAMETER");

break;

case ERROR\_INVALID\_SERVICE\_ACCOUNT:

addLogMessage("Error: ERROR\_INVALID\_SERVICE\_ACCOUNT");

break;

case ERROR\_SERVICE\_EXISTS:

addLogMessage("Error: ERROR\_SERVICE\_EXISTS");

break;

default:

addLogMessage("Error: Undefined");

}

CloseServiceHandle(hSCManager);

return -1;

}

CloseServiceHandle(hService);

CloseServiceHandle(hSCManager);

addLogMessage("Success install service!");

return 0;

}

int RemoveService() {

SC\_HANDLE hSCManager = OpenSCManager(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS);

if (!hSCManager) {

addLogMessage("Error: Can't open Service Control Manager");

return -1;

}

SC\_HANDLE hService = OpenService(hSCManager, serviceName, SERVICE\_STOP | DELETE);

if (!hService) {

addLogMessage("Error: Can't remove service");

CloseServiceHandle(hSCManager);

return -1;

}

DeleteService(hService);

CloseServiceHandle(hService);

CloseServiceHandle(hSCManager);

addLogMessage("Success remove service!");

return 0;

}

int StartService() {

SC\_HANDLE hSCManager = OpenSCManager(NULL, NULL, SC\_MANAGER\_CREATE\_SERVICE);

SC\_HANDLE hService = OpenService(hSCManager, serviceName, SERVICE\_START);

if (!StartService(hService, 0, NULL)) {

CloseServiceHandle(hSCManager);

addLogMessage("Error: Can't start service");

return -1;

}

CloseServiceHandle(hService);

CloseServiceHandle(hSCManager);

return 0;

}

Работа этих функций основана на использовании функций OpenSCManager и OpenService. OpenSCManager получает handle на service control manager на указанной машине и устанавливает связь с ним. Далее с помощью функции OpenService можно получить handle на определенный сервис указанного SCM. В качестве третьего аргумента данной функции можно указать желаемый способ доступа к сервису, в том числе его создание, запуск или удаление.

Функция main:

int \_tmain(int argc, \_TCHAR\* argv[]) {

servicePath = LPTSTR(argv[0]);

if (argc - 1 == 0) {

SERVICE\_TABLE\_ENTRY ServiceTable[1];

ServiceTable[0].lpServiceName = serviceName;

ServiceTable[0].lpServiceProc = (LPSERVICE\_MAIN\_FUNCTION)ServiceMain;

if (!StartServiceCtrlDispatcher(ServiceTable)) {

addLogMessage("Error: StartServiceCtrlDispatcher");

}

}

else if (wcscmp(argv[argc - 1], \_T("install")) == 0) {

InstallService();

}

else if (wcscmp(argv[argc - 1], \_T("remove")) == 0) {

RemoveService();

}

else if (wcscmp(argv[argc - 1], \_T("start")) == 0) {

StartService();

}

}

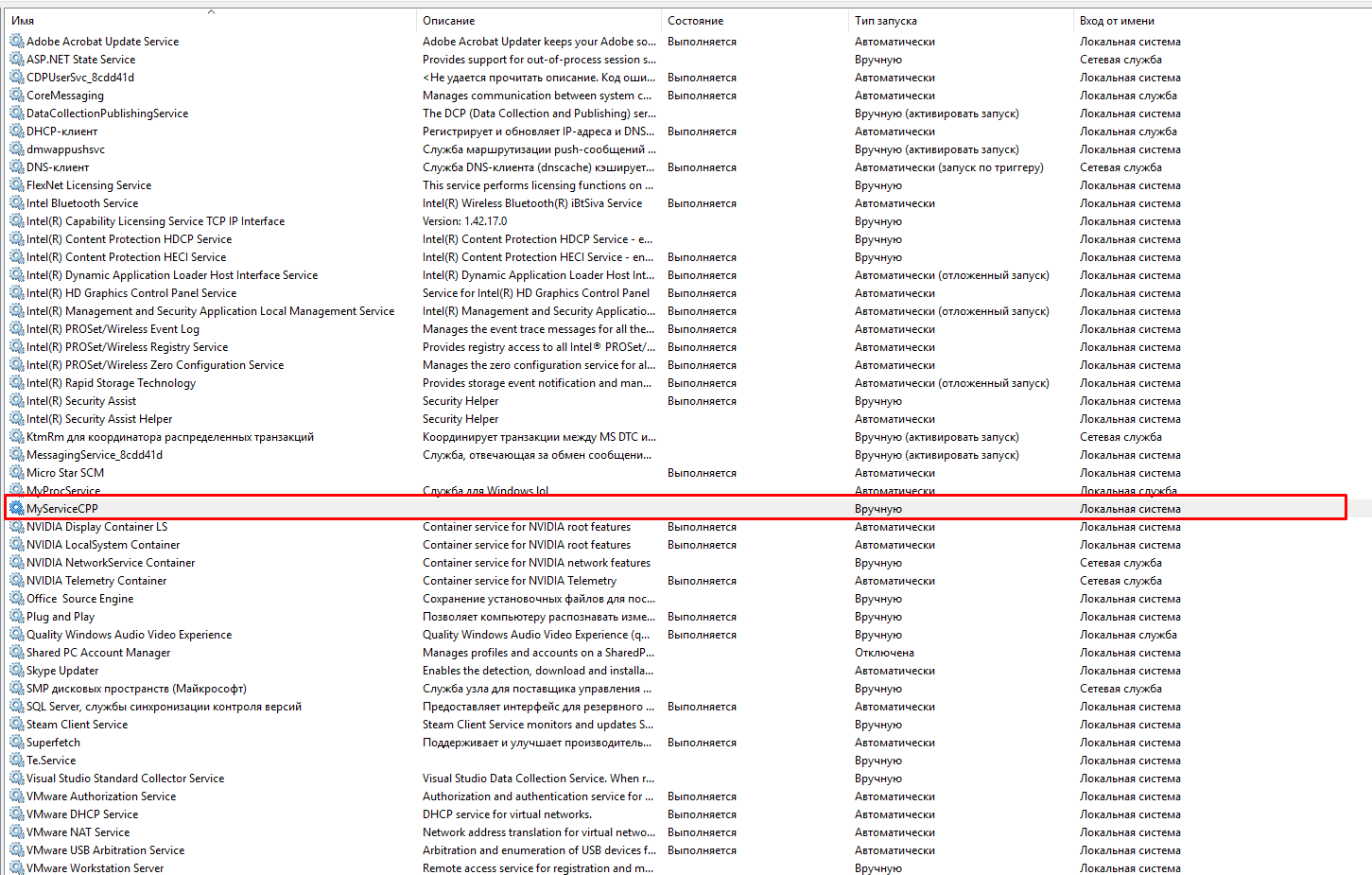
Собрав проект, мы можем устанавливать и запускать сервис из консоли:

D:\Study\10 семестр\Проектирование ОС\lab2\Service\Debug>Service.exe install

D:\Study\10 семестр\Проектирование ОС\lab2\Service\Debug>Service.exe start

D:\Study\10 семестр\Проектирование ОС\lab2\Service\Debug>Service.exe remove

После установки сервис отображается в менеджере сервисов Windows:



**Linux**

**Создание инсталлируемой версии**

В данной работе рассматривался процесс создания .deb пакета для Ubuntu 16.01. Формат .deb пакетов позволяет устанавливать в систему различные программы и библиотеки как в виде бинарных файлов, так и из исходных файлов. Рассмотрим процесс создания пакета с бинарным файлом.

Создадим новый каталог и заполним его следующим содержимым:  
kivi@kivi-VirtualBox:~/sp\_labs/proc-logger\_1.0-1$ tree.

├── DEBIAN

│   ├── control

│   └── copyright

└── usr

└── local

└── bin

└── proc-logger

4 directories, 3 files

Рассмотрим этот каталог более подробно. Имя самого каталога – proc-logger\_1.0-1 это имя пакета + версия. В нем содержится два подкаталога: DEBIAN и usr. В каталоге DEBIAN содержатся файлы с описанием пакета. Каталог usr содержит в себе устанавливаемые бинарные файлы, они расположены так же, как будут расположены после установки в систему. В данном случае мы устанавливаем единственный бинарный файл proc-logger, который будет лежать в /usr/local/bin/.

Файл DEBIAN/control – это файл с описанием пакета. Данный файл обязателен. Рассмотрим его содержимое:

kivi@kivi-VirtualBox:~/sp\_labs/proc-logger\_1.0-1$ cat DEBIAN/control

Package: proc-logger

Version: 1.0-1

Architecture: i386

Section: misc

Maintainer: Azat Abdullin <azat.aam@gmail.com>

Description: Process Logger

Program loggs information about all new processes

.

Long description :)

Рассмотрим его содержимое:

* Package – имя пакета, которое будет использоваться менеджером пакетов
* Version – версия программы и пакета, используется для обновлений пакетов и разрешения зависимостей
* Architecture – архитектура, под которую был собран бинарный файл. Допустимые значения: i386, amd64, all, source. all используется для скриптов. source используется для компилируемых пакетов с исходниками
* Sectoin – тип задачи, для которой используется приложение.
* Description – описание пакета

Так же здесь можно указывать: зависимости пакета, размер пакета и различная дополнительная информация.

Так же можно добавлять дополнительные файлы для пакета:

* copyright – файл с описанием лицензии
* changelog – история изменений
* rules – правила компиляции если в пакете исходные файлы
* conffiles – список файлов конфигурации
* dirs – список создаваемых директорий
* menu – файл с описанием пунктов меню
* md5sums – контрольные суммы файлов
* watch – мониторинг сайта программы

Создать пакет и установить его можно будет с помощью утилиты dpkg:

kivi@kivi-VirtualBox:~/sp\_labs$ dpkg-deb --build proc-logger\_1.0-1/

dpkg-deb: building package 'proc-logger' in 'proc-logger\_1.0-1.deb'.

kivi@kivi-VirtualBox:~/sp\_labs$ sudo dpkg --install proc-logger\_1.0-1.deb

Selecting previously unselected package proc-logger:i386.

(Reading database ... 304380 files and directories currently installed.)

Preparing to unpack proc-logger\_1.0-1.deb ...

Unpacking proc-logger:i386 (1.0-1) ...

Setting up proc-logger:i386 (1.0-1) ...

kivi@kivi-VirtualBox:~/sp\_labs$ proc

proc proc-logger

kivi@kivi-VirtualBox:~/sp\_labs$ proc-logger log.out

Printing results into: log.out

exit

Как мы видим, утилита установлена и нормально работает. Удаление происходит так же с помощью менеджера пакетов.

Для создания установочных пакетов для других систем Linux так же существуют свои инструменты. Их список зависит от используемой системы, а точнее от пакетного менеджера. Например, в системе Arch Linux используется пакетный менеджер pacman. Вся информация для создания пакета помещена в файл PKGBUILD. Когда вы запускаете makepkg, он ищет PKGBUILD в текущей рабочей директории и затем собирает приложение из исходного кода, следуя инструкциям в PKGBUILD. После успешной компиляции полученные бинарные файлы, как и вся необходимая метаинформация (такие как версия пакета и зависимости), архивируются в пакетный файл имя.pkg.tar.gz, который легко может быть установлен при помощи команды pacman -Up <имя\_файла\_пакета>. Файл PKGBUILD содержит все инструкции для создания пакета, которые напрямую интерпретируется оболочкой bash (не бойтесь, если ничего не поняли). Переменные, используемые здесь, определены в статье ABS, но наиболее важные/сложные переменные также описаны и здесь. Чтобы создать новый пакет, сперва вы должны создать пустую директорию; предпочтительней назвать ее /var/abs/local/<ИМЯ\_ПАКЕТА>.

Таким образом, можно сказать что механика создания пакета для Arch похожа на Debian, различия только в деталях.

**Создание демона Linux**

Демон — компьютерная программа в системах класса UNIX, запускаемая самой системой и работающая в фоновом режиме без прямого взаимодействия с пользователем. Исходный код основного процесса демона:

int workingProcess() {

table.setUpLogger(logFile);

for(;;) {

table.updateTable();

sleep(1);

}

return 0;

}

Данный процесс просто раз в секунду обновляет таблицу процессов и выводит информацию о них в указанный лог файл.

Исходный код процесса-монитора, который следит за текущим состоянием процесса-демона и перезапускает его при необходимости. В данную функцию можно добавить дополнительные обработчики сигналов при необхдимости.

int monitorProcess() {

int pid;

int status;

sigset\_t sigset;

siginfo\_t siginfo;

// настраиваем сигналы которые будем обрабатывать

sigemptyset(&sigset);

// сигнал остановки процесса пользователем

sigaddset(&sigset, SIGQUIT);

// сигнал для остановки процесса пользователем с терминала

sigaddset(&sigset, SIGINT);

// сигнал запроса завершения процесса

sigaddset(&sigset, SIGTERM);

// сигнал посылаемый при изменении статуса дочернего процесса

sigaddset(&sigset, SIGCHLD);

sigprocmask(SIG\_BLOCK, &sigset, NULL);

// данная функция создаст файл с нашим PID'ом

setPidFile(PID\_FILE);

// бесконечный цикл работы

for (;;) {

// создаём потомка

pid = fork();

if (pid == -1) {

break;

} else if (!pid) {// если мы потомок

// данный код выполняется в потомке

// запустим функцию отвечающую за работу демона

status = workingProcess();

// завершим процесс

exit(status);

} else {// если мы родитель

// данный код выполняется в родителе

// ожидаем поступление сигнала

sigwaitinfo(&sigset, &siginfo);

// если пришел какой-либо ожидаемый сигнал

// убьем потомка

kill(pid, SIGKILL);

status = 0;

break;

}

}

// удалим файл с PID'ом

unlink(PID\_FILE.c\_str());

return status;

}

Функция main, которая запускает демон и завершает свою работу.

int main(int argc, char\*\* argv) {

int status;

int pid;

// если параметров командной строки меньше двух, то покажем как использовать демона

if (argc != 2) {

std::cout << "Usage: ./proc-logger conffile" << std::endl;

return -1;

}

logFile = std::string(argv[1]);

// создаем потомка

pid = fork();

if (pid == -1) {// если не удалось запустить потомка

// выведем на экран ошибку и её описание

std::cout << "Error: Start Daemon failed " << strerror(errno) << std::endl;

return -1;

} else if (!pid) {// если это потомок

// данный код уже выполняется в процессе потомка

// разрешаем выставлять все биты прав на создаваемые файлы,

// иначе у нас могут быть проблемы с правами доступа

umask(0);

// создаём новый сеанс, чтобы не зависеть от родителя

setsid();

// переходим в корень диска, если мы этого не сделаем, то могут быть проблемы.

// к примеру с размантированием дисков

chdir("/");

// закрываем дискрипторы ввода/вывода/ошибок, так как нам они больше не понадобятся

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);

// Данная функция будет осуществлять слежение за процессом

status = monitorProcess();

return status;

}

else {// если это родитель

// завершим процес, т.к. основную свою задачу (запуск демона) мы выполнили

return 0;

}

}

**Вывод**

В данной работе мы познакомились с возможностями ОС Windows и Linux для создания и установки программ. Сначала были изучены основы создания инсталляторов для Windows. Пакет InstallShield для Visual Studio обладает большими возможностями для создания и настройки инсталляторов. В простейшем случае данный инсталлятор просто копирует исполняемый файл программы в указанную папку.

Далее были изучены основы создания служб в Windows. Службы создаются с помощью Visual Studio. Минусом является то, что в Visual Studio нет встроенных инструментов для создания служб на С++.

Далее были изучены основы создания установочных пакетов для Ubuntu. Изученная система позволяет создавать пакеты как из бинарных файлов, так и компилировать эти бинарные файлы из исходных файлов во время установки.

**Список использованных источников**

1. Документация WinAPI и Visual Studio: <https://msdn.microsoft.com/library>
2. Wiki <https://wiki.debian.org/ru/dpkg>