Operating systems. Lecture 14

Michał Goliński

2019-01-15

Introduction

Recall

- Linux distiributions – differences and similarities
- Administering a Linux system
  - Package manager
  - Managing services
  - Networking basics
- Overview of the most important tools

Plan for today

- Architecture of Windows
- PowerShell

History of Windows

MS-DOS

In the early 1980 IBM approached Microsoft to license its popular BASIC compiler for the new IBM PC. Microsoft suggested CP/M from the company Digital Research as the operating system, but the deal did not go through. IBM returned to Microsoft, who bought a CP/M clone, relicensed it and sold it to IBM. The first version of MS-DOS (Disk Operating System) was born.

Windows on DOS

Inspired by the graphical user interface, Microsoft in 1990 released Windows 3.0 (the previous version were not very successful). It was not a true operating system, just a graphical shell that ran inside MS-DOS. All programs ran in the same address space: a bug in any of them could have brought the system to halt.

In 1995 the Windows 95 was released which had many features a full-fledged operating system, e.g., virtual memory, process management, and 32-bit programming interfaces. It

still lacked security: processes were not well isolated from the operating system and hence it was not generally very stable.

Windows NT

In parallel to developing DOS-based Windows version Microsoft began to create a new operating system from scratch. The first version, called Windows NT 3.1 was released in 1993. It had supported two types of API: IBM OS/2 and Win32 – a 32-bit extension of the API used by Windows 3.0. This version was targetted mostly at servers: the DOS-based Windows versions had a much better compatibility with many existing 16-bit programs.

Windows NT cont.

NT was further refined into NT 4.0 and Windows 2000. The first mainstream consumer targeted version of Windows NT was Windows NT (NT version 5.1) released in 2001. Because it supported the same Win32 API as Windows 9x it had very good compatibility with existing software. In fact Microsoft had a dedicated team whose only task was to ensure compatibility (including replicating undocumented behaviour of older versions if some software relied on that behaviour).

Windows NT cont.

The next version of Windows: Vista was widely regarded as bloated and the release did not get the adoption Microsoft hoped for. The shortcomings were remedied in Windows 7 improving performance and user experience even if it had relatively few functional changes compared to Vista.

Modern Windows

Computing industry changed again: a push for mobile but still quite popular devices (iPhone was released in 2007, Windows 7 in 2009). When Microsoft planned the next version of Windows they wanted to have a core system that would work on many devices, not only PCs. This brought the more tablet-like user interface (e.g., large flat buttons) to the PC which in the end alienated many users. Some shortcomings were remedied with Windows 8.1 (e.g., the Start button was brought back).

Modern Windows cont.

The push into mobile created another API that was more targeted for mobile computing: WinRT. WinRT gives users a subset of functionality of the full Win32 API, but allows the application to run a wide range of systems. This idea still lives as the Universal Windows Platform and application published in the Windows Store.

Modern Windows cont.

Of course the current version of Windows is Windows 10. It has some new features compared to Windows 8.1 but probably the biggest change is in the update philosophy. There

are no further releases but semi-annual big updates bring new features and remove those deemed obsolete.

Architecture

The kernel

Core of the Windows operating system is the NTOS kernel. It is contained in the file `ntoskrnl.exe`. Kernel of course implements system calls, but contarary to Linux these are not public: only programmers at Microsoft use the kernel system calls directly. This allows for greater flexibility if some of the interfaces were to change (although it seems they are pretty stable).

The kernel contains the memory manager, scheduler, the registry API etc.

Native API

The native API is the set of routines that are available for processes at boot time before other subsystems become functional. They are contained in `ntdll.dll` and make use of the kernel system calls.

Applications linking to this library are said to use the *native subsystem* (although doing this is rare). Most of this API is left undocumented as it is not considered stable.

The registry

Windows uses a specialized hierachical database for storing settings etc. It might be though of a filesystem specialized for many very small files. It is stored on the disk in a few file called *hives*. The database is not accessed like a usual filesystem, but uses specialized API.

Subsystems

User programs do not interface the kernel directly, but use so-called **subsystems**. The most important of these is the Win32 subsystem, but in the past OS/2 and POSIX subsystems were also used. This allows to run programs written using the Win32 API. Subsystems are startedted by the Session Manager Subsystem executable `smss.exe`. This is an equivalent of the Linux `init`, as it is the first user program started by the Windows kernel. One of its tasks is to overwrite files that were queued up from before the last reboot (e.g., by an update).

Win32 subsystem

The Win32 subsystem implements the Win32 API that were the common subroutines used by programmers for Windows 95 and the old Windows NT. Of course it expanded a lot during the years. The Win32 API calls are translated by the subsystem to Native API calls. The subsystem is officialy called the Client/Server Runtime Subsystem and is started by the `csrss.exe` excutable.

Subroutines are defined in `kernel32.dll` (most of the general API), `gdi32.dll` (drawing graphics primitives), `user32.dll` (user interface), `comctl32.dll` (standard controls).

Example

Programming Windows API is far beyond the scope of this lecture (and equally far beyond the lecturers capablities). We will just see one example – a program that creates an empty window:

Example

```
// Example comes from:
// https://docs.microsoft.com/en-
 ↪  us/windows/desktop/learnwin32/learn-to-program-for-windows
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>

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

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR
 ↪  pCmdLine, int nCmdShow) {
    // Register the window class.
    const wchar_t CLASS_NAME[] = L"Sample Window Class";

    WNDCLASS wc = {};
    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstance;
    wc.lpszClassName = CLASS_NAME;
    RegisterClass(&wc);

    // Create the window.
    HWND hwnd = CreateWindowEx(
        0,                              // Optional window
         ↪  styles.
        CLASS_NAME,                     // Window class
        L"Learn to Program Windows",    // Window text
        WS_OVERLAPPEDWINDOW,            // Window style
                                        // Size and position
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
 ↪  CW_USEDEFAULT,
        NULL,       // Parent window
        NULL,       // Menu
        hInstance,  // Instance handle
```

```
        NULL        // Additional application data
    );

    if (hwnd == NULL) {
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);

    // Run the message loop.

    MSG msg = {};
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
 ↪  LPARAM lParam) {
    switch (uMsg) {
        case WM_DESTROY: {
            PostQuitMessage(0);
            return 0;
        }
        case WM_PAINT: {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);
            FillRect(hdc, &ps.rcPaint, (HBRUSH)(COLOR_WINDOW +
 ↪  1));
            EndPaint(hwnd, &ps);
        }
        return 0;
    }
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

PowerShell

Introduction

For a log time Windows lacked a decent shell. Scripts could have been defined and run using the Windows Script Host, but this is a non-interactive environment. Granted, Windows had the `cmd.exe` for a very long time (this is a evolution of the old MS-DOS prompt), but compared to bash it is very limited and constrained.

In 2006 Microsoft released a new shell – PowerShell – designed from scratch to make life of administrators much simpler. Even though it is inspired by `bash`, some things work completely differently.

Overview

Bash was created by many people, similarly the coreutils package is a result of a long evolution. That's why even though the basic Linux tools have a similar interface, one can still find multiple inconsistencies between them.

PowerShell aim was to reduce the clutter and make everything more consistent. Its creators decided, that commands should not send text to each other but full fledged .NET objects. It's possible to create pipelines as in bash. Commands naming was to be kept consistent by using the Verb-Noun convention. A parameter naming convention was created as well.

Most importantly: Windows now has a home for command-line tools.

.NET

The .NET Framework is a software package create by Microsoft that is able to seamlessly integrate multiple programming languages. It does it by compiling source code not to machine code, but to the so-called Intermediate Language (IL). The main languages of the .NET ecosystem are C#, Visual Basic .NET and C++/CLI.

IL is run by an application virtual machine called Common Language Runtime (CLR). In principle this allows for the .NET platform to be system-agnostic. This is the same idea that is used by Java and other JVM languages (Kotlin, Scala, Clojure). The main implementation however was created by Microsoft just for the Windows system.

.NET Core

In 2015 Microsoft began open-sourcing the basic libraries that underlie the .NET Framework. The project is called .NET Core and is expanding all the time. .NET Core used to omit parts of the Framework that had to do with GUI programming, but version 3.0 that is to be released in 2019 is slated to have the necessary libraries included (Windows only for now). The version 2.x allows to write command line tools and web applications that work on multiple platforms.

Versions of PowerShell

Right now there atwo version of PS: PowerShell integrated into Windows and new PowerShell Core that uses .NET Core. Only the latter is cross-platform and receives new features. The Windows one is effectively frozen.

Powershell basics

Verb-Noun

PowerShell creators wanted to have more discoverability of the included commands (called **cmdlets**). An official name of the cmdlet is Verb-Noun, e.g.:

- `Get-Command`
- `Get-Member`
- `Sort-Object`
- `Format-Table`
- `Stop-Process`
- `Get-Verb`

Variables

Variables are created and referenced using the `$` character, e.g., `$a` is a variable. Contrary to bash variables, variables do have types, and it is possible to invoke methods on those objects:

```
$a = Get-Date
$a.AddDays(1)

(Get-Date).AddDays(1)
```

Comparison operators

We have the following operators:

- arithmetic: `-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`.
- strings: `-like`, ``-notlike` (wildcards), `-match`, `-notmatch` (regular expressions)
- containment: `-contains`, `-notcontains`, `-in`, `-notin`.
- replace: `-replace`, `-creplace`

Examples

```
”Sunday” -match ”sun”
”Sunday”, ”Monday”, ”Tuesday” -match ”sun”
”Windows” -in ”Windows”, ”PowerShell”
”book” -replace ”B”, ”C”
”book” -creplace ”B”, ”C”
```

Logical operators

Different logical conditions can be joined using the logical operators `-and`, `-or`, `-xor`, `-not` (`!`).

```
($a -gt $b) -and !(($a -lt 20) -or ($b -lt 20))
```

The `if` instruction

`if` has the syntax similar to C (note the `elseif` clause):

```
if ($a -gt 2) {
    Write-Host "The value $a is greater than 2."
}
elseif ($a -eq 2) {
    Write-Host "The value $a is equal to 2."
}
else {
    Write-Host ("The value $a is less than 2 or" +
        " was not created or initialized.")
}
```

The C loops

PowerShell has a for loop that is taken from C:

```
for($i=1; $i -le 10; $i++) {
  $i
}
```

The `while` and `do-while` loops are also similar to their C counterparts.

The `foreach` loop

Probably the most useful loop is the `foreach` loop, e.g.,:

```
$letterArray = "a","b","c","d"
foreach ($letter in $letterArray)
{
  Write-Host $letter
}

1,2,3 | Write-Host
```

Note that often it is not necessary: cmdlets normally iterate over all their arguments:

A short list of commands

`Get-`

The command with the verb `Get-` return a list of items:

- `Get-Command Sort-*` – returns list of commands starting with `Sort-`.
- `Get-Member` – returns list of object fields and methods
- `Get-Verb` – returns the official list of verbs in cmdlets names
- `Get-ChildItem` – list contents of directories, registry hives etc.
- `Get-Process` – list running processes
- `Get-Help` – get help for a command

`Out-`

- `-Out-Host -Paging` – see output one screen at a time
- `-Out-Null` – discards output

`Where-Object`

`Where-Object` is the filtering cmdlet. The following commands are equivalent they show files smaller than 1kB in the current directory:

```
Get-ChildItem -File | Where-Object {$_.length -lt 1024}
Get-ChildItem -File | where length -lt 1kB
```

Select-Object

This command can select specific properties from objects as well as does simple selections from collections:

```
Get-Process | Select-Object -Property ProcessName, Id, WS
Get-Process | Select-Object -Property ProcessName,@{Name="Start
↪  Day"; Expression = {$_.StartTime.DayOfWeek}}
```

Reference

The full list of available commands can be found here.

Probably the best place to start learning PowerShell (with examples) is this part of the documentation.