Windows Native API

Roger Orr

OR/2 Limited

How do applications communicate with the O/S kernel?

Applications and the Kernel

- In common with many operating systems, Windows has two modes
  - A mode for user programs ('user')
  - A mode for the O/S ('kernel')
- A user mode application program
  - cannot directly access hardware
  - can only access memory locations it 'owns'
- Kernel mode allows
  - direct access to hardware
  - access to the whole of physical memory

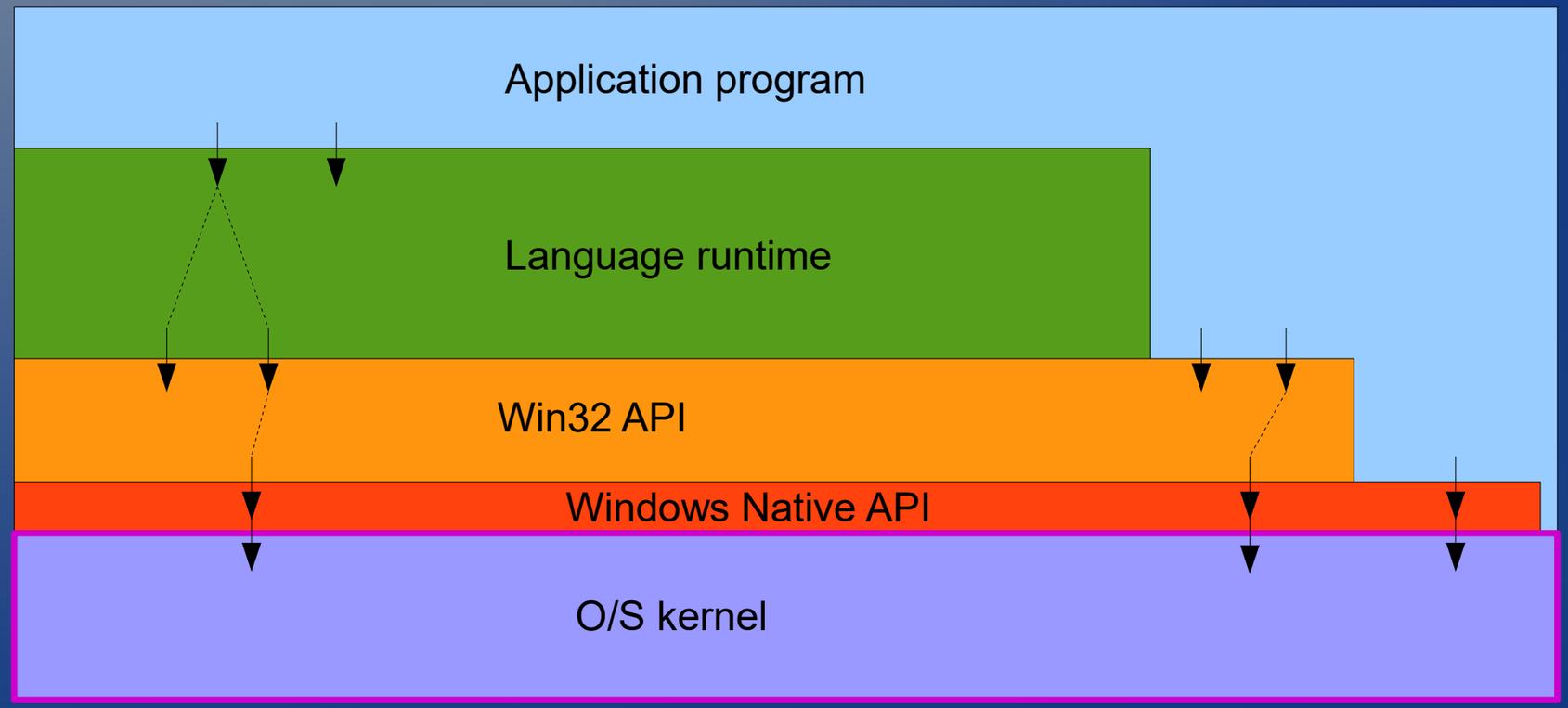
Applications and the Kernel

- Applications request services from the O/S by making a system call
  - the mode changes to kernel mode
  - the request and its parameters are verified
  - the operation is performed
  - the kernel returns a status code (to indicate whether the request succeeded)
- However, it is rare to make these requests directly: Windows provides a higher level API known as the Win32* API

  * (even when it's 64-bit)

Applications and the Kernel

- Programming languages in turn often further wrap the Win32 API to provide their library runtime functions



A simple example

- Consider this C++ program

```
#include <cstdlib>

int main() { exit(42); }
```

- This calls the C++ library function `exit()`
- This (eventually) invokes the Win32 function `ExitProcess()`
- This in turn invokes the Windows Native function `NtTerminateProcess()`

A simple example

- Consider this C++ program

```
#include <cstdlib>

int main() { return 42; }
```

- This *implicitly* calls the C++ library function `exit()`
- So it behaves like the previous example
- However, I highlight this as there's more to the language runtime than just *your* calls

A simple example

- Let's go one layer down...

```
#include <windows.h>
int main()
{
  ExitProcess(42);
}
```

- Note that this does not let the C++ runtime terminate as gracefully, but it does allow some Win32 teardown (eg DLL unload)

A simple example

- Or we can terminate 'with prejudice':

```
#include <windows.h>
int main()
{
  auto self = GetCurrentProcess();
  TerminateProcess(self, 42);
}
```

- This does less Win32 tear-down

A simple example

- Another layer down... **NtTerminateProcess**
- Firstly, we need to provide a prototype for the function, since only a few Windows Native calls are provided in the Windows Kits

```
extern "C" NTSTATUS NTAPI
NtTerminateProcess(
    IN HANDLE ProcessHandle,
    IN NTSTATUS ExitStatus);
```

A simple example

- Then we call the function

```
int main()
{
  auto self = GetCurrentProcess();
  NtTerminateProcess(self, 42);
}
```

- Note that this may not let the Win32 runtime terminate gracefully: for example it skips Silent Process Exit monitoring

Inside a native call

- The function body is something* like this:

```
NtTerminateProcess:
        mov     r10,rcx
        mov     eax,2Ch
        syscall
        ret
```

- The **`syscall`** instruction transitions to kernel mode. Register `eax` is the code for the function to invoke.

* dependent on the version of Windows (see next slide)

Inside a native call

- The precise details of the function body differ as the mechanisms used to enter kernel mode have changed subtly with different versions of Windows and changes in hardware support:
- `int 2E` - initially windows used an interrupt
  `sysenter` - a special instruction simply added for speed
  `syscall` - a 'new improved' instruction (AMD/Intel split)
- The *principle* is unchanged: inside the kernel there is basically a jump table driven by the function code argument and targetting the desired function

Inside a native call

- The target function in the kernel has the same name as the original NtXxx function
- However, this implementation function runs in the kernel with the O/S privileges and can therefore perform actions that the user mode program is not able to do

Note on kernel development

- The target NtXxx functions are also callable from *within* the kernel too. However, in this case since the **caller** is trusted code there is an additional choice of an 'internal' function, with a prefix of 'Zw' rather than of 'Nt'
- The Zw version of the function can skip some of the validation of the arguments supplied in the function call
- If you're trying to find information about a native call, try to search for *both* the names

Inside the kernel

- I'm not going to cover much about the Windows kernel itself.
  - It'd take too long
  - I'm not a driver level guy
- There is information about the Windows kernel available from many sources, such as:
  - Windows Driver Kit (WDK)
  - Books, such as “Windows Internals”
  - “NT Insider” at www.osr.com/nt-insider/

Argument validation

- The native API is where applications transition from the 'protected' environment into the kernel. It's important to verify the arguments passed to the service call to ensure that the program cannot accidentally - or maliciously -
  - Crash or corrupt the O/S
  - Use resources it does not own
  - Perform actions requiring higher privilege

Return codes

- The native API normally returns an `NTSTATUS` value (a 32-bit unsigned value), with the top two bits set on error
- Some of the possible error values are documented in WinNt.h, or in NtStatus.h, such as:

```
STATUS_OBJECT_NAME_NOT_FOUND (0xC0000034)

STATUS_NO_TOKEN (0xC000007C)
```

Return codes

- The function `RtlNtStatusToDosError` in `NtDll` converts the `NTSTATUS` native error value into a user-mode error value
- For example, the native error code `0xC0000034` is translated to 2 which is `ERROR_FILE_NOT_FOUND`

Types of arguments

- There arguments used by the native API can be broken down into the following rough categories:
  - Simple values
  - Handles
  - Pointers to memory
  - Strings
  - Object Attributes

Simple value arguments

- Many of the simple values are integers – the commonest example being `ULONG` (a 32-bit unsigned value). There are also a number of arguments that are enumeration values
- For example, `NtQueryInformationProcess` takes an argument of type `PROCESS_INFORMATION_CLASS` which has values like `ProcessMemoryPriority`
- Other types include `LARGE_INTEGER` (64-bit) and `BOOL`
- In quite a few cases the simple values are the same ones as that of a corresponding higher-level Win32 API, as we saw earlier with `NtTerminateProcess`

Enumeration values

- Some of the enumerations used in the Native API are the same as those used in the Win32 API and some of the others are documented by Microsoft
- Most of the others can be found in the PDB file for WinTypes.dll; for example:

```
> dt ALPC_MESSAGE_INFORMATION_CLASS
wintypes!ALPC_MESSAGE_INFORMATION_CLASS
   AlpcMessageSidInformation = 0n0
   AlpcMessageTokenModifiedIdInformation = 0n1
   AlpcMessageDirectStatusInformation = 0n2
   AlpcMessageHandleInformation = 0n3
```

Handle arguments

- While on 64-bit Windows handles are 64-bit values in practice they currently seem to be (possibly sign-extended) 32-bit values
- This provides obvious benefits for interoperability with 32-bit programs running on the WOW subsystem

Handle arguments

- Many kernel resources expose themselves via an opaque handle type
- Typically the initial Open or Create call takes a pointer to handle as the first argument, and subsequent operations take the handle value as the first argument
- The handle value is usually the same value as that exposed by equivalent Win32 calls, so a mix of native and Win32 calls can be used (with some restrictions)

Pointer to memory arguments

- Pointers to memory are usually presented as a *pair*: an address and a length
- Many of the APIs have a clear separation between *input* and *output* buffers
- In a few cases buffers are modified in-place

String arguments

- Most of the native API takes strings as:

```
struct UNICODE_STRING {
  USHORT Length; // Length of string in bytes
  USHORT MaximumLength; // Maximum length
  PWSTR  Buffer; // Pointer to Unicode string
};
```

- A small number of APIs take a `PWSTR` argument (together with a length)
- In both cases note the string is of 16-bit Unicode (UTF-16) characters

String arguments

- The `UNICODE_STRING` structure uses unsigned 16-bit integers for the length, even on a 64-bit platform, which means that strings are limited to 32,767 characters
- This is rarely a significant issue in practice!

Object attributes arguments

- Many of the APIs take an `OBJECT_ATTRIBUTES` argument which specifies the name and some additional attributes of the object being accessed. Here's the structure:

```
typedef struct _OBJECT_ATTRIBUTES {
  ULONG           Length;
  HANDLE          RootDirectory;
  PUNICODE_STRING ObjectName;
  ULONG           Attributes;
  PVOID           SecurityDescriptor;
  PVOID           SecurityQualityOfService;
} OBJECT_ATTRIBUTES;
```

Object attributes arguments

- The root directory is a handle to use for relative names. The name is a little misleading as it applies to more than files – for example it is used for registry keys
- Unlike the Win32 API there is no implicit 'current directory' for calls
- The full names for files and directories in the native API may be unfamiliar:

  `\??\C:\projects\accu\conference\2019` or
  `\Device\HarddiskVolume5\projects\accu\conference\2019`

- We'll come back to that shortly!

Some security considerations

- The Native API is the gateway from user mode to privileged mode and so there are a number of implications for both security and process separation
- The degree of caution taken over argument handling has increased since the early days of Windows NT
- I'm not going to cover the recent *hardware* level vulnerabilities (Meltdown and Spectre) although they have affected the O/S API

Simple value arguments

- The security implications of simple values are similar to those in any API; the only difference is that if something is *out of range* it might produce exploitable behaviour in the kernel which is at a higher privilege level
-
```
NTSTATUS ZwCreateEvent(
  _Out_    PHANDLE            EventHandle,
  _In_     ACCESS_MASK        DesiredAccess,
  _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
  _In_     EVENT_TYPE         EventType,
  _In_     BOOLEAN            InitialState);
```
- What could happen if I were to provide an invalid value?

Handle arguments

- It is important to check that the handle value being passed in is right for the API being used, that it belongs to the calling process, and that is has the right permissions
- Earlier checks were done with the handle value as it arrived; but malicious programs found they could close and reopen the handle in other threads so that the *same* handle value referred to a *different* entity later on in the call, allowing access above the privilege level of the caller

Pointer to memory arguments

- Memory buffers present a few challenges:
  - Does the calling process have access to it?
  - With the right permissions?
  - *All* of it?
  - How does the kernel ensure that it maps the right page(s) from the user program?
  - What if the memory is swapped out?
  - What if the user changes the contents?

Access to memory arguments

- The kernel needs to check that the calling program has provided a valid pointers to memory for which it holds the required access (for example, write access if the API will write into the buffer).
- This check needs to be made for the full range of addresses in the buffer

Lifetime of memory arguments

- If the kernel will be retaining the buffer (for example, for I/O) then the memory may need locking so that the physical addresses of pages can be passed to hardware
- Remember that while *this* thread is no longer active in the user's application, *other* threads can be modifying the memory being addressed

String arguments

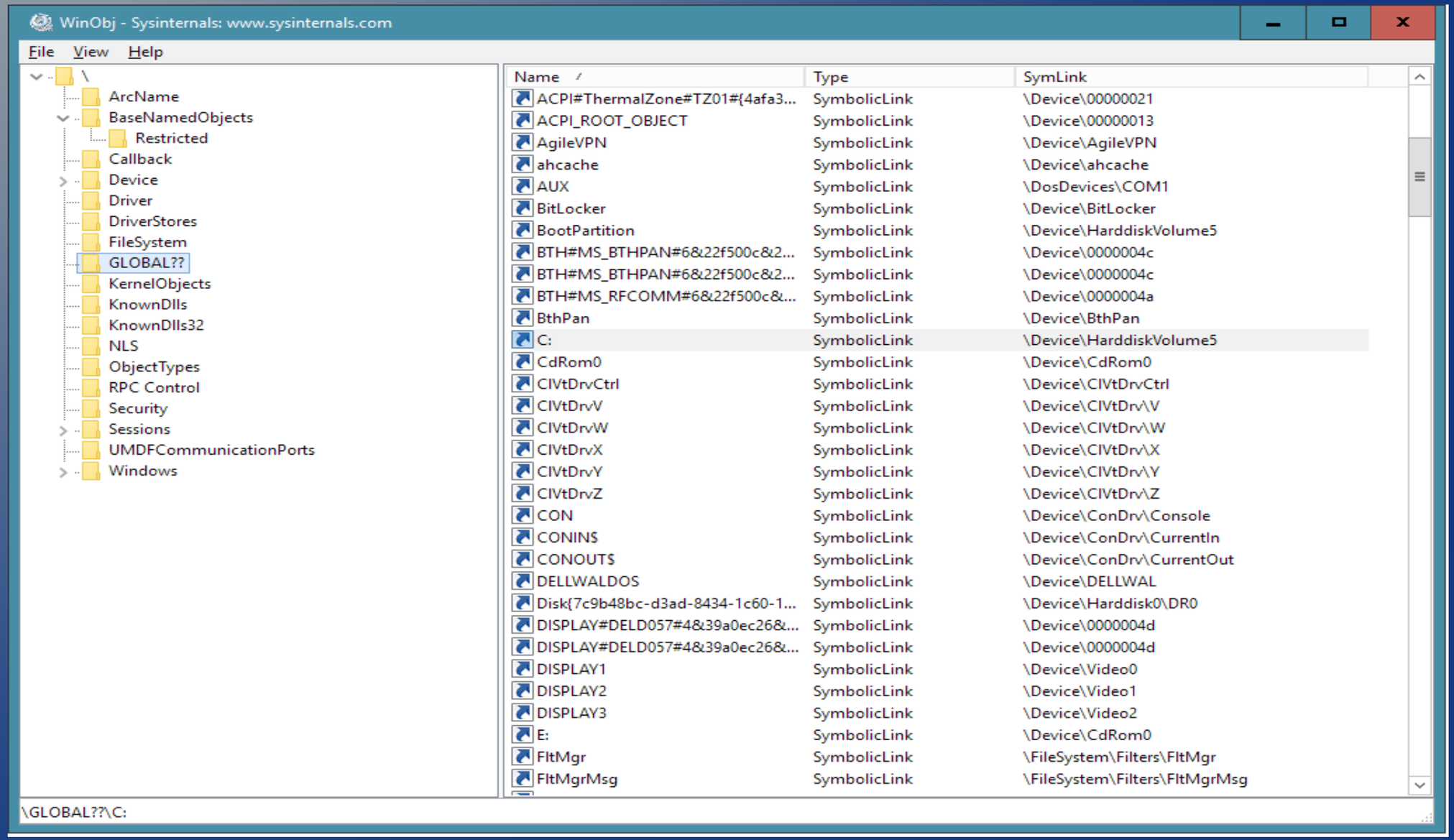
- The buffers used to hold string arguments have the same set of restrictions as other pointers to memory
- However, there are additional wrinkles with strings as they may contain various 'odd' characters or invalid character sequences
- UTF-16 processing is notoriously hard to get right
- Embedded `NUL` characters (among others) can cause 'interesting' interaction with the Win32 API which often uses `LPCSTR`

Object namespace

- As I briefly mentioned when looking at 'object attributes' the names of entities may be different when using the Native API rather than the Win32 API
- The kernel has a hierarchical namespace containing objects of various classes, such as: Device, Event, Key, Mutant*, Section, Semaphore, and SymbolicLink
- The Sysinternals “WinObj” tool lets you view the object hierarchy

  * A Mutex, with an odd name for historical reasons

Object namespace - WinObj



Object namespace

- We saw earlier:

  `\??\C:\projects\accu\conference\2019` or
  `\Device\HarddiskVolume5\projects\accu\conference\2019`

- The top level “??” is the global namespace, which has a symbolic link to the actual device
- `\Device\HarddiskVolume5` is the device name for the C: drive on my desktop PC
- The *rest* of the path is not in the object namespace, but is understood by the device driver itself

Object namespace

- There are quite a few symbolic links in the object namespace – here are some of the other aliases for the same hard disk:

```
\Device\BootDevice
\??\BootPartition
\??\Volume{661a4c3d-06b0-45e2-98aa-3e6b647c1f37}
```

- These are typically set up when the system starts up and devices are initialised
- Few application programmers need to be concerned about the set of names

Object namespace

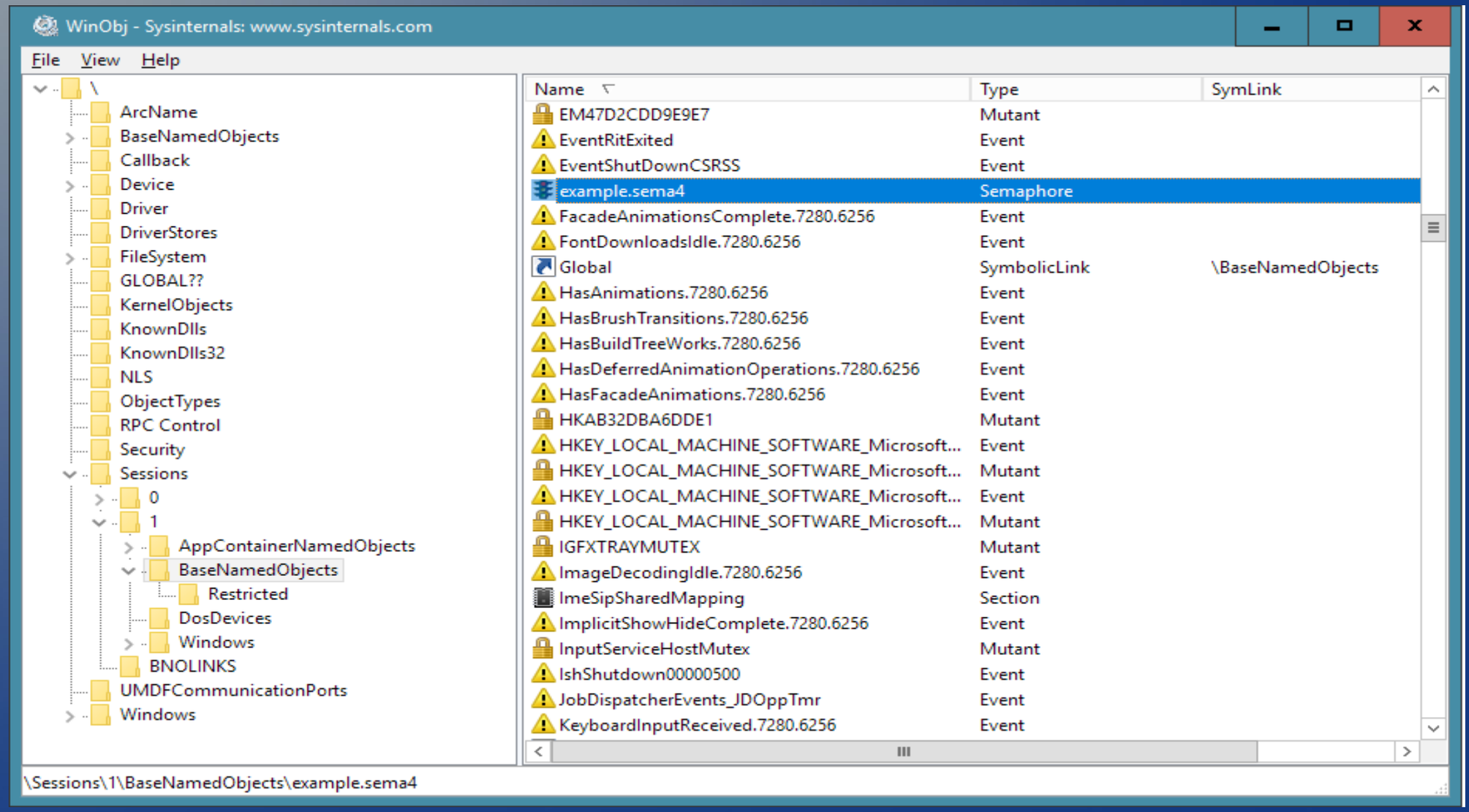
- Some of the **Win32** API calls support access via the global namespace – but, just to make it more confusing, using the prefix of `\\?` rather than `\??`
- For example: `dir \\?\c:\temp`
- This *actually* opens `\??\c:\temp`
- However, this syntax allows filenames longer than `MAX_PATH` and disables some the Win32 mappings - for instance you can create a filename called “...”

More than just the filesystem

- As is probably already obvious from the list of classes the object namespace is used for much more that just the filesystem
- Most of the Win32 calls that take *names* start by finding a root name in the object namespace
- For example, the registry API accesses paths like `\Registry\Machine\Software`
- Creating a named Semaphore adds an item to the object namespace – for example

```
CreateSemapore(0,0,1,"example.sema4")
```

Object namespace - WinObj



Categorising the Native API

- There are various ways to categorise the 500 or so functions in the Native API. For example:

Atom (5)
Device (31)
File (48)
LPC (47)
Object (20)
Registry (42)
Synchronization (35)
Transaction (49)

Debug (17)
Environment (17)
Job (8)
Memory (35)
Process (44)
Security (38)
Time (17)
WOW64 (20)

- And a few that are hard to categorize (28)

Categorising the Native API

- But wait – there's more!
- So far I have only mentioned the functions located in NtDll.dll
- There are (at least) three other mechanisms that give user mode programs access to functionality in the Windows kernel
  - NtUserXxx functions for UI functionality
  - NtGdiXxx functions for graphics
  - WSL system calls (lxss.sys and lxcore.sys)

Categorising the Native API

- Most of the work for the windowing subsystem takes place in kernel (for various reasons, such as performances). The API is loosely split into “user” (winuser.h) and “gdi” (wingdi.h) functions
- NtUserXxx functions such as NtUserCreateWindowEx provide the core functionality for the Win32 CreateWindowEx
- NtGdiXxx functions such as NtGdiSaveDC implement SaveDC
- (The mapping is not always *quite* this direct)

Categorising the Native API

- While the graphic subsystem uses some new data structures the basic mechanism is essentially the same
  - The function codes are above 0x1000
  - The implementation entry point is in win32k.sys
- The return codes are messier – functions return a variety of types (such as BOOL and various handle types) so there's not such a consistent way to detect failures
- I won't cover any more ground about this set of functions

Windows Subsystem for Linux

- Windows 10 supplied a whole new way to execute user-mode programs – WSL
- This allows running native Linux binaries on Windows
- The interface to the O/S is via the standard Linux syscall mechanism
- Inside the kernel there is a device driver `lxss.sys` that provides the functionality of the Linux system call interface
- The implementation provides a cached shim over direct calls to ZwXxx functions

Windows Subsystem for Linux

- The Linux syscall mechanism uses the same underlying syscall mechanism as NtDLL - but with *different* register conventions
- The set of codes used is stable (**unlike** on Windows) but does overlap
- When the O/S detects that the calling process is running as a “pico process” it passes the system call on to lxss.sys
- Again, this talk is not primarily about WSL so that's all I intend to say about it

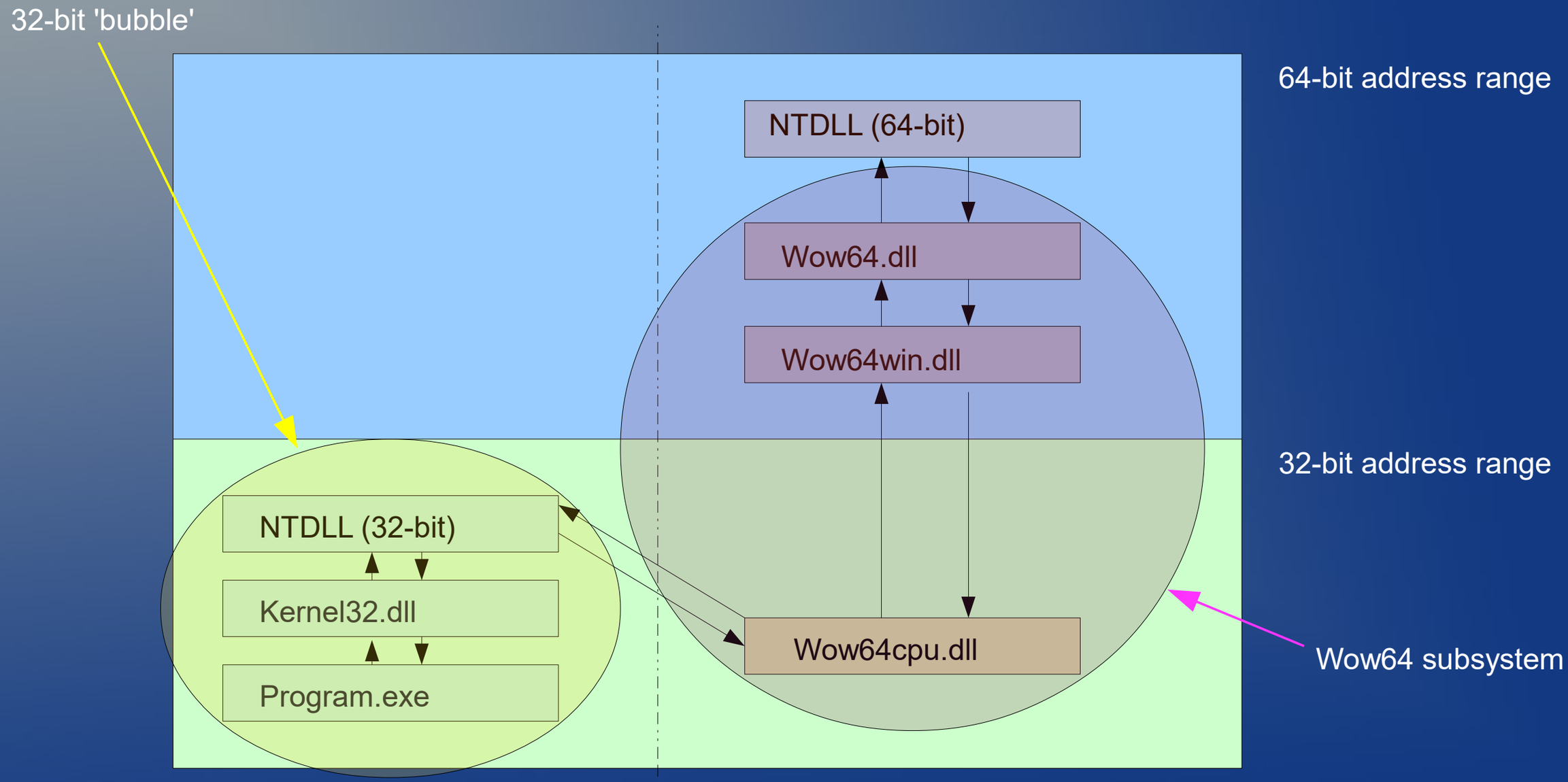
WOW64

- The 64-bit version of Windows (“x64”) can also run 32-bit programs (“x86”). How is this achieved?
- The 32-bit EXE and its required DLLs are loaded into the low 4GB of linear address space for the process and the selectors (CS, DS, etc):
  - Have a 32-bit address range
  - Use 32-bit mode (this defines the interpretation of the instructions and the register set available)

WOW64

- The running program makes calls exactly as usual to other functions in the 32-bit address space
- It also makes calls to the 32-bit Native API to access O/S resources
- On a genuine 32-bit O/S these calls would go into the kernel in a corresponding way to the 64-bit cases discussed above
- On a 64-bit O/S the calls are *intercepted* by the wow64 subsystem and eventually call the 64-bit Native API

WOW64


32-bit 'bubble'
64-bit address range
NTDLL (64-bit)
Wow64.dll
Wow64win.dll
32-bit address range
NTDLL (32-bit)
Kernel32.dll
Program.exe
Wow64cpu.dll
Wow64 subsystem

WOW64

- The 32-bit Native API makes an indirect call to invoke the system call, for example:

```
NtTerminateProcess:
   mov      eax,0x7002c
   mov      edx,Wow64SystemServiceCall
   call     edx
   ret      0x4
```

- Register eax is the code for the function to invoke (low word) and some mapping information (encoded into the high word)
- `Wow64SystemServiceCall` contains a jump into 64-bit land

WOW64

- Wow64SystemServiceCall makes a “far jump” to a 48-bit target address consisting of a 16-bit selector (segment register) and a 32-bit offset. The new CS is a 64-bit one and so the process can access 64-bit instructions and registers.
- A load of a different DS gives access to the full 64-bit address range
- On return from 64-bit land another far jump returns to the 32-bit world
- Sometime the jump to 64-bit land is referred to as “Heaven's Gate”

WOW64

- Once in 64-bit land the arguments to the system call need to be translated
- 32-bit **values** are zero-extended and 32-bit **addresses** are also zero-extended (as the first 4GB of the 64-bit world maps to the same linear address range as the 32-bit world)
- Data structures may need converting
- For example, in 32-bit programs `UNICODE_STRING` is 8 bytes and `OBJECT_ATTRIBUTES` is 24 bytes
- In the 64-bit API they are 16 and 48 bytes respectively, so new structures need creating

WOW64

- Next some additional data transformations are applied (see next slide) to help provide a familiar environment to 32-bit programs
- Then the 64-bit Native API call is made
- On return *reverse* translations are performed:
  - 64 bit values are narrowed
  - Returned data structures are converted into the 32-bit format
- This is normally transparent to the user; there are occasionally places where the join shows

WOW64

- The translation layer changes **filenames** and **registry keys** to provide a (nearly) seamless 32-bit environment on the 64-bit O/S. For example:
- The directory `System32` in `C:\Windows` is mapped to `SysWow64`
- The registry key `Software` in `HKEY_LOCAL_MACHINE` is mapped to `Software\Wow6432Node`
- The mappings are relatively complex, change a little between versions of Windows, and have exceptions

WOW64

- The 32-bit Windows API adds some extra functions to allow 32-bit programs to opt out of the mappings and access the underlying 64-bit filesystem and registry
- For the file system a call to Wow64DisableWow64FsRedirection will affect subsequent calls by the calling **thread**
- For the registry access the function RegDisableReflectionKey can be called for a specified **registry key**

WOW64

- The Windows Native API also adds some extra functions to allow 32-bit programs to access a limited amount of 64-bit functionality.

  For example:

- `NtWow64GetNativeSystemInformation` can be used to obtain information about the hosting 64-bit system (`NtQuerySystemInformation` returns information about the 32-bit 'virtual' system)
- `NtWow64[Read/Write]VirtualMemory64` allows a 32-bit process to access memory above the 4Gb limit

WOW64

- WOW64 even supports **debugging** of 32-bit programs (with a very small number of restrictions)
- You can also debug a 32-bit program with a 64-bit debugger (since it 'is' a 64-bit program)
- There is no support for debugging a 64-bit program with a 32-bit debugger...

Documentation

- The WDK (Windows Driver Kit) does now document a proportion of the Native API
  - I believe some of this was in response to pressure from customers and OEMs
- Many functions have been re-engineered by the open source ReactOS project
- `undocumented.ntinternals.net` has some useful information
- Other web sites, especially ones on software security (!)

Tools

- There are some free tools that let you debug calls to the Native API including:
- Strace for NT from BindView (archive.org)
- `drstrace` from Dr Memory `www.drmemory.org`
- StraceNT from `intellectualheaven.com`
- NtTrace from `github.com/rogerorr/NtTrace`
- There are also some commercial tools

Conclusion

- I need one of these, too.