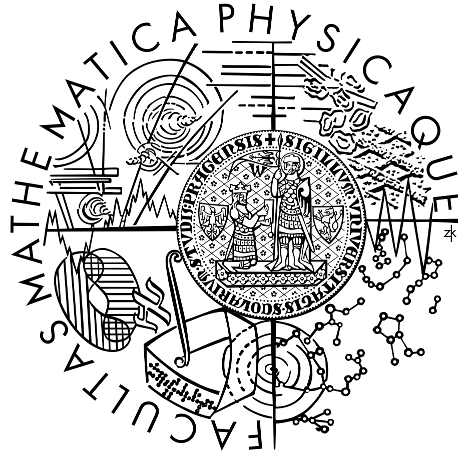
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Roman Kápl

Tracing Function Calls in Windows NT Kernel

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D.
Study programme: Computer Science
Study branch: Programming

Prague 2015

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

Title: Tracing Function Calls in Windows NT Kernel

Author: Roman Kápl

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Operating systems are complex and hard to understand. Students that want to learn more about internal operation of the *Windows NT* system can use tools such as *WinDbg*, *WinObj* or *Process Monitor*. However, these tools are either hard to use or do not offer sufficient level of detail. This thesis implements a new tool focused on monitoring of the *I/O* handling, called *WinTrace*. It can monitor key *I/O* events, such as execution of dispatch routines, completion routines, interrupts and deferred procedure calls. To make the understanding of the recorded events easier, *WinTrace* can summarize them as graphical diagrams. While the tool is primarily targeted at students, it should also be valuable to driver developers when debugging real-world problems or as a general purpose function tracer. We also hope the thesis will be useful to anyone hooking functions in the NT Kernel, as we identify the problems that can be encountered during the implementation.

Keywords: Windows, kernel, function tracing, debugging

Abstrakt: Operační systémy jsou složitý software a je těžké pochopit, jak fungují. Studenti, kteří by chtěli vědet více o fungování operačního systému Windows NT, mohou použít programy jako je WinDbg, WinObj nebo Process Monitor. Bohužel, tyto programy buď není jednoduché ovládat nebo neposkytují dostatečně nízkoúrovňové informace o systému. Výsledkem naší práce je nový program, nazvaný WinTrace, který se zaměřuje na sledování vstupně/výstupních požadavků. Je schopný monitorovat klíčové vstupně/výstupní události, jako je spouštění obslužných funkcí ovladaču (dispatch routines), dokončovacích funkcí (completion routines), obsluha přerušení a volání DPC. Aby byl výstup programu lépe pochopitelný, WinTrace umí zaznamenané události shrnout do grafických diagramů. I když je program určen hlavně pro studenty, měl by být cenným pomocníkem i pro programátory při ladění problémů nebo jako obecný nástroj pro sledování volání funkcí. Také doufáme, že tato práce bude přínosná pro všechny, kteří se snaží zachytávat nebo měnit funkce v jádře Windows NT, jelikož popisujeme časté problémy, na které lze narazit.

Klíčová slova: Windows, jádro, sledování funkcí, ladění

This work would not be possible without my supervisor, who has provided numerous suggestions how to improve this thesis, my parents, who have shown support and understanding, and also the local store selling suprisingly strong energy drinks.

I also have to congratulate the people behind the *ReactOS* project for the amount of work they have done in documenting and re-implementing the *NT* kernel and thus saving me a substantial amount of debugging time.

Contents

1 Introduction **5**
1.1 Overview of Windows I/O . . . 5
1.2 Problem statement . . . 6
1.3 Goals . . . 7
1.4 Required knowledge . . . 7

2 I/O in NT kernel **9**
2.1 NT API layers . . . 9
2.2 Devices, drivers and files . . . 10
2.3 Objects . . . 12
2.3.1 Naming . . . 13
2.3.2 Symlinks . . . 13
2.4 Driver stacks . . . 13
2.5 Recapitulation . . . 15
2.6 I/O . . . 16
2.6.1 I/O locations . . . 16
2.6.2 Asynchronous I/O . . . 17
2.6.3 I/O completion routines . . . 17
2.6.4 I/O without IRPs . . . 18
2.7 Interrupts and synchronization . . . 19
2.7.1 Disabling interrupts . . . 19
2.7.2 DPCs . . . 19
2.7.3 IRQL . . . 20
2.8 Plug and Play . . . 21
2.8.1 Enumeration and PnP driver stacks . . . 21
2.8.2 PnP tree . . . 22

3 Existing utilities **25**
3.1 I/O system concepts . . . 25
3.2 Debuggers . . . 26
3.2.1 WinDbg . . . 26
3.2.2 Live WinDbg . . . 29
3.2.3 Third-party debuggers . . . 29
3.3 Object inspection tools . . . 29
3.3.1 WinObj . . . 30
3.3.2 Device Manager . . . 30
3.3.3 DeviceTree . . . 31
3.4 Monitoring tools . . . 32
3.4.1 Process Monitor . . . 32
3.4.2 Event Viewer . . . 33
3.4.3 Logman . . . 34
3.4.4 IrpTracker . . . 34
3.4.5 Function tracing tools . . . 35
3.5 Tool comparison . . . 35
3.6 Proposal for our tool . . . 36

4 Gathering information **39**
4.1 Methods . . . 39
Debugger . . . 39
Event Tracing for Windows . . . 40
Callbacks . . . 40
Filter drivers . . . 40
Dispatch routines . . . 41
Code instrumentation . . . 41
4.2 Conclusion . . . 42

5 Instrumentation **43**
5.1 Libraries . . . 43
5.1.1 Detours . . . 43
5.1.2 EasyHook . . . 43
5.2 Methods . . . 44
5.2.1 Breakpoints . . . 44
5.2.2 Hardware breakpoints . . . 45
5.2.3 Import table hooking . . . 45
5.2.4 Prologue overwriting . . . 45
5.2.5 Analysis . . . 46
5.3 Implementation . . . 46
5.3.1 32-bit execution redirection . . . 47
5.3.2 32-bit call to original . . . 48
5.3.3 64-bit execution redirection . . . 48
5.3.4 64-bit call to original . . . 49
5.4 Pitfalls . . . 49
5.4.1 Memory protection . . . 49
5.4.2 Non-standard functions . . . 49
5.4.3 Compiler optimizations . . . 50
5.4.4 Structured Exception Handling . . . 50
5.4.5 Kernel Patch Protection . . . 50
5.5 Monitoring events . . . 51
5.5.1 Hooked functions . . . 51
5.5.2 32-bit interrupts . . . 52
5.5.3 64-bit interrupt hooking . . . 53
5.6 Collecting events . . . 53
5.6.1 Writing to a file . . . 54
5.6.2 Event Tracing for Windows . . . 54
5.6.3 Lockless circular buffer . . . 54
5.6.4 Dropping events . . . 55

6 Visualization and parsing **57**
6.1 Event view . . . 57
6.2 Object view . . . 58
6.2.1 Threads . . . 59
6.2.2 Interrupts and DPCs . . . 61
6.2.3 Events . . . 61
6.3 Parsing . . . 62
6.3.1 Goals . . . 62

6.3.2 Data model . . . 63
6.3.3 Parser threads . . . 65
6.3.4 Grouping requests . . . 66
6.3.5 Identifying objects . . . 67
6.3.6 Sequential operation . . . 69
6.3.7 Tolerance to malformed data . . . 70
6.3.8 Saving the diagrams . . . 71
6.4 Line routing . . . 71
6.4.1 Analysis . . . 71
6.4.2 Algorithm description . . . 72
6.5 Additional views . . . 73
6.5.1 Call stack view . . . 73
6.5.2 Sequence diagram view . . . 74

7 User Documentation 75
7.1 Requirements . . . 75
7.2 Recording . . . 75
7.2.1 64-bit versions of Windows . . . 75
7.2.2 WinTrace GUI . . . 76
7.2.3 Console . . . 76
7.3 Opening previous traces . . . 77
7.3.1 Index file . . . 77
7.4 Browsing . . . 77
7.5 Understanding request diagrams . . . 78
7.5.1 Object view . . . 78
7.5.2 Call tree view . . . 79
7.5.3 Sequence diagram view . . . 80
7.6 Searching . . . 80
7.7 Export . . . 81
7.7.1 XML . . . 81
7.7.2 Diagrams . . . 82
7.8 Kernel objects reference . . . 82

8 Developer Documentation 85
8.1 Technologies . . . 86
8.2 Directory organization . . . 86
8.3 Events . . . 87
8.4 File format . . . 87
8.4.1 Trace file . . . 87
8.4.2 Executive objects . . . 88
8.4.3 Events . . . 89
8.5 Driver . . . 89
8.5.1 Executive object listing . . . 90
8.5.2 Hook engine . . . 90
8.5.3 Hook definitions & handlers . . . 91
8.5.4 Event ring buffer . . . 91
8.5.5 64-bit interrupt hooks . . . 92
8.5.6 Process filtering . . . 92
8.5.7 Reliability . . . 92

8.6 ETR Library . . . 93
8.6.1 Driver controller . . . 93
8.6.2 Trace file . . . 93
8.7 WinTrace GUI . . . 94
8.7.1 Data model . . . 94
8.7.2 Background workers . . . 96
8.7.3 UI and Qt models . . . 98
8.7.4 Utilities and helpers . . . 101
8.8 Console utility . . . 102

9 Customization **103**
9.1 Building WinTrace . . . 103
9.2 EvtGen code generator . . . 103
9.3 Writing a new hook . . . 104
9.4 Driver code segments . . . 107
9.5 GUI code snippets . . . 107

Conclusion **109**

Bibliography **111**

1. Introduction

Operating systems are an important component of a computer. Understanding how they work internally is considered useful for developing good applications. Most IT-oriented university degrees include operating system courses as part of their curriculum. However, modern operating systems are very complex and consequently difficult to understand.

This thesis will focus on helping the students understand the *Windows* operating system and how it works internally. *Windows* was chosen, because it is a modern succesful operating system, running on both server and desktop machines. Moreover, learning about the internals of a system the students are likely to be using has certain appeal to it.

The operating system's kernel typically has several responsibilities: input/output, memory management, process management etc. To limit the scope of our work, this thesis will mainly focus on the way the *Windows* kernel handles *I/O* requests and how does it solve the various related problems.

1.1 Overview of Windows I/O

Before defining our goals, we would like to briefly to introduce the most important concepts specific to the way *Windows* handles *I/O*. The topic will be later revisited in Chapter 2 in more detail.

The starting point for understanding *I/O* in *Windows* is the common abstraction it uses to keep track of connected devices, loaded drivers and open files. All three are known as *executive objects* and can be handled uniformly, regardless of their type (*executive object* can be seen as base class). Since devices, drivers and files are central to the *I/O*, understanding *executive objects* is crucial.

One of the common properties of *executive objects* is that they can have a name. The name serves the same purpose as a file name does – it allows user-space applications to identify and access the *executive objects*. The analogy goes even further, since the names form a hierarchy, and thus resemble a file-system. By browsing this hierarchy, we can learn a lot about the drivers, devices and other objects managed by the kernel (`/sys` file-system in *Linux* is similar in this regard).

Windows also introduces a data-structure for representing the *I/O* requests, regardless of the type of request (e.g. read, write or control) and the device (e.g. harddisk, network card). This structure is known as *interrupt request packet* (*IRP*) and contains all the information necessary for completing the request. The *IRPs* are the backbone of the *I/O* system and they are also needed for some more advanced features in *I/O* handling.

Because the *IRP* contains all information related to the request, the *I/O* handling can be *asynchronous*. *Asynchronous* means that an *I/O* call to the kernel may return before the request is completely finished, so that the application can do other useful work instead of waiting for the hardware. The information inside *IRP* is then used to complete the request processing when the hardware is ready (typically signalled by an interrupt).

Another important feature, contributing to the modularity of the *I/O* system, is that devices can be managed by multiple drivers. Each driver is responsible

for certain aspect of the device's functionality. The set of drivers associated with the device is called a *driver stack*. A typical example of a driver stack would be an *NTFS* driver providing a file-system functionality and an antivirus driver providing a real-time malware protection.

Windows also includes two mechanisms that enable the device drivers to handle interrupts more efficiently. The main perormance concern is *interrupt latency* – the delay from the time a device requests an interrupt to the point where it gets serviced by the operating system. A large *interrupt latency* can lead to e.g. lost packets or audio stuttering. To have a low *interrupt latency*, the processor should be running with interrupts enabled most of the time. *Windows* solves this by prioritizing interrupts (thus allowing at least some interrupts to be run). It also provides a mechanism for a function's execution to be deferred until interrupts are enabled again, called *deffered procedure call* (*DPC*).

As can be seen, *executive objects*, *IRPs*, *DPCs*, *drivers stacks* and their interaction is the basis for understanding the handling of *I/O* request in *Windows*, down to the point where it gets serviced by hardware. Of course, there are more concepts linked with *I/O* handling in *Windows*, but they are either details that are not needed to understand the general architecture, or additional features and extensions.

1.2 Problem statement

The general goal of this thesis is to design and implement a tool that would help with the explanation of the internal operation of the *Windows I/O* system, by showing the *executive objects*, *IRPs* and *DPCs* in a live system. Of course, such utilities exist and indeed served as an inspiration, but they are far from ideal for our purpose. To better understand the problems, let us look at two utilities, representative of the two major problems, and then evaluate them.

WinObj is a simple tool for viewing the *executive objects* in a system. It is a graphical tool, with the ability to view the namespace as a tree and browse the individual directories. Additionally, it can show basic information about each *executive object*.

WinDbg is a debugger, but unlike regular debuggers, it can pause and inspect the whole operating system, not just a single program. It has commands to show nearly any data-structure in the *Windows* kernel, including *executive objects*, *DPCs* and *IRPs*. The advantage is having the full power of the debugger available, including the ability to set-up breakpoints and observe interesting processes as they happen.

Each of the tools has its advantages and disadvantages. While *WinObj* is easy to use, it is very limited. It can only show *executive objects* that have a name. The information shown about the objects is very basic – for example, there is no way to find which devices are managed by a particular driver.

WinDbg, on the other hand, is very powerful and can be used to inspect all aspects of the kernel. However, it is hard to use, since it is basically a text-based debugger and its huge number of cryptic commands can be intimidating for novice users. On top of that, configuring *WinDbg* traditionally involves connecting two instances of *Windows*, one running *WinDbg* itself, the other one running the observed system.

WinObj and *WinDbg* represent the opposite ends of "limited but easy" and "powerful but hard to use" spectrum. Our aim is to design the tool in such way that it will display more information than *WinObj*, but still retain a graphical presentation that does not overwhelm the user, while avoiding the problems of difficult configuration. This will require a compromise between the amount of information presented and the ease of use.

1.3 Goals

If the tool for inspecting the internal operation of *Windows* should be useful to students, it must offer advantages over the existing utilities (so far we have mentioned *WinDbg* and *WinObj*). The design of the tool can be broken in three major parts:

1. Evaluate the existing utilities that can be used for inspecting the *I/O* system and determine which functionality is not covered by the existing easy to use tools. Propose a feature-set that will cover the most important missing *I/O* concepts.

2. Find a suitable way of obtaining the information, by evaluating the existing *APIs* and alternative unsupported methods. Keep in mind that the information collection method should not require complex set-up and should support recent version of *Windows* (that is *Windows 7* and *Windows 8*, at the moment of writing this thesis). Both *x86* and *x64* platforms should be supported.

3. Choose a graphical representation for the collected data. The representation should be easy to understand, using common concepts like block diagrams, trees etc. It should allow the student to easily explore the available information, like *WinObj* does.

These are our three primary goals, however, the tool has the potential to be useful for driver development, both as a learning tool and a debugging tool. If the tool is general and powerful enough, it is and advantage, but advanced features should not contradict the design goal of being easy to use.

1.4 Required knowledge

Before tackling the first goal (evaluating existing utilities), the reader should be familiar with the basic architecture of *Windows NT* operating system, in the area of *I/O* handling. Should the reader feel that he/she already knows about *driver stacks*, *DPCs* and *PnP*, he/she can continue with Chapter 3.

If this is not the case, Chapter 2 explains these concepts at the level required for our work. For a much more in-depth explanation of the *NT* kernel internals, refer to one of the *Windows Internals* [4] books and to *Kernel-Mode Driver Architecture* section [6] on *MSDN*.

Otherwise a general familiarity with the operation of a classic monolithic kernel is assumed: threads, processes, synchronization primitives, virtual memory, kernel vs. user mode and interrupt handling.

Since the tool is fairly low-level, we will often mention mechanisms specific to the *x86* architecture, like *IDT*, but we will try to explain them briefly when used. *Intel*'s *Software Developer's Manual* [3] is the complete and authoritative reference for these topics, although it may be a bit indigestible.

2. I/O in NT kernel

In the introductory Chapter 1, some concepts like *interrupt request packets* or *executive objects* were introduced. This chapter will explain them properly, in the breadth that is needed for further discussion in this thesis.

The intention is *not* to write a device-driver development tutorial. For this reason, the focus will be on the general concepts, not on technical details. More advanced topics like user-mode drivers, I/O cancellation, minidrivers or *Kernel Mode Driver Framework* will not be covered either. Similarly, when we talk about fields of some data-structure, we only mention the ones that are most important and relevant to the concepts we are explaining. Readers that want to know more, should visit the *Kernel-Mode Driver Architecture* section [6] on *MSDN*.

The explanation will begin with the relationship between *NT* kernel and the rest of the system (section 2.1). After that, the focus will move on to the topics we have already briefly mentioned in the introduction (Chapter 1), but in greater detail, that is drivers, devices and files (section 2.2), *executive objects* in general (section 2.3), *driver stack* (section 2.4), *I/O* requests (section 2.6) and the problem of efficient handling of interrupts (section 2.7). One of the topics not mentioned in the introduction is the *Plug and Play* support. Althought it is an addition to the basic *I/O* model, it is important for the modern system and we cover it in section 2.8.

2.1 NT API layers

To better understand the kernel of the contemporary *Windows* operating system (like *Windows 8*), let us look in the history. *Windows 8* is a descendant of *Windows NT 3.1*, first publicly released in 1993. Its kernel was designed by already experienced *David Cutler*, who had previously worked on the *VMS* operating system. From the start it was architected to be a modern, secure, portable, multitasking system with memory protection and still retains the same basic design – the topics covered by this chapter were already present in the original release of *Windows*, with the exception of *Plug and Play* support. For more information about the historical background of *Windows NT*, the reader can consult the *Showstopper* book [5].

One of the lesser-known facts is that NT was designed to run applications for multiple operating systems popular at the time: *OS/2*, 16-bit *Windows* and *POSIX*. Only later in the development cycle of the *NT* kernel did a 32-bit version of the *Windows API* became the prefered way of writing applications. The consequence is the existence of a common base layer, called *Native API*, an undocumented *API* hosted in the `ntdll.dll` library. All the remaining *APIs* call into this undocumented native layer, while the *Native API* in turn communicates directly with the kernel using system calls.

The diagram of this organization is in figure 2.1. At the top layer are the APIs intended for use by regular applications. The *Windows API* is the primary one, and other *APIs* may rely on it for some functions. However, in the end the *Windows API* passes the functions calls to the the *Native API*.

The *Native API* then performs a system call to pass the request to the kernel.

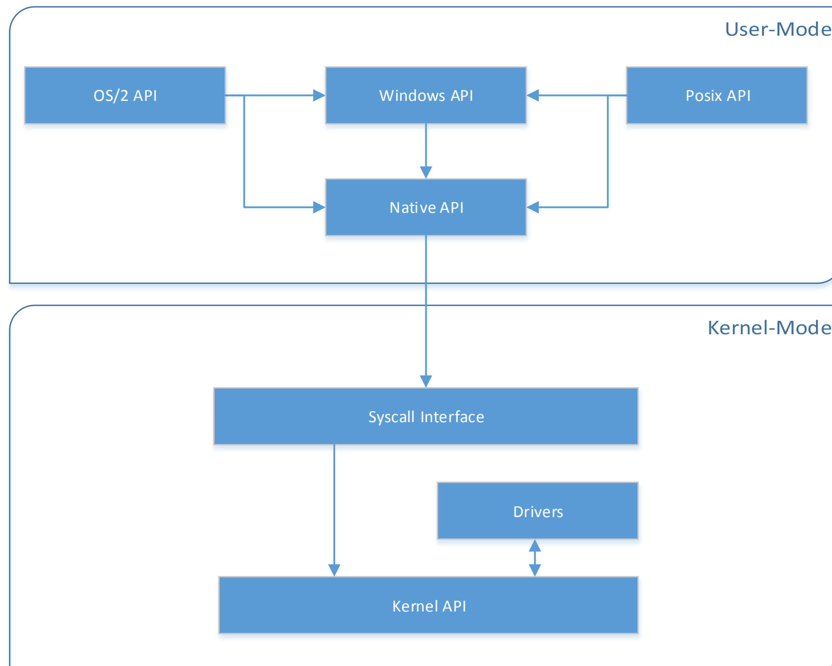


Figure 2.1: *API*s provided by the *Windows* operating system.

The functions in the *Native API* closely mirrors the internal organization of the kernel and indeed most functions in the *Native API* have direct counterpart in the kernel.

Apart from the *Native API* counterparts, the kernel also contains a lot of functions for accessing hardware, managing interrupts, using kernel heap etc. We will call all of the functions inside the kernel the *Kernel API* (they go under the name *Driver Support Routines* on *MSDN*).

When the call from the *Native API* gets to the handler in the kernel, the arguments are checked for validity, and the requested function in the *Kernel API* is called. This kernel function is finally responsible for doing the heavy lifting and servicing the call. Often the functionality is entirely implemented inside the kernel (such as waiting for an object, mapping memory).

For *I/O* functions, however, the kernel must consult a driver for the device in question. In case of *NT*, drivers are *dynamically linked libraries* (*DLL*s) loaded into the address space of the kernel, but with `.sys` extension. The drivers provide a collection of functions for reading from, writing to and otherwise managing a connected device of given type.

2.2 Devices, drivers and files

The main goal of the *I/O* system in the kernel is to provide hardware-independent access to devices connected to the system. To understand how *NT* does that, we must understand how it represents the connected devices, their drivers and files opened on those devices.

The *NT* kernel uses a *C* structure called `DEVICE_OBJECT` for keeping track of

connected devices. This structure contains a pointer to the driver managing the device, any driver-specific device data, the number of times this device is opened and the security descriptor, identifying who can access the device.

For representing the drivers, the system uses a `DRIVER_OBJECT` structure. The word *driver* can refer to either this structure, or to the code of the driver in the form of `.sys` file, but this rarely causes confusion, since each driver has corresponding `DRIVER_OBJECT` and vice versa.

For us, the most important part of the `DRIVER_OBJECT` is an array of pointers to functions. These functions are responsible for writing to, reading from, opening and otherwise manipulating the device managed by this driver. This array is called the driver *dispatch-table* and is how the kernel communicates with the driver. Using an analogy from *C++*, a driver can be thought of as a class and the device is instance of this class.

As on most systems, before a particular file is accessed, it must be *opened*. On *NT* this is done using the `NtCreateFile` function (corresponds to `CreateFile` in *Windows API*). The file to be opened is identified by a path – but in different format then the usual *Windows*/*DOS* path. Suppose that the user wanted to open a file named `C:\TEST.TXT`. The file-name gets translated to `\GLOBAL??\C:\TEST.TXT` before calling `NtCreateFile` in the *Native API*.

The `\GLOBAL??\C:` is the name of the device where the file resides and is used by *NT* to find the associated `DEVICE_OBJECT`. The `GLOBAL??` prefix signifies that the name is a system-wide driver-letter mapping. The name translation is more complex then described, but for our purposes we only need to know how devices are named internally in *NT*.

The remainder of the path is the name of the file on the device, in this case `\TEST.TXT`. Some devices, serial port for example, do not have a file-system, but still can be opened. In that case, the path consists only of the device name, like `\GLOBAL??\COM1`. *Windows* does not distinguish between files and devices in any important ways and when we speak about *open file*, it can mean an *open device* as well.

Each time a file on a device is opened a new structure called `FILE_OBJECT` is created, to represent the open file. The structure keeps track of current file position, the underlying device, permitted access, locks etc. Because the `FILE_OBJECT` will be used for any subsequent *I/O* operations, the driver can use it to cache data about the file for quicker access (in its `FileContext` member).

Since the pointer to the `FILE_OBJECT` can not be returned to the user-space application, an identifier called *handle* is created and the mapping from the handle to the `FILE_OBJECT` is stored in the *handle table* of the calling process (similar to file descriptor on *Unix*).

Once the file is opened and the process gets a handle, it can use the handle to perform all *I/O* operations, such as reading and writing. The operating system can perform the *I/O*, because the `FILE_OBJECT` contains the file name and the underlying device. When the program is done manipulating the file, it closes the handle using `NtClose` and the associated `FILE_OBJECT` is destroyed.

2.3 Objects

The previous section 2.2 has introduced three data structures: drivers, devices and files. They have properties in common with lot of other kernel structures. All three are referencou-counted. Devices have names (like `\GLOBAL??\COM1`) and their use can be restricted using security descriptors. *File objects* can be accessed from user-space, using handles.

Let us look at another kernel structure - the mutex synchronization primitive. It too can be named, accessed using handle, protected using security descriptor and its lifetime is managed using reference-count. It makes sense to abstract these four traits into a general structure.

This common abstraction is called an *executive object* (or sometimes just *object*). The component of *NT* kernel called the *object manager* is responsible for providing the common functionality. Under the hood, each *executive object* is a heap-allocated data-structure, preceded by an `OBJECT_HEADER` structure that maintains the common information.

For illustration, the memory layout is shown in 2.2, with the `FILE_OBJECT` as an example. The file object starts with the common object header, identifying it as a file object. The number of handles and pointers referencing it is also maintained in the header. Information about the security can also be stored there, but it is unused for a `FILE_OBJECT`. Security information only makes sense for objects that can be *opened*, like devices or synchronization primitives, but the `FILE_OBJECT` already represents an opened file. User-mode components refer to the file object indirectly, using the process handle table. The kernel-mode components can also use handles, but have the option of referring to the object directly by the pointer.

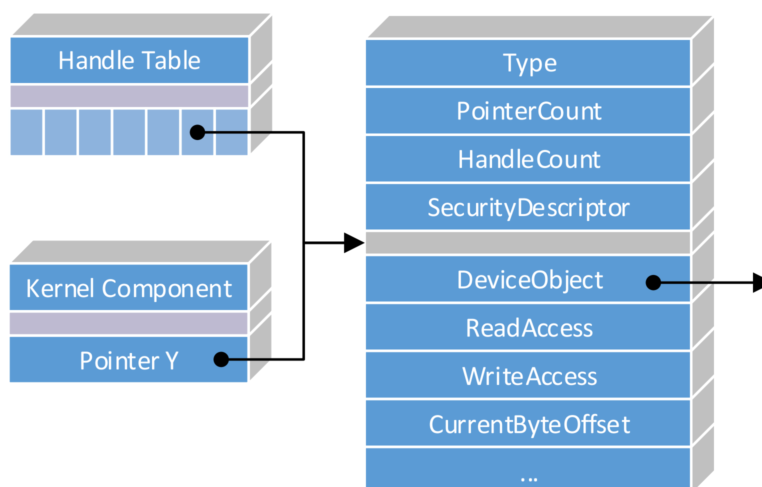


Figure 2.2: Layout of the *executive object*, `FILE_OBJECT` in this case.

Windows uses *exectuive objects* as a basis for wide range of data structures, from synchronization primitives to processes. For our purposes, *executive objects* are most important as the basis for devices, drivers and open files.

2.3.1 Naming

One of the important traits of the objects is that they can be named. An example was already given in section 2.2: devices have names, like `\GLOBAL??\COM1`.

The reader may have already guessed that object names in *NT* form a hierarchy, very similar to the *Unix* file system hierarchy (but kept only in memory). The parts of the hierarchy, such as `\GLOBAL??\` are called *directories* and are also represented by *executive objects*. The hierarchy is then often called *object manager* namespace. It can be explored using the *WinObj* [18] tool, as seen in figure 2.3.

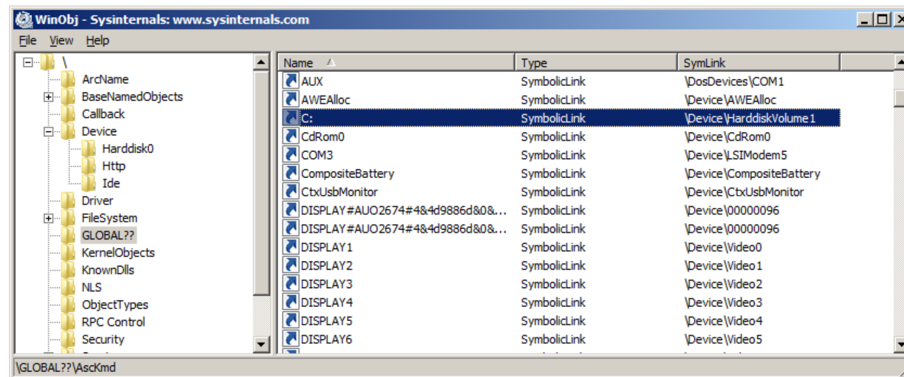


Figure 2.3: Listing of the *object manager* directory, `\GLOBAL??\` in *WinObj*.

Note that this object hierarchy is only visible at the level of *Kernel API* and *Native API*, but no such concepts exists in the *Windows API*. Devices at *Windows* (and *DOS*) level have have only simple names like `C:` or `COM1`.

2.3.2 Symlinks

It is common that a device has multiple names. For example, since the days of *DOS*, *PRN* is an alias for *LPT1*. As a more recent example, an *USB* device may register itself under several names, if it supports several functions.

This functionality is realized using *symlinks*. *Symlinks* are *executive objects*, which point to another part of the *NT* namespace, in a similar way file-system *symlinks* do. Indeed, no devices really reside in the `GLOBAL??` directory, but are all located in the `Device` subtree. All objects in `GLOBAL??` directory are symlinks to the real device objects (but this is transparent to user programs).

The *WinObj* example in figure 2.3 displays the real name of the devices in the rightmost column, called “SymLink”.

Please note that there is no counter-part to hard-links and each *executive object* may have only one name.

2.4 Driver stacks

There are cases when a behaviour of a third party driver needs to be changed. For example, an additional functionality needs to be implemented atop of an existing driver, or a bug fixed. This section will explain the way this feature works.

This is typical for a file-system drivers, where additional drivers may provide real-time anti-virus protection, versioning or auditing on top of the original file-system driver [1]. These additional drivers each have a chance to handle an incoming *I/O* request and either decide to complete it themselves (e.g. block access to infected file) or pass it to the original driver. They are also called *filter drivers* for this reason.

The *NT* kernel is aware of this need and allows additional drivers to be *attached* to an existing device. The terminology here is a bit overloaded, because the way this concept is implemented is by using the `AttachedDevice` member of the `DEVICE_OBJECT` to form a stack of *devices* (each managed by a different driver). Thus this organization is called both *device stack* and *driver stack* (even on *MSDN*). The term *device* could then refer to individual *device object* in the stack or the whole stack representing the hardware device. We use the term *device node* to refer to the whole device stack associated with one hardware device.

A simple *driver stack* can be seen in figure 2.4. In this case, there are two *device objects* in the stack. The bottom driver is the regular *NTFS* driver, with an anti-virus protection driver attached. When an *I/O* request comes, the system directs it to the attached filter driver instead of the original one. Multiple drivers can be attached to one (forming a stack), in which case the last one in the chain (also called *topmost*) is called.

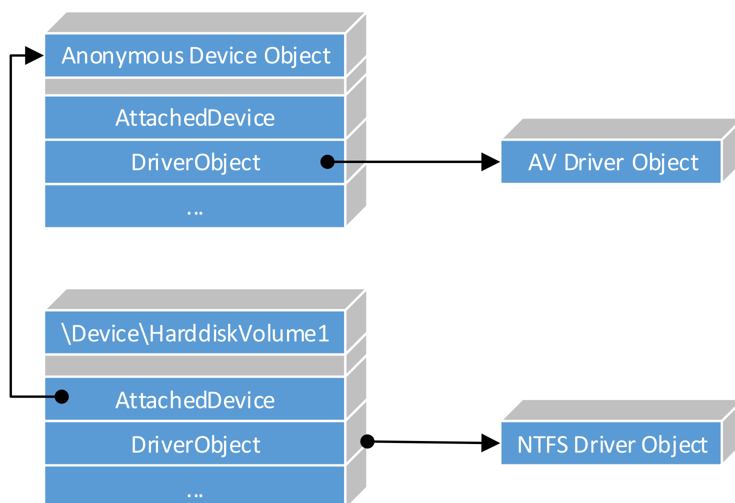


Figure 2.4: Illustration of a simple driver-stack.

Also note that both *device objects* in the stack are *executive objects* and so both can be independetly named.

The concepts of *driver stacks* is important for file-system and block-device drivers. Moreover, it is important for the implementation of *Plug and Play* support that will be covered in section (2.8).

[1]File-system developers are now encouraged to use a new way of modifying file-system behaviour, called minifilters.

2.5 Recapitulation

At this point, it should be clear how the drivers, devices and actual connected hardware are related. To reiterate, when a user program wants to write into a file, the following actions happen:

1. The user program uses *Windows API* `CreateFile` to open `C:\test.txt`.

2. `CreateFile` then rewrites the path to `\GLOBAL??\C:\test.txt` and calls `NtCreateFile` in `ntdll`.

3. `NtCreateFile` calls the kernel, using syscall.

4. Kernel calls the `NtCreateFile` from *Kernel API*.

5. `NtCreateFile` locates the device node in the object manager namespace. Because `\GLOBAL??\C:` is a *symlink*, the request gets redirected to the path `\Device\HarddiskVolume1`.

6. `NtCreateFile` checks the security descriptor of `HarddiskVolume1`.

7. `NtCreateFile` creates the `FILE_OBJECT`, fills in the name of the file (`test.txt`), pointer to the `DEVICE_OBJECT` and asks the driver attached at the top of the `HarddiskVolume1` device stack to open the file. The driver will either handle the request itself or pass it down the chain of attached drivers.

8. If the creation was successful, a handle is created for the `FILE_OBJECT` and returned to the user-space program.

Once the file is opened, the user program owns a handle to the requested file and can issue the write operation:

1. The user program calls *Windows API* `WriteFile` with the handle from the open request and the data.

2. `WriteFile` calls `NtWriteFile` in `ntdll`.

3. `NtWriteFile` calls the kernel, using syscall.

4. Kernel calls `NtWriteFile` from *Kernel API*.

5. `NtWriteFile` resolves the handle and obtains the `FILE_OBJECT` from the open request. It checks if the file was opened with write permissions.

6. `NtWriteFile` asks the topmost driver of the file's device node to write the data. The driver will either handle the request itself or pass it down the chain of attached drivers.

2.6 I/O

We have shown how the system keeps tracks of files, devices and drivers and how it can find the suitable drivers for handling the given *I/O* request. The *I/O* request is directed to the *topmost* driver in the stack, and a function from its *dispatch table* will be called. How do these functions look?

```
NTSTATUS (DispatchDoSomeIO*) (DEVICE_OBJECT* DeviceObject, IRP* Irp)
```

All dispatch routines look the same, regardless of the type of the *I/O* request. The key is in the `IRP` structure. This structure is a self-contained description of the whole *I/O* request and has all necessary fields for describing all the types of the request (open, read, write etc.). The name stands for *I/O request packet*.

For example, a write request should certainly fill in these fields of the *IRP* (will be filled in by `NtWriteFile` in the kernel):

1. The function requested, in this case having code `IRP_MJ_WRITE`.
2. The `FILE_OBJECT` pointer on which we should operate.
3. The buffer containing the data to be written.
4. Size of this buffer.
5. File position from which to read.
6. The thread that has called `NtWriteFile` (remember that *IRP* is self-contained).

The *IRP* also contains a field for returning information: the number of bytes read/written and the error code.

Having a central structure describing any type of *I/O* allows the kernel to apply unified treatment to many operations and to reduce the number of arguments to functions. Also, because *IRPs* are self-contained, the request can exist on its own (it is not an implicit part of a function call), which in turn allows *NT* to have asynchronous *I/O*.

2.6.1 I/O locations

IRPs are typically shared by all drivers in the driver stack. In other words, when a driver needs to pass a request down the chain, it reuses the existing *IRP*.

IRP contains members for storing per-driver information. The information is stored in `IO_STACK_LOCATION` structure. There is one *I/O location* for each driver on the driver stack.

Fields common to all drivers on the stack (e.g. buffers, thread) are stored in the *IRP* itself, whereas the request details – the function code, data size, `FILE_OBJECT` etc. – are stored in the *I/O locations*. This gives each filter driver a chance to modify the request it wants to pass to the driver below it, without constructing new *IRP*.

2.6.2 Asynchronous I/O

The *dispatch routines* as we have described them so far are *blocking* – when they return, the *I/O* is already completed. However, if the device is slow compared to the *CPU*, most of the time will be spent by waiting for the device to react. More useful work could be done in the meantime.

The solution is to let the *dispatch routine* return as soon as the request has been handed over to the hardware. The application is then notified about the completion of the *I/O* by signaling a synchronization event (or a *NT* specific mechanism called *I/O completion port*). This kind of *I/O* handling is called *asynchronous* and is generally accepted to have better scalability.

NT has supported asynchronous *I/O* from the start. If a driver returns `STATUS_PENDING` from its dispatch routine and sets a pending flag in the *IRP*, it signals to the caller that it has *not* completed the request and will do so at a later time. The function `IoCompleteRequest(IRP*)` is later called by the driver, when the hardware has finished the request. This functions is responsible for the last steps of *I/O* processing, such as notifying the caller.

If the driver wants to complete the request synchronously, for example when the data is in cache, it can still do so. It just needs to first complete the request using `IoCompleteRequest` and then return status code other than `STATUS_PENDING` from its dispatch routine, so that the caller will know the request is already completed.

2.6.3 I/O completion routines

The *asynchronous I/O* processing, as described, has one disadvantage. The filter drivers that have passed the *IRP* down the driver stack have no way of manipulating it after the request is completed, because they have already returned from their *dispatch routines*.

For this reason, each driver can associate so called *completion routines* with the lower-level driver's *I/O location*. When the lower-level driver finally calls `IoCompleteRequest`, the *completion routine* associated with its *I/O location* is invoked. To make matters more complicated, the *completion routine* itself may return `STATUS_MORE_PROCESSING_REQUIRED` to indicate that its driver needs more time to process the request and will call `IoCompleteRequest` at later time to resume the completion. If the *completion routine* does not interrupt the completion in this way, completion routines in the higher-level *I/O locations* are called. When all completion routines have been run, the *IRP* is finally complete and the caller is notified about it.

Completion routines are typically used by *filter drivers* to monitor the results of the *I/O*, or to clean up their internal structures associated with the request. Example of the completion routine operation and the organization of an *IRP* is given in figure 2.5. The stack consists of two drivers, the base NTFS driver and the anti-virus filter driver. Each of them is represented by its own device objects (see figure 2.4 for an example of driver stack). Both drivers will operate on the same file object.

Additionaly, the anti-virus drivers has registered a *completion routine*, that will be triggered when the *NTFS* driver has finished reading the data. At this

point, the data can be scanned for malicious code and the completion routine would return a failure code if the data is infected, denying the request. [2]

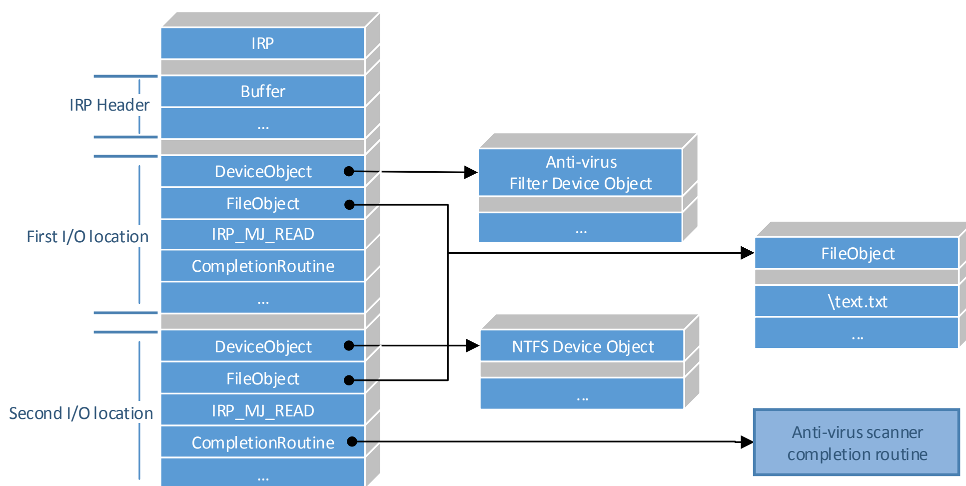


Figure 2.5: Illustration of a anti-virus scanner setting a completion routine.

Since most applications use traditional blocking *I/O*, it may look as if *IRPs* are rarely processed asynchronously. This is however not the case, because even if the user-mode application uses a blocking API, the *IRPs* are internally processed asynchronously and the API used by the applications simply blocks until the asynchronous processing is complete.

2.6.4 I/O without IRPs

The *I/O* handling process as described so far is used for majority of *I/O*, but processing *I/O* using *IRPs* tends to have higher overhead than regular function calls or direct hardware access. There are two instances in *NT* kernel, where the *I/O* requests are not represented by *IRPs*.

When application accesses a file, the system may decide to use the *fast I/O* dispatch routines first. These routines receive the parameters directly (they do not need an *IRP*) and are always synchronous. However, they may return `FALSE` to indicate to the operating system that it needs to use the regular *IRP*-based dispatch routines. Usually, *fast I/O* routines only perform the operation if all needed data is already in the cache. If it is not and hardware must be accessed, they return `FALSE`.

The second case is using a device directly connected to a *bus*. The driver controlling the device writes to the *I/O ports* or *memory mapped regions* directly, without using any other driver. A lower-level driver for a bus to which the device is connected may be used to map those *I/O* ranges into memory.

[2]If the program is reading the file byte-by-byte, the anti-virus scanner would not detect the malicious code. Commercial scanners use different approach.

2.7 Interrupts and synchronization

Handling interrupts brings additional complexity to the kernel. The main problem is that interrupts can occur at any time and the system may be in an inconsistent state.

Let us first look how a naive solution for this problem could be implemented and then explain its performance problems. Then the we will cover the way *NT* solves these problems, using two mechanisms already covered in the introduction – *DPCs* and *IRQL*.

2.7.1 Disabling interrupts

The kernel often needs to access global data-structures from different threads. An example of such data-structure is the list of threads that are ready to run, maintained by the scheduler.

This list needs to be accessed frequently, every time threads are created, become ready to run or wait. Because *NT* is a pre-emptively multi-tasked operating system, it needs a way of switching threads when they have been running for too long. An interrupt periodically generated by the system clock is used to switch threads, so it too must access this list.

To prevent corruption of the ready-to-run list we need a synchronization primitive governing the exclusive access. Ordinary synchronization mechanisms, like mutexes, are not usable, because they themselves need to access the ready-to-run thread list when the caller needs to wait for the primitive.

Spinlock is a synchronization primitive similar to the mutex, but instead of pausing the waiting thread, it *busy waits* – repeatedly checks if the lock is already free. However, interrupts must be disabled when the spinlock is held, to prevent the clock interrupt from firing and trying to lock the list again.

The frequent disabling of interrupts is the performance problem *NT* is trying to avoid. Disabling interrupts for a long time means that the delay between the hardware requesting interrupt and the processor responding (an *interrupt latency*) can be significant. This is undesirable, because some hardware needs a quick response. For example an audio card uses interrupt to signal that it is running out of data to play and needs a next part of a song [8]. Large interrupt latency could lead to buffer underrun and glitches in the playback. Network cards and other types of real-time hardware faces similar problems.

Apart from interrupts being disabled when the kernel is holding the spinlock, as described above, interrupts are also disabled in interrupt handlers. This is necessary to prevent re-entrancy. *NT* is able to cut down the time spent in both these spin-locks and in interrupt handlers.

2.7.2 DPCs

NT offers drivers a mechanism to reduce the time spent in their interrupt handlers and in turn reduce the *interrupt latency*.

It is based on the presumption that interrupt handlers have two parts. The first part is time critical, and needs to be serviced immediately. The second part however, can be delayed. The second part is mostly concerned with additional processing of data and includes the call to `IoCompleteRequest`.

The mechanism for running the second part is called *deferred procedure call* (*DPC*) and defers the execution until all time-critical parts of interrupt handlers are completed. The reverse is also true, the *DPC* can be interrupted by the critical interrupt processing.

As an example, the timeline in figure 2.6 shows how regular threads, hardware interrupts and *DPCs* interrupt each other. Threads can be interrupted by both *DPCs* and hardware, while *DPCs* can be interrupted only by hardware interrupts.

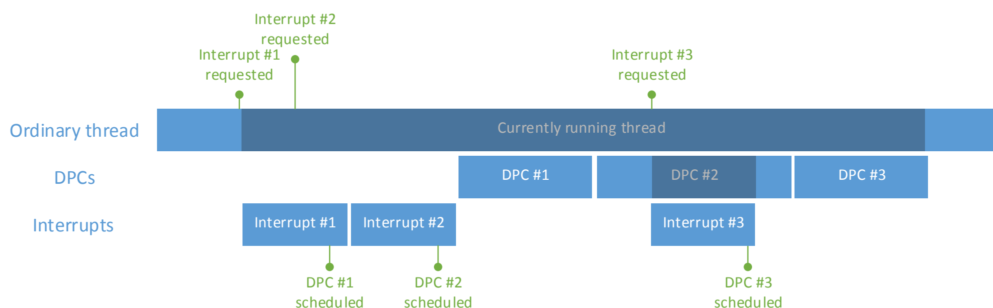


Figure 2.6: Timeline of interrupts and *DPCs* running. The gray areas represent interrupted code.

Now let us return to the synchronization problem from section 2.7.1, where both interrupt handlers and regular threads needed to manipulate the scheduler's ready-to-run list. This lead us to disabling interrupts when the list's spinlock was held.

If the scheduler takes advantage of *DPCs*, it can move the code for switching threads from a clock interrupt handler to the associated *DPC*. Therefore, a regular thread locking the ready-to-run list does not need to disable interrupts, but just *DPCs*, further reducing the time spent with interrupts disabled.

2.7.3 IRQL

The introduction of *DPCs* in previous section (2.7.2) has decreased the time spent in interrupt handlers. *DPCs* actually resemble hardware interrupts, but have lower priority – this can be seen from their behaviour in figure 2.6. The difference is only that they are triggered by the system, not by external hardware.

The priorities of *DPCs*, interrupts and regular code are called *interrupt request levels* (*IRQL*) in *NT* terminology. They are represented by numeric constants defined in the kernel headers, where numerically lower value means a lower priority. The basic rule of *IRQL* is that code with higher *IRQL* may interrupt code with lower *IRQL*.

So far we have seen regular threads, *DPCs* and interrupts. `PASSIVE_LEVEL` corresponds to regular threads (the lowest priority), `DISPATCH_LEVEL` is the priority of code running in *DPC*. Hardware interrupts have various levels, according to the device, but higher than `DISPATCH_LEVEL`. *NT* defines several other levels but they are rarely used by device drivers.

A lot of behaviour in *NT* is described in terms of *IRQL* and there is additional terminology. If code wishes to disable interrupts or *DPCs*, it can *raise* its *IRQL*. As we have seen, disabling *DPCs* or interrupts is often done when acquiring spinlock, so it is often called *acquiring spinlock at certain level*. In our scheduler

example, we would say that we need to acquire the spinlock at `DISPATCH_LEVEL`. Also lot of functions *require* at least certain *IRQL*, meaning that their behaviour is undefined, unless running at that *IRQL* or lower.

2.8 Plug and Play

Features like device stacks, object manager namespace and *IRQL* hierarchy were already present in the original *Windows NT 3.1* and form the core of the *I/O* system. The last important *I/O* piece of a modern consumer system is the *Plug and Play* (*PnP*) support. This refers not only to the ability to connect devices to the running system (hotplugging), but also to the autodetection of devices like *PCI* cards. Originally, this required manual driver installation and often configuration of *I/O* ports and *interrupts*.

Windows 2000 was the first in the *NT* line of systems to introduce a support for automatically enumerating and configuring the devices connected to the computer. The name *Plug and Play* (*PnP*) reflects the fact that user should be able to connect the device and use it right away.

This section will cover the way *PnP* support is implemented in the terms of the *device objects* and *drivers*. However, the concrete interface, such as the new dispatch functions used for communication with the kernel *PnP* manager will not be be described. More advanced concepts like device removal, power relations etc. will not be mentioned either.

2.8.1 Enumeration and PnP driver stacks

There are two phases of installing new *PnP* devices. The first phase involves noticing that a new device was added to the system. This is the task of a bus driver and the *bus device node* it manages. The *bus device node* represents a bus, port, hub or any other entity to which other devices can be connected.

The driver *enumarates* the physical devices connected to the bus and if it notices a new one, it must create a new *device node* for it. However, at this point the driver for the detected device is not installed, and so the bus driver itself acts as the driver for the device. The *device object* managed by the bus driver is known as *Physical Device Object* (*PDO*).

Let us use a video capture card connected to *PCI* slot as an example. When the *PCI* driver will notice a new device connected to the bus, it will create the *PDO* for it. However, this does not mean that the new device node knows how to tune the video card and receive channels. The purpose of the *PDO* is to represent the raw device. In the case of *PCI PDO*, it will allow access to the *PCI* configuration registers. The bus device and the *PDOs* are the two devices at the bottom of the example figure 2.7.

After creating the *PDO*, there is a second phase. The *PnP* manager is notified about the new device. It will ask the *PDO* about the device type. For *PCI*, this is the vendor and product ID stored in the configuration data. Based on this information, *Windows* can lookup the appropriate driver, load it and ask it to attach itself to the *driver stack* of the new device node. Thus the driver stack will be composed of the *PDO* and the new *device object*, called *Functional Device Object* (*FDO*). The *PnP* manager will configure the new device in cooperation

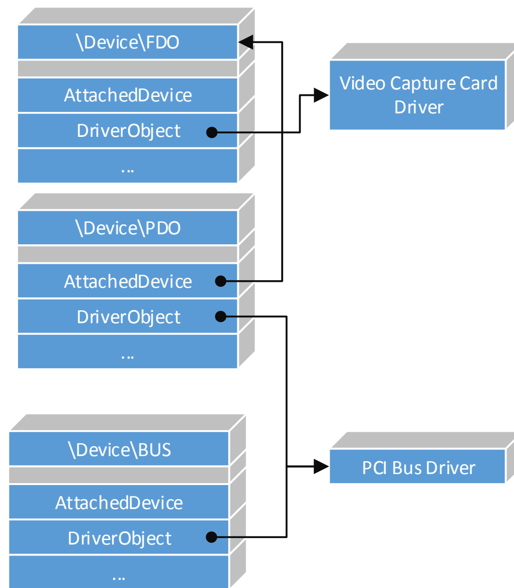


Figure 2.7: Organization of the devices configured by *PnP* manager.

with the *FDO* and video capture applications will be able to use the card. When a request to tune to a specific channel arrives, it is first handled by the *FDO*, which will issue hardware-specific commands to the *PDO*.

2.8.2 PnP tree

The *device nodes* in a system form a parent-child relationship, where the parent is the bus and the child is the connected hardware (which in turn can be a bus too).

This *device tree*, formed by the parent-child relationship, can be viewed by a tool distributed with *Windows*, called the *Device Manager*. By default it lists devices by their category, but can be made to show the *device tree* using the "Devices by connection" option in the menu.

The following example of this tree (figure 2.8) should help the reader realize the role *driver stacks* and *filter drivers* play in the *NT I/O* subsystem. To fit the example image on one page, we have omitted most of the devices and show only a subtree.

The root device of this tree is called "ACPI x86-based PC" and its driver is the *PnP* manager itself. The devices listed under the root node are not connected to any bus and are called *root-enumerated devices*. They represent the devices that are not *PnP* capable, possibly because they are integral part of the system or they are purely virtual devices.

One of those devices is named "Microsoft ACPI-Compliant System". This device is not a regular bus to which devices can be physically plugged in, but instead uses the *Advanced Configuration and Power Interface* (*ACPI*) standard to discover the devices present on the computer's motherboard. Excluding simple devices like buttons and laptop lid sensors, most devices are connected through the *PCI* bus. This includes the *USB controller* controller that hosts the root USB hub. The only device connected to the hub is the on-board *Realtek WiFi* adapter.

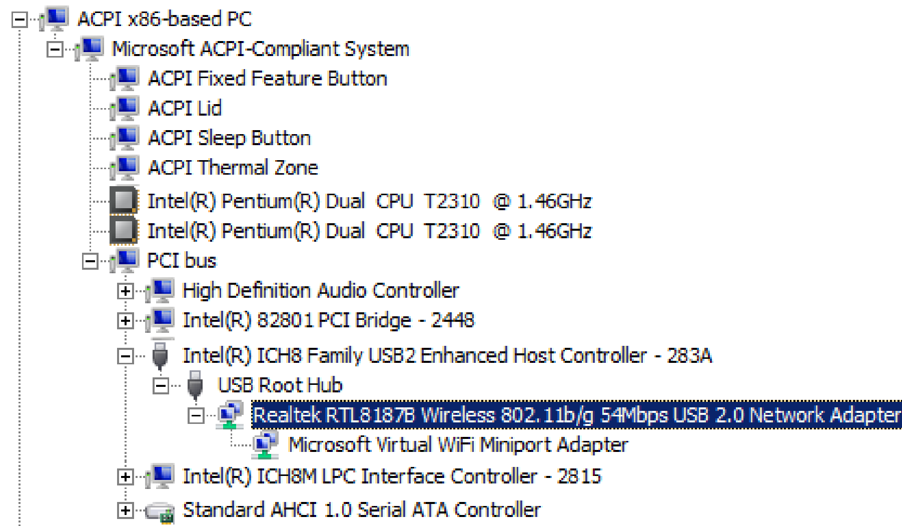


Figure 2.8: *Device Manager* MMC snap-in showing a portion of a device tree on a laptop.

The figure 2.8 also illustrates another interesting phenomenon, the use of virtual devices and gives another example of how filter drivers can be used. In this case, the *Realtek WiFi* adapter should not have any child devices, since no other devices can be attached to a *WiFi* adapter.

However, *Windows 7* includes a technology for connecting to multiple networks with a single wireless card [9]. To the user it looks as if two adapters are present on the system – one is the original one, the second one is a new *virtual* adapter (since there is no corresponding physical adapter on the system). The technology works by rapidly switching the physical adapter between the two connected networks.

This functionality is not provided by the *Realtek's* driver, but instead by a filter driver sitting above the *Realtek's* driver. First the filter driver maintains the illusion that the original adapter is connected to only one network and second it changes the *Realtek* network adapter into a *bus*, to which the virtual adapter is connected. The device nodes used are illustrated in figure 2.9.

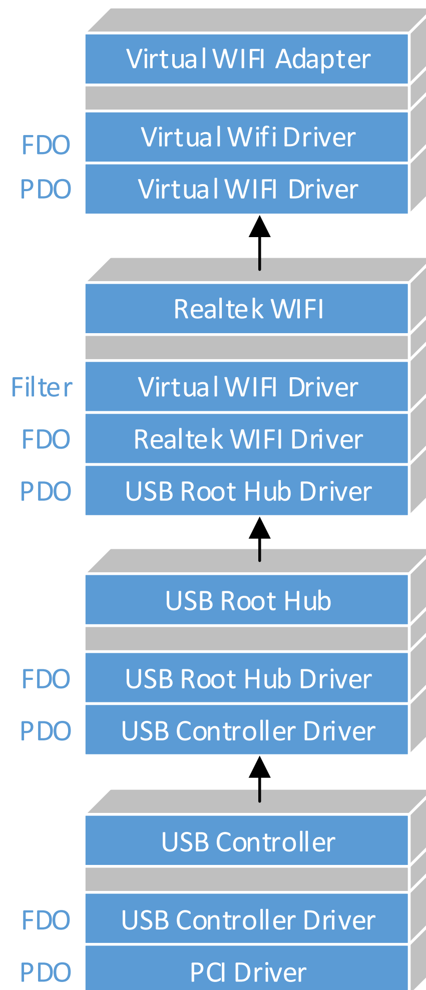


Figure 2.9: Virtual Wifi driver stack illustrating the situation from figure 2.8.

3. Existing utilities

Our first goal, defined in Chapter 1, is to evaluate the existing tools that could be useful to a student learning how *Windows NT* handles *I/O* requests. To design our tool, it is important to know what areas of the *I/O* system the existing tools cover and how user-friendly they are.

The tools were selected on the basis of being useful to the student of *NT I/O* internals. This excludes tools that either focus on different part of the operating system then *I/O* and tools that are intended for specific driver development tools.

The selected tools come mainly from three sources: *OsrOnline* [14] (a website for driver developers), *SysInternals* [7] (authors of the respected *Windows Internals* [4] book) and tools distributed with *Windows* and its *Windows Driver Kit* [6]. Readers who want to know more about *Windows* internals (not specifically the *I/O* subsystem) are encouraged to visit the two mentioned sites and experiment with the utilities.

The tools can be divided into three categories. The first one are debuggers, represented by *WinDbg*. They are hard to use, but are very powerfull and flexible. The second category are specialized tools for inspecting the data-structures kept by the *NT* kernel, such as *executive objects*. *WinObj* is a typical example of such a tool. The last category are tools for monitoring activities in the system, such as the *Event Viewer* application distributed with *Windows*

To have a better idea what are the *I/O* concepts that the tools should explain, this chapter will start with the review of the most important ones, based on the introduction to the *NT*'s *I/O* architecture in Chapter 2. After that, the available tools will be described, grouped by the categories mentioned above, followed by the analysis of their weakness and what types of functionality is not covered. Finally, we will propose the feature-set for our new tool, based on the previous analysis.

3.1 I/O system concepts

The basic principles of the *Windows I/O* were explained in Chapter 2. To better understand the goals for our new tool, the concepts from Chapter 2 will be repeated here (but without explaining them).

The concepts should be taken as a feature suggestions for our new tool. This does not mean that the tool must cover all of them. Some of the concepts are sufficiently covered by existing tools. For others, it might be technically difficult to gather any information about them.

The starting point for our explanation was the communications between the kernel and the applications. This involves the *Windows API*, *Native API* and the *Kernel API* and the translation between them.

The next area are the data-structures kept by the kernel and the relationships between drivers, devices and open files. Several *NT* specific concepts in this area are the *executive objects* and their concrete examples: *device objects*, *driver objects* and *file objects*. These objects are also important for explaining the *Plug and Play* (*PnP*) mechanism. The key *PnP* concepts are *functional device objects*, *physical device objects*, *filter drivers* and the relationship between buses and their children.

The next area is the actual handling of the *I/O* requests, centered around *interrupt request packets* (*IRP*). Because the *Windows I/O* system is asynchronous, the life-cycle of an *IRP* is complicated. The handling of an *IRP* may involve other *IRPs* (for sub-requests), driver *dispatch routines*, *completion routines*, interrupts and *deferred procedure calls* (*DPCs*). Unifying the interrupts and *DPCs* is the *interrupt request level* (*IRQL*), describing the priority level of a code.

Of course, all of the areas are related. For example, the *PnP* manager uses *IRPs* for discovering and configuring the devices. The *Kernel API* must translate the application's request into an *IRP* and relies on the *object manager* namespace for finding the correct device and its drivers.

The concepts mentioned above are sufficient for understanding most of the *I/O* system. There are certain areas we have omitted, because they are either rarely encountered (*I/O* cancellation, fast *I/O*), or they are special cases of the basic concepts (user-mode drivers, driver frameworks, minifilters, minidrivers).

3.2 Debuggers

Debuggers are useful for understanding the the internal operation of the *I/O* system. They can inspect memory, pause the debugged system, set-up breakpoints at interesting events and even change variables.

Our area of interest will be debuggers capable of debugging the whole operating system, as opposed to debugging a single program running on that system.

3.2.1 WinDbg

A natural choice for debugging an *NT* operating system is *WinDbg*, because it is well-integrated with the *NT* kernel and is the "official" *Windows* debugger, supported by *Microsoft*. It is distributed as part of the *Windows Driver Kit* [6].

WinDbg comes with the usual features one would expect of debugger, e.g. breakpoints, watched variables, expression evaluation, single stepping, symbolic names for memory addresses and line numbers. In addition to these basic capabilities, it comes with *debugger extensions* for displaying *NT* structures like *IRPs*, *executive objects* and *device stacks*.

Before using *WinDbg*, the user must set-up two connected computers (or at least virtual machines), if he wants to use the full power of the debugger. The first computer will run *WinDbg*, and debug the *Windows* operating system running on the other computer. The supported methods for connecting the two systems are direct serial port connection, special *USB* cable, *FireWire* or *Ethernet*.

The easiest way for a student to configure it is probably to set-up virtual machine with bridged mode networking and configure *WinDbg* to connect to this virtual machine. Serial port connection is also supported by virtual machines, but it is very slow. The other modes are not usable with any common virtualization software.

To have a better idea how *WinDbg* can be used and what are its problems, two examples of inspecting the *I/O* system will be given. One will show how to monitor system activity using breakpoints, the second one will show how to browse *executive objects*.

Monitoring dispatch functions

As an example, suppose that we want to illustrate how dispatch routines are called in order to handle *I/O*. *WinDbg* will be used to put a breakpoint on a dispatch routine call and display some details about the call.

Figure 3.1 illustrates such *WinDbg* session. Before explaining the actual commands and their purpose, notice the *GUI*. Even though *WinDbg* is a graphical debugger, it is controlled using text-based commands, entered into the command-line at the bottom. Only simple commands, like *Go*, are available in the toolbar and the menu. The output of the commands is also textual and is displayed in the output area in the middle of the *WinDbg* window. The output area also includes the entered commands, prefixed with `kd>`.

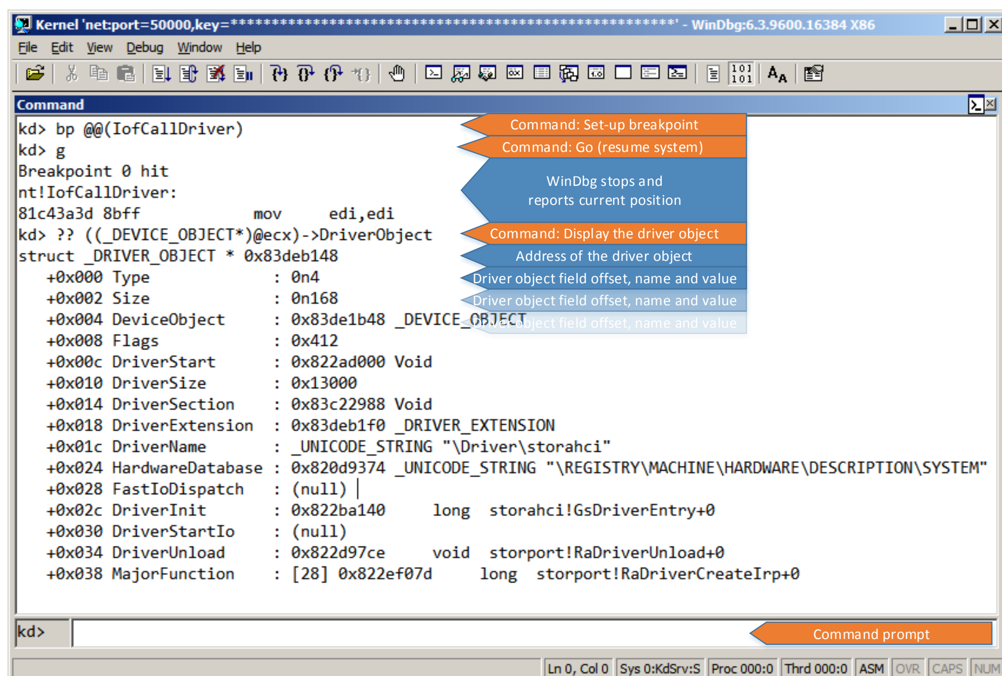


Figure 3.1: *WinDbg* session used to illustrate the calling of *dispatch routines*.

The *WinDbg* session in figure 3.1 was started by connecting to the debugged system and pausing it (this is not seen in the screenshot). Next, a breakpoint (`bp` command) was setup on `IofCallDriver`, a function that invokes the driver dispatch functions. This way, the debugger will pause when any dispatch function is called. The system was resumed (`g` command, short for go). However, the breakpoint was hit immediately and the debugger informs us that we are now in the `IofCallDriver` function and about to execute instruction `mov edi, edi`.

We would like to know which driver is going to be called. We can extract the `DRIVER_OBJECT` pointer from the `DEVICE_OBJECT` that `IofCallDriver` accepts as a first argument. A command for displaying *C* data-structures like `DRIVER_OBJECT` is `??` (*query* command). However, we must know that according to the calling convention, the first parameter to `IofCallDriver` is stored in register `ecx`. The debugger responds by listing all the fields in `DRIVER_OBJECT` and their values.

This example illustrates well the problems with using *WinDbg*. First of all, the

user must know the *WinDbg* commands and their syntax. Moreover, the user must find out where to set-up breakpoints, what variables to query etc. Sometimes it gets even more complicated and he/she must resort to querying registers, as in our example.

Inspecting executive objects

Purpose of this example is to demonstrate the *WinDbg* extensions for working with the *NT* kernel, concretely *executive objects*. Other extensions (not used in this example) can be used for browsing the driver stacks and the *Plug and Play* (*PnP*) tree.

Example session is recorded in figure 3.2. The `!object` command is used to show the `\GLOBAL??` directory in the *NT* namespace. The debugger first shows the properties common to all *executive objects* (memory address, reference counts etc.). It then proceeds to list the names and addresses of the objects stored in this directory.

```
1: kd> !object \\Global??
Object: 8be08f38  Type: (8533ae90) Directory
    ObjectHeader: 8be08f20 (new version)
    HandleCount: 2  PointerCount: 219
    Directory Object: 8be01110  Name: GLOBAL??

    Hash Address  Type          Name
    ---- -------  ----          ----
         921da328 SymbolicLink  vmx86
     33  8bf959d0 SymbolicLink  C:
         92cf5728 SymbolicLink  {DB2B4279-B5CF-4626-9DBA-32D0ECE44C87}
         8bf89e28 SymbolicLink  AUX
         8bfba168 SymbolicLink  MAILSLOT
         92cfbc68 SymbolicLink  NDISWANIPV6
         93385c28 SymbolicLink  HDAUDIO#FUNC_01&VEN_10EC&DEV_0660&SUBSYS_1179FF40&REV_100
         9331d1c0 SymbolicLink  HDAUDIO#FUNC_01&VEN_10EC&DEV_0660&SUBSYS_1179FF40&REV_100
         8bfba3d0 SymbolicLink  PCI#VEN_8086&DEV_2836&SUBSYS_FF401179&REV_03#3&11583659&0

1: kd> !object \\Global??\C:
Object: 8bf959d0  Type: (8533adc8) SymbolicLink
    ObjectHeader: 8bf959b8 (new version)
    HandleCount: 0  PointerCount: 1
    Directory Object: 8be08f38  Name: C:
    Target String is '\Device\HarddiskVolume1'
    Drive Letter Index is 3 (C:)
```

Figure 3.2: *WinDbg* connected to a *Windows 8* virtual machine, displaying part of the NT namespace using the `!object` extension.

Finally, the same command (`!object`) is used to find out the real device name of the *symlink* `\GLOBAL??\C:`. Using the `!object` command, it is possible to iteratively browse all *NT* objects.

3.2.2 Live WinDbg

The inability to run on the same system as the one it is debugging is natural limitation for a kernel debugger. If the entire operating system is paused by the debugger, including the debugger's user interface , there is no way to resume the system. For this reason, the user interface must not be part of the debugged system.

However, if the debugger is not used to set breakpoints, handle exceptions and break into the system, it is safe to run it on the same machine. Support for this was added to *WinDbg* by the *LiveKd*[12] utility from *SysInternals* [7]. The support is now included directly in recent versions of *WinDbg*.

All of the extensions for showing kernel objects work and the session seen in figure 3.2 could have been run on a live system. However, *LiveKd* does not solve any of the problems with *WinDbg*'s commands and terse output.

3.2.3 Third-party debuggers

There are alternative debuggers from third parties, which can be used instead of *WinDbg*. However, they mostly share the same problems (text-based, hard to learn) and are not well-integrated with the *NT* kernel. On the other hand, they have different approach to system debugging, that can eliminate the two-machine problem.

Virtual machines like *VirtualBox* and *VMWare* can also act as debuggers for the guest operating system. *VirtualBox* provides its own debugger [10] with custom syntax and commands. It does not provide even basic *Windows*-specific support, like symbol names. *VMWare* [11] allows the *The GNU Project Debugger* (*GDB*) or any other debugger using the same protocol to debug the *guest* operating system. Again, *GDB* is *Unix*-centric and does not support the *PDB* file format for *debug symbols*.

Debuggers that can debug the same system that they are running on are available (also called *single-host* debuggers). They solve the problem of requiring two systems, but due to their nature, require their own drivers and have portability issues. One of the early examples is *SoftIce* by *NuMega*, now discontinued. Another, contemporary examples are the *BugChecker* [15] and *HyperDbg* [16] debuggers.

3.3 Object inspection tools

One of the sources of information about the *I/O* subsystem are the memory structures it maintains, primarily the *device objects* and the *driver objects*. There are specialized tools for viewing these *NT* objects.

In contrast with debuggers, the following tools are fully graphical and present information in a nicer way. On the other hand, they only show global data-structures that do not change. They do not capture the operation of the kernel and the temporary data-structures (e.g. IRPs).

3.3.1 WinObj

Executive objects in the object manager namespace can be viewed using *WinObj* [18] from *SysInternals* [7]. It displays the namespace in form of a tree, in a similar way *Windows Explorer* displays files and directories.

For each *executive object* it can display reference counts, handle counts and access rights. For some object types, like *events*, *symbolic links* and *mutants* it can display additional information.

However *WinObj* is limited to basically listing the *executive objects* and does not provide any details for *device objects* and *driver objects* and their relationships. Moreover, it can only display executive objects that have a name. It also does not show memory locations of the *executive objects*, which might be useful if this tool is used together with a debugger.

Example screenshot of the tool can be seen in figure 3.3 (this is the same screenshot as in Chapter 2, repeated for convenience).. Here *WinObj* displays the same directory that was previously listed by *WinDbg* in figure 3.2.

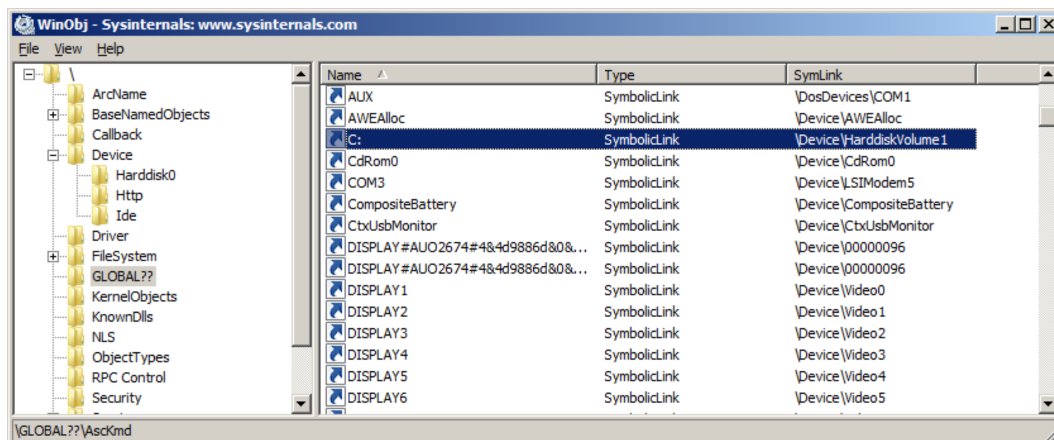


Figure 3.3: *WinObj* displaying the same part of *Object Manager* namespace as *WinDbg* in figure 3.2.

3.3.2 Device Manager

To display the devices installed on a system a *Microsoft Management Console* snap-in called *Device Manger* can be used. This tool is pre-installed on all *Windows* systems and even if it looks simple on the surface, a careful user can learn a lot of information from it.

By default *Device Manager* simply lists the devices by categories and does not display lot of devices. This can be changed by selecting the "Devices by connection" option and checking the "Show hidden devices" option in the *View* menu. This displays the actual *PnP* device tree of the system. Example of this tree is in figure 3.4, with detailed explanation in section 2.8.2 of the Chapter 2.

Combined with the "Details" tab in the properties of each device it provides nearly complete picture of the *PnP* manager's structures. However, when it comes to *device nodes* and *driver stacks*, the information is neither complete nor well formatted, although *Windows 8.1* has made improvements in this regard.

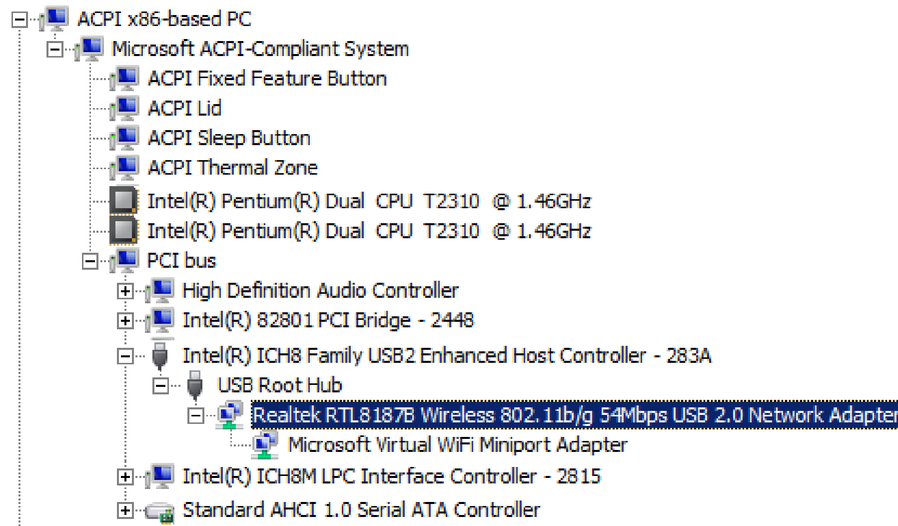


Figure 3.4: *Device Manager* MMC snap-in showing a portion of a device tree on a laptop.

A slightly more powerful commandline version of the *Device Manager*, called `devcon`, is included with the developer tools for *Windows* (e.g. *Visual Studio*, *WDK*).

3.3.3 DeviceTree

Another tool similar to *Device Manager* is *DeviceTree* from *OSR Online* [14]. While *Device Manager* is designed for system administrators, *DeviceTree* is intended for driver developers and includes more low-level detail.

The utility displays *driver objects* and *device objects*. The information about these objects is presented in a way that closely resembles the actual implementation of these objects, including memory addresses. The tool is also fully aware that devices and drivers are also *executive objects* and displays their names and reference counts.

It can show the *devices* and *drivers* from two perspectives. One of them is the *device tree*, the same one that can be displayed by *Device Manager*, but this time it includes the individual *device objects* in the driver stack. The second perspective is the driver perspective and lists the devices by the drivers they are managed by.

Example of the the *device tree* perspective is in figure 3.5. It shows the same device tree as *Device manager* in figure 3.4. Some of the devices from figure 3.4 are not shown here, because *DeviceTree* and *Device Manager* show the devices in different order.

You can also notice that in *DeviceTree* there are more intermediate nodes between the root of the tree and the selected devices. This is caused by the fact that *DeviceTree* displays the entire *driver stacks* as part of the *device tree*, not just the individual device nodes (the acronym `PDO` is used for *Physical Device Objects* and `FDO` for both *Functional Device Objects* and *filter drivers*).

The example nicely illustrates how the *virtual WiFi* discussed in 2.8.2 is implemented, because we can see the filter driver `\Driver\vwifbus`. This shows

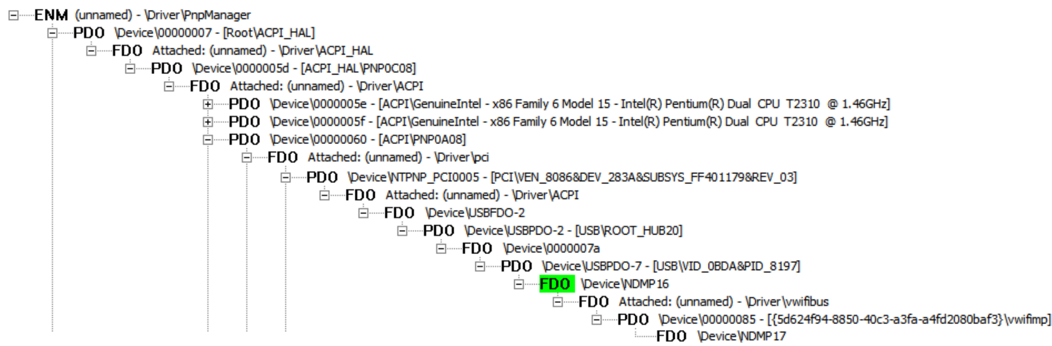


Figure 3.5: *DevTree* displaying the same device tree as *Device Manager* in figure 3.4.

that the more complex tree displayed by *DeviceTree* can be useful.

DeviceTree does a fairly good job and fulfills our goals when it comes to displaying kernel objects. However, improvements could be made to the presentation of the tree, which is cluttered and does not display human-readable device names.

3.4 Monitoring tools

A lot can be explained about the *I/O* subsystem by looking at the objects it maintains. Another method of learning about it is to actually watch it at work. This is the purpose of monitoring tools that record certain aspect of the *I/O* subsystems activity and show the events that have happened.

Unlike debuggers, they do not pause the whole system when the interesting event happens, but only record it and continue. Hence, they are generally able to run on the system they are monitoring.

They can be divided into two categories, depending on the set of events they monitor. One group of tools is similar to debuggers, because they are configurable and can monitor practically any code execution. The second group is designed to monitor only some well-defined set of events (in this case *I/O* activity).

3.4.1 Process Monitor

Process Monitor [19] by *SysInternals* [7] can monitor *I/O*, registry, network and process creation/termination operations on the whole system. For each of these events, it records the parameters passed to the system (such as file path), the thread and process of the caller, timestamp, duration of the request, and the results of the request.

While *Process Monitor* can show which files are read and written, it stops at the level of *Kernel API*. It does not monitor how the *I/O* request is processed internally and does not show the *I/O* handling activity mentioned in Chapter 2.

On the other hand we have to note that it has no installation requirements and is very easy to use. The only thing the user has to do is click the *Capture* button and browse the events.

There are many tools similar to *Process Monitor* in scope and functionality, such as *SystemSpy* [20] or *Strace for NT* [21]. They will not be discussed, because they provide more or less the same level of detail.

3.4.2 Event Viewer

Windows includes a centralized infrastructure for collecting events reported by the system and applications, called *Windows Event Log*. The collected events can be viewed using *Microsoft Management Console* snap-in named *Event Viewer*.

We are particularly interested in one group of logs, named *Applications and Services Logs → Microsoft → Windows → Kernel-** . These event logs record detailed system activity, including *I/O*.

An example, event recorded by the *Kernel-Disk* log is in the figure 3.6. *Event Viewer* can view the events in either *Friendly View* or *XML View*, however the *Friendly View* is only a simple reformating of the data. The figure 3.6 shows the *XML View*.

Interestingly enough, *Event Viewer* also provides pointers to the *IRP* and the *file object*. While the *IRP* pointer is not very useful, because by the time the event gets to the user the *IRP* will be deallocated, the user can use a debugger to view information about the associated *file object*.

```
<Event
  xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  <System>
    <Provider Name="Microsoft-Windows-Kernel-Disk"
              Guid="{c7bde69a-e1e0-4177-b6ef-283ad1525271}" />
    <EventID>11</EventID>
    <Version>0</Version>
    <Level>4</Level>
    <Task>0</Task>
    <Opcode>0</Opcode>
    <Keywords>0x8000000000000000</Keywords>
    <TimeCreated SystemTime="2015-04-08T14:02:45.247344800Z" />
    <EventRecordID>1</EventRecordID>
    <Correlation />
    <Execution ProcessID="4" ThreadID="64" ProcessorID="0"
               KernelTime="641" UserTime="0" />
    <Channel>Microsoft-Windows-Kernel-Disk/Analytic</Channel>
    <Computer>rntb</Computer>
    <Security />
  </System>
  <EventData>
    <Data Name="DiskNumber">0</Data>
    <Data Name="IrpFlags">0x60043</Data>
    <Data Name="TransferSize">8192</Data>
    <Data Name="Reserved">0</Data>
    <Data Name="ByteOffset">3146170368</Data>
    <Data Name="FileObject">0x86494008</Data>
    <Data Name="IORequestPacket">0x85888d48</Data>
    <Data Name="HighResResponseTime">547</Data>
  </EventData>
</Event>
```

Figure 3.6: Data provided by kernel event about disk IO.

3.4.3 Logman

The API underpinning the *Windows Event Log* is called *ETW* (*Event Tracing for Windows*). Some of the events *ETW* [13] can provide are not displayed in *Event Viewer*, but can instead by captured using a command-line utility called `logman`.

To list the kernel events available on your system, use this command: `logman query providers "Windows Kernel Trace"`. Apart from the disk *I/O* and file *I/O*, there are events that inform about *DPCs* and *ISRs*.

The data can be captured using the `logman` utility, stored in a file and viewed using `tracerpt` tool.

While both `logman` and *Event Viewer* are interesting, their level of detail is limited. For example, they can provide information about file, disk and network *I/O*, but not other device types . Furthermore they do not show how the *I/O* is processed by the *driver stack* layers. The log does not contain any links between the interrupts, *DPCs* and the *I/O* activity. Judging by the given usage examples on *MSDN*, the toolset is probably geared towards debugging performance problems using *xperf*.

3.4.4 IrpTracker

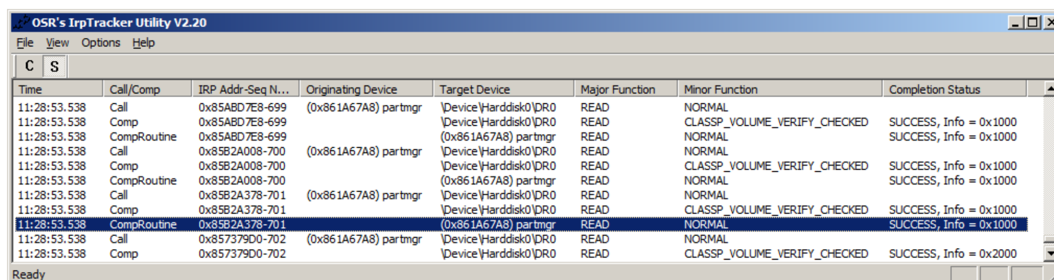
It is clear that a utility that provides monitoring of the actual execution of dispatch and completion routines is needed to track the life of the request. This is exactly the goal of *IrpTracker*, available from *OSR Online* [17].

It can track entry to a dispatch routine, the completion of an *IRP* and invocation of *completion routines*. It does not show the exit from dispatch routines and the user must chose beforehand which *device object* to track. There is no convenient option to track the whole *driver stack* for a given *device node*.

However, for each *IRP* completion event and dispatch routine entry, a snapshot of the whole *IRP* is available. Part of the *IRP* snapshots are the *I/O locations* of previous drivers, which gives at least some insight to the working of the *driver stack*.

An example *IrpTracker* session, demonstrating its depth of infromation, is shown in figure 3.7. Here, we have decided to track the `\Harddisk0\DR0` device. The detailed *IRP* view can be activated by clicking on the individual events. If the event is right-clicked, all other events concerning the same *IRP* are highlighted.

The disadvantage of *IrpTracker* is that it does not work on 64-bit systems and there are no plans to rectify this.



Time	Call/Comp	IRP Addr-Seq N...	Originating Device	Target Device	Major Function	Minor Function	Completion Status
11:28:53.538	Call	0x85ABD7E8-699	(0x861A67A8) partmgr	\Device\Harddisk0\DR0	READ	NORMAL	
11:28:53.538	Comp	0x85ABD7E8-699		\Device\Harddisk0\DR0	READ	CLASSP_VOLUME_VERIFY_CHECKED	SUCCESS, Info = 0x1000
11:28:53.538	CompRoutine	0x85ABD7E8-699		(0x861A67A8) partmgr	READ	NORMAL	SUCCESS, Info = 0x1000
11:28:53.538	Call	0x85B2A008-700	(0x861A67A8) partmgr	\Device\Harddisk0\DR0	READ	NORMAL	
11:28:53.538	Comp	0x85B2A008-700		\Device\Harddisk0\DR0	READ	CLASSP_VOLUME_VERIFY_CHECKED	SUCCESS, Info = 0x1000
11:28:53.538	CompRoutine	0x85B2A008-700		(0x861A67A8) partmgr	READ	NORMAL	SUCCESS, Info = 0x1000
11:28:53.538	Call	0x85B2A378-701	(0x861A67A8) partmgr	\Device\Harddisk0\DR0	READ	NORMAL	
11:28:53.538	Comp	0x85B2A378-701		\Device\Harddisk0\DR0	READ	CLASSP_VOLUME_VERIFY_CHECKED	SUCCESS, Info = 0x1000
11:28:53.538	CompRoutine	0x85B2A378-701		(0x861A67A8) partmgr	READ	NORMAL	SUCCESS, Info = 0x1000
11:28:53.538	Call	0x857379D0-702	(0x861A67A8) partmgr	\Device\Harddisk0\DR0	READ	NORMAL	
11:28:53.538	Comp	0x857379D0-702		\Device\Harddisk0\DR0	READ	CLASSP_VOLUME_VERIFY_CHECKED	SUCCESS, Info = 0x2000

Figure 3.7: *IrpTracker* tracking the `\Harddisk0\DR0` device.

3.4.5 Function tracing tools

Another approach to monitoring system activity is to let the user choose what to monitor. This is usually done at a function call granularity. Like with debuggers, the disadvantage is that the user must know which functions to look for. On the other hand they leave the system undisturbed, whereas debuggers cause a significant slowdown each time a breakpoint is hit.

We have to emphasize that we need a tool that supports monitoring an application without access to its source code and is able to work in kernel's environment. Many of the available tools are intended for managed languages and user-space applications.

Tools capable of monitoring the kernel are available for other platforms. *DTrace* [1] and *SystemTap* [2] are two examples of such tools for *Solaris* and *Linux*. For *Windows*, there are two following research projects with similar goals.

NTrace

One tool for monitoring functions is *NTrace* [23] a research framework for function boundary tracing, meaning that it can report the enter, exit and abnormal exit from a function. Unfortunately, the project is still in research phase and it is not publicly available.

Fay

Another suitable choice when it comes to monitoring function execution is *Fay* [22]. The basic functionality is the same as for *NTrace*, but it is extended with the support for remote operation and aggregation of data over a cluster of computers. This is however not any significant advantage for our audience. It is not clear if it can be used as stand-alone command-line tool, but provides an easy to use C# API based on the *LINQ* technology.

However, it is not clear if the project is publicly or commercially available.

3.5 Tool comparison

Each presented category of tools (debuggers, object inspection and monitoring) has their own weaknesses and strengths. Let us summarize the capabilities of the tools presented so far and look at the type of functionality that is missing in each category.

The main problems of **debuggers** are their user unfriendly user interfaces and cryptic commands. The only suitable one is *WinDbg*, because the others are not designed with *NT* kernel in mind.

The utilities for **object inspection** cover most of the kernel's data structures related to *I/O*. Specially the *DeviceTree* program (section 3.3.3) covers almost all aspects of the driver and device organization, although its presentation might be improved in minor ways (icons, more perspectives, translation of pointers to symbolic names, etc.). The only missing feature in this category is the ability to inspect *opened files* (`FILE_OBJECT` structures), this can only be done using debugger

Monitoring tools can offer much more detailed explanation of *I/O* activity than inspecting global data-structures. The problem with tools like *Event Viewer* (section 3.4.2) or *Process Monitor* (section 3.4.1) is that they do not provide the sufficient level of detail for understanding the life-cycle of an *I/O* request. *IrpTrakcer* is able to provide some of the details, but not as much as would be available with a general *function tracer*, namely information about *Interrupt Service Routines(ISRs)*, *Deferred Procedure Calls(DPCs)* and *dispatch function* exits. *Function tracers*, on the other hand are very general tools and the user must know what functions to monitor. Moreover, no function tracer is currently publicly available for *Windows.*

The common problem of the monitoring tools is also the nature of the output, which is a simple unformatted stream of events. *IrpTracker* tries to help the user by highlighting related events, but this idea could be extended further.

There are certainly uses cases for a new tool. It could be either a more user-friendly graphical debugger specialized for the *NT* kernel, or a low-level monitoring tool with easily understandable output.

3.6 Proposal for our tool

Based on the knowledge of the existing tools, we want to specify a feature-set for our new tool. The first concern is the nature of the tool – if it is going to be able to stop and inspect with the system (like debugger), or record the system activity (monitoring tool) or only inspect the kernel's global data-structures.

The previous section (3.5) already concludes that tools for inspecting objects cover their ground pretty well. The next decision is if our new tool will be a full-blown debugger, or just a monitoring tool.

It might seem that writing a debugger (with improved graphical interface) might be a better option, because it is more powerful. However, it is doubtful if our users (students) require the ability to stop and inspect the system and the powerful features might be just confusing. Moreover, we would inherit the technical problems of full system debuggers.

Monitoring tools are typically simpler and we consider the user interfaces of programs such as *Process Monitor* or *IrpTracker* very easy to use. They also encourage "exploration": the user can run his favorite application and see all the *I/O* requests it is making and chose the interesting ones, whereas the debugger would pause the system each time it encounters a request. We would like to retain this this design, where the user just has to click "start" and go through the results.

IrpTracker already provides a lot of low-level information, so we will use it as a starting point of our design (description of *IrpTracker* is in section 3.4.4). We propose the following improvements:

- Include more *I/O* concepts, namely the execution of *deferred procedure calls* and *interrupts* in the event log.

- Support both *x86* and *x64* platforms.

- Instead of highlighting events handling the same request (like *IrpTracker* does), provide alternate view of the event stream, that groups events by the requests they handle.

- Show summary of the request using a graphical diagram.
- Integrate it with limited *WinObj* functionality, so that users can explore objects mentioned in the *IRPs*.

IrpTracker monitors only selected devices. Because there is no easily detectable connection between a *DPC* and a device, there is no way to filter *DPCs*. The same is true for interrupts. For this reason, the selective capture feature of *IrpTracker* will be dropped and the whole system will be monitored instead.

This is the basic outline. As our tool gets more concrete form, we might want to add more useful features, if they do not clash with the basic usage pattern.

To actually implement this tool, the two remaining goals (as defined in Chapter 1) must be solved. The first one is to find a way of monitoring the system that can provide all the information our tool should display – this will be the focus of the two following chapters. The second way is coming up with a way to show and organize this information – covered in Chapter 6.

4. Gathering information

The main task of our tool will be to monitor the kernel's *I/O* activity. The purpose of this chapter is to find a suitable way of doing so. The available *APIs* will be evaluated along with alternative unsupported methods.

The most important criteria for choosing a methods of collecting the information is the type of events they support and the information they reports about each event. The events that our tool should were defined in the previous chapter (namely section 3.6). They are:

- start and end of a *driver dispatch routines*, including important information about the *IRP* (operation, file name, etc.) and the return value from the dispatch routines

- completion of the request

- start and end of a completion routines, including its return value

- start and end of *deffered procedure calls* (*DPCs*)

- start and end of *interrupt service routines* (*ISRs*)

- scheduling of *DPCs*

Together with the ability to explore *executive objects* and the *PnP* tree (as covered by existing tools), this list covers the basic concepts of *Windows I/O* – *IRPs*, *DPCs*, *dispatch routines*, *interrupts* and the asynchronous *I/O* handling (for explanation of these concepts, see Chapter 2).

Of course, supporting any other types of events is an advantage. For example, the tool could monitor alternative *I/O* paths, *I/O* cancellation and similar more advanced topics.

Secondary criteria is the degree to which the method relies on internal operation of the system that might change in the future. In an ideal case, it should be using a stable, documented API.

The method should also be reasonably fast. This requirement is not about exact measurements, but it only points out that we are going to monitor fairly frequent events and the system should not grind to halt completely. However, the tool we are writing is not meant to be used on productions systems and even a slowdown in the order of magnitude is tolerable.

4.1 Methods

The following list is a summary of the *APIs* and methods that offer the opportunity to inspect the kernel's operation at certain points. Their documentation can be found on *MSDN* (with the exception of the *code instrumentation* approach). The methods presented here are used by the existing tools described in the Chapter 3.

Debugger can be used to set-up breakpoints in the functions we are interested in. *WinDbg* offers an API to access its debugging engine. However, this would bring in the requirement of two computers.

Additionally, even when using the relatively fast *Ethernet* connection, *WinDbg* can respond slowly even during regular debugging. It is not certain if it could handle monitoring frequently-called functions.

+ anything can be monitored
+ stable, documented API
– speed
– common debuggers require two systems

Event Tracing for Windows API can be used to record predefined events in the kernel such as disk access, file access, interrupts, *DPCs* or Network IO. This API is used by the *Event Viewer* and `logman` tools. It is also used by *Process Monitor* for logging network access.

However, the set of events is limited and new ones can not be added to an existing kernel code. As an example it is able to report the execution of a *DPC*, but it is not able to distinguish when it was *scheduled*, entered and left.

+ stable, documented API
+ designed to be very fast
– no detailed events about request handling
– only some informations about interrupts and *DPCs*

Callbacks are provided at certain points in the kernel to monitor and alter system activity. One function to register callbacks is `ObRegisterCallbacks` for monitoring thread and process operations. Similar API is provided for registry, called `CmRegisterCallback` and for file-system requests, in the form of *file system mini-filters*.

These APIs are used by *Process Monitor* to capture information and are fully documented. However, there are not any callbacks for other events (like *ISRs*, *DPCs* and *dispatch routines*).

+ stable, documented API
– only events related to file and network access
– no detailed events about request handling
– no events for *ISRs* and *DPCs*

Filter drivers are drivers inserted above or below the driver in the *driver stack*. All *I/O* requests targeted to the driver will hence go through the filter driver.

One of the ways to accomplish this is to convince *PnP* manager to insert the *filter driver*. This change will only be applied after the system is restarted or the device is reconnected. Additionally it can not be used for devices not managed by *PnP* manager, like file-systems or non-*PnP* capable devices.

Another way is to rewire the *device stack* ourselves, which is prone to race-conditions and bugs, because *device objects* are not meant to be attached to the bottom of an existing stack, only to the top.

Finally, it does not solve the problem of monitoring *DPCs* and *ISRs*.

+ stable, documented API
+ events for *completion routines* and *dispatch routines*
− may be difficult to install for all devices
− no events for *DPCs* and *ISRs*

Dispatch routines provided by the driver can be replaced with our own. Pointers to the driver's *dispatch routines* are stored in its `DRIVER_OBJECT`. These function pointers can be replaced by pointers to our functions, which will collect the events and call the original *dispatch routines*.

IrpTracker is probably using this technique, but reverse engineering would be needed to prove this.

Since *Deferred Procedure Call* routines are registered with the system in a similar way to *driver dispatch routines*, they could also be replaced. However, unlike the `DRIVER_OBJECT` structure, which can be found through the *Object Manager* namespace, the corresponding `KDPC` structures are scattered through memory and can not be easily found.

ISRs are stored in a global table called *IDT* and can be replaced there. However, on 64-bit systems, this table is monitored for unauthorized modifications, by a mechanism known as *PatchGuard*.

+ unsupported, but not likely to change
+ events for *completion routines* and *dispatch routines*
+ can handle interrupts, but with problems
− no events for *DPCs*

Code instrumentation Kernel's code can be modified (instrumented) to insert our own monitoring points into the existing code. This is exactly what tools like *Fay* [22] and *NTrace* [23] do. This means rewriting the machine code of individual functions, either in memory or on-disk, to invoke additional monitoring code.

This is naturally not supported by *Windows* and code changes in future version may make the instrumentation method fail. Moreover, 64-bit *Windows* include

a mechanism called *PatchGuard*, designed to catch precisely these unsupported kernel modifications.

Code instrumentation also offers versatility similar to a kernel debugger, because it is not limited to pre-defined set of events, but can be easily extended to monitor new functions.

+ the set of events is extendable

− undocumented and unsupported

− interferes with *PatchGuard*

4.2 Conclusion

Unfortunately, the supported *APIs* (with the exception of a debugger API) are limited. For example, none of the supported methods can monitor the scheduling, start and end of a *deferred procedure call*.

Using a debugger for monitoring the execution of the code would probably be a performance problem. *WinDbg* is barely able to keep up with regular debugging, and it would probably slow down the system to an unusable rate if breakpoints were placed on the code we need to monitor.

So if we really want our tool to show all the events, the only suitable method is *code instrumentation*. Its biggest disadvantage is that it is unsupported and *Microsoft* has even taken active measures to prevent unauthorized kernel modifications, in the form of *PatchGuard* protection. However, there are ways to combat *PatchGuard* (will be discussed in section 5.4.5) and we feel that the flexibility offered by the code instrumentation method outweights this disadvantage.

The ability of code instrumentation to monitor any function means that we can later extend our tool with new events, if we discover the need for it. It will also be extendable by the user, who will be able to define new kernel functions to monitor. This will give him/her the full power of a function tracer, like *NTrace*, and will make the tool more useful to driver developers.

Because our tool will use similar approach to other function tracing tools, we have decided to name it *WinTrace*. This name will be used to refer to the software in rest of the thesis.

5. Instrumentation

WinTrace, our tool for monitoring *I/O* requests, needs to record *I/O* related events inside the kernel. In section 4.1 of the previous chapter we have shown that no documented *API* suits our needs and we will need to instrument the kernel's code.

First, it needs to be clarified what does *code instrumentation* mean in context of our tool. One of the meanings of this term is the complete rewriting of the machine code, for example for the purpose of program analysis or profiling (this is done by libraries like name *PIN* [25] or *DynamoRIO* [26]).

WinTrace needs more limited form of instrumentation than defined above: Chapter 4 has specified the that should be monitored and all of them correspond to either an entry to kernel function or its exit. For example, one of the monitored events is the *dispatch routine* entry, which corresponds to the `IofCallDriver` function entry.

Thus, the only requirement for our instrumentation method is that it must be able to intercept function entry and exit and call a function that will record this event. This is also often called *hooking* and the called functions is called *hook handler*. These term will be used in the following parts of the text.

There are three main decisions that must be made regarding the instrumentation component of our tool. First, the right way of hooking the functions, suitable for kernel environment, must be chosen and implemented. We will look at available libraries and common hooking methods. Second, a set of kernel functions that will be hooked must be selected, in a way that will allow monitoring of all needed events. Last problem is choosing an appropriate way of storing those events.

5.1 Libraries

In this section, we will present two libraries for hooking functions. More libraries exist, but they are not prepared to work in the kernel environment that has different *APIs* and restrictions compared to the user-mode.

For this reason, we have selected only libraries that explicitly advertise kernel-mode support: *Detours* and *EasyHook*.

5.1.1 Detours

The best-known library for hooking on the *Windows* platform is called *Detours* [30], developed by *Microsoft Research*. Unfortunately, the free version supports only 32-bit processors and does not work in kernel mode.

However, a kernel mode port is provided in a third party version by *Vito Plantamura* [31], but it does not support 64-bit machines at all.

5.1.2 EasyHook

A self-proclaimed replacement for *Detours* is the *EasyHook* [32] library. It provides additional features, like managed *C#* API, algorithm for avoiding deadlocks and

most importantly support for kernel-mode hooking. It is available under *LGPL* license and supports both *x86* and *x64*.

However, it looks like the kernel-mode hooking support is a second-class citizen. We have tested this library, specifically we took the test distributed with the library and modified it slightly, just to test if the kernel can really run with a hooked function. Unfortunately, the tested function was not hooked properly and the kernel crashed.

This is a design problem: *EasyHook* allocates memory during the redirection of the execution from the hooked function to the hook handler. This is fine in user mode, but memory allocations in kernel-mode may fail if `IRQL >= DISPATCH_LEVEL`.

Furthermore, *EasyHook* is not able to intercept recursive calls, this is again a design decision. Since lot of the functions we want to monitor can call themselves recursively (such as dispatch functions), *EasyHook* would have to be heavily rewritten to fix the problems and make it suit our needs.

5.2 Methods

Because both *Detours* and *EasyHook* libraries are not suitable for our purpose, this section will cover what techniques can be used to hook methods without them.

The methods presented in this section are commonly used for writing rootkits and other malware. The *Phrack Magazine* (`http://phrack.org`) offers lot of articles on this topic, for example *NTIllusion: A portable Win32 userland rootkit* [27]. The methods are also used by software like *NTrace* [23] (*hotpatching*) or *Linux* tracing backend – *KProbes* [24] (*breakpoints, prologue overwriting*) and *Detours* and *EasyHook* themselves (prologue overwriting).

5.2.1 Breakpoints

The first technique relies on mechanism used by debuggers to pause the executing program at certain location. When a debugger sets a breakpoint, it does so by overwriting the program code with an `INT 3` instruction. The same mechanism can be used to hook a function entry and exit. To do that, `INT 3` instruction needs to be placed at the beginning of the function and every `RET` instruction must be replaced with `INT 3`.

The `INT 3` instruction works by causing a software interrupt, invoking interrupt vector #3. The interrupt handler for this vector would be replaced with a function recording the event. After recording the event, the function must also simulate the effects of the instruction overwritten by the breakpoint and resume the code execution.

The instrumentation itself is easy to implement, because `INT 3` instruction has one-byte opcode and thus can replace even the shortest instructions. Most of the complexity lies in replacing the interrupt vector and writing a correct handler. The handler must simulate the overwritten instruction. It also needs to cooperate with the OS debugger and pass any unrecognized breakpoints to it.

The one-byte opcode also makes the hooking and unhooking race condition free, even when other threads are running the code. If the opcode was longer, the processor might execute partially overwritten instructions.

5.2.2 Hardware breakpoints

There are four debug registers (`DR0-3`) available on *Intel* processors. Each of them can be programmed with a memory address that will generate an exception if accessed. This allows us to intercept function calls, without modifying the function code itself.

The appeal of this method is that it is not detected by integrity checking mechanisms like *PatchGuard*.

Unfortunately, the limited number of available breakpoint registers makes it impractical to monitor more functions, which we need to do. The cooperation with *OS*'s debugging support is also problematic, because it uses the same debug registers.

5.2.3 Import table hooking

Knowledge of the *PE* (Portable Executable) file format used by the *NT* kernel and its drivers can be used to replace operating system functions with our own.

PE contains a structure called *IAT* (Import Address Table), for dynamic linking of modules (which is how drivers are linked to the *NT* kernel). When the module is on disk, *IAT* contains pointers to names of functions to import. The loader resolves the imported functions and replaces the name pointers with the addresses of the functions with that name. The module then uses indirect jump instructions to call the imported function (such as `CALL dword ptr [IAT_entry]`).

To replace the hooked function, import table hooking hooking simply overwrites the *IAT* entry for the hooked function with the *hook handler*. If the *hook handler* accepts the same arguments and has the same calling convention, it will be successfully executed instead of the original function. The address in *IAT* entry is backed up and used to call the original function from the *handler*, to preserve the original behaviour.

This mechanism has the advantage of being selective, with module granularity – some modules may have their API calls instrumented, some not. However this method cannot be used, when the call does not go thought the dynamic linking mechanism. This is true for private (non-exported) functions, which we do not care about, but also for calls inside one module. As an example, `IoCompleteRequest` calls `KeSetEvent` (both exported functions), but we would not capture this call, because it occurs inside of `ntoskrnl.exe` without crossing the module boundary.

5.2.4 Prologue overwriting

Another hooking strategy is to directly overwrite the beginning of the function with unconditional jump instruction. The jump redirects execution to the *hook handler*, which acts as a wrapper around the original function, but gives us the chance to execute the monitoring code.

To preserve the behaviour of the original function, the hook handler first simulates the effects of the previously overwritten instruction(s) and then continues the execution in the unmodified rest of the function.

There must not be any jump to the overwritten bytes of the function, but typical code does not jump to function prologue. Additionally, the programmer

must be careful to pause all other threads when it is modifying the code, or a partially overwritten instructions could be executed.

Hotpatching

Hotpatching is an ability to install software updates without restarting the computer. To support it, *Windows* must be able to replace functions at runtime and uses technique very similar to prologue overwriting, but with additional support of the compiler and the linker.

The compiler and linker place certain restrictions on hotpatchable code, that make *hotpatching* considerable easier and race-condition free. More information about the way *Windows Update* applies *hotpatches* is available in an *OpenRCE* article [34].

Prologue overwriting can take advantage of the restrictions *Microsoft C* compiler and linker place on code. This will make it easier to implement and it will avoid the need to pause all other threads.

5.2.5 Analysis

In general, *IAT* hooking, breakpoints, hardware breakpoints and prologue overwriting are all suitable methods for hooking functions.

Hardware breakpoints and *IAT* hooking can not be used. *Hardware breakpoints* are ruled out, because there are only four of them and we need to hook more locations than that. *Import Address Table* hooking is not suitable, because it is not able to intercept all calls.

The two suitable methods are breakpoints and *prologue overwriting*. Because the breakpoint method requires implementing the debug exception handler and dissecting functions to find all `RET` instructions, it is harder to implement. *Prologue overwriting* is simpler, compared to breakpoints, especially when we take advantage of the *hotpatching* support.

There are two libraries that could make prologue overwriting easier, but *Detours* does not support the *x64* platform in kernel-mode and *EasyHook* crashes. Instead, we will implement a variant of the *prologue overwriting* ourselves, while taking advantage of the *hotpatching mechanism*.

5.3 Implementation

The function hooking method that *WinTrace* is going to use (*prologue overwriting*) was already described in section 5.2.4, but only in general terms.

There are lot of details that are not evident from the general description. Because hooking is a central part of *WinTrace*, his section will show the concrete code modifications that must be done. The following text should also be useful for anyone implementing this technique, because the materials describing the system behaviour (especially in 64-bit cases) is scattered over the internet.

WinTrace, our tool overwrites the beginning of selected kernel functions in order to place jump to our monitoring code there. The goal of this section is to show the exact code modifications that are performed

The problem of code modification can be divided into two parts. The first one is redirecting the execution, using jumps to our hook handler. The second one is calling the original function (which is now patched) from the *hook handler*, to preserve the original behaviour of the system.

The hooking method for 32-bit systems is slightly different from the method for 64-bit systems. Because 32-bit systems are easier to hook, they will be described first.

5.3.1 32-bit execution redirection

Our hooking method relies on the fact that the *NT* kernel was compiled with *hotpatching* support.

Functions in 32-bit executbales compiled with the `/HOTPATCH` switch are guaranteed to begin with no-op instruction `MOV edi, edi` (this has been described by *Raymond Chen* [33]). Thanks to the `/FUNCTIONPADMIN` linker switch they will also be preceded with a 5 byte padding area. The layout of a hotpatchable function can be seen in 5.1, in the left part of the diagram.

The redirection consists of two jumps. The first one is a short jump to the padding area and replaces the `MOV edi, edi` instruction. The second jump is in the padding area, which has enough space for a full 32 bit absolute jump. The destination of the second jump is the *hook handler*.

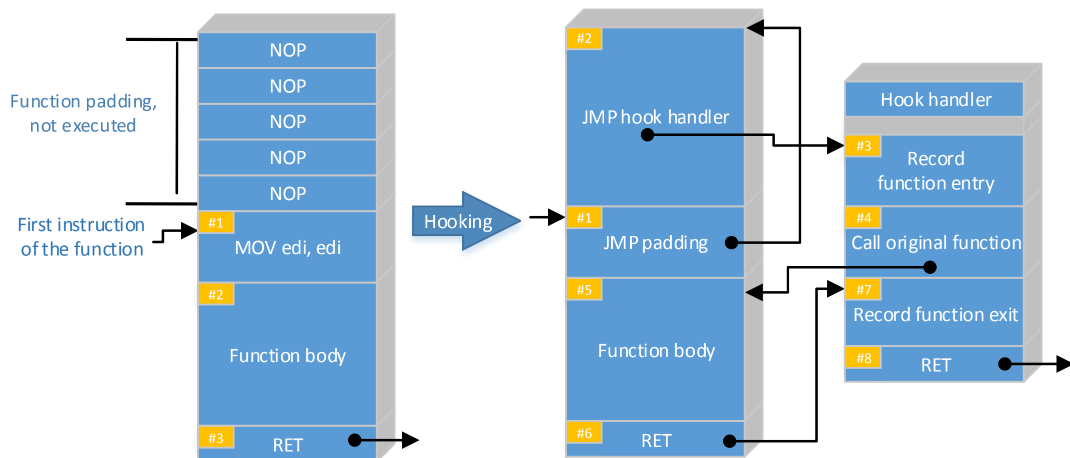


Figure 5.1: Execution schema for hooked 32-bit functions.

The *hook handler* is an ordinary *C* function, but it must of course have the same signature and use the same calling convention as the original function.

The two byte no-op provided by the compiler is very important, because the whole patching operation can be made atomic like this: first, the absolute jump instruction is written to the scratch space. This is safe, because no code is executing the padding. To activate the hook, the no-op instruction is atomically replaced. This can be done, because the compiler guarantees that it will be word-aligned.

5.3.2 32-bit call to original

The *hook handler* must call the original function to retain is functionality. This is simple with hotpatchable code, since we can be sure that the `MOV edi, edi` instruction is not an essential part of the patched function. When we need to call the original function, it can be done by calling `original_function + 2bytes`.

The complete execution order of individual instructions is shown in figure 5.1, both for original and hooked functions.

5.3.3 64-bit execution redirection

Compared to hooking 32-bit executables, there are two complications for 64-bit ones.

First, the compiler no longer provides the no-op instruction at the beginning of every function. Instead, the compiler guarantees that the first instruction is at least two bytes long. This behavior is not publicly documented, but it has been noticed that compiler generated functions often contain redundant `REX` instruction prefix to make the first instruction longer. It also seems to be confirmed by *MASM* code sample [35] on *MSDN*.

Second, the padding area is not long enough for a full 64-bit jumps. The guaranteed function padding is 6 bytes on 64-bit platforms, which is enough to hold a 32-bit relative jump, but not a full 64-bit absolute jump.

The not-yet hooked 64-bit function can be seen in the left part of figure 5.2. The right part shows the hooked functions and will be described in the following text.

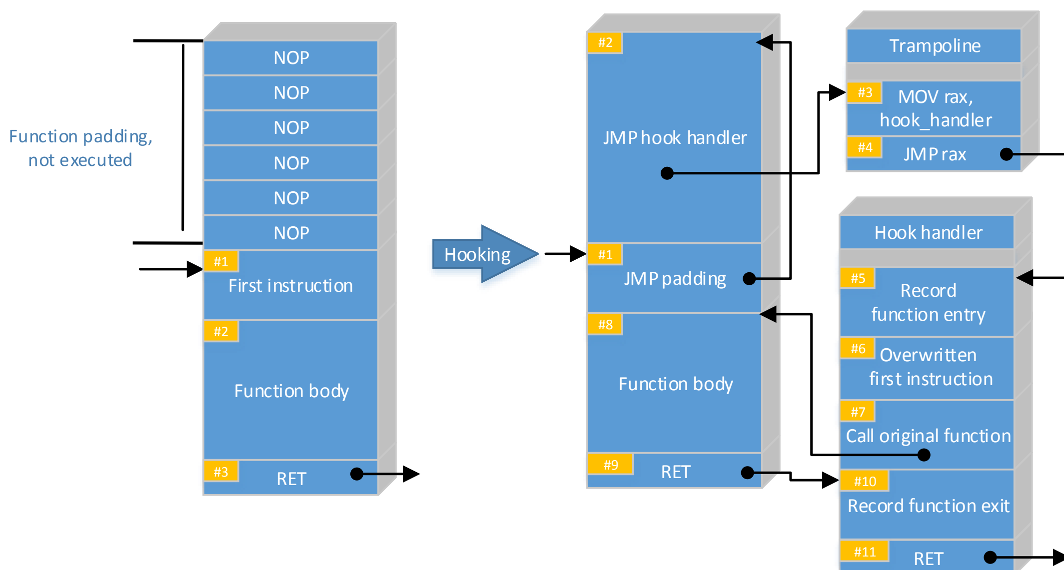


Figure 5.2: Execution schema for hooked 64-bit functions.

The solution to the problem of small padding area is to introduce yet another jump, in a memory allocated in the range of the 32-bit jump in the padding. Those three chained jumps will redirect the execution to our *hook handler*.

Because direct jump with 64-bit immediate value is not supported on *x64*. We instead use this replacement:

```
MOV rax, target
JMP rax
```

Allocating space for this 12-byte code sequence is problematic, because the kernel does no export any suitable function for mapping memory at arbitrary locations in kernel memory range (in user mode, this could be done using `VirtualAlloc` function). Instead of allocating memory, the surrounding memory can be scanned for large enough function padding that can be overwritten. We are currently not aware of any better solution.

5.3.4 64-bit call to original

Unlike in the 32-bit hotpatchable functions, where the first instruction was a replaceable no-op, the first instruction is necessary for successful execution of the patched function on *64-bit* platforms. Before overwriting the function, we must copy the first instruction aside and concatenate it with a jump to the second instruction of the original function. Calling this constructed peace of code has the same effect as calling the original function.

This is a slight complication, because in order to back-up the first instruction, we need to know its length. This involves at least partially understanding the instruction encoding of *x64* processor. To do this, our implementation uses a library called *Hacker Disassemblers Engine 64* [36], because it is a lightweight library with no dependencies on user-space functions.

5.4 Pitfalls

Should a reader decide to implement some of the presented hooking methods in kernel-mode, there are few problems he/she might encounter. They are not particularly hard to work-around (with the exception of *PatchGuard*), but it is better to know about them beforehand, because some of them are hard to identify and debug.

5.4.1 Memory protection

Kernel's code pages are marked as read-only and modifying them will cause a blue screen of death. One possible way to turn of this protection is to disable the *Write Protection* bit in the processor's control register, to circumvent this.

This problem might not manifest on a testing machine, because the protection mechanism is not activated on certain configurations, namely if the kernel uses 4 MB pages.

5.4.2 Non-standard functions

Some of the functions exported by the kernel are not full-fledged functions, but only short dynamic dispatch stubs like this:

```
nt.IoCompleteRequest:
  JMP     qword ptr [nt.pIofCompleteRequest]
```

The purpose of this stub is to transfer execution to address stored in variable `nt.pIofCompleteRequest`.

Most hooking methods will choke on this instruction sequence, because it does not start with a `MOV edi,edi` (our hooking on 32-bit platforms), it contains a jump (our 64-bit hooker would not recognize this function prologue) and is too short (which would be a problem for *EasyHook*).

The key is recognizing these special functions and applying a work-around. For example we find the final target of this jump instruction (stored in the `nt.pIofCompleteRequest` variable) and hook that function instead. We rely on the fact that the value of the variable rarely changes when the system is running. Another possibility is to avoid hooking these functions altogether or make the hook handlers do the dynamic dispatch themselves.

5.4.3 Compiler optimizations

In section 5.3 it was mentioned that the *hook handler* must have the same signature and calling convention as the function we are replacing. One of the things that the calling convention defines is the set of *preserved registers*, i.e. registers that the callee may not change.

However, since the whole kernel is one module, the compiler (or its link-time optimization phase) may apply additional optimization to calls inside one module. For example, it may notice that the callee preserves more registers than is defined by the calling convention and rely on this when allocating registers inside the caller.

Our hook handler must then play by the same rules as the original function and also preserve those registers. In *WinTrace*, we wrap the *hook handler* in a short assembly function that takes care of saving all registers.

5.4.4 Structured Exception Handling

64-bit versions of *Windows* have changed the exception handling mechanism. Functions no longer register their exception handlers in a linked list when they execute. Instead, all exception handling data is described in special *PE* image section and the operating system parses this information when it needs to handle exception. Because this mechanism relies on the exception handling tables, it is known as *table-based* model.

A programmer writing in assembly-language must generate these exception handling tables himself for every function that may cause an exception or call a function that may cause exception. Assemblers like *MASM* [35] or *YASM* [37] both support a convenient pseudo-operations for generating exception handling tables.

In our case, the assembly wrapper function responsible for saving registers must have this exception handling data.

5.4.5 Kernel Patch Protection

Since *Windows Server 2003 Service Pack 1*, critical kernel data (kernel code, service dispatch table, interrupt dispatch table etc.) on 64bit editions of *Windows*

is protected by mechanism informally known as *PatchGuard*. This is a piece of code that periodically checksums protected memory areas and crashes the system if the checksums do not match with unmodified kernel data.

The dynamic instrumentation approach is in clash with *PatchGuard*. Fortunately, attaching a debugger to the system disables the protection, because the debugger needs to overwrite kernel code too. Forcing our user to attach a debugger to the system is an unfortunate solution, considering that ease of use and minimal configuration are one of our goals.

PatchGuard, like all software protections, can be subverted. *PatchGuard* is a moving target and changes regularly with new version and service packs. An example for subverting a recent *Windows 8.1 PatchGuard* is given by *Positive Technologies* [28]. Unlike attaching debugger, this does not need user cooperation (and is used by some malware). We distribute a similar utility for disabling *PatchGuard* with *WinTrace*.

The kernel modifications can also be hidden from *PatchGuard*. One of the approaches is to utilize the separate instruction and data *TLBs* on *Intel* architecture processors. An example of this approach is the *SPIDER* (Stealthy Binary Program Instrumentation) [29]. The basic idea is to mark the page with stealth modifications as invalid and handle the page fault exception. For data access, the page is pointed to the original contents, for execute access the page is pointed to the modified code. The faulting instruction is then single-stepped over, which poisons the appropriate *TLB*. After the instruction is executed, the page table entry is marked as invalid again. The result is that processor will execute modified code, but *PatchGuard* will not detect any checksum changes, because it gets the original data.

Implementing some for of this advanced technique would be ideal, but is out of scope of this thesis.

5.5 Monitoring events

The goal for the monitoring part of our utility is to be able to provide information about at least the events mentioned in Chapter 4. This monitoring is based on *hooking*, which replaces selected kernel functions with *hook handlers* that record the events and call the original function.

The goal of this section is to find out what functions we need to hook to cover all the events we want, as defined in Chapter 4.

5.5.1 Hooked functions

Often, it is evident which function should be hooked for monitoring which event. For example the scheduling of a *Deffered Procedure Call* directly corresponds to `KeInsertQueueDpc` function. In other cases, this may be more complicated.

Where possible we want to avoid hooking *internal*, non-exported, operating system functions, because their addresses may change between different versions and even service-packs of the *OS*. Finding their address would then involve either version-specific code (generally bad idea), signature detection (fragile) or downloading and parsing debug symbols. These problems are avoided altogether by using only functions exported by the kernel.

The following list describes the functions we are going to hook and why:

- `KeInsertQueueDpc` is used by drivers to schedule invocation of *Deferred Procedure Calls*. Our tool uses it to intercept the execution of *DPCs* by replacing the *dpc routine* supplied to `KeInsertQueueDpc` by its own monitoring call.
- `IoCallDriver` is short function for calling the appropriate *dispatch routine* of a driver. Thus all *dispatch routine* invocations go throught this function..
- `IoCompleteRequest` is used for completing *IRPs*. It also runs all the *completion routines* registered inside the `IRP` structure. Our tool replaces the addresses of those *completion routines* with its own monitoring code.
- `HalBeginSystemInterrupt` is called before the actual *Interrupt Service Routine* registered by a driver is run.
- `HalEndInterrupt` is called just after the execution of driver-registered *ISR*.

Hooking these functions and replacing *DPC* and completion routine pointers enables us to monitor all of the events in Section 4.

5.5.2 32-bit interrupts

There is a problem with monitoring interrupt using `HalBeginSystemInterrupt` and `HalEndInterrupt`, because `HalEndInterrupt` is sometimes not called by the operating system. We need to understand why this happens, in order to eliminate the problem.

The inconsistency is caused by a mechanism called *lazy IRQL*, used by *Windows* to implement its concept of *Interrupt Request Level* efficiently. When *lazy IRQL* is used, raising *IRQL* does not immediately disable interrupts with lower priority. The disabling of interrupts would require communication with interrupt-controller, but *I/O* is slow. Instead when *IRQL* is raised, the new *IRQL* is only written to a per-processor variable.

When an interrupt comes, `HalBeginInterrupt` first checks if the interrupt should have been masked. If this is the case `HalBeginInterrupt` returns `FALSE` to signal that handling of this interrupt should be aborted.

The system remembers that an interrupt was aborted and re-executes it when the *IRQL* drops and the aborted interrupt is no longer masked. More detailed description of interrupt handling on *NT*, based on `hal` disassembly, is given in the `haldisasm.md` file (part of the *WinTrace* source). *Lazy IRQL* is also described in the *Windows Internals* book [4], int the section 3.1.1 (Interrupt Dispatching).

So the lack of `HalEndInterrupt` call is explained by the fact that its counterpart, `HalBeginInterrupt`, sometimes returns `FALSE`, thus aborting the interrupt handling. The inconsistency can be eliminated by ignoring `HalBeginInterrupt` altogether when it returns `FALSE`.

The missing `HalEndInterrupt` call would be a problem for a user going through the events, because it gives him the wrong idea about system activity. It would be even more serious if the events would be processed automatically by a program, because the missing call would have similar effect to unclosed bracket in a programming language. Since one of the proposed features of *WinTrace* is automatic grouping of events, we needed to fix this.

5.5.3 64-bit interrupt hooking

`HalBeginInterrupt` and `HalEndInterrupt` functions unfortunately do not exist on 64-bit platforms, because the *HAL* (*Hardware Abstraction Layer*) interface is different.

The only comparable equivalent is the kernel's interrupt handler, a function called `KiInterruptDispatch` and its sister functions like `KiChainedDispatch`. These methods are registered as interrupt handlers in the processor's *IDT* (*Interrupt Descriptor Table*). However, they do not support the *Windows* hotpatching mechanism (because they are written in assembler) and they are not exported.

Instead of hooking these internal functions, our tool actually replaces entries in the *IDT*. This mechanism is not used on *32-bit systems*, because the hooking of `Hal*Interrupt` methods is sufficient there. Moreover, replacing the interrupt handler in *IDT* requires very careful, architecture-specific implementation.

The new interrupt handler must duplicate some of the work done by regular *Windows* interrupt handler, before it can safely call kernel functions. First of all, it needs to save all registers. Then it needs to prepare the environment kernel code expects, concretely load pointer to *processor control block* into the `gs` register (this is done by `SWAPGS` instruction). The *processor control block* stores information about the task running on current processor and is needed by functions such as `PsGetCurrentThreadId` and `KeGetCurrentProcessorNumber`.

Since the goal is to also monitor the end of an interrupt, the regular *Windows* interrupt handler we are going to call must be convinced to return execution to us. This is done by constructing a replica of processor's *interrupt frame* on the stack. Processor's interrupt frame contains information about code executing before the interrupt has happened. The `IRET` instruction is used to return from interrupt handlers and restores the state of processor from the *interrupt frame*. Our replica of it, however, instructs the processor to execute rest of our monitoring code.

The downside of this approach (compared to `Hal*Interrupt`) is that it monitors interrupts as seen by the processor. Because of *lazy IRQL*, the interrupts are actually handled a bit differently. This may cause slight inconsistencies in the reported events, but we are not aware of any easy fix.

5.6 Collecting events

The chosen instrumentation method (*prologue overwritting*) allows us to run arbitrary code when functions are executed. This code will gather information about the current system state (like thread id), function call arguments and function return values. The problem is where and how to store this data.

The collection method must be able to sustain the amount of events generated by *WinTrace*. *WinTrace* generates events for interrupts, *DPCs* and *I/O* requests. The number of interrupts and *DPCs* can be measured using *Performance Monitor* (part of *Windows*). The *I/O* activity can be estimated using *Process Monitor* [19]. Each *DPC* generates three events (start, end, schedule), interrupt two (start, end) and each *I/O* request about 40 (dispatch start, dispatch end, completion routine start, completion routine end, multiplied by the maximum expected depth of driver stack – 10). The concrete numbers will vary by system and load, but it is

safe to say it will be several thousand events per seconds[1].

The amount of events gathered also makes it impossible to store them only in memory, they need to be moved to disk eventually. Storing them in memory would limit the length of monitoring sessions to minutes at most (depending on the memory capacity).

The collection method must also be able to work in the restricted environment where it may be invoked (like interrupt handlers). The method should also take into account that it is running as a part of an arbitrary function – there is a potential of causing deadlock or infinite recursion by calling other hooked functions.

5.6.1 Writing to a file

A naive solution to the problems is directly using an *API* such as `ZwWriteFile`, to store the events to a file on disk.

However, this is not possible. First of all, `ZwWriteFile` can not be called during interrupts and *DPCs* (it requires `IRQL == PASSIVE_LEVEL`). Even if we could use `ZwWriteFile`, its per-call overhead is too big.

5.6.2 Event Tracing for Windows

There is an *API* that could be used for our purpose. *ETW* was already mentioned in Chapter 3 as the provider of events about disk *I/O*, *DPCs*, etc. However, other applications including drivers can register as event providers and consumers. This allows *ETW* to be used as a backend for transferring events from kernel to either a file or user-space application.

We have tried using *ETW*, but we were having problems with deadlocks in interrupt routines. According to *MSDN*, `EtwWrite` should be callable at any *IRQL*, but our machine was locking up in `EtwpReserveTraceBuffer`, where it was trying to acquire an already taken spinlock. For this reason, we have moved to a custom solution.

5.6.3 Lockless circular buffer

Because the solutions presented so far are inadequate, our implementation uses a global shared circular buffer with one reader and multiple concurrent writers.

The writers reserve space in the buffer by moving the buffer head using atomic compare-and-swap. They write the event data into the reserved space and mark the segment as written. If the writer notices the buffer is getting full, it signals a buffer-is-full notification event.

The buffer is serviced by a single reader (started by the control utility) that periodically flushes the data directly to file using `ZwWriteFile` API. It then moves the buffer tail forwards, freeing up space. Moving the tail pointer forward is not a synchronization problem, because writers relying on old value of tail pointer see less free space and will drop the event (which is not desirable, but does not lead

[1] This estimate is consistent with the amount of events we have measured once *WinTrace* was implemented.

to data corruption). Details about the trace file format and subsequent handling of the data can be found in Chapter 8.

ETW uses per-processor buffers, which should make it more scalable than our method. We feel that by having a single buffer, we get easier implementation, less copying of data and better ordering of events across different processors, which outweighs the possible performance issues.

5.6.4 Dropping events

What happens when the buffer becomes full? New events are dropped. Sadly, this is the only sensible decision for a monitoring utility watching the very same *I/O* system it uses. If the ring-buffer is full, flushing it would cause more *I/O* to be generated.

Depending on the usage, some users can tolerate dropped events, while the same trace may be useless for others. This is even more important for our tool, that does post-processing on the data and relies on the fact that some start/end events are balanced. A lost event about interrupt end may then cause a whole stride of subsequent events to be considered part of that interrupt.

Our goal for the logging subsystem is to have no dropped events on a regular system used for desktop activities (applications starting, file copying, etc.). This is mostly a question of choosing appropriate buffer sizes.

6. Visualization and parsing

One of the goals for our tool (*WinTrace*) is to present information in easily understandable way (we have defined this goal in Chapter 1). One of the ways of achieving better understandability is grouping the events according to the *I/O* operations they are part of and providing diagrams of those operations. that the captured events describe.

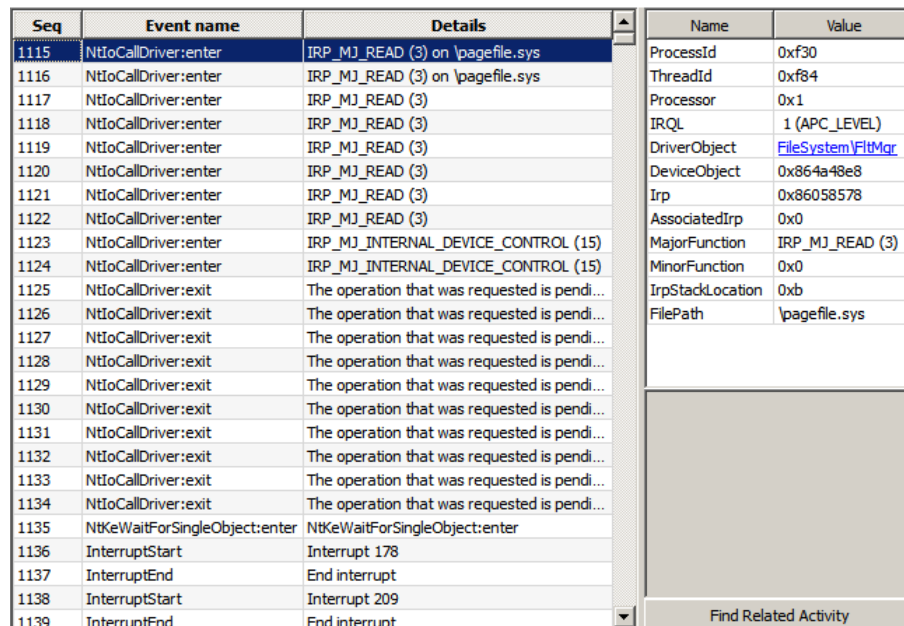
After that it first introduces the *Object view*, our way of visualizing the same data. Then it explains the algorithm for extracting information displayed by the diagrams *Object view*.

Apart from *Object view* and the chronological listing of events, *WinTrace* also supports two other types of views, one based on *sequence diagrams* and the other on the graphical visualization of the function call tree that both show some information that is not evident from *Object view*.

6.1 Event view

The monitoring component of *WinTrace* produces stream of events about system activities. The simplest method of displaying this information is to list the events in chronological order. This is similar to they way tools like *ProcessMonitor* [19] or *IrpTracker* [14] display their data.

An example of the *event view* is given in figure 6.1. The view is divided into two panes. The left pane lists the events in their chronological order, each event having its name and brief description. The right pane shows all the details the instrumentation code has recorded for the currently selected event.



Seq	Event name	Details
1115	NtIoCallDriver:enter	IRP_MJ_READ (3) on \pagefile.sys
1116	NtIoCallDriver:enter	IRP_MJ_READ (3) on \pagefile.sys
1117	NtIoCallDriver:enter	IRP_MJ_READ (3)
1118	NtIoCallDriver:enter	IRP_MJ_READ (3)
1119	NtIoCallDriver:enter	IRP_MJ_READ (3)
1120	NtIoCallDriver:enter	IRP_MJ_READ (3)
1121	NtIoCallDriver:enter	IRP_MJ_READ (3)
1122	NtIoCallDriver:enter	IRP_MJ_READ (3)
1123	NtIoCallDriver:enter	IRP_MJ_INTERNAL_DEVICE_CONTROL (15)
1124	NtIoCallDriver:enter	IRP_MJ_INTERNAL_DEVICE_CONTROL (15)
1125	NtIoCallDriver:exit	The operation that was requested is pendi...
1126	NtIoCallDriver:exit	The operation that was requested is pendi...
1127	NtIoCallDriver:exit	The operation that was requested is pendi...
1128	NtIoCallDriver:exit	The operation that was requested is pendi...
1129	NtIoCallDriver:exit	The operation that was requested is pendi...
1130	NtIoCallDriver:exit	The operation that was requested is pendi...
1131	NtIoCallDriver:exit	The operation that was requested is pendi...
1132	NtIoCallDriver:exit	The operation that was requested is pendi...
1133	NtIoCallDriver:exit	The operation that was requested is pendi...
1134	NtIoCallDriver:exit	The operation that was requested is pendi...
1135	NtKeWaitForSingleObject:enter	NtKeWaitForSingleObject:enter
1136	InterruptStart	Interrupt 178
1137	InterruptEnd	End interrupt
1138	InterruptStart	Interrupt 209
1139	InterruptEnd	End interrupt

Name	Value
ProcessId	0xf30
ThreadId	0xf84
Processor	0x1
IRQL	1 (APC_LEVEL)
DriverObject	FileSystem\FltMgr
DeviceObject	0x864a48e8
Irp	0x86058578
AssociatedIrp	0x0
MajorFunction	IRP_MJ_READ (3)
MinorFunction	0x0
IrpStackLocation	0xb
FilePath	\pagefile.sys

Find Related Activity

Figure 6.1: Raw stream of events, as displayed by *WinTrace*.

Showing this stream of events to the user as-is is far from ideal. One of the problems is the interleaving of events that belong to different requests (because of

threading and asynchronous completion). One improvement would be to group the events by the request they belong to. It would be event better to show subordinate requests together (such as disk I/O caused by filesystem read).

The other problem is the verbosity of the information. Even simple requests that take advantage of the driver stack model may take screen full of events.

6.2 Object view

The goal of the *object view* is to solve the problems connected with *event view*, while emphasizing the *I/O* handling concept of *driver stacks*.

A common way of illustrating the layered driver model is a list of *driver stack* diagrams. Example of such device stack is the diagram from *MSDN* [38] in figure 6.2. Each *device node* is represented by a block consisting of boxes representing the individual *device objects* (and in turn drivers). The device nodes are connected with arrows representing the parent-child *Plug and Play* (*PnP*) relationship.

When figure 6.2 mentions *device stack*, it corresponds to our term *driver stack*. Both terms are used interchangeably on *MSDN*.

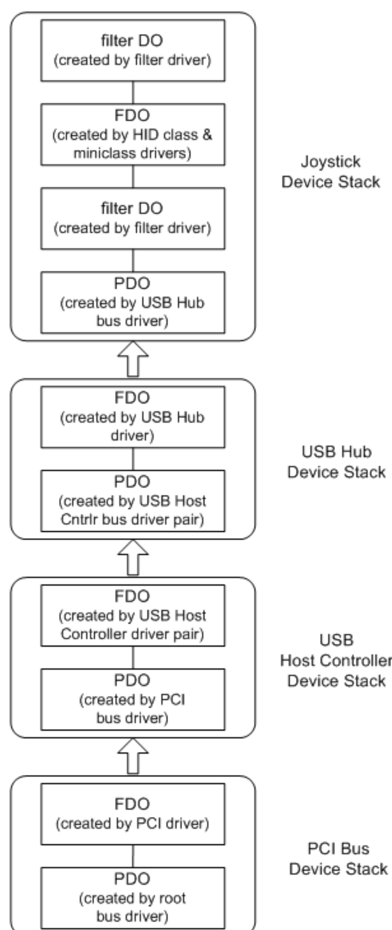


Figure 6.2: Illustration of the device stack for *USB* Joystick, taken from *MSDN* [38].

The *driver stack* diagrams display the data structures maintained by kernel

(the *Plug and Play* relationship and the *driver stacks*), whereas we need to display live requests. However, by replacing *device nodes* by *IRPs* and *device objects* with *I/O* locations, we can draw very similar diagrams. Two *IRPs* will be connected with arrows if one is the creator of the other.

Example of this diagram can be seen in figure 6.3. This time, we are reading data from *USB* harddisk (author does not have a joystick). The upper *IRP* is the original request created by user application and shows the drivers in the storage stack. The storage stack has created two subordinate *IRPs* to perform the disk reads and writes.

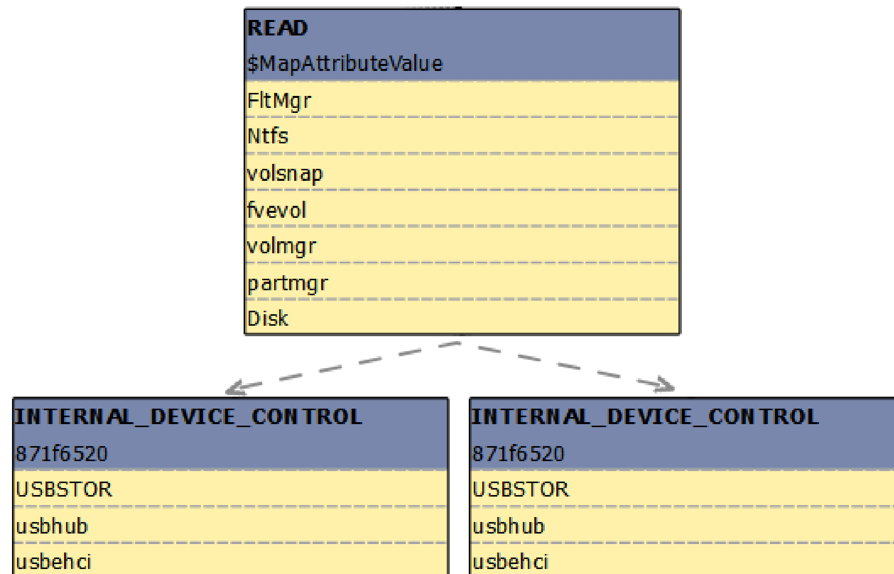


Figure 6.3: Illustration of the requests used to service a read from USB disk, as reconstructed from the recorded events.

6.2.1 Threads

The *driver stack* diagram gives us nice overview about the participating *drivers* and the *IRPs*, but does not show the low-level operations like invocation of the *dispatch functions* and *IO completion*, which are available in the *event view*. For example it is not possible to know, if the request was completed synchronously or asynchronously.

To connect *Object view* and low-level events, so called *pins* (dots) are placed near each row in the driver stack. These pins represent the operations on the *IO location*: pins to the left represent the start of the driver dispatch routine, pins to the right the end of the dispatch routine and pins in second column to the right the completion of the location by `IoCompleteRequest`.

To show the order in which the operations happened, the pins are connected by a line, in chronological order. This line starts with a marker at the top of the diagram and represents one logical thread of execution.

In figure 6.4, we can see an example of a request with pins and connecting lines. This time the requests do not have deep *driver stacks* or subordinate requests. Instead one of the requests completes the other one.

The application causing this behavior is *Google Chrome* communicating with its tabs throught named pipes. It first tries to read from the pipe, but the request is pended, because no data is ready yet. This first request is the one on the right. The thread first enters the dispatch routine, but then immediately exits it. If we were to click on the pin to the right of the *READ* request, we would see that the dispatch routine returned `STATUS_PENDING`.

After that comes another thread (shown on the left) and starts a *WRITE* request to the same pipe. The pins on this thread line are numbered for convenience. The next thing the thread does is completing the previous *READ* request, because new data has been written to the pipe. Finally, it completes its own request and returns from the dispatch routine.

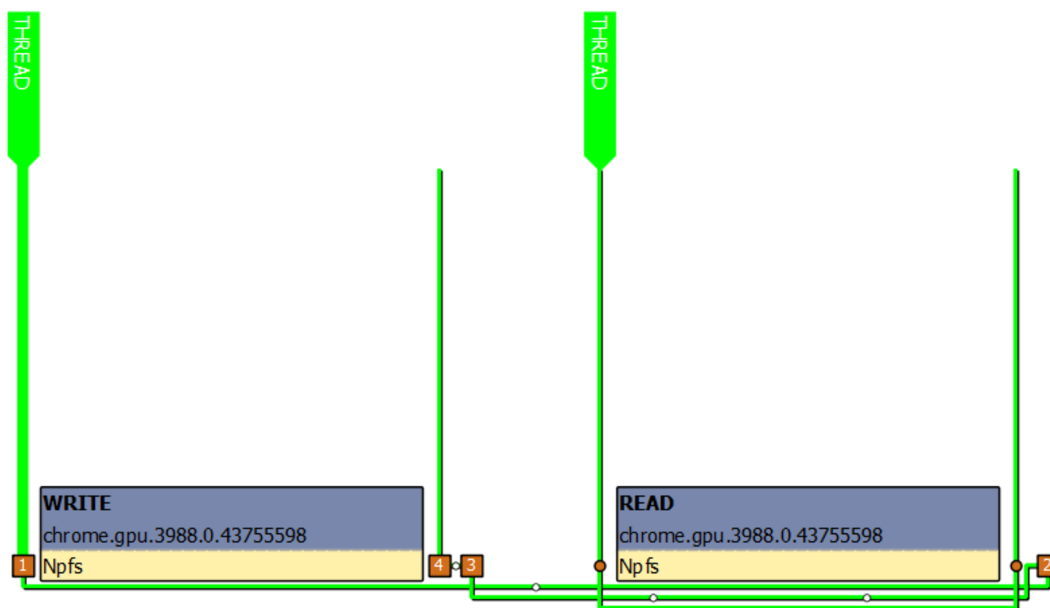


Figure 6.4: Chrome reading and writing to a named pipe.

Evaluation

These diagrams do not show the order of execution of events on different threads. In the figure 6.4, there is actually no way to see which of the threads came first. This is not a serious problem, because most of the time the user can infer order of the threads from their actions – in our example the thread on the left could not have completed the *READ* request, before it was started by the thread on the right. For the actual interleaving of the threads, the user can always consult the event view.

During testing we noticed that it is hard to follow the thread lines, especially when they are long and tangled. For this reason several visual cues are provided, like drop-shadows, mouse-hover effects and numbering of the pins on a selected line. The lines are also routed carfully and only right angle intersections are alowed. The details of the line routing algorithm will be described in section 6.4.

We are not aware of any formal name or prior description for this kind of visualization. Compared to simple chronological listing of events it is certainly an improvement. It is able to illustrate the driver stacks and the completion process for *IRPs*, including synchronous and asynchronous completion.

6.2.2 Interrupts and DPCs

As described, *object view* displays most of the information available, except *DPCs* (*Deferred Procedure Calls*) and *ISRs* (*Interrupt Service Routines*).

Showing their execution is important, because most asynchronous requests are completed by interrupts (indirectly via *DPCs*). The straightforward way is to connect events that happen during interrupt using a line, like we do for threads (perhaps using different color). Another line will connect the *DPC* events.

The problem is that the actual request completion does not happen in the interrupt. The interrupt only reads the volatile hardware state and schedules a *DPC* routine, which in turn completes the *IRP*. This means that no event on the *interrupt* line is pinned to the *IRP* objects and the line would be empty.

Our solution is to explicitely show the relationship between the *interrupt* and *DPC* by introducing a new type of block. So in addition to *device stacks*, we will have *DPC* objects in our diagram. The pin to the left of *DPC* object will represent the scheduling of the *DPC* and will usually be connected to the interrupt handling line. The pin to the right will represent the execution of the *DPC* routine and will always be the first pin on the line connecting the *DPC* events.

Example of a simple request that was completed asynchronously using interrupt is shown in figure 6.5. This time, it is a request fetching data from the operating systems's paging file. There are two *IRPs*, the higher level one (1) responsible for file system and partitions, the lower level one (2) responsible for the actual *ATA* disk. The requests are started by a regular thread (the green line), but it returns without completing either of the *IRPs*. Both requests are completed by a *DPC* (the red line). The link between the *DPC* and the interrupt that has scheduled it is the *Deferred Procedure Call* object (3), which is connected to both lines.

6.2.3 Events

A common occurence in *I/O* handling is the use of *kernel events* for synchronization. This is typically done by a driver wishing to synchronously wait for an asynchronous *IRP*. Thus, we have decided to extend *object view* with the ability to display this kind of synchronization.

Blocks will be used for displaying *kernel events* (as they are for *DPCs* and *IRPs*). This event block will have two pins, the one on the left will represent the *wait* operation, the one on the right will represent the signalling of the event.

It is traightforward to extend the monitoring component to report these events by hooking `KeInitializeEvent`, `KeSetEvent`, `KeWaitForSingleObject` and some of their variants.

An example of a request using an event is given in figure 6.6. The *Ntfs* driver passes the I/O down to lower-level drivers, but decides to wait synchronously for result of the *IRP* (1) using an event (2).

The event is signaled by a *DPC* when it tries to complete the request. The waiting *Ntfs* driver is woken up and completes the rest of the request synchronously. If we didn't include the event object in the diagram, the user might by puzzled. Without it, it is not obvious how does the *Ntfs* driver know that the *DPC* routine has completed the lower-level parts of the request.

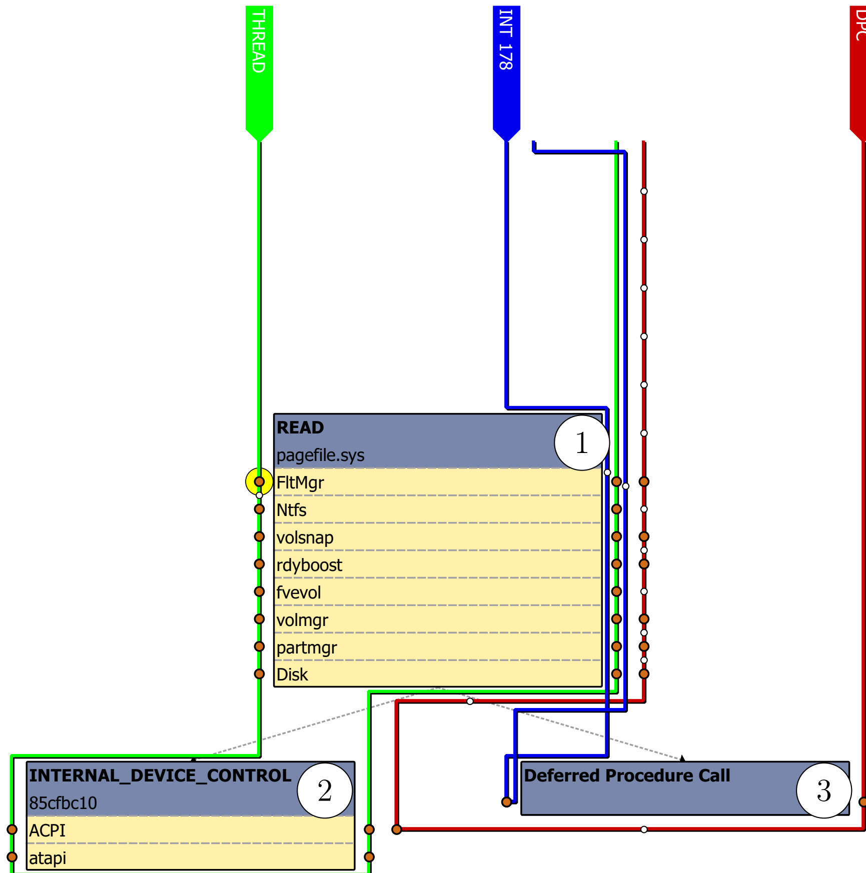


Figure 6.5: Page fault handling request, displayed in object view

6.3 Parsing

Object view displays various objects (*IRPs*, *DPCs*, *kernel events*). Thanks to the instrumentation, we can monitor all the events needed for displaying the information in *object view*. However, is not yet clear, how to:

- know which events should be on the same connecting line (in other words, group events into threads, *DPCs* and interrupts),
- find the objects (*IRPs*, *DPCs*, *kernel events*),
- organize the objects into a parent-child relationship,
- recognize which events constitute a group of related requests that should share one diagram.

6.3.1 Goals

The component responsible for solving this problem is called *parser* in *WinTrace* (we do not wish to imply any similarity with the usual context-free language parsers). The parser component reads the input events one by one, solves the problems mentioned above and reports the visual components that make up each

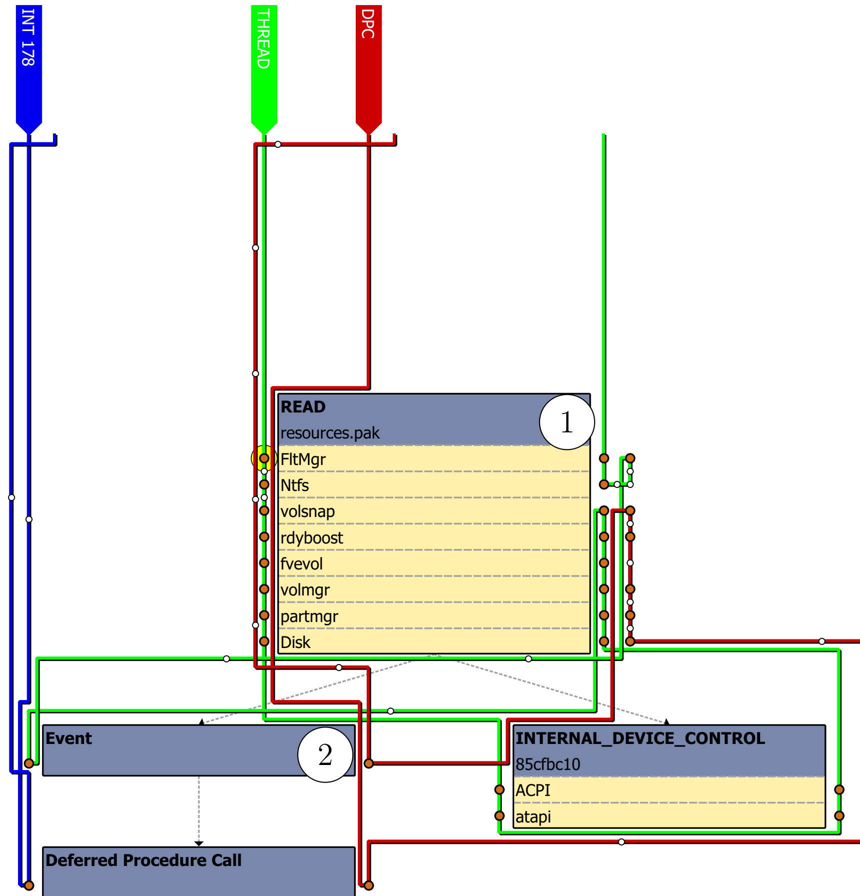


Figure 6.6: File read request, displayed in *object view*. *Ntfs* driver uses an event to synchronously wait for completion of the rest of the request.

request diagram (without information about their layout). The diagrams are then listed in the GUI and the user can choose which one to display.

In case events are lost, the sequence of events may appear to be nonsensical, but the parser must be prepared for that. It is inevitable that it will display garbage, but it should not crash under any circumstances. Ideally, it should be able to recover over time and start showing meaningful diagrams again.

The parser must not rely on keeping all events resident in memory, because the trace files can easily grow to hundreds of megabytes. It should instead read them incrementally and display the requests in near real-time as new events come in.

6.3.2 Data model

The purpose of this section is to more formally clarify what should be the output of a parser (the previously mentioned "diagram description"). This data model is meant to be a middle step between the raw events and a fully drawn diagram. It contains information about the visual elements displayed by the object view – *IRPs*, *IO locations*, *DPCs*, *kernel events*, pins and the lines connecting those pins. It also contains the labels and colors of these objects, but does not include any information about their positions and sizes.

The rendering and layout of the diagram is responsibility of the object view component. There are also performance reasons for this split, because laying out each and every request diagram is unnecessary, considering that the user will never view most of them.

ER diagram of the parser output is shown in figure 6.7. The most important objects in the diagram are *blocks*. This is a common representation for *IRPs*, *DPCs* and kernel events. The blocks have a name and can be hyperlinked with the details of the underlying kernel object, if its available in *Object Manager* namespace. If the block represents an *IRP*, it will be subdivided into block parts, each representing the *IO location*. Again each block part has a label and can be linked to a kernel object.

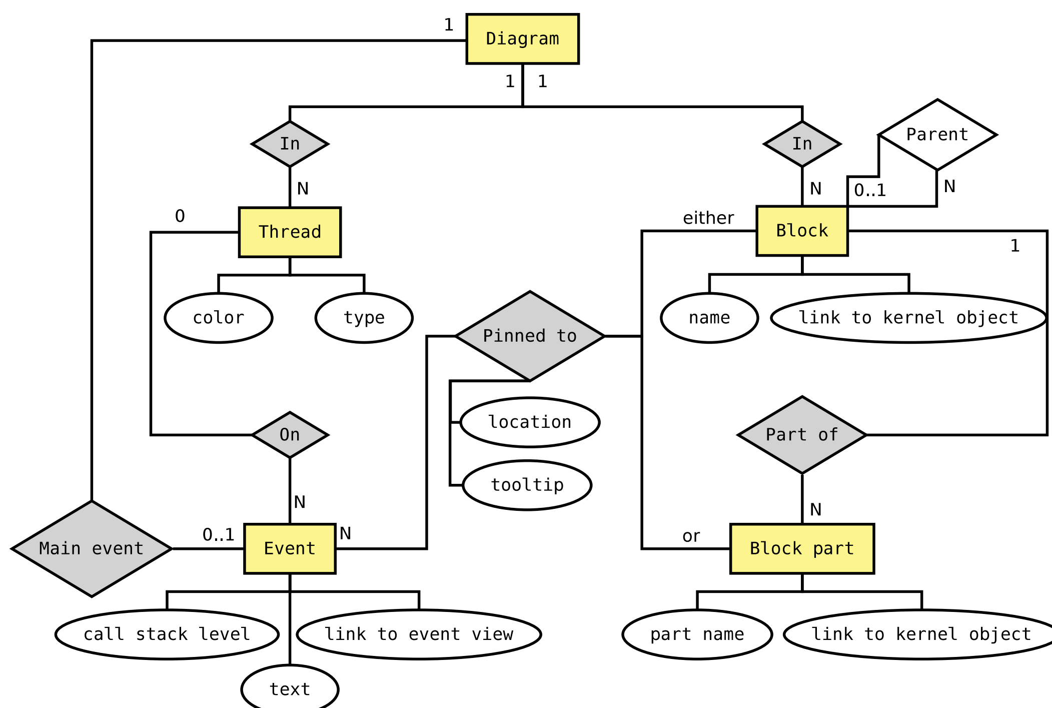


Figure 6.7: ER diagram of parser output.

The next responsibility of the parser is to separate events into their respective execution contexts: *threads*, *DPCs* and *interrupts*. It reports those execution contexts, the color that should be used for the connecting line and the type of the context. Each execution context is a list of events, in the order they have happened. In the following text, we will call the execution contexts *parser threads*, or just simply *threads* if it is obvious from the context that we do not mean regular OS threads.

The events themselves consist of a brief description, ID of the event (for opening the event in *event view*) and the number of enclosing function calls (we will show later, why it is useful). Events can also be pinned to the blocks (or their parts), on either side, with additional explanation text available as a tool-tip.

Each diagram has a one event designated as a "main event". This event is used to deduce the name of the whole diagram and it is typically then event that has started the first request.

6.3.3 Parser threads

The first task we must solve is identifying the separate execution contexts – *parser threads*. This is important, because the next improvements to the parser (like reconstruction of the I/O stack) will require the ability to maintain per-thread state.

Fortunately, each monitored event is tagged with the *OS*'s thread ID, where it happened. But as we have mentioned, OS threads do not map directly to our *parser threads*. One of the reasons is that *DPCs* and *interrupts* do not have separate thread IDs, because they execute in an arbitrary thread context.

Moreover, we do not want an OS thread to be represented by one *parser threads*, but instead by multiple smaller *parser threads*, each containing just one request. The *parser threads* can then be thought of as either *DPCs*, *interrupts* or *syscalls* (the analogy with *syscalls* is a bit loose – the request may have been issued by kernel mode component, in which case there is no *syscall*).

If we were to represent an OS thread by a single *parser thread*, we would probably end up with a long thread, spanning the whole length of the recorded events, issuing large numbers of unrelated I/O requests. Since a thread is part of one diagram, all its requests would have to be in the same giant diagram.

These are the reasons, why the parser maintains a stack of *parser threads* for each OS thread, modeling the interruption of *parser threads* by each other (*DPCs* interrupt regular threads, interrupts can interrupt any type of *parser thread*). The topmost *parser thread* on the stack is the currently executing one. The threads below them are the ones that were interrupted.

New threads are pushed onto the stack, when either an event marking the start of *DPC* or interrupt is spotted, or when a start of "interesting" function call is spotted. Currently, we consider only the function `IoCallDriver` as interesting – all `PASSIVE_LEVEL` I/O handling happens inside this function.

Symmetrically, threads are popped from the stack when an event marking the end of *DPC* or interrupt is spotted. Regular threads (not interrupts or *DPCs*) are popped from the stack when the interesting function exits.

However, to know that a function has exited, we must maintain a call stack for each *parser thread*, changing it each time function entry or exit event is encountered. We will also make use of the call stack we are going to maintain later on.

To see and example of the events being parsed, look at figure 6.8. For simplification, we assume all of the events happened on the same OS thread. The names used for the events are the actual event identifiers, as used by *WinTrace*.

The *parser threads* are marked with colored vertical lines. If there is a line of corresponding color to the left of the event, it means that the event is part of that thread (and all such events will be connected by a line in object view).

The first two events, describing entry and exit from `KeWaitForSingleObject` function are not marked with any color, because this function is not "interesting". The next event, start of `IoCallDriver` function, pushes a thread onto the thread stack, and so is part of this new thread. But the three following events are part of different thread, because the interrupt start event pushes a new thread on the stack. The thread is popped by interrupt end, but is immediately replaced by a *DPC* thread. The events after `KeInsertQueueDpc:dpcexit`, which pops the *DPC* thread, are again considered part of the original request.

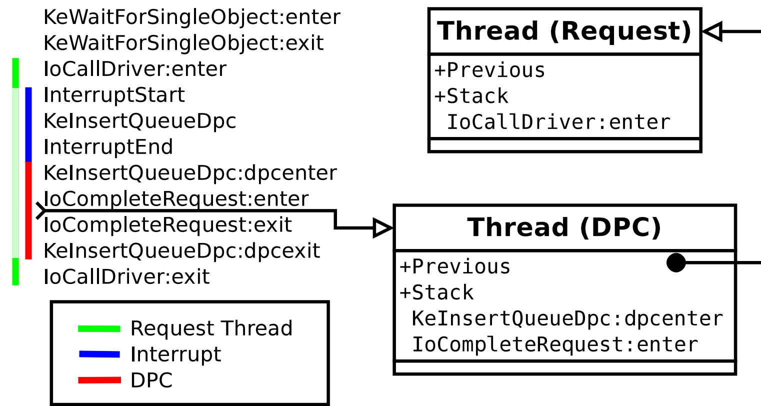


Figure 6.8: Result of parsing a sequence of events. We assume all events in this example have occurred on the same OS thread. Duration of threads is marked by vertical lines.

We also show how the thread stack looks (the thread stack is actually implemented as a linked list), just between the calls to `IoCompleteRequest:enter` and `IoCompleteRequest:exit`.

6.3.4 Grouping requests

The next problem is to decide what an individual diagram should contain. This is a question of finding a balance between drawing all requests in one giant diagram and giving each *IRP* it's own diagram. The latter option looks satisfactory, until we realize that drivers often issue additional subordinate *IRPs* to service the original one. As an example, refer to previously mentioned figure 6.5, where the file-system driver uses another *IRP* to read from the disk. We would like the diagrams to include at least this subordinate *I/O* requests.

However, saying that a request is subordinate to another is a different way of expressing the fact that the request is the cause of the other one. In some cases, reconstructing this kind of relationship from the recorded events is basically impossible. This is common problem when reconstructing information from the events, because the kernel code can communicate by means we do not monitor, like setting shared variables. Shared variables are contrived example, but it is common that new I/O request is issued inside a *DPC* routine, which in turn is scheduled by interrupt. Finding the cause of the interrupt is not easy and multiple request may even be serviced by a single interrupt. For this reason, the diagrams will always be an approximation in some regards.

We have chosen another, more broad criterion, for grouping requests. This helps us capture the cases where either the detection of subordinate I/O has failed, or where the requests are related by other means, such as in figure 6.4.

First, we define that two requests are in a relationship, precisely if there is a *parser thread* that operates on both of these requests. We can see that in figure 6.4, the two request are related, because the highlighted thread operates on both of them. First, it starts the *WRITE* request (pin 1) and then it completes the *READ* request (pin 2). The relationship can be straightforwardly extended to the remaining two types of objects: *DPCs* and *events*.

The relationship rule is symmetric with respect to the objects and it is also reflexive – if we know about an object, it is because some thread included an event operating on it. When we take transitive closure of the relationship, its equivalence classes are the sets of objects we will show in each diagram. The threads displayed will be those threads that have touched at least one object in the diagram.

This algorithm is able to correctly assigns subordinate I/O to the same diagram as their cause (in majority of cases, including the one where the I/O request is created by a *DPC* routine).

Objects still need to be in parent-child relationship, but only for the purpose of organizing the diagram in a shape of a tree. A simple set of heuristics is used, namely:

- *IRP* is child of the other if their *dispatch routines* are nested on the stack.
- *DPC* is child of either the *IRP* it completes or the event it signals.
- *event* is child of the request that has initialized it.

6.3.5 Identifying objects

The next requirement is to identify the different objects in the request, because we need to show objects in the resulting diagram. However, there is another reason: the algorithm for grouping requests (described in the previous section 6.3.4) needs to know whether two events refer to the same object.

At first glance it looks simple, since the instrumentation can store the memory addresses for all functions that manipulate objects. We could then assume that if the function call uses the same memory address, it refers to the same object. However this reasoning is flawed, because objects can reuse memory addresses. Consider the following piece of C code:

```
void Foo(){
  KEVENT done;

  KeInitializeEvent(&done, NOTIFICATION_EVENT, FALSE);
  StartWorkOnNewThread(&done);
  KeWaitForSingleObject(&done, Executive, KernelMode, FALSE, NULL);

  KeInitializeEvent(&done, NOTIFICATION_EVENT, FALSE);
  StartWorkOnNewThread(&done);
  KeWaitForSingleObject(&done, Executive, KernelMode, FALSE, NULL);

}
```

The code starts some work on background thread and waits for the event that will be signaled when the work is done. It then repeats the whole process once more.

Both calls to `KeWaitForSingleObject` use the same memory address (`&done`), but the objects they refer to should be considered separate for all practical purposes, because they were re-initialized in the meantime. The solution is to

notice the `KeInitializeEvent` call and consider the address `&done` as containing a new object from that point.

Less obvious is the case of *DPCs*. Each *DPC* is represented by a *C* structure `struct KDPC` and is initialized by a call to `KeInitializeDpc`. This structure is often initialized when the driver for particular device is started and is then repeatedly scheduled by the interrupt service routine (using `KeInsertQueueDpc`). Technically, all the `KeInsertQueueDpc` calls refer to the same *DPC* object.

However, that would mean that all requests serviced by this particular driver share a common *DPC* object. According to the rules for grouping requests, they would be drawn in the same diagram.

We solve the identification of objects by tracking their states. When the object reaches the *destroyed* state, it is assumed that all following functions using the same memory address refer to a different object. We should keep in mind that a particular definiton of object's lifetime affects how tightly are requests grouped into diagrams.

So, instead of equating the *DPC* object to the `KDPC` structure, it will represent a single invocation of the deferred routine.

I/O locations

Instead of tracking lifetime of the whole *IRPs*, we track their individual *IO locations*. State diagram of an *IO location* can be seen in figure 6.9. The *IO location's* life always starts with the entry to its dispatch routine (`IoCallDriver:enter`). It becomes destroyed, when it has been both completed (`IoCompleteRequest: completed`) and its dispatch routine has exited (`IoCallDriver:exit`). Synchronous requests are completed first and then their dispatch routine exits, asynchronous requests are completed some time after the dispatch routine returns `IO_STATUS_PENDING`.

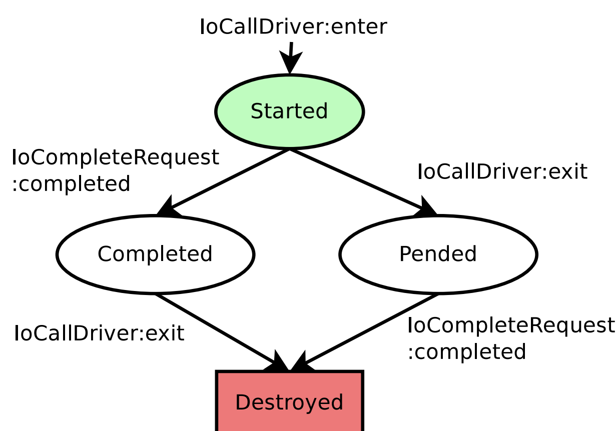


Figure 6.9: Lifecycle of an I/O request.

The *IO locations* are then represented as block parts, inside a block representing the whole *IRP*. *IO locations* are part of the same block if their *IRPs* have the same memory address and their dispatch routines are nested on the call stack. The second rule guards against problems with re-used *IRP* memory addresses.

Events

When we first implemented tracking of events, we occasionally got huge diagrams (more than 100 requests). This is caused by drivers signaling and waiting for kernel-wide events, which groups otherwise unrelated requests. For this reason, we have decided to reduce the tracking to only events created inside the request. Thus an events lifetime is defined by the scope of its enclosing function (we optimistically assume it is allocated on the stack).

The state tracking diagram is shown in figure 6.10. The event is created by its initialization function, `KeInitalizeEvent`. Since we keep track of call stacks for all threads, we know the function executing at the time `KeInitalizeEvent` was called. The kernel event is alive until this function exits. While the event is alive, all `KeWaitForSingleObject` and `KeSetEvent` functions using the same memory address are assumed to refer to this event.

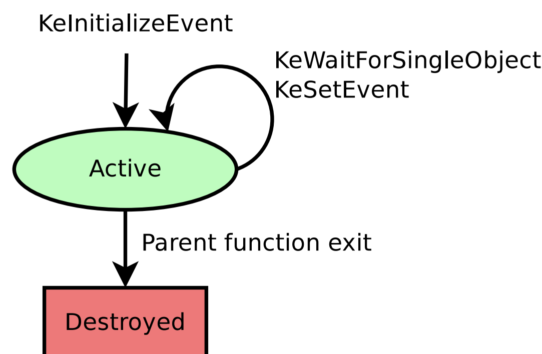


Figure 6.10: Lifecycle of a synchronization event.

DPCs

DPC objects capture the relationship between the thread(s) that have queued the *DPC* and the thread that is executing the *DPC* routine.

DPC is first noticed when it is scheduled for execution by `KeInsertQueueDpc` (figure 6.11). After that it can be scheduled again (may happen when two interrupts arrive in rapid succession), until it is finally executed by the OS (`KeInsertQueueDpc:dpcenter`). At that point the lifecycle restarts and any new scheduling of the same *DPC* will be considered as a new *DPC* object.

6.3.6 Sequential operation

The parser should be able to work by reading the events one by one and emit the diagrams as soon as possible. Both tracking of threads and of object lifetimes, as described, works this way. The problem is with grouping of threads and objects into diagrams, which takes a transitive closure of the objects relation. However, the equivalent result can be obtained by the following algorithm that only reacts to new events.

The algorithm keeps set of unfinished diagrams, where each diagram is a set of objects and threads that will be displayed in it. Each thread and object is in exactly one diagram. The set of unfinished diagrams is initially empty.

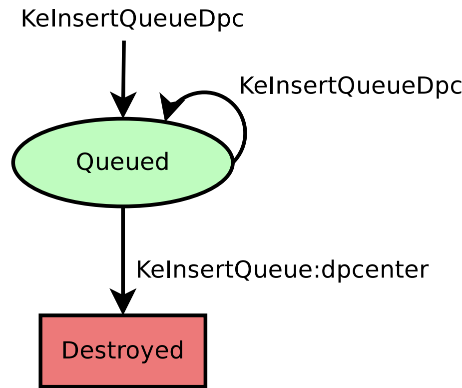


Figure 6.11: Lifecycle of a *Deferred Procedure Call*.

When a new *parser thread* is registered, it forms a new diagram by itself. If the thread creates a new object, the object is added to the threads diagram. If the threads

When a new thread is born (see *Parser threads* section 6.3.3), it forms a single new diagram. If the thread then creates an object (for example using `IoCallDriver:enter`), the object is added to the current activity. If the thread accesses already existing object (`KeSetEvent` on a currently live event), we replace the object's and accessing thread's diagrams with their union (in other words, we merge them).

From the rules above it follows, that there is no way for a diagram to change, if all its threads have exited and all its objects have been destroyed. Therefore, as soon as the last thread exits or the last object is destroyed, the diagram can be removed from the set of unfinished diagrams and shown to the user.

6.3.7 Tolerance to malformed data

The events may be "malformed", in a sense that a given sequence of the operations should never happen. This is most often caused by lost events, but we can not rule out the possibility of our parser relying on false assumptions (*I/O* handling in *Windows* can be very complex). In each case, the parser should never crash, but that is solely a question of programming discipline and proper error reporting. It is more difficult to make sure that the parser recovers after while and starts producing correct diagrams again.

One solution is allowing the threads and objects to tim-eout. Any object that is not touched for certain number of events, is forcibly moved to the *destroyed* state in its lifecycle. Likewise, if no new event appears on a thread, the thread is forcibly exited.

Sadly, this is only partial solution, since some types of malformed data can make the parser confused for a long time. For example a lost *end interrupt* event can lead it to conclusion that the interrupt thread has never ended. In the future, we may look into imposing a length limit on threads, for example. The actual limit values must be set in such way that they will be useful, but will not cause problems for real-life legitimate requests.

6.3.8 Saving the diagrams

To reduce storage requirements, we have decided not to store the diagrams produced by the parser, but only indexing information, from which they can be re-parsed. This also has the additional benefit of not having to design a storage format for those diagrams.

When the trace data is parsed, we throw away all the information produced by the parser, apart from the list of events contained in that diagram. We then store the diagram's first, last and main events into a separate *index file*. The diagram can be reconstructed by parsing it's range of events for a second time.

On the downside, this may cause speed degradation when the user tries to access the diagram. But in our experience, parsing the diagram takes less time then laying it out and drawing it.

6.4 Line routing

In the first prototype of *object view*, we have connected the pins using straight lines. This made the lines hard to follow for these reasons:

- Lines can be hidden behind blocks.
- When lines intersect at small angles, it is easy to confuse them.
- More lines in the same area can create an untraceable mesh.

The problems were improved by introducing visual cues. Drop shadows make the lines easier to follow, the same way outlined fonts can be read against complex backgrounds (all film subtitles have outlines). The lines also react to mouse pointer, changing their visual appearance.

6.4.1 Analysis

However, the routes the lines take must be improved. The problems we have identified are similar to the description of ideal graph edge, taken from a paper [41] describing the method used by *Graphviz* [42] package:

> Though we cannot formally define what it means for an edge connecting two vertices to appear natural, we believe good solutions avoid other vertices in the graph, stay close to a shortest path between the endpoints, do not turn too sharply, and avoid unnecessary inflections.

Typical diagram drawing software offers the user some form of zig-zag lines or splines to connect the edges. Often the user is expected to find a suitable route for the lines themselves, which is not acceptable for us . An example can be seen in application *PraxisLive*, based on the *NetBeans Visual Library* (figure 6.12). Edges drawn by this library do not need manual placement and are automatically routed. Since the lines are zig-zag, they they will most likely intersect in right angles (which is desirable). However, the lines do not try to avoid each other – indeed, in the left part of the figure, two lines are routed dangerously close and can be easily confused.

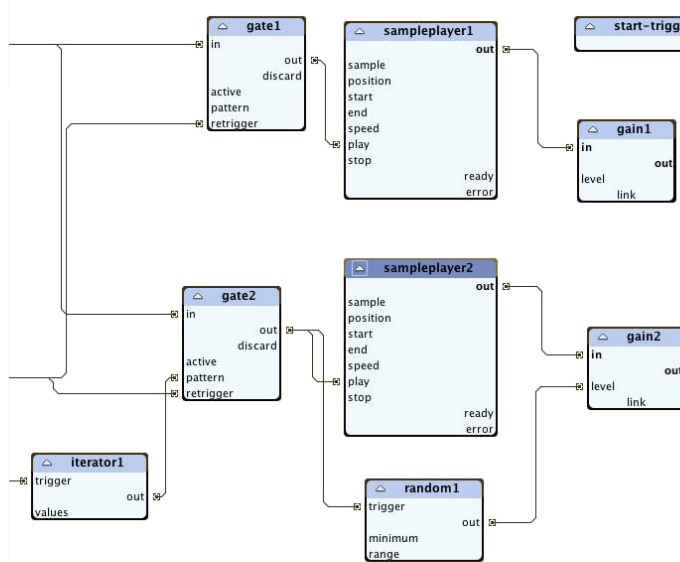


Figure 6.12: Node editor in the *PraxisLive* applicatio, taken from Mac Softpedia [43].

The *Graphviz* paper [41] identifies the line routing problem as a variant of path-finding in a two-dimensional space. They solve the problem by finding a shortest path between the two connected points and fitting a spline along the path. However, for simplicity the authors of the paper route the lines independently, meaning that they can not make any effort to avoid each other (the actual *Graphviz* implementation may be improved in this regard).

The problem is also remarkably similar to the routing of single-layer printed circuit board. Routing a circuit is a process of finding paths for the copper traces connecting pins of individual board components. The common solution for printed circuit boards is to impose a grid on the layout space and use a grid path-finding algorithm instead. The only difference is that we want to allow perpendicular traces to intersect.

We prefer the grid-based approach, although both come close to satisfying the visual requirements. Since our diagrams have a high density – a larger *IRP* has about 8 *IO locations*, each having three pins, we were worried about the mutual line avoidance in the *Graphviz* approach. Apart from that, it is a matter of aesthetic taste (zig-zag lines vs. splines).

6.4.2 Algorithm description

Grid path-finding is implemented by *Lee's maze routing* algorithm [39]. The algorithm is designed to find a shortest path from a source cell to a sink cell, avoiding obstructed cells. In our case, the obstructed cells are obstructed by blocks and previously routed lines.

Lee's maze routing is a wave propagation algorithm. It's operation is illustrated in figure 6.13, where it tries to find a path between legs of two chips (marked *s* and *T*). There are two phases. The first phase begins by marking the neighbors of the *s* cell with 1. The digits represent shortest distance from the cell to the cell *s*. In each step, the shortest-distance information is propagated to neighbors

of already marked cells, in a wave-like fashion, until the T cell is reached. The digits in the figure are given distinct colors to emphasize the wave-like patterns they form.

When the T cell is reached, a second phase, called *retrace*, begins. This phase reconstructs the actual shortest path from the marked cells by going to a cell with a lower mark in each step. This way it will eventually reach cell marked 1, a direct neighbor of the s cell.

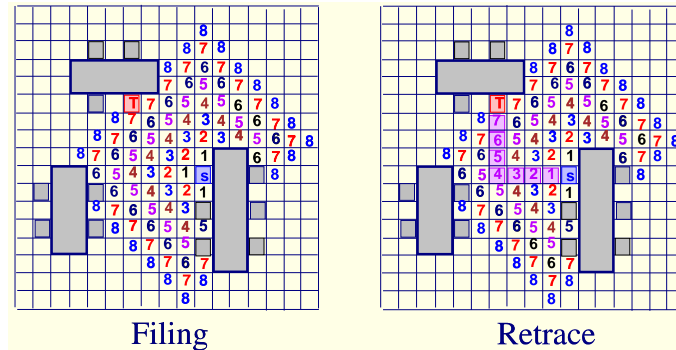


Figure 6.13: Operation of the Lee's algorithm, taken from Northwestern University slides [40]

The algorithm is slightly modified to recognize four types of cells: free, fully obstructed, horizontaly obstructed and vertically obstructed. The fully obstructed cells correspond to blocks and turning points (corners) of previously routed lines. The straight line segments are represented by horizontally or vertically obstructed cells. This allows the wave propagation phase to find the shortest path even if it crosses another line at right angle.

The algorithm is linear with regards to the number of cells in the grid. This means it is quadratic with regard to the size of the diagram. To speed up routing process for large diagrams, we make the wave not reach too far from the ideal connection line between the source and sink. We consider the ideal connection line between two cells to go first in straight horizontal then straight vertical direction. This makes the algorithm $O(n)$ with respect to the connection distance.

6.5 Additional views

In addition to *object view*, two other types of views are provided. They both emphasize different concepts (*call stacks* and *object ownership*). Unlike *object view*, the retain the chronological order of event view and complement it in this regard.

Since we consider *object view* to be the most important and because all three views use the same information provided by the parser, only a quick description is given for both views.

6.5.1 Call stack view

Call stack view consists of threads organized into columns with constant spacing. Events are drawn in the order they appear in the trace data, top to bottom, each

in the respective column.

Each event is drawn as a circle, with its name to the right of it. The names are indented, depending on the call stack depth.

6.5.2 Sequence diagram view

Sequence diagram is similar to *call stack* view, but does not show the names of the event (unless, of course, the user hovers above the event). Instead, lines are drawn between the columns, if there is a transfer of ownership of a block.

The transfer of ownership is deduced from the pinning of the events to objects. An object is considered to be owned by a thread, if the last event pinned to that object is on that thread. If an ownership changes, a line is drawn between the new and old pinned events.

Call tree view does not display kernel objects, but threads with events laid out in linear fashion. Each event is indented according to the call stack depth, forming a tree.

In *Sequence diagram* view, threads are displayed side by side. When a kernel object changes ownership, the handover is represented by an arrow between the two threads.

7. User Documentation

WinTrace is a program for helping students understand the internals of the *NT* kernel. It is focused on the way *NT* handles *I/O*. It records detailed information about the system activity (e.g. what drivers are called to handle *I/O*) to a *trace file* and displays to the user the events that have happened.

This chapter will explain how to use *WinTrace*. It is structured into common uses-cases, such as how to record system activity (section 7.2) or browse the recorded events (section 7.4).

It also contains the explanation of the supported kernel objects and their properties (sections 7.8). Events are not documented, because they directly correspond to function calls and the relevant documentation can easily be found on *MSDN*.

7.1 Requirements

WinTrace was tested on *Windows 7* and *Windows 8*, both 32-bit and 64-bit version. However, if you have 64-bit edition, extra steps described in section 7.2.1 must be taken.

If you only plan to view existing files captured on another system, all *Windows* versions higher then *Windows XP SP3* are supported.

7.2 Recording

Before recording new events, keep in mind that *WinTrace* is highly invasive debugging utility and it might crash the whole operating system. Save all your work before recording any events using *WinTrace*.

There are two options for recording events using *WinTrace*. The preferred one is the *GUI* utility, `wintrace.exe`. A reduced-footprint console version, `cwintrace.exe` is also avaialable.

7.2.1 64-bit versions of Windows

If you are running a 64-bit version of *Windows*, there are extra steps that must be taken before you can successfully record system activity with *WinTrace*. If you only plan to view existing events or have a 32-bit system, you can skip this section.

WinTrace requires modification of the monitored operating system. By default, *Windows* includes mechanism to prevent those modifications. This mechanism is known as *PatchGuard*.

A utility for disabling *PatchGuard* is included in the *WinTrace* distribution, in the `patchguard-killer` directory. Follow the included `readme.txt` to disable *PatchGuard* on your system. After disabling patchguard, you can proceed with the rest of this guide.

7.2.2 WinTrace GUI

First, you have to start *WinTrace GUI* application, by running the `wintrace.exe` file. To record system activity, *WinTrace* must be running with administrator privileges.

After starting *WinTrace*, wait a few seconds until the "Installing monitoring driver ..." message disappears and the "Record" button will become enabled. If *WinTrace* is not running as an administrator, the "Record" button will be grayed out and an error message shown in figure 7.1 will appear. Relaunch *WinTrace* as an administrator.

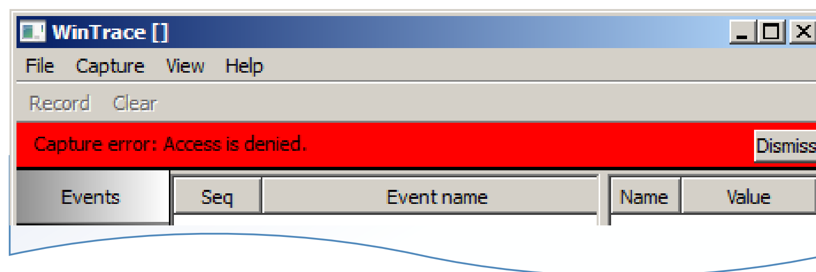


Figure 7.1: *WinTrace* running without sufficient priviledges.

Before starting the capture, you should decide where you want to store the events. The name of the currently opened file is shown in the *WinTrace*'s titlebar.

By default, *WinTrace* captures to a temporary file that will be deleted when the application is closed. If you plan on keeping the captured events, choose a location for the trace file. This is done by selecting "File → Capture to..." menu item and choosing the file. If you change your mind, you can get back to capturing to a temporary file using the "File → Capture to temporary" menu item.

After you have chosen the location for your capture file, you can start capturing events using the "Record" button or using the "Capture → Record" menu item. As soon as you click the "Record" button, you should see new events the "Events" tab and the statistics at the bottom of the screen will update.

To stop the capture, click the "Record" button again. *WinTrace* may still continue with some background processing of the events, but it does not capture any new ones. You can add new events to the capture file using the "Record" button again.

Process ignore

By default, *WinTrace* does not monitor its own activity. To include events caused by *WinTrace* itself in the capture, uncheck the "Capture → Ingore WinTrace Events" menu item.

7.2.3 Console

Events can also be captured using the `cwintrace.exe` utility. The easiest way to use it is just to run in it (again making sure that you launch it with administrator

privileges). `cwintrace` will automatically install the driver, and start capturing into a file in the current directory.

The file will be named `trace_YYYYmmdd_hhmmss.etr` (after the current time) and can be viewed by `wintrace.exe` at a later time or on different machine.

To stop the capture, simply close the `cwintrace.exe` window, kill the process or stop it using `Ctrl+C` key combination.

`cwintrace` also supports two command-line parameters:

- `/S` will enable silent mode, supressing most `cwintrace` output.

- `/I` will include events caused by `cwintrace`'s own activity in the recorded events. By default, they are excluded.

7.3 Opening previous traces

Recorded events, stored in the files with an extension `.etr` can be viewed by *WinTrace GUI*. These files are either produced by the *WinTrace GUI* itself or by `cwintrace`.

To browse an existing file, start `wintrace.exe`. It does not require administrator priviledges, if you want to open existing files. Select the "File → Open ..." menu item and choose the previously captured *trace file*.

As soon as you open the file, you should see events in the "Events" tab (assuming the file is not empty) and you can start browsing them. If the file is large, additional events will be loaded from the file in the background (reported by the "Indexing" and "Parsing" progress bars). The contents of the "Kernel Objects" tab will also be loaded from the file, so that you will have a snapshot of *NT executive objects* matching the time when the events were captured.

7.3.1 Index file

You may have noticed that a file with extension `.idx` was created in the same directory where your `.etr` file resides. This file speeds up the opening process and eliminates the "Indexing" and "Parsing" phases when opening the files, so that all your events and activities are available instantly. However, you can delete the file and it will be recreated.

This also means that the `.etr` file you are opening must be located on writable media, otherwise the `.idx` file can not be created and the opening will fail.

7.4 Browsing

Once you have started *WinTrace* and captured some events or opened an existing file, you have three ways of looking at the information in the file, selected by the tabs at the left side of the screen: "Events", "Kernel Objects", "Requests"

"Events" tab shows table of all the events that were captured. Each row in the table shows the event's order in the file, the type of the event (the "Event name" column) and the summary of the event ("Details" column).

To view all details about the event, select it in the table and it will show in the panel on the right. If some property of the event does not fit in the table, select it to display it in the text-area below.

"Kernel Objects" lists the named *executive objects* that were in the *NT object manager* namespace when the file was created. It is similar to the popular utility `WinObj`, but displays more details about the devices and drivers. It is controlled in similar manner to the "Events" tab, with the difference that the *executive objects* are organized into a tree. If you select and object from the tree, its details will be shown on the right, in the same way "Events" tab shows event's details. The various object types and their properties will be described in section 7.8.

"Requests" does not show individual events, but instead lists the whole *I/O* requests. If the user selects a request, the request will be shown as a diagram on the right.

7.5 Understanding request diagrams

There are three ways of showing requests in *WinTrace*. "Objects" view focuses on the data-structures used for *I/O* handling (*IRPs*, drivers, *I/O* locations), "Call tree" focuses on the function calls and "Sequence" shows the interaction of different threads in a form of sequence diagram. The diagram types can be selected using the push buttons in the diagram.

7.5.1 Object view

The first type of diagram is the "Object view". An example of a mouse *I/O* request shown in object view is in figure 7.2. The main information is the three objects that can be encountered in the diagrams: *IRPs*, *DPCs* and synchronization events. They are shown as blue blocks (for example the "Deferred Procedure Call" block in the diagram). *IRPs* also have a list of *I/O* locations displayed under the blue block (in this case, there is only one *I/O* location for the `mouclass` driver). There are no synchronization events in the figure 7.2.

The objects have brown dots (*pins*) on their sides. They represent the operations on those objects. For *IRPs*, the pins on the left side represent the entry into the *dispatch routine* and the pins on the right represent the exit from the dispatch routine and the completion of the *I/O location* using `IoCompleteRequest`. Similar convention is used for scheduling *DPCs* (left side), running *DPCs* (right side), signalling events (right side) and waiting for events (left side).

The pins are connected using lines representing the execution contexts. Green lines are ordinary threads, red lines are *deferred procedures* and *blue lines* are interrupts. The lines connect the pins in the order in which the operations were executed. There are also small white pins on the lines, representing other events that have happened in the execution context (but are not linked to any object in the diagram).

The diagram shown in figure 7.2 should be interpreted as: this request for reading from device `PointerClass0` consists of one *IRP*. The *IRP* has one *I/O* location, corresponding to the `mouclass` driver. The dispatch routine for the `mouclass` driver was called, but it has exited without completing the request

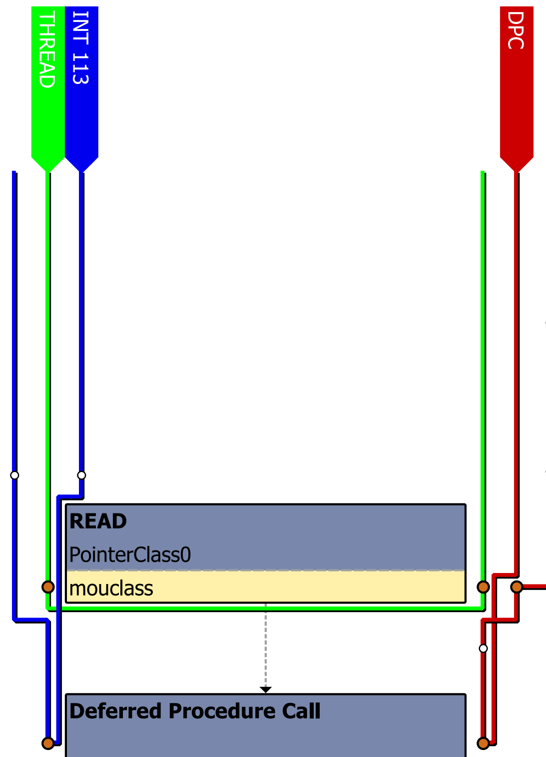


Figure 7.2: *WinTrace* diagram of a mouse *I/O* request in object view.

(green line). Later an interrupt arrived (blue line), scheduling a *DPC*. The *DPC* was executed (red line), finally completing the request.

7.5.2 Call tree view

The second type of diagram is the "Call Tree" view. It shows the execution contexts (interrupts, *DPCs*, threads) sides by side, with events represented by brown circles on the lines. The events are indented according to the nesting of function in the execution context.

One advantage of call tree view is that unlike object view, it maintains the order of the events accross different execution contexts.

The same request as in figure 7.2 is shown in figure 7.3, but rendered using the call tree view.

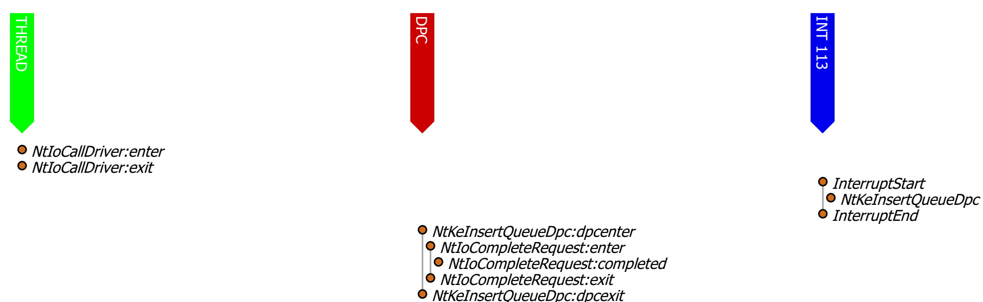


Figure 7.3: *WinTrace* diagram of a mouse *I/O* request in call tree view.

7.5.3 Sequence diagram view

The last avaialble type of diagram is loosely based on the “Sequence diagrams”. It focuses on how the objects are handled by different execution contexts, by emphasizing the change in the execution context currently handling the object by arrows.

The same request as in figure 7.2 is shown in figure 7.3, but rendered using the sequence diagram view.

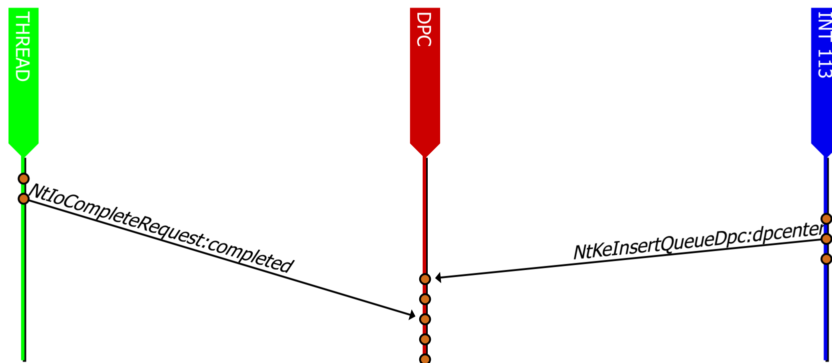


Figure 7.4: *WinTrace* diagram of a mouse *I/O* request in sequence diagram view.

7.6 Searching

Usually, the trace file contains a lot of events. If you are looking for a specific one, say the one related to your application, it may be burried among the vast number of other events related to the system’s other activity.

WinTrace provides highlighting feature to solve the problem. Select “View → Highlight” and use the dialog to select the highlighting criteria you want to use. The highlighting criteria consists of multiple filters or’ed together (new filters are added using the “New filter“ button). Each filter matches a given type of event.

Additionally, the filter can be restricted to match only events with certain field values. The field values you can match again depend on the event type you have selected for the filter. The available matching operators depend on the type of the field. Numeric fields (including pointer) can use the standart relational operators, while strings and binary buffers can use the “Contains”, “Equal”’ and “Not-Equal” operators.

Also be sure to specify the correct data-type if your are matching agains strings. For example, most *NT* file-paths are UTF-16 encoded and you have to select that you are entering an “Unicode“ string , not “ASCII” string.

This dialog is shown in figure 7.5. The highlighting criteria is configured to highlight start of dispatch routines for `READ` requests (function code 3) from a file name containing `chrome`.

After the user sets up highlighting criteria, it can select highlighted events using the “View → Next highlighted” and “View → Previous highlighted” menu items.

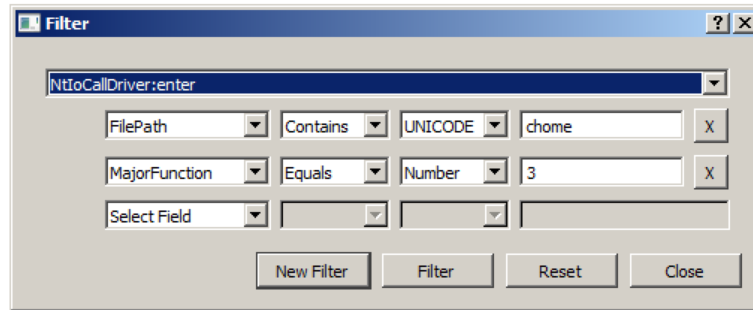


Figure 7.5: *WinTrace* highlighting dialog.

WinTrace does not offer a functionality to search through activities (request diagrams). However, you can view activity for a given event, using the "Find Related Activity". Combined with event highlighting, it allows you to find the activity you want.

7.7 Export

WinTrace can export data to be used by other programs. It can export the recorded events into *XML* format and also the individual diagrams, one by one as *PDF* files.

7.7.1 XML

This feature is available from the "File → Export as XML menu Item". It exports all the events stored in the curently opened file, in a XML format.

The exported file has the following structure:

```
<wintrace-export>
  <events>
    <event id="320906" type-id="2" type="NtIoCallDriver:enter">
      <int name="ProcessId">2304</int>
      <int name="ThreadId">2496</int>
      <int name="Processor">0</int>
      <int name="IRQL">0</int>
      <!-- ... skipped some fields for brevity ... -->
      <pointer name="DriverObject">0x85d0c4c0</pointer>
      <int name="IrpStackLocation">10</int>
      <utfstring name="FilePath">\tmp\bench.fio</utfstring>
    </event>
    <!-- ... rest of the events ... -->
  </events>
<wintrace-export>
```

WinTrace internally supports five types of event properties, exported as following tags:

- `pointer` – contains hexadecimal integer, with `0x` prefix.

- `int` – contains decimal integer
- `string` – contains text.
- `utfstring` – contains text.
- `buffer` – contains hexedecimal representation of the data, each byte with `0x` prefix.

7.7.2 Diagrams

Individual diagrams can be exported using the "View → Export request diagram". It exports the diagram currently displayed in the "Requests" tab.

The diagram is exported as vector graphics, in *PDF* format. The resulting file contains the whole diagram, not just the part currently displayed on screen.

7.8 Kernel objects reference

This section describes the supported *executive objects* in the "Kernel Objects" tab and explains their properties.

All executive objects have the following properties:

- `TypeName` – the type name of the object, as defined by the *NT* kernel
- `Path` – complete *NT* path to the object
- `HandleCount` – number of handles referencing the object.
- `ReferenceCount` – number of kernel pointers and handles referencing the object.
- `Pointer` – the memory address of this object inside the kernel.

The last three properties (`Path`, `HandleCount`, `ReferenceCount`) may not be displayed for some objects (mostly devices). This is caused by technical limitations inside *WinTrace*. The `HandleCount` and `ReferenceCount` properties have nonsensical value on 64-bit systems. For explanation of these values, see *The Case Of The Bloated Reference Count* article [44].

For devices, drivers and symlinks, *WinTrace* can display additional properties.

Devices

WinTrace can display additional properties for executive objects of type `Device`.

Since devices are represented using `DEVICE_OBJECT`s, *WinTrace* can display many of its properties: `DeviceReferenceCount`, `Flags`, `Characteristics`, `AttachedDevice`, `NextDevice` and `DriverObject`. For their explanation, see the *MSDN* documentation of `DEVICE_OBJECT`.

For faster navigation through the device stack, *WinTrace* also displays a property named `TopmostDevice`. This is link to the topmost device in the driver stack.

Driver

Driver objects are represented by the `DRIVER_OBJECT` structure. It contains the linked list of devices managed by the driver. This list is shown as the `ManagedDevices` property.

As with devices, there is the second list, called `TopmostDevice`, that contains the topmost devices in the managed device stacks, for easier navigation.

Symlinks

The only additional property displayed for symlinks is the target *NT* path, displayed in the property `Target`.

8. Developer Documentation

WinTrace is composed of several cooperating components. The most user-visible one is its GUI executable, named `wintrace.exe`. It can capture the monitoring data, display it and draw request diagrams.

The *GUI* cooperates with a driver, named `tracedrv32.sys` (or for 64-bit systems `tracedrv64.sys`). It is the core component that actually records the monitored events and transfers them to the *GUI*, where they are further processed, before being displayed. The driver is used, because it runs in kernel memory and so can do the hooking described in Chapter 5.

Instead of using the *GUI*, the user can choose to use the console version of *WinTrace*, called `cwintrace.exe`. This version is very lightweight, but can not display the events, it is only meant to capture them, to be later displayed by *WinTrace*.

Both utilities rely on a shared component, called *etr* (event trace) library. This is a static library, responsible for the communication with the driver and for accessing the files used by *WinTrace*. You can see in figure 8.1 how the various components fit together.

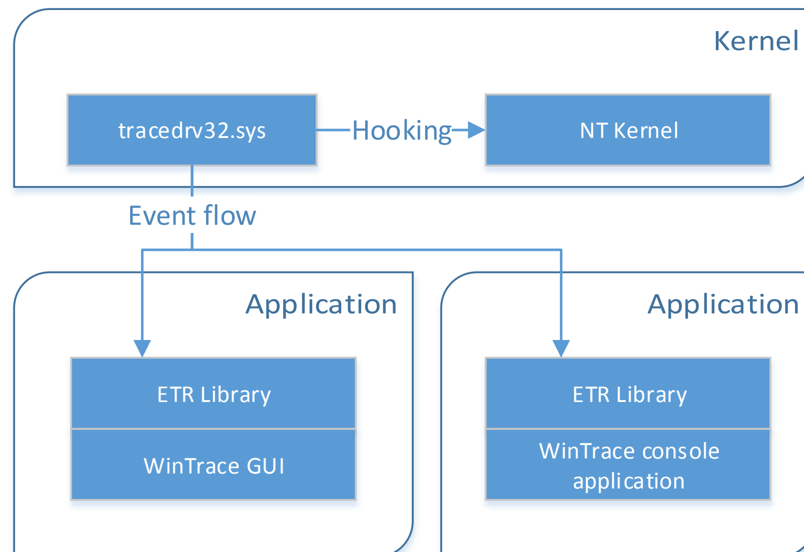


Figure 8.1: The major components of *WinTrace*.

The chapter will first describe the general organization of the project – its technologies and its directory organization. Before describing the individual components, the central piece of the *WinTrace* must be covered – the events and how are they stored in a file and memory. The individual components will be described in the order of their dependency on each other – first the driver, then *etr*, *GUI* and the *console app*)

8.1 Technologies

WinTrace is written in *C* and *C++* and built with *Microsoft Visual Studio 2013* compilers (but other versions should work too).

The kernel driver is written in pure *C*, a traditional choice for writing kernel drivers. It uses a lightweight third-party disassembler library, called *Hacker Disassembler Engine* [36], to correctly hook functions.

The userland components are written in *C++*. *WinTrace* uses the *Qt* library for displaying its *GUI*. The *etr* library and the console executable (`cwintrace.exe`) intentionally avoid using the *Qt* framework, to keep their footprints smaller. Another library used by the *WinTrace* GUI is *SQLite 3* [46], for indexing the trace files and keeping other information facilitating random access.

WinTrace uses two build systems, for the kernel and userland components. The kernel components are built using *Visual Studio* project files. *QMake* build system is used for the userland components, because it is the easiest way to build *Qt*-based applications. It is used even for the components not linked to *Qt*.

The last piece of the build process is the `evtgen.exe` utility. It takes *XML* descriptions of the events and generates code for both the kernel mode component (that captures the events) and the *WinTrace* GUI (that displays the events). The use of `evtgen` utility and the build process will be discussed in Chapter 9 in more detail.

8.2 Directory organization

The directory structure reflects the organization of components inside the *WinTrace* project.

- / `cwintrace` – console utility for capturing data
- / `etr` – shared library for access to *trace file*
  - / `tracedrvcomm.h` – the definition of the communication interface between the driver and the application
- / `events` – *XML* definitions of events
- / `evtgen` – utility for generating driver and kernel code from the event *XML* definitions
- / `gen` – files generated by `evtgen`
- / `tracedrv` – the hooking and monitoring driver
  - / `hooks` – code for hooking functions
  - / `disasm` – *Hacker Disassembler Engine*
  - / `capture` – ring buffer implementation, code for listing the *executive objects* and other code used by the monitoring code
  - / `tracedrv.sln` – *Visual Studio* driver build file
- / `wintrace` – *GUI* utility
- / `dist.cmd` – helper script for copying build artifacts to one directory

/ `evtgen.cmd` – helper script for running `evtgen`

/ `wintrace.pro` – *QMake* project for `cwintrace`, `etr`, `evtgen` and `wintrace`

8.3 Events

WinTrace monitors lot of activities in the kernel. The descriptions of those activities are called *events* and are the backbone of *WinTrace*.

Events are produced by the monitoring code in the kernel, each time a hooked function is executed. They include the processor id, thread id, process id, *IRQL* and the type of the *event* (e.g. interrupt start, dispatch routine called). In addition to these fields common to all events, each type of event has its additional information (e.g. interrupt vector number for interrupt start).

The supported types of events are described by *XML* files in the `events` directory. Each *XML* file describes what function to hook, the monitoring code (code run before and after the hooked function) and description of the events produced by the monitoring code, including the additional information about the events.

Our utility, called `evtgen.exe` reads those *XML* files and generates the hooking code for the driver (`gen/kernel_hooks.c`). It also produces the code necessary for parsing and displaying the events in the *WinTrace* UI (this code is in `gen/client_hooks.c`). This enables us to keep event definitions at one place, instead of splitting them across the *GUI* that consumes them and the driver than generates them. The detailed description of the *XML* will be described later, in Chapter 9.

During their lifetime, events are stored at various places. When events are created, they are first stored in a ring-buffer in the driver component. The client application (*GUI* or console) periodically issues *IOCTL* requests to the driver, flushing all events from the ring-buffer to a file. The *GUI* reads the events from this file when it needs to display or process them.

8.4 File format

To store the event data, a file format must be defined for the storage of events. The file format must support both random access (used by the *GUI*) and very fast appending of new data to avoid losing events when capture is in progress. To support these two contradictory goals, we have decided to actually use two files, each designed for one of the goals: *trace file*, containing the events and *index file*, providing random-access.

This section will only describe the *trace file*. The *index file* is only used by the *WinTrace GUI* and will be described in the section covering the *GUI* component.

8.4.1 Trace file

The events collected by the kernel are stored directly in *trace file*, a binary log with extension `etr`. The file starts with a simple header (`struct Header` in `etr/traceile.cpp`), containing signature, architecture of the system where the data was captured and version of the file.

The architecture (32-bit vs 64-bit) defines the sizes of some fields in the following parts of the file. *WinTrace GUI* is able to read both 32-bit and 64-bit versions of the file, regardless of the system it is running on.

8.4.2 Executive objects

After the header follows dump of all *NT execuive obejcts* at time of the file creation. They are used to provide *WinObj*-like capabilities in *WinTrace GUI* and also to show object names instead of pointers in the listing of events in the *GUI*.

The file format of the *executive objects* is defined in diagram 8.2. All objects start with a common header. The first field is the length of the following data. After that follows the object path (i.e. `\GLOBAL??\C:`), the type of the object and a flag specifying if more data follows.

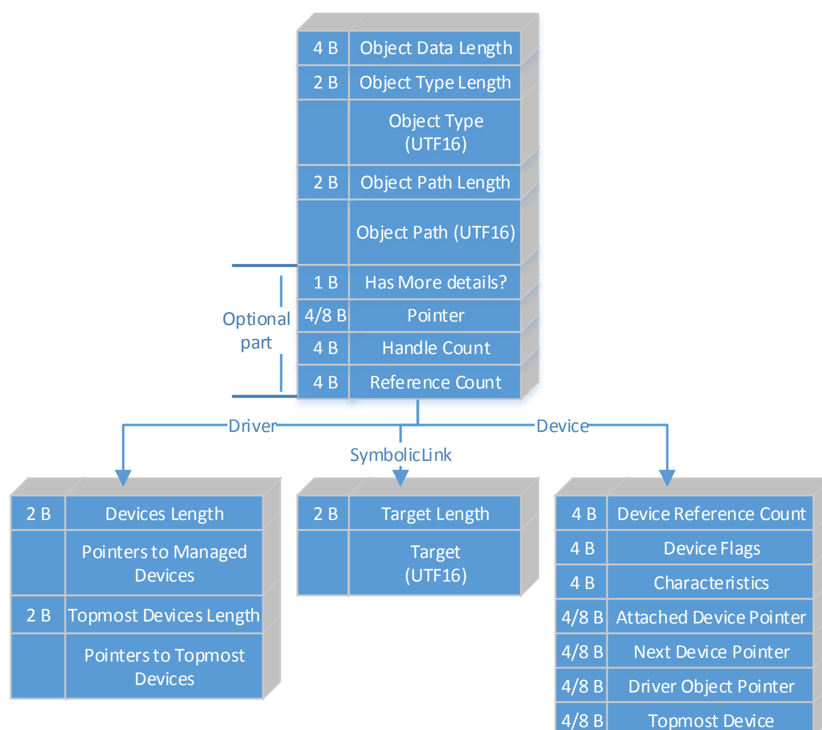


Figure 8.2: On-disk format of the supported *NT executive objects*.

The path and type strings are stored in common format used through *WinTrace*. The first two bytes specify the length of the string in bytes (not characters). After that follows the string characters encoded using UTF16 encoding, without any zero termination.

If the object's "has more details flag" is set (this flag is not be set, if the object could not be opened by the driver), the following data is the pointer and reference counts. If the objects is either device, driver or symlink, it has even more extended information, as defined in the diagram.

The end of *executive objects* listing in the file is marked with a special object of length zero. After that follows the captured events.

8.4.3 Events

The remainder of the file contains only the events themselves. Each event consists of event header and data. The header is defined as the following *C* structure:

```
struct Header{
  uint16_t size;      // size of the whole event, including this header
  uint16_t processor; // index of processor on which the event occured
  uint16_t irql;      // Interrupt Request Level at the time
  uint16_t typeId;    // type of the event, as defined in gen/client_hooks.h
  uint16_t pid;       // process ID
  uint16_t tid;       // thread ID
}
```

After the header follows the actual payload, consisting of individual *event fields*, specific to the given event type. Each event field may be either UTF16 string, *ASCII* string, binary data or number. Numbers are stored as 32-bit or 64-bit big-endian values, depending on the platform. Strings and binary data are stored as 16-bit length with the binary data immediately following (the same format as used in *executive objects*).

Although the events in the file are not explicitly numbered, we use their order in the file as *IDs*. By convention, the first event in the trace file has ID = 1.

8.5 Driver

WinTrace relies on a driver to provide its monitoring and object inspection capabilities. The driver is built in two version, 32-bit and 64-bit one, from a common source. The correct driver version is loaded by *WinTrace* when it is first started.

The *WinTrace* GUI or console applications communicates with the driver by means of a virtual device named `\GLOBAL??\TraceDrvMff`. This device handles *IOCTLs* for starting and stopping the monitoring, reading the events, listing *executive objects*, querying the driver version and changing settings. The list of *IOCTL* codes and related command structures are in the file `etr/tracedrvcomm.h`.

The core of the driver is the `main.c`, which calls in to the various components of the driver, when *IOCTLs* arrive. The components can be seen in diagram 8.3.

The core components are the *executive object lister*, *hooking engine*, *event ring buffer*, special hooking component for interrupts (only used on *x64*, see section 5.5.3 for details and the component maintaining the currently ignored process.

The above mentioned components in turn rely on other helper components, namely the library for in-memory *I/O* (buffers), the third-party disassembler and the list of functions to hook along with their hook handlers (generated by `evtgen`). The monitoring and hooking code also has two special memory allocating requirements. The first one is allocating memory during *DPCs* (`ExAllocatePool` does not support this) and the second one is allocating memory in 32-bit range of some other address (used for hook handler trampolines on 64-bit systems).

The following text will describe the top-level components, in the diagram's order, along with the components they rely on.

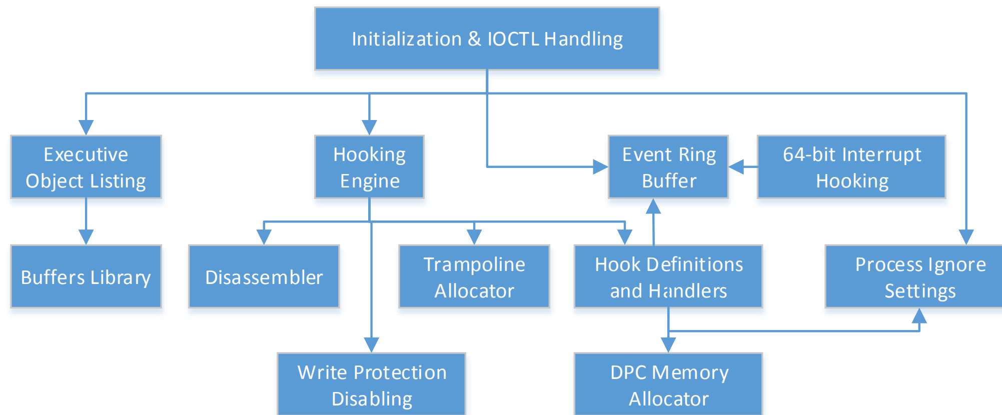


Figure 8.3: Components of the driver. Arrows denote dependencies.

8.5.1 Executive object listing

The file `capture/query_objects.c` supports listing of a given *NT object manager directory* and storing the list of encountered objects in a buffer. The format of the provided information is the same as the format of objects in trace file (described in section 8.4.2).

To write the *object information* into the buffer provided by the user application, *WinTrace* uses the file `etr/buffers.c`. This file provides functions for reading and writing to binary streams (similar to *C#'s* `MemoryStream`).

8.5.2 Hook engine

The key part of the driver is the hooking engine, that implements the hooking methods described in Chatper 5. The API is provided in file `hooks/hooks.h` and revolves around the `FUNCTION_HOOK` structure. User of this API allocates memory in non-paged pool for the `FUNCTION_HOOK` structure, initializes it using `InitializeHook`, giving information about the function to hook and the *hook handler*.

After that, the hook can be activated and deactivate at any time, using `ActivateHook` and `DeactivateHook`. The hook engine automatically takes care of choosing the appropriate hooking technique for the current architecture. The caller must be prepared that the hook activation and deactivation may fail, for example because of unusual function code.

The architecture-independent hook management functions (like initialization, getters, setters) reside in the `hooks/hook.c` file. The functions doing the hooking itself, that is `ActivateHook` and `DeactivateHook` are architecture specific and are in the `hooks/hook32.c` and `hooks/hook64.c` files. These files also rely on short assembler stubs (for saving registers), implemented in `hooks/hook_stub32.asm` and `hooks/hook_stub64.asm`. The project's build settings ensure that a correct file gets built for the target architecture.

The hooking needs to rewrite code, however, code segments are typically write protected. `hooks/write_protection.c` file exports functions for enabling and disabling write protection on the processor.

On 64-bit systems, the hook engine also needs to partially disassemble the hooked function. The third-part *Hacker Disassembler Engine 64* is included in the `disasm` directory in source code form.

Another problem that the 64-bit hook engine needs to solve is the allocation of memory for a full 64-bit jump, which does not fit in the function's padding. The memory for this trampoline must be in the range of the 32-bit jump in the functions padding. This support is realized in `hooks/near_memory.c`.

To find a suitable place for the memory allocation near another address, the driver obtains a list of loaded kernel modules and find to which one the given address belongs. It then tries to allocate memory before this module, using `MmMapLockedPagesSpecifyCache` and keeps track of this memory, for deallocation and sharing with other trampolines.

8.5.3 Hook definitions & handlers

The *hook handlers* are generated from the XML definitions by `evtgen`. They are placed in the `gen` directory, in file named `kernel_hooks.c`. This files contains the list of functions to hook, along with their hook handlers. The `main.c` file can then correctly setup `FUNCTION_HOOK` structures based on information in this file.

The hook handlers contain some boilerplate code needed to do process filtering and call the original function. They also include code segments from the *XML* definitions, to be executed before and after calling the original function. These segments collect the needed information and post events to the ring buffer, using the `EventPost` API from `capture/event_buffer.h`.

One of the common requirements of the hook handlers is to replace callbacks with a hooked version. As an example, consider this function for scheduling *DPCs*:

```
KeInsertQueueDpc(PRKDPC Dpc, PVOID arg1, PVOID arg2)
```

Apart from monitoring this function, we want to know when the *DPC* was eventually executed. To do this, we replace the `KDPC->DefferedRoutine` callback, so that our own code is run. However, our monitoring code needs to execute the original function, with the two arguments originally provided to `KeInsertQueueDpc`. To pass this information using the `arg1` pointer, memory needs to be allocated, but `ExAllocatePool` can not be used, because the *IRQL* is probably too high. Instead, atomic lookaside list defined in `capture/dpc_buffer.c` is used.

8.5.4 Event ring buffer

The ring buffer for storing events is provided by file `capture/event_buffer.c`. It supports multiple concurrent writers, without locking (using only atomic operations).

If a hook handler wants to post event to this buffer (and does not want to use the macros generated by `evtgen`), it can use the `EventPost` API. This calls takes an array of `EventField` structures, each field describing memory location and its size. The memory locations' contents will be stored to the ring buffer, along with data common to all events: process id, thread id, processor id and *IRQL*. Special version of this API for interrupt routines, with explicit *IRQL* parameter is named

`EventPostExplicitIrql` (remember that *IRQL* during interrupt handling may not be reliable on some platforms).

Second set of routines is provided for the readers of the buffer and for setting and tearing it up. The driver has two options of reading the buffer (both exposed to the userspace using *IOCTLs*). `ReadOutEventBuffer` copies the events from the buffer to caller-supplied memory and returns the number of copied events, their size and number of dropped events since the last call. This is a legacy mechanism used by older version of *WinTrace*.

Another option is to call `WriteEventBuffer`. This call will write the events directly to a file handle provided by the caller, avoiding copying the data to userspace. `WriteEventBuffer` returns statistics about the events transferred in a `ReadOutResult` structure. It contains not only information about event counts, but also indexing information. This information allows random access to the written events, without parsing them sequentially.

Both functions for flushing the ring-buffer wait for until the buffer is sufficiently full (about one third), before flushing it. However, it waits for only maximum of about one second. This mechanism prevents eating up of CPU by unnecesarilly frequent flushing of the buffer, yet still keeping the *UI* updated at reasonable rate.

8.5.5 64-bit interrupt hooks

64-bit interrupt hooking is realized in the file `idt.c`. Externally, the file exports a simple API for disabling and enabling the IDT hooks. When the interrupts are hooked, all hardware interrupts generate events in the ring buffer.

The hooking works by going through all processors on the system and modifying their *Interrupt Descriptor Table*, replacing interrupt handlers with our own. The interrupt handler is constructed from the template in the `hooks/idt_hook_64.asm` file, by filling in the interrupt vector number and the original handler.

8.5.6 Process filtering

WinTrace contains a rudimentary form of process filtering – it can exclude `PASSIVE_LEVEL` events from the utility's own process. This is to filter out superfluous trace events capturing the *WinTrace*'s own activity. The filtering must be activated be the userspace utility when it connects and is managed by `process_ignore.c`.

The `process_ignore.c` file also contains functions for querying if a given event should also be filtered out.

8.5.7 Reliability

Crashes in drivers usually bring down the whole system with a *Blue Screen of Death*. For this reason, the driver should be able to survive the crash of the more complicated and bug-prone controlling utility. When the handle of the controlling device is closed, all hook are automatically removed.

8.6 ETR Library

A small library is used for code shared between *WinTrace GUI* and `cwintrace.exe`. The library is named *ETR* (event trace). It provides convenience classes to manipulate the trace file and communicate with the driver.

It also contains two files defining the communication interface of the driver. The first one is `tracedrvcomm.h`, it contains *IOCTL* numbers, event header definition and other structures used by the *IOCTls*.

The second one is `buffers.c` and provides support for storing more complex structures (such as descriptions of *executive objects*) in a in-memory buffers.

8.6.1 Driver controller

`TraceDriver` is a class representing the driver described in section 8.5. It provides methods for installing, uninstalling, starting and stopping the driver.

After the client application (*WinTrace GUI* or console) has installed and started the driver using the `TraceDriver` class, it can use the `TraceCommand` class to communicate with the driver. `TraceCommand` describes the *IOCTl* request to be sent to the driver. To actually send requests, the `TraceCommand` object must be paired with `TraceBuffer` object, representing the buffer for receiving the data back from the driver. The application must set-up `TraceBuffer` large enough to accomodate the largest expected response from the driver.

8.6.2 Trace file

Second role of the *ETR* library is providing support for reading and writing events in the trace file. The trace file is represented by the `TraceFile` class. It can be used both to create a new trace file and open an existing one.

When a trace file is created, the `TraceFile` class can be used to write its header. It also cooperates with the `TraceDriver` to list all named *executive objects* and store then in the file. After the header is written, the application can use the `TraceFile` instance to append new events to the file.

On the other hand, if a trace file is opened, `TraceFile` reads the header and the list of *executive objects* stored in the file. The *executive objects* are passed to the provided `TraceFileObjectListener`, so that they can be displayed to the user.

While the application will only be appending new events at the end of the file, it may be reading events from multiple points in the file, using multiple threads (we will later see that *WinTrace GUI* actually does lot of background processing). For this reason, a separate class, `TraceFileCursor`, is used to read events from the file. It support sequential reading of events and also seeking. However, the seeking is not based on event IDs, but the application must provide raw file offsets. It is the application's responsibility to remember the offsets.

The events are read in *blocks*, each block comprising several events. In the current implementation, blocks contain exactly one event. The block mechanism is present to support event comppression, a feature that is not yet implemented.

8.7 WinTrace GUI

The main utility that the users will run is the *Qt Widgets* [45] *GUI*, `wintrace.exe`. It can capture new events, open existing *trace files* and show the data in the form of diagrams.

Conceptually, *WinTrace* is divided into four components, shown in figure 8.4: the data model, providing access to the data stored in the trace file (with the help of an index file), *Qt* models, feeding the tables and lists, custom widgets and background workers.

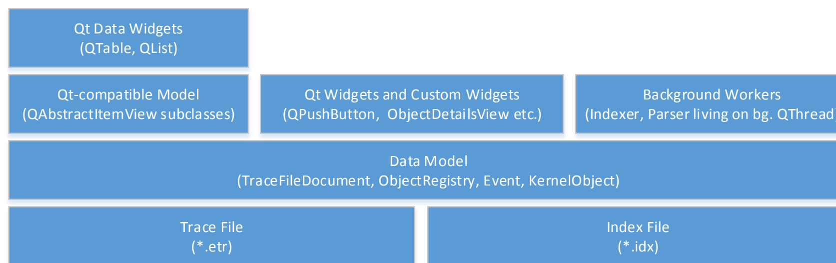


Figure 8.4: Layers of the *WinTrace GUI* application.

These components communicate using regular method calls and also using the *signals and slots* mechanism of *Qt framework*. *Signals and slots* behave similary to *events* in *C#*, but have a different threading semantics that is used to implement the background workers. For more information, see the *Qt* documentation topic *Signals and slots* [47].

8.7.1 Data model

The data model is collection of classes that keep track of the currently opened file, its events, *executive objects* and graphical diagrams (called *activities* internally and "Requests" in the *UI*).

The root of the *data model* is the `TraceFileDocument` class, representing the current *trace file* and its associated *index file*. The *index file* is a *SQLite* database, storing information facilitating random access to the *trace file*. The *index file* is created when a new *trace file*, or an existing one without *index file*, is opened. If an existing *trace file* is opened, the *index file* is recreated in the background, if necessary.

`TraceFileDocument` manages the life-cycle of these two files and supports the `open`, `new`, `newTemporary` and `close` operations. All other components are connected to `TraceFileDocument` and listen to the `closed` and `opened` signals (and their variants) and refresh accordingly.

`TraceFileDocument` itself does not keep events in memory, but only maintains the current number of *events* and *activities*. Other components (*Qt* models, custom widgets and background workers) only load the events when they need them.

New events and activities can be added to the `TraceFileDocument`, because new events may be recorded, or an existing *trace file* may be indexed in the

background. `TraceFileDocument` keeps other components up-to date when this happens, using signals.

Events and kernel objects

The events and kernel objects themselves are derived from class `DataObject`, which in turn is derived from `Cache::Object`.

`Cache::Object` is the common base class for reference-counted objects. Even if their reference-count reaches zero, they are not destroyed but are kept in a cache. This mechanism is used by the *UI* to keep recently accessed events in memory.

`DataObjects` are objects that have a description and certain number of properties (for example, *interrupt vector number* for and interrupt start event). Subclasses of `DataObjects` may also implement methods for displaying their description and formatting their fields in a human-readable manner.

Events are subclasses of class `Event`, deriving from `DataObject`. The concrete subclasses of events are generated from their *XML* descriptions and are stored in the `gen/client_events.h` file. They provide the concrete fields for the event type and the code for custom descriptions.

When the application needs to load an event from disk, it must have its own `TraceFileCursor` (from *ETR* library, described in section 8.6.2). Components running on *UI* thread may use the special class `UiTraceFileCursor` instead. It can ask `TraceFileDocument` to position its *cursor* at the wanted *event ID*. Because the file indexing is not precise, the seek may end up positioning the file a few events before the wanted one. It is the responsibility of the caller to skip those events.

For example, if the application wants to load a new event from a file, it can be done like this:

```
size_t block_size, current_id;
// ask the TraceFileDocument to position the file
traceFileDocument->seekToEvent(traceFileCursor, EVENT_ID, &current_id);
while(true){
  cursor.readBlock(&block_size); // read a next block of events
  for(size_t i = 0; i< block_size; i++){
    if(current_id++ == EVENT_ID)
      return Event::createEvent(traceFileCursor, traceFileDocument->is64();
    else
      // skip events until we find the one we want
      traceFileCursor->readEvent(NULL, NULL);
  }
}
```

The event's properties (or properties of any `DataObject`) can be accessed like this:

```
printf("Irp = 0x\%p\n", evt->number(NtIoCallDriver_enter::Irp);
```

The *executive objects* are also represented as `DataObjects`, they are instances of `KernelObject` class. Unlike events, they are kept completely in memory, because there is a limited number of them. Other then that, they work similarly to events. The class responsible for keeping track of them is `ObjectRegisry`, part of `TraceFileDocument`. Unlike *events*, their classes are not generated from *XML*, but hard-coded in the file `NtObject.h`.

Since both *events* and *executive* objects are derived from `DataObject`, they can be searched using the same mechanism. The `ObjectQuery` class is used for finding `DataObjects` that meet certain criteria. It enables the programmer to write a set of conditions over the object fields, like "`PathName` contains *Windows*". It also comes with an UI class `FilterDialog` for constructing these queries using drop-down lists. It is used for highlighting events and for resolving pointers to symbolic events.

8.7.2 Background workers

Most of the monitoring operation of *WinTrace* takes place in background. This consists of the recording of events, their indexing and parsing. The background workers are tied to the `TraceFileDocument` and update it as a works gets done. The changes in `TraceFileDocument` then propagate to the *UI*.

There are four background worker classes in *WinTrace*:

- `CaptureProvider` – installs and starts the *WinTrace* driver, flushes the driver's ring buffer regurarly using *IOCTL*.
- `OnlineIndexer` – fills the `events` table in the index file, as new events are captured.
- `ActivityParser` – parses the newly indexed events, grouping them into *activities* and storing them into the `activities` table.
- `FileIndexer` – reconstructs the `events` table in the index file by reading the trace file.

Each of those workers is derived from `QObject` and lives on a separate background `QThread`. They react to changes in the `TraceFileDocument` (opening, closing, new indexed events) by starting and stopping their background work.

In figure 8.5, you can see how the workers cooperate and process the events. In this case, *WinTrace* is capturing new events (the worker configuration is a bit different when processing an existing trace file).

Life of an event starts in the monitoring code inside hooked interrupts or functions. All event data is gathered and written to the lockless ring-buffer (5.6.3). If the buffer is full, they are dropped. The format of the events in the ring-buffer is the same as the on-disk format in *trace file*.

The user mode *WinTrace* application communicates with the kernel driver using *IOCTLs*. This is the responsibility of the `CaptureProvider` worker. It repeatedly instructs the driver to flush the ring-buffer by sending it the `CMD_WRITE_EVENTS` command.

The driver splits the information from the ring-buffer into two parts. The events themselves get written to the file provided by the `CaptureProvider`, but the driver also produces *indexing information*, consisting of file offsets of the *events*. For efficiency, these file offsets are recorded only once for each 4 kB segment of events. The *indexing information* is returned to the `CaptureProvider`. However, the information does not get immediately stored in the slot *SQLite* index file, but is only queued for storage. This helps to avoid *event drop*, because the controller can be fully devoted to reading the ring-buffer.

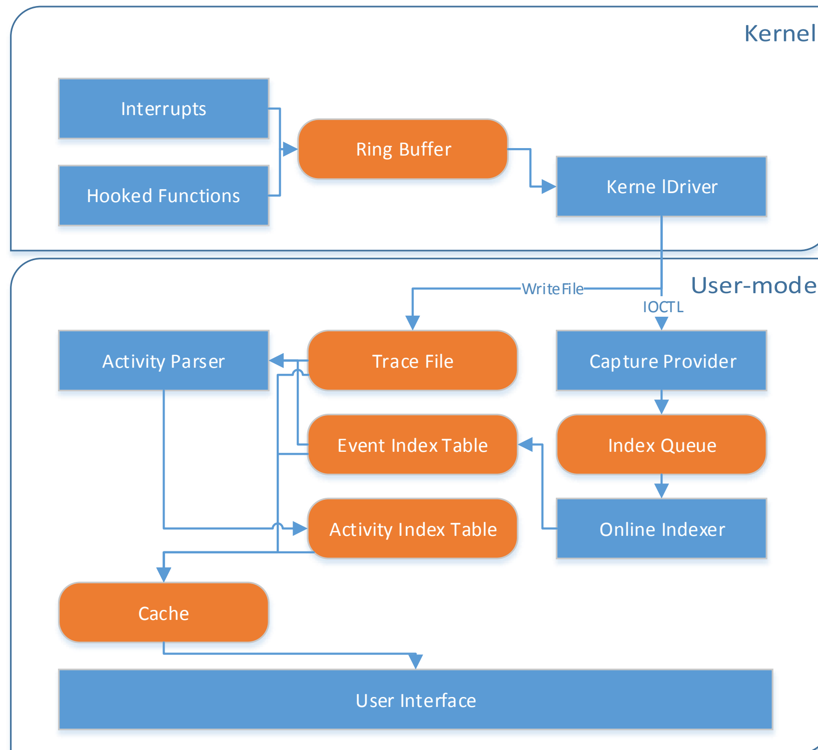


Figure 8.5: Flow of information in the *GUI* application and kernel driver. Active participants are blue, storage components are orange.

The responsibility of writing the index data to the *event index* (table `events` in the *index file*) is given to another thread, called `OnlineIndexer`. As soon as `OnlineIndexer` writes the indexing information to the *event index*, it informs the `TraceFileDocument` that new events are available.

The information about the newly indexed events propagates to the *UI*, which updates its list of events.

The newly indexed events also get noticed by the `ActivityParser`. It reads the new events from the *trace file* and identifies which events should be shown together in a diagram (algorithm in section 6.3.4). It stores the first and last events in the diagram to the table `activities` and informs `TraceFileDocument` that new activities are available. The change again gets propagated to the *UI*.

This was the description of *WinTrace* capturing new events. *WinTrace* can also regenerate the index file for an existing *trace file*. In that case, `FileIndexer` sequentially sequentially scans the trace file, periodically informing `TraceFileDocument` about newly indexed events. The rest of the pipeline is the same.

The remainder of this section will describe the background workers in more detail.

CaptureProvider

This is the driver's controller and manages its installation. But, most importantly, it is responsible for instructing the driver to flush the kernel ring-buffer to a file periodically.

When *WinTrace* GUI is started, `CaptureProvider` tries to connect to the kernel driver using the `TraceDriver` class from *ETR* library. If it can not connect, it tries to install the driver and connect again. If an error occurs, the `CaptureProvider` will go into *failed* mode and will not allow any tracing to happen (and the *GUI* buttons will be grayed out).

If the `CaptureProvider` has not failed, the capture of the events can be started using the *Qt* slot `start(HANDLE)`. The `HANDLE` is obtained from the currently open `TraceFile`. When capture is started, first all hooks are enabled and then the `CaptureProvider` periodically flushes the kernel buffer. Capture can be stopped using the `stop` slot, or automatically when `TraceFileDocument` is closed.

OnlineIndexer

The `CaptureProvider` also obtains indexing data from the kernel, in the form of `ReadOutResult` and array of `IndexPoints` (defined in the shared interface `tracedrvcomm.h`). It queues a message to the `OnlineIndexer` class running in its own background thread.

The `OnlineIndexer` waits for these messages and for each `IndexPoint` received it creates a row in the `events` table of the *index file*. It then emits a signal notifying the `TraceFileDocument` (running on the UI thread) about the addition of newly indexed events. The `EventTableModel` is connected to `TraceFileDocument` and will show these new events

ActivityParser

Another component listening for addition of new events is `ActivityParser`. It runs when it detects that there are unparsed events in the *index file* (either unparsed file is opened or new events are added to an index).

`ActivityParser` maintains its own `TraceFileCursor` for reading the new events and feeds them to the `EventStreamParser`. It listens to the description of activities by the parser and stores the ranges of events displayed in each acitivty into the `index file`.

FileIndexer

WinTrace GUI can open files with missing or incomplete index. `FileIndexer` is run when an existing file is opened and it seeks to the last indexed event. It then goes throught the rest of the file using a `TraceFileCursor` and recreates the missing indexing information.

8.7.3 UI and Qt models

The root of the *UI* hierarchy is the class `MainWindow`. This class is also directly responsible for the menus, toolbar and the mode selection bar on the left. It gives commands to the background workers and displays their status.

A screenshot of the *GUI* in all three modes is shown is in figure 8.6. The classes implementing the concrete areas of the *GUI* have been marked and will be described in more detail. The three modes are embedded inside `QStackedWidget` to select between them.

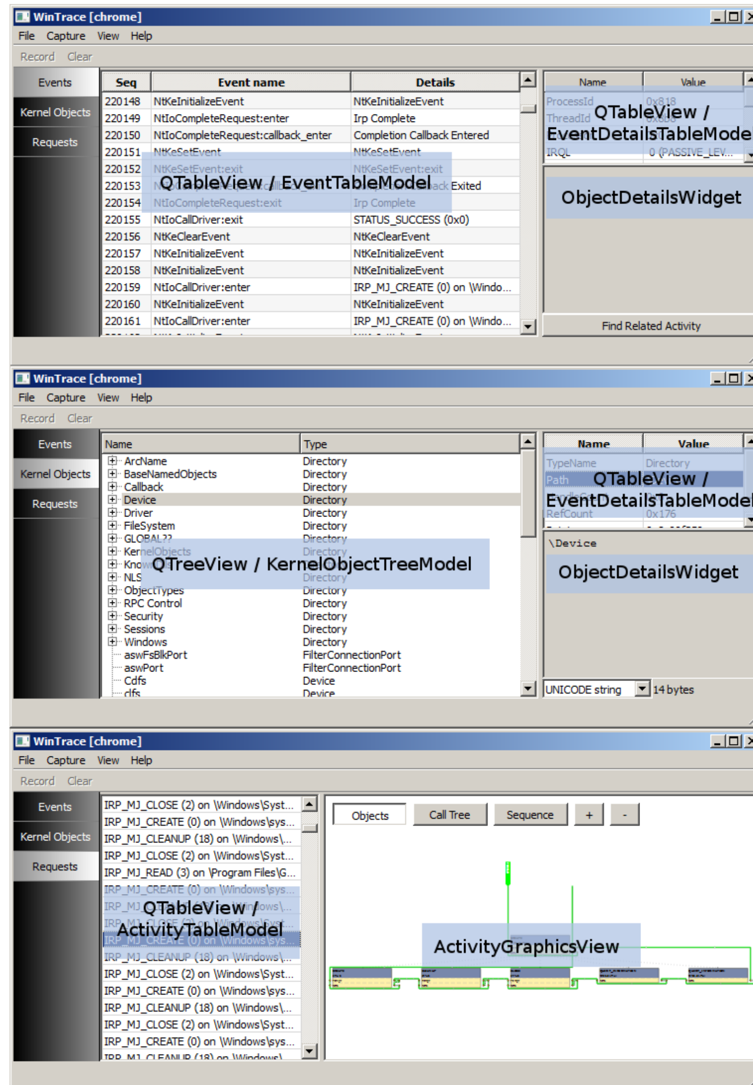


Figure 8.6: Screenshot of the *GUI* with the responsible classes marked.

EventTableModel

QT uses the model-view concept for its classes and `EventTableModel` implements the `QAbstractTableModel` for the *event view*. The model has two columns, one displaying the type of the event, the second one displaying the description provided by the event.

On the backed it is connected to the `TraceFileDocument` to keep track of the amount of events in the file. When it needs to actually display the events, it uses the `UiTraceFileCursor` to load them from the file.

Because each loading of *event* from the *trace* file means consulting both the *SQLite* database and seeking in the trace file, the class uses `Cache` class to cache the most recently displayed `Event` instances.

It reacts to the *signals* from `TraceFileDocument` about closing, opening and adding new events to the document to keep its view up-to date.

ObjectDetailsWidget

Part of the *event view* are the details of the currently selected object. They are displayed by widget called `ObjectDetailsWidget` that is capable of displaying any `DataObject` (and hence also `Event`). This widget is also used for displaying the details of `KernelObjects` on the second screen.

Upper part of the `ObjectDetailsWidget` is a QT table displaying data from `EventDetailsTableModel` . This model simply displays all fields of a given `DataObject`. It can also ask the `DataObject` for more appropriate text for the existing ones. For example, *Interrupt Request Level* values are shown including their symbolic names, such as `PASSIVE_LEVEL`. It can also ask the `DataObject` to provide link to a relevant `KernelObject`, using `ObjectQuery`. The property then has link appearance and will take the user to the object described by the `ObjectQuery` if clicked.

Lower part of the `ObjectDetailsWidget` just provides a `QTextEdit` area for displaying the currently selected property and to choose display mode for binary data.

KernelObjectTreeModel

The *executive objects* are displayed using `QTreeView` class from *Qt*. The underlaying model is `KernelObjectTreeModel`, that connects to `ObjectRegistry` and constructs a tree from the list of `KerelObjects` and their paths (e.g. `\Device\xyz`)

ActivityTableModel

Internally the groups of requests that constitute one diagram are called *activities* in *WinTrace*. Their list is shown using `QListView` from *Qt*, with the underlying model being `ActivityTableModel`. This model is connected to the `TraceDocument` and uses the `activities` table in the *SQLite* index file.

The names of the activities are derived from the first events of the activity. This model, like `EventTableModel` uses cache to remember the currently displayed names of activities (in the form of `Activity` class). Without the cache it would be even slower than `EventTableModel`, because the *main events* of activities tend to be scattered across the *trace file*, requiring more seeking.

ActivityGraphicsView

This view shows the currently selected *activity* as a diagram. Because outside this view *activities* only exist as a name and range of events, its graphical representation must be reconstructed. This is done by re-parsing the range of events constituting the activity. The parsing algorithm is factored out into the class `EventStreamParser` and its operation and interface will be explained with other *Utilities* classes.

The view itself is based on `QGraphicsView`, a scene-graph based drawing library. Each of the displayed widgets is implemented using a subclass of `QGraphicsItem`. The following graphical items are used for displaying the view:

- `BlockItem` – the whole block (*IRP*, *DPC*, *kernel event*).

- `BlockPartItem` – one text line inside the block, usually a *IO location*, but the block's name is drawn using this item too
- `PinItem` – either a big pin, next to a `BlockPartItem`, or a smaller free pin representing an operation not related to any `BlockItem`
- `ExplanationItem` – additional text displayed as a tool-tip when user has the mouse positioned above a pin.
- `ThreadItem` – displays the marker at the start of thread line and is responsible for keeping track of all the pins on the line.
- `ConnectingLine` – a line connecting two pins (or a pin and marker). It is composed of multiple control points, depending on how it was routed.
- `Arrow` – show the arrow between blocks, representing their parent-child relationship.
- `TransitionLine` – arrow representing change of ownership of one object (used in *sequence diagram view*)

Construction of the view's scene is done in `ActivityGraphicsView::show` and consists of following phases:

1. The activity is parsed and the *blocks*, *block parts*, *pins* and *threads* reported by the parser are created as graphical items.
2. The *blocks* are positioned (as a tree).
3. The arrows marking the parent-child relationship are drawn in the tree.
4. The pins are positioned near the *block parts* they belong to.
5. Connecting lines are routed and drawn, using class `LineRouter` implementing the algorithm described in section 6.4.

This construction sequence is valid for *object view*, the primary view. The *call stack view* and *sequence view* lack some of the features like blocks and routing of connecting lines and do not implement the respective phases.

8.7.4 Utilities and helpers

The classes described so far rely on set of utility classes to solve particular problems we have encountered. The two most complex ones, worth mentioning, are:

EventStreamParser

The interface of the *event parser* (as described in section 6.3), is represented by abstract class `EventStreamParser`. It uses the *push* parsing model. Events are fed to the parser using `parse` method and it in turn calls the consumer using the `::Callback` and `::SimpleCallbacks` listeners.

The `Callback` listener gives information about all the entities defined in the parser data model in section 6.3.2. `::SimpleCallbacks` reports only the events

that form an activity, on the grounds that the full description can later be re-parsed from those events.

The concrete implementation of parser for *NT I/O* requests resides in the class `NtRequestParser`. It recognizes synchronization events, *DPCs* and *I/O* requests. It is based on the `RequestParser` that provides common functionality for parsing function calls and tracking the current states of objects.

Collapsible slots

Qt offers the ability to connect signals and slots across threads. It then automatically takes care of sending messages between threads when the signal is emitted. This mechanism is well suited for controlling the background workers.

However, using this mechanism for sending the results of background workers to the *UI* thread easily overwhelms the *UI* threads, which can not catch up with the rate of updates and the cross-thread messages start queing.

Two classes, `CollapsibleEvent` and `CollapsibleSlot` are solution to this problem. They provide a transport mechanism for messages where the last message override all previous messages. An example of such message is the update of the number of indexed events. These two classes then take care of discarding old messages if the *UI* thread becomes overwhelmed.

8.8 Console utility

`cwintrace.exe` is a utility for capturing the *trace files*, but without the capability to display them. It does not even generate *index files*, since they can be recreated by the *GUI*.

`cwintrace.exe` delegates most of its functionality to the *ETR* library. It installs the driver using `TraceDriver`, initializes the `TraceFile`, dumps the *NT* manager objects to it and starts collecting events using `TraceCommand`.

9. Customization

WinTrace can be extended (at compile time) with new types of hooked functions and events. This is done using *XML* files defining the functions to hook and the structure of the events they report.

The *XML* files are processed by an `evtgen` utility, generating *C* code for the driver and the *GUI*.

9.1 Building WinTrace

To extended *WinTrace*, the code generated by `evtgen` must be compiled, with the rest of the *WinTrace* code.

WinTrace was built with the following tools:

- *Visual Studio 2013 Professional*, with *Windows Driver Kit*
- *Qt 5.3* built for *Visual Studio 2013* toolchain
- *Qt Creator* – optional, *QMake* can be used instead

Open the `wintrace.pro` project file with *Qt Creator*. *Qt Creator* will ask you to configure the project's debug and release build. Select the *Qt 5.3 for MSVC 2013* (or other configuration you are using). You also needs to create build directories. We recommand `../build/debug` and `../build/release` (relative to the source directory), because these paths are used by our scripts.

Then open the `tracedrv/tracedrv.sln` solution file with *Visual Studio 2013* and follow these steps:

1. Build *Debug* version of the *evtgen* subproject using *Qt Creator*.
2. Run `evtgen.cmd` to generate code.
3. Build all versions of *tracedrv* using *Visual Studio* (*Batch Build* can be used). This includes *Debug*, *Release*, 64-bit and 32-bit (and combinations).
4. Build *Debug* and *Release* versions of the *wintrace* project and its subprojects using *QtCreator*.
5. Run `dist.cmd` to gather build artifacts in the `dist-debug` and `dist-release` directories (optional).

9.2 EvtGen code generator

WinTrace relies on code generator to generate files for hooking functions and displaying events. These files are included in the *tracedrv* and *wintrace* projects. The following files are generated by *evtgen*:

- `kernel_hooks.c` – list o functions to hook and the related hook handlers
- `kernel_hooks.h` – hook ID constants and `kernel_hooks.c` interface

- `client_hooks.c` – list of hooks for the *GUI* (currently unused)
- `client_hooks.h` – interace for the list of hooks (currently unused)
- `client_events.c` – list of events, their names and their fields, code for generating event descriptions
- `client_events.h` – classes representing each type of event and interface to the event list
- `driver_version.h` – timestamp for the generated file version (used to detect driver/client version conflicts)

The syntax for running `evtgen` is:

```
evtgen $source_xml_dir $output_code_dir
```

`$source_xml_dir` is the path to the `events` directory containing the *XML* definitions. `$output_code_dir` is path to the `gen` directory, where the generated files will be placed.

9.3 Writing a new hook

This section will describe the *XML* definitions of hooks. The main parts of the definition are the description of the hooked function, definition of the events that are produced by the hook and the hook handling code.

Each hook is defined in its own XML file and looks like this:

```
<hook cc="stdcall/fastcall" function="HookedFunctionName"
      irql="override IRQL level">
  <!-- event descriptions -->
  <event name="event name">
    <int name="field name 1" />
    <pointer name="field name 2" />
    <utfstring name="field name 3" />
    <string name="field name 4"/>
    <buffer name="field name 5" />
    <description>description generation C++ code</desription>
    <properties>properties formatting C++ code</properties>
  <event name="event name">
  <!-- hooked function details -->
  <signature>argument list of the hooked function</signature>
  <retval>return value type</retval>
  <!-- hooking code tags -->
  <code>C code executed before the hooked function</code>
  <post-code>C code executed after the hooked function</post-code>
  <additional-code>void helper_function(){ ... }</additional-code>
  <include>additional_header.h</include>
</hook>
```

When writing hook definitions, you can check the *XML* for validity agains the schema bundled in `events/schema.xsd`. The meaning of the tags is explained in the following text:

attr `cc` inside `<hook>`, optional, default: `stdcall`

Calling convention of the hooked function, valid values are `stdcall` and `fastcall`. The calling convention is ignored when hooking on *x64* machines.

attr `function` inside `<hook>`, required

Name of the function. This function must be exported by the kernel, as it will be looked up by `MmGetSystemRoutineAddress`

.

attr `irql` inside `<hook>`, optional, default: `KeGetCurrentIrql()`

By default, the hook engine gets the current IRQL by calling `KeGetCurrentIrql`. The IRQL is used for process filtering. However, some hooked functions are executed at times when `KeGetCurrentIrql` can not be called. In that case, use this override to supply the hook engine with constant IRQL, like `HIGH_LEVEL`. You probably do not need to use this attribute, unless you are hooking interrupt handlers.

tag `<signature>` inside `<hook>`, required

Argument list of the hooked function, in C syntax, eg.:
`PVOID ptr, INT number`

tag `<retval>` inside `<hook>`, optional, default: `DWORD_PTR`

Return value of the hooked function. Can use any valid type defined in C or in `ntddk.h`, such as `NTSTATUS`.

tag `<code>` inside `<hook>`, required, unless `<post-code>` is provided

C code segment executed before the hooked function. This code will be executed by the driver and will typically post an event about the function entry to the ring-buffer. The functions and variables available to this code segment are discussed in section 9.4.

tag `<post-code>` inside `<hook>`, required, unless `<code>` is provided

C code segment executed after the hooked function. This code will be executed by the driver and will typically post an event about the function exit to the ring-buffer. In addition to the functions and variables discussed in section 9.4, this code segment can also access the `retval` variable (its type is defined by the `retval` tag), containing the value returned by the hooked function.

tag `<additional-code>` inside `<hook>`, optional

Additional C declarations and definitions included in the generated. These definitions and declarations will be available in `code`, `post-code` and `additional-code` segments.

tag `<include>` inside `<hook>`, optional

Additional C includes. These includes will be available in `code` and `post-code` segments.

attr `name` inside `<event>`, optional, default `event`

Name of the event. The *GUI* will display event names prefixed with their hook, in the form `hook:event`. The hook's name is taken from the file name in which it is defined. Because the event name is also used as a class name for the event, it must be valid C identifier.

tag `<int>` inside `<event>`, [0..n]

Defines an integer field in the event. Integer fields can have values corresponding to the architecture's int type (32-bit or 64-bit).

tag `<pointer>` inside `<event>`, [0..n]

Defines a pointer field in the event. Pointer fields can have values corresponding to the architecture's pointer type (32-bit or 64-bit). Technically, they are alias for `<int>`, but can be used for clarity.

tag `<bufffer>` inside `<event>`, [0..n]

Defines a buffer field in the event. Buffer fields can have variable length and will be displayed as a hex-dump by the *GUI* (altough this can be overriden).

tag `<string>` inside `<event>`, [0..n]

Defines an *ASCII* field in the event. String fields can have variable length and will be displayed as a text by the *GUI* (altough this can be overriden).

tag `<utfstring>` inside `<event>`, [0..n]

Defines an *UTF-16* field in the event. String fields can have variable length and will be displayed as a text by the *GUI* (altough this can be overriden).

tag `<name>` inside `<field>`, required

All field definitions (`<pointer>`, `<int>`, `<buffer>`, `<string>`, `<utfstring>`) must have this attribute. The name must be a valid C identifier.

tag `<description>` inside `<event>`, optional, default returns the event name

C++ code for generating short summary of this event as `QString`. This code will be embedded inside the *GUI* inside the `Event` subclass representing this event. For description of the methods and variables it can access, see section 9.5. Example:

```
return QString("Function xyz entered");
```

tag `<properties>` inside `<event>`, optional, no formatting by default

C++ code for formatting existing properties. The formatting is done throught calling functions of the `DataObject::PropertyListener` provided in the `properties variable`. For more information about the avaialable methods and variables, see section 9.5. Example:

```
    properties.add("MajorFunction",
        KernelConstants::majorFunctionName(number(MajorFunction)));
```

9.4 Driver code segments

The tags `code` and `post-code` define code segments that will be included in the hook handler. Apart from these two code segments, the hook handler function also contains some generated boilerplate code and code to call the original unhooked function.

The code in the `code` and `post-code` segments is guaranteed to have access to several variables it needs. First of all, it has access to the arguments of the hook handler (as defined by the `signature` tag). In addition to that, it can use the `hcb` variable of type `PFUNCTION_HOOK`, if it needs access to this structure. `post-code` segment also has access to the return value of the hooked function, in form of `retval` variable.

The `code` tag can change the values of the arguments in order to modify system behaviour. The `post-code` tag can change the return value of the function. In both cases, be careful because it is very easy to crash the system.

Because the code segments usually post events to the ring-buffer, `evtgen` provides several convenience wrapper macros around the ring-buffer *API*. The fields of an event are described by an event descriptor: an array of memory locations from which the field values should be copied (essentially a gather list).

The following macros are provided, supposing that `event` is an event with field `field`.

- `event(x)` – defines an event descriptor with the variable name `x`.
- `event_field(x, value)` – fills the event decriptor field `field`. `value` must be pointer to `DWORD_PTR` and must be valid until the event is posted. Moreover, the field must be defined as `int` or `pointer`.
- `event_field(x, size, value)` – fills the event decriptor field `field`. `value` must be pointer to memory buffer of given size in bytes and must be valid until the event is posted. Moreover, the field must be defined as `buffer`, `string` or `utfstring`.
- `event(x)` – posts an event `event`, with the values taken from the event descriptor.

9.5 GUI code snippets

One of the tasks of `evtgen` is to generate a class for each event found in the definitions. This class will implement methods for generating description of the event and formatting properties, with bodies taken from the `description` and `properties` tag.

Thus, both code segments have full access to the `Event` using `this` pointer. The most important methods of the `Event` class are the `number` and `buffer` methods, providing access to the event's fields contents. Both methods accept field index constants defined inside the generated class (e.g. `number(MajorFunction)`).

In addition to the event methods, both code segments have access to the `ObjectRegistry` class, using the `registry` variable. `ObjectRegistry` holds the *executive objects* and allows them to be queried using variety of criteria. This is

usefull for resolving pointers to the object names, or linking properties to those objects.

The central class for finding *executive objects* is the `ObjectQuery` class, constructed by `ObjectRegistry::query`. `ObjectQuery` consist of several conditions or'ed together, represented by `Filter` class. Each `Filter` defines the type of object that will be matched and a list of and'ed conditions over the object's fields. To actually build the `ObjectQuery`, you can use a *fluent* interface, like this:

```
registry.query().filter(NtObject::ktype())
    .where(NtObject::Pointer, 0xFA809090).next()
```

The `where` is used to add new conditions to a filter and `next` is used to close the current filter (and return the whole query). The result of a query can be obtained using the `run` method. `NtObject::Pointer` is the field index of the memory address field for an *executive object*. For other supported fields, see the generated documentation. `NtObject::ktype()` returns the generic type of all *executive objects*. Other supported objects are `NtDriver`, `NtDevice` and `NtSymlink`.

Queries often come useful when formating existing properties. This is the job of the code inside `properties` tag. This tag has access to additional variable, `properties` of type `DataObject::PropertyListener`. Through this object, it can change the property list of an event in two ways.

First, it can add completely new property. The `add` method accepts the property name and the property value. If a name of an existing property is given, the property is overwritten (this is often used to replace enumeration values by their names).

Second, it can link the property to some *executive object*. The user can then click the property to open that object. This is done using the `link` method, which takes the property name and an `ObjectQuery`.

If you are not sure about any details, consult either the generated documentation for the *GUI* or the existing definitions in the `events` directory.

Conclusion

The goal of this thesis was to write a tool for inspecting the *I/O* system in the *Windows* kernel, focused on the needs of students. Three main goals were specified in the Chapter 1:

1. inspection and monitoring of *I/O* objects, offering more information than comparable tools
2. easy and understandable representation of the information
3. minimal configuration and set-up, ease of use

Goal evaluation

Based on the analysis of the existing tools in Chapter 3, we have designed our tool, *WinTrace*. It is similar to existing tools like *IrpTracker* [17] and *Process Monitor* [19], but offers more information. Notably, we include information about *DPCs*, *interrupts* and *synchronization*.

To make the information easier to understand, *WinTrace* formats the information in the form of graphical diagrams. It even offers three types of diagrams, each having different level of detail and emphasizing different concepts. This greatly enhances the tools usability for students, who do not have to "decrypt" the raw streams of events and make sense of it. Instead, the data can be viewed on a request-by-request basis, graphically. Our algorithm for visualizing the data can even find interesting relationships between requests and show them in one diagram.

WinTrace is also reasonably easy to use. On 32-bit systems, it requires no installation and set-up and the user only needs to click on the "Record" button. Regretfully, to monitor low-level system activity, *WinTrace* must be very intrusive and modify the running system. Thus, special precautions must be made on *64*-bit machines. *Kernel Patch Protection* (aka *PatchGuard*) must be disabled, by using a third-party utility shipped with *WinTrace*. This is a slight inconvenience for the user.

Despite this last technical problem we consider the work to be successful and *WinTrace* to be useful addition the existing set of tools.

Additional achievements

WinTrace can also be useful to driver developers and other experienced users, as a debugging tool, because an ability to extend *WinTrace* with monitoring of new functions was implemented. *WinTrace* can be extended with new monitoring points by writing their *XML* definitions and recompiling.

The thesis has also shown some of the caveats of hooking functions and interrupts in *Windows NT* kernel by abusing the hotpatching mechanism and could be used as a guide. Many articles available on the Internet are not complete and may fail on 64-bit platforms, so this will be useful to anyone doing kernel hooking.

Future work

During the development of *WinTrace*, we have gathered lot of ideas how the tool could be improved, that were not implemented due to time constraints.

The most important addition would be the ability to make *WinTrace* more similar to existing tracing software, like *DTrace* [1] or SystemTap [2]. The difference between *WinTrace* and this software is that new monitoring points are added to *WinTrace* at compilation time, whereas *DTrace* and *SystemTap* have rich scripting languages that allow run-time definition of new monitoring points (called probes in their terminology). Allowing run-time configuration of probes would make *WinTrace* even more useful as a general debugging tool, while it could retain the current easy to use *GUI*.

Apart from this ideas, there are lot of minor features that could be implemented:

- Make *WinTrace* work on 64-bit systems out of the box, ideally by masking itself from *PatchGuard* using virtualization.
- Monitor communication between the hardware and the driver (io register read and writes), possibly using virtualization to do so.
- Monitor alternate *I/O* paths, like *fast I/O*.
- Monitor API calls, like `NtReadFile`, to put *IRPs* into perspective.
- Extend the integrated *Object Manager* viewer with more properties and *PnP* relations, to bring it on par with *DeviceTree* [14].
- Further improve line routing by using weighted cells to avoid concentration of lines in small areas.
- Test the hooking implementation on bigger sample of exported methods.
- Make the file format compatible between different versions of *WinTrace*.
- Compress captured data transparently.

Bibliography

[1] Sun Microsystems, contributions by Joyent. *illumos Dynamic Tracing Guide.* http://dtrace.org/guide/, [online March 2015].

[2] Prasad, Vara, Cohen, William, Ch. Eigler, Frank, Hunt, Martin, Keniston, Jim, Chen, Brad. *Locating System Problems Using Dynamic Instrumentation.* Presented at: Ottawa Linux Symposium, 2005.

[3] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* January 2015. Intel order number: 325384-053US.

[4] Russinovich, Mark. E and Solomon A., David with Ionescu, Alex. *Windows Internals*, fifth edition. Washington, DC: Microsoft, 2009. ISBN 9780735625303.

[5] Zachary G. Pascal. *Showstopper! the Breakneck Race to Create Windows NT and the Next Generation at Microsoft.* United States: E-Reads, 1994. ISBN 9780759285811.

[6] Microsoft Corporation. *Kernel-Mode Driver Architecture documentation* . MSDN library, https://msdn.microsoft.com/en-us/library/windows/hardware/ff557560(v=vs.85).aspx, [online May 2015].

[7] Windows Sysinternals. *Microsoft* Corporation. , [online May 2015].

[8] *What's Cooking in PulseAudio's glitch-free Branch.* http://0pointer.de/blog/projects/pulse-glitch-free.html, [online July 2015].

[9] Microsoft Corporation. *Virtual WiFi.* http://research.microsoft.com/en-us/um/redmond/projects/virtualwifi/faq.htm, [online May 2015].

[10] Oracle Corporation. *Oracle VM VirtualBox User Manual*, Chapter 12. https://www.virtualbox.org/manual/ch12.html#ts_debugger, [online May 2015].

[11] snare. *VMware debugging II: "Hardware" debugging.* http://ho.ax/posts/2012/02/vmware-hardware-debugging/, [online May 2015].

[12] Russinovich, Mark. *LiveKd.* https://technet.microsoft.com/en-us/sysinternals/bb897415.aspx, [online March 2015].

[13] Ntdebugging Blog *Part 1 - ETW Introduction and Overview.* http://blogs.msdn.com/b/ntdebugging/archive/2009/08/27/etw-introduction-and-overview.aspx, [online March 2015]

[14] *OSR Online Download Section.* http://www.osronline.com/section.cfm?section=27, [online March 2015].

[15] Plantamura, Vito *Open-Source Single-Host Kernel Debugger for Windows 2000 and XP.* http://bugchecker.com/?about, [online March 2015]

[16] Fattori, Aristide, Paleari, Roberto, Martignoni, Lorenzo, Monga, Mattia. *Dynamic and Transparent Analysis of Commodity Production Systems*. Presented at: ASE 2010.

[17] OSR Online. *IrpTracker*. http://www.osronline.com/article.cfm?article=199, [online March 2015]

[18] Russinovich, Mark. *WinObj*. https://technet.microsoft.com/en-us/library/bb896657.aspx, [online March 2015].

[19] Russinovich, Mark. *Process Monitor*. https://technet.microsoft.com/en-us/library/bb896645.aspx, [online March 2015].

[20] Matoušek David. *Nástroj pro kontrolu systémových volání pro Windows*. Praha, 2007, diplomová práce. Charles University, Faculty of Mathematics and Physics .

[21] BindView Corporation. *Strace for NT*. http://seriss.com/people/erco/ftp/winnt/strace/.

[22] Erlingsson, Úlfar, Peinado, Marcus, Peter Simon, Budiu Mihai. *Fay: Extensible Distributed Tracing from Kernels to Clusters*. Presented at: ACM Symposium on Operating Systems Principles (SOSP), 2011

[23] Passing, Johannes, Schmidt, Alexander, von Löwis, Martin, Polze, Andreas. *NTrace: Function Boundary Tracing for Windows on IA-32*. ISBN: 978-0-7695-3867-9, presented at: 20th Working Conference on Reverse Engineering (WCRE).

[24] Kenniston, Jim, Panchamukhi, Prasanna S, Hiramatsu, Masami . *Kernel Probes*. Linux GIT tag v3.19.3, Documentation/kprobes.txt.

[25] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*. https://software.intel.com/en-us/articles/pintool, [online May 2015].

[26] MIT ?. *DynamoRIO - Dynamic Instrumentation Tool Platform*. http://www.dynamorio.org/, [online May 2015].

[27] kdm (kodemaker). *NTIllusion: A portable Win32 userland rootkit*. http://phrack.org/issues/62/12.html, [online July 2015].

[28] Ermolov, Mark, Shiskin, Artem. *Microsoft Windows 8.1 Kernel Patch Protection Analysis*. http://www.ptsecurity.com/press/Windows_81_Kernel_Patch_Protection_Analysis.pdf, [online March 2015].

[29] Deng, Zhui, Zhang, Xiangyu, Xu, Dongyan. *SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization*. http://www.cerias.purdue.edu/assets/pdf/bibtex_archive/2013-5.pdf, [online March 2015]. Purdue University, West Lafayette, IN 47907-2086.

[30] Microsof Research. *Detours*. http://research.microsoft.com/en-us/projects/detours/, [online March 2015].

[31] Plantamure Vito. *Kernel Detours*. http://www.vitoplantamura.com/index.aspx?page=kdetours, [online March 2015].

[32] Easy Hook. https://easyhook.codeplex.com/SourceControl/latest#trunk/EasyHookSys/main.c, [online March 2015].

[33] Chen Raymond. In his blog *Old New Thing*. http://blogs.msdn.com/b/oldnewthing/archive/2011/09/21/10214405.aspx, [online March 2015].

[34] EliCZ. *Microsoft Patching Internals*. http://www.openrce.org/articles/full_view/22, [online March 2015].

[35] Microsoft Corporation. *Raw Pseudo Operations*. MSDN library, https://msdn.microsoft.com/en-us/library/ms235231.aspx, [online March 2015].

[36] Patkov, Veacheslav. *Hacker Disassembler Engine*. http://vxheaven.org/vx.php?id=eh04, [online May 2015].

[37] Johnson Peter. *YASM Manual - win64 Structured Exception Handling*. https://www.tortall.net/projects/yasm/manual/html/objfmt-win64-exception.html, [online March 2015].

[38] Microsoft Corporation. *Example WDM Device Stack*. MSDN library, https://msdn.microsoft.com/en-us/library/windows/hardware/ff544545%28v=vs.85%29.aspx, [online March 2015].

[39] Dion Jeremy. *Fast Printed Circuit Board Routing*. Western Research Laboratory, Digital Equipment Corporation. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-88-1.pdf, [online March 2015].

[40] Zhou Hai. *EECS 357 Introduction to VLSI CAD* slides. http://users.eecs.northwestern.edu/~haizhou/357/lec6.pdf, [online March 2015].

[41] Dobnkin, David P., Gansner, Edmen R., Koutsofios Eleftherios, North Stephen C. *Implementing a General-Purpose Edge Router*. http://www.graphviz.org/Documentation/DGKN97.pdf, [online May 2015].

[42] *Graphviz - Graph Visualization Software*. http://www.graphviz.org/, [online May 2015].

[43] *Praxis-LIVE_3.jpg*. http://i1-mac.softpedia-static.com/screenshots/Praxis-LIVE_3.jpg, [online May 2015].

[44] *The Case Of The Bloated Reference Count: Handle Table Entry Changes in Windows 8.1*. Alex Ionescu. http://www.alex-ionescu.com/?p=196, [online July 2015].

[45] Digia Plc. *Qt Developer Resources*. http://www.qt.io/developers/, [online May 2015].

[46] Hipp D. Richard . *SQLite Home Page*. https://www.google.cz/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8&client=ubuntu#q=sqlite3, [online May 2015].

[47] Digia Plc. *Qt Documentation - Signals & Slots*. http://doc.qt.io/qt-5/signalsandslots.html, [online May 2015].