UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

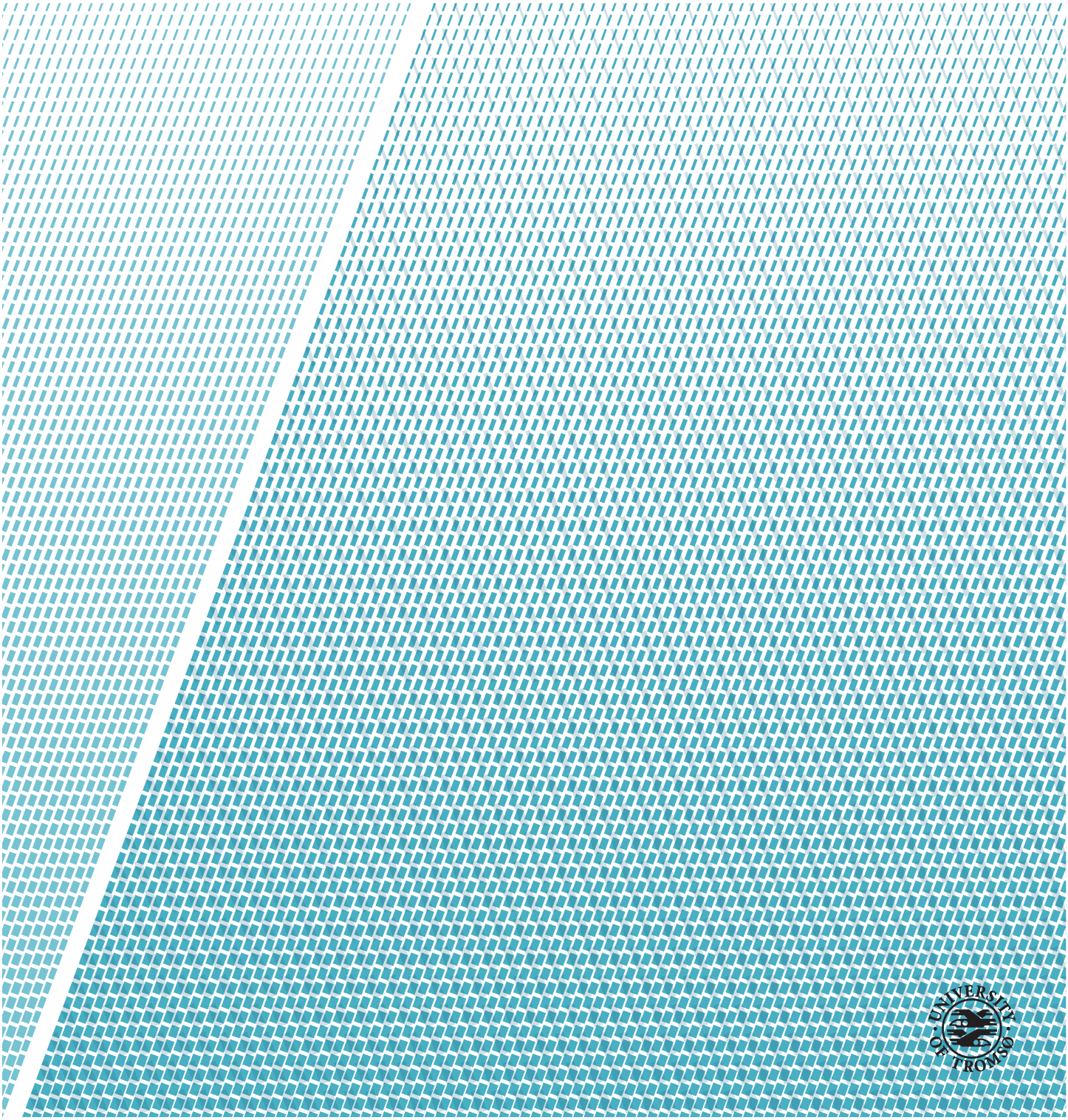
Casuar

A Protected Library OS for running Windows applications on top of Vortex

—

Erlend Helland Graff

INF-3990 Master thesis in Computer Science June 2015



"Ow! My brains!"
–Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

Abstract

Today, virtual machines (VMs) are commonly employed to encapsulate and isolate workloads in the cloud, enabling efficient utilization of hardware resources through the use of statistical multiplexing. Still, there is a significant overhead associated with the use of VMs; each VM instance has to contain a complete OS environment to support the execution of applications that are dependent on the specific services provided by that OS. Ultimately, this has led to the development of alternate, more light-weight approaches to virtualization.

A library OS trades isolation for performance, by allowing applications to execute natively on a host rather than inside a VM. All necessary OS abstractions are provided through user-mode libraries that run as part of the address space of each application. This commonly results in smaller resource footprints and better performance for applications. However, there are a few drawbacks to the library OS approach. First, it is either costly or difficult to enable sharing between multiple processes. Second, application compatibility can only be achieved at a higher level than the application binary interface (ABI), unless applications are modified to exploit alternate interfaces.

The protected library OS (PLOS) is a novel architectural abstraction that is similar to the traditional library OS, but also facilitates hosting of multi-process applications, and uses virtualization technology to target compatibility at the ABI level. It has already been demonstrated as a promising architecture, through the implementation of a PLOS that mimics the Linux 3.2.0 kernel, capable of running complex, unmodified Linux applications like Apache, MySQL, and Hadoop.

This thesis presents Casuar—a new PLOS that targets compatibility with Windows applications. By implementing a subset of the core OS services provided by the Windows NT kernel, we have been able to run Native applications and system DLLs on both Windows and Casuar. We evaluate the performance of Casuar experimentally, by comparing the system to native Windows and Wine through a series of micro-benchmarks. Our results show that Casuar attains near-native performance for a number of system services, and in many cases significantly outperforms Wine.

Acknowledgements

I would like to express my first and foremost thanks to my advisors, Dr. Åge Kvalnes and Dr. Steffen V. Valvåg, for your guidance, invaluable insights, and believing in this project! Also thanks to Robert Pettersen for suggesting the initial idea that led to this madness.

Thanks to my fellow students, especially Kristian Elsebø, Vegard Sandengen, Michael Kampffmeyer, Jan-Ove 'Kuken' Karlberg, Einar Holsbø, Bjørn Fjukstad, and Magnus Stenhaug. Thank you for all your help, for taking part in the obsessions, and for your presence in general. You have all contributed to de-"trasig"-fying all the time spent at the university!

Finally, I would like to thank my family and our dog, Áidna, for expressing their loving support.

Contents

List of Figures

List of Tables

List of Code Listings

List of Code Definitions

List of Abbreviations

ABI application binary interface

ALPC Advanced Local Procedure Call

APC asynchronous procedure call

API application programming interface

APIC Advanced Programmable Interrupt Controller

CPU central processing unit

DLL dynamic-link library

DPC deferred procedure call

FIFO first in first out

HAL hardware abstraction layer

I/O input/output

IPC inter-process communication

IPI inter-processor interrupt

IRP I/O request packet

IRQL interrupt request level

IRR interrupt request register

ISR interrupt service routine

MMU memory management unit

NLS National Language Support

NMI non-maskable interrupt

OKRT omni-kernel runtime

OS operating system

PDB program database

PE Portable Executable

PEB process environment block

PLOS protected library OS

QoS quality-of-service

RDP Remote Desktop Protocol

SEH structured exception handling

SLA service level agreement

SLO service level objective

TCB thread control block

TCP Transmission Control Protocol

TEB thread environment block

TLS thread-local storage

TPR Task Priority Register

VM virtual machine

VMM virtual machine monitor

VMX virtual-machine extensions

VT Virtualization Technology

1 Introduction

Over the past few years, cloud computing [1] has emerged as an increasingly popular paradigm for offering access to computing resources over the Internet [2]. Cloud platforms enable users to deploy both single software applications and large infrastructures through dynamic and on-demand provisioning of virtual appliances. Virtualization technology is intrinsic to cloud computing—encapsulation of workloads in VMs allows for fault isolation, security isolation, and environment isolation between cloud tenants [3], [4], [5]. It also facilitates efficient utilization of hardware resources by using statistical multiplexing [6] for hosting multiple VMs on a single physical machine [2], [4], [7], [8], [9]. This makes it possible for cloud providers to offer cost-effective service models, where resources are metered and customers pay only for what they use [10].

Virtualization is commonly used to host multiple OSs on a single, physical machine, by compartmentalizing each in a separate VM. A *VMM* running on the host provides the VM abstraction, and is responsible for multiplexing the available hardware among a number of isolated VM instances [11], [12]. Traditionally, a VM is manifested as a virtual hardware interface that is functionally equivalent to the actual hardware of the host machine [11], [12], [13], [14]. This form of virtualization, known as *full virtualization*, enables hosting of a stock OS within each VM [11], [14], [7].

The overhead of providing the VM abstraction can be high, especially when a VM runs I/O-intensive tasks [15], [3], [7]. Therefore, it is commonplace for modern VMMs to provide software-based interfaces in place of, or in addition

to, the parts of the hardware interface that are particularly costly to virtualize [16], [14], [7], [5]. This optimization, known as *paravirtualization*, is heavily used by modern, state-of-the-art VMMs, such as Xen [17], Hyper-V [18], and KVM [19]. For instance, it allows a VMM to replace virtualized I/O devices with more efficient, buffer-based software abstractions [14], [5]. These low-cost VMM-provided interfaces can, however, only be exploited by a customized OS.

Besides paravirtualization, advances have also been made in explicit hardware support which contribute to overhead mitigation, in particular for VM workloads that are CPU-intensive [20], [21]. Despite these improvements, there is still a significant difference in attained performance when a task is run natively or in a VM. This performance discrepancy can to some extent be attributed to the VMM having to multiplex hardware resources among VMs without knowledge of the urgency or timeliness of VM tasks [22]. For example, intolerable jitter and processing delays may be the outcome of suboptimal scheduling decisions [5].

Container-based virtualization [23] is a light-weight alternative to traditional VM-based virtualization technology [24]. In a container-based system, the host OS is extended with functionality for partitioning the user space into logically separate *containers*, which isolate applications rather than OSs. Virtualization is performed at the ABI level, which means that containers provide weaker isolation guarantees than VMs, but with the benefit of achieving near-native performance [24]. Containers are, unlike VMs, transparent to the host OS. Also, the resource footprint of a container is significantly smaller than that of a VM, because applications execute directly on top of the host OS.

The popularity of container-based virtualization has increased drastically during the last few years—especially in combination with recent cloud-friendly deployment and orchestration tools, such as Docker [25], rkt [26], and Kubernetes [27]. Container systems have evolved from the *chroot* concept, used in Unix-based OSs to restrict the file system access of an application, to include additional support for isolating other OS resources, such as process trees, network interfaces, and CPU-, memory-, and I/O-consumption [24]. Implementations of container-based virtualization exist on multiple platforms; FreeBSD Jails [28] and Solaris Zones [29] are integrated natively into their respective host OSs, whereas Linux uses implementations that extend the kernel, such as Linux-VServer [23], OpenVZ [30], and LXC [31].

A drawback of container-based virtualization is that the OS can only host applications built for a particular ABI. Recent works [16], [32], [33], [34], [35] have explored ways to tackle this problem, by offloading the implementation of OS abstractions from a host OS or VMM to user-mode libraries, leaving

protection and isolation as host OS responsibilities. This is analogous to the *library OS* concept advocated by earlier work, such as Cache Kernel [36], Exokernel [37], [38], Nemesis [39], and Disco [40].

A library OS decouples an application from the particular interface offered by the host OS. Isolation between applications is achieved by linking each with a separate library OS instance that executes as part of the application's private address space. A weakness of this approach, however, is that it is costly, and in some cases very difficult, to orchestrate and enable sharing of resources across processes [38], [34].

Vortex [9], [7], [5], [8] is a recent, experimental OS that investigates novel approaches to virtualization through a new architectural abstraction—a *PLOS*. Instead of providing virtual hardware interfaces like conventional VMMs, Vortex exposes a paravirtualized software interface comprising high-level commodity OS abstractions, such as files, network connections, memory mappings, processes, and threads. A PLOS molds these abstractions into an ABI that is compatible with the system call interface of an existing OS. It sits on top of a thin virtualization layer, through which it obtains supervisory control over applications. Unlike traditional library OSs, the PLOS abstraction is designed to host multiple processes and facilitate sharing between these. The virtualization layer allows the same PLOS instance to execute in the address space of each process, while retaining strong isolation between applications.

A PLOS is, not unlike containers, completetely transparent to Vortex. All applications running on top of a PLOS are scheduled directly by Vortex at a fine-grained level. However, Vortex provides stronger isolation guarantees than existing container-based systems; resource management is enforced by Vortex at the application-level, allowing two processes in the same PLOS to get different logical views of available resources and their quotas. In addition, the virtualization layer introduces a privilege boundary between processes and their PLOS, equivalent to the separation between user mode and kernel mode in a regular OS.

The PLOS approach has already been proven viable, through the implementation of a PLOS that mimics the Linux 3.2.0 kernel [7], [5], [8], [9]. By supporting a common subset of system calls, this PLOS is capable of running complex, unmodified Linux applications, such as Apache, MySQL, and Hadoop. As a continuation of this work, we have explored the possibilites of implementing a similar PLOS for supporting the execution of Windows applications on top of Vortex. We previously proposed an architecture for a Windows-compatible PLOS [41].

A central goal for our architecture was to enable reuse of functionality within

existing Windows components to a large extent. We thus explored the possibility of targeting compatibility at the system call level, in accordance with the PLOS model. Our findings suggested that this is possible, and that it allows us to rely on existing user-mode DLLs that applications depend on, rather than having to reimplement their functionality. Specifically, we implemented a system call ABI compatible with the calling convention used in Windows on x64 architectures, and a loader component that parses DLLs and unpacks their executable images into the address space of an application running on top of the PLOS. Together, these mechanisms constitute a part of the execution environment that is needed to be able to host existing Windows applications. Continuing our previous work, this thesis focuses on the evaluation and implementation of system services and similar functionality that is exposed directly to the applications, and which is required to support their actual execution.

1.1 Thesis Statement

Drawbridge [34] demonstrated that a library OS could offer a Windows-compatible interface capable of running major applications such as Microsoft Excel, PowerPoint, and Internet Explorer. This work required refactoring and reimplementation of tens of thousands of lines of code in user-mode DLLs to exploit a Drawbridge-defined ABI and to accommodate the limitation that all DLLs had to depend on a single library OS hosted in a single process.

We conjecture that it is possible to improve upon the conventional library OS architecture. Specifically, our thesis is:

> *The protected library OS architecture permits unmodified multi-process Windows applications and user-mode DLLs to run under a Windows library OS.*

1.2 Targeted Applications

We do not believe it is tractable to build a feature-complete PLOS that retains full binary compatibility with Windows, unless essentially creating a full-blown copy of Windows. However, we are convinced that a Windows-based PLOS would be able to support a large number of commonly used Windows applications with significantly less effort. In this thesis, we do not specifically aim to support a predetermined set of existing applications. Instead, our goal is to build a PLOS that meets the most common application requirements and allows applications to execute on both Windows and this PLOS, without modi-

fying binaries or DLLs to introduce dependencies on non-native, PLOS-specific interfaces. Specifically, the PLOS architecture allows us to target compatibility with the existing ABI of Windows. We do this through the implementation of a subset of the ABI, while retaining the semantics of the corresponding functionality in Windows.

The PLOS abstraction, by itself, imposes few or no limitations on what kind of functionality may be implemented and which applications may be supported in a Windows-compatible PLOS. However, the implementation of the PLOS architecture in Vortex inherits some restrictions from the current implementation of Vortex. One such restriction is that Vortex has no graphical support, as Vortex is primarily built for data centers rather than desktop workstations.

Many Windows applications, whether they are desktop applications or not, provide window-based graphical interfaces. Thus, it would be advantageous to support the execution of such applications on a Windows-compatible PLOS. Although the lack of graphical support in Vortex is a restriction, it is by no means a hindrance to achieving this. For example, Drawbridge [34] showed that it is possible to provide interaction with an application's graphical user interface through a Remote Desktop Protocol (RDP) connection.

We have, however, chosen not to include graphical support in the scope of this thesis, and instead focus exclusively on achieving compatibility for some of the core OS services provided by Windows. We target compatibility only with applications that are built for the 64-bit x64[1] architecture, as this is the only platform currently supported by Vortex.

1.3 Methodology

Computer science is one of the youngest science disciplines, being developed over just a little more than 60 years. The commonly accepted definition of computing as a science is "the systematic study of algorithmic processes—their theory, analysis, design, efficiency, implementation, and application—that describe and transform information" [42]. This description was presented in 1989 by the *Task Force on the Core of Computer Science*, formed by ACM and the IEEE Computer Society, as part of their final report, which concluded their effort towards specifying a scientific framework for the fields of computer science and computer engineering. The report also identified three major paradigms that together form the basis for scientific work within the area of computing:

1. Note that we will use the term *x64* throughout this thesis to describe both the Intel x86-64 and the AMD64 platforms collectively, unless otherwise specified.

Theory is rooted in mathematics. Mathematical objects and their relationships are studied, and hypotheses are formed to describe their behavior. These hypotheses are subsequently proven or falsified to develop coherent, valid theories that can be interpreted and applied within the other paradigms.

Abstraction is rooted in the experimental scientific method. The primary focus is on the investigation of phenomenons. Hypotheses are used to construct models and form predictions that are tested experimentally.

Design is rooted in engineering. Requirements and specifications are identified, and theory and abstraction is applied to design, implement, and test systems that perform useful actions.

This thesis is rooted in the area of *systems research*, which to some degree belongs to all three paradigms. First, we use existing knowledge about Windows to devise a number of requirements for our system, and design components that can fulfil the requirements, aided by theory and abstraction. This is the focus of Chapter 3 and Chapter 4. Then, in Chapter 5 we use abstraction to formulate a methodology for investigating the behavior of Windows applications that is not already known to us. By following an iterative process, and applying a number of techniques that we create using design and theory, we successively gain more knowledge about Windows; we use the process to discover new requirements, refine existing ones, and implement functionality that satisfy these. Finally, through testing and experiments, we demonstrate the capabilities of the system and evaluate its usefulness.

1.4 Summary of Contributions

This thesis makes the following contributions:

- We strengthen the viability of the PLOS architecture as an improvement over the traditional library OS through the implementation of Casuar—a PLOS for running Windows applications on top of Vortex.

- We evaluate the most fundamental synchronization and signaling abstractions in Windows—interrupt request levels (IRQLs), asynchronous procedure calls (APCs), and blocking synchronization—which are prerequisites to supporting the execution of any Windows application. We also give a detailed description of how these are implemented in Casuar on top of the paravirtualized software interface of Vortex.

- We evaluate higher-level subsystems in the Windows NT kernel for man-

agement of executive objects, memory, and I/O. Then, we describe how the most commonly used system services can be supported through selective implementation of just a small subset of the functionality implemented in Windows.

- We present the architecture and implementation of a memory monitor that can be used to trace memory accesses from user mode, and describe how we can use this information to infer application dependencies on undocumented data structure fields that are part of the Windows ABI.

- We describe a mechanism for producing stack traces, which we use to provide necessary context for implementing missing functionality that is exposed through undocumented parts of the Windows ABI.

- We demonstrate that our Casuar implementation is capable of hosting a special type of Windows applications, known as Native applications.

- We experimentally evaluate Casuar through a number of micro-benchmarks that demonstrate low overhead for several implemented system services.

1.5 Outline

The remainder of the thesis is structured as follows.

Chapter 2 presents the existing architectures of Windows NT and Vortex, including details about how Vortex implements the PLOS abstraction, before outlining the architecture of Casuar. The chapter also presents related work.

Chapter 3 describes the most essential synchronization and signaling mechanisms in Windows, which are used extensively as part of implementing higher-level system services. Throughout the chapter, each mechanism is evaluated, and we provide a detailed description of how the corresponding functionality is implemented in Casuar.

Chapter 4 gives an overview of some of the largest and most important high-level components in the Windows NT kernel, and how Casuar replicates a subset of their implementations to provide the necessary services to hosted Windows applications.

Chapter 5 presents a methodology that we use to discover and implement

application dependencies on undocumented parts of the Windows ABI. The chapter describes some central challenges to achieving this, and presents two techniques that help us tackle them—a memory monitor for tracing memory accesses from user mode, and a mechanism for producing stack traces that provide context about missing functionality. At the end of the chapter, we demonstrate that our Casuar implementation, through the use of these techniques, is able to host Native applications.

Chapter 6 evaluates Casuar, by comparing the system to Windows and Wine through a series of micro-benchmarks.

Chapter 7 concludes the thesis and outlines future work.

2 Architecture

In this chapter, we present the architecture of Casuar as a protected library OS (PLOS) for running Windows applications on top of Vortex. First, we give an overview of Windows NT, its main architectural components, and the user-mode subsystems that define the interfaces between Windows and its applications. Next, we describe Vortex and the omni-kernel architecture that Vortex implements. We explain the implementation of the virtualization environment that Vortex provides to a PLOS. Then, we detail Casuar's architecture, and how it aims to target compatibility with Windows applications through extensive reuse of existing user-mode libraries. Finally, we present related work.

2.1 Windows NT

Microsoft has developed and commercially released operating systems under the Windows brand since the early 1980s. Today's incarnations of Windows belong to the Windows NT family, which was established when Windows NT 3.1 hit the market in July 1993 [43]. The architecture of Windows NT can even be traced back as far as the 1970s, with its design originating from the development of VAX/VMS from Digital Equipment Corporation [44]. Although Windows NT is no longer used as a commercial product name, starting with the release of Windows 2000, the Windows kernel is still developed under an internal NT version number.

The NT kernel version has traditionally been incremented for every new, major release of Windows. For example, the latest versions of Windows—*Windows 8.1* and *Windows Server 2012 R2*—are instances of Windows NT 6.3. The same NT kernel also powers Microsoft's mobile platform, *Windows Phone 8.1*. Similarly, the previous version, NT 6.2, is currently the basis for the *Xbox One* entertainment system [45], as well as for *Windows 8* and *Windows Server 2012*. However, with the upcoming release of Windows 10, Microsoft has decided to change the NT kernel version to 10.0 instead of 6.4, to let the Windows product version and NT kernel build version stay in sync.[1]

Figure 2.1 gives an overview of the NT architecture and its various parts. As shown, the Windows environment can be divided into two layers: *kernel mode* and *user mode*. The kernel-mode layer encompasses all core OS functionality that requires privileged access to system resources such as the CPU, physical memory, and I/O devices. Its main components are the Windows Kernel and the Windows Executive—both of which are contained in the `ntoskrnl.exe` system executable file. In contrast, the user-mode layer contains all applications, which run as *processes* in a non-privileged processor execution mode. There is a strict separation between user-mode applications and the OS; processes have limited access to hardware, and may only interact with the system resources indirectly through a system call interface that is managed by the kernel-mode layer. In addition, each process is given a separate, private addess space, to isolate processes from each other.

The *Windows Executive* corresponds to the upper part of the kernel-mode layer. It consists of a number of components or subsystems—such as the *Memory Manager*, the *I/O Manager*, and the *Process Manager*—that manage different parts of the system. These executive components provide abstractions over most of the system's resources, and make them available to user-mode applications and device drivers via corresponding system services.

All executive services are built on top of the *Windows Kernel*. It implements a set of low-level OS functionality that, to a large extent, interfaces directly with the underlying hardware platform. This includes mechanisms for traps and system calls, context switching and scheduling of threads, dispatching of interrupts and exceptions, and multiprocessor synchronization services and primitives. Kernel services are managed through a collection of *kernel objects* and a number of basic functions that operate on these. The Windows Executive encapsulates the kernel objects in more complex *executive objects* and uses these to extend the functionality of the kernel to provide higher-level system services.

1. A few, early preview builds of Windows 10 did in fact use 6.4 as the kernel build version, before it was changed to 10.0.

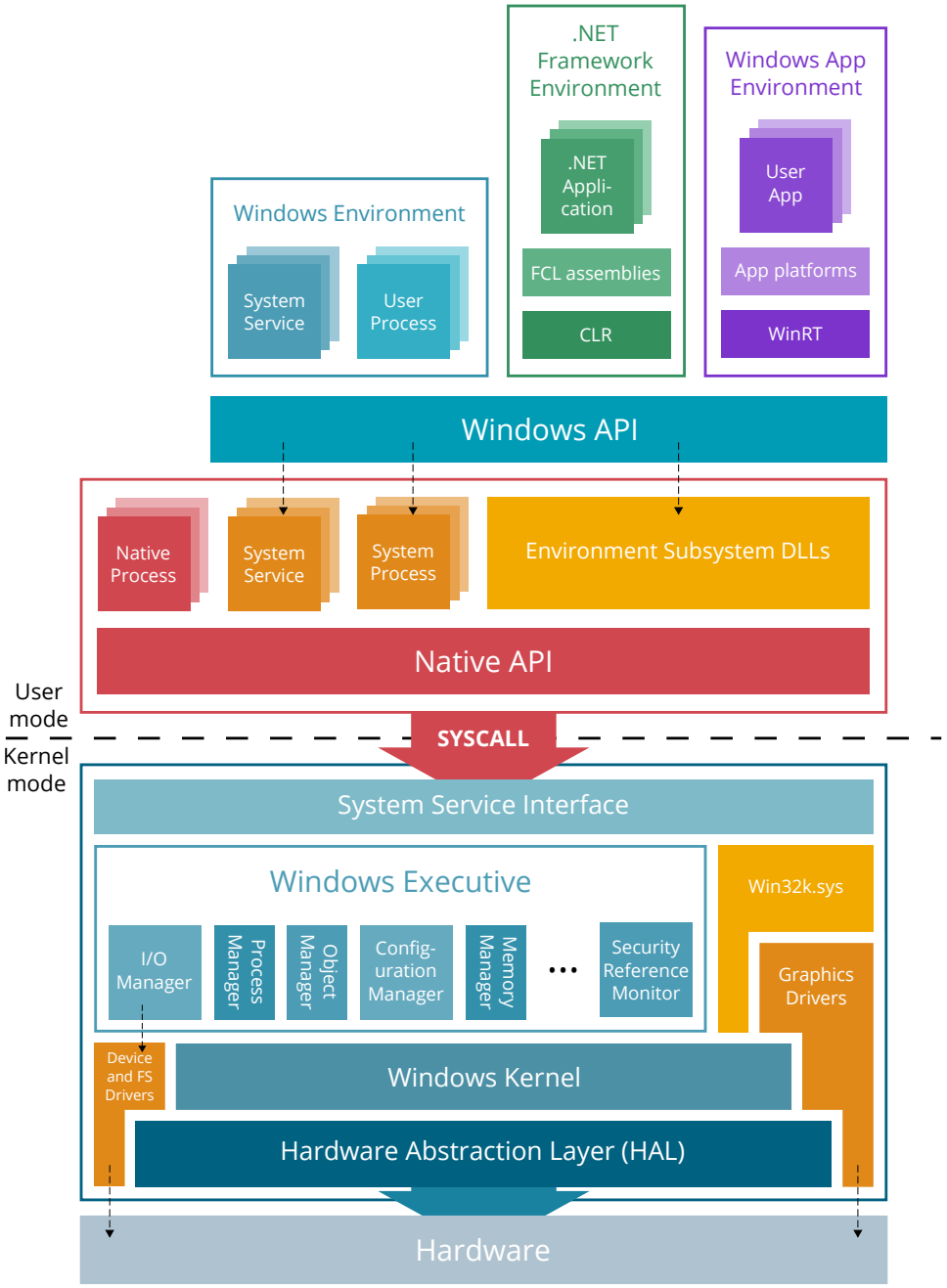


Figure 2.1: An overview of the layered architecture of Windows NT. Illustration is derived from [46, Ch. 2].

Apart from the Windows Kernel and the Windows Executive, the kernel-mode layer also contains drivers for I/O devices, file system, network, graphics, and similar, as these typically need direct access to hardware or system resources. This includes both native and third-party drivers, which Windows allows to be dynamically installed and loaded. Finally, a hardware abstraction layer (HAL) constitutes the lowest-level part of the kernel-mode layer. It is a kernel-mode module—loaded from `hal.dll` by the Windows Kernel—that is designed to hide machine-dependent differences in the underlying hardware platform. On x64, the HAL is for example used to allocate interrupt vectors on behalf of device drivers, and it provides a portable interface for requesting software interrupts on different CPUs.

In Windows, a large number of system components also reside in user mode. These include system support processes, which perform necessary initialization and management of the system, and native Windows services, which are responsible for parts of the functionality that is available to a Windows application. The system call interface provided by the Windows Executive is not used directly by user-mode applications, because it is undocumented, and Microsoft reserves the right to make changes to it between different versions of Windows. Instead, Windows defines different *environment subsystems* that are implemented in user mode and offer a broader, more convenient API to applications. The *Windows Subsystem* provides the *Windows API*, which is the primary interface used by almost all Windows applications. The Windows API is fully documented, and behaves to a large degree consistently across different Windows versions. As is shown in Figure 2.1, all .NET applications and Windows apps also run indirectly on top of the Windows API.

The Windows API is exported by a large number of DLLs, such as `kernel32.dll`, `user32.dll`, and `gdi32.dll`. These, in turn, are implemented on top of the *Native API*—the lowest-level API available to applications and services in user mode [46, Ch. 1–2], [47]. Similarly to the system call interface, the Native API is undocumented and subject to change between NT releases. It also constitutes the *Native subsystem* in which *Native applications* run. Examples of Native applications are system support processes such as the *Windows Subsystem process* (`csrss.exe`) and *Windows Session Manager process* (`smss.exe`), which implement parts of the Windows Subsystem and cannot therefore be Windows applications themselves. Almost all Native applications are developed internally by Microsoft as part of Windows.

The Native API consists mainly of two parts: a set of system call stubs for invoking Windows Executive system services, and a set of run-time library functions that provide more convenient interfaces to Native applications and Windows Subsystem DLLs. In contract to the Windows API, which is implemented by several DLLs, support processes, and services, the Native API is

provided almost entirely by a single DLL—`ntdll.dll`. This DLL is also special, because it contains the function that is used as entry point for every process in Windows—regardless of its subsystem—in addition to some other functions that can be called by the Windows Kernel. For these reasons, `ntdll.dll` is loaded as part of the address space of all running processes.

2.2 The Vortex Omni-Kernel

As pointed out in Chapter 1, clouds commonly benefit from the many strong isolation properties of virtualization and its opportunities for statistical multiplexing. However, modern VM technology does not provide sufficient isolation between VMs that are consolidated on the same physcial host [3]. This means that the resource consumption of a workload may affect the performance of co-located workloads [4], due to contention on shared resources—a concept referred to as *performance interference* [3], [9].

Cloud providers commonly have to meet a number of requirements for the services offered to tenants. Such requirements are typically governed by service level agreements (SLAs), in which non-functional aspects are expressed as a number of service level objectives (SLOs)—each corresponding to a measurable characteristic that is often defined in terms of available resources [2]. Providing performance guarantees in a virtualized environment that is subject to performance interference is, however, non-trivial. Lack of rigorous control over resource allocation may result in SLO violations. In addition, implicit sharing of certain hidden, physical resources that are not easily virtualizable, such as caches and buses, can cause interference that may affect the performance of other resources in the system [9]. The result may be that SLOs—even when these are retained—no longer adequately express quality-of-service (QoS), as opposed to in an isolated, non-virtualized system, where SLO guarantees will typically always coincide with perceived QoS [2].

Possible ways to deal with performance interference include employing strict partitioning of existing resources between VM instances, or to over-provision by reserving additional resources for on-demand repurposing [2]. However, either of these approaches comes at the cost of less efficient utilization of available hardware.

The *omni-kernel architecture* [9] was designed with the premise of employing *pervasive monitoring and scheduling* to ensure complete control over all resource allocation. It is built on two fundamental abstractions—resources and schedulers. *Resources* are software components that provide fine-grained control over hardware or software functionality and expose interfaces for the

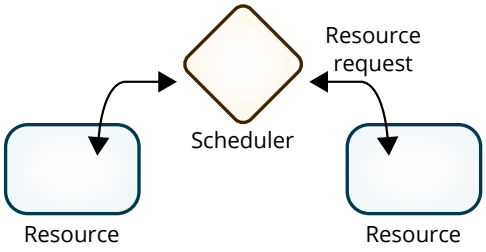


Figure 2.2: Schedulers control the message-passing between resources in the omni-kernel architecture. Illustration is derived from [9].

use of this functionality. A resource can depend on the functionality provided by other resources, and uses asynchronous message passing to send requests to these. The resources are organized in a *resource grid* according to their dependencies, where *schedulers* are interpositioned between every pair of communicating resources, as illustrated in Figure 2.2. The schedulers are responsible for dispatching and ordering request messages that are passed between resources. They process information about resource usage, which is measured extensively throughout the system, and use it to make scheduling decisions that, for instance, are in accordance with predetermined SLOs.

Vortex [9], [7], [5], [8] is an omni-kernel implementation for Intel x64 architectures. The Vortex omni-kernel is structured as a monolithic kernel with a layered design, as detailed in Figure 2.3. Most of its functionality is implemented as resources in the resource grid layer; for example, the *CPU resource* is used to allocate CPU-time, the *memory resource* manages allocation of physcial memory, and the *process resource* and *thread resource* implement commodity process and thread abstractions. Device drivers are also implemented as specialized resources that interface with hardware.

A resource in the resource grid can export interfaces to the Vortex system service interface, in order to make its functionality available to applications. The system service interface is the highest layer of the Vortex omni-kernel, and consists of all functions that are made available to processes through the system call ABI. The resource grid is implemented on top of the *omni-kernel runtime (OKRT)*—a framework that manages resources and schedulers, and provides the mechanisms for message-passing between these. At the lowest level, Vortex implements a *OKRT hardware abstraction layer (HAL)* that, similarly to the Windows NT HAL described in Section 2.1, is used to hide platform-specific details from the OKRT and resource grid.

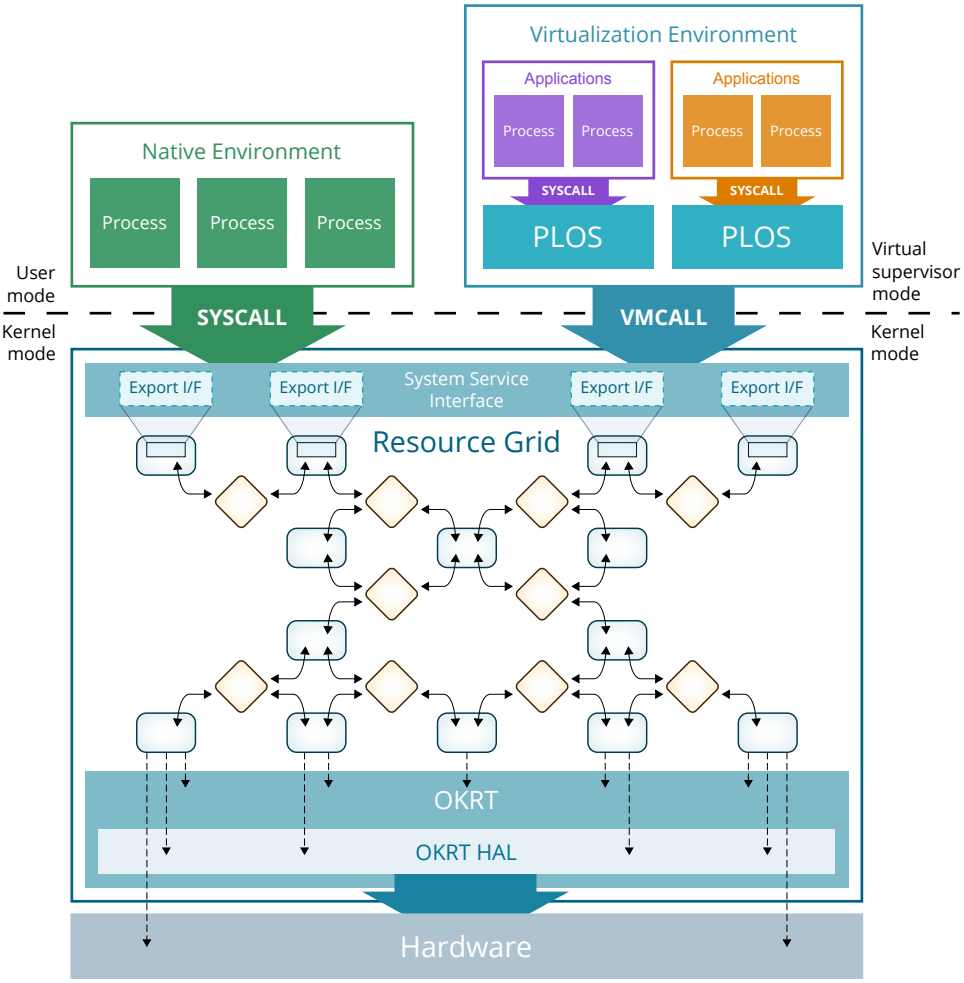


Figure 2.3: An overview of the layered architecture of Vortex.

2.2.1 Protected Library Operating Systems

The *protected library OS (PLOS)* abstraction outlined in Chapter 1 is implemented in Vortex by exploiting hardware support for virtualization; Vortex uses the virtual-machine extensions (VMX) extensions that are part of the Intel Virtualization Technology (VT) to create a virtualization environment in which each PLOS and its applications runs. The virtualization environment introduces an extra privilege level that separates a PLOS from its hosted applications. The PLOS is allowed to execute with virtual supervisor rights on a virtual CPU, whereas applications that are hosted by the PLOS run in a virtual user mode. As a result, system calls from the applications will trap directly to the PLOS, thus allowing the PLOS to target compatibility at the ABI level. At the same time, the system service interface of Vortex is made available to the PLOS through a

vmcall ABI, providing the same functionality that is exposed to native Vortex applications, but with slight differences.

A PLOS behaves like a regular OS kernel from the perspective of its hosted applications. Vortex splices the memory region containing the PLOS executable image and data structures into the address space of every child process started by the PLOS. This ensures that all applications running on top of a PLOS will trap into the same PLOS instance, and lets the PLOS facilitate sharing of state between applications.

Vortex implements its virtualization environment using the same virtual CPU abstraction that is used by conventional VMMs to provide the VM abstraction. However, the virtualization environment in Vortex differs significantly from a VM. A VMM exposes a fixed number of virtual CPUs to the VM, which are used by a contained OS to schedule threads internally. Because the OS implements its own thread abstraction and scheduler, the VMM has little or no insight into what type of tasks are executed inside the VM. This means that the VMM loses opportunities for making optimal and fine-grained scheduling decisions, which in turn might hurt I/O performance. A PLOS, on the other hand, does not implement its own thread abstraction; rather, it relies on the high-level abstractions that are already provided by Vortex. Vortex virtualizes each thread separately, by providing each with a separate virtual CPU, and thereby retains full control over scheduling of all threads in the PLOS and its applications.

The virtualization environment allows a PLOS to differentiate access rights to memory mappings, in order to protect pages from being accessed by applications executing in virtualized user mode. This is achieved through the Vortex system call `vx_mmap()`, which lets the PLOS specify the privilege level of each memory region. Vortex does not maintain a separate set of shadow page tables for each PLOS, as is done for a VM in a conventional VMM. Instead, all memory mappings are allocated in the ordinary page tables to reduce overhead.

To prevent a PLOS from accessing the Vortex kernel, Vortex exposes a copy of the top-level page directory to the virtualization environment that does not contain mappings for the Vortex kernel's page tables. Whenever the PLOS or one of its applications needs to allocate a new page table with an entry in the page directory, the entry is mirrored in the virtual page directory. The active page directory pointer of the CPU is automatically changed from the virtual to the real page directory every time the PLOS traps to the Vortex kernel, and is changed back upon leaving the kernel.

2.3 Casuar

As stated in Chapter 1, Casuar is a continuation of previous work where we did initial exploration of the possibilities for creating a Windows-compatible PLOS. The architecture of Casuar is therefore the same as proposed earlier in [41], of which we provide an overview here.

Recall that one of the main goals of Casuar's architecture is to facilitate extensive reuse of functionality that is already available through existing user-mode DLLs. The Windows API that is implemented by such DLLs comprises more than 100,000 callable functions [34], and would require a significant effort to re-implement [41]. In contrast, there are only 433 non-graphics related system calls in Windows NT 6.3 [41], and most of the functionality in the Windows APIs is built on top of an even smaller subset of these.

By targeting application compatibility through the system call interface, we believe it is possible to support the execution of Windows applications with less effort than would be required to achieve compatibility at the Windows API level. Figure 2.4 illustrates the architecture of Casuar, adhering to this approach. Casuar will effectively replace the entire NT kernel (`ntoskrnl.exe`), and provide alternate implementations for abstractions normally provided by the Windows Kernel and Windows Executive. Although we have not found it feasible to reuse any functionality from the NT kernel, we recognize the separation of concerns between the low-level Windows Kernel and the higher-level Windows Executive, and use the same separation when we implement the equivalent functionality in Casuar. In Chapter 3, we describe the implementation of low-level synchronization and signaling mechanisms in Casuar that correspond to abstractions from the Windows Kernel. We describe the implementation of higher-level executive services in Chapter 4.

There is a potential drawback to targeting system call compatibility with Windows. As may be recalled from Section 2.1, the system calls are a subset of the undocumented Native API. It is therefore not straightforward to determine what functionality is expected by user-mode applications through the system call interface. In addition, there is a risk that new versions of Windows may introduce changes to the Native API. The latter limitation could be circumvented by targeting only specific versions of Windows. Moreover, it seems that drastic changes to the system call ABI are not frequent, although Microsoft reserves the right to perform such changes. In Chapter 5, we detail how we approach the former challenge to reach an implementation that is capable of hosting Native Windows applications.

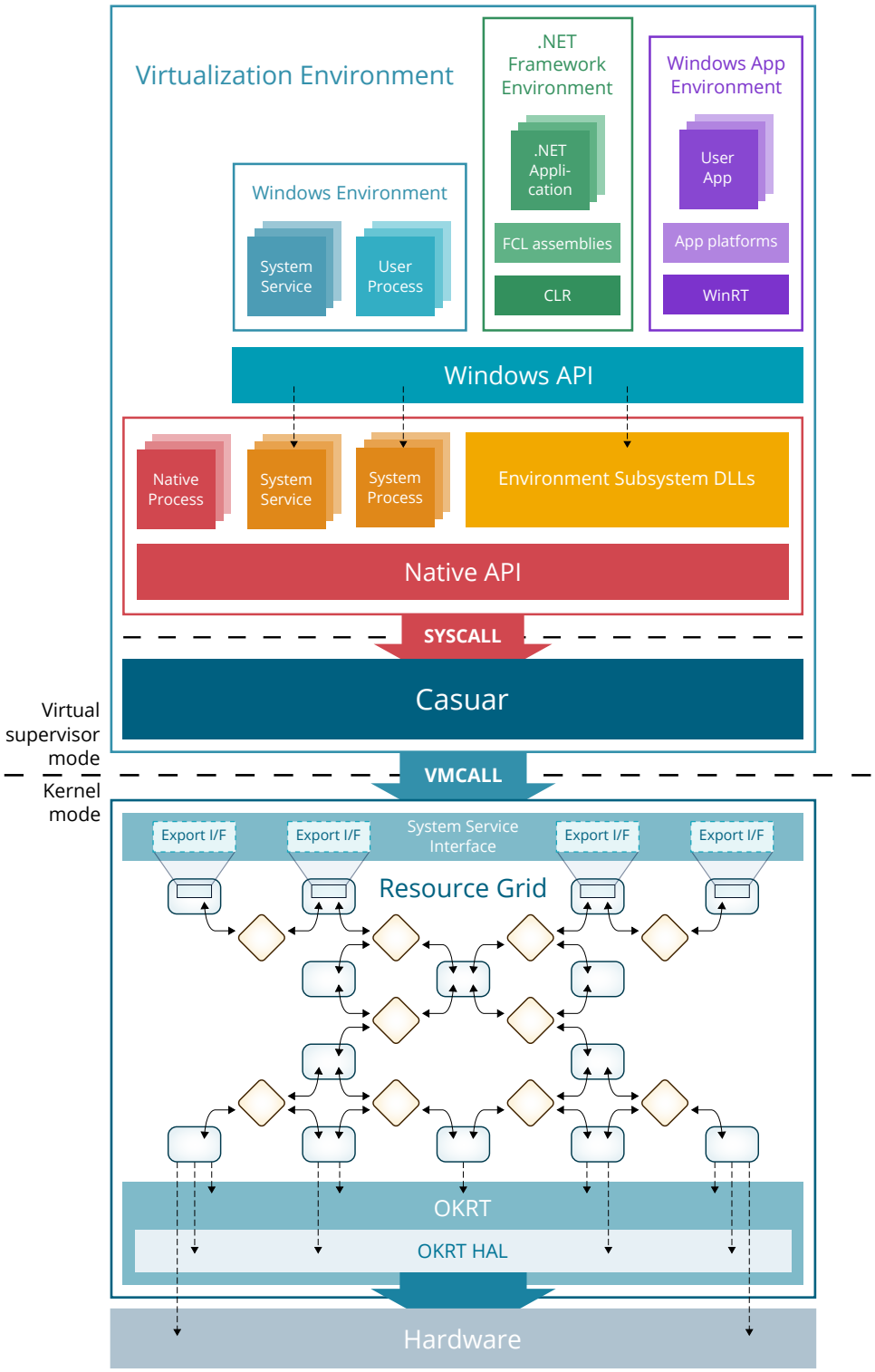


Figure 2.4: Architecture of Casuar as a protected library OS.

2.4 Related Work

While there exist several systems that enable cross-platform application compatibility, there are very few that are built for running Windows applications on platforms different from Windows. The open-source Wine project [48] is one of the largest and perhaps the most well-known of such efforts. It allows Windows applications to execute on POSIX compatible OSs, such as Linux. Wine is currently able to run more than 10,000 Windows applications—including Word and Excel from the Microsoft Office suite and a large number of complex 3D games—and has partial support for at least another 10,000 applications.

Wine targets binary compatibility mainly at the Windows API level, but also implements a portion of the Native API. This is done by replacing several system DLLs—such as `ntdll.dll`, `kernel32.dll`, and `user32.dll`—with alternate implementations that effectively emulate the Windows application environment on top of the native POSIX API. All parts of Wine are implemented in user mode; a separate Wine server process facilitates synchronization across processes through inter-process communication (IPC) [49]. In this regard, Wine is somewhat similar to a traditional library OS. The Wine server can be thought of as a means to enable sharing between multiple processes. However, relying on a separate process to orchestrate this might hurt performance for certain workloads; applications that depend on the server process will have to wait for it to be scheduled by the host OS, and there is inevitably some overhead associated with the communication between processes.

ReactOS [50] is another open-source project that aims towards compatibility with existing Windows applications. It is a stand-alone OS that re-implements most kernel-mode and user-mode components of Windows. The implementation follows the architecture of Windows NT closely, and is to a large extent based on reverse engineering of actual functionality in Windows. Although the project has been around for more than 15 years [51], it is still in the alpha stage, and only fully supports a small number of applications.

Drawbridge [34] is a research prototype that refactors Windows 7 into a library OS. By evaluating Windows' system service interface, the authors found that it is possible to re-implement most of the system calls in user mode, on top of a much smaller kernel-mode ABI. This was done while retaining enough functionality to be able to run major desktop applications, such as Microsoft Excel, PowerPoint, and Internet Explorer. Security isolation is achieved by running each application on a different instance of Drawbridge. A security monitor is interpositioned between Drawbridge and the host OS to enforce different logical views of the system resources, such as the file system and Windows registry, for each application. Support for graphics and input from human interface devices (i.e. keyboard and mouse) is provided through Remote

Desktop Protocol (RDP) connections. Each Drawbridge instance gets a separate RDP session that the end-user can connect to from an RDP client on the host OS.

The Drawbridge system was evaluated by comparing it to running applications on Windows—both natively, and in Hyper-V VMs. The memory overhead and start-up time of a Drawbridge application was only slightly higher than that of native applications. In contrast, both the memory footprint and boot-time of a VM-contained application was in several cases shown to be an order of magnitude larger. These findings emphasize the advantages of a library OS architecture. However, Drawbridge is also subject to the traditional limitations of library OSs; existing Windows DLLs had to be reimplemented in order to emulate the NT system call ABI in user mode, and the system is unable to host and facilitate sharing in multi-process applications.

3

Low-level Synchronization and Signaling Mechanisms

In this chapter, we evaluate some of the fundamental synchronization and signaling mechanisms that the Windows Kernel provides, and which is used by the Windows Executive to implement higher-level abstractions. For each such mechanism, we also describe how we implement a corresponding abstraction in Casuar, based on the paravirtualized software interface provided by Vortex. First, we detail an interrupt prioritization scheme, known as interrupt request levels (IRQLs), that is used extensively by the Windows Kernel. We explain how software interrupts are used as the delivery mechanism for another abstraction—asynchronous procedure calls (APCs)—and how Casuar implements functionality for emulating interrupts. Next, we describe APCs, what they are used for in Windows, and why we require an equivalent abstraction in Casuar. Then, we provide an overview of how the Windows Kernel implements primitives for blocking synchronization, and describe Casuar's approach to offering corresponding blocking services. Finally, we show how the functionality for suspending and resuming a thread is implemented, through the combined use of blocking primitives and APCs.

3.1 Interrupt Request Levels (IRQLs) and Software Interrupts

Windows employs a prioritization scheme for interrupts called *interrupt request levels (IRQLs)* [46, Ch. 3], [52]. Each logical processor has a *current IRQL* attribute—a number that determines the priority of the task currently executing on the processor at any point of time, where a greater number indicates a higher priority. Every interrupt vector allocated in the Windows Kernel is assigned an IRQL that reflects its relative importance. When an interrupt is dispatched to a processor, the processor's IRQL is automatically raised to that of the interrupt vector, before the associated interrupt service routine (ISR) is executed. Similarly, the processor's IRQL is automatically lowered back to its previous level when the ISR returns. The current IRQL may also be raised or lowered explicitly, using the functions `KeRaiseIrql()` and `KeLowerIrql()` of the Windows Kernel.

Raising the IRQL to a given level *n* will temporarily disable or *mask* all interrupts with priority less than or equal to *n*. Conversely, the processor may be interrupted at any time to process an interrupt at a higher priority than the current IRQL [46, Ch. 3]. Interrupts that cannot be delivered right away, because the processor is already executing a task at an IRQL greater than or equal to that of the interrupt, will be registered as *pending*. Once the processor's IRQL is lowered, all pending interrupts with IRQLs higher than the new, current IRQL will be dispatched in priority order from highest to lowest, and the processor's current IRQL will immediately be raised to the IRQL of the highest-priority pending interrupt to be serviced. Figure 3.1 illustrates an example of how the IRQL changes when interrupts are being serviced by the processor.

On x64, there are 16 different IRQLs numbered 0–15 [46, Ch. 3], as shown in Figure 3.2. Each IRQL is assigned a specific meaning or purpose by the Windows Kernel. The lowest IRQL, known as PASSIVE LEVEL, is the default priority level, where all interrupts are enabled. All user-mode code and most kernel-mode code runs at PASSIVE LEVEL. It is also one of just a few IRQLs that is not associated with any interrupt vectors.[1]

IRQLs 3–11 and 13–15 are used for hardware interrupts. The first of these IRQL ranges, known collectively as DIRQL, is used for generic device interrupts, and the second range is used for special system interrupts, such as the periodic

1. As of Windows 8, it is possible for drivers to register ISRs that will run at PASSIVE LEVEL in response to an interrupt [53], [54]. Note, however, that the actual interrupt vector from which such an interrupt originates will always be associated with a DIRQL, and the passive-level ISR is run in the context of a special system worker thread through a deferring mechanism.

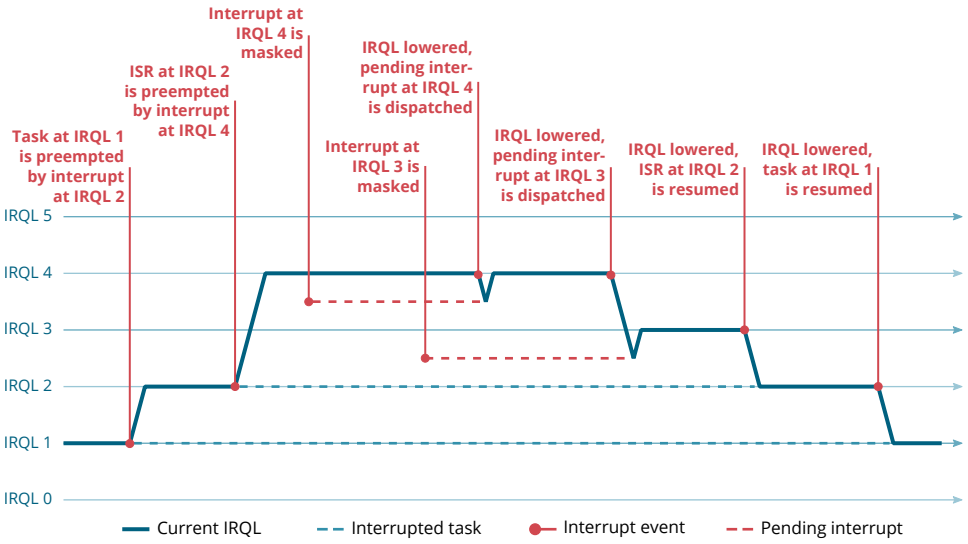


Figure 3.1: Example of how a processor's IRQL may change in the face of interrupts. The figure illustrates a timeline of interrupt events from left to right. In the example, the processor is initially running at IRQL 1, and the system receives interrupts at IRQLs 2, 3, and 4 at various points of time. Interrupts at IRQLs above the processor's current IRQL are dispatched right away, whereas interrupts at lesser or equal IRQLs are masked. When an interrupt is masked, it will be pending delivery until the current IRQL is lowered below the IRQL of that interrupt. As soon as an ISR completes, the current IRQL is lowered back to the IRQL at which the processor was running before it was interrupted to execute the ISR. However, if there are pending interrupts at IRQLs above the new, lesser target IRQL, the processor's current IRQL will immediately be raised again to the IRQL of the highest-priority pending interrupt, before it even reaches the target IRQL.

system clock interrupt, a cache coherency inter-processor interrupt (IPI), and non-maskable interrupts (NMIs). IRQL 12 (SYNCH LEVEL) is used to synchronize access to some shared resources across differerent processors.

Finally, IRQLs 1 and 2 are used for software interrupts. Windows has two kinds of software interrupts—*deferred procedure calls (DPCs)*, which run at DISPATCH LEVEL (IRQL 2), and *asynchronous procedure calls (APCs)*, which are dispatched at APC LEVEL (IRQL 1). Both DPCs and APCs are very important mechanisms in the Windows Kernel.

DPCs are software interrupts that are targeted to run on a *specific processor*. They are typically used by ISRs running at a higher IRQL to defer processor-specific work to a lower IRQL, so that ISRs can execute as quickly as possible [55]. The thread scheduler in Windows—commonly referred to as the *dispatcher*—also runs at DISPATCH LEVEL [52]. It is either invoked directly by

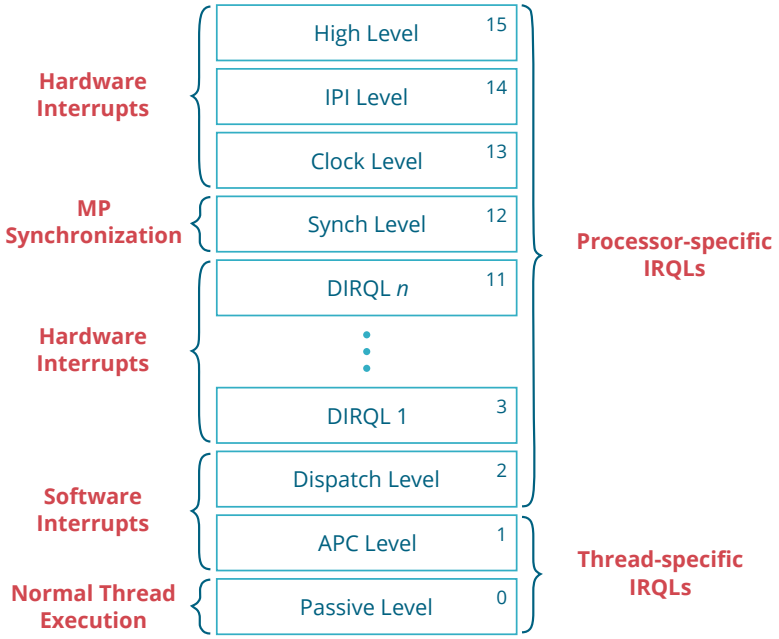


Figure 3.2: IRQLs used in Windows on x64. Note: this figure is based on figures from [46, Ch. 3].

a thread calling a dispatcher function (for example if the thread yields or enters a wait state), in which case the IRQL will be raised manually using `KeRaiseIrql()`, or it is invoked in the context of a DPC.

When a thread is being scheduled to run on the processor, it is given a time slice or *quantum* that limits the amount of CPU time it gets before it may be preempted to allow another thread to run [46, Ch. 5]. The preemption mechanism is driven by the system clock interrupt at CLOCK LEVEL, which will post a DPC that invokes the dispatcher once the IRQL is lowered below DISPATCH LEVEL, after the clock ISR completes [56], [46, Ch. 3].

Because the dispatcher runs at DISPATCH LEVEL, threads are scheduled to run at IRQLs below DISPATCH LEVEL. If a thread raises the current IRQL to DISPATCH LEVEL, it will, in effect, temporarily disable preemption,[2] since the dispatcher will not be able to run [52]. However, if a thread raises the IRQL to APC LEVEL, it may still be preempted by the dispatcher to run another thread at either PASSIVE LEVEL or APC LEVEL. Then, when the preempted thread is scheduled to run at a later point of time, it will resume its execution at APC LEVEL. This means that an IRQL below DISPATCH LEVEL is considered an attribute of the currently running *thread* instead of an attribute of the *processor* that hosts the thread, and there is a logical separation betwen *processor-specific* or *high* IRQLs—the levels above and equal to DISPATCH LEVEL—and *thread-*

specific or *low* IRQLs—the ones below DISPATCH LEVEL.

It follows that APCs—the other kind of software interrupts in the Windows Kernel—are interrupts that are targeted to run in the context of a *specific thread*, in contrast to DPCs, which were targeted at a *specific processor*. Although the APC LEVEL interrupt vector will be associated with a particular processor, the dispatcher will make sure that an APC is delivered to a specified target thread, and not just to whichever thread is currently running on the processor at the time when the interrupt is received. APCs are most commonly used in Windows to perform I/O completion tasks that must run in the context of the same thread that initiated an I/O operation [52]. The APC abstraction is, as opposed to DPCs, also exposed to user-mode code. For example, the `QueueUserApc()` Windows API call [58], and its underlying `NtQueueApcThread()` Native API system call, allows a thread to post an APC to another thread. Other examples are the `ReadFileEx()` and `WriteFileEx()` functions, which are used to initiate asynchronous read and write operations on a file, respectively, that take as argument a completion callback that will be run in the context of an APC [52]. APCs will be discussed further in Section 3.2.

The nature of IRQLs in Windows imposes certain restrictions on the programming model of kernel-mode components and drivers. One such restriction is that only code running at thread-specific IRQLs are allowed to initiate blocking operations. This is because interrupts served at processor-specific IRQLs will be executed in the context of an arbitrary thread that has only been temporarily interrupted, and which should be able to continue running as soon as the interrupting ISR completes [46, Ch. 3]. If an ISR at or above DISPATCH LEVEL were allowed to block, it would effectively be blocking the thread that was currently running on the processor. As a consequence, spinlocks is the only synchronization mechanism that is allowed to be used at processor-specific IRQLs [57].

If two or more ISRs need to synchronize through a spinlock, they all need to be at the same IRQL. Moreover, this IRQL must be at least as high as the highest IRQL at which the spinlock may be acquired anywhere in the system [57]. If any of these requirements are violated, deadlock may occur.[3] There are also many kernel functions that may be called only at certain IRQLs. All

2. In Windows, this is typically done when a thread acquires a spinlock in the kernel, to make sure that the thread is not scheduled out while other threads may be spinning in a busy-wait loop on another processor, waiting to acquire the spinlock [57].
3. For instance, if an ISR executing at IRQL n acquires a spinlock s, and is afterwards interrupted to run an ISR at IRQL $n+1$ that also attempts to acquire that same spinlock s, then the processor will deadlock. The interrupted ISR cannot continue until the interrupting ISR has completed, and the spinlock will thus never be released, allowing the interrupting ISR to complete.

kernel functions that are available to drivers, for which this restriction applies, will have specified in their documentation the exact IRQLs that are allowed. Finally, a function in the kernel is typically allowed to raise the current IRQL temporarily, but it is generally not allowed to lower the IRQL below the level at which it was invoked.

3.1.1 Emulating Software Interrupts in Casuar

Because the APC abstraction is exposed to user-mode, we have deemed it likely that an equivalent mechanism is needed in Casuar to support the system calls that make use of APCs. By the same reasoning, it seems unlikely that a DPC mechanism will be needed, since—as far as we have observed—there are no system calls that expose or depend on this abstraction, either directly or indirectly. Furthermore, Vortex does not expose the abstraction of a virtualized CPU in the same way as a conventional VMM. Instead, Vortex virtualizes each thread separately to retain control over how threads are scheduled. This means that Casuar will not deal with virtual interrupts or implement an internal thread dispatcher, which would be reasons for wanting DPCs in the first place.

To support APCs, we need a mechanism similar to CPU interrupts that allows preempting the execution of a thread to have it perform some other work, and afterwards returning it to the point of interruption. Despite the restrictions it imposes on our programming model, we also need the abstraction of thread-specific IRQLs, so a thread may prevent the delivery of APCs at certain times. Finally, we must be able to queue up pending interrupts that have been masked by the current IRQL, and defer the delivery of these until the IRQL is lowered.

On x64, IRQLs are implemented with hardware support from the local Advanced Programmable Interrupt Controller (APIC) of each processor. A processor's current IRQL is equivalent to the value of the local APIC's Task Priority Register (TPR),[4] which is mirrored in the processor's `CR8` control register for fast access. The local APIC also handles queuing and delivery of pending interrupts [59, Ch. 10.8]. When the APIC accepts an interrupt that has been masked, it is registered by setting a bit corresponding to the interrupt vector in the interrupt request register (IRR) of the APIC. Later, when the pending interrupt is delivered, the corresponding bit is cleared in the IRR.

As we cannot rely on the functionality of a virtualized APIC, we have chosen to emulate the TPR and IRR of the local APIC in software. This is done on a

4. Further details about the APIC subsystem and the TPR may be found in [60, Ch. 15] and [59, Ch. 10.8].

per-thread basis, since we only need to represent thread-specific IRQLs. Each thread maintains a *current IRQL* variable and a *pending IRQL interrupt* bitmask in its kernel-level thread control block (TCB). Raising and lowering the thread's current IRQL is done explicitly through calls to the functions `irql_raise()` and `irql_lower()`, which are shown in Code Listing 3.1. These functions will only ever be called in the context of the thread itself, so whenever a thread lowers its IRQL, it can check for pending interrupts and deliver them by invoking the respective ISRs directly through normal function calls.

Code Listing 3.1: Implementation of `irql_raise()` and `irql_lower()` as interface to changing a thread's current IRQL.

```
1  #define CURRENT_IRQL (CURRENT_THREAD->tcb_irql)
2
3  irql_t
4  irql_raise(irql_t new_irql)
5  {
6      irql_t old_irql;
7
8      irql_lock_acquire();
9
10     assert(new_irql >= CURRENT_IRQL);
11     old_irql = CURRENT_IRQL;
12     CURRENT_IRQL = new_irql;
13
14     irql_lock_release();
15
16     return old_irql;
17 }
18
19 void
20 irql_lower(irql_t new_irql)
21 {
22     irql_lock_acquire();
23
24     assert(new_irql <= CURRENT_IRQL);
25     CURRENT_IRQL = new_irql;
26
27     check_for_pending_irql_interrupts();
28
29     irql_lock_release();
30 }
```

The function `check_for_pending_irql_interrupts()`, shown in Code Listing 3.2, is used to check for and deliver pending interrupts. As long as the thread's pending IRQL interrupt bitmask has any bits set corresponding to interrupts at IRQLs higher than the IRQL at which the function was invoked, it will clear the bit for the highest-priority pending interrupt, call the associated ISR at the IRQL of that interrupt, and then the process is repeated after the ISR returns. Only a single, sequential pass over the bitmask is needed to check for and deliver all pending interrupts, as only interrupts at IRQLs below or equal to the current IRQL will be registered as pending at any point of time—interrupts at IRQLs greater than the current IRQL will be delivered right away. The only exception is that after an ISR has returned, the bit corresponding to the IRQL

of that ISR must be checked again, because the ISR itself may have requested an interrupt at the same IRQL to be delivered after the ISR completes.

Code Listing 3.2: Implementation of `check_for_pending_irql_interrupts()`.

```
static void
check_for_pending_irql_interrupts(void)
{
    irql_t highest_irql, old_irql, isr_irql;

    highest_irql = HIGHEST_BIT_SET(
            CURRENT_THREAD->tcb_pending_irql_interrupt_mask);

    if (highest_irql == 0)
        return;

    // Save IRQL
    old_irql = CURRENT_IRQL;

    while (1) {
        isr_irql = HIGHEST_BIT_SET_BELOW_EQ(
                CURRENT_THREAD->tcb_pending_irql_interrupt_mask,
                highest_irql);

        if (isr_irql <= old_irql)
            break;
        else if (isr_irql < highest_irql)
            highest_irql = isr_irql;

        CLEAR_BIT(CURRENT_THREAD->tcb_pending_irql_interrupt_mask,
                isr_irql);
        CURRENT_IRQL = isr_irql;

        invoke_irql_isr(isr_irql);
    }

    // Restore IRQL
    CURRENT_IRQL = old_irql;
}
```

To actually post an interrupt targeted at a specific thread, there are two different cases to be considered; a thread may request an interrupt to be delivered either to *itself*, or to *another thread*. In the first case, the ISR may be called directly if the IRQL of the interrupt is greater than the current IRQL. Otherwise, the interrupt is registered as pending by setting the corresponding bit in the pending IRQL interrupt bitmask, and will be delivered when the thread later calls `irql_lower()`. This functionality is implemented by the function `irql_request_interrupt()`, shown in Code Listing 3.3.

Whereas the first case is trivial, since all code is run in the context of the same thread, the second case is different because it requires a mechanism to manipulate the execution flow of some other, specified thread. On Windows, interrupt requests are handled by the hardware abstraction layer (HAL), and there is a similar separation between interrupt requests targeted at the same processor or another processor, where the HAL will issue an inter-processor

Code Listing 3.3: Implementation of `irql_request_interrupt()`, used by a thread to request an interrupt to be delivered to itself.

```
1  void
2  irql_request_interrupt(irql_t irql)
3  {
4      irql_t old_irql;
5
6      irql_lock_acquire();
7
8      old_irql = CURRENT_IRQL;
9      if (old_irql < irql) {
10         // Raise IRQL to that of ISR
11         CURRENT_IRQL = irql;
12
13         invoke_irql_isr(irql);
14
15         // Lower IRQL back to what it was before
16         CURRENT_IRQL = old_irql;
17         check_for_pending_irql_interrupts();
18     } else {
19         SET_BIT(CURRENT_THREAD->tcb_pending_irql_interrupt_mask, irql);
20     }
21
22     irql_lock_release();
23 }
```

interrupt (IPI) in the latter case. If the interrupt request is an APC targeted at a specific thread, the HAL will deliver an internal interrupt to the current processor if the requesting thread and the target thread are the same or they are hosted on the same processor core. Otherwise, if the thread is hosted on a remote core, the HAL will request an IPI targeted at that core.

In Casuar all threads are managed by Vortex, and we therefore require special support from Vortex through its paravirtualized system service interface for emulating inter-thread interrupts. Conveniently, Vortex provides the system calls `vx_thread_getcontext()` and `vx_thread_setcontext()`, for retrieving and altering the register context of a given thread. They are typically used together with `vx_thread_suspend()` and `vx_thread_resume()`, because the register context of a thread may only be accessed when the thread is in a suspended state.

By suspending a thread and modifying its thread context—specifically its stack, segment registers, and instruction pointer—before resuming it afterwards, it is possible to effectively interrupt the thread to have it execute some ISR. However, it is important that the thread be able to resume its previous execution from before the point of interruption, after the ISR completes. On x64, when an actual interrupt occurs, the CPU will trap into the kernel and push an interrupt frame on the kernel stack before the ISR is invoked. This frame is often referred to as a *machine frame*, and consists of the current stack segment (SS),

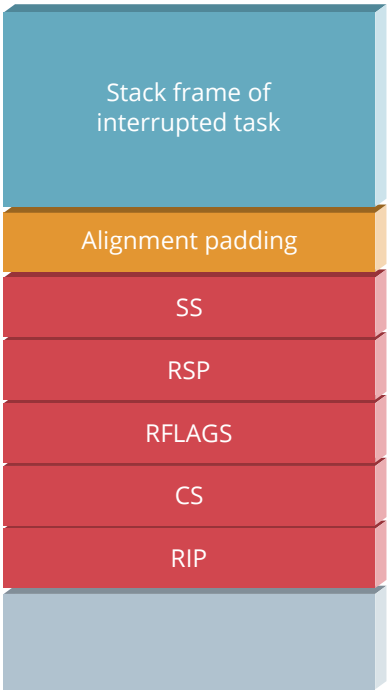


Figure 3.3: Layout of a machine frame that is pushed onto a kernel stack by the CPU when an interrupt occurs.

stack pointer (RSP), flag register (RFLAGS), code segment (CS), and instruction pointer (RIP) [59, Ch. 6.14].[5]

The segment and stack registers are needed as part of the machine frame, because the CPU may have been interrupted while executing code in user mode. Transitioning into kernel mode from user mode will result in both a change of privilege level and a stack switch from a user stack to a kernel stack, and it must be possible to restore the user stack and the previous privilege level afterwards. Before the frame is pushed, the CPU will make sure to align the stack pointer to a 16-byte boundary, due to restrictions in calling-conventions on x64. Figure 3.3 illustrates what the kernel stack will look like after a machine frame is placed on the stack.

The last thing an ISR does, after it has completed its work, is typically to execute an IRET instruction to resume the previously interrupted task. This is a special instruction in the x64 instruction set that will pop the machine frame off the stack, and use it to restore the registers contained within. The

5. Note that on x86, the SS and RSP registers are only pushed if the interrupt resulted in a privilege level change (i.e. the CPU was executing code in user mode), whereas on x64 they are pushed unconditionally [59, Ch. 6.14].

only side effects of the IRET instruction that are not determined directly by the contents of the machine frame, are related to non-maskable interrupts (NMIs) [61, Ch. 3.2], [59, Ch. 6.7], [62, Ch. 25.3]. Hence, since we do not deal with any sorts of virtual interrupts, we are able to exploit the effect of the IRET instruction as part of a mechanism for emulating interrupts.

Specifically, our approach is to construct a machine frame manually, populated from the current register context of the thread to be interrupted. Then, we place this machine frame on the kernel stack of the thread, and update the stack pointer in the thread context to point to the beginning of the machine frame—thereby simulating the stack push operation that is performed by the CPU when an interrupt occurs. Finally, the interrupt handler is given an entry point that, after calling an ISR, will execute an IRET instruction to restore the thread's previous state from the constructed machine frame.

We need to make sure that the ISR to be called as the result of an interrupt is invoked at its corresponding IRQL. However, raising the IRQL of the interrupted thread is preferrably done in context of the thread itself, as we need to make sure the thread is resumed at the same IRQL as before. Furthermore, the functionality for invoking an ISR is already implemented in the function `check_for_pending_irql_interrupts()`. Therefore, we use as interrupt handler a function that is invoked at the same IRQL that the thread was running at prior to interruption. Before an interrupt is posted, the bit corresponding to the interrupt is set in the pending IRQL interrupt bitmask. Then, when the interrupt is dispatched, the interrupt handler will simply call `check_for_pending_irql_interrupts()` to invoke all pending ISRs, as shown in Code Listing 3.4.

Code Listing 3.4: Implementation of the C code IRQL interrupt handler.

```
1  void
2  irql_interrupt_handler()
3  {
4      irql_lock_acquire();
5                  ⋮
6
7      check_for_pending_irql_interrupts();
8                  ⋮
9
10     irql_lock_release();
11 }
```

There are, however, a few special considerations that must be dealt with when using this approach. On x64, the SYSCALL instruction [63, Ch. 4.2] is used to implement fast system calls. When the system call handler is invoked after trapping to kernel mode, the CPU is still executing on the user stack. Thus, it is necessary to get hold of the kernel stack pointer in order to switch stacks.

To allow easy access to control structures in user mode and kernel mode, x64 provides a mechanism that allows an arbitrary pointer to be stored in the processor's GS base register, and a SWAPGS instruction [63, Ch. 4.2] to exchange the current value of the GS base with another hidden register called IA32_KERNEL_GS_BASE. The hidden register is typically used to contain the GS base belonging to kernel mode when in user mode, as well as the other way around, and the SWAPGS instruction is used when trapping to or exiting from the kernel, to change the value of the GS base.

Windows uses the GS base to point to the processor control region (KPCR) structure of the current processor in kernel mode, and the thread environment block (TEB) of the current thread in user mode. Similarly, in Casuar, we let the GS base contain a pointer to the current thread's kernel-level TCB when in kernel mode, and make sure to retain the GS base as a TEB pointer when in user mode, because it is heavily depended on by DLL code. We execute SWAPGS as the first instruction in the system call handler, and may then obtain the kernel stack pointer from the TCB by addressing it relatively to the GS register. After the system call completes, the system call handler will change the GS base back to the TEB pointer by executing another SWAPGS instruction before returning to user mode.

Considering that a thread may be interrupted while executing code in the system call handler, it is possible for the interrupt to happen both before and after any of the SWAPGS instructions. This means that once the interrupt handler is invoked, it is not straightforward to determine whether a SWAPGS instruction should be executed or not, merely by examining the previous privilege level in the CS register of the machine frame. On Windows and other OSs running on x64, this problem does not occur, because the system call mechanism can be programmed to disable all interrupts in response to the SYSCALL instruction, before the system call handler is invoked. In our case, this is not possible, so we have found another solution to the problem.

We use a single interrupt handler, but implement two versions of the entry point to that handler—one that executes SWAPGS on entry and exit, and one that does not. After the thread to be interrupted has been suspended, its register context is examined to see if the thread has already loaded the proper GS base, corresponding to its current privilege level, or not. This is possible to determine, because it is known what the GS base value should be when the thread is operating in kernel mode—namely the address of the thread's kernel-level TCB. If the thread was interrupted from kernel mode, and the GS base already contains the TCB pointer, then the entry point without the SWAPGS instruction is chosen. Otherwise, the entry point with SWAPGS is used instead. The resulting implementation of the interrupt handler entry points is shown in Code Listing 3.5.

Code Listing 3.5: Implementation of the assembly code IRQL interrupt handler entry points.

```
1  .macro IRQL_INT_ENTRY
2      # Allocate trap frame (MACHFRAME is already pushed to stack)
3      subq    $TRAP_SIZE_MINUS_MACHFRAME_TYPE0, %rsp
4      # Save volatile registers
5      SAVE_INT_TRAPFRAME
6                  ⋮
7                  ⋮
8      callq   irql_interrupt_handler
9                  ⋮
10                 ⋮
11     # Restore volatile registers
12     RESTORE_INT_TRAPFRAME
13     addq    $TRAP_SIZE_MINUS_MACHFRAME_TYPE0, %rsp
14 .endm
15
16 .globl asm_irql_int_entry_from_user
17 asm_irql_int_entry_from_user:
18     # Used as handler when GS contains user-mode value.
19     # At this point, a type 0 MACHFRAME has been pushed to stack.
20
21     swapgs          # Make sure GS points to CURRENT_THREAD.
22     IRQL_INT_ENTRY
23     swapgs          # Swap GS base back to whatever it was before
24     iretq
25
26 .globl asm_irql_int_entry_from_kernel
27 asm_irql_int_entry_from_kernel:
28     # Used as handler when GS contains kernel-mode value.
29     # At this point, a type 0 MACHFRAME has been pushed to stack.
30
31     IRQL_INT_ENTRY
32     iretq
```

A second consideration is to ensure that the machine frame is placed at the end of the interrupted thread's kernel stack, so that previous stack frames are not overwritten. If the thread is interrupted from user mode, the end of the kernel stack is already available in a `tcb_stack_current` variable in the thread's TCB, used to switch stacks upon system call entry. However, if the thread is interrupted from kernel mode, special care must be taken because there is a small window in the beginning and end of the system call handler where the RSP still contains the value assigned to it from user mode, before the stack switch occurs. A malicious user application could potentially alter the RSP to point to memory locations within the kernel. If this is not detected, then critical kernel data structures could be overwritten by stack frames as the kernel stack grows from the initial RSP.

Every kernel stack has a known *base*—the memory address directly following the last byte of allocated stack memory, from which the stack grows downwards—and *limit*—the address of the first allocated byte, which indicates the maximum capacity of the stack. First of all, we require that the RSP

lies between the base and limit of the current kernel stack if the thread was interrupted from kernel mode. This will always be the case when the RSP contains the value of the true kernel stack pointer. Hence, if the RSP is not within the kernel stack area, it must have been assigned from user mode. In this case, it is safe to use the value in `tcb_stack_current` as stack pointer instead, because the thread came from user mode, and this is the value that would have been loaded by the system call handler if the thread had been interrupted after the stack switch.

Still, a user application could guess where the kernel stack is located in memory, and alter the RSP accordingly. By pointing the RSP at the start of the kernel stack, the application could attempt to overwrite previous stack frames, or it could try to provoke a stack overflow by placing the RSP near the end of the stack. To prevent previous stack frames from being overwritten, we make sure that the RSP is not higher than the value in the `tcb_stack_current` variable. A correctly placed kernel stack will always grow downwards from this value, so the RSP must have been altered from user mode if it lies between `tcb_stack_current` and the base. As in the previous case, we can replace the RSP with `tcb_stack_current` without risking to overwrite any frames that are in use.

To prevent stack overflow, we cannot rely on any boundary checks, because it cannot be determined if an RSP between `tcb_stack_current` and the stack limit is a true kernel stack pointer or not. Instead, we require that the available space between the RSP and the stack limit is sufficiently large to contain the necessary stack frames to serve the ISR. In sum, these checks will guarantee that an ISR will execute on a consistent kernel stack, even when the RSP has been altered in user mode.

This concludes the description of how we have implemented a mechanism for emulating software interrupts. We use this mechanism to implement APCs, which are described in more detail in Section 3.2. The final implementation of the function used to post an interrupt targeted at a remote thread is shown in Code Listing 3.6.

Code Listing 3.6: Implementation of `irql_interrupt_remote_thread()`. Certain details have been simplified or omitted.

```
1  void
2  irql_interrupt_remote_thread(win_tcb_t *thread, irql_t irql)
3  {
4      vx_threadcontext_t  th_ctx;
5      vx_vaddr_t          interrupt_handler, rsp;
6      vxerr_t             vxerr;
7
8      irql_lock_acquire_thread(thread);
9
10     if (IS_BIT_SET(thread->tcb_pending_irql_interrupt_mask, irql)) {
11         irql_lock_release_thread(thread);
12         return;
13     }
14
15     SET_BIT(thread->tcb_pending_irql_interrupt_mask, irql);
16
17     if (irql <= thread->tcb_irql) {
18         irql_lock_release_thread(thread);
19         return;
20     }
21                 :
22
23     vxerr = vx_thread_suspend(thread->tcb_rid, VX_TIME_NTIME);
24                 :
25
26     vxerr = vx_thread_getcontext(thread->tcb_rid, &th_ctx);
27                 :
28
29     irql_lock_release_thread(thread);
30
31     // Get handler depending on current GS base of thread
32     interrupt_handler = get_interrupt_handler(thread, &th_ctx);
33
34     // Get kernel stack where it is safe to allocate MACHFRAME
35     rsp = get_interrupt_stack(thread, &th_ctx);
36
37     // Allocate MACHFRAME on stack and populate it from current context
38     allocate_machframe(&th_ctx, &rsp);
39
40     // Set new register context for invocation of interrupt handler
41     th_ctx.tc_registercontext.rc_ss  = GDT_KERNEL_DATA;
42     th_ctx.tc_registercontext.rc_rsp = rsp;
43     th_ctx.tc_registercontext.rc_cs  = GDT_KERNEL_TEXT64;
44     th_ctx.tc_registercontext.rc_rip = interrupt_handler;
45
46     vxerr = vx_thread_setcontext(thread->tcb_rid, &th_ctx);
47                 :
48
49     vxerr = vx_thread_resume(thread->tcb_rid);
50                 :
51
52 }
```

3.2 Asynchronous Procedure Calls (APCs)

In Section 3.1, we described asynchronous procedure calls (APCs) as software interrupts that are targeted to run in context of a specific thread. However, the Windows Kernel abstracts APCs at a higher level than CPU interrupts. Instead, it models an APC as a special sort of function call that will be invoked with a given set of parameters within a given target thread [64]. The APC LEVEL software interrupt is therefore best thought of as a delivery mechanism for APCs, rather than being synonymous to an APC.

Windows operates with two types of APCs: kernel APCs and user APCs [46, Ch. 3], [65]. User APCs are used to call a function in user mode, and kernel APCs run in kernel mode. They also differ in how they are delivered to a thread. Kernel APCs interrupt the execution of a thread, unless the thread has explicitly disabled delivery of APCs, whereas user APCs are only delivered when the thread allows them to be. Specifically, a thread must enter what is known as an *alertable* state from user mode to be able to receive user APCs [66], [67]. A thread running in user mode will be *non-alertable* during its normal execution, and can only become alertable by calling a wait functions, such as `SleepEx()` or `WaitForSingleObjectEx()`, with a special `Alertable` flag set to `TRUE`. The concept of *alertability* will be elaborated on in Section 3.3.

APCs are represented by `KAPC` kernel objects, each containing information about which function to call, what parameters will be passed to the function, and which thread the APC is targeted at. The main function of an APC is known as its *normal routine*. It is always invoked at PASSIVE LEVEL, and must be a kernel-mode function for kernel APCs or a user-mode function for user APCs. In addition, all APCs have a *special routine*, which is always a kernel-mode function. It executes at APC LEVEL before the normal routine, and may modify the arguments stored in the `KAPC` before they are passed to the normal routine. The special routine can even change which normal routine should be called after the special routine returns, or prevent the normal routine from running at all. The type definitions for the normal and special routine of an APC, as used in the Windows Kernel, are shown in Code Definition 3.1.

User-mode code can only specify the normal routine to be called by a user APC; which kernel routine is used depends on the system service that is invoked to queue the APC to a thread. The kernel routine is commonly used to free the `KAPC` object associated with the APC, as this object has to be allocated dynamically by the system in most cases. This is, for instance, the case with the `NtQueueApcThread()` system call, used to implement the `QueueUserAPC()` Windows API function.

There are two types of kernel APCs—*normal* kernel APCs and *special* kernel

Code Definition 3.1: Windows Kernel type definitions for normal and special routine of an APC. Definitions are borrowed from [68].

```
typedef void (*PKNORMAL_ROUTINE)(
    IN void *NormalContext,
    IN void *SystemArgument1,
    IN void *SystemArgument2
);

typedef void (*PKKERNEL_ROUTINE)(
    IN KAPC *Apc,
    IN OUT PKNORMAL_ROUTINE *NormalRoutine,
    IN OUT void **NormalContext,
    IN OUT void **SystemArgument1,
    IN OUT void **SystemArgument2
);
```

APCs [46, Ch. 3], [65]. The difference between these is that special kernel APCs have only a special routine, whereas normal kernel APCs have both a special and a normal routine. Furthermore, special APCs are regarded as of higher importance than normal APCs, and are delivered before any pending normal APCs.

A thread executing in kernel mode may disable the delivery of normal kernel APCs by entering a *critical region*, using the `KeEnterCriticalRegion()` function of the Windows Kernel. To disable both normal and special kernel APCs, `KeEnterGuardedRegion()` may be used instead to enter a *guarded region*.[6] The functions `KeLeaveCriticalRegion()` and `KeLeaveGuardedRegion()` are used as counterparts, to leave a critical or guarded region, respectively. When either type of kernel APC is re-enabled, as result of calling one of these functions, the thread will check for and deliver all pending APCs that are now allowed to be received. This is similar to how our implementation of `irql_lower()`, from Code Listing 3.1, works for IRQL interrupts.

The `KeInitializeApc()` function of the Windows Kernel is used to initialize a `KAPC` object, and the `KeInsertQueueApc()` function is called to queue an APC for delivery to a thread. The prototypes for these functions are shown in Code Definition 3.2. Among the arguments to `KeInitializeApc()` is the `KAPC` object to initialize, the thread to target, a flag that specifies whether it is a kernel or user APC, the special and normal routine to associate with the APC, and the first parameter that will be passed to those routines. Two more APC routine parameters are specified as arguments to `KeInsertQueueApc()`, which will be passed as second and third argument to the special and normal routine of the APC.

6. It is also possible for a thread to raise its current IRQL to APC LEVEL to disable all APCs, as discussed in Section 3.1.

Code Definition 3.2: Windows Kernel interface for initializing an APC and enqueuing it to a thread. Definition is borrowed from [68].

```
NTKERNELAPI void
KeInitializeApc(
    IN KAPC *Apc,
    IN KTHREAD *Thread,
    IN KAPC_ENVIRONMENT Environment,
    IN PKKERNEL_ROUTINE KernelRoutine,
    IN PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL,
    IN PKNORMAL_ROUTINE NormalRoutine OPTIONAL,
    IN MODE ApcMode,
    IN void *NormalContext
);

NTKERNELAPI BOOLEAN
KeInsertQueueApc(
    IN KAPC *Apc,
    IN void *SystemArgument1,
    IN void *SystemArgument2,
    IN KPRIORITY Increment
);
```

In addition to the normal and special routine, the `KeInitializeApc()` function takes an optional *rundown routine* as argument. This is a special function that is called only when the kernel flushes the APC queues as a result of the thread being terminated. It is typically used to perform cleanup actions, such as freeing `KAPC` objects that have been allocated dynamically.

When `KeInsertQueueApc()` is called, the given `KAPC` object is placed onto an APC queue that belongs to the target thread. Each thread maintains two such queues, implemented as circular lists, in its `KTHREAD` control structure—one for kernel APCs and the other for user APCs. Normal APCs are inserted at the tail of the APC list, whereas special APCs are placed in front of all normal APCs, after any previously enqueued special APCs. That is, the tail of the special APC list is at the head of the normal APC list, as illustrated in Figure 3.4.

After an APC has been enqueued, the target thread is notified for APC delivery unless the APC is not currently allowed to be received by the thread. Otherwise, if the APC is a kernel APC, and the thread is in a normal running state, an APC LEVEL interrupt is posted to the thread to execute an interrupt service routine (ISR) installed by the APC subsystem. However, if the thread is in a wait state, and APC delivery is allowed, it is signaled for wake-up instead. If a thread was awakened from an alertable wait state by a user APC, it will return from the wait after delivering pending APCs. A kernel APC, on the other hand, will only temporarily awaken a waiting thread, and the wait will be resumed after the APC has completed. The mechanisms related to waits and signaling will be detailed further in Section 3.3.

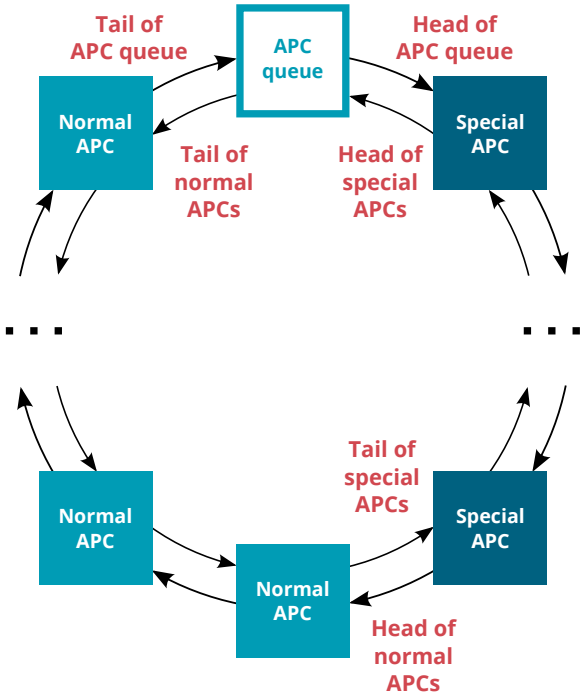


Figure 3.4: APC queue implemented as a circular list of KAPC objects. Normal APCs are inserted at the tail of the APC list, whereas special APCs are inserted at the tail of the special APC list, in front of all normal APCs. APCs are dequeued for delivery from the head of the APC list.

When a thread is interrupted or signaled to process APCs, it will deliver kernel APCs first, before checking to see if user APCs should be delivered as well. All pending APCs that are allowed to be received will be processed before the thread resumes its previous task. This applies to user APCs as well as kernel APCs; hence, once a thread enters an alertable state from user mode by calling a wait function, it will not return from that function until either the user APC list has been emptied, or the thread is forced out of the alertable state by a special alert from the kernel [66].

Threads dequeue KAPC objects from the head of the kernel and user APC list, and thus deliver APCs in the order in which they have been queued. The only exception to this, is that the delivery of special APCs will preempt the normal routine of any normal APC that may have been running at the time when a special APC is queued to the thread [65]. In this case, the special APCs will be delivered in the context of an APC LEVEL interrupt, and the preempted normal routine will resume once there are no more special APCs to process.

On the other hand, a normal kernel APC will never be preempted by another

normal kernel APC. Although it is possible for the normal routine of a normal kernel APC to be temporarily interrupted by an APC LEVEL interrupt, since the routine is executing at PASSIVE LEVEL, the APC LEVEL ISR makes sure not to start executing the next normal APC from the kernel APC list as long as another kernel APC is already running. This is done by checking a special `KernelApcInProgress` flag in the `KTHREAD` structure, which is set at APC LEVEL before starting to run an APC routine, and cleared after the routine returns.

3.2.1 Implementing APCs in Casuar

Recall from Section 3.1 that the reason we wanted to implement the abstraction of APCs in Casuar is to support system services that depend on it. System calls such as `NtQueueApcThread()`, `NtReadFile()`, and `NtWriteFile()` take as arguments a callback to be invoked as the normal routine of a user APC, and a parameter that is passed to the routine. However, it is only *user APCs* that are exposed directly to user mode. Kernel APCs are only used internally in Windows NT. The functions `KeInitializeApc()` and `KeInsertQueueApc()` are not even available for use by third-party drivers.

Still, there are several system services that depend on kernel APCs indirectly, as they are used to serialize asynchronous operations to a thread. This is, for example, the case for the system calls `NtSuspendThread()` and `NtResumeThread()`, which implement the functionality used by the Windows API functions `SuspendThread()` and `ResumeThread()`. Other examples are the `NtGetContextThread()` and `NtSetContextThread()` system calls, which are used by the `GetThreadContext()` and `SetThreadContext()` functions.

There are some details concerning the APC implementation in the Windows Kernel that we have not yet shed light on. The astute reader may have noticed that the prototypes of `KeInitializeApc()` and `KeInsertQueueApc()` from Code Definition 3.2 contain an *APC environment* argument and a *priority increment* argument, respectively, which we did not describe earlier. Both parameters relate to functionality in Windows that we do *not* replicate in Casuar.

The priority increment is very specific to how the thread dispatcher is implemented in Windows. Recall from Section 3.1 that we have no dispatcher component in Casuar, because all thread scheduling is done directly by Vortex. Hence, this parameter is irrelevant to our implementation, and so we will not consider it further. The reasons for not implementing APC environments, on the other hand, are mostly because of their complexity—they are part of a larger mechanism in the Windows Kernel that allows a thread belonging to some process to temporarily execute in the address space of another process.

Specifically, the functions `KeAttachProcess()` and `KeStackAttachProcess()` are provided by the Windows Kernel for detaching a thread from the process in which it is currently executing, and attach it to some other, specified process. This allows a thread to gain access to code and data in the address space of another process, which is not otherwise available from the original process that the thread was created in. Afterwards, a thread must return to its original process, by calling `KeDetachProcess()` or `KeUnstackDetachProcess()`. Calls to `KeStackAttachProcess()` and `KeUnstackDetachProcess()` are allowed to be nested, as long as the thread eventually ends up back in its original process.

The APC mechanism in Windows has to accommodate the possibility of a thread becoming attached to different processes at various times, because APCs are highly dependent on executing in a particular process address space; an APC targeted to run in the thread's original address space will not be able to run if the thread is attached to another process, as code and data from the original address space, which is needed by the APC, will become unavailable. However, it is required that the thread still be able to queue up APCs for execution in the original process, and that APCs explicitly targeted at the attached process may also be queued and delivered.

In Windows, this is implemented by having each thread maintain different sets of APC queues for the original process and the attached process if these are not the same. The queues and other APC specific state that is dependent on a particular address space, such as the `KernelApcInProgress` flag, is contained in a `KAPC_STATE` structure [69]. A thread's `KTHREAD` structure contains a `KAPC_STATE` for the currently active address space, and also has two APC state pointers—one that points to the APC state of the original process, and one that points to that of the attached process.

The caller of `KeStackAttachProcess()` must allocate memory for a `KAPC_STATE` that will be used to contain the thread's previous APC state, and later supply it as argument to `KeUnstackDetachProcess()` to restore the state [70]. The thread's `KAPC_STATE` and APC state pointers are updated accordingly when these functions are being called. This means that when an APC is queued to a thread, the APC state pointers allow the APC to be targeted to run in either the original or the current process address space in a consistent manner.

The APC environment argument to `KeInitializeApc()` is simply a flag that indicates which APC state pointer will be used, and thus, onto which APC queue the APC will be inserted. An APC may either be targeted at the original address space, the current address space at the time when `KeInitializeApc()` is called, or whichever address space will be currently active at the time when `KeInsertQueueApc()` is called.

In Vortex, there are no similar mechanisms that would allow a thread to switch address space or in any way make it feasible to emulate such behavior. Regardless, we—perhaps somewhat optimistically—find it reasonable to assume that we will not face any situation where this functionality would in fact be needed. Therefore, we have chosen not to represent APC environments or any structure similar to `KAPC_STATE`, and instead embed all APC state directly into the kernel-level TCB of each thread.

Apart from lacking the concept of APC environments, our implementation of APCs follows that of the Windows Kernel closely. APC delivery is mainly built on top of the IRQL interrupt mechanism detailed in Section 3.1. This makes it straightforward to handle kernel APCs, as the APC LEVEL interrupt will trap into kernel mode, where both the special and normal routine of the kernel APC will be called right away. However, when delivering a user APC, the special routine must first be invoked in kernel mode, and the kernel must afterwards make sure that the normal routine is executed in user mode.

In Windows, the invocation of a user APC's normal routine does not happen immediately after its special routine has returned. Instead, the thread is intercepted to call the normal routine just before it returns back to user mode, after the system call or interrupt that resulted in APC delivery completes. The way this is implemented in Windows is fairly simple, and so we chosen to base our implementation on the same approach.

When a thread traps to the kernel as the result of a system call, interrupt, or exception, it will allocate a *trap frame* on its kernel stack, which is used to preserve the thread's user-mode register state. Normally, when the thread later returns from the kernel, it will restore its previous state from the trap frame, including the previous instruction pointer (`RIP`) that specifies the return address. However, when a user APC is being delivered to the thread, the trap frame is manipulated to let the thread return to the function `KiUserApcDispatcher()` in `ntdll.dll` instead.

To make sure that a thread will be able to return to its previous state after delivering user APCs, a `CONTEXT` record is populated from the state in the trap frame before the trap frame is changed, and pushed onto the user stack of the thread. The normal routine pointer and the parameters to the routine are placed at the beginning of the `CONTEXT` frame, as arguments to `KiUserApcDispatcher()`, which in turn will take responsibility for invoking the routine. In addition, a machine frame is pushed onto the user stack before the `CONTEXT` record. It contains the same stack pointer (`RSP`) and instruction pointer (`RIP`) as is found in the `CONTEXT` record, but is used to support stack unwinding, as part of the structured exception handling (SEH) mechanism in Windows [71]. Figure 3.5 illustrates the layout of the user stack before `KiUserApcDispatcher()`

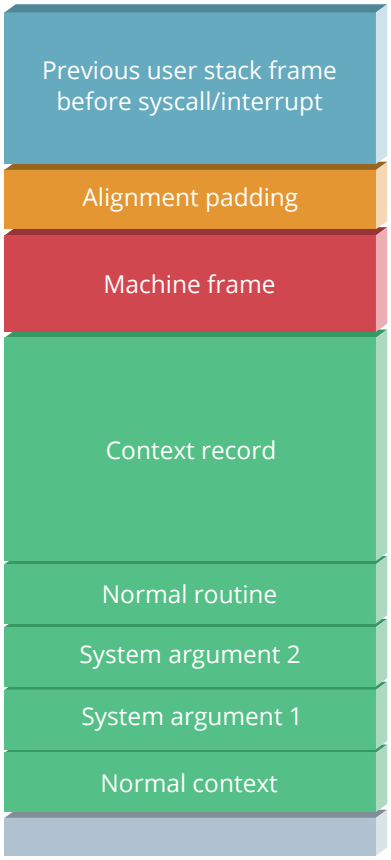


Figure 3.5: Layout of user stack before dispatching a user APC to user mode. The `CONTEXT` record contains the arguments to `KiUserApcDispatcher()` and the previous user-mode register state of the thread. The latter will be restored after pending user APCs have been delivered.

is invoked.

After the normal routine of a user APC has returned, `KiUserApcDispatcher()` will call the `NtContinue()` system call, passing a pointer to the `CONTEXT` record as argument. `NtContinue()` will then check if there are any more pending user APCs to deliver. If so—and the thread has not received a special alert—it will update the `CONTEXT` frame to contain the normal routine and parameters of the next user APC, and the thread will return back to `KiUserApcDispatcher()` to deliver the APC.

Finally, if the thread has received an alert, or the user APC queue has been completely drained, the `NtContinue()` system call will restore the register context contained in the `CONTEXT` record, and return back to user mode. Because the previous `RSP` and `RIP` are among the registers that are restored, the thread

will, in effect, return to the same place as before being intercepted to deliver user APCs.

3.3 Blocking Synchronization

One of several important functions of the Windows Kernel, is to provide fundamental *synchronization mechanisms* that can be used by the Windows Executive and—by extension—user mode applications, to coordinate concurrent access to shared resources. Apart from certain atomic operations, the lowest-level locking primitive offered by the Windows Kernel is *spinlocks* [46, Ch. 3]. Spinlocks provide non-blocking mutual exclusion—a processor that needs to acquire a spinlock will spin in a busy-wait loop as long as the spinlock is acquired by another processor, until it succeeds in taking ownership of the lock.

In general, spinlocks waste CPU cycles that could potentially be utilized by other tasks or workloads in the system. However, as may be recalled from Section 3.1, no other synchronization mechanism is available to code exceuting at high IRQLs, such as interrupt service routines (ISRs), deferred procedure calls (DPCs), and the thread dispatcher. High-IRQL tasks cannot be scheduled out in favor of other tasks, because they execute in context of an arbitrary thread that should not be prevented from running. Thus, to reduce negative impact on a system's overall performance, all code at high IRQLs is designed to run for only very short periods of time. In addition, the Windows Kernel offers queued spinlocks with FIFO semantics that may be used to further improve efficiency in scenarios that are subject to a high degree of lock contention [72]. Consequently, the performance penalty of using spinlocks at high IRQLs is kept reasonably low.

Threads, however, are used in a system to express more complex and long-lived tasks than ISRs and DPCs. They may be scheduled in any order by the dispatcher, and there are no hard restrictions on the length of a thread's quantum. Since they execute at low IRQLs, threads may also be preempted at any time by the dispatcher, except for special cases where the current IRQL has been temporarily raised to DISPATCH LEVEL. Hence, the system greatly benefits from giving threads the ability to synchronize through blocking mechanisms.

By blocking, a thread will relinquish control of the processor while waiting for some condition to be met, allowing other threads to run in the meantime. Because blocking is a scheduler-provided service, the dispatcher is given insight into what a thread is waiting for. This allows the dispatcher to make better scheduling decisions, as blocking threads will be considered ineligible to run on any processor. Although there is some overhead associated with performing

a blocking operation, the cost will be significantly smaller than that of a non-blocking alternative for threads that acquire and hold exclusive resources for longer periods of time. Finally, blocking abstractions are convenient as part of an operating system's application programming model—especially for threads that need to acquire some resource to be able to progress, and cannot overlap the wait asyncronously with other useful work.

3.3.1 Dispatcher Objects

The Windows Kernel provides several different blocking synchronization primitives to the higher-level Windows Executive layer. Each such primitive is represented as a *dispatcher object*—a special type of kernel object that can be *waited on* by one or more threads [73], [46, Ch. 3]. All dispatcher objects have a common *dispatcher header* structure, used to store each individual object's synchronization state and a list of threads waiting for the object.

The *event object* [74] is one of the most commonly used types of dispatcher objects in Windows. It provides a basic signaling mechanism that allows a thread to wait for the event to be *set* by another thread. An event is represented by a `KEVENT` structure that contains nothing more than a dispatcher header. As such, it is also the most fundamental blocking primitive of the Windows Kernel. Examples of other dispatcher objects are timers, queues, mutexes, and semaphores.

Some dispatcher objects do not correspond to raw synchronization mechanisms, but are instead larger kernel objects that embed a dispatcher header to obtain certain synchronization capabilities and become waitable. Threads and processes are examples of such objects. Both use the dispatcher header to provide join semantics; if a thread or process is waited on, the caller thread will block until the specified thread or process terminates.

The functionality for performing blocking waits is made available by the Windows Kernel through mainly three functions—`KeWaitForSingleObject()`, `KeWaitForMultipleObjects()`, and `KeDelayExecutionThread()`. The first two functions allow a thread to block while waiting to synchronize with one or more dispatcher objects. Almost all higher-level blocking synchronization mechanisms in Windows are projected down onto one of these two functions. In addition, Windows API functions such as `WaitForSingleObjectEx()` and `WaitForMultipleObjectsEx()` make their functionality directly available to user-mode applications through corresponding system calls. The last function, `KeDelayExecutionThread()`, is used for unconditional waits, where the calling thread is put to sleep for a specified time interval. Windows API functions such as `Sleep()` and `SleepEx()` are implemented on top of it, through the system

call `NtDelayExecution()`. Although the caller does not supply a dispatcher object as argument to this function, it uses a timer object internally to perform the wait operation.

A dispatcher object is always in one of two logical states—*signaled* or *non-signaled* [46, Ch. 3]. When a thread waits for a dispatcher object, it will in effect wait for the object to attain a signaled state. If the thread initiates a wait operation for only a single object, and that object is already in a signaled state, the wait is satisfied immediately and the calling thread will return from the wait procedure without blocking. Otherwise, if the object is in a non-signaled state, or the thread is waiting for other objects to be signaled as well, the thread will block until its wait can be satisfied.

A thread that is waiting for multiple objects can specify its wait type as `WaitAny` or `WaitAll`. If the wait type is `WaitAny`, the wait operation is satisfied when either of the objects attain a signaled state. This wait type is also assigned internally by the Windows Kernel to all single-object wait operations. The other type, `WaitAll`, indicates that the thread is waiting for more than one object, and that the wait will not be satisifed until all of the objects have been signaled. Regardless of wait type, a wait operation will also be satisfied if the thread specifies a timeout interval when calling a wait procedure, and the timeout expires before the wait could be satisfied otherwise.

Threads that have to block while waiting for an object to become signaled are placed on the object's wait list. When the object is signaled, one or more threads will be removed from the list and awakened. All released threads with wait type `WaitAny` are known to have their wait satisifed at this point. They can therefore be signaled for *unwait*, meaning they will return from the wait procedure immediately after wake-up. However, a thread with wait type `WaitAll` might still be waiting for other objects to become signaled. At the time of releasing the thread, the system cannot determine with certainty if doing so will result in satisfaction of the wait operation or not. Because of this, the thread is instead signaled to test if the wait has been satisfied for all specified objects. If the wait could indeed be satisfied after the thread was awakened, the thread will unwait itself and return from the wait procedure. If not, it will re-initiate the wait operation and continue to block.

Exactly how many of the waiting threads are released when an object is signaled, and what makes an object transition between the signal states, depends on the type of object. Dispatcher objects can be divided into two categories—*notification objects* and *synchronization objects*. When a notification object is signaled, all blocked threads are released from its wait list. The object then enters a signaled state, and will remain signaled until it is manually reset. Synchronization objects, on the other hand, release only a specific number of

threads when signaled, and they might automatically reset to a non-signaled state afterwards. Mutexes, semaphores, and queues are examples of synchronization objects, whereas processes and threads are notification objects. Events and timers come in two variants—a synchronization type, which always satisfies the wait of only a single thread when signaled, and a notification type.

Most synchronization objects abstract some sort of resource that can be acquired exclusively by one or more threads. A mutex, for example, represents a blocking lock that may be held by only one thread at a time. It is typically initialized in a signaled state, and transitions to a non-signaled state after being acquired by some thread. When the mutex is later released, it re-enters a signaled state and will attempt to unwait a single thread from its wait list. Unless the wait list is empty, the mutex will most likely be handed over to one of the waiting threads, in which case the mutex becomes non-signaled once more. In contrast, a queue object is initially non-signaled, and enters a signaled state when an item is placed on the queue. It will remain signaled as long as there are items left on the queue, and becomes non-signaled once the last item is dequeued.

To support the signaling semantics of different synchronization objects, the Windows Kernel implements the signal state of a dispatcher object as a *signal count* integer in the object's dispatcher header. A positive value means that the object is in a signaled state, and a value below or equal to zero indicates a non-signaled state. Notification objects only ever have a signal count of zero or one, whereas the signal count of a synchronization object may assume other positive or negative values. Each time a thread completes its wait for a synchronization object, thereby acquiring a resource, the signal count is decreased. Similarly, the signal count is increased whenever a resource is made available for acquisition. As an example, the signal count of a queue object corresponds to the number of items on the queue (which will always be a non-negative value). A mutex, on the other hand, will have a signal count less than or equal to one—a negative value reflects the number of times it has been acquired recursively by the owning thread.

When a synchronization object is signaled, the signal count determines how many waiting threads may be released. Threads are released from the object's wait list in FIFO order, and the signal count is decreased for each unwaited thread (with wait type `WaitAny`) that has not already had its wait satisfied.[7] If the number of threads that are signaled for unwait is equal to the signal count, the synchronization object will enter a non-signaled state (as the result

7. The Windows Kernel takes special care to handle races between unwaits, where more than one signaled object could concurrently attempt to unwait the same thread. If a thread is waiting for multiple objects with wait type `WaitAny`, only one of the attempted unwaits may be allowed to satisfy its wait.

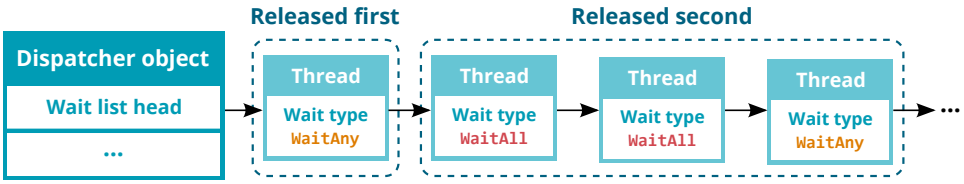


Figure 3.6: Example illustrating how threads are released from a dispatcher object's wait list. When the object is signaled, the system will continue to release threads from the head of the list as long as the signal count is positive. The signal count is lowered each time a thread with wait type `WaitAny` is successfully unwaited. Releasing threads with wait type `WaitAll` does not affect the signal count.

of the signal count being lowered to zero); otherwise, the synchronization object will remain in a signaled state. No more threads are released once the object becomes non-signaled. Note that the signal count is not decreased when releasing threads with wait type `WaitAll`, because it is not known if these threads will be able to acquire a resource from the synchronization object until after they have been awakened. This is beacuse the Windows Kernel requires all objects that are waited upon to become signaled before a thread may acquire resources from either of them [75]. Hence, the number of threads that are released might be greater than the value of the signal count at the time when the object is signaled. This is illustrated in Figure 3.6.

If a kernel APC is posted to a blocking thread, the thread will be signaled for APC delivery, unless the thread has disabled kernel APCs before it initiated the wait operation. This will cause the thread to wake up and deliver pending kernel APCs. Afterwards, the thread will check if its wait condition was satisfied while APCs were running, and either unwait itself or restart the wait accordingly. Delivery of kernel APCs will happen regardless of whether the wait was initiated from kernel mode or user mode.

When a thread calls one of the wait procedures, it may specify whether the wait is *alertable* or not [57], [66]. A thread that is in a non-alertable wait state may only be awakened if signaled for either unwait, test of wait satisfaction, or delivery of kernel APCs. However, if a thread is in an alertable wait state, it will also be awakened if it receives an alert or a user APC.

Alerts are a type of wake-up messages that are used by the Windows Executive to abort wait operations for special cases such as thread termination and cancellation of I/O operations. There are two types of alerts—*user-mode* alerts and *kernel-mode* alerts. A thread will only receive user-mode alerts if its wait operation was initiated from user mode, whereas kernel alerts may be received irrespective of the privilege level at which the wait procedure was called. User

APCs are also delivered to a thread only when waiting from user mode. If a thread is signaled for alert or user APC, its wait operation will be satisfied regardless of its wait type and the signal state of the specified dispatcher objects. When the thread receives an alert, it transitions from an alterted state to a non-alerted state, and returns from the wait procedure immediately thereafter. In the case of user APCs, the thread will not return from the wait until all pending user APCs that have been queued to the thread are delivered, as previously detailed in Section 3.2.

3.3.2 Implementation of Blocking in Windows

When a thread blocks in the Windows Kernel, it will transition from a running state to a wait state. As part of this process, the system needs to make changes to dispatcher structures that control the scheduling of threads on the processor that hosts the blocking thread. Hence, all wait-related operations execute at DISPATCH LEVEL, as they need to prevent other dispatcher services that are accessing the same structures from running.

Still, a task running on another processor may at any given time attempt to send an alert or APC to the thread, or to signal a dispatcher object that the thread is waiting for. This means that a thread's wait could potentially be satisfied while the thread is setting up the wait and preparing to block, because of an external wake-up condition that happens concurrently. To prevent races in such scenarios, all procedures related to waits and signaling therefore need to synchronize through a rigorous protocol, which is described throughout the following paragraphs.

All modification of state specific to the waiting thread is protected by a spinlock that belongs to that thread. Similarly, each dispatcher object has its own lock that protects, among other things, its signal count and wait list. Recall from Subsection 3.3.1 that an object's wait list keeps track of all threads that are waiting for the object. However, the list does not actually link together the `KTHREAD` structures of the waiting threads directly; instead, it contains a *wait block* structure for each thread.

A wait block ties together a dispatcher object with a thread that is waiting for that object. By using wait blocks, the Windows Kernel not only allows each dispatcher object to know about all threads that are waiting for it, but also lets each thread maintain a list of the objects it is waiting for; as illustrated in Figure 3.7, each wait block will at the same time be part of both the dispatcher object's wait list and a wait block list belonging to the thread. In addition, each wait block contains a *wait block state* that is used to synchronize a thread that is performing a wait operation with one or more signalers in a consistent

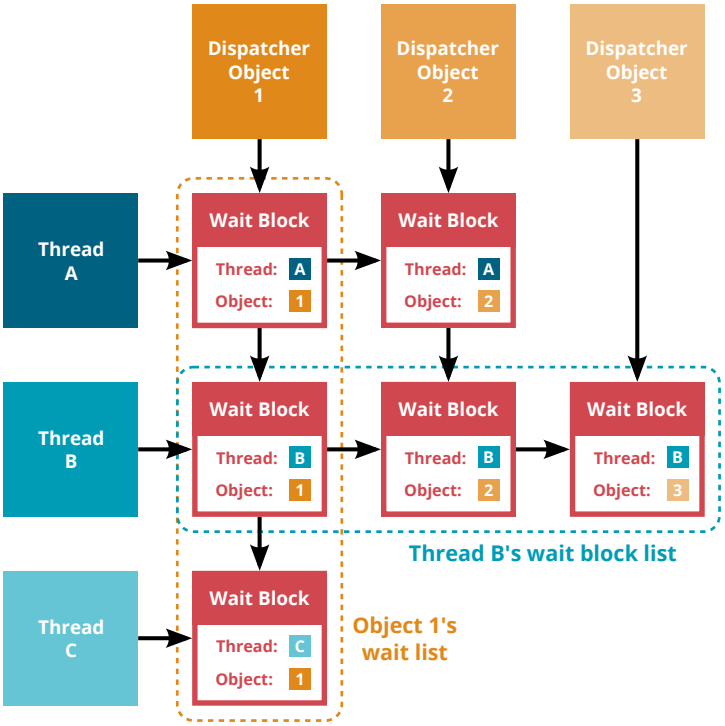


Figure 3.7: Illustration of how wait blocks links together dispatcher objects with threads waiting for the objects. Each wait block will always be part of both a dispatcher objects's wait list, used to locate threads waiting for that object, and a thread's wait block list, specifying all objects that thread is waiting for. Note that a dispatcher object's list of wait blocks corresponds to the logical list of threads shown previously in Figure 3.6.

manner [46, Ch. 3].

When a thread initiates a wait operation, it goes through several different intermediate states before it can block. First, the current IRQL is raised to DISPATCH_LEVEL. The thread will then check for pending alerts and APCs. If the thread has pending kernel APCs, the IRQL will be lowered temporarily to APC_LEVEL to deliver these, and the thread will continue trying to wait afterwards. On the other hand, the wait will be satisfied immediately if either the thread is alertable and has received an alert or user APC, or the thread has pending user APCs that have not yet been delivered after a previous alertable user-mode wait. Otherwise, the thread enters an *in-progress* wait state.

The thread lock is held during the transition to the in-progress wait state, to synchronize with other tasks that might try to deliver alerts or APCs to the thread at the same time. This synchronization is especially important because the delivery method for APCs changes when the thread enters a wait state;

recall from Section 3.2 that the Windows Kernel uses interrupts to deliver APCs to running threads, whereas a thread is signaled for wake-up instead if in a wait state. In all cases where another thread performs actions that depend on the state of a target thread, such as signaling the thread for alert or APC, the thread lock of the target thread must be acquired by the other thread first.

After the thread has reached the in-progress wait state, it will setup the wait blocks for each dispatcher object, lock all objects, and test for wait satisfaction. If, at this point, the wait is already satisfied, the thread will update the signal count of each object correspondingly and return from the wait procedure. The thread will also return if the wait was not satisfied, but the caller has specified a timeout value of zero or the timeout has already expired. In all other cases, the thread will enqueue each wait block to the wait list of each corresponding object, and try to continue to the *committed* wait state.

While the thread has been in the in-progress wait state, it may have been signaled for alert or APC, or one or more of the dispatcher objects may have become signaled. In the first case, the signaler will have updated a *wait register* that belongs to the thread to inform it of pending alerts or APCs. In the second case, the first dispatcher object that signals the thread for wake-up will change the thread's wait state from *in-progress* to *aborted*. This happens regardless of whether the thread was signaled for unwait or for wake-up without immediate wait satisfaction.

When the thread tries to enter the committed wait state, it will check its wait register and see if its wait status has been altered. Again, the thread lock is used to synchronize with potential signalers. If the thread's state or wait register has been changed by a signaler, the thread will not be able to proceed with the wait operation—either the wait has already been satisfied, or the thread is signaled to deliver kernel APCs or test for wait satisfaction manually. In the former case, the thread will exit the wait procedure, whereas in the latter case, the thread will re-start the wait and try to enter the in-progress wait state again. In addition, the thread will make sure all wait blocks are linked out of their respective dispatcher object wait lists before returning or continuing in response to the signal. However, if the thread has not been signaled while in the in-progress wait state, it can now safely enter the committed wait state. Once the thread reaches this state, it will finally be able to block.

In the Windows Kernel, each logical processor core is represented by a processor control region (`KPCR`) structure and a processor control block (`KPRCB`) sub-structure. The `KPRCB` contains all data and structures that are used by the dispatcher to schedule threads on the corresponding processor. Among these are the *ready queue—the list of threads that are to be scheduled on the processor—* and the `KPRCB` *wait list*—the list of blocking threads [46, Ch. 5].[8] When a thread

blocks, the KPRCB of the current processor is locked by acquiring a spinlock specific to that KPRCB. Then, the thread is moved from the ready queue to the wait list, and the dispatcher is invoked to schedule another thread after releasing the KPRCB wait lock.

If the thread is signaled while in the committed wait state, the signaler will actually awaken the thread, rather than abort a wait operation that is being set up. It does so by changing the thread's state from *waiting* to *deferred ready*, and correspondingly moving the thread to the deferred ready queue of the KPRCB associated with the thread [76]. The KPRCB's wait lock is held during this operation. Following this, the dispatcher will eventually schedule the thread to run again on the processor. When that happens, the dispatcher will also make sure that all wait blocks belonging to the thread have been linked out of their wait lists first—in the same way as when a wait is aborted from the in-progress wait state. After the thread wakes up, it will then return directly from the wait procedure.

3.3.3 Implementing Blocking Waits in Casuar

The implementation of blocking waits in Casuar follows closely that of the Windows Kernel, but with some notable exceptions. We do not implement any mechanism or structures that correspond to the KPCR or KPRCB. Because there is no dispatcher component in Casuar, there is no need for the *deferred ready* wait state either. The actual blocking of a thread is performed by having it suspend itself using the Vortex system call `vx_thread_suspend()`. Correspondingly, the `vx_thread_resume()` system call is used by a signaler to awaken a thread from the committed wait state.

Unlike Windows, we do not implement wait timeouts using timer dispatcher objects. In fact, Casuar does not include any abstraction that corresponds to a timer object at all. Instead, we exploit the fact that `vx_thread_suspend()` can be called with an optional timeout argument. Some translation of timeout values is necessary, because timeouts in Windows are specified in 100 ns units, and may be either relative or absolute, whereas the timeout parameter to `vx_thread_suspend()` is expressed in milliseconds, and is always relative. However, this involves significantly less work than would be required to implement the generic timer object abstraction.

8. Actually, Windows associates multiple ready queues with each processor—each for a different priority level. However, this will not be detailed further, as it does not affect how blocking is implemented in the Windows Kernel. More information about the internals of the thread dispatcher may be found in [46, Ch. 5].

The only dispatcher objects we fully support are events, threads, and processes, although the implementation of the wait and signaling mechanisms is sufficiently generic that it could easily be extended to include mutexes and semaphores as well. For synchronization with dispatcher objects, we only implement one wait procedure, `thread_wait_for_single_object()`, which corresponds to the `KeWaitForSingleObject()` function of the Windows Kernel. An overview of its implementation is shown in Code Listing 3.7. The implementation of a procedure corresponding to `KeWaitForMultipleObjects()` has been deferred to future work. In addition, Casuar includes a simple implementation of a function `thread_delay_execution()`, which corresponds to `KeDelayExecutionThread()`. It uses `thread_wait_for_single_object()` to wait for an event object that is allocated locally on the caller's stack with the supplied timeout. Because the event is not visible outside of the function, it cannot be signaled, meaning that the wait will only be satisfied if either the timeout expires, or the wait is alertable and the thread receives an alert or user APC.

Using `vx_thread_suspend()` and `vx_thread_resume()`, however, is not completely straightforward, because the semantics implemented by Vortex is slightly different from that of traditional suspend and resume operations. Internally, Vortex associates a *suspend count* with each thread to resolve races between suspend and resume operations that could be submitted to the same thread concurrently. This value is incremented for each suspend call, and decremented on every resume. When `vx_thread_suspend()` is called, the target thread will only be suspended if the suspend count is positive after its value has been incremented. However, when `vx_thread_resume()` is called, the thread will be awakened regardless of what the suspend count value is. Vortex delegates the responsibility for balancing the suspend count to the application. In other words, this must be dealt with in Casuar's implementation of blocking waits.

In Casuar, there are three cases in which a thread will be suspended:

- The thread suspends itself in order to block, after entering the committed wait state.

- The thread is suspended by another thread as part of the mechanism for posting an interrupt to the first thread (see Code Listing 3.6 in Section 3.1).

- The thread generates an exception (such as a page fault or general protection fault) that cannot be handled by Vortex. Normally, this will result in the process being terminated. However, a process may inform Vortex that it wants to handle exceptions on behalf of its threads. In this

Code Listing 3.7: Implementation of the wait procedure for synchronizing with a single dispatcher object. Certain details have been simplified or omitted for the sake of brevity.

```
NTSTATUS
thread_wait_for_single_object(void *object,
                              winsys_wait_reason_t wait_reason,
                              winnt_pmode_t wait_mode, vx_bool_t alertable,
                              vx_int64_t *timeout)
{
    winsys_wait_block_t    *wait_block;
    vx_uint64_t             abs_wakeup_time;
    vx_bool_t               has_timeout;
    vx_time_t               vxtimeout;
    NTSTATUS                status;
                :
                :
    CURRENT_THREAD->tcb_wait_irql = thread_enter_dispatch_mode();
    has_timeout = setup_timeout(timeout, &abs_wakeup_time);

    while (VX_TRUE) {
        status = thread_enter_in_progress_waiting_state(wait_reason,
                        wait_mode, alertable);
        if (status == STATUS_KERNEL_APC)
            continue; // Signaled for kernel APC / wait satisfaction test
        else if (status != STATUS_SUCCESS)
            return status; // Signaled for alert or user APC

        wait_block = setup_single_wait_block(object);

        kobj_lock_acquire_at_dispatch_level(object);
        if (try_satisfy_wait(object, &status)) {
            kobj_lock_release_to_dispatch_level(object);
            break;
        }

        if (!get_vxtimeout(has_timeout, abs_wakeup_time, &vxtimeout) {
            status = STATUS_TIMEOUT;
            kobj_lock_release_to_dispatch_level(object);
            break;
        }

        enqueue_wait_block(object, wait_block);
        kobj_lock_release_to_dispatch_level(object);

        status = thread_commit_wait(wait_block, vxtimeout);
        if ((status != STATUS_KERNEL_APC) && (status != STATUS_TIMEOUT))
            return status; // Signaled for unwait

        // Signaled for kernel APC or test of wait satisfaction
        CURRENT_THREAD->tcb_wait_irql = thread_enter_dispatch_mode();
    }           :
                :
    thread_lock_acquire_at_dispatch_level();
    CURRENT_THREAD->tcb_thread_state = THREAD_STATE_RUNNING;
    thread_lock_release_to_dispatch_level();
                :
                :
    return status;
}
```

case, the faulting thread will be suspended, and an exception message is delivered to the process instead. We use this for monitoring and debugging purposes, as will be described in Chapter 5.

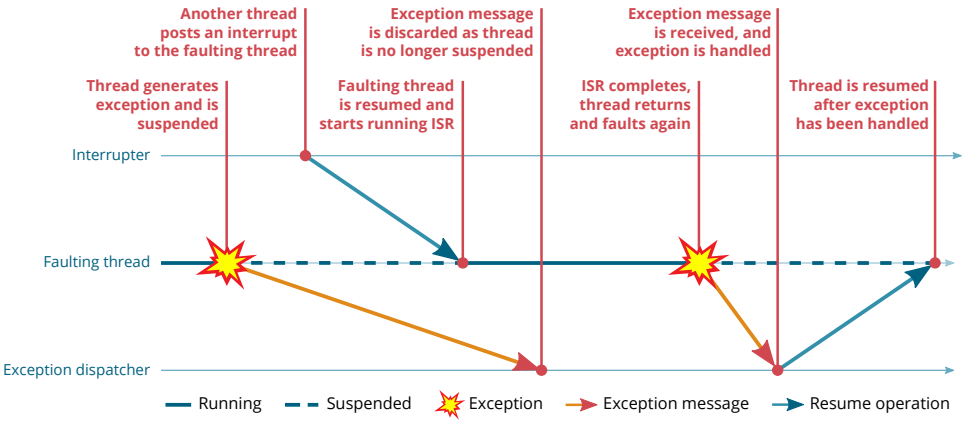
The first two cases happen as the result of explicit calls to `vx_thread_suspend()`. This means that the caller can take the necessary actions to synchronize with other tasks that are issuing matching calls to `vx_thread_resume()`. In the first case, the thread's wait state is changed to *committed* before suspending the thread, and in the second case, the thread lock of the target thread (to be interrupted) is acquired first, and the caller ensures the target thread is in a *running* state.

However, the suspend operation in the third case will be performed automatically by Vortex in response to an exception, which may occur at any time. In particular, the thread will still be in a running state from Casuar's point of view when this happens, although it has been suspended by Vortex. Hence, one thread might end up trying to interrupt another thread that has already been suspended due to an exception. This, in itself, cannot be directly prevented, and results in a few, special cases that Casuar must handle explicitly to ensure that the suspend count is kept in balance.

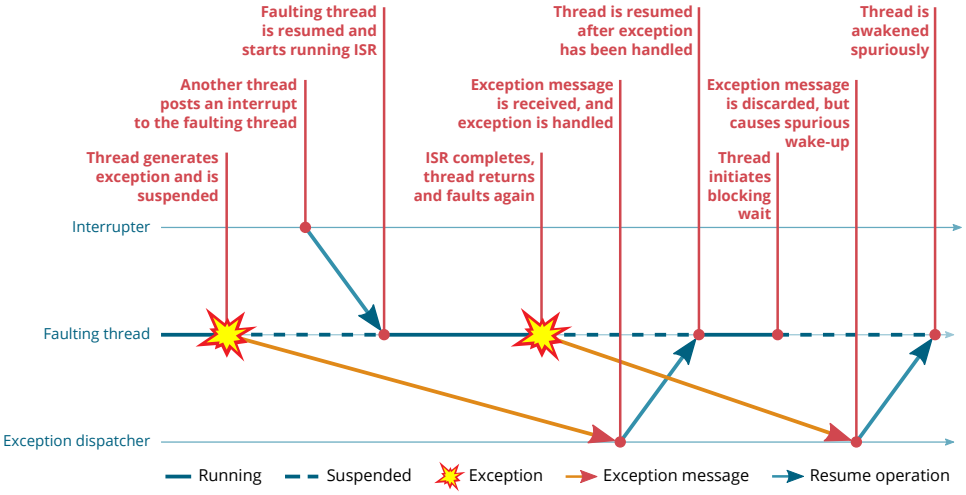
Recall from Section 3.1 that interrupts are delivered by suspending the target thread, updating its context, and then resuming it again to have it execute an interrupt handler. If this happens to an already suspended thread, it will first become double-suspended, and then be resumed with a suspend count that is out of balance. After the interrupt handler has executed, the thread will return to its previous context. This means that the thread will return to the previously faulting `RIP`. Then, when the thread executes the same instruction as before, it will likely fault again, which means that the thread will be suspended once more and Vortex will generate another exception message.

When a process receives an exception message from Vortex, it will be processed by a separate exception dispatcher worker thread that is set up for each process by Casuar. This processing is performed asynchronously with other tasks in the system. Hence, it could happen that the exception dispatcher tries to handle an exception for a thread that has faulted, but that thread is no longer suspended because it has been interrupted. This scenario is illustrated in Figure 3.8a. Moreover, the worker thread might receive multiple exception messages for the same exception, if the faulting thread has been interrupted one or more times before the exception dispatcher has had time to handle the first exception message for that thread. This is shown in Figure 3.8b.

For the exception dispatcher to be able to process exception messages properly, it first needs to synchronize with potential interrupters of the faulting thread.



(a) Exception dispatcher receives an exception message for the faulting thread, but cannot handle it, because the thread has received an interrupt and is no longer suspended. The exception message is therefore discarded. This can be done, because the exception dispatcher will receive a new exception message after the thread returns from the ISR. As part of discarding the message, the exception dispatcher will issue a resume (not shown in the figure) to balance the thread's suspend count. This has no other side effects because the thread is already running.



(b) Faulting thread receives an interrupt, and an extra exception message is therefore generated for the same exception. After the first message is handled, the second message must be discarded. If, at that time, the thread has initiated a blocking wait, it will be awakened spuriously when the exception dispatcher issues a resume for the discarded message.

Figure 3.8: Examples of races between a faulting thread, an interrupter, and the exception dispatcher thread. When a thread generates an exception, it is suspended by Vortex, and an exception message will eventually be delivered to the exception dispatcher thread. However, the faulting thread is still running from Casuar's point of view. The thread might therefore be awakened to receive interrupts, which results in two special cases (see above) that must be handled to balance the thread's suspend count.

It does so by acquiring the thread lock of that thread. Next, the exception dispatcher will assume that the thread is suspended, and try to retrieve its thread context. This operation will fail if the thread is not suspended, and if so, the worker will release the thread lock and simply discard the exception message. The reasoning behind this approach is that the thread can only be running if it was interrupted, and in this case, the thread will eventually return to the faulting instruction and generate another exception message that could be processed at a later time. To prevent the same exception from being handled more than once, the worker will compare the `RIP` from the thread context with the faulting `RIP` from the exception message. If these are not the same, it is most likely because the exception has already been handled, and the `RIP` has been advanced. In this case, the exception message will also be discarded. Otherwise, the worker will change the thread's state to a separate *exception* state before releasing the thread lock. This will prevent the thread from being interrupted or signaled while the exception is being handled. Afterwards, the exception dispatcher will grab the thread lock again, and change the thread's state back to *running*. However, now it also needs to check for pending APCs that could have been posted while the thread was in the exception state. If there are pending APCs, the thread will be resumed to execute the APC LEVEL interrupt handler, and if not, it will be resumed normally.

Note that the exception dispatcher will need to issue a resume operation for every exception message it gets, even if the message is discarded, in order to balance the initial suspend operation that generated the message. This means that the suspend count will be balanced *eventually*, even though it may be out of balance while a thread is executing an interrupt handler. In Casuar, this guarantee is sufficient for all practical purposes. However, another consequence is that the implementation of blocking waits will need to handle spurious wake-ups. This is because the exception dispatcher may discard old exception messages for a thread even while that thread is in a committed wait state, as is also shown in Figure 3.8b.

In general, to make sure that the suspend count of every thread is balanced properly, the implementation of blocking waits in Casuar needs to consider all possible sources to a resume operation, which are the following:

- The thread is resumed by the exception dispatcher, after discarding an exception message.
- The thread is signaled for unwait, alert, APC, or test of wait satisfaction.
- The blocking thread's timeout expires, and the thread is resumed by Vortex.

Note that the resume operation that is issued as part of the mechanism for interrupting a thread cannot lead to a wake-up of a blocking wait, since interrupts are only allowed when the thread is in a running state.

To determine whether a wake-up is spurious or not, the thread will acquire its thread lock immediately after waking up, and examine its own state. When a signaler awakens a thread from a committed wait state, it acquires the thread lock and changes the state of the thread to *running* before issuing the resume operation. Thus, if the thread is still in a waiting state after returning from the call to `vx_thread_suspend()`, and the wake-up was not caused by a timeout, then the thread must have been awakened spuriously. In this case, the thread will issue an extra resume operation to balance its own suspend operation, transition back to the running state, and then re-start the wait operation. Note that in the time between a thread is being signaled and it wakes up, it is not possible for the thread to receive any interrupts, although its state has been changed to *running*. This is because the thread will remain at DISPATCH LEVEL during the entire wait.

If the thread wakes up due to a timeout, the suspend count is self-balancing. However, it is possible that the timeout has been racing against a resume operation from a signaler, so the thread needs to check its state in this case as well. If the thread is in a waiting state, the suspend count is already in balance. On the other hand, if the thread is in the running state, then the thread has been signaled and needs to issue an extra suspend operation to balance the suspend count against the resume from the signaler.

Code Listing 3.8 shows the implementation of the function that is used for blocking a thread in Casuar, after the thread has reached a committed wait state. The code shows how the different wake-up scenarios are handled with regards to balancing the suspend count, as discussed above. Note that the return value of this function is the same as the return value from the call to `thread_commit_wait()` shown in Code Listing 3.7 (although we do not include a code listing for that function here).

3.4 Suspend and Resume

The last thing we consider in this chapter is how to implement functionality for suspending and resuming a thread, according to the semantics of the Windows system calls `NtSuspendThread()` and `NtResumeThread()`. In Section 3.2, we used these functions as examples of system services that directly depend on kernel APCs. After having described the functionality for performing blocking waits in Section 3.3, we are now able to explain how the suspend and

Code Listing 3.8: Implementation of blocking in Casuar. Certain details have been simplified or omitted for the sake of brevity.

```
NTSTATUS
thread_do_wait_suspend(vx_time_t timeout)
{
    winsys_thread_state_t   thread_state;
    vx_int64_t              wait_status;
    vxerr_t                 vxerr;
    irql_t                  wait_irql;

    wait_irql = CURRENT_THREAD->tcb_wait_irql;

    vxerr = vx_thread_suspend(CURRENT_THREAD->tcb_rid, timeout);

    thread_lock_acquire_at_dispatch_level();

    wait_status = CURRENT_THREAD->tcb_wait_status;
    thread_state = CURRENT_THREAD->tcb_thread_state;

    if ((vxerr == VXERR_OK) && (thread_state == THREAD_STATE_WAITING)) {
        /* We got a spurious wake-up.
         * Balance suspend count against our own suspend.
         */
        (void) vx_thread_resume(CURRENT_THREAD->tcb_rid);

        CURRENT_THREAD->tcb_thread_state = THREAD_STATE_RUNNING;

        /* Restart wait. Kernel APC flag indicates that wait is not
         * necessarily satisfied yet.
         */
        wait_status = STATUS_KERNEL_APC;
    } else if (vxerr == VXERR_TIMEOUT) {
        if (thread_state == THREAD_STATE_RUNNING) {
            /* We have had a race between timeout and intended resume.
             * Because timeout is self-balancing, this means that our
             * suspend count is not in balance, and we must compensate
             * for the extra resume.
             */
            (void) vx_thread_suspend(CURRENT_THREAD->tcb_rid,
                                     VX_TIME_NTIME);
        } else {
            /* Actual timeout */
            CURRENT_THREAD->tcb_thread_state = THREAD_STATE_RUNNING;
            wait_status = STATUS_TIMEOUT;
        }
    }          :

    thread_lock_release_to_dispatch_level();
               :

    return wait_status;
}
```

resume operations of a thread are implemented in the Windows Kernel. The implementation of the corresponding functionality in Casuar is practically the same.

Every thread has an associated suspend count, scheduler APC, and suspend event, which are contained in its `KTHREAD` structure. Unlike Vortex, a thread in Windows will remain suspended as long as the suspend count is positive. In addition, the suspend event will be signaled when the suspend count is zero, and otherwise be non-signaled.

When a thread is suspended, its suspend count is incremented. Then, if the suspend event is in a signaled state, its state is changed to non-signaled, and a scheduler APC is posted to the thread as a normal kernel APC. The event's object lock is used to synchronize callers to the suspend function, and the APC delivery mechanisms will prevent the scheduler APC from being queued more than once until it has executed. When the APC later runs in the context of the thread to be suspended, it will make the thread wait for its own scheduler event, using `KeWaitForSingleObject()`, `orthread_wait_for_single_object()` in the case of Casuar.

Correspondingly, a resume operation will decrement the thread's suspend count. If the suspend count is lowered to zero, the suspend event will be signaled, causing the suspended thread to be signaled for unwait. The thread will then wake up and return from the scheduler APC function, continuing its previous execution.

An interesting side effect of using APCs to implement suspend and resume operations, is that threads that are inside a critical or guarded region will implicitly be protected against unwanted suspension without having to use any additional protection mechanisms. Moreover, because the suspend and resume operations are implemented on top of the standard mechanism for blocking waits, all potential races that involve changes of the thread's state are already handled by that mechanism.

3.5 Summary

In this chapter, we have described a number of important thread synchronization and signaling mechanisms that constitute fundamental parts of the Windows Kernel. After evaluating each mechanism, we have detailed how it has been replicated in Casuar with focus on preserving the semantics of the abstractions as they are perceived by user-mode applications. In the following chapter, we will shift our focus to a higher abstraction layer, as we look at

the services and subsystems that the Windows Executive builds on top of the lower-level Windows Kernel, and how we implement corresponding services in Casuar.

4

Executive Services

Most of the functionality in the Windows NT kernel resides in the Windows Executive. The Executive is structured as a collection of larger components or subsystems, each providing a set of high-level services to applications. In this chapter, we give an overview of some of the most central executive components—the Object Manager, I/O Manager, and Memory Manager. For each, we also describe how we have implemented a subset of its functionality in Casuar to support commonly used system services. At the end of the chapter, we briefly summarize the functionality of some of the components that we do not implement in Casuar, and the reasons for not doing so.

4.1 Object Manager

Recall from Chapter 2 that the Windows Executive provides a large number of different abstractions that are made available to kernel-mode drivers and user-mode applications through Windows' system service interface. Almost all such abstractions are expressed as executive objects, and many of them represent shared system resources like files, devices, registry keys, and memory sections, which may be accessible from multiple processes running on the system. The different types of resources are managed by separate executive components; for example, files and devices are exported by the I/O Manager, whereas a registry key is an interface to the Configuration Manager. However, the executive objects themselves are managed by a single executive component—

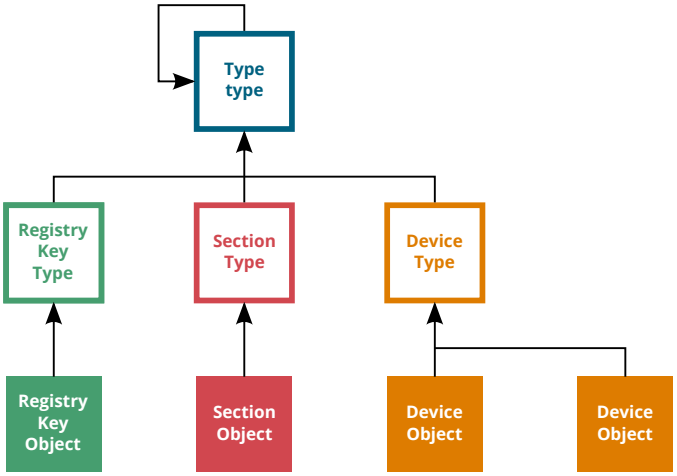


Figure 4.1: Object type hierarchy. All objects have an object type. Object types are also represented as objects. Their object type is the special *type-type*. Because the type-type is itself an object type, the Object Manager assigns it as its own object type.

the *Object Manager*.

The Object Manager is used to allocate and keep track of reference-counted executive objects on behalf of the other executive subsystems [46, Ch. 3]. Each subsystem defines and registers an *object type* for each different type of resource or abstraction through the Object Manager's interface. Associated with an object type is a number of callbacks that are implemented by the subsystem that registers the object type. The callbacks provide the interface between the Object Manager and the resource represented by the object type.

Every executive object is an instantiation of some object type. In addition, the object types are themselves represented as objects. Their object type is a special *type-type*, which is also its own object type. This is illustrated in Figure 4.1. Because of its dependency on itself, the type-type object is setup manually by the system as part of initializing the Object Manager.

The Object Manager also provides a uniform interface for accessing resources and abstractions provided by the other executive components from user mode [77]. Most of these resources are part of the global NT namespace, which is rooted in the Object Manager. The Object Manager organizes the namespace hierarchically using *object directories*, which contain pointers to executive objects or other directories. The directory structures also support symbolic links (or "symlinks"), to be able to expose more convenient object paths to the applications. This structuring of the NT namespace is illustrated in Figure 4.2.

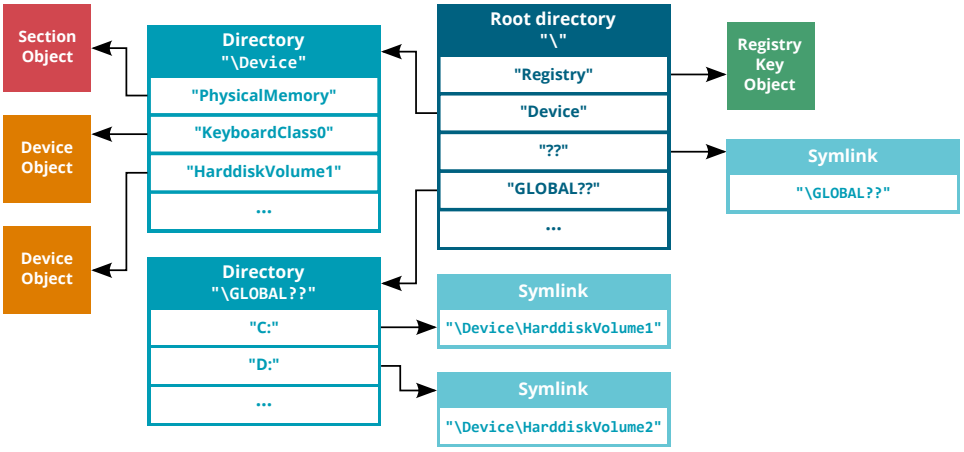


Figure 4.2: Hierarchical structure of the global NT namespace. The upper levels of the namespace are structured using object directories and directory symlinks that are setup by the Object Manager. The lower levels are implemented by the executive subsystems, through their registered object types. In the illustrated example, a lookup with the NT path "`\??\C:\Windows\System32`" in the object directories would first redirect "`\??`" to "`\GLOBAL??`", so the updated path would be "`\GLOBAL??\C:\Windows\System32`". Then, "`\GLOBAL??\C:`" would be translated to "`\Device\HarddiskVolume1`". Because "`\Device\HarddiskVolume1`" points to a device object, the lookup of the remaining path "`\Windows\System32`" would be performed in the sub-namespace implemented by that device.

All lookup operations to the NT namespace are centralized by the Object Manager. When an application requests access to a resource from user mode, it does so through some system service that is exported by the executive subsystem that implements the resource. For example, the system call `NtOpenFile()` is commonly used to open a file or directory in the file system, and is exported by the I/O Manager. However, the lookup operation for a resource is not handled directly by each subsystem [78]. Instead, the Object Manager centralizes all lookups to the NT namespace. Importantly, this allows all access control to be centralized as well. Although most executive subsystems export resources that differ significantly in their implementation, most of them depend on the same security mechanisms—the access rights of a process needs to be verified before it can be allowed to use a certain resource. During lookup operations, the Object Manager interacts with the Security Reference Monitor—the executive component responsible for enforcing security policies—to verify that the process has the necessary permissions to access the resource and may be granted all the requested access rights. If the process is given access, the Object Manager will then use the object type callbacks to let each subsystem handle the implementation-specific part of the lookup.

Because executive objects reside in kernel mode, applications running in user mode cannot be given access to manipulate these directly. Instead, Windows uses *handles* to refer to objects indirectly, similarly to how file descriptors are used in Unix-based systems. The Windows Executive and Object Manager maintains a *handle table* for each process, in addition to a separate handle table for kernel mode. Each time a process is given access to an object, an entry is allocated for that object in the handle table of the process. The handle table entry is used to contain a pointer to the executive object, together with the access rights that was granted for the object. Then, the system will construct a handle that refers to the entry, and return it to the process. Internally, the handle contains the index in the handle table of the allocated entry, but to the process, it is merely treated as an opaque structure that is used to interface with the object.

After a process has acquired a handle, by opening or creating a resource, it can supply the handle to other system calls to perform subsequent operations on that same resource. If the resource is a dispatcher object (see Chapter 3), and the handle has the sufficient privileges, the caller can synchronize with the object by supplying the handle as argument to either `NtWaitForSingleObject()` or `NtWaitForMultipleObjects()`. Both of these functions are exported as system services by the Object Manager. However, the Object Manager allows the caller to specify handles not only to dispatcher objects, but to other types of executive objects as well.

Most executive objects are not dispatcher objects, but rather embed one or more dispatcher objects—such as an event—in their structures for synchronization purposes. The Object Manager allows each object type to optionally specify a *default object*—a dispatcher object that is used to give objects of that type synchronization capabilities. The default object is typically expressed as an offset into the structure of an executive object, where a contained dispatcher object is located. This allows each object of a given object type to use its own, separate dispatcher object for synchronization. It is also possible for an object type to let all objects of that type share the same dispatcher object. When, for example, the `NtWaitForSingleObject()` procedure is called, the Object Manager will lookup the executive object for the supplied handle, and use the default object of the object or object type as argument to `KeWaitForSingleObject()`. This allows for a flexible and convenient interface to user-mode applications. For example, if an application supplies a handle to a file object, the event object that is embedded in the file object will be used internally to perform the synchronization on behalf of the file.

The handle table structures in Windows are designed to be highly performant under concurrent workloads that allocate and access entries [78]. In addition, the tables automatically scale up on demand. When a process is created, it

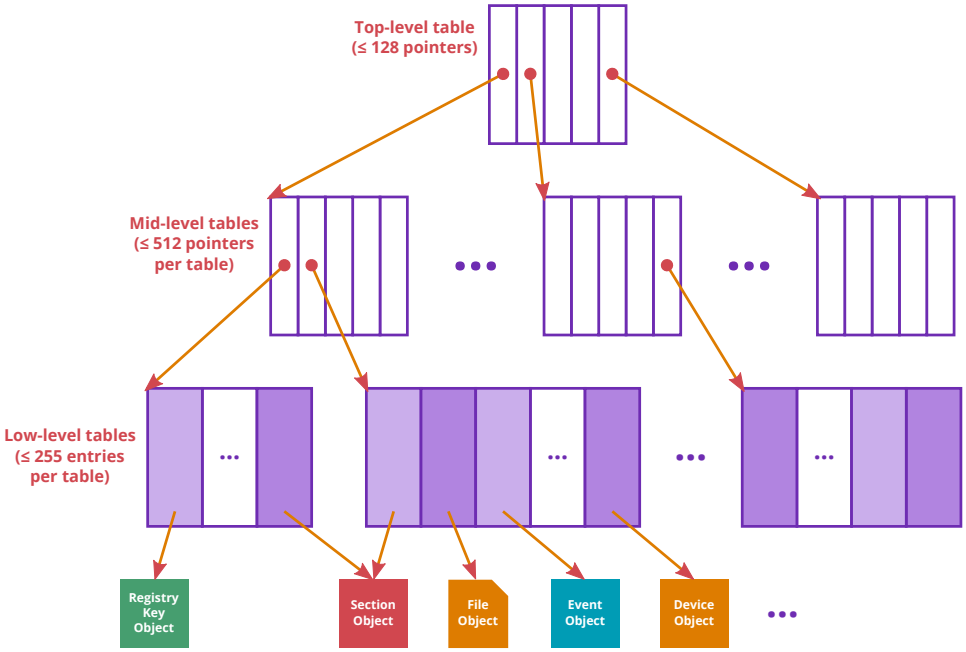


Figure 4.3: Overview of handle table structure. The handle table is expanded up to a three-level structure, on demand. The figure illustrates the case where the maximum of three level have been allocated. The top-level and mid-level tables contain table pointers, whereas the low-level tables contain actual entries. Each entry points to an executive object, and also contains the granted access right to the object, associated with the handle. As shown, it is possible for two or more handle table entries to point to same object, even with different access rights.

is given a single-level handle table. It consists of one memory page that can contain up to 255 entries, each occupying 16 bytes (on x64).[1] At the time when the last entry in that table has been allocated, Windows will expand the handle table to a two-level structure; a root page is allocated to contain pointers to low-level tables, the existing table is linked as the first low-level table, and a second low-level table is allocated to make another 255 entries available for use. With a two-level structure, the handle table can contain up to 127.5K entries. Similarly, the handle table may even be expanded to a three-level structure, allowing a little less than 16M handles to be referenced by each process. An illustration of the handle table structure is shown in Figure 4.3.

The object type callbacks that are supported by the Object Manager include a *delete* method that is called whenever the last reference to an object in the

1. Each page has room for 256 entries, but the first entry in each low-level table is used as a special *leaf* entry, which contains the base handle index of the respective low-level table.

system is given back to the Object Manager (i.e. the object's reference count is lowered to zero), and *open* and *close* procedures that allows a subsystem to respond when a handle to an object is created or destroyed. One of the most important callbacks is, however, the *parse* procedure, which is used to open named resources from the NT namespace.

The Object Manager supports both named and unnamed objects. Named objects are those that may be part of the NT namespace, whereas unnamed objects cannot. More specifically, a named object is pointed to by some object directory entry. A file object, for example, is *not* a named object. Even though it represents a file path that is part of the NT namespace, no file objects are directly referenced by any object directory. Instead, they are part of the sub-namespace implemented by a device object (e.g. a harddisk volume). A device object, on the other hand, is a named object that typically represents the root of its own namespace. The object must be named for its namespace to be part of the global NT namespace.

All object types that support either named objects or objects that may represent a directory abstraction in a sub-namespace, are required to implement a parse procedure. The only exception is the object directory type, since the lookup in object directories is handled directly in the Object Manager. Figure 4.4 illustrates the process of looking up an object in the NT namespace from a given path. Details that relate to access control and interaction with the Security Reference Monitor has not been included in the figure.

4.1.1 Implementation of an Object Manager in Casuar

Casuar implements an Object Manager component that corresponds to a subset of the functionality offered by the Object Manager in Windows. It has full support for allocation and reference-counting of objects, creation and registering of object types, and setup and allocation of handle tables and handles on behalf of processes. It also includes the necessary object directory and symlink structures for structuring an NT compatible namespace, together with procedures for parsing NT paths and performing lookup of objects.

However, Casuar's implementation does not include any form of access control, which is a significant part of the Windows implementation. The most important reason for not implementing any security mechanisms, is that it is extremely difficult to do properly and correctly, and would widen the scope of our work significantly. Moreover, since our system is built with the main focus being on application-level virtualization, we assume that the processes that will run on top of our system do not behave maliciously. In addition, Vortex already provides mechanisms for isolating applications from each other. In the case where two

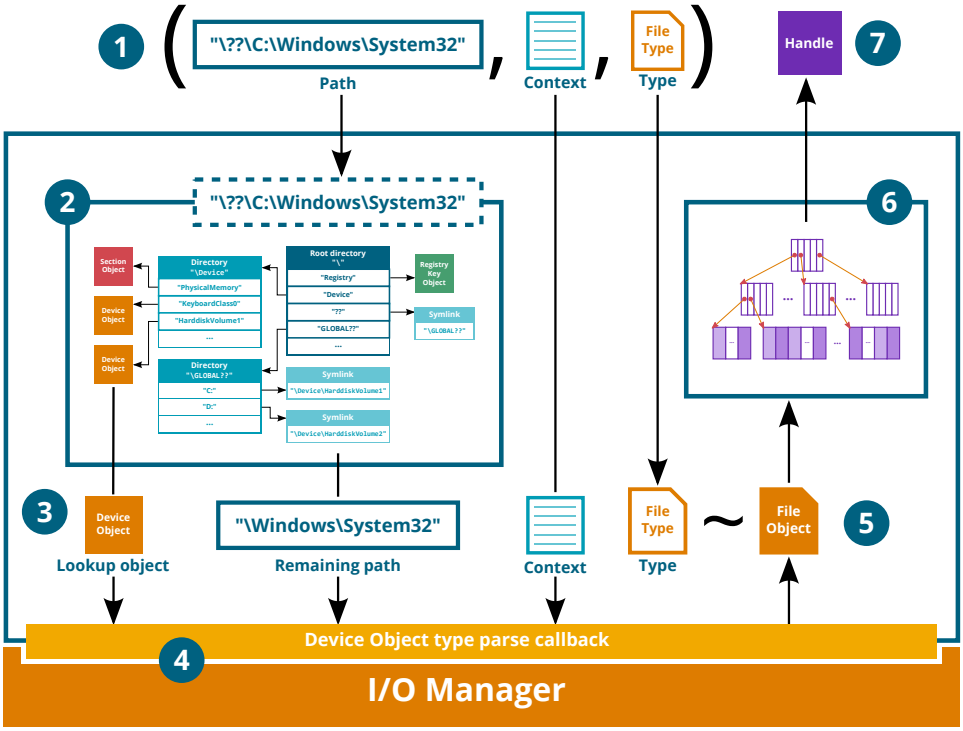


Figure 4.4: Lookup of objects in the NT namespace.

❶ The caller specifies the path to look up, a subsystem-specific context pointer (optional), and the expected object type. In the shown example, the caller opens the file system directory "`\??\C:\Windows\System32`".

❷ The Object Manager traverses the path, following symlinks, until it finds a directory entry that points to an object (instead of another directory or symlink). The object and the remaining path is returned from the internal look-up procedure. In this example, the path prefix "`\??\C:\`" is resolved to "`\Device\HarddiskVolume1`", which points to a device object for that volume.

❸ The object, remaining path, and context is passed to the parse callback of the object's object type.

❹ The parse procedure executes, and interacts with the executive subsystem that registered the object type. In this example, the lookup for the remaining path is handled by the I/O Manager.

❺ If the lookup succeeded, an executive object is returned from the parse callback. This object might have been allocated by the parse procedure, using the appropriate Object Manager routines. In this example, this is a file object representing the "`System32`" file directory. The Object Manager makes sure the returned object has the same object type as requested by the caller. If it does not, the lookup fails.

❻ The Object Manager allocates a handle for the returned object in the handle table of the process.

❼ A handle to the object is returned to the caller.

or more applications need to be separated in different security domains, they could be setup to run on different instances of Casuar. Hence, they would be unable to cause negative effects on other processes. A consequence, however, of not supporting any security mechanisms, is that Casuar will not be able to support any advanced Windows applications the depend on or use security features of Windows directly, such as interfacing with user management or querying access control lists.

There are also other aspects of the Object Manager in Windows that do not apply to our implementation. For example, it is used to manage charging of resource quotas according to, for example, memory usage. In Casuar, this is already handled by Vortex, as part of the omni-kernel design.

4.2 I/O Manager

The I/O Manager in Windows provides the interfaces for performing I/O operations to most devices in the system. It defines the *device*, *driver*, and *file* object types and their interactions with kernel-mode drivers, user-mode applications, and the other executive components. Almost all I/O operations go through the file abstraction, where a file object may represent anything from a regular file in a file system, to a named pipe or network connection. The system calls `NtOpenFile()` and `NtCreateFile()` are used to open or create file objects, and the functions `NtReadFile()` and `NtWriteFile()` constitute the interface for performing the actual read and write operations to the file.

A device object may represent either a physical hardware device, such as a hard drive or network interface card, or an abstract, higher-level resource, such as a file system or TCP connection. Driver objects similarly represent kernel-mode drivers that provide the interfaces between the I/O Manager and the devices. Devices and matching drivers are typically organized in layered I/O stacks for each separate resource, where each location in the stack corresponds to a separate abstraction level—higher-level abstractions such as file systems are built on top of lower-level abstractions such as hard drives [79].

The I/O model implemented by Windows is completely asynchronous. Whenever a driver or application initiates an I/O operation, the I/O Manager allocates an I/O request packet (IRP) for that request. The IRP is dispatched to the driver at the top of the I/O stack associated with the file object, and will typically be passed downwards through the stack. Associated with the IRP is an *I/O stack location* structure [80] for every layer in the stack, each containing the arguments to the respective driver for the I/O operation to be performed. At each layer, the driver may either complete its processing directly, or defer the

completion to a later time if it has started an asynchronous operation that is currently pending. It is also possible for the driver at each layer to allocate new IRPs, which will represent smaller I/O operations that are part of the larger operation. In this case, the IRPs will be chained to each other and to the initial IRP—the so-called *master IRP*. The master IRP will only complete once all subordinate IRPs have completed.

Each IRP is assigned a *major function code* [81] that specifies the main type of operation to be performed. When a driver registers itself in the system, it specifies a *dispatch routine* callback for each major function code it supports. For example, every driver must implement a dispatch routine for the `IRP_MJ_CREATE` major function code, which is used to respond to an open or create operation [82]. Other examples are `IRP_MJ_READ` and `IRP_MJ_WRITE`.

When an asynchronous I/O operation finishes, a special kernel APC associated with the operation's IRP will be queued to the thread that initiated the operation. This is done so that the necessary I/O completion actions will be performed in the context of that thread. The driver at each stack location in the I/O stack may register a *completion routine* that will be invoked when the IRP completes. Drivers may also optionally allow IRPs to be canceled [83]. In this case, a *cancel routine* is associated with the IRP. The system ensures that an IRP may only be canceled if it has not already completed; either the cancel routine or the completion routine will be called, but not both.

The I/O Manager may attempt to cancel an IRP, if the operation takes too long to complete and an internal timeout expires. The application that initiated the I/O operation may also request that it be canceled, through the `NtCancelIoFile()` and `NtCancelIoFileEx()` system calls. In addition, when a thread terminates, all its pending I/O operations must be either canceled or completed. To support this behavior, the I/O Manager enqueues every IRP that is created on behalf of a thread to an IRP queue that belongs to that thread, before the IRP is dispatched to a driver. When the IRP is canceled or completes, it is removed from the queue. If a process requests that all pending IRPs belonging to a specific thread should be canceled, the I/O Manager will use the IRP queue to locate the corresponding IRPs.

When an application opens or creates a file, it may optionally specify that all I/O operations are to be performed synchronously against the file. Also, if a file was opened for asynchronous I/O, the caller may wait for any single I/O operation to complete. In both cases, the `KeWaitForSingleObject()` wait procedure is used to perform the synchronization (see Section 3.3). The procedure is either called directly by the I/O Manager, or by the application through the `NtWaitForSingleObject()` system call, depending on the type of I/O operation and which mode the file was opened in. Similarly, the synchronization may be

done either using the file's built-in event object, or a user-supplied event.

Some I/O routines, such as `NtReadfile()` and `NtWriteFile()` allow the caller to specify a completion routine that will be invoked in context of a user APC after the I/O operation completes successfully. If the file was opened for synchronous and alertable I/O, the I/O Manager will deliver the user APC before returning from the I/O routine. Otherwise, the APC is delivered the next time the caller performs an alertable wait operation from user mode.

4.2.1 I/O in Casuar

Vortex already provides an extensive I/O interface to Casuar. The Vortex omni-kernel implements file system and networking support, and Vortex also provides a user-mode library to its applications for simplifying the use of asynchronous I/O. Different I/O resources are exposed to the applications as part of Vortex' namespace. For example, the file system resource is available from "`/fs`", and TCP endpoints can be accessed through "`/network/tcp/server`" and "`/network/tcp/client`".

In Casuar, we have implemented a small part of the functionality corresponding to the I/O Manager in Windows. We currently support I/O only against regular files and directories in the file system, and we do not support cancellation of I/O operations. However, to make the implementation extensible to allow for future additions, our implementation relies on many of the same abstractions as used in Windows' I/O model—such as devices and IRPs. Because all I/O operations in Casuar are built directly on top of the software interfaces provided by Vortex and its user-mode library, there is no need for a layered I/O stack or a separation between devices and drivers. We have therefore merged the concepts of a device and driver into a single device abstraction. Similarly, we do not represent I/O stack location structures, and instead embed the I/O operation arguments to the device directly in our IRP structure.

To natively integrate the file system of Vortex into the NT namespace, we have abstracted the interaction with Vortex' namespace as a `VortexNS` device. In Windows, most applications interface with the file system through system drive letters, such as "`C:\`" and "`D:\`". These drive letters are represented as entries in the "`\GLOBAL??`" object directory, and are mapped to devices such as harddisk volumes via object directory symlinks (see Figure 4.2 from Section 4.1). In Casuar, we exploit this structure, and similarly setup drive letters to map to "`\Device\VortexNS`" instead.

Specifically, we have setup the custom drive letters "`L:\`" (for libraries) to point to "`\Device\VortexNS\fs\lib`", and "`A:\`" (for applications) to point to

"`\Device\VortexNS\fs\app`". The first is used as the system drive, where we store all system files (such as DLLs) that a Windows application depends on, and the second is used to separate the applications from the libraries. When, for example, an application requests access to "`\??\L:\Windows\System32`", the Object Manager will make sure that the request is redirected to the `VortexNS` device. Moreover, using this setup, the Object Manager will pass "`\fs\lib\Windows\System32`" as the remaining path to the `VortexNS` device, which means that the backslash characters in the path simply need to be substituted with forward slashes to yield the corresponding Vortex namespace path.

4.3 Memory Manager

The Memory Manager in Windows is responsible for implementing virtual memory and managing the address space of each process [84, Ch. 10]. It divides the physical memory into separate memory pools, and provides memory heaps on top to allow other kernel-mode components to allocate and free memory dynamically. Moreover, it provides support for protecting memory pages with different access rights, locking pages, and swapping memory to paging files. Implementation-wise, there are many features in the Memory Manager that we will not describe here. Instead, our focus is on the interface that it exports to user mode, and which we need to support in Casuar.

From the perspective of a user-mode application, there are three main services that the Memory Manager provides through its system service interface:

- Allocation and freeing of *anonymous memory*, and protection of memory. This corresponds to system calls such as `NtAllocateVirtualMemory()`, `NtProtectVirtualMemory()`, and `NtFreeVirtualMemory()`.

- Creation and mapping of *section objects*, which represent either shared memory or memory-mapped files. This corresponds to system calls such as `NtCreateSection()`, `NtOpenSection()`, `NtMapViewOfSection()`, and `NtUnmapViewOfSection()`.

- Querying existing mappings or sections. This corresponds to system calls such as `NtQueryVirtualMemory()` and `NtQuerySection()`.

When an application performs a request for anonymous memory, it specifies whether the memory is to be *reserved* or *committed* [85]. If a page is reserved, it is marked as allocated in the address space of the process. However, the page is not yet bound to physical memory and therefore cannot be accessed. On the

other hand, if a page is committed, it will be backed by physical storage. A free page may either be committed directly, or it may be reserved first and then committed at a later time.

Section objects are used as abstractions for portions of memory that can be mapped into and shared between one or more address spaces [86], [84, Ch. 10]. A section is either file-backed, page-file-backed, or it corresponds to shareable anonymous memory [87]. The Windows Image Loader, implemented in `ntdll.dll`, also uses section objects to represent prototypes for memory-mappings of DLLs. Every Windows DLL is structured according to the Portable Executable (PE) image format. It typically consists of multiple segments—for code, data, metadata, etc.—that will be unpacked to different memory locations when the DLL image is loaded into the address space of a process. A single section object is always used to describe the mapping of each DLL. The Memory Manager therefore structures every file-backed section as a collection of subsections, each pointing to a different byte range inside the file. For image files, there is one subsection for each PE segment, plus an additional subsection for the PE header that is present at the start of every DLL. Section objects for regular files, on the other hand, typically consist of only a single subsection.

A process can map one or more *views* of a section into its address space. If the section is backed by an image file, the subsections will determine which file ranges are mapped to which memory locations, relative to a base address for the view. Otherwise, if the section is backed by a regular file, the caller may specify which range in the file should be mapped into the view [88].

In Casuar, most of the functionality required to implement memory management on behalf of Windows applications is already present from Vortex' side. Vortex exports the system calls `vx_mmap()` and `vx_munmap()`, which provide powerful abstractions for allocating and protecting memory regions. However, Vortex lacks functionality for querying the system about the state of existing memory mappings, and does not support shared memory.

To be able to handle memory queries, we associate a memory descriptor list with each process, which is updated every time a user-mode memory mapping is created, changed, or destroyed. Without support from Vortex, however, we are unable to include any functionality for shared memory sections. Hence, the only type of section objects we implement are file-backed sections. Finally, Casuar includes a PE parser component to support the creation and mapping of section objects for PE image files, as is needed by the Image Loader. The implementation of this parser, however, has already been covered in previous work [41].

4.4 Other Executive Components

There are many other components in the Windows Executive that provide important functionality in Windows, but for which we do not implement equivalent mechanisms in Casuar. Some, because they are not relevant to Casuar, and others, because they are outside the scope of this thesis and have been deferred to future work. We provide a brief overview here of a few of the most central components or mechanisms that we have left out.

Process Manager Casuar needs to manage processes and threads to be able to host Windows applications. However, most of the required functionality is already provided by Vortex, through its high-level process and thread abstractions. Additional mechanisms that are involved as part of starting user-mode threads are implemented directly in corresponding system services. Hence, there is no need for a separate Process Manager component in Casuar.

Security Reference Monitor As briefly described in Section 4.1, the Security Reference Monitor is the central component responsible for enforcing security policies in Windows [46, Ch. 6]. Its security model is mainly based on configurable access control lists, which apply to both the file system and other subsystems such as the registry. Casuar does not implement any functionality for access control, and depends on mechanisms implemented by Vortex for isolating applications from one another. This comes at the cost of not being able to support applications that depend explicitly on the interface exported by the Security Reference Monitor.

Configuration Manager The Configuration Manager is the executive subsystem responsible for implementing the Windows Registry [46, Ch. 4]. As the registry is frequently used by many Windows applications, Casuar will likely need to be extended to include a registry component in the future.

Transaction Manager Starting with NT 6.0, the Windows Executive was extended to include support for Transactional NTFS and Transactional Registry [89]. The transaction support is extensively used by the Windows Update components when applying hotfixes or updating the system with service packs. This allows the system to protect itself from corruption to a larger degree, if a failure should occur while modifying system files or registry keys. However, there are not many other Windows applications that depend on this transaction support, so at this point we deem it highly unlikely that a Transaction Manager would be needed in Casuar.

Advanced Local Procedure Calls (ALPCs) ALPC is the main mechanism for inter-process communication in Windows [46, Ch. 3]. It is, among other

things, used for communication between the Windows Subsystem process (`csrss.exe`) and the processes that run on top of the Windows Subsystem. The reason for not implementing functionality for ALPC in Casuar is because the mechanism is based on shared memory, which Vortex currently does not support. It is, however, one of the important mechanisms that must be implemented to be able to support regular Windows applications.

Networking Casuar currently lacks network support, which is a drawback. Properly implementing it, however, requires a lot of work even though Vortex has full support for TCP and UDP, which is why it has been deferred to future work. The main reason is that the network interfaces in Windows depend on support from several different kernel-mode drivers, which interact with system DLLs in user mode [46, Ch. 7]. For example, the DLLs that implement *Winsock*—Windows' implementation of Berkeley sockets—explicitly try to load the `ws2ifsl.sys` driver through corresponding system calls, and this driver will call back into the DLLs [90].

Cache Manager The Cache Manger is responsible for implementing functionality such as the file system cache, and ensuring that section views are consistent, both across different address spaces and with the underlying storage (in the case of file-backed sections). The functionality of the Cache Manager corresponds partly to functionality that is already present in Vortex, and partly to functionality that Casuar does not yet support (the latter especially concerning section objects).

PnP Manager and Power Manager The Plug and Play (PnP) Manager and the Power Manager are both responsible for managing hardware resources. Hence, they are completely irrelevant to Casuar, because Vortex will abstract all hardware as resources on behalf of its applications, and the resources are only available to them through the system call interface.

4.5 Summary

This chapter detailed some of the most essential components of the Windows Executive, and their high-level services that are exposed to applications. We also described how Casuar selectively includes parts of their functionality, to provide an adequate implementation of a few of the most commonly used system services provided by these components.

In the next chapter, we describe the methodology and techniques that we have used to complement the mechanisms from this chapter and the previous

chapter with the necessary functionality to let Casuar support the loading and execution of Native applications.

5

Achieving ABI Compatibility

In the previous two chapters, we identified some of the major components and mechanisms in Windows NT that are essential to supporting the execution of a Windows application. We have given an overview of how each is implemented in Windows, and have described the approaches we have taken to provide alternate implementations in Casuar. However, this functionality is far from sufficient to actually load and run Windows applications as processes on top of Casuar. The application binary interface (ABI) that an application depends on consists of other, smaller parts that must be supported as well.

To achieve ABI compatibility with a Windows application, it is necessary to identify and correctly implement all the functionality that it needs to run. This is quite challenging, because most parts of the ABI are undocumented. In this chapter, we describe a methodology that we have devised for driving out the necessary functionality, based on blackboxing the behavior of an application. We also present a number of techniques that we use to aid our implementation effort. Finally, we demonstrate that by combining the implementation from the previous two chapters with the methodology and techniques from this chapter, we are able to run Native applications on both Windows and Casuar.

5.1 Basic Approach

The Windows Kernel and Windows Executive export a little over 400 different system calls [41]. Most Windows applications depend on merely a small subset of these. Still, we know with certainty that the components and mechanisms described in Chapter 3 and Chapter 4 do not account for all of the functionality that is needed to run an application. Additional support is required for the Windows Image Loader to be able to perform the necessary initialization of a process, before it even starts to run the application code.

The lifetime of a user-mode process may be divided into two phases—the *loading phase* and the *execution phase*. After the Windows Executive has created and initialized the process in kernel mode, the main thread of the process starts executing in user mode from the function `LdrInitializeThunk()` in `ntdll.dll`. This function is used to invoke the Windows Image Loader, which will perform necessary preparation of user-mode data structures that are specific to the process and its threads. It is used as the effective entry point of every user-mode thread. After the image loader has completed its work, the thread will begin to execute code from a user-defined entry point. If it is the main thread, it will use the application-wide entry point from the application's executable. The execution phase begins when the main thread starts to run the actual application code, and it lasts until the process terminates.

The work that is done by the Windows Image Loader includes creating a process heap, setting up thread-local storage (TLS), and initializing the process environment block (PEB) and thread environment block (TEB) structures accordingly. As indicated by its name, the loader will also make sure all DLLs that the application depends on are loaded into the address space of the process, and execute any DLL-specific initialization code as needed. The loading phase cannot be bypassed, because other DLL functions that are used by an application depend on the structures that are initialized by the loader. To support the execution of a Windows application on top of Casuar, our primary focus has therefore been on implementing the necessary functionality that is required by the loader. After the loading phase completes, any additional functionality needed to support an application depends on the specific behavior of that application.

To be able to support the execution of the image loader, it is necessary to implement additional system calls. However, it is *not* desirable to evaluate every single system call up-front to decide whether to implement it or not. This would be very time-consuming, especially since most system calls are undocumented, and even their general behavior might be unknown. In addition, we have very little a priori knowledge concerning the details of the loading phase. Available literature [46, Ch. 3] only presents a broad description, from which we cannot

infer much about the loader's exact interaction with the system. This indicates that we should merely rely on the execution itself to determine which system calls and other functionality to implement. The notion is strengthened when considering the work that is needed to support an application's execution phase, where we cannot assume that any implementation details are known whatsoever.

As a result, we treat all user-mode code as a *black box*, and adhere to the following basic approach for driving out the functionality that is needed to run a Windows application:

1. We start by preparing the execution environment of a process, based on our current knowledge.

2. Next, we start executing user-mode code from the `LdrInitializeThunk()` function.

3. At some point, the image loader will try to perform an operation that depends on a feature that has not yet been implemented. The operation will interact with the system through some interface where we should be able to detect the dependency.

4. We identify and evaluate the missing feature.

5. Then, we respond to the missing feature by implementing sufficient functionality to support the user-mode request, and make changes to the execution environment as necessary.

6. Finally, we re-start the execution, and continue to iterate this process until the loading phase completes successfully.

There are, however, some challenges to this approach. First, we need to be able to *detect dependencies* on unimplemented functionality at a fine-grained level. For system calls, this is straightforward, because every system call will generate a trap to Casuar that must be handled explicitly. However, Windows applications depend on more than just system calls. System DLLs, such as `ntdll.dll`, also have dependencies on several user-mode data structures that are part of the address space of the process, such as the PEB and TEB. These structures are not only initialized by the loader; the DLLs also rely on the structures to have been partly initialized by the kernel, even before the process starts executing user-mode code. Hence, they may be considered part of the application binary interface (ABI) between user mode and kernel mode. As with system calls, we do not want to evaluate every possible structure member. A major challenge is therefore to determine which structure fields are actually being accessed from

user mode—and hence must be initialized by Casuar—and what values these fields should assume.

Second, it may be difficult to identify and evaluate a missing feature, even though we know the name of the system call or structure member that has not yet been implemented or initialized. Again, most system calls and structure fields are undocumented. Providing a sufficiently working and correct implementation might in itself require blackboxing. A central question is how to obtain enough information to know how a new feature is supposed to be implemented or expected to behave.

Third, because our above approach is based on successive refinement, it is hard to make progress without a clear measure of progress. Specifically, we need to be able to determine whether adding or modifying functionality has allowed the loader or application to continue its work towards successful completion. If, for example, a change in our system has introduced erroneous behavior, the image loader could start executing error-handling code. This should be clearly noticeable for us to be able to fix such an error.

Throughout the following sections, we describe how we have taken on the above three challenges. In Section 5.2, we describe our implementation of a mechanism that allows us to monitor every memory access that is made to a user-mode data structure, thus overcoming the first challenge. Following that, Section 5.3 explains how we have implemented functionality for producing stack traces, which may provide additional context for dealing with the other two challenges. Finally, we demonstrate in Section 5.4 that, through the use of the methodology and techniques described in this chapter combined with the implementations from the previous chapters, we are able to run Native applications on top of both Windows and Casuar.

5.2 Monitoring Memory Accesses to User-Mode Data Structures

In Windows, there are mainly three user-mode data structures that must be initialized before a process starts executing from the `LdrInitializeThunk()` entry point. These are the PEB and TEB structures, which are specific to each process and thread, and the `KUSER_SHARED_DATA` structure, which is shared globally between all processes. In the first two structures, only some of the members need to be assigned values by the kernel. Other members are setup either by the Windows Image Loader or by initialization code in other system DLLs. In contrast, the third structure is setup entirely by kernel-mode

code.

Determining *what* value to assign to each structure field largely involves trying to deduce the *meaning* of the field, based on its name. In addition, we also need to figure out *which* fields must be set. The structures are quite large; for example, the TEB contains more than 6 KB of data, spread over more than 100 structure fields. Hence, we want to identify which fields are actually accessed by an application, and only assing values to those that are expected to be initialized by the kernel. However, memory accesses made by user-mode code are not directly visible to kernel mode because, unlike system calls, they are not explicit interfaces to the kernel. It is therefore necessary to use some kind of instrumentation to be able to trace memory accesses.

Specifically, we want to let every access within a given memory region—corresponding to a user-mode structure—cause a trap to kernel mode, similar to how system calls behave by default. This way, the kernel will be able to log all relevant memory accesses, so we can use this information to determine offline which structure members must be initialized when preparing the execution environment of a process. If, during the loading phase, a field is *written* to without being previously read, we can determine with certainty that the field is initialized by the loader. This means that we do not need to assign a value to that field in Casuar. Conversely, if the first access to a structure field is a *read* operation, this indicates a dependency on a value that must have already been set by the kernel.

Our main idea is to exploit the paging mechanism on x64 and force generation of page faults for all accesses within certain memory pages. Vortex allows Casuar to memory-map pages with either supervisor or user access rights. If a thread tries to access a supervisor page from user mode, the memory management unit (MMU) will generate an access fault, and the thread will trap to the Vortex kernel. Normally, such a fault will cause the process to be terminated. However, recall from Chapter 3 that Vortex allows a process to handle exceptions on behalf of its threads. If the process has subscribed to exception messages, the offending thread will instead be suspended, and the process is notified about the exception. In Casuar, we have turned this mechanism on for all processes, so we are able to handle page faults.

The typical way to handle an exception is to make appropriate changes to the system or process environment, such that the underlying cause for the exception is resolved. Then, when a thread resumes its execution after the exception has been handled, it will execute the previously faulting instruction again. Unless the instruction causes another exception, the thread will be able to proceed. However, to be able to monitor all memory accesses to a page, we have to make sure that every access results in a trap. If the access rights of the

page are changed to allow the faulting memory instruction to succeed, then subsequent accesses to the page will also succeed without generating a page fault. On the other hand, if we do nothing in response to the fault, the thread will continue to fault on the same instruction indefinitely.

Our approach is to leave the memory mappings as is, and instead perform the memory access from kernel mode, on behalf of user mode. This is possible, because the memory pages that have been protected with supervisor rights will be accessible to the exception handling code executing in kernel mode, without causing another exception. However, the faulting instruction cannot simply be re-executed in kernel mode. Instead, we have created a mechanism for emulating the effect of the instruction, so it behaves to user mode as if the instruction executed normally on the CPU.

When a thread generates an access fault to a monitored memory page, we decode the instruction byte code from the faulting RIP, emulate the memory access corresponding to that instruction, and finally advance the RIP beyond the instruction before resuming the thread. When the thread wakes up, it will continue to execute code from the next instruction, instead of restarting the faulting instruction. Using this technique, we effectively virtualize a user mode thread's memory accesses; threads will be oblivious to the fact that some instructions are emulated instead of executed by the CPU. The net result is that Casuar is able to log every single access to the pages it is monitoring.

To do the actual emulation of memory instructions, we implement a *memory instruction emulator* component with functionality for decoding a subset of the x64 instruction format. We only support instructions that access memory, and we implement only instructions that are actually needed to handle accesses to the monitored data structures. Table 5.1 summarizes the instructions we currently support.

Most memory-related x64 instructions take two operands—a memory operand and either an immediate value or a CPU register. The memory instruction emulator performs memory access operations corresponding to the memory operand, and uses the thread context of the faulting thread to read from and write to its registers. In addition, it makes sure that other side effects of an instruction are faithfully emulated. For example, many arithmetic instructions update the RFLAGS register as well as the destination operand of the instruction.

However, the emulator does not access the memory within the monitored pages directly. Instead, it is modeled to interact with a memory interface consisting of read and write callbacks. The callbacks are supplied by a separate *memory monitor* component. The memory monitor allows the system to register a range

Table 5.1: Number of implemented instructions in x64 memory instruction emulator.

Mnemonic	Meaning	# supported opcodes
`and`	Bit-wise logical AND	3
`btr`	Bit test and reset	1
`bts`	Bit test and set	1
`cmp`	Compare	5
`dec`	Decrement	1
`inc`	Increment	1
`mov`	Copy	5
`movzx`	Copy zero-extended	2
`movsx`	Copy sign-extended	1
`or`	Bit-wise logical OR	4
`test`	Test for set bits	5

of memory pages that will be monitored, together with a handler that is specific for each range. Each user-mode data structure to be monitored will be mapped to a different set of pages, and will register a handler with the monitor. The handler has a separate set of callbacks for logging read and write accesses to its associated page range.

When the system receives an exception message for a page fault, the memory monitor will demultiplex the exception using the fault address. If the fault address lies within one of the registered page ranges, the memory monitor will locate the associated handler. Next, it invokes the memory instruction emulator to decode the faulting instruction. Emulation of the instruction will result in at least one call to the read or write callback of the memory monitor's supplied memory interface. The memory address to be accessed is calculated from the instruction code and, possibly, the value of some register from the faulting thread's register context. When either of the memory monitor's callbacks is invoked, the monitor takes responsibility for performing the corresponding memory access. Then, it translates the memory address to an offset, relative to the base address of the located handler's page range. Finally, the memory monitor invokes the read or write callback that belongs to the handler (depending on the type of memory access), and supplies the offset as argument. This allows the handler to map the offset to the name of a member in the data structure represented by the handler, and log the access. Figure 5.1 illustrates the overall interaction between the memory instruction emulator, the memory monitor, and the monitor handlers.

Every time the user-mode code tries to access a field from a monitored data structure, the access will be registered by one of the loggers. However, if it is a

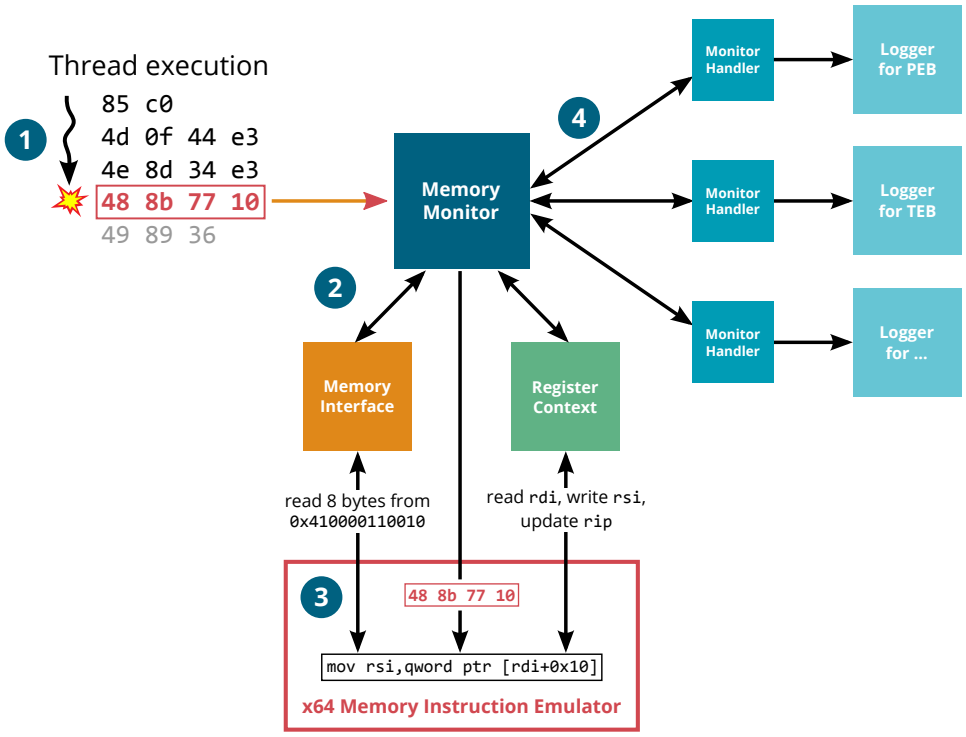


Figure 5.1: Casuar's memory monitor architecture.

❶ When a thread generates an exception from user mode, the exception is forwarded to the memory monitor.

❷ If the exception was a page fault within a monitored page, the memory monitor will handle the exception and invoke the memory instruction emulator. The monitor passes the faulting thread's register context and a memory interface (consisting of read and write callbacks) to the emulator.

❸ The memory instruction emulator decodes the faulting instruction, and emulates its effect by updating the register context and interacting with the memory interface.

❹ When the emulator issues read or write operations to the memory monitor's memory interface, the memory monitor will perform the corresponding memory access. Then, the handler that owns the monitored page is notified about the access, so the access can be logged.

read operation to a field that should have been initialized by the kernel, it does not make sense to allow the faulting thread to proceed afterwards, since the continued execution will depend on the field to have been setup properly. For this reason, all loggers are configured to allow accesses only to a predetermined set of structure members. If a logger detects an access to a field that is not in this set, it will log the access and then halt the system. Note that this only happens when we need to implement some missing feature or assign an initial value to a previously unassigned structure field. Afterwards, we manually add the field to the set of structure members that are allowed to be accessed. From this approach, we get full control over which fields are accessed, and can be completely confident that we detect all dependencies on those fields.

5.3 Using Stack Traces to Provide Context

Via the mechanism described in the previous section, we are able to detect all user-mode dependencies on both system calls and data structure members. Moreover, we also know the *names* of most system calls and structure fields, because they are either exported from the relevant system DLLs, or they are public information. However, system calls and structure fields are merely interfaces to underlying features of Windows NT. As these are for the most part undocumented, their names might not by themselves yield enough information that we are able to implement equivalent functionality in Casuar.

Gaining more insight about some unimplemented feature is a matter of acquiring additional information about the *context* in which the feature is used. As pointed out in Section 5.1, being able to do this will also aid our evaluation of whether a corresponding feature in Casuar has been implemented correctly; if we have more information about *why* a system call is invoked or a structure member is accessed, it is possible to infer whether the user-mode code behaves as expected or not.

When a thread traps to the Casuar kernel, the `RIP` of the user-mode instruction that caused the trap will be available to the kernel. This, by itself, provides a little bit of information; for a system call, the `RIP` will be within the `ntdll.dll` function that implements the corresponding system call stub and therefore also specifies the name of the system call, and for accesses to structure fields, the `RIP` may reveal from which function the access originated. However, if the `RIP` is considered together with the `RSP` from user mode, it is possible to construct a complete traceback of the call stack containing all preceding function calls—a so-called *stack trace*.

The implementation of Windows relies heavily on structured exception han-

dling (SEH)—its native mechanism for handling exceptions, both in kernel mode and user mode [91]. There are some similarities between SEH and the built-in exception support of C++ [71]. SEH allows C functions to use special `__try` and `__except` keywords for executing code that might generate exceptions, and handle them accordingly. This also means that Windows requires special support from the Microsoft C compiler to be able to implement SEH [92].

When an exception occurs, the native exception handling code in Windows will look for an exception handler (corresponding to an `__except` block in the original, compiled C code). If the function that triggered the exception does not handle it, the thread will return to the caller of that function and the system will query it for an exception handler instead. The exception will continue to bubble up the stack until the system either locates a matching exception handler or, if the exception is unhandled, the process is terminated. This means that the exception handling code needs to be able to *unwind* the call stack in a consistent manner, regardless of where in the code the exception occured.

The layout of the call stack in x64 Windows is dictated by the Microsoft x64 ABI [93]. Each time a function is called, the `RIP` is pushed onto the stack as a return address, so the callee will be able to return back to the caller after it has finished executing. This marks the start of a new stack frame. The stack is used to store local variables and function arguments (except for the first four arguments, which are passed in registers). In addition, some CPU registers are specified as *nonvolatile* in the x64 ABI, meaning their values are part of the caller's state and must be preserved. To be able to utilize nonvolatile registers, a function will save their old values on the stack, and later restore them before returning to the caller.

When a function returns normally, it will execute instructions to restore all nonvolatile registers and free all stack space that has been consumed by its stack frame. Then, it will pop the `RIP` off the stack and use it to do the actual return operation. However, if an exception occurs, the thread will not be able to execute the instructions for cleaning up the stack, which are part of the normal return path. This means that the exception handling code will need to determine where to locate the nonvolatile registers and the `RIP`, so these may be restored, as well as the amount of stack space that has been allocated.

To facilitate the unwinding of a stack frame, the x64 ABI requires that every function consist of three parts: a *prolog* at the start of the function, an *epilog* at the end of the function, and a function body inbetween. Only the prolog and epilog are allowed to manipulate the stack pointer—all allocation of stack space must happen in the prolog, and the corresponding clean-up must be done

in the epilog.[1] This simplifies the common case, where exceptions occur within the function body. It also requires a function to allocate sufficient stack space in the prolog for the maximum number of arguments that is used in any call to another function. However, for functions written in C, this is handled entirely by the compiler.

The Portable Executable (PE) binary of an application or DLL contains all the necessary information that the exception handling code needs to be able to unwind the stack of any function within that binary. Every function has an associated *unwind information* structure (`UNWIND_INFO`) [95], which is contained as an entry in the PE file's *exception table* [96]. The structure records an *unwind code* [97] for each instruction in the function's epilog that either manipulates the stack or accesses a register that is needed during unwind. Together, the unwind codes of a function allow the exception handling code to reverse the effect of the function's prolog when an exception occurs.

However, the unwind information is not usable only for handling exceptions. It may also be used to produce stack traces, without having to exceute exception handlers or force a thread to return from a function. This is done by Windows debuggers such as Windbg, through the available Debugger Engine API in Windows [98]. In this case, the primary use of the unwind information is to locate the `RIP` in each stack frame and find where the next stack frame starts.

As previously mentioned in Chapter 4, Casuar already includes a PE parser from previous work [41]. We have extended the parser to support retrieval of unwind information for a given `RIP`. Using this, we have implemented the necessary functionality for producing stack traces. However, the unwind information only provides the memory addresses of each function call. In order to get stack traces that are meaningful and usable to us, it is also necessary to map the function addresses to their corresponding function names.

It is possible to use the PE export table of a DLL to resolve function addresses of public symbols to their respective symbol names. Table 5.2 shows an example stack trace without symbol names, and Table 5.3 show the same trace where the names of all public symbols have been included. However, as can be seen, most of the functions that are part of the trace are internal, private symbols, which cannot be resolved using the export tables. Hence, it is not sufficient to rely exclusively on export tables for resolving function names, as they do not

1. There is one exception, where the function body may use the stack for dynamic allocation, using `alloca()`. However, in these cases, one of the nonvolatile registers will be used as frame pointer to indicate to the exception handling code where the static portion of the stack frame starts [94].

reveal enough information to gain any insights from the traces.

Table 5.2: Example stack trace where no function names have been resolved.

RIP	Location	Symbol identifier
0x40000029aeea	<ntdll.dll+0x9aeea>	ntdll.dll!.text+0x99eea
0x400000c123ba	<kernelbase.dll+0x123ba>	kernelbase.dll!.text+0x113ba
0x400000c14da5	<kernelbase.dll+0x14da5>	kernelbase.dll!.text+0x13da5
0x400000c025e1	<kernelbase.dll+0x025e1>	kernelbase.dll!.text+0x15e1
0x400000255619	<ntdll.dll+0x55619>	ntdll.dll!.text+0x54619
0x400000254de2	<ntdll.dll+0x54de2>	ntdll.dll!.text+0x53de2
0x400000254c19	<ntdll.dll+0x54c19>	ntdll.dll!.text+0x53c19
0x400000254bfb	<ntdll.dll+0x54bfb>	ntdll.dll!.text+0x53bfb
0x400000222516	<ntdll.dll+0x22516>	ntdll.dll!.text+0x21516
0x4000002211d0	<ntdll.dll+0x211d0>	ntdll.dll!.text+0x201d0
0x40000025830d	<ntdll.dll+0x5830d>	ntdll.dll!.text+0x5730d
0x4000002cf264	<ntdll.dll+0xcf264>	ntdll.dll!.text+0xce264
0x4000002ba188	<ntdll.dll+0xba188>	ntdll.dll!.text+0xb9188
0x400000256a5a	<ntdll.dll+0x56a5a>	ntdll.dll!.text+0x55a5a

Table 5.3: Example stack trace from Table 5.2, where PE export tables are used to resolve function names.

RIP	Location	Symbol identifier
0x40000029aeea	**<ntdll.dll+0x9aeea>**	**ntdll.dll!NtQuerySystemInformation+0xa**
0x400000c123ba	<kernelbase.dll+0x123ba>	kernelbase.dll!.text+0x113ba
0x400000c14da5	<kernelbase.dll+0x14da5>	kernelbase.dll!.text+0x13da5
0x400000c025e1	<kernelbase.dll+0x025e1>	kernelbase.dll!.text+0x15e1
0x400000255619	<ntdll.dll+0x55619>	ntdll.dll!.text+0x54619
0x400000254de2	<ntdll.dll+0x54de2>	ntdll.dll!.text+0x53de2
0x400000254c19	<ntdll.dll+0x54c19>	ntdll.dll!.text+0x53c19
0x400000254bfb	<ntdll.dll+0x54bfb>	ntdll.dll!.text+0x53bfb
0x400000222516	<ntdll.dll+0x22516>	ntdll.dll!.text+0x21516
0x4000002211d0	<ntdll.dll+0x211d0>	ntdll.dll!.text+0x201d0
0x40000025830d	**<ntdll.dll+0x5830d>**	**ntdll.dll!LdrLoadDll+0x99**
0x4000002cf264	<ntdll.dll+0xcf264>	ntdll.dll!.text+0xce264
0x4000002ba188	<ntdll.dll+0xba188>	ntdll.dll!.text+0xb9188
0x400000256a5a	**<ntdll.dll+0x56a5a>**	**ntdll.dll!LdrInitializeThunk+0xe**

To be able to resolve private symbols, it is necessary to use additional debug information. A PE file typically includes a special *CodeView* debug section that either contains such information or, more commonly, points to a separate program database (PDB) file containing the debug symbols. In Windows, the PDB files for most system DLLs are made available by Microsoft for debugging purposes [99]. Because PDB is a proprietary format, Microsoft also implements a DbgHelp library [100] that is used by debuggers or other third-party applications to extract information from the PDB files.

It is not convenient to reuse the DbgHelp library in Casuar, because it runs in user mode and is dependent on a number of other DLLs. Instead, we have implemented our own PDB parser. This allows us to exploit debug information associated with system DLLs that are loaded by an application. Our implementation corresponds to only a small subset of the functionality available from the DbgHelp library. Because there is little available information about the structure of PDB files, we have restricted Casuar's implementation to only include what is needed for mapping addresses of private functions to their respective

function names. However, this allows us to produce stack traces where we are able to resolve virtually all internal function calls. Table 5.4 illustrates what the stack trace from Table 5.2 and Table 5.3 looks like when this is done.

Table 5.4: Example stack trace from Table 5.2 and Table 5.3, where PDB files are used to resolve function names.

RIP	Location	Symbol identifier
0x40000029aeea	<ntdll.dll+0x9aeea>	ntdll.dll!NtQuerySystemInformation+0xa
0x400000c123ba	<kernelbase.dll+0x123ba>	kernelbase.dll!KernelBaseDllInitializeMemoryManager+0x3a
0x400000c14da5	<kernelbase.dll+0x14da5>	kernelbase.dll!_KernelBaseBaseDllInitialize+0x12745
0x400000c025e1	<kernelbase.dll+0x025e1>	kernelbase.dll!KernelBaseDllInitialize+0x11
0x400000255619	<ntdll.dll+0x55619>	ntdll.dll!LdrpCallInitRoutine+0x41
0x400000254de2	<ntdll.dll+0x54de2>	ntdll.dll!LdrpInitializeNode+0x176
0x400000254c19	<ntdll.dll+0x54c19>	ntdll.dll!LdrpInitializeGraph+0x75
0x400000254bfb	<ntdll.dll+0x54bfb>	ntdll.dll!LdrpInitializeGraph+0x57
0x400000222516	<ntdll.dll+0x22516>	ntdll.dll!LdrpPrepareModuleForExecution+0x14e
0x4000002211d0	<ntdll.dll+0x211d0>	ntdll.dll!LdrpLoadDll+0x338
0x40000025830d	<ntdll.dll+0x5830d>	ntdll.dll!LdrLoadDll+0x99
0x4000002cf264	<ntdll.dll+0xcf264>	ntdll.dll!LdrpInitializeProcess+0x1684
0x4000002ba188	<ntdll.dll+0xba188>	ntdll.dll!_LdrpInitialize+0x636dc
0x400000256a5a	<ntdll.dll+0x56a5a>	ntdll.dll!LdrInitializeThunk+0xe

The stack trace shown in Table 5.4 is an example of an actual trace that was generated by Casuar in response to an unimplemented feature in the system call `NtQuerySystemInformation()`. This example illustrates how the symbol names for the functions leading up to the system call may be useful. In this case, they could help determine why the system call was invoked, and how to implement the missing feature. For example, we see that the `kernelbase.dll` DLL has been loaded, and that the loader was currently executing the DLL's initialization code. Moreover, it is clear that the `NtQuerySystemInformation()` was called to retrieve some information that is needed to initialize a memory manager component in the DLL.

Another example, which illustrates how strack traces can be used to quantify progress, is shown in Table 5.5. In this case, the trace was generated after an access to a previously unimplemented field `ActivityId` in the TEB. The access happened after a call to `NtQuerySystemInformation()` that did not return the proper result. From the stack trace, we get a clear indicator that an error has occured. When seen in light of previous stack traces that do not contain error indicators, it is possible to pinpoint the source of the error quite accurately (which in this case is the `NtQuerySystemInformation()` system call). Then, the stack trace can be used to help drive out a correct implementation of the system call—when the error trace no longer appears, we can be somewhat certain that our implementation is correct.

Table 5.5: Example of a stack trace indicating an error in Casuar's implemented interface.

RIP	Location	Symbol identifier
0x400000278d45	<ntdll.dll+0x78d45>	ntdll.dll!EtwEventWriteNoRegistration+0x5d
0x4000002cc8dd	**<ntdll.dll+0xcc8dd>**	**ntdll.dll!LdrpLogFatalLdrEtwEvent+0xad**
0x4000002cd3be	**<ntdll.dll+0xcd3be>**	**ntdll.dll!LdrpInitializationFailure+0x66**
0x4000002ba284	<ntdll.dll+0xba284>	ntdll.dll!_LdrpInitialize+0x637d8
0x400000256a5a	<ntdll.dll+0x56a5a>	ntdll.dll!LdrInitializeThunk+0xe

5.4 Results

Using the techniques described in the previous sections together with the mechanisms from Chapter 3 and Chapter 4, we have been able to implement all necessary functionality for completing the loading phase of Native applications running on top of Casuar. In adherence to traditions for software development, we have written a *Hello world* Native application, as shown in Code Listing 5.1, and let it be the first application to successfully run on both Windows and Casuar. Because Native applications do not have default I/O streams (i.e. `stdin` and `stdout`) associated with them, we have used the `NtDrawText()` system call to write "`Hello, world!`" on the Windows boot screen. The same system call is implemented in Casuar to write to the system log, as Vortex does not have any graphical interface. Figure 5.2 shows a screenshot of the Hello world Native application running on Windows Server 2012 R2 (NT 6.3, build 9600).

Code Listing 5.1: Implementation of Hello world Native application.

```
1  void
2  NtProcessStartup(void *arg)
3  {
4      UNICODE_STRING  us;
5      LARGE_INTEGER   delay;
6
7      RtlInitUnicodeString(&us, L"Hello, world!\n");
8      NtDrawText(&us);
9
10     // Wait 10 seconds
11     delay.QuadPart = -100000000;
12     NtDelayExecution(FALSE, &delay);
13
14     NtTerminateProcess(NtCurrentProcess(), STATUS_SUCCESS);
15 }
```

Table 5.6 and Table 5.7 show complete overviews over all TEB and PEB structure fields that had to be initialized by Casuar in order to complete the loading phase of a Native application that is run alongside NT 6.3 system DLLs. All of these fields have been identified using the memory monitor described in Section 5.2. Although we have assigned values to all of the listed structure members, we do not have complete information about the semantics of each. For example, we do know that the PEB fields `AnsiCodePageData`, `OemCodePageData`, and

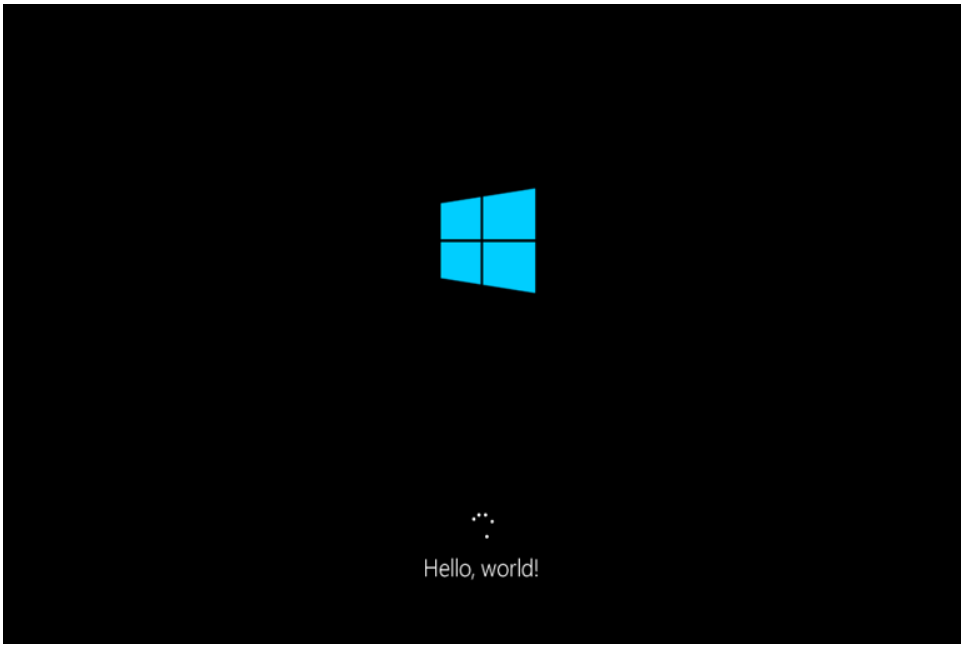


Figure 5.2: Hello world Native application run in Windows at boot-time.

Table 5.6: TEB fields that must be initialized by Casuar to complete the loading phase of a Native application using NT 6.3 DLLs. Offsets are relative to NT 6.3 struct definitions.

Offset	Field name	Assigned value
0x0008	NtTib.StackBase	Stack base of user stack
0x0010	NtTib.StackLimit	Stack limit of user stack
0x0030	NtTib.Self	Pointer to start of TEB
0x0040	ClientId.UniqueProcess	Unique ID
0x0048	ClientId.UniqueThread	Unique ID
0x0060	ProcessEnvironmentBlock	Pointer to PEB
0x02c8	ActivationContextStackPointer	NULL
0x17ee	SafeThunkCall	FALSE
0x17ee	InDebugPrint	FALSE
0x17ee	HasFiberData	FALSE
0x17ee	SkipThreadAttach	FALSE
0x17ee	WerInShipAssertCode	FALSE
0x17ee	RanProcessInit	FALSE
0x17ee	ClonedThread	FALSE
0x17ee	SuppressDebugMsg	FALSE
0x17ee	DisableUserStackWalk	FALSE
0x17ee	RtlExceptionAttached	FALSE
0x17ee	InitialThread	Set for main thread
0x17ee	SessionAware	FALSE

`UnicodeCaseTableData` have to point to National Language Support (NLS) codepages, and as such we have had to implement support for this. However, we do not know what the `pShimData` field in the PEB is supposed to point to, or what the possible values for the `AppCompatFlags` field are.

Table 5.7: PEB fields that must be initialized by Casuar to complete the loading phase of a Native application using NT 6.3 DLLs. Offsets are relative to NT 6.3 struct definitions.

Offset	Field name	Assigned value
0x002	BeingDebugged	FALSE
0x003	ImageUsesLargePages	FALSE
0x003	IsLegacyProcess	FALSE
0x003	IsImageDynamicallyRelocated	TRUE
0x003	SkipPatchingUser32Forwarders	FALSE
0x003	IsPackagedProcess	FALSE
0x003	IsAppContainer	FALSE
0x010	ImageBaseAddress	Base address of process' PE image
0x020	ProcessParameters	RTL_USER_PROCESS_PARAMETERS
0x050	ProcessInJob	FALSE
0x050	ProcessInitializing	TRUE
0x050	ProcessUsingVEH	FALSE
0x050	ProcessUsingVCH	FALSE
0x050	ProcessUsingFTH	FALSE
0x068	ApiSetMap	Pointer to apisetschema.dll section
0x080	TlsBitmapBits	0
0x0a0	AnsiCodePageData	Pointer to NLS codepage
0x0a8	OemCodePageData	Pointer to NLS codepage
0x0b0	UnicodeCaseTableData	Pointer to NLS codepage
0x0b8	NumberOfProcessors	Number of logical CPUs
0x0bc	NtGlobalFlag	0
0x0c0	CriticalSectionTimeout	0
0x0c8	HeapSegmentReserve	0x100000
0x0d0	HeapSegmentCommit	8192
0x0d8	HeapDeCommitTotalFreeThreshold	4096
0x0e0	HeapDeCommitFreeBlockThreshold	65536
0x0e8	NumberOfHeaps	0
0x230	PostProcessInitRoutine	NULL
0x240	TlsExpansionBitmapBits	0
0x2c8	AppCompatFlags	0
0x2d8	pShimData	NULL
0x2f8	ActivationContextData	NULL
0x308	SystemDefaultActivationContextData	NULL
0x318	MinimumStackCommit	8192
0x340	FlsBitmapBits	0

In Table 5.8, we also show all system calls that are invoked during the loading phase. For each system call, we indicate which subsystem in Casuar implements its functionality. Functions that correspond to services in the Windows Executive that are not implemented by either the Object Manager, I/O Manager, or

Memory Manager are listed with "Executive" as subsystem, and Windows Kernel services are listed with "Kernel". The system call `NtApphelpCacheControl()` has been implemented to return an error code, indicating an unsupported operation. For completeness, we also include all other system calls that are currently implemented by Casuar in Table 5.9.

Table 5.8: System calls that are used by the loading phase of a Native application using NT 6.3 DLLs.

Casuar subsystem	Name of system call	Implemented support
Executive	`NtQueryInformationProcess`	Partial support
	`NtQueryPerformanceCounter`	No
	`NtQuerySystemInformation`	Partial support
Kernel	`NtContinue`	Fully supported
	`NtTestAlert`	Fully supported
Object Manager	`NtOpenDirectoryObject`	Fully supported
I/O Manager	`NtOpenFile`	Open regular files
	`NtQueryVolumeInformationFile`	Partial support
Memory Manager	`NtAllocateVirtualMemory`	Most common operations
	`NtFreeVirtualMemory`	Most common operations
	`NtQueryVirtualMemory`	Most common operations
	`NtProtectVirtualMemory`	Most common operations
N/A	`NtApphelpCacheControl`	No

Naturally, we have also attempted to load and run regular Windows subsystem applications. However, it turns out that we are not able to get past the loading phase for this type of application, because it will—as part of loading `kernel32.dll`—try to communicate with the Windows subsystem process (`csrss.exe`) through ALPC, which we currently do not support in Casuar (as explained in Chapter 4). In other words, we cannot support existing regular Windows applications that are built on top of the Windows API at this point. This may be seen as a drawback, since there exist very few Native applications, and these are mostly support processes that are started at boot-time. Typically, Native applications are either present only to allow regular applications to run—such as `smss.exe` and `csrss.exe`—or they need to obtain exclusive access to system resources at boot-time—such as the `autochk.exe` file system validation application that requires raw disk block access. However, all of our efforts towards supporting Native applications are also part of the work required to support regular Windows applications, and are therefore highly relevant.

Because we only have available a small number of specialized Native applications, we have taken the approach to write our own Native applications, similar to the Hello World application, for testing the functionality of Casuar. In the

Table 5.9: Other system calls that are implemented by Casuar.

Casuar subsystem	Name of system call	Implemented support
Executive	`NtCreateThreadEx`	Creating threads for current process
	`NtTerminateThread`	Terminate current thread
	`NtTerminateProcess`	Terminate current process
	`NtDisplayString`	Print string in system log
	`NtDrawText`	Print string in system log
Kernel	`NtDelayExecution`	Fully supported
	`NtYieldExecution`	Fully supported
	`NtQueueApcThread`	Queue to threads in same process
	`NtSuspendThread`	Suspend threads in current process
	`NtResumeThread`	Resume threads in current process
	`NtCreateEvent`	Fully supported
	`NtWaitForSingleObject`	Fully supported
Object Manager	`NtCreateSymbolicLinkObject`	Fully supported
	`NtClose`	Fully supported
I/O Manager	`NtCreateFile`	Create/open regular files
	`NtReadFile`	Read from regular files
	`NtWriteFile`	Write to regular files
	`NtQueryAttributesFile`	Partial support
	`NtFlushBuffersFile`	Fully supported
Memory Manager	`NtCreateSection`	Partial support
	`NtOpenSection`	Partial support
	`NtQuerySection`	Partial support
	`NtMapViewOfSection`	Partial support
	`NtUnmapViewOfSection`	Partial support

next chapter, we use these to evaluate Casuar experimentally through a series of micro-benchmarks.

6 Evaluation

While implementing Casuar, our main focus has been on ensuring the correctness of its functionality, in order to achieve compatibility with a subset of the Windows ABI. However, performance is also an important aspect to the usefulness of a PLOS, as the PLOS architecture was developed with the intentions of improving upon the traditional library OS abstraction as a light-weight alternative to VMs.

In this chapter, we evaluate Casuar using micro-benchmarks that measure the overhead of a number of system calls and I/O operations. We run the same benchmarks on Windows and Wine, to compare the performance of Casuar to the performance of these existing systems.

6.1 Experimental Setup

We have written a number of small Native applications for running micro-benchmarks that we use to measure the overhead of selected system calls and I/O operations. We use these benchmarks to compare Casuar to both Windows and Wine. Specifically, we run the benchmark applications natively on Windows Server 2012 R2 (NT 6.3, build 9600), and on Wine 1.7.44 running on Ubuntu 14.04.2 LTS with Linux kernel version 3.13.0-24, as well as on Casuar.

All experiments are run on a Dell PowerEdge M600 blade server. The server

is equipped with two Intel Xeon E5430 2.66 GHz Quad-Core processors. Each core has separate 64×8 way 32 KB L1 data and instruction caches, and each pair of cores shares a 6 MB 64×24 way L2 cache (for a total of 12 MB L2 cache per processor). Each processor has a 1333 MHz front-side bus and is connected to 16 GB of DDR-2 main memory running at 667 MHz.

When running our benchmark applications on Vortex, Windows, and Linux, we attempt to reduce possible background interference that could affect our measurements. On Windows, we execute all benchmarks in a minimal environment at the boot-stage, where no other processes contend for CPU time or other resources. Similarly, we run Wine on top of a small Ubuntu installation, where only system-critical tasks, such as kernel-mode driver worker threads, are allowed to execute in the background.

In addition, we pin each benchmark thread to a separate CPU core, different from the boot core, which is typically subject to most interference from system tasks such as interrupts, deferred procedure calls (DPCs), and similar. This also prevents the thread schedulers from moving thread between cores for balancing load. We have observed that migration of threads is otherwise done on a frequent basis in Windows, typically motivated by power savings benefits. However, we have also seen that this can affect our measurements, for exmaple because threads achieve a lesser degree of CPU cache locality.

6.2 System Call Benchmarks

We have measured the time it takes to perform a number of different system calls on Windows, Casuar, and Wine. Although it is practically impossible to eliminate interference from interrupts or the thread scheduler, we have taken measures to reduce the interference for most of the system calls that we benchmark. Through observation, we have found that background interference seems more common within few, short time windows, rather than being distributed evenly over a larger time interval. For example, the thread schedulers of Vortex, Windows, and Linux will typically preempt a thread each 10 ms, when the thread's quantum expires, but do not cause interference inbetween.

Based on our observations, we have chosen to quantify the cost of each type of system call by counting the number of sequential invocations that can be completed within a certain time interval. We use an upper bound to limit the length of this interval, so we can make sure it is sufficiently small to avoid scheduler preemption, and sufficiently large to capture a large amount of possible interference. By dividing the measured time interval with the number of completed invocations, we obtain a mean value for how much time is

consumed by a single invocation. This process is then repeated many times, to get a sample sufficiently large to have statistical significance.

Under the assumption that interference occurs frequently within small time windows, only some of the means in the sample should be affected to a large degree. Such subsamples can then be removed from the sample, through the use of outlier removal techniques. This will reduce the variance between the remaining data in the sample, and the grand mean of the sample will yield a more precise estimation of the actual cost of a single system call. Note especially that it is easier to perform outlier removal on a sample of means, than if the sample consisted of one observation per system call invocation; in the latter case, a larger number of subsamples could be affected by interference to the extent that these are no longer outliers.

Formally, we assume an infinite population with mean μ and variance σ^2, and draw (through measurements) m samples from the population, each consisting of n observations. We let the random variable X_{ij} represent observation i within sample j, where $1 \leq i \leq n$ and $1 \leq j \leq m$, and we assume that all X_{ij} are independent and identically distributed. We let x_{ij} be the assumed value of X_{ij}, which represents the time that is used by a single system call invocation. However, we do not observe the x_{ij} directly, but instead count the number of occurrences within a certain time interval. We let

$$T_j = \sum_{i=1}^{n_j} X_{ij}$$

represent the total length of the time interval in the jth sample, t_j be the assumed value of T_j, and n_j be the counted number of events within the interval t_j. For each j, t_j will be approximately the length of some predetermined upper bound, but we allow t_j to exceed this bound slightly to make it possible to obtain one observation in a sample even when the upper bound is zero.

Furthermore, we let the random variable Y_j represent the sampling mean of sample j, and let y_j be the assumed value of Y_j. That is,

$$Y_j = \frac{T_j}{n_j},$$

where the sample size n_j varies between samples, because it corresponds to the counted number of system calls within the time interval t_j, and this interval is likely to vary between samples.

From our initial assumption about the population, we know that $\mathrm{E}\left[X_{ij}\right] = \mu$ and $\mathrm{Var}\left[X_{ij}\right] = \sigma^2$. According to the Central Limit Theorem, the sampling

distribution of the sampling means Y_j is approximately normal, given that n_j is sufficiently large, and it can be shown that $\mathrm{E}\left[Y_j\right] = \mu$ and $\mathrm{Var}\left[Y_j\right] = \frac{\sigma^2}{n_j}$.

Let

$$\overline{Y} = \frac{1}{m}\sum_{j=1}^{m} Y_j$$

be the grand mean of the samples Y_j of means and $\overline{y}$ its assumed value. Then

$$\mathrm{E}\left[\overline{Y}\right] = \frac{1}{m}\sum_{j=1}^{m} \mathrm{E}\left[Y_j\right] = \mu$$

and

$$\mathrm{Var}\left[\overline{Y}\right] = \frac{1}{m^2}\sum_{j=1}^{m} \mathrm{Var}\left[Y_j\right] = \frac{1}{m^2}\sum_{j=1}^{m} \frac{\sigma^2}{n_j} = \frac{r\sigma^2}{m^2}, \quad \text{where} \quad r = \sum_{j=1}^{m} \frac{1}{n_j}.$$

We define the typical estimator S_Y^2 of $\mathrm{Var}\left[Y_j\right]$ for each j as

$$S_Y^2 = \frac{1}{m-1}\sum_{j=1}^{m} \left(Y_j - \overline{Y}\right)^2.$$

However, S_Y^2 is *not* an estimator of the population variance σ^2; rather, it can be shown that $\mathrm{E}\left[S_Y^2\right] = \frac{r\sigma^2}{m}$. In other words, an estimator for σ^2 can be obtained by multiplying S_Y^2 with m/r. We define

$$S^2 = \frac{mS_Y^2}{r} = \frac{mS_Y^2}{\sum_{j=1}^{m} \frac{1}{n_j}}$$

as an unbiased estimator for the population's true σ^2, let s^2 be its assumed value, and finally use $s = \sqrt{s^2}$ to estimate the standard deviation σ of the population. Although s will be a biased estimator, the bias should be unsignificant for large m.

In the results shown in the remainder of this section, the estimated cost of a system call corresponds to $\overline{y}$, and the standard deviation correponds to s. We calculate $\overline{y}$ and s after outliers have been removed using a 95% prediction interval test.

In summary, we are able to limit the effects that background interference has on our benchmark results. We do this by measuring the time it takes to execute

a sequence of system calls instead of a single system call. Instead of using a fixed number of system calls, we count the number of system calls that can be completed within a bounded time interval, and use this to estimate the time ($\overline{y}$) it takes to complete a single system call. Finally, we are also able to calculate the standard deviation (s) of the individual system calls, although we only measure the sequences of system calls. The reason for doing this calculation, is that the standard deviation s is a measure for the variability of how much time is used to execute each system call, instead of only a measure for the precision of $\overline{y}$.

6.2.1 Benchmark results

We have run benchmarks for system calls corresponding to some of the Windows Kernel and Windows Executive services that we implement in Casuar. All measurements are done in context of a single thread, which executes a sequence of system calls. We repeat each experiment 500 times (this value corresponds to the sample size m). We also vary the upper bound for the time intervals depending on the type of operation to be performed, and on which platform the benchmark is executed. The bound is chosen based on experimentation, to attempt to reduce the standard deviation for all experiments. The value we use typically lies between 200 µs and 800 µs. For a few operations, we have used a bound of 0 µs, to let samples consist of only a single observation.

The results from benchmarking the Windows Kernel synchronization services are shown in Figure 6.1. From this, we see that Wine performs significantly worse than Windows for all of the synchronization-based operations that we have measured. We attribute most of these performance penalties to communication overhead between the benchmark application and the Wine server process, which is used to centralize synchronization operations. This exemplifies our previous claim that it is difficult to implement efficient sharing between processes, when binary-compatibility is achieved through a software layer that runs in the same address space as the application (as is the case for both Wine and conventional library OSs).

We also see that in many of the cases, Casuar exhibits comparable performance to native Windows. Casuar is also in most cases much faster than Wine. For example, we have measured the overhead of the `NtWaitForSingleObject()` system call; by waiting for a non-signaled dispatcher object with a timeout value of zero, we effectively make the function examine the state of the dispatcher object, without causing the calling thread to enter a wait state. From our measurements, we see that our implementation of this function is nearly as fast as the implementation in Windows.

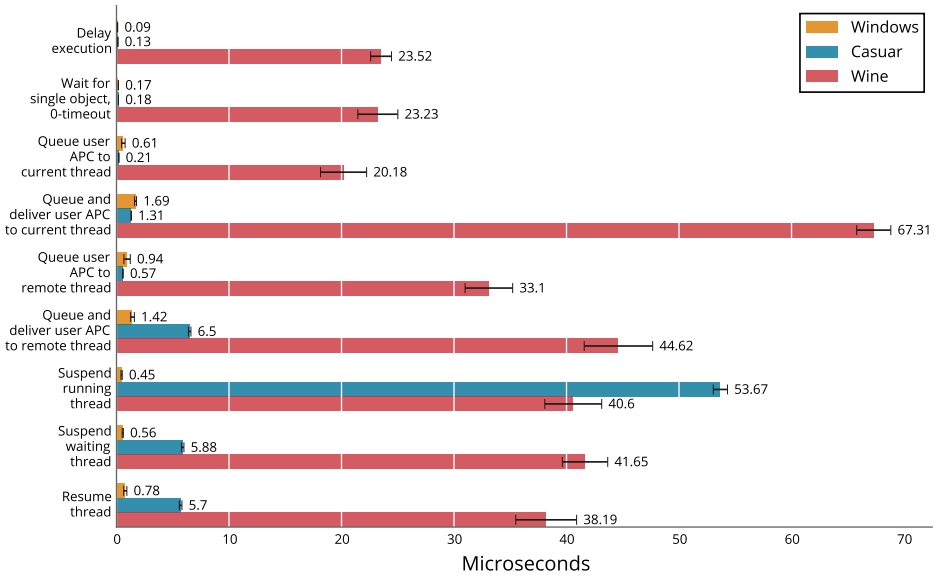


Figure 6.1: Benchmark of synchronization and signaling services (corresponding to system calls provided by the Windows Kernel). The bar chart compares the performance of each operation on Windows, Casuar, and Wine (smaller is better). Error bars indicate standard deviation.

In a few cases, we even see that Casuar is faster than Windows; specifically, we are able to queue APCs to a thread, and both queue and deliver APCs from a thread to itself, with less overhead than Windows. We believe the reason for this is that all APC delivery in Windows is dependent on CPU software interrupts generated by the NT hardware abstraction layer (HAL), even when a thread queues an APC to itself. As may be recalled from Chapter 3, the interrupt request level (IRQL) mechanism in Windows is implemented with hardware-support from the processor's local APIC. Lowering the IRQL below APC LEVEL will unmask pending APC interrupts, which originate either from a local software interrupt that was generated on the same CPU core, or from an inter-processor interrupt (IPI) sent from another core. Windows does not differentiate between the case where a thread sends an APC to itself, or it sends an APC to another thread on the same CPU core; both cases result in a local software interrupt. In contrast, Casuar implements the IRQL abstraction entirely in software. As a consequence, we are able to deliver APCs from a thread to itself with only the overhead of a regular function call for invoking the APC interrupt handler, which is cheaper than generating an actual software interrupt.

However, Casuar's implementation of APC suffers from a significant overhead when delivering an APC to another thread. By measuring the time used to sus-

pend a running thread different from the current thread, we implicitly quantify the cost of delivering a kernel APC to a remote thread. Recall from Chapter 3 that such APCs are delivered in Casuar by emulating software interrupts, using the Vortex system calls `vx_thread_getcontext()` and `vx_thread_setcontext()`. These functions will, by necessity, synchronize with the interrupted thread. To prevent race conditions, the thread context operations have to be performed by Vortex on the same CPU core that hosts the remote thread. Hence, calling these functions involve sending IPIs to another core and waiting for a context record to be retrieved from or updated on that core. As a result, we see that our implementation of APC delivery to other threads performs even worse than Wine's implementation, which already is many times slower than the implementation in Windows. A possible way to improve our implementation could be to extend the software interface of Vortex to natively provide mechanisms for sending interrupts to threads, for example by adding support for a virtualized APIC in its virtualization layer.

By measuring the time required to suspend a waiting thread and to resume a thread, we similarly quantify the cost of signaling a thread for wake-up. We see that the cost of suspending a thread approximately equals the cost of resuming a thread, both on Windows and Casuar. Casuar is a bit slower than Windows, but is still many times faster than Wine.

Figure 6.2 shows the results from benchmarking executive services implemented in the Object Manager, I/O Manager, and Memory Manager. As with the results shown in Figure 6.1, the performance of Casuar's implementations is comparable to that of Windows in several cases. Especially the implementation of Casuar's Object Manager seems to be only slightly slower than in Windows. However, we have only performed measurements in non-contended scenarios, so we do not currently have enough evidence to fully support this claim.

We have tried to characterize the overhead of the I/O Manager in Casuar by performing I/O operations that are essentially no-ops. Specifically, we have measured the time it takes to complete asynchronous read and write operations of zero bytes, and the time it takes to flush all pending writes to the file system when there are no outstanding write operations. For the read and write operations, Casuar performs on par with Windows, whereas Wine is many times slower. It should, however, be noted that the current implementation of Wine does not implement asynchronous I/O for regular files, so some of the additional overhead could be attributed to this fact. Somewhat surprisingly, the flush operation executes many times faster on Casuar than on either Windows or Wine.

Finally, we see that the executive operations in Casuar that perform the worst, when compared to Windows, are those related to memory allocation. We

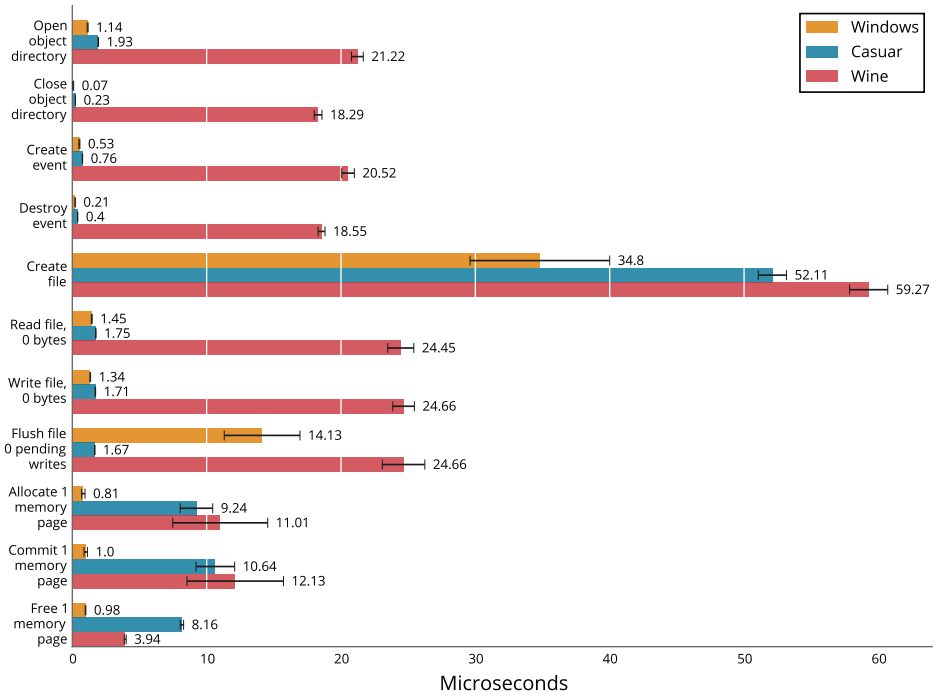


Figure 6.2: Benchmark of executive services in the Object Manager, I/O Manager, and Memory Manager. The bar chart compares the performance of each operation on Windows, Casuar, and Wine (smaller is better). Error bars indicate standard deviation.

attribute most of this overhead to Vortex, as Vortex performs almost all memory management tasks on behalf of Casuar. We assume there exist opportunities for improvement, but do not delve into the specifics of the Vortex implementation here.

6.3 I/O benchmarks

In addition to benchmarking individual system calls, we also attempt to characterize the overhead of the I/O Manager in Casuar in more detail, since I/O operations are typical performance sensitive tasks. We do this by measuring the latency of read and write operations of different sizes against regular files. However, because Vortex, Windows, and Linux implement different file systems with different semantics, we are not concerned here with the time it takes to complete a file operation to disk. Instead, we measure the time it takes to access the disk block cache (in the case of Windows) or the buffer storage of

an in-memory file system (in the case of Vortex and Linux).

When measuring latency of I/O operations, we do not use the same approach as described in Section 6.2. Because an I/O operation is typically a more expensive operation than a system call, and thus takes much longer time to complete, it is sufficient to let each sample contain a single observation. As with the system call benchmarks, we use a sample size of 500.

We measure four different types of accesses. Figure 6.3 and Figure 6.4 show the results for asynchronous, unbuffered read and write operations. With these configurations, Windows and Casuar are both able to return from the I/O operations before the operations have completed, returning `STATUS_PENDING` to the caller. In contrast, it should be noted that Wine does not support asychronous I/O towards regular files, and instead completes all such I/O operations synchronously. This means that Wine's seemingly severe performance overhead shown in Figure 6.3 and Figure 6.4 is not directly comparable to the performance shown for Windows and Casuar in the same figures.

For the asynchronous file operations, we see that the latency in Windows is getting larger as the size of the I/O operations increases, whereas Casuar's latency does not grow significantly. There are small differences when the size is below 8 MB. However, Casuar is clearly able to respond faster than Windows for sizes above 8 MB.

Figure 6.5 and Figure 6.6 show the results for synchronous file operations, for buffered read operations and unbuffered write operations, respectively. The first figure shows that read operations from block caches in Windows and Linux are faster than read from the in-memory file system in Vortex. In the second benchmark, Windows has to write to NTFS buffers before being able to complete the file operation. The latency of Windows is therefore not comparable to Wine or Casuar in this case. However, we see for both benchmarks that Wine is able to complete the I/O operation towards a Linux in-memory tmpfs file system before the other two systems, and that Casuar performs the worst in these cases. In sum, these findings indicate opportunities for improvements in Vortex' I/O stack.

6.4 Summary

In this chapter, we have evaulated the performance of Casuar experimentally by running a series of micro-benchmarks for system calls and I/O operations on Casuar, Windows, and Wine. By comparing the results from the system call benchmarks, we see that most implemented system services in Casuar

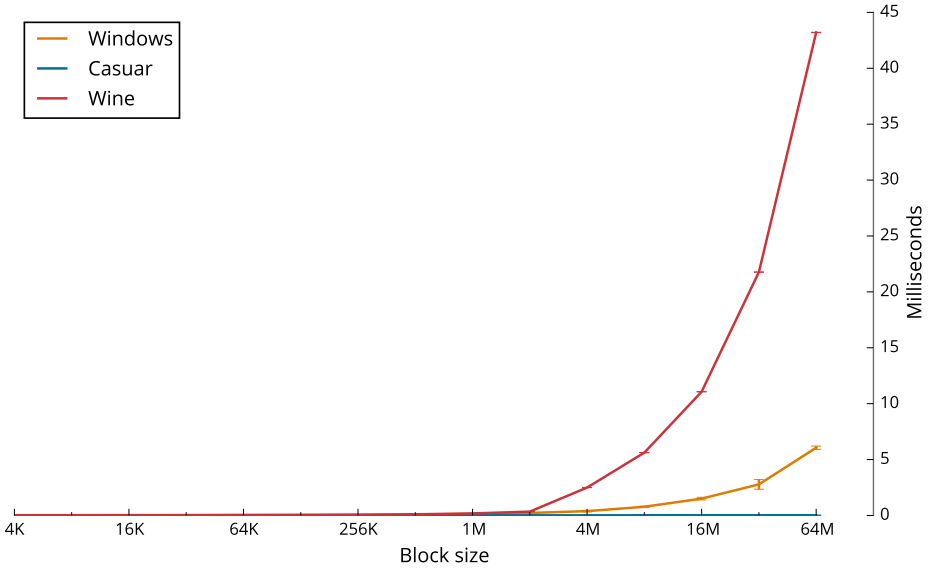


Figure 6.3: Measured time to complete an asynchronous, unbuffered read operation to a file. The plot compares the performance of the file operation on Windows, Casuar, and Wine for different block sizes. Error bars indicate standard deviation.

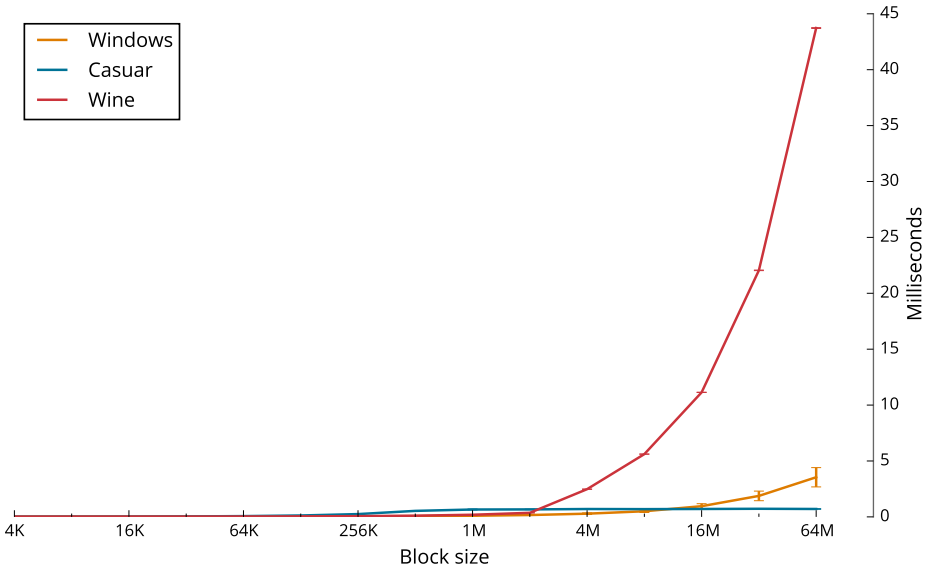


Figure 6.4: Measured time to complete an asynchronous, unbuffered write operation to a file. The plot compares the performance of the file operation on Windows, Casuar, and Wine for different block sizes. Error bars indicate standard deviation.

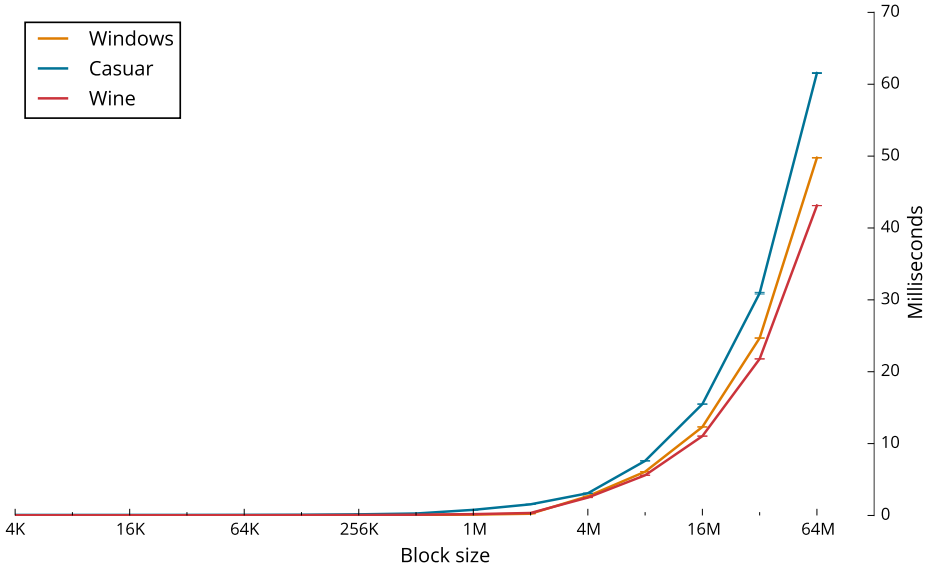


Figure 6.5: Measured time to complete a synchronous, buffered read operation to a file. The plot compares the performance of the file operation on Windows, Casuar, and Wine for different block sizes. Error bars indicate standard deviation.

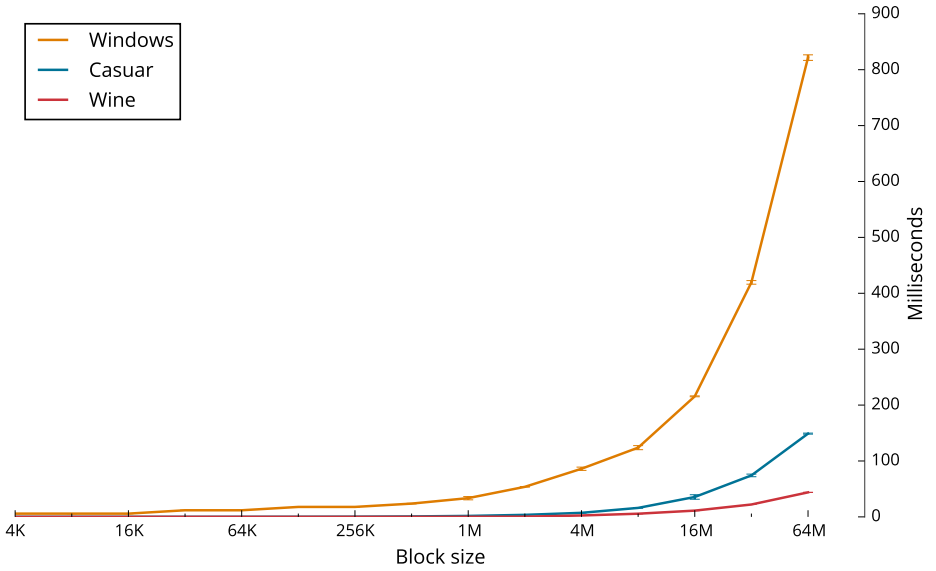


Figure 6.6: Measured time to complete a synchronous, unbuffered write operation to a file. The plot compares the performance of the file operation on Windows, Casuar, and Wine for different block sizes. Error bars indicate standard deviation.

perform only slightly worse than Windows, and significantly better than Wine. In a few special cases, our implementation is even faster than that of native Windows. However, we have also identified mechanisms that currently exhibit higher latency than both Windows and Wine, and which could most likely be improved. For example, our implementation of emulated software interrupts is currently very slow, and could be improved through modifications of the software interface provided by Vortex.

We demonstrate low overhead in our implementation of the I/O Manager. The low latency becomes especially significant as the transfer size of file operations increases. However, we see that the I/O stack in Vortex has room for improvements, as both the block cache of Windows and the Linux tmpfs file system used by Wine outperform the in-memory file system used by Vortex.

7

Concluding Remarks

In this chapter, we summarize our results and discuss how they relate to our thesis statement. We conclude by outlining possibilities for future work.

7.1 Results

The protected library OS (PLOS) architecture was designed to be a light-weight alternative to virtual machines (VMs), where applications are virtualized rather than entire OSs. Unike traditional library OSs, a PLOS facilitates hosting of multi-process applications, and can target compatibility with applications built for another OS at the application binary interface (ABI) level without requiring modification of existing binaries.

The PLOS has already been demonstrated as a promising architecture, through the implementation of a Linux-compatible PLOS capable of running complex, unmodified Linux applications like Apache, MySQL, and Hadoop on top of Vortex. The focus of this thesis has been to further strenghten the viability of the PLOS as a suitable general-purpose architecture for application virtualization. Specifically, our thesis is:

> *The protected library OS architecture permits unmodified multi-process Windows applications and user-mode DLLs to run under a Windows library OS.*

We have implemented a PLOS—Casuar—that targets compatibility with Windows applications through the implementation of a commonly used subset of Windows system calls. To accomplish this, we have studied the architecture of the Windows NT kernel, and its Windows Kernel and Windows Executive components, which implement most of the system services that are part of the NT system call ABI. We have identified and evaluated the most essential thread synchronization and signaling mechanisms that are part of the Windows Kernel, and implemented equivalent constructs in Casuar. Building on these, we have also analyzed the most important subsystems of the Windows Executive, and implemented selected subsets of their functionality in Casuar to provide higher-level system services to applications.

Retaining the semantics of the NT ABI in Casuar requires thorough understanding of the underlying mechanisms of Windows NT. To implement the mechanisms corresponding to functionality of the Windows Kernel and Windows Executive in Casuar, we have to a large degree depended on existing documentation or descriptions from available literature. However, most of the NT ABI is undocumented and used indirectly by applications through DLLs, which are loaded as part of each application's process address space. In addition, the ABI also comprises a number of data structures, containing fields that are initialized by the NT kernel and accessed by user-mode DLL code.

Through the implementation and use of a memory monitor mechanism and a mechanism for producing stack traces, we have been able to identify all ABI dependencies, and obtain additional information about the purpose and functionality of various undocumented system calls that are used by system DLLs. This has allowed us to implement all the necessary functionality in Casuar for loading and running applications that are built on top of the Native API of Windows. We have demonstrated that we are able to run the same applications and DLLs on both Windows and Casuar, without introducing PLOS-specific modifications to either.

Although we have not demonstrated any specific experiments where multi-process applications are hosted on the same instance of Casuar, this is nonetheless directly facilitated by the PLOS architecture, as exemplified by the implementation of the Linux-compatible PLOS [7], [5], [8], [9]. The only restriction on sharing between processes in our Casuar implementation, is that we do not support attaching threads that are part of a process to the address space of another process, as detailed in Chapter 3.

Evaluating these findings, we see that our thesis holds; we are able to run unmodified Windows applications and user-mode DLLs on top of Casuar, and the PLOS architecture facilitates hosting of multi-process applications. We have also evaluated the performance of Casuar experimentally, by comparing

the system to Windows and Wine through a series of micro-benchmarks. Our results indicate that the PLOS architecture does not by itself impose significant overhead on the execution of virtualized applications. Instead, we see that Casuar attains near-native performance for a number of system services, and performs many operations several times faster than Wine. From the results, we have also suggested possible areas for improvement, both in Casuar and Vortex, concerning operations that are slower than on Windows or Wine.

7.2 Future Work

Although we regard the ability to run Native applications on Casuar as a significant accomplishment, the system would benefit more from the ability to run regular Windows subsystem applications as well. As previously outlined in Chapter 4 and Chapter 5, there are a few additional mechanisms that will have to be supported in Casuar for this to be possible. We reiterate some of them here, and also suggest other improvements or additions that could be made to the Casuar implementation:

ALPC ALPC is the next mechanism that must be supported to be able to load Windows subsystem applications. During the loading phase, all Windows subsystem applications will use ALPC to communicate with the Windows Subsystem process (`csrss.exe`). Achieving support for ALPC in Casuar will likely involve modifications to Vortex, as the mechanism is based on shared memory, which Vortex does not currently support.

Sessions Sessions are used to provide different sandboxes in Windows with different logical views of parts of the NT namespace. They are, for example, used to isolate system services from normal applications started by the end-user.

Windows Registry Many Windows applications use the Windows Registry frequently, for example to access configuration data. It is also likely that a registry must be implemented to be able to complete the loading of regular Windows subsystem applications.

Graphics support Although we do not primarily target desktop applications, there are many Windows applications that use window-based graphics to provide configuration interfaces. We believe it would be possible to follow the same approach as Drawbridge [34], and provide graphical interfaces to applications through RDP connections. This would require a large amount of work. Specifically, we would have to implement functionality corresponding to the `win32k.sys` kernel-mode driver in Windows, which

implements a separate system call table for graphics-related services. In addtion, we would have to implement support for the RDP protocol.

Network support For Casuar to be able to support a wide range of applications in the future, we would have to implement support for the socket-based network APIs in Windows. This would require implementing corresponding functionality to a number of kernel-mode drivers.

List of References

[1] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011.

[2] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, New York, NY, USA, 2010. ACM.

[3] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS-07, pages 200–209, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. EC2 Performance Analysis for Resource Provisioning of Service-oriented Applications. In *Proceedings of the 2009 International Conference on Service-oriented Computing*, ICSOC/ServiceWave'09, pages 197–207, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Audun Nordal, Åge Kvalnes, and Dag Johansen. Paravirtualizing TCP. In *6th international workshop on Virtualization Technologies in Distributed Computing*, pages 3–10, 2012.

[6] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. Efficient Resource Provisioning in Compute Clouds via VM Multiplexing. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 11–20, New York, NY, USA, 2010. ACM.

[7] Audun Nordal, Åge Kvalnes, and Dag Johansen. Balava: Federating Private and Public Clouds. In *2011 IEEE World Congress on Services*,

pages 569–577, 2011.

[8] Audun Nordal, Åge Kvalnes, Robert Pettersen, and Dag Johansen. Streaming as a Hypervisor Service. In *7th international workshop on Virtualization Technologies in Distributed Computing*, 2013.

[9] Åge Kvalnes, Dag Johansen, Robbert van Renesse, Fred B. Schneider, and Steffen Viken Valvåg. Omni-Kernel: An Operating System Architecture for Pervasive Monitoring and Scheduling. Technical Report IFI-UiT 2013-75, Department of Computer Science, University of Tromsø, 2013.

[10] Farzad Sabahi. Secure Virtualization for Cloud Environment Using Hypervisor-based Technology. *International Journal of Machine Learning and Computing*, 2(1):39–45, February 2012.

[11] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[12] Stuart E. Madnick and John J. Donovan. Application and Analysis of the Virtual Machine Approach to Information System Security and Isolation. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 210–224, New York, NY, USA, 1973. ACM.

[13] L. H Seawright and R. A. MacKinnon. VM/370: A Study of Multiplicity and Usefulness. *IBM Syst. J.*, 18(1):4–17, mar 1979.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[15] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for HPC Systems. In *Proceedings of the 2006 International Conference on Frontiers of High Performance Computing and Networking*, ISPA'06, pages 474–486, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[17] The Xen Project. http://www.xenproject.org/. [Online].

[18] Microsoft. Hyper-V. https://technet.microsoft.com/en-US/windowsserver/dd448604.aspx. [Online].

[19] KVM. http://www.linux-kvm.org/. [Online].

[20] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 2–13, New York, NY, USA, 2006. ACM.

[21] VMware. Performance Evaluation of Intel EPT Hardware Assist. Technical report, Mar 2009. Available at http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

[22] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware Virtual Machine Scheduling for I/O Performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 101–110, New York, NY, USA, 2009. ACM.

[23] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, 2007. ACM.

[24] Miguel Gomes Xavier, Marcelo Veiga Neves, Fabio Diniz Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, pages 233–240, Washington, DC, USA, 2013. IEEE Computer Society.

[25] Docker. https://www.docker.com/. [Online].

[26] rkt - App Container runtime. https://github.com/coreos/rkt. [Online].

[27] Kubernetes by Google. `http://kubernetes.io/`. [Online].

[28] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[29] Barbara Higgins, Laura Hartman, Owen Allen, and Shanthi Srinivasan. *Solaris Zones - Oracle Enterprise Manager Ops Center User's Guide 11g Release 1 Update 3*. Oracle, Nov 2011.

[30] OpenVZ. `http://openvz.org/`. [Online].

[31] LXC. `https://linuxcontainers.org/`. [Online].

[32] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 44–54, New York, NY, USA, 2007. ACM.

[33] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[34] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM.

[35] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[36] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[37] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[38] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Brice no, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 52–65, New York, NY, USA, 1997. ACM.

[39] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE J.Sel. A. Commun.*, 14(7):1280–1297, September 2006.

[40] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, New York, NY, USA, 1997. ACM.

[41] Erlend Helland Graff. Initial Design and Implementation of a Windows VM OS for Vortex. Bachelor's Thesis in Computer Science, University of Tromsø – The Arctic University of Norway, January 2014.

[42] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a Discipline. *Communications of the ACM*, 32(1):9–23, January 1989.

[43] Microsoft. A history of Windows. `http://windows.microsoft.com/en-us/windows/history`, 2013. [Online; accessed 20-May-2015].

[44] Mark Russinovich. Windows NT and VMS: The Rest of the Story. `http://windowsitpro.com/windows-client/windows-nt-and-vms-rest-story`, 1998. [Online; accessed 20-May-2015].

[45] Microsoft. Xbox One operating system versions and system updates. `http://support.xbox.com/en-US/xbox-one/system/system-update-operating-system`. [Online; accessed 20-May-2015].

[46] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1*. Microsoft Press, Redmond, WA, USA, 6th edition, 2012.

[47] Mark Russinovich. Inside the Native API. http://netcode.cz/img/83/nativeapi.html, 2004. [Online; accessed 20-May-2015].

[48] Wine project. http://www.winehq.org/. [Online].

[49] Wine architecture - Wine Developer's Guide. https://www.winehq.org/site/docs/winedev-guide/x2591. [Online].

[50] ReactOS. https://www.reactos.org/. [Online].

[51] ReactOS. ReactOS/History. https://www.reactos.org/wiki/index.php?title=ReactOS/History&oldid=34714, 2014. [Online; accessed 11-June-2015].

[52] Microsoft Corporation. Scheduling, Thread Context, and IRQL. White paper. Available at http://www.microsoft.com/whdc/driver/kernel/IRQL.mspx, September 2014.

[53] Microsoft. Using Passive-Level Interrupt Service Routines (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/hh698277(v=vs.85).aspx. [Online; accessed 25-February-2015].

[54] Microsoft. Supporting Passive-Level Interrupts (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/hh451035(v=vs.85).aspx. [Online; accessed 25-February-2015].

[55] Microsoft. Introduction to DPC Objects (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff548024(v=vs.85).aspx. [Online; accessed 26-February-2015].

[56] Enrico Martignetti. Windows Vista APC Internals. Available at http://www.opening-windows.com/download/apcinternals/2009-05/windows_vista_apc_internals.pdf, May 2009.

[57] Microsoft Corporation. Locks, Deadlocks, and Synchronization. White

paper. Available at http://www.microsoft.com/whdc/driver/kernel/LOCKS.mspx, April 2006.

[58] Microsoft. QueueUserAPC function (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms684954(v=vs.85).aspx. [Online; accessed 26-February-2015].

[59] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, September 2014.

[60] Don Anderson and Tom Shanley. *Pentium Processor System Architecture*. Mindshare PC System Architecture. Addison-Welsey, 1995.

[61] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*, January 2015.

[62] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3*, January 2015.

[63] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*, January 2015.

[64] Microsoft. Asynchronous Procedure Calls (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms681951(v=vs.85).aspx. [Online; accessed 08-March-2015].

[65] Microsoft. Types of APCs (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff564853(v=vs.85).aspx. [Online; accessed 11-October-2014].

[66] Microsoft. Waits and APCs (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff565592(v=vs.85).aspx. [Online; accessed 11-October-2014].

[67] Microsoft. Alertable I/O (Windows). https://msdn.microsoft.com/en-

us/library/windows/desktop/aa363772(v=vs.85).aspx. [Online; accessed 09-March-2015].

[68] Albert Almeida. Inside NT's Asynchronous Procedure Call. http://www.drdobbs.com/inside-nts-asynchronous-procedure-call/184416590, November 2002. [Online; accessed 11-October-2014].

[69] CodeMachine Inc. Catalog of key Windows kernel data structures. http://www.codemachine.com/article_kernelstruct.html, April 2012. [Online; accessed 28-September-2014].

[70] Microsoft. KeStackAttachProcess routine (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff549659(v=vs.85).aspx. [Online; accessed 14-March-2015].

[71] Microsoft. Structured Exception Handling (C/C++). https://msdn.microsoft.com/en-us/library/swezty51.aspx. [Online; accessed 15-March-2015].

[72] Microsoft. Queued Spin Locks (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff559970(v=vs.85).aspx. [Online; accessed 16-March-2015].

[73] Microsoft. Introduction to Kernel Dispatcher Objects (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff548068(v=vs.85).aspx. [Online; accessed 09-March-2015].

[74] Microsoft. Defining and Using an Event Object (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff543006(v=vs.85).aspx. [Online; accessed 13-May-2015].

[75] Microsoft. WaitForMultipleObjects function (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms687025(v=vs.85).aspx. [Online; accessed 16-May-2015].

[76] David B. Probert. Windows Kernel Internals - Thread Scheduling. http://www.i.u-tokyo.ac.jp/ss/lecture/new-documents/Lectures/03-ThreadScheduling/ThreadScheduling.ppt, Sep 2004. [Online].

[77] David B. Probert. Windows Kernel Architecture Internals. http://research.microsoft.com/en-us/um/redmond/events/wincore2010/Dave_Probert_1.pdf, Apr 2010. [Online].

[78] Dave Probert. Architecture of the Windows Kernel. http://www.cs.fsu.edu/~zwang/files/cop4610/Spring2015/windows.pdf, April 2008. [Online; accessed 22-May-2015].

[79] Microsoft. I/O Stack Locations (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff551821(v=vs.85).aspx. [Online; accessed 24-March-2015].

[80] Microsoft. IO_STACK_LOCATION (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff550659(v=vs.85).aspx. [Online; accessed 24-March-2015].

[81] Microsoft. IRP Major Function Codes (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff550710(v=vs.85).aspx. [Online; accessed 22-May-2015].

[82] Microsoft. IRP_MJ_CREATE (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff550729(v=vs.85).aspx. [Online; accessed 22-May-2015].

[83] Microsoft. Canceling IRPs (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff540748(v=vs.85).aspx. [Online; accessed 4-April-2015].

[84] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2*. Microsoft Press, Redmond, WA, USA, 6th edition, 2012.

[85] Microsoft. ZwAllocateVirtualMemory routine (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff566416(v=vs.85).aspx. [Online; accessed 19-February-2015].

[86] Microsoft. Section Objects and Views.

https://msdn.microsoft.com/en-us/library/windows/hardware/ff563684(v=vs.85).aspx. [Online; accessed 18-April-2015].

[87] Microsoft. File-Backed and Page-File-Backed Sections (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff545754(v=vs.85).aspx. [Online; accessed 18-April-2015].

[88] Microsoft. ZwMapViewOfSection routine (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff566481(v=vs.85).aspx. [Online; accessed 18-April-2015].

[89] Microsoft. Kernel Transaction Manager. https://msdn.microsoft.com/en-us/library/windows/desktop/bb986748(v=vs.85).aspx. [Online; accessed 24-May-2015].

[90] Maxim S. Shatskih. How WinSock works. https://www.osronline.com/showthread.cfm?link=134510, Jul 2008. [Online; accessed 19-April-2015].

[91] Microsoft. Structured Exception Handling (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx. [Online; accessed 28-May-2015].

[92] Exceptional Behavior - x64 Structured Exception Handling. *The NT Insider*, 13(3), may 2006. Available at https://www.osronline.com/article.cfm?article=469.

[93] Microsoft. Overview of x64 Calling Conventions. https://msdn.microsoft.com/en-us/library/ms235286.aspx. [Online; accessed 28-May-2015].

[94] Eugene Ching. A walk in x64 land. http://www.codejury.com/a-walk-in-x64-land. [Online; accessed 15-March-2015].

[95] Microsoft. struct UNWIND_INFO. https://msdn.microsoft.com/en-us/library/ddssxxy8.aspx. [Online; accessed 15-March-2015].

[96] Ken Johnson. Introduction to x64 debugging, part 3. http://www.nynaeve,net/?p=11. [Online; accessed 15-March-2015].

[97] Microsoft. struct UNWIND_CODE. https://msdn.microsoft.com/en-us/library/ck9asaa9.aspx. [Online; accessed 15-March-2015].

[98] Microsoft. Examining the Stack Trace (Windows Debuggers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff543082(v=vs.85).aspx. [Online; accessed 28-May-2015].

[99] Microsoft. Use the Microsoft Symbol Server to obtain debug symbol files. https://support.microsoft.com/en-us/kb/311503. [Online; accessed 29-May-2015].

[100] Microsoft. Symbol paths (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms680689(v=vs.85).aspx. [Online; accessed 28-May-2015].