Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Feasibility Study of Building a User-mode Native Windows NT VMM

Bernhard Pöss

Studienarbeit

Verantwortlicher Betreuer:	Prof. Dr. Frank Bellosa
Betreuende Mitarbeiter:	BA of EE Joshua LeVasseur

9. Mai 2005

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 9. Mai 2005

Bernhard Pöss

Abstract

The concept of pre-virtualization offers new ways for building a virtual machine monitor. It is now possible to implement a virtual machine monitor in user-mode without suffering the hassle of other approaches such as para-virtualization. This work determines the extents to which a native virtual machine monitor may be implemented under Windows NT. A design proposal will be devised for an user-mode virtual machine monitor under Windows NT which supports paging, synchronous interrupt delivery and timer virtualization. Finally, the basics of implementing a native Windows NT virtual machine monitor are discussed.

Contents

Chapter 1

Introduction

The motivations for building a virtual machine monitor in user-mode are manifold. Building at least parts of a virtual machine monitor in kernel mode compromises stability and security of the underlying system [Dik00]. Furthermore, no administrative privileges are required, if the virtual machine monitor runs in user mode. There are solutions for Linux [Alo04] and L4 [LeV], but for Windows NT, no such solution is available.

In the following the feasibility of building a virtual machine monitor on top of Windows NT is discussed. We will show how to maintain security and stability of Windows NT while nearly reaching the speed of a VMM that is implemented in kernel-mode. The monitor harks back to pre-virtualization [LeV] for building a user-mode virtual machine monitor on the IA32 architecture that in parts runs as a native application. Virtualized functionality will cover paging as well as synchronous interrupt delivery and timer virtualization. First we give an introduction into virtual machine research, defining formal specifications for virtual machines and virtual machine monitors. Afterwards we explain the difficulties of virtualizing the IA32 architecture and illustrate the mechanisms the virtual machine monitor will implement. Next we give an overview of virtualization techniques, introducing pre-virtualization and afterburning. The constraints of the Win32 API will be discussed and it will be pointed out why it is not usable for a virtual machine monitor. After this, we describe the functional range of the native API, along with explanations how to utilize it. The description covers threads, processes, ports, sections and virtual memory mechanisms. Then other approaches for building virtual machines are discussed, namely User-mode Linux and Cooperative Linux. We propose a design for building a native user-mode Windows NT virtual machine monitor using pre-virtualization. Finally, the basic implementation of a native Windows NT virtual machine monitor is depicted.

Chapter 2

Background

2.1 Virtual Machines

A *Virtual Machine* (VM) is a hardware-software duplicate of a real existing computer system. The host processor executes a statistically dominant subset of the instructions in native mode. Instructions not executed on the host processor are executed on a virtual processor [Gol72]. A *Virtual Machine Monitor* (VMM) creates efficient and isolated programming environments that are duplicates which provide the users with the appearance of direct access to the real machine environment [RE00]. There exist two types of VMMs, referred to as Type I and Type II [RE00]. A Type I VMM runs on a bare machine. It is an operating system with virtualization mechanisms. A Type II VMM runs as an application. The operating system that controls the real hardware of the machine is called the *host OS*. Every OS that is run in the virtual environment is called a *guest OS*.

When executing a virtual machine, some processor instructions can not be executed directly on the processor [Pop74]. These instructions would interfere with the state of the underlying VMM or host OS and are called *sensitive instructions*. The key to implementing a VMM is to prevent the direct execution of sensitive instructions. Some sensitive instructions in the IA32 architecture are privileged, meaning that if they are not executed in most privileged hardware domain, they will cause a general protection exception. However, as explained later, there exist some instructions that are sensitive but not privileged. A VMM has to deal with them as well.

2.1.1 Hypervisors

A *hypervisor* is the part of a VMM that runs in privileged mode. It can be either a regular operating system such as Linux or Windows NT, or an operating system especially designed for VMMs such as Xen. The extent to which a VMM might be built on a specific hypervisor strongly depends on the features the hypervisor supports. A VMM needs full control over a guest OS's address space. This is not provided by Windows NT for example, which reserves the upper 2GB of it's address spaces for internal use. Furthermore a VMM must be able to trap privileged instructions or to get and set the context of a thread. Building a VMM on a hypervisor not natively supporting VMMs is a compromise to fit the constraints of the hypervisor's API.

2.1.2 Pure-, Para- and Pre-virtualization

Pure-virtualization depends on a VM that is an exact copy of a given physical machine. Except for timing, a guest OS executes in the same way on the virtual hardware as it would on real one [Gol74]. Thus the same guest OS binary can run on the VM and on the bare hardware. The guest OS executes in unprivileged mode. Every sensitive instruction causes a trap into kernel-mode. There it is emulated by the VMM. Non-sensitive instructions are executed directly by the hardware. Pure-virtualization depends on the ability to trap all sensitive instructions, which is only possible if all sensitive instructions are privileged and cause an exception when executed in user mode [Pop74]. For this reason pure-virtualization is not possible on all present architectures, in particular not on Intel IA32 [RE00]. Pure-virtualization causes significant costs, since every sensitive instruction causes a trap, which in turn leads to a mode switch. Traps are expensive on modern architectures, as they flush the pipeline, and cost many cycles [LeV].

Para-virtualization tries to reduce these costs by making changes to the guest OS. The changes are typically made at source code level, to provide a higher-level abstraction than the physical hardware called the virtual hardware. The guest OS is then ported to that virtual hardware. This allows to adapt the guest OS to specific hypervisors. Unfortunately the reduced costs for trapping sensitive instructions are bought with significant engineering costs [LeV].

Pre-virtualization [LeV] performs automatic compile-time or boot time para-virtualization of the guest OS. Sensitive instructions are located and replaced with emulation code. The emulation code is the unprivileged part of the VMM and is called the *wedge*. The wedge is specific to the hypervisor. It bridges the semantic gap between the hardware interface provided by the hypervisor, and the modified interface presented to the guest OS. The wedge is part of the guest OS protection domain, allowing the guest OS to access its functionality with minimal overhead. This supports *in-place* virtualization within the regular instruction stream. The wedge also includes a virtualization kit for platform devices, supporting transformation of memory operations that access device registers. Engineering cost for pre-virtualization are much lower than for para-virtualization.

2.1.3 Afterburning

The basic idea of pre-virtualization is to virtualize sensitive instruction in a user-level wedge rather than in the privileged hypervisor and to do this in an automated process. Sensitive operations are thus redirected to wedge functions. This is achieved by *afterburning* [LeV]. Afterburning provides two modes of operation, which produce different types of a virtualized guest OS binary. In *static* operational mode, the afterburner works quite similarly to para-virtualization. A guest OS binary is generated, which is dedicated to a specific hypervisor. The *dynamic* operational mode provides boot-time dynamic linking of the guest OS to a hypervisor. The generated guest OS binary may run on all hypervisors as well as on raw hardware. This is done by adding several no-operation instructions after a sensitive instruction and by appending a *patchup-table* to the guest OS binary. The patchup table contains the addresses of sensitive instructions. A hypervisor locates the sensitive instructions via the patchup-table and rewrites the instruction and the following no-operation instructions with the hypervisor-specific emulation code.

2.1.4 IA32 Architecture

The IA32 architecture has grown over years and from the i386 to the Pentium 4 the instruction set was expanded with every new hardware generation. The following section gives a short introduction of the instruction set of the Pentium 4 with the main focus on sensitive instructions as well as the paging, exception- and interrupt handling mechanisms. Robin and Irvine [RE00] explain the Pentium's sensitive instructions in detail.

Robin and Irvine [RE00] have identified 27 instructions that are sensitive but not privileged. One of them for example is the `POPF` instruction which restores the `EFLAGS` register from the stack. If the instruction is not executed in most privileged level, no exception is raised, but only parts of the register are restored.

The Pentium 4 supports two memory management concepts, segments and paging. An extensive explanation into both concepts is found in [Inta, Intb]. The host OS for this work, Windows NT, uses a variation of the basic flat segment model. All segment selectors, except CS, have constant values. An application is allowed to execute between 0 and 2GB, and the kernel has full control over the whole 4GB address space. A boot-time switch, "/3GB", exists in Windows NT to give applications control over a 3GB address space but it is rarely used and thus neglected for the implementation of a VMM. Paging can be used with or without segmentation. The IA32 architecture uses a two-level page translation. Three data structures, *page directory*, *page table* and *page* are used to translate virtual addresses into physical ones. The page directory holds up to 1024 page-directory entries. Each entry points to a page table. A page table contains up to 1024 page-table entries each pointing to a 4KB page of physical memory. A page is a 4KB, 2MB or 4MB flat address space. Paging is controlled by three flags in the processor's control register:

- The PG flag enables the page-translation mechanism.
- The PSE flag enables large page sizes: 4MB pages or 2MB pages (when the PAE flag is set)
- The PAE flag enables 36-bit physical addresses. This physical address extension can only be used when paging is enabled.

The physical address of the page table in use is held in the CR3 processor control register. Thus if the VMM controls load instructions into the CR3 register, it has full control over the paging mechanism; the current page table is therefore always known to the VMM. Furthermore, since every task uses its own page table, the current task is identified by the value in the CR3 register.

Interrupt and exception handling is done transparently to application programs and the operating system. Both are forced transfers of execution from the currently running program or task to a special procedure or task called a *handler*. Interrupts and exceptions occur at random time during the execution of a program. Interrupts are generated in response to signals from hardware or by a software program executing the `INT n` instruction. Exceptions are software caused, either by a processor-detected program-error, or software generated (a program executes the INTO,INT 3 or BOUND instruction) or by a machine check indicating an internal machine error. Exceptions are classified as *faults*, *traps* or *aborts*. A fault is an exception that can generally be corrected, and that, once corrected allows the program to be restarted with no loss of continuity. The return address (saved contents of CS and EIP registers) for the fault handler points to the faulting instruction. A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address of a trap points to the instruction to be executed after the trapping instruction. An abort does not always report the precise location of the instruction causing the exception. It does not allow restart of the program or task that caused the exception. With each exception or interrupt a identification number, called a *vector*, is associated. The range of a vector is 0 through 255. Vectors 0 through 31 are assigned to exceptions and the non-maskable interrupt (NMI); the remaining 32 through 255 are designated as user-defined interrupts. If more than one exception or interrupt is pending, the processor services them in a special order as prescribed by their priorities. Priorities are assigned hard-coded by the IA32 architecture from 1 to 8 with 1 being the highest and 8 being the lowest priority. The interrupt exception table (IDT) associates each exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt. The IDT is located using the IDTR register. This allows every task to have its own IDT and thus to have custom exception handlers.

2.2 Windows NT

Throughout this document the term *Windows NT* will always refer to Windows NT 5.0 and higher versions.

The user mode system component closest to the kernel is the NTDLL.DLL [Sol98]. Every supported subsystem (Win32, POSIX, OS/2) is built upon it. The API that utilizes that component is called the *native API*. Since Microsoft wishes to maintain the flexibility of the native API, its documentation is only in parts available to the public. However independent developers have collected much information about it and made this information publicly available [Neb00,Now05]. The following gives a short overview of the problems that occur when using Windows NT as a hypervisor. We describe the constraints of the Win32 API and the functions and mechanisms of the native API that are used in this work.

2.2.1 Using Windows NT as a hypervisor

Windows NT was not designed to be a hypervisor. Therefore it lacks many features that a VMM needs. The Windows NT kernel reserves the upper 2GB of an application's address space for internal use. The guest OS must be modified to execute only in the lower 2GB rather than in a 4GB address space. No functionality exists for trapping sensitive instructions. A user-mode VMM has to find a way to deal with them without executing kernel-mode code. Windows NT does not allow to trap system calls or interrupts. A user-mode VMM must trap them without changing the guest OS applications.

2.2.2 Constraints of the Win32 API

The Win32 API implements a mechanism called *structured exception handling*. Structured exception handling allows to handle hardware and software exceptions transparently to the running application. It is divided into two parts, *frame-based* exception handling and *vectored* exception handling.

Frame-based exception handling allows to handle the case that an exception occurs within a certain sequence of code. A frame-based exception handler has the form

```
__try {
    // guarded body
} __except(filter-expression) {
    // exception fall back
}
```

The __try keyword indicates a guarded body of code to the compiler; the filter-expression is a filter-function which handles the exception and the exception fall back code closes the application if the exception could not be handled by the filterexpression. The exception is only caught if generated within the guarded code body. The filter function receives information about the exception via a parameter pushed on the stack. If the stack pointer is not valid, in the sense that it points to a valid mapped address in the address space of an application, the filter function is not called and the application is immediately destroyed.

Vector based exception handlers are called regardless of the call frame. They are executed in the order they were added before any frame based handler. A vector based exception handler receives information about the exception via a parameter pushed on the stack. Again, if the stack pointer is not valid, the application is immediately destroyed. The dependance on a valid stack pointer is a fuzzy substantial constraint

```
PVOID AddVectoredExceptionHandler(
  ULONG HandlerNumber,
  PVECTORED_EXCEPTION_HANDLER VectoredHandler
);
```

Figure 2.1: Adding a vectored exception handler. VectoredHandler gets a pointer to the handler function, HandlerNumber sets the order in which the handler will be executed. The handler with the lowest handler number not equal to zero will be executed first.

of the Win32 exception handling, because an application has to ensure that the stack pointer is always valid, otherwise it will be destroyed.

The Win32 API allows a process to manipulate or determine the status of pages within its own address space or another one as long as it has the permissions to do so. Particularly the following operations are allowed:

- Reserve a range of a process's address space. This does not involve reserving physical storage.
- Committing a range of reserved pages in a process's address space.
- Specify access rights for a range of pages.
- Create a file-mapping from a range of pages.
- Map / Unmap pages from the file-mapping into a process's address space.

The problem with all these functions is that they only support a minimum granularity of 64k. Thus for example mapping pages of 4KB size is not possible. It is also impossible to manipulate the page addressed at 0x0 since this is prohibited by the Win32 subsystem.

2.2.3 Windows NT Native API

Because of the constraints of the Win32 API, exception handling requires a valid stack pointer, and a minimum mapping granularity of 64KB, the Windows NT native API must be used to virtualize certain IA32 operations. Every object in the native API, regardless of its type, is identified by an OBJECT_ATTRIBUTES structure. This structure defines naming, access rights and other security related attributes. In the following we give a detailed description of the native API functions and mechanisms used throughout this document.

Threads

Windows NT implements the concept of kernel level threads (KLT). Below is a description of the system services that create and manipulate thread objects.

- `NtCreateThread`: A thread object is created within an existing address space. This involves specifying the desired access to the newly created object as well as its context and a stack region.
- `NtOpenThread`: Opens a thread object with the desired access rights.
- `NtTerminateThread`: Terminates a previously created thread object. If a thread is the last one within a process and tries to terminate itself, an error status is returned.
- `NtSuspendThread`: Suspends execution of a thread object without terminating it.
- `NtResumeThread`: Resumes a previously suspended thread object.
- `NtGetContextThread`: Retrieves one or a group of values from the context of a thread including all registers contents.
- `NtSetContextThread`: Sets one or more values in the context of a thread object.

Processes

Windows NT uses processes as an identifier and container for several attributes of an address space.

- `NtCreateProcess`: Parameters include the desired access to the newly created object, inheritance from another process (not necessarily the creator), optionally a handle to an image section granting execute access, a debugging port and an exception port. The created process does not contain any thread.
- `NtOpenProcess`: Specified parameters are the desired access to the opened process object and either a object attribute that identifies the process by name or its client id.
- `NtTerminateProcess`: Terminates a process object including all thread objects it contains.

Virtual Memory

The virtual memory routines of the native API utilize the minimal supported page granularity of the underlying architecture. Therefore on IA32 the granularity is 4KB.

- `NtAllocateVirtualMemory`: Reserves and optionally commits a range of pages within a process's address space.
- `NtFreeVirtualMemory`: Decommits or releases a range of pages in a process's address space.
- `NtRead-` / `NtWriteVirtualMemory`: Reads or writes a specified amount of memory from a processes address space.
- `NtProtectVirtualMemory`: Protects a range of pages within a process's address space. Protection attributes cover read and write access rights. Setting executable rights is only supported in Windows XP SP2 and Windows Server 2003.
- `NtFlushVirtualMemory`: Flushes a range of pages that are mapped to a file.

Sections

Sections are objects that can be mapped into the address space of a process. They are created out of file objects whereby the system backed memory (swap space) is also treated as a file. Windows NT uses sections to implement shared memory mechanisms.

- `NtCreateSection`: Creates a section object from a file. If the file handle is NULL, the object is created using the system backed memory.
- `NtOpenSection`: Opens a section object identified by a name.
- `NtMapViewOfSection`: Maps a view of a section to a range of virtual addresses.

Ports

Windows NT implements local procedure calls (LPC) using port objects. A port acts like a mailbox with one sender and one receiver bound to the it. The sender and the receiver exchange messages through this mailbox. Port objects must be used to receive and process messages sent by the operating system, such as debug and exception messages. A port message consists of several items, including the message type and size, the size of the appended data and optionally shared memory sections are specified. The amount of data that can be transferred with a port message is limited to 300 Bytes.

- `NtCreatePort`: Creates a port object with a maximal supported message and data size. Optionally a specified name is assigned to the object.
- `NtConnectPort`: Creates a port object connected to a named port.
- `NtListenPort`: Listens on a port for a connection request message.
- `NtAcceptConnectPort`: Accepts or rejects a connection request to a port object.

- `NtCompleteConnectPort`: Completes the port connection process.
- `NtRequestPort`: Sends a request message to a port.
- `NtRequestWaitReply`: Sends a request message to a port and waits for a reply.
- `NtReplyPort`: Sends a reply message to a port.
- `NtReplyWaitReplyPort`: Sends a reply message to a port and waits for a reply message.
- `NtReplyWaitReceivePort`: Optionally sends a reply message to a port and waits for a incoming message.

Native Applications

Since the Win32 API does not fully expose the functionality of the native API, Win32 applications may also use the native API. This is possible to a certain extent, but often conflicts with the Win32 API, for example the exception port of a Win32 process is always assigned to the general function port of the Win32 API. Besides this, the Win32 API restricts the layout of a Win32 process's address space, the page addressed at 0x0 for example is reserved and protected by the Win32 API. These limitations make it necessary for a VMM to use *native applications*. Native applications are applications not running on top of a subsystem of Windows NT (Win32, etc) but on top of the NTDLL.DLL. They are built using the Windows NT driver development kit (NTDDK) build utility [Rus98]. Native applications can not be loaded using a single system call.

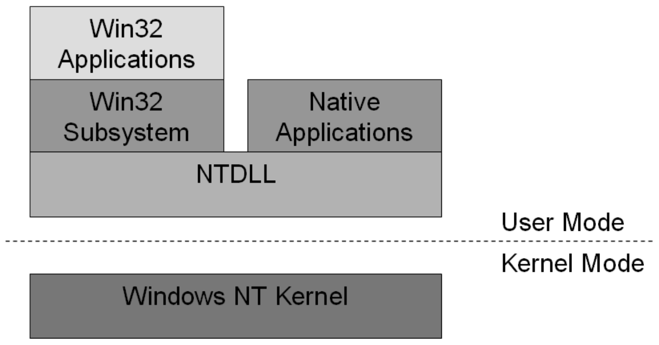

Figure 2.2: Native Applications. Native applications run directly on the NTDLL library. Win32 applications run on the Win32 API which in turn is a subsystem of the NTDLL library.

Instead, several steps are needed to load and start a native applications.

1. Create a section object from the executable file.
2. Create a process and assign the executable image to the process.
3. Create a thread with initial parameters (stack, IP) from the executable.
4. Start the thread.

After these steps a native application is running in user mode. The standard libraries shipped with the NTDDK are not usable with native applications. They have to be written from scratch for the use of a native application.

Chapter 3

Related Work

In this chapter we describe User-mode Linux, an attempt to run Linux in user-mode of another Linux system, and Cooperative Linux which executes Linux in kernel-mode on Windows NT.

3.1 User-mode Linux

User-mode Linux (UML) [Dik00] is a port of the Linux kernel that runs in the user-mode of a Linux system. It uses a para-virtualized relinked Linux kernel. Linux processes created by the UML kernel (UML process) run identically on the UML kernel as they would on a native Linux kernel. No recompilation of applications is required.

3.1.1 Executing a VMM in user-mode on Linux

Several problems occur when executing a VMM in user-mode on Linux. First, Linux reserves the upper 1GB of an application's address space for internal use. Second, Linux receives system calls via a special interrupt vector. These can not be trapped and must be caught using the debugging API of Linux. Third, sensitive instructions can not be allowed to execute. They must be emulated by the VMM.

3.1.2 Design

UML solves the problem of the reduced address space by relinking the kernel binary to operate between 2.5GB and 3GB. This decreases the usable virtual memory size for UML processes to 2.5GB. Sensitive instructions are trapped using para-virtualization. The Linux kernel sources are rewritten to execute emulation code instead of sensitive instructions. If an UML process calls a system call, the UML tracing thread which is registered as a debugger for every UML process is notified. The tracing thread then rewrites the number of the system call with the number of `getpid`. Because getpid only delivers the id of the currently running process, it is harmless to let the host kernel execute it. The *real* system call, the one which the UML process originally called is then executed by the UML kernel. This allows Linux applications to run unchanged on a real Linux and a User-mode Linux system.

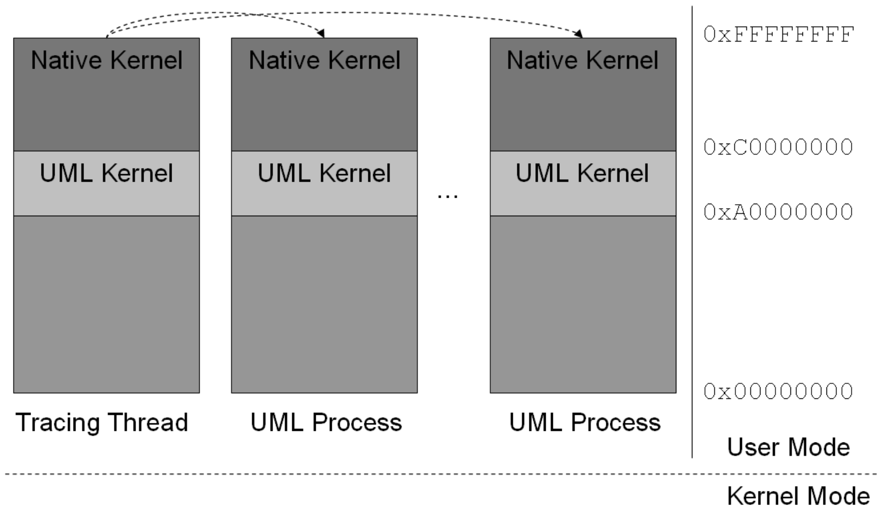

Figure 3.1: User-Mode Linux Design. The tracing thread is responsible for redirecting signals to UML processes and for tracing system calls from UML processes.

3.1.3 Analysis

The design of User-Mode Linux doesn't compromise the host kernel. Therefore it is as stable and secure as the guest OS it is running. Nevertheless the tracing thread is a severe performance bottleneck since every trap of a UML process into kernel-mode needs to be traced [Dik00, Ste02].

3.2 Cooperative Linux

Cooperative Linux (coLinux) [Alo04] virtualizes Linux in kernel-mode on Windows NT. The Linux kernel is changed at source code level to coexist with the Windows NT kernel. Cooperative Linux starts a driver in kernel mode which is frequently called by a Windows NT process from user-mode. This process is called the *Super process*. The coLinux driver is responsible to switch between the Windows NT and the Linux kernel. Since both kernels, the Windows NT and the Linux kernel, assume full control over the physical memory it must be somehow split between the both. coLinux achieves this by reserving a fixed continuous set of physical pages for the host OS driver.

3.2.1 Analysis

CoLinux performs very fast compared to UML [Alo04]. Nevertheless, it seriously affects security and stability of the underlying Windows NT system. If the virtualized Linux crashes, the whole system might crash. Additionally, administrative access on the virtualized Linux may potentially lead to administrative access on the Windows NT system.

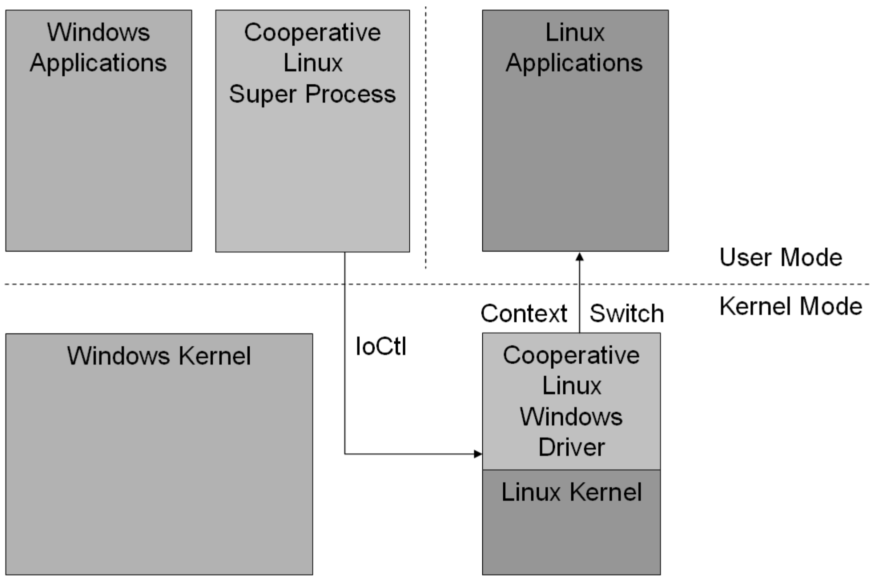

Figure 3.2: Cooperative Linux Design. The Super process frequently calls the Cooperative Linux driver via IoCtl. The driver then switches the complete context of the CPU to the virtualized Linux.

Chapter 4

Proposed Solution

Our design goals are to nearly reach the speed of coLinux while maintaining security and stability of the UML approach. Figure 4.1 points out the involved threads and processes and the communication flow between them.

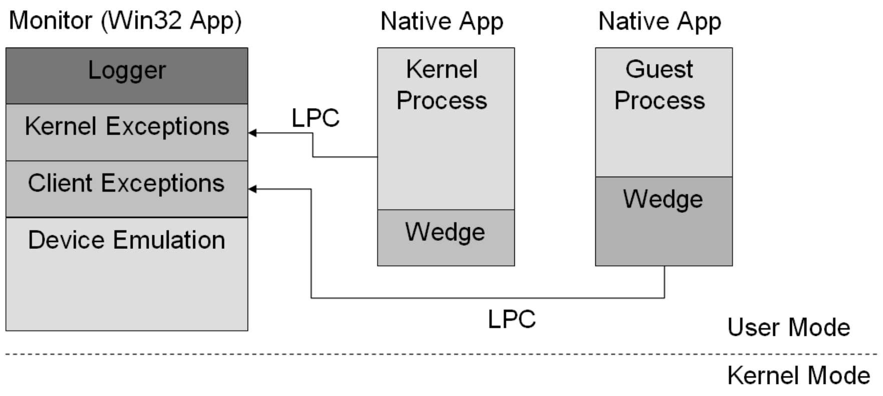

Figure 4.1: Windows NT Native VMM Design. The Monitor itself is a Win32 application, the guest OS kernel and processes run as native applications. They communicate with each other via named ports. The guest OS kernel and processes output to the logger using LPC. Exceptions and software-generated interrupts are sent via the exception port to the responsible exception handling thread by the underlying Windows NT system.

Mapping the 400MB kernel image into every guest OS processes (GP) address space is very ineffective and results in severe performance decrease. Another method has to be found that ensures that the guest OS kernel (GK) memory is protected from GP accesses but it must be possible for the GK to access GP memory. Thus the GK and the GPs are implemented in several Windows NT processes with one Windows NT thread in each process. These threads are called *guest threads* since they execute either guest OS kernel or user code. At a given time, only one guest thread, either the GK or one of the GPs is running. All other guest threads are suspended until an interrupt or an exception occurs. Instead of one tracing thread, an exception handler thread for each guest thread is created, which listens to the respective exception port and handles exceptions and software-generated interrupts. The monitor does not rely on a specific address space layout and is implemented within a Win32 application. For the GK and the GP continuous virtual memory is needed starting at 0x0. Furthermore the exception port is required to catch exceptions. This is not guaranteed by the Win32 API and therefore the GK and the GPs are implemented as native applications. Data about the virtual CPU is needed by the exception handler threads as well as the wedges. Thus the virtual CPU data is stored in a memory page, the VCPU page, which is shared between the monitor, the guest OS kernel and the guest OS processes.

4.1 Address Space Layout

Virtual memory of 2GB is available to user-mode tasks in Windows NT. Thus the available virtual memory for the guest OS kernel (GK) is shortened to 0.4GB instead of 1GB when running in native mode. The wedge only needs a minimal amount of virtual memory therefore the GK is linked between 0x40000000 and 0x70000000. The remaining virtual memory of approximately 0.1GB is reserved for the wedge. The guest OS processes (GP) need to execute below the GK's link address in the range of 0x0 to 0x40000000. As on a native machine, the GK has access to virtual memory in the range of 0x0 to 0x70000000. Figure 4.2 shows an overview of the virtual memory layout.

Virtual Memory Area	Size	Usage
`0x00000000 - 0x3FFFFFFF`	1.5GB	Guest OS Process
`0x40000000 - 0x6FFFFFFF`	0.4GB	Guest OS Kernel
$\geq$ `0x70000000`	0.1GB	Wedge, VCPU page

Figure 4.2: Layout of Guest OS Kernel/Process Virtual Memory. The GK is linked to 0x40000000 and uses memory between 0x0 and 0x70000000. A GP executes between 0x0 and 0x40000000. The remaining memory is used by the wedge code and the shared VCPU page.

4.2 Physical Memory

Every operating system assumes that it has full control over the physical memory of a machine. Thus a VMM has to emulate real physical memory to the guest OS kernel (GK). UML does this by mapping views from a file into the GK's and the guest OS processes' address space. This work introduces a slightly different approach by using a file backed by the virtual memory system of Windows NT. No `"real"` file is created, instead the system uses either available physical memory or swap space to satisfy requests on the file. The physical memory file object (PM) has a global name and is therefore accessible by all processes with appropriate permissions. The monitor as the creator and the GK as well as all GPs access views of the PM. Section 4.5 explains how views of the PM are mapped to pages in the GK and the GPs.

4.3 IRQ and exception handling

IRQ and exception handling is essential for modern operating systems to implement preemption and paging. The duty of a VMM is, to handle IRQs and exceptions as transparent to the guest OS kernel (GK) and the guest OS processes (GP), as the underlying architecture would. The GK and the GPs must not see any difference between executing on a native or a virtualized system. As explained in Section 2.1.4, the sources for interrupts are divided into two classes, software-generated and hardware generated interrupts. Software-generated interrupts occur either indirectly (exception thrown by the CPU) or directly (e.g. by executing `INT n`).

Hardware-Generated Interrupts are implemented by the device emulation threads in the monitor (4.4). For that purpose an i8259 interrupt controller is emulated. Whenever an interrupt is pending in a device it signals this using a named port to the interrupt controller thread. This thread then uses the monitor's interrupt routines to deliver it. This involves for example checking whether interrupts are actually enabled or masked. The priority of a pending interrupt is also checked, to assure the same execution order as on the IA32 architecture. If all checks are passed the interrupt controller thread (ICT) first stops execution of the currently running guest thread. This simulates the signaling of an interrupt to the processor [Intb]. Afterwards the Interrupt Descriptor Table (IDT) is searched for a matching handler. If none is found, the ICT displays an error message and the virtual system is stopped. If a handler is found, it is executed as prescribed by the IA32 architecture including correct pushing of registers on the stack. The ICT looks up the internal guest thread table to find the one who should execute the interrupt handler. The context of the guest thread is then set and the handler is executed.

Software-Generated Interrupts happen implicitly during execution of a guest thread. They are first handled by the associated exception threads in the monitor. Except for page fault exceptions (see 4.5) all exceptions are immediately passed to the exception handler located in the IDT. For that purpose the exception thread searches the IDT for a proper handler. If none is found, an error message is displayed and execution is stopped. If a handler is found and no other interrupt is executed, the running guest thread is suspended. The exception thread looks up the guest thread table to find the guest thread which must execute the handler routine. Now the context of the found guest thread is set appropriate after saving the context like prescribed by the IA32 architecture [Intb]. When the handler routine is done, the exception thread continues execution of its associated guest thread.

4.4 Device Emulation

A detailed description of all emulated devices is beyond the scope of this work. Instead, details of the general device design are explained. An operating system on the IA32 architecture communicates with devices via I/O Ports and Memory Managed-I/O. This kind of communication is handled by the wedges in the GK and the GPs. Devices in turn communicate with the OS via interrupts. Shared memory is implemented transparently by the paging system.

Writing to or reading from an I/O Port is done by using the IN and OUT instructions. These are sensitive instructions and are thus rewritten by the afterburner. The wedge then redirects writes or reads into LPC messages to the corresponding device threads.

Device threads use the interrupt controller thread to communicate with the guest OS. Data is then transmitted using the PCI emulation thread.

4.5 Paged Memory Virtualization

Paging is a key feature of every modern operating systems. Thus paged memory virtualization is a very important part when designing a VMM. The IA32 architecture implements paging via page-tables. The address of the page-table in use is stored in the CR3 register; paging is switched on and off via the paging bit in the CR0 register. To emulate this fact, the wedge flushes the complete guest OS address space after a CR3 write (the virtual memory in the range from 0x0 to 0x70000000). Thereby the wedge emulates the TLB flush which occurs automatically if the CR3 register is written.

If a page-fault exception occurs, the handling thread first checks the CR0 register. If paging is not enabled, the handler assumes a physical access and maps the page idempotently from the physical memory file (PM) to the faulting guest thread. However if paging is enabled, the handler has to search the page-table for the physical page frame [Intb]. If a physical page frame is found, the handler maps it from the PM into the virtual address space of the faulting guest thread. If the page frame is not found, the page fault exception handler routine is invoked (See section 4.3 IRQ and exception handling).

4.6 Scheduling

The CR3 register identifies the page-table in use. Since two different guest OS Processes (GP) have different CR3 values the register value may be taken as an identifier for guest threads. The wedge maintains a list in which the guest thread ids are associated with CR3 values. After the guest OS kernel (GK) switches to user-mode using IRET this table is searched for a fitting guest thread. If none is found, a new Windows NT process is created with a new guest thread in it. The problem is that the guest OS doesn't inform the processor when a process is destroyed. There is no way for a VMM to know if a GP has already been destroyed by the guest OS. Therefore the VMM has to do a garbage collection in which all guest threads except the running one is destroyed. This doesn't result in the loss of information because all needed information for a process is stored in physical memory which is never deleted.

Chapter 5

Implementation

Though a complete implementation of a native VMM is beyond the scope of this work, the basic implementation of the paging system and the wedge will be discussed in the following. Starting point is an afterburned Linux 2.6 binary in dynamic mode (2.1.3 Afterburning).

5.1 Building Native Applications

Native applications are built using the build utility of Microsoft's Windows NT Driver Development Kit (NTDDK) [Rus98]. Behavior of the build utility is controlled by a SOURCES file.

```
TARGETNAME=testNativeApp
TARGETPATH=obj
TARGETTYPE=PROGRAM
SOURCES=testNativeApp.cpp
LINK_FLAGS=/NODEFAULTLIB
UMENTRYABS=NtProcessStartup
UMBASE=0x70010000
UMTYPE=NT
```

Figure 5.1: Example listing of a SOURCES file to build native applications

UMTYPE specifies the type of application to be built and must be set to NT. UMENTRYABS sets the entry point of the executable which is not predefined for native applications. Also important is not to use the default libraries since this will result in an undefined behavior of the application (e.g. calls to printf fail if no heap was initialized). During the build process, a executable file is created that must be loaded and started manually because no Windows NT service exists to do that.

5.2 Loading Native Applications

Now that the native binary is built, a customized loader must load and start it. During this step the exception port of the application is also assigned to an exception handling thread. This is done in several steps:

- Open the executable image using `NtOpenFile`.
- Create a section from the opened file with `NtCreateSection` and close the file using `NtClose` afterwards.
- Create a port with `NtCreatePort`. This port will be later assigned as the exception port of the new process.
- Create a process using `NtCreateProcess`. Important parameters are the executable section and the port that were previously created.
- Read the desired stack region and the entry point from the executable section. Afterwards the section object is closed.
- Allocate the stack region in the virtual memory of the new process and create a guard page to protect it from stack overflows.
- Create a thread context for the initial thread (Figure 5.2). EIP and ESP register values are set with the previously gathered information from the executable binary.

```
context.SegGs= 0
context.SegFs= 0x38
context.SegEs= 0x20
context.SegDs= 0x20
context.SegSs= 0x20
context.Eflags= 0x3000
```

Figure 5.2: Initial Context for a Windows NT thread

- Create an initially suspended thread in the new process with the allocated stack and predefined context using `NtCreateThread`.
- Create the process parameters using `RtlCreateProcessParameters` and copy them into the new process address space.
- Resume the suspended thread with `NtResumeThread`.

5.3 Sending and Receiving Messages through a Named Port

If two threads are not able to exchange handles, because they are running in different address spaces, they have to use a named port to communicate with each other. The name must be unique within the name space of the local system, besides that it may be chosen freely.

The communication is divided into a server and a client part. The server listens at the port and accepts or rejects incoming client requests.

Server Part The server first calls `NtListenPort`, this blocks the thread until a connection request is received on the port. After receiving the request, the server has two options. The first one is to establish the connection using the named port. This is a bad choice since it blocks the port throughout the connection time. The second method is to create a thread and a second port. The newly created thread establishes the connection using the second, unnamed port. To do this, the server calls `NtAcceptConnectPort` with the second port and the request message as parameters and after that `NtCompleteConnectPort`. Now a connection between the client and the server through the second port is established. The server enters a message loop and blocks to receive messages from the client using e.g. `NtReplyWaitReceivePort`.

Client Part The client calls `NtConnectPort` to connect to the server through the named port. After that the client sends messages to the server using e.g. `NtRequestWaitReplyPort` with the opened port object as parameter.

5.4 Getting output from a Native Application

As mentioned earlier (5.1) native applications must not use the default libraries shipped with the NTDDK. Therefore a method has to be found to get output from native applications. This is done using a logger thread in the monitor process. The logger thread listens to a named port (AFTERBURN_LOGGING) in a message loop. Every time a message is received, the data section of the message is interpreted as a string which in turn is displayed in the console of the monitor.

Chapter 6

Conclusion

Building a native Windows NT VMM is feasible. Although a complete implementation is beyond the scope of this work the proposed design covers all important aspects of the VMM. Our design shows how to implement paging mechanisms and synchronous interrupt delivery. With the basic implementation, we introduced a starting point which implements the fundamental mechanisms of the VMM. Future work will include a complete implementation of the VMM. The first step towards a complete implementation will be to implement all afterburning functions to get the Linux Kernel booting into user mode. Later the scheduling logic must be implemented including garbage collection and exception handling.

Bibliography

[Alo04] Dan Aloni. Cooperative linux. In *Proceedings of the Linux Symposium Volume One*, 2004.

[Dik00] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.

[Gol72] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[Gol74] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6), 1974.

[Inta] Intel Corporation. *Intel IA32 Software Developer's Manual Volume 1: Basic Architecture*.

[Intb] Intel Corporation. *Intel IA32 Software Developer's Manual Volume 3: System Programming Guide*.

[LeV] Joshua LeVasseur. Afterburning and the accomplishment of virtualization. *Whitepaper*.

[Neb00] Gary Nebbett. *Windows NT/2000 Native API Reference*. Sams, 2000.

[Now05] Tomasz Nowak. Undocumented functions for windows nt/2000. `http://undocumented.ntinternals.com`, 2005.

[Pop74] Gerald J. Popek. Formal requirements for virtualizable third generation architectures. In *Proceedings of the 4th Symposium on Operating System Principles*, 1974.

[RE00] John Scott Robin and Cynthia E. Ervine. Analysis of the intel pentium's ability to support secure virtual machine monitor. In *Proceedings of the USENIX 2000 Annual Technical Conference*, 2000.

[Rus98] Mark Russinovich. Inside native applications. `www.sysinternals.com`, 1998.

[Sol98] David A. Solomon. *Inside Windows NT Second Edition*. Microsoft Press, 1998.

[Ste02] Udo Steinberg. Fiasco microkernel user-mode port, 2002.