TECHNISCHE UNIVERSITÄT DRESDEN

Department of Computer Science Institute of Systems Architecture, Operating Systems Group

VIRTUALIZATION

Julian Stecklina (jsteckli@os.inf.tu-dresden.de)

Dresden, 2013/12/3

00 Goals

Give you an overview about:

- virtualization and virtual machines *in general*,
- hardware virtualization on x86,
- our research regarding virtualization.

We will *not* discuss:

- lots and lots of details,
- language runtimes,
- how to use XEN/KVM/. . .

00 Outline

What's Virtualization?

Very Short History

Virtualization on x86

Example: L4Linux

Example: NOVA

Example: Karma VMM

01 Outline

What's Virtualization?

Very Short History

Virtualization on x86

Example: L4Linux

Example: NOVA

Example: Karma VMM

01 Starting Point

You want to write a new operating system that is

- secure,
- trustworthy,
- small,
- fast,
- fancy.

but . . .

01 Commodity Applications

Users expect to run all the software they are used to (“legacy”):

- browsers,
- Word,
- iTunes,
- certified business applications,
- new (Windows/DirectX) and ancient (DOS) games.

Porting or rewriting all is *infeasible*!

01 One Solution: Virtualization

"By virtualizing a commodity OS [..] we gain support for legacy applications, and devices we don't want to write drivers for."

"All this allows the research community to finally escape the straitjacket of POSIX or Windows compatibility [...]"

[Ro07]

01 Virtualization

virtual existing in essence or effect though not in actual fact

`http://wordnetweb.princeton.edu`

"All problems in computer science can be solved by another level of indirection."

Butler Lampson, 1972

01 Emulation

Suppose you develop for a system *G* (*guest*, e.g. an ARM-base phone) on your workstation *H* (*host*, e.g., an x86 PC). An *emulator* for *G* running on *H* precisely emulates *G*'s

- CPU,
- memory subsystem, and
- I/O devices.

Ideally, programs running on the emulated *G* exhibit the same behaviour as when running on a real *G* (except for timing).

01 Emulation (cont'd)

The emulator

- simulates every instruction in software as its executed,
- prevents direct access to *H*'s resources from code running inside *G*,
- maps *G*'s devices onto *H*'s devices,
- may run multiple times on *H*.

01 Mapping *G* to *H*

Both systems may have considerably different

- instructions sets and
- hardware devices

making emulation slow and complex (depending on emulation fidelity).

01 $G = H$

If host and emulated hardware architecture is (about) the same,

- interpreting every executed instruction seems *not necessary*,
- near-native execution speed should be possible.

This is (easily) possible, if the architecture is *virtualizable*.

01 From Emulation to Virtualization

A virtual machine is defined to be an

"efficient, isolated duplicate of a real machine."

(Popek, Goldberg 1974)

The software that provides this illusion is the *Virtual Machine Monitor* (VMM, mostly used synonymous with *Hypervisor*).

01 Idea: Executing the guest as a user process

Just run the guest operating system as a normal user process on the host. A virtual machine monitor process needs to handle:

01 Idea: Executing the guest as a user process

Just run the guest operating system as a normal user process on the host. A virtual machine monitor process needs to handle:

- address space changes,
- device accesses,
- system calls,
- ...

Most of these are not problematic, because they trap to the host kernel (SIGSEGV).

01 A hypothetical instruction: OUT

Suppose our system has the instruction `OUT` that writes to a device register in **kernel** mode.

How should it behave in **user** mode?

Option 1:
Just do nothing.

Option 2:
Cause a trap to kernel mode.

01 A hypothetical instruction: OUT

Suppose our system has the instruction `OUT` that writes to a device register in **kernel** mode.

How should it behave in **user** mode?

Option 1:
~~Just do nothing.~~

Option 2:
Cause a trap to kernel mode.

Otherwise device access cannot be (easily) virtualized.

TECHNISCHE UNIVERSITÄT DRESDEN

01 Virtualizable?

. . . is a property of the *Instruction Set Architecture* (ISA).

A *sensitive* instruction

- *changes* or
- *depends* in its behavior

on the processor's configuration or mode.

A *privileged* instruction causes a *trap* (unconditional control transfer to privileged mode) when executed in user mode.

01 Trap & Emulate

If all sensitive instructions are privileged,
a VMM can be written.

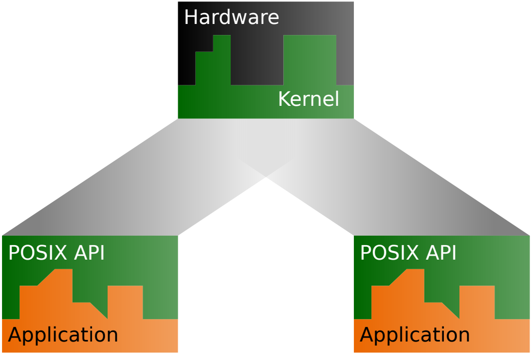
- execute guest in unprivileged mode,
- emulate all instructions that cause traps.

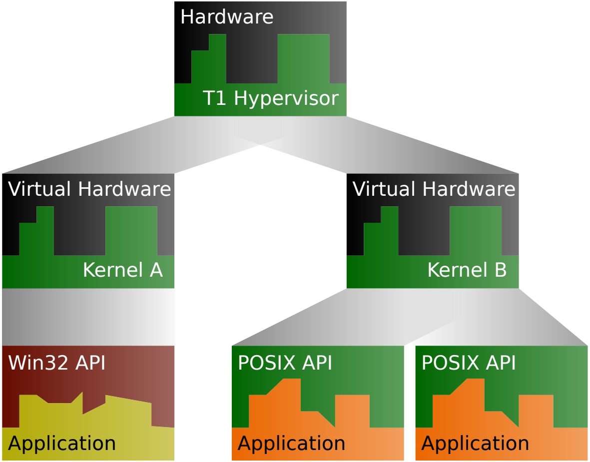
01 Trap & Emulate (cont'd)

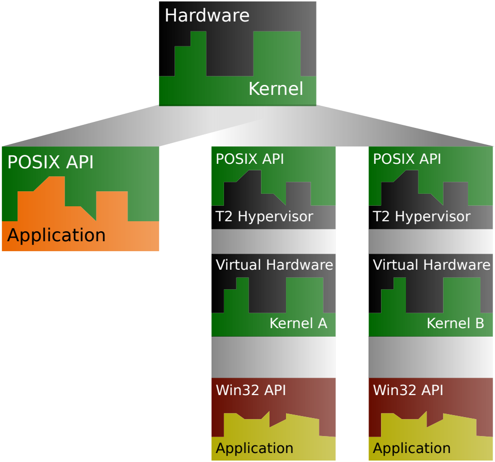
Will be topic of seminar on January 21st:

Formal Requirements for
Virtualizable Third-Generation Architectures
`http://portal.acm.org/citation.cfm?id=361073`

01 Where to put the VMM?




Hardware
T1 Hypervisor
Virtual Hardware
Kernel A
Virtual Hardware
Kernel B
Win32 API
Application
POSIX API
Application
POSIX API
Application



Hardware
Kernel
POSIX API
Application
POSIX API
T2 Hypervisor
Virtual Hardware
Kernel A
Win32 API
Application
POSIX API
T2 Hypervisor
Virtual Hardware
Kernel B
Win32 API
Application

01 Type 1 vs. Type 2

Type 1 are implemented on the bare metal (*bare-metal hypervisors*):

- no OS overhead
- complete control over host resources
- high maintainance effort

Popular examples are

- Xen,
- VMware ESXi.

01 Type 1 vs. Type 2 (cont'd)

Type 2 hypervisors are an extension (kernel module, application) to an existing OS (*hosted hypervisors*):

- doesn't reinvent the wheel
- performance may suffer
- usually need kernel support for access to CPU's virtualization features

Popular examples are

- KVM,
- VMware Server/Workstation,
- VirtualBox,
- ...

01 Paravirtualization

Why all the trouble? Just “port” a guest operating system to the interface of your choice.

Paravirtualization (in contrast to faithful virtualization) can

- provide better performance,
- simplify VMM

but at a maintainance cost and you need the source code!

Examples are Usermode Linux, MkLinux, L4Linux, Xen/XenoLinux, DragonFlyBSD VKERNEL, . . .

Compromise: Use paravirtualized drivers for I/O performance (KVM virtio, VMware).

01 Reimplementation of the OS Interface

Why deal with the OS kernel at all? Reimplement its interface! E.g. *wine* reimplements (virtualizes) the Windows ABI.

- Run unmodified Windows binaries.
- Windows API calls are mapped to Linux/FreeBSD/Solaris/MacOS X equivalents.
- Huge moving target!

Can also be used to recompile Windows applications as native applications linking to *winelib* $\Rightarrow$ API "virtualization"

01 Recap

- *Virtualization* is an overloaded term. Classification criteria:
  - **Target**
    real hardware, OS API, OS ABI, . . .
  - **Emulation vs. Virtualization**
    Interpret some or all instructions?
  - **Guest Modifications?**
    Paravirtualization

01 Recap (cont'd)

- A (Popek/Goldberg) *Virtual Machine* is an
  - efficient,
  - isolated
  - duplicate of a real machine.
- The software that implements the VM is the *Virtual Machine Monitor* (hypervisor).
- Type 1 ("*bare-metal*") hypervisors run as kernel.
- Type 2 ("*hosted*") hypervisors run as applications or kernel modules on a conventional OS.

02 Outline

"Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications."

"Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications."

Survey of Virtual Machine Research
Robert P. Goldberg
1974

02 Early History: IBM

02 Early History: IBM

Virtualization was pioneered with IBM's CP/CMS in ~1967 running on System/360 and System/370:

CP
: Control Program
  provided System/360 virtual machines.

CMS
: Cambridge Monitor System (later Conversational Monitor System)
  single-user OS.

At the time more flexible and efficient than time-sharing multi-user systems.

02 Early History: IBM (cont'd)

CP encodes guest state in a hardware-defined format.

SIE Start Interpretive Execution (instruction)
runs the VM until a trap or interrupt occurs. CP resume control and handles trap.

CP provides:

- memory protection between VMs,
- preemptive scheduling.

Gave rise to IBM's VM line of operating systems.
First release: 1972
Latest release: z/VM 6.3 (Jul 23, 2013)

02 Virtualization is Great

- Consolidation
  - improve server utilization
- Isolation
  - isolate services for security reasons or
  - because of incompatibility
- Reuse
  - run legacy software
- Development

. . . but was confined to the mainframe world for a very long time.

. . . fast forward to the late nineties . . .

03 Outline

03 Is x86 Virtualizable?

x86 has several virtualization holes that violate Popek&Goldberg requirement.

- Possibly too expensive to trap on every privileged instruction.
- `popf` (pop flags) silently ignores writes to the Interrupt Enable flag in user mode. Should trap!
- More in the seminar.

03 VMware Workstation: Binary Translation

First commercial virtualization solution for x86, introduced in ˜1999. Overcame limitations of the x86 architecture:

- translate problematic instructions into appropriate calls to the VMM on the fly
- can avoid costly traps for privileged instructions

Provided decent performance but:

- requires complex runtime translation engine

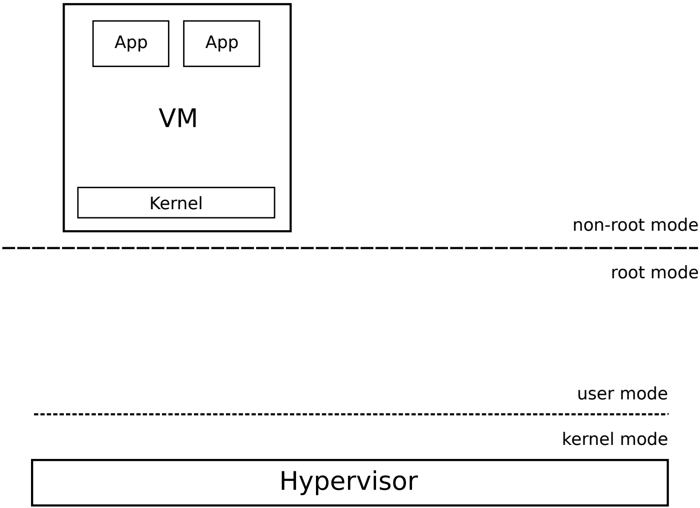
Other examples: KQemu, Virtual Box, Valgrind

03 Hardware Support for Virtualization

Late Pentium 4 (2004) introduced hardware support for virtualization: Intel VT. (AMD-V is conceptually very similar)

- root mode vs. non-root mode
  - duplicates x86 protection rings
  - root mode runs hypervisor
  - non-root mode runs guest
- situations that Intel VT cannot handle trap to root mode (**VM Exit**)
- special memory region (VMCS) holds guest state
- reduced software complexity

Supported by all major virtualization solutions today.


App
App
VM
Kernel
non-root mode
root mode
user mode
kernel mode
Hypervisor

03 Instruction Emulator

Intel VT and AMD-V still require an instruction emulator, e.g. for

- running 16-bit code (not in AMD-V, latest Intel VT),
  - BIOS
  - boot loaders
- handling memory-mapped IO (need to emulate instruction that caused a page fault)
  - realized as non-present page
  - emulate offending instruction
- ...

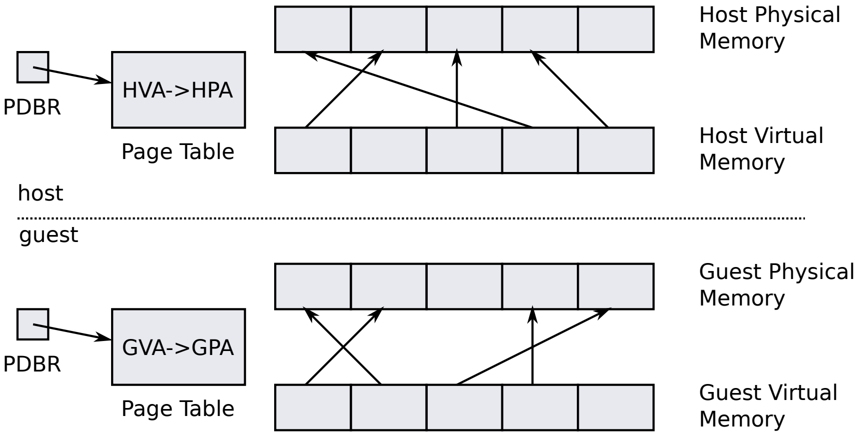
03 MMU Virtualization

Early versions of Intel VT do not completely virtualize the MMU. The VMM has to handle guest virtual memory.

Four different types of memory addresses:

HPA	Host Physical Address
HVA	Host Virtual Address
GPA	Guest Physical Address
GVA	Guest Virtual Address

Usually GPA = HVA or other simple mapping (offset).

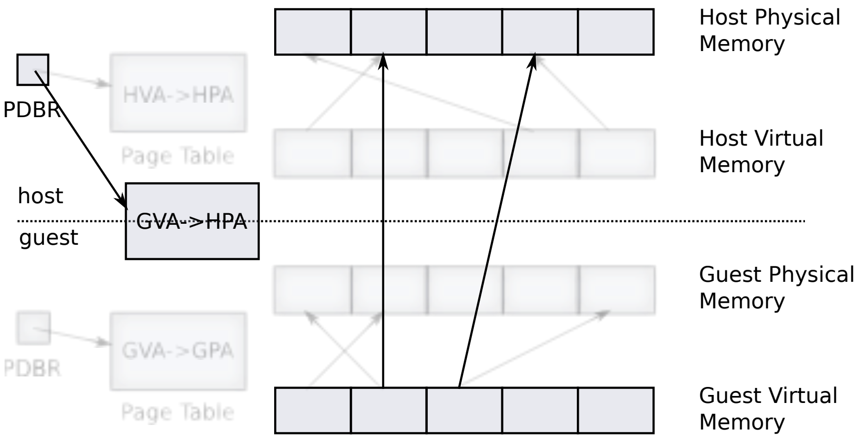


Host Physical
Memory
HVA->HPA
PDBR
Page Table
Host Virtual
Memory
host
guest
Guest Physical
Memory
GVA->GPA
PDBR
Page Table
Guest Virtual
Memory

03 Shadow Page Tables

If the hardware can handle only one page table, the hypervisor must maintain a shadow page table that

- merges guest and host page table (maps from GVA to HPA),
- must be adapted on changes to virtual memory layout.


Host Physical Memory
Host Virtual Memory
Guest Physical Memory
Guest Virtual Memory
PDBR
HVA->HPA
Page Table
host
guest
GVA->HPA
PDBR
GVA->GPA
Page Table

03 Shadow Paging in a Nutshell

1. page fault in guest (GVA)
2. traps to VMM
3. parse guest page tables (GVA $\Rightarrow$ GPA)

4. mapping found
5. parse host page table (HVA $\Rightarrow$ HPA)
6. create shadow entry
7. resume guest

4. no mapping found
5. resume guest with page fault

03 Drawbacks of Shadow Paging

Maintaining Shadow Page Tables causes significant overhead, because they need to be updated or recreated on

- guest page table modification,
- guest address space switch.

Certain workloads are penalized. Shadow Page Tables can be implemented as software TLBs.

03 Nested Paging

Introduced in the Intel *Nehalem* (EPT) and AMD *Barcelona* (Nested Paging) microarchitectures, the CPU can handle

- guest and
- host page table

at the same time. Can reduce VM Exits by *two orders of magnitude*, but introduces

- measurable constant overhead ($< 1\%$)

03 Nested Paging (cont'd)

Event	Shadow Paging	Nested Paging
vTLB Fill	181,966,391	
Guest Page Fault	13,987,802	
CR Read/Write	3,000,321	
vTLB Flush	2,328,044	
INVLPG	537,270	
Hardware Interrupts	239,142	174,558
Port I/O	723,274	610,589
Memory-Mapped I/O	75,151	76,285
HLT	4,027	3,738
Interrupt Window	3,371	2,171
Sum	**202,864,793**	**867,341**
Runtime (seconds)	645	470
Exit/s	**314,519**	**1,845**

(Linux Kernel Compile)

04 Outline

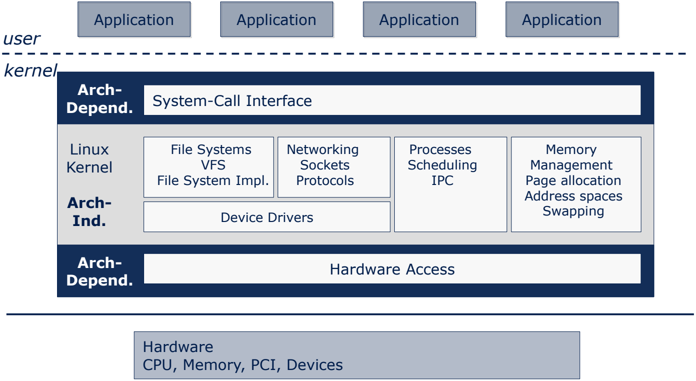
04 L4Linux

. . . is a paravirtualized Linux first presented at SOSP'97 running on the original L4 kernel.

- L4Linux predates the x86 virtualization hype
- L4Linux 2.2 supported MIPS and x86
- L4Linux 2.4 first version to run on L4Env
- L4Linux 2.6 uses Fiasco.OC's paravirtualization features

The current status:

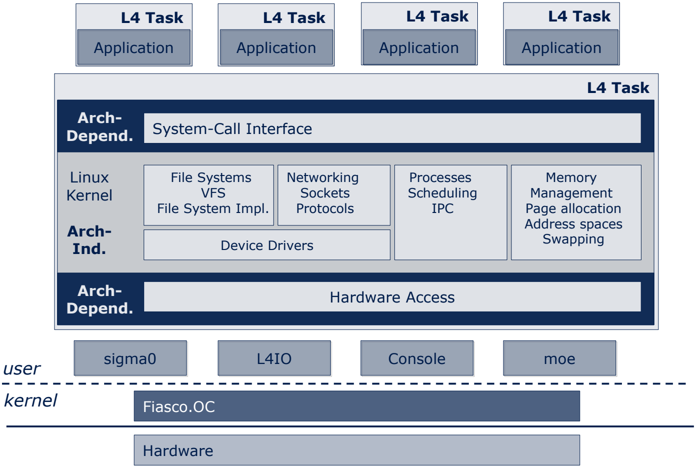
- based on Linux 3.11
- x86 and ARM support
- SMP


Application
Application
Application
Application
user
kernel
Arch-
Depend.
System-Call Interface
Linux
Kernel
Arch-
Ind.
File Systems
VFS
File System Impl.
Networking
Sockets
Protocols
Processes
Scheduling
IPC
Memory
Management
Page allocation
Address spaces
Swapping
Device Drivers
Arch-
Depend.
Hardware Access
Hardware
CPU, Memory, PCI, Devices

04 Porting Linux to L4

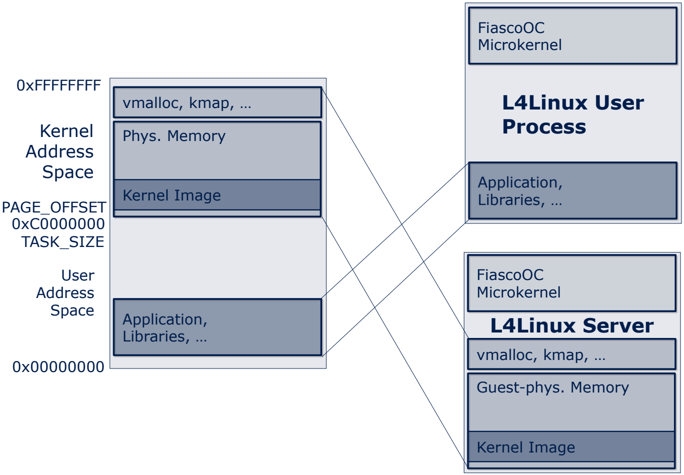
Regard L4 as new hardware platform. Port small architecture dependent part:

- system call interface
  - kernel entry
  - signal delivery
  - copy from/to user space
- hardware access
  - CPU state and features
  - MMU
  - interrupts
  - memory-mapped and port I/O


L4 Task
Application
L4 Task
Application
L4 Task
Application
L4 Task
Application
L4 Task
Arch-
Depend.
System-Call Interface
Linux
Kernel
Arch-
Ind.
File Systems
VFS
File System Impl.
Networking
Sockets
Protocols
Processes
Scheduling
IPC
Memory
Management
Page allocation
Address spaces
Swapping
Device Drivers
Arch-
Depend.
Hardware Access
sigma0
L4IO
Console
moe
user
kernel
Fiasco.OC
Hardware

04 L4Linux Architecture

- L4 specific code is divided into:
  - x86 and ARM specific code
  - hardware generic code
- Linux kernel and Linux user processes run each with a single L4 task.
  - L4Linux kernel task does not see a L4Linux process virtual memory


0xFFFFFFFF
Kernel Address Space
PAGE_OFFSET
0xC0000000
TASK_SIZE
User Address Space
0x00000000
vmalloc, kmap, …
Phys. Memory
Kernel Image
Application, Libraries, …
FiascoOC Microkernel
L4Linux User Process
Application, Libraries, …
FiascoOC Microkernel
L4Linux Server
vmalloc, kmap, …
Guest-phys. Memory
Kernel Image

04 L4Linux Challenges

The L4Linux kernel "server" has to:

- access user process data,
- manage page tables of its processes,
- handle exceptions from processes, and
- schedule them.

L4Linux user processes have to:

- "enter" the L4Linux kernel (living in a different address space).

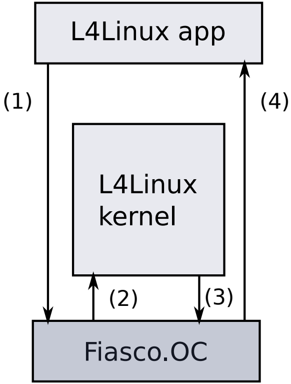
04 Kernel Entry

Normal Linux syscall interface (int 80h) causes trap.

- L4Linux server receives exception IPC.

Heavyweight compared to native Linux system calls:

- two address space switches,
- two Fiasco kernel entries/exits



04 Threads & Interrupts

The old L4Linux has a thread for each user thread and virtual interrupt.

- Interrupts are received as messages.
- Interrupt threads have higher priority than normal Linux threads (Linux semantics).
- Interrupt threads force running user process (or idle thread) into L4Linux server.
- Linux uses CLI/STI to disable interrupts, L4Linux uses a lock.

A synchronization nightmare.

04 L4Linux on vCPUs

Simplify interrupt/exception handling by introducing vCPUs (Fiasco.OC), which are special threads that:

- have dedicated interrupt entry points,
  - need to differentiate between interrupt and systemcall
- can be rebound to different tasks,
  - simulates address space switches
- can mask interrupts
  - emulates Interrupt Enable flag
  - don't need that lock anymore

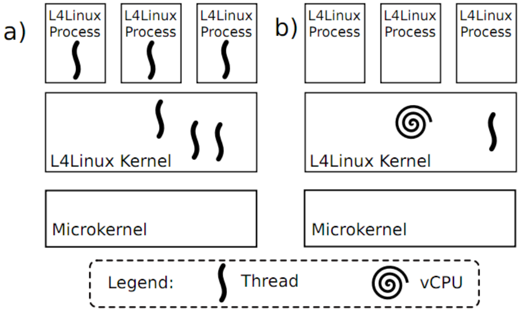


FIGURE 3: *(a) L4Linux implemented with threads and (b) L4Linux implemented with vCPUs.*

04 L4Linux as Toolbox

Reuse large parts of code from Linux:

- filesystems,
- network stack,
- device drivers,
- . . .

Use hybrid applications to provide this service to native L4 applications.

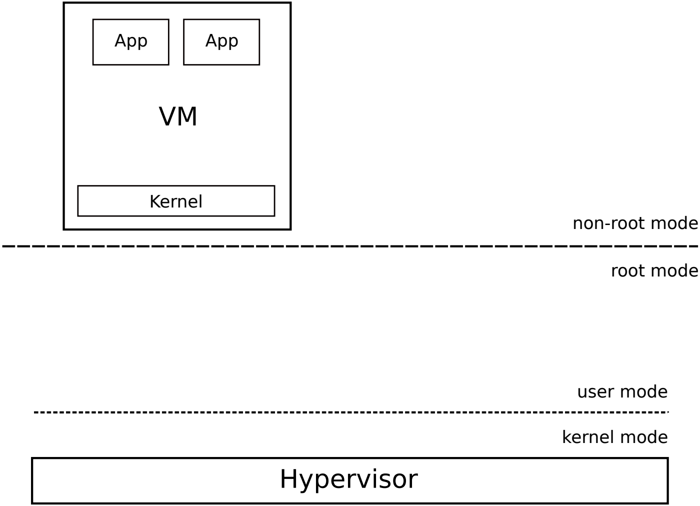
Will be topic of upcoming lecture.

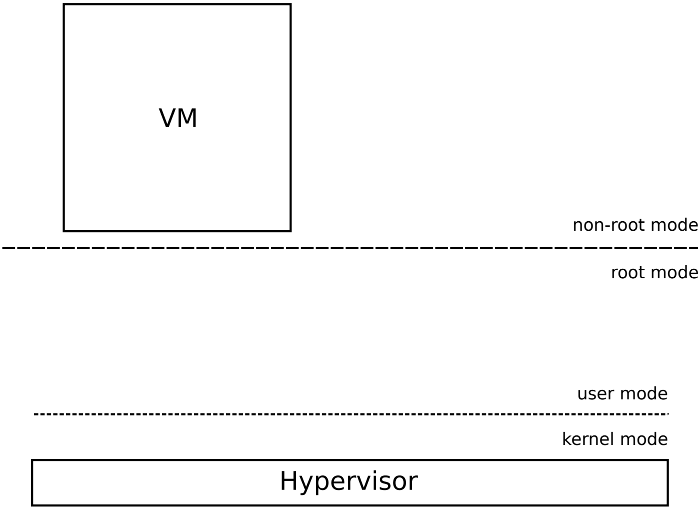
05 Outline

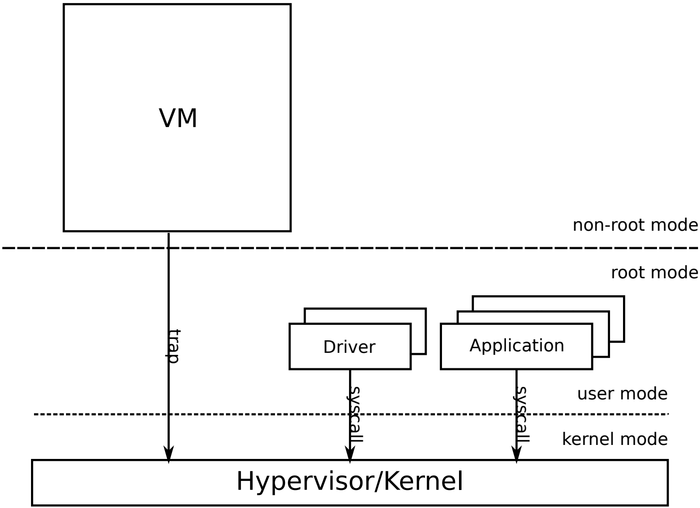
TECHNISCHE UNIVERSITÄT DRESDEN

05 Starting Point

The NOVA OS Virtualization Architecture is a operating system developed from scratch to support virtualization.


App
App
VM
Kernel
non-root mode
root mode
user mode
kernel mode
Hypervisor


VM
non-root mode
root mode
user mode
kernel mode
Hypervisor


VM
non-root mode
root mode
trap
Driver
Application
syscall
syscall
user mode
kernel mode
Hypervisor/Kernel

05 Secunia Advisory SA25073

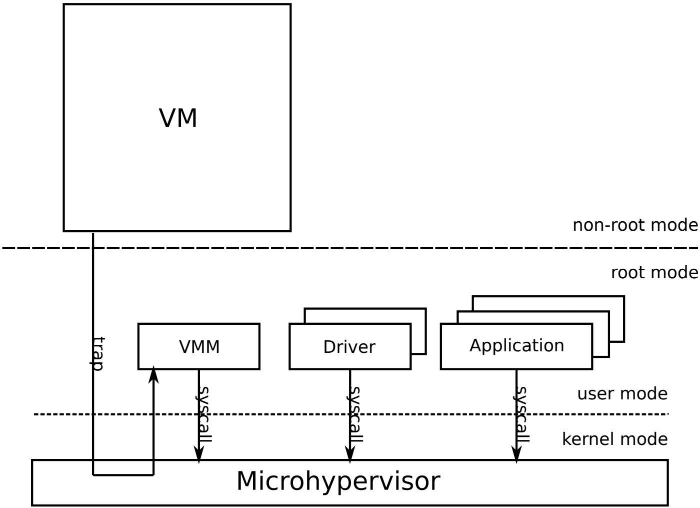
http://secunia.com/advisories/25073/

- "The size of ethernet frames is not correctly checked against the MTU before being copied into the registers of the NE2000 network driver. This can be exploited to cause a heap-based buffer overflow."
- " An error within the handling of the aam instruction can result in a division by zero."
- ...

05 TCB of Virtual Machines

The *Trusted Computing Base* of a Virtual Machine is the amount of hardware and software you have to trust to guarantee this VM's security. (More in lecture on Security)
For e.g. KVM this (conservatively) includes:

- the Linux kernel,
- Qemu.


VM
non-root mode
root mode
trap
VMM
Driver
Application
syscall
syscall
syscall
user mode
kernel mode
Microhypervisor

05 What needs to be in the Microhypervisor?

Ideally nothing, but

- VT-x instructions are privileged:
  - starting/stopping a VM
  - access to VMCS
- hypervisor has to validate guest state to enforce isolation.

05 Microhypervisor vs. VMM

We make a distinction between both terms [St10, Ag10, Le04].

Microhypervisor

- "the kernel part"
- provides isolation
- mechanisms, no policies
- enables safe access to virtualization features to userspace

VMM

- "the userland part"
- CPU emulation
- device emulation

05 NOVA Architecture

Reduce complexity of hypervisor:

- hypervisor provides low-level protection domains
  - address spaces
  - virtual machines
- VM exits are relayed to VMM as IPC with guest state,
- one VMM per guest in (root mode) userspace,
  - possibly specialized VMMs to reduce attack surface
  - only one generic VMM implement so far

05 VMM: Needed Device Models

For a reasonably useful VMM, you need

- Instruction Emulator
- Timer: PIT, RTC, HPET, PMTimer
- Interrupt Controller: PIC, LAPIC, IOAPIC
- PCI hostbridge
- keyboard, mouse, VGA
- network
- SATA or IDE disk controller

But then you still cannot run a VM . . .

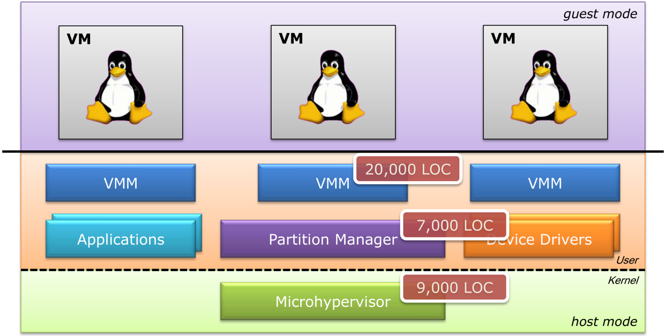
05 VMM: Virtual BIOS

VMM needs to emulate (parts of) BIOS:

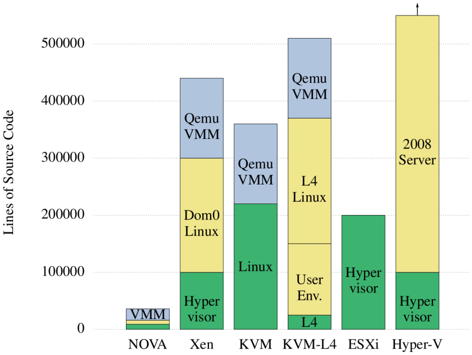
- memory layout
- screen output
- keyboard
- disk access
- ACPI tables

Mostly used for bootloaders and early platform discovery (memory layout).

05 NOVA OS Virtualization Architecture



05 TCB compared



05 Further Topics

- nested virtualization
- device driver reuse
- PCI passthrough
- ...

06 Outline

06 Example: Karma VMM

Idea: Reduce TCB of VMM by using paravirtualization *and* hardware-assisted virtualization.

- Implemented on Fiasco using AMD-V
- Small VMM: 3800 LOC
- 300 LOC changed in Linux
- No instruction emulator required
  - no MMIO
  - no 16-bit code
- Only simple paravirtualized device models required: 2600 LOC
  - salvaged from L4Linux

Started as Diplomarbeit by Steffen Liebergeld, now maintained at `http://karma-vmm.org/`.

06 Recap: Examples

- L4Linux is the paravirtualized workhorse on L4 Fiasco/Fiasco.OC:
  - reuse Linux applications,
  - reuse Linux components.
- NOVA provides faithful virtualization with small TCB for VMs:
  - one VMM per VM,
  - run unmodified commodity operating systems.
- Karma uses hardware virtualization extensions to simplify paravirtualization.

06 Next Weeks

Next week's lecture will be about Legacy Containers and OS Personalities.

Don't forget to read until January 21^{th}:

Formal Requirements for
Virtualizable Third-Generation Architectures
`http://portal.acm.org/citation.cfm?id=361073`

- Adam Lackorzynski et al.
  Virtual Processors as Kernel Interface
  https://www.osadl.org/fileadmin/dam/rtlws/12/Lackorzynski.pdf
- Härtig et al.
  The performance of μ-kernel-based systems
  http://dl.acm.org/citation.cfm?id=266660
- Timothy Roscoe et al.
  Hype and Virtue
  https://portal.acm.org/citation.cfm?id=1361401
- Robert P. Goldberg, 1974
  Survey of Virtual Machine Research
  http://cseweb.ucsd.edu/classes/wi08/cse221/papers/goldberg74.pdf
- Gerald J. Popek, Robert P. Goldberg, 1974
  Formal requirements for virtualizable third generation architectures
  https://portal.acm.org/citation.cfm?id=361073
- Udo Steinberg, Bernhard Kauer, 2010
  NOVA: A Microhypervisor-Based Secure Virtualization Architecture
  http://os.inf.tu-dresden.de/papers_ps/steinberg_eurosys2010.pdf

- Joshua LeVasseur et al, 2005
  Pre-Virtualization: Slashing the Cost of Virtualization
  `http://www.l4ka.org/downloads/publ_2005_levasseur-ua_cost-of-virtualization.pdf`
- Sorav Bansal and Alex Aiken, 2008
  Binary Translation Using Peephole Superoptimizers
  `http://theory.stanford.edu/~sbansal/pubs/osdi08_html/index.html`
- M. Rosenblum and T. Garfinkel, 2005
  Virtual Machine Monitors: Current Technology and Future Trends
  `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1430630`
- Steffen Liebergeld, 2010
  Lightweight Virtualization on Microkernel-based Systems
  `http://os.inf.tu-dresden.de/papers_ps/liebergeld-diplom.pdf`
- Muli Ben-Yehuda et al, 2010
  The turtles project: Design and implementation of nested virtualization
  `http://www.usenix.org/events/osdi10/tech/full_papers/Ben-Yehuda.pdf`
- Ole Agesen et al, 2010

The evolution of an x86 virtual machine monitor
http://portal.acm.org/citation.cfm?id=1899930

LeVasseur et al, 2004
A Sledgehammer Approach to Reuse of Legacy Device Drivers
http://dl.acm.org/citation.cfm?id=1133617