Hardware virtualization

On Intel x86_64

Jan Dubský & Vít Kabele, MFF UK

HyPike

- The hypervisor for PikeOS
- Software project at MFF UK
- Included the development of own kernel
- Successfully finished
  - Able to run own kernel and PikeOS guests

Presentation roadmap

- Top to bottom
- We will go from the high-level abstractions
- via the required OS interfaces
- down to the hardware

~~More~~ Other details in the NSWI150 course (Virtualizace a cloud computing)

Full virtualization

- Unmodified kernel in userspace
  - binary translation
- Hardware assisted virtualization

Paravirtualization

The guest is aware of being virtualized.

- Can take the advantage of this

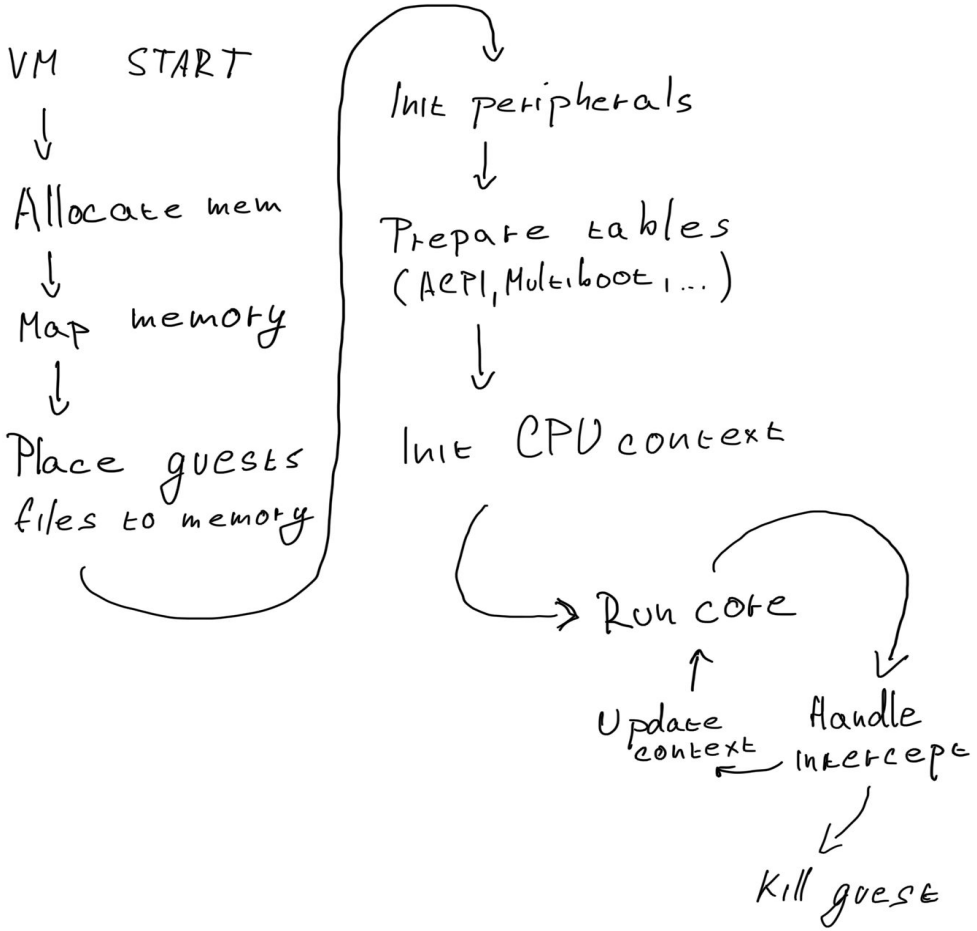
Examples:

- Modified kernels running somehow
- Virtio
- HyperV

Requirements on the VM

- Should behave (almost) as standard process
- Can be preempted, resource limits applies properly
- Multiple VMs not a problem
- What happens in VM should stay in VM (x expensive context switches)
- Memory swapping
- Live migration
- Can be suspended and run again

VM startup/lifecycle

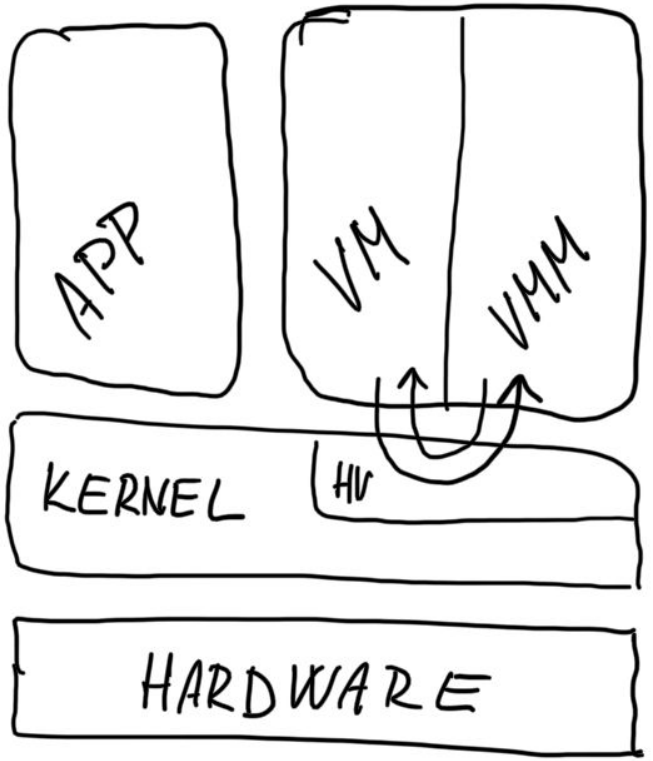


Guest machine boot

- Needs its own bootloader, as usual bootloaders expect BIOS
  - And BIOSes do nasty things with computers
  - Cache as RAM, RAM initialization etc.
  - All of this has to be properly emulated/or patched BIOS (SeaBIOS)
- VM can be also started in an arbitrary state
  - Long mode boot - Usually is not supported in vanilla distributions (without patches)
- Let’s do a compromise
  - Do the work of bootloader and pass the control to the kernel
  - For example we use the GNU multiboot2 specification.
    - Which is nice, but neither Linux nor BSDs supports it
    - Both Linux and BSD have its own boot protocols

Hypervisor vs. VMM (Virtual Machine Manager)

- Hypervisor is the bare metal virtualization controller
  - The kernel or its module
- VMM is the process controlling hypervisor
  - Userspace program (typically)
  - Commanding hypervisor via syscalls
  - Emulating features not implemented in the hypervisor
- VMM is responsible for what to do, hypervisor for how to do it
- Terminology varies



Hypervisor API

- Syscalls (KVM uses ioctls on /dev/kvm)
- VM state is associated with file descriptor
- Examples of calls:
  - Mapping physical memory
  - Run core
  - Injecting interrupts
  - Handling I/O port
  - Emulate unknown instructions

Intel VMX

- The CPU has to be switched to the VMX mode - needs VMXON area
- Each guest is represented by a VMCS - a single page in memory which stores:
  - Guest state
  - Host state
  - Virtualization control registry
- VMCS is read/written using VM* instructions
- VMCS can be active, current and inactive
  - Active - CPU can cache parts of the state - the in-memory representation doesn’t have to be up-to-date.
  - Current - All VM* instruction are related to this VMCS Implies Active.
  - Inactive - The in-memory state reflects the real state.

VM Entry

- VM is launched using VMLAUNCH instruction
- VM Entry verifies that both host and guest state are consistent, same as ensures that MV control registry values are correct
- In case of any errors causes a special VM Exit, which indicates VM entry fail
- Absence of any detailed reporting what went wrong

VM intercepts

- a.k.a. VMExit on Intel
- Selected instructions cause VMExits
- Different generations of CPUs supports different sets
- Root x non-root domain transitions
- Host OS is responsible for emulation of the instruction which caused the VMExit
- Formal requirements: Popek & Goldberg theorem

VM Exit

- Very similar to interrupt handler in a normal kernel
- Just one VM Exit handler for all VM Exits
- VM Exist are identified using VM Exit numbers
- Separate mechanism from interrupt delivery
- The handler has to store general purpose registry and then decide based in the VM Exit number which handler to call.
- Once the VM Exit is handled, the hypervisor returns to VM using VMRESUME instruction.

EPT

- Another level of page tables for virtualization - below the guests page tables
- Have very similar structure to page tables
  - Dirty bits, accessed bits
- Map so called guest-physical addressed to host-physical
- Guest-virtual address translates using guest page tables to guest-physical address (If guest uses paging). That address in then translated using EPT.
- EPT violation causes a VM exit, which allows the host kernel to emulate memory mapped accesses.
  - But the hypervisor has to decode the causing instruction
  - It’s fun to write an instruction parser ;)

I/O ports

- One of the first thing that the kernels try
- Legacy (likely?), slow (for sure) way of communication with peripherals
- String instructions
- Segment hell
- Otherwise nice semantics

MSR virtualization

- Hypervisor configures a MSR bitmap
  - guest accesses of masked MSRs cause a VM Exit.
- On top of that, VMX allows the hypervisor to configure which registers should be stored and loaded on VM Exit.
  - Possible, but slow!
  - VMX won’t store host MSRs on VM Entry - the hypervisor has to do so manually - that sucks
- Some MSRs are are stored directly in VMCS
  - Those which are frequently used
  - EFER, PAT etc.

Control registry virtualization & CPUID

- VMX enforces some control bits to be set - this also applies to guest
- The host has a possibility to hide the real CR state from the guest
- Some CR bits are not modified by VM Exit
- There is a bug which allows changing a hardcoded bit value

- CPUID causes a VM Exit
- Hypervisor emulates its behaviour
  - Even for example change of a MSR value
- Used to be used as serializing instruction for RDTSC, which is obviously not a good idea in virtualized environment

Preemption timer

- The easy way how to exit from VM after some time without involving external interrupt source.
- The only timer dedicated to virtualization only (and though not shared with the host OS)
- Ticks only in non-root domain
- Counts down to zero and then causes VM Exit
- The frequency is derived from the host TSC frequency
- What happens without invariant TSC?

Interrupt delivery

- VMX has very minimalistic interrupt delivery support
- The host can inject a single interrupt in every VM Entry
- Hypervisor is responsible for ensuring that host state allows interrupt delivery
- Hypervisor has to implement interrupt prioritization mechanism
  - The table in the manual is definitely not trivial
- VMX provides interrupt delivery window and NMI delivery window functionalities
  - Host states that it wants to gain control once an interrupt can be injected to the guest

External interrupt delivery

- Host has to emulate all external interrupt controllers
  - No hardware support except for general interrupt support
- Host has to emulate at least PIC and Local APIC
  - Local APIC has virtualization support in newest versions, but still requires implementation of an instruction parser.
- Interrupt delivery logic and prioritization must be implemented properly
  - Intel manual is sometimes very underspecified
- CPU support doesn’t care about external devices.
  - Must be emulated in the software
  - PIC, PIT, UART, RTC, Keyboard...

Questions

Thank you for attention

Jan Dubský & Vít Kabele, MFF UK