Virtualisation

Johan Montelius

KTH

2017
The role of the operating system - provide a virtual environment for a process.
the kernel

user space

- code (.text)
- data
- heap
- stack

kernel space

- segm. table
- page table
- inter. table

MMU
IDTR

3 / 24
indirect execution

Who is in control?
- control the registers of the MMU and you control the virtual address space
- control the IDTR and you control what will happen when we have an interrupt
- instructions to set MMU or IDT registers are privileged instructions

Limited direct execution:
- only work with mapped memory in user space,
- only execute non-privileged instructions,
- for a limited amount of time.
Interrupts

Synchronous interrupts - exceptions:

- faults:
  - page fault
  - privilege violation
  - divide by zero, ...

- programmed exceptions:
  - system call (INT 0x80)
  - debug instructions

Asynchronous interrupts:

- timer interrupt
- hardware interrupt: I/O complete,
  ...

The kernel is interrupt driven.
Virtualisation

process

operating system

hypervisor - virtual machine manager (VMM)

hardware
Why?

Utilisation of hardware.

Also provided by a multi-task operating system, what is new?

Applications are completely separated from each other.

What do two processes in an operating system share?

Applications can use different operating systems.

Is this important?
the Hypervisor

Provide virtualisation of the hardware:

- a virtual cpu, part of the processing power
- a virtual memory, the illusion of physical memory

*I think we have seen this before.*
Provide *limited direct execution* i.e. allow each guest operating system to execute in *user space* and only perform non-privileged operations.

What is the first thing an operating system wants to do?
The virtual IDT

The operating system is running in non-privileged mode.
a system call

Hypervisor  Guest operating system  Application

running system call
INT 0x80

handle interrupt
check OS IDT
call OS procedure

handle interrupt

return to user

handle interrupt

return to user

resume execution
What about virtual memory?

- virtual addresses
- guest operating system
- physical addresses
- hypervisor
- machine addresses
- hardware

- regular translation tables
- second level translation

This will be expensive!
User process uses virtual addresses that are automatic translated by the hardware (using page table and the MMU) to physical addresses.

A *page fault* invokes the kernel that, if allowed, maps a missing page and return to the user process.
second level paging

Hypervisor | Guest operating system | Application

handle interrupt

call OS procedure

map missing page

update page table

modify page table

return to OS

return to user

return to user

running page fault

resume execution
If the guest operating system is executing in user mode - how does it protect itself from the application process that is also running in user mode?

If we allow the guest operating system to run in kernel mode - then the hypervisor can not protect it self.
system call revisited

Hypervisor | Guest operating system | Application
---|---|---
| | kernel space | user space
| | in user mode | in user mode
| | | system call
| | | INT 0x80

- change tables
- OS now in user space
- in user mode
- handle interrupt
- return to user
- resume execution
- change tables
- OS in kernel space
Hardware support needed - available in both AMD and Intel x86 processors.

With hardware support, hypervisors can provide near “bare metal” performance.
Utilisation of hardware.

Applications are completely separated from each other.

Applications can use different operating systems.

What if we skip this.
An operating system uses several name spaces: memory addresses, file paths, port numbers, device interrupt requests, process id, user id, …

Provide a *container*, a separate environment with its own name spaces.

Processes in different containers are completely separated from each other … but they use the same kernel.
containers

operating system

hardware
Utilisation of hardware.

Applications are completely separated from each other.

Applications can use different operating systems.

Why do they have to run on the same hardware?
Hardware emulators can be surprisingly efficient.
Types of virtual machines

- **Emulators**
  - Can emulate a different hardware than the host machine (QEMU, Simics).

- **Virtual machines**
  - Choose operating system but hardware is set (Xen, KVM, VirtualBox, VMware).

- **Containers**
  - Separated name spaces in the same operating system (Dockers, Linux Containers).

- **Runtime systems**
  - Dedicated to a language (JVM, Erlang).
Multiple operating systems running on the same machine.
Each operating system provided a virtual hardware.
With hardware support, near bare metal execution speed can be obtained.
Other types: emulators, containers, runtime environments.