Virtualisation

Johan Montelius

KTH

2019
The role of the operating system - provide a virtual environment for a process.
the kernel

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

code (.text)
data
heap
stack

user space

kernel space
the kernel

user space

code (.text)  data  heap  →  stack

kernel space

MMU  IDTR
the kernel

code (.text) | data | heap | stack | segm. table | page table

user space

kernel space

MMU

IDTR
the kernel

user space

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

kernel space

- segm. table
- page table
- inter. table

MMU

IDTR
Who is in control?
indirect execution

Who is in control?
- control the registers of the MMU and you control the virtual address space
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
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
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:

...
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,
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,
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.
Synchronous interrupts - exceptions:

- faults:
Synchronous interrupts - exceptions:

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

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

- programmed exceptions:
Synchronous interrupts - exceptions:

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

- programmed exceptions:
  - system call (INT 0x80)
  - debug instructions
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.
Synchronous interrupts - exceptions:
- faults:
  - page fault
  - privilege violation
  - divide by zero, ...
- programmed exceptions:
  - system call (INT 0x80)
  - debug instructions

Asynchronous interrupts:
- timer interrupt
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, ...
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

hardware
Virtualisation

- operating system
- hardware
Virtualisation

- process
- process
- process

- operating system

- hardware
Virtualisation

process

process

operating system

hardware
Virtualisation

- process
- hardware
- operating system
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?
Why?

Utilisation of hardware.
Why?

Utilisation of hardware.

Also provided by a multi-task operating system, what is new?
Utilisation of hardware.

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

Applications are completely separated from each other.
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?
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.
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?
Provide virtualisation of the hardware:

- a virtual cpu, part of the processing power
the Hypervisor

Provide virtualisation of the hardware:

- a virtual cpu, part of the processing power
- a virtual memory, the illusion of physical memory
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.
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?
set up IDT
The virtual IDT

Hypervisor

set up IDT
pass control to OS

Guest Operating system
The virtual IDT

- Hypervisor
  - set up IDT
  - pass control to OS

- Guest Operating system
  - initialize OS
The virtual IDT

Hypervisor

set up IDT

pass control to OS

Guest Operating system

initialize OS

set up IDT
The operating system is running in non-privileged mode.

Hypervisor

- set up IDT
- pass control to OS
- handle interrupt

Guest Operating system

- initialize OS
- set up IDT
The virtual IDT

- Hypervisor
  - set up IDT
  - pass control to OS
  - handle interrupt
  - save ref to IDT of OS

- Guest Operating system
  - initialize OS
  - set up IDT
the virtual IDT

Hypervisor → Guest Operating system

set up IDT
pass control to OS
initialize OS
set up IDT

handle interrupt
save ref to IDT of OS
pass control to OS
The virtual IDT

Hypervisor

set up IDT
pass control to OS
handle interrupt
save ref to IDT of OS
pass control to OS

Guest Operating system

initialize OS
set up IDT
The virtual IDT

Hypervisor

- set up IDT
- pass control to OS
- handle interrupt
- save ref to IDT of OS
- pass control to OS

Guest Operating system

- initialize OS
- set up IDT
- continue as if nothing happened

The operating system is running in non-privileged mode.
<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Guest operating system</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>running</td>
</tr>
<tr>
<td>Hypervisor</td>
<td>Guest operating system</td>
<td>Application</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>running</td>
<td>system call</td>
<td></td>
</tr>
<tr>
<td>Hypervisor</td>
<td>Guest operating system</td>
<td>Application</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>running system call</td>
<td>INT 0x80</td>
<td></td>
</tr>
</tbody>
</table>
Hypervisor  Guest operating system  Application

running system call

INT 0x80
A system call is a mechanism for communication between an application and the underlying operating system. In the context of virtualization, when an application running on a guest operating system needs to perform a system call, it typically does so by executing an interrupt with the system call number, in this case `INT 0x80`. The guest operating system, in turn, handles this interrupt and passes the system call to the hypervisor, which then facilitates the interaction between the application and the underlying hardware or another virtualized environment.
a system call

Hypervisor  Guest operating system  Application

running system call

INT 0x80

handle interrupt
check OS IDT
A system call involves the following steps:

1. **Hypervisor**
   - handle interrupt
   - check OS IDT
   - call OS procedure

2. **Guest operating system**
   - INT 0x80

3. **Application**
   - running system call

The diagram illustrates the flow of a system call from the application layer to the hypervisor, where the interrupt is handled, and finally the OS procedure is called.
A system call involves the following steps:

1. **Application** running a system call
2. **INT 0x80** is issued to handle the interrupt
3. **Hypervisor** checks the OS IDT
4. **Guest operating system** is called

The process flow is as follows:

- **Application** → **INT 0x80** → **Hypervisor** → **check OS IDT** → **call OS procedure** → **Guest operating system** → **handle interrupt** → **Application**
A system call involves interaction between the Hypervisor, Guest operating system, and Application. When an application running on the Guest OS performs a system call, it typically uses an interrupt, such as INT 0x80. This interrupt signal is handled by the OS, which checks the IDT (Interrupt Descriptor Table) to find the correct procedure to handle the interrupt. Once the appropriate OS procedure is called, it handles the interrupt, completing the system call process.
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
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
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
a system call

Hypervisor | Guest operating system | Application

running system call

INT 0x80

handle interrupt
check OS IDT
call OS procedure

handle interrupt

handle interrupt
return to user

return to user

return to user
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

11 / 26
What about virtual memory?

- process
- guest operating system
- hypervisor
- hardware

This will be expensive!
What about virtual memory?

- virtual addresses
- guest operating system
- hypervisor
- hardware
What about virtual memory?

- virtual addresses
- regular translation tables
- guest operating system
- physical addresses
- hypervisor
- hardware
What about virtual memory?

- virtual addresses
- regular translation tables
- second level translation
- guest operating system
- physical addresses
- hypervisor
- machine addresses
- hardware
What about virtual memory?

- virtual addresses
- 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.
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 |

running
<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Guest operating system</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>running</td>
<td>page fault</td>
</tr>
</tbody>
</table>
second level paging

Hypervisor          Guest operating system          Application

running

page fault
second level paging

Hypervisor | Guest operating system | Application

→ running page fault

handle interrupt
second level paging

Hypervisor  Guest operating system  Application

running
page fault

handle interrupt
call OS procedure
second level paging

Hypervisor  Guest operating system  Application

running
page fault

handle interrupt
call OS procedure
second level paging

Hypervisor  Guest operating system  Application

running page fault

handle interrupt
call OS procedure

map missing page
second level paging

Hypervisor  Guest operating system  Application

handle interrupt

running page fault

call OS procedure

map missing page
update page table
second level paging

Hypervisor  Guest operating system  Application

running
page fault

handle interrupt

call OS procedure

map missing page

update page table
second level paging

Hypervisor  Guest operating system  Application

running
page fault

handle interrupt

call OS procedure

map missing page

update page table

modify page table
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

running page fault
second level paging

Hypervisor | Guest operating system | Application

Running page fault

handle interrupt → call OS procedure → map missing page → update page table

modify page table ← return to OS
second level paging

Hypervisor                      Guest operating system                      Application

running

page fault

handle interrupt

call OS procedure

map missing page

update page table

modify page table

return to OS

return to user
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 OS  return to user

running page fault
### Second Level Paging

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Guest operating system</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle interrupt</td>
<td>call OS procedure</td>
<td>running page fault</td>
</tr>
<tr>
<td>map missing page</td>
<td>modify page table</td>
<td></td>
</tr>
<tr>
<td>update page table</td>
<td>return to OS</td>
<td></td>
</tr>
<tr>
<td>return to user</td>
<td></td>
<td>return to user</td>
</tr>
</tbody>
</table>
second level paging

Hypervisor | Guest operating system | Application

running page fault

handle interrupt

call OS procedure

map missing page

update page table

modify page table

return to OS

return to user

return to user

resume execution
...wait a second

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 cannot protect itself.
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 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 cannot protect itself.
system call revisited

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Guest operating system</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kernel space</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in user mode</td>
</tr>
<tr>
<td>Hypervisor</td>
<td>Guest operating system</td>
<td>Application</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td>kernel space</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in user mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>system call</td>
</tr>
</tbody>
</table>
system call revisited

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Guest operating system</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kernel space</td>
<td>user space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in user mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>system call</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INT 0x80</td>
</tr>
</tbody>
</table>
system call revisited

Hypervisor   Guest operating system   Application

kernel space  user space

in user mode
system call
INT 0x80
system call revisited

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

change tables
system call revisited

Hypervisor

kernel space

Guest operating system

user space

Application

in user mode

system call

INT 0x80

change tables

OS now in user space
Hypervisor

kernel space

Guest operating system

user space

Application

in user mode

system call

INT 0x80

change tables

OS now in user space
system call revisited

Hypervisor | Guest operating system | Application
kernel space | user space | in user mode

change tables
OS now in user space

INT 0x80

in user mode
system call revisited

Hypervisor

kernel space

calculate tables

Guest operating system

user space

in user mode

Application

system call

INT 0x80

in user mode

handle interrupt

OS now in user space
System call revisited

Hypervisor
kernel space
change tables
OS now in user space

Guest operating system
user space

Application
in user mode
system call
INT 0x80

in user mode
handle interrupt
return to user
Hypervisor

kernel space

change tables

OS now in user space

Guest operating system

user space

in user mode

system call

INT 0x80

Application

in user mode

handle interrupt

return to user
system call revisited

Hypervisor
kernel space
change tables
OS now in user space
change tables

Guest operating system
user space

Application
in user mode
system call
INT 0x80

in user mode
handle interrupt
return to user
system call revisited

Hypervisor  Guest operating system  Application

kernel space  user space

in user mode  system call  INT 0x80

change tables
OS now in user space

in user mode
handle interrupt

return to user

change tables
OS in kernel space
system call revisited

Hypervisor | Guest operating system | Application
---|---|---
kernel space | user space | in user mode

system call

- INT 0x80
- change tables
- OS now in user space

in user mode

- handle interrupt
- return to user

- change tables
- OS in kernel space
System call revisited

- Hypervisor
- Guest operating system (kernel space)
- Application (user space)

**Change tables**
- OS now in user space

**Handle interrupt**
- In user mode
- Return to user

**Resume execution**
.. thank god for harware
Hardware support:
Hardware support:

- Available in both AMD and Intel x86 processors
Hardware support:

- Available in both AMD an Intel x86 processors

- Allows hypervisors to provide near “bare metal” performance.
.. a different approach
Para-virtualization: change the operating system that you want to virtualize.
Para-virtualization: change the operating system that you want to virtualize.

- Change kernel modules in the operating system.
Para-virtualization: change the operating system that you want to virtualize.

- Change kernel modules in the operating system.
- Recompile source code or patch binary code.
The original goal
The original goal

Utilisation of hardware.
The original goal

Utilisation of hardware.
The original goal

Utilisation of hardware.

Applications are completely separated from each other.
The original goal

Utilisation of hardware.

Applications are completely separated from each other.
The original goal

Utilisation of hardware.

Applications are completely separated from each other.

Applications can use different operating systems.
The original goal

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.
An operating system uses several name spaces: memory addresses, file paths, port numbers, device interrupt requests, process id, user id, ...
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.
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 ...
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

hardware
containers

operating system

hardware
containers

operating system

hardware
the original goal
Utilisation of hardware.
Utilisation of hardware.
Utilisation of hardware.

Applications are completely separated from each other.
Utilisation of hardware.

Applications are completely separated from each other.
Utilisation of hardware.

Applications are completely separated from each other.

Applications can use different operating systems.
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?
emulating hardware

x86 hardware
Hardware emulators can be surprisingly efficient.

- **operating system**
- **x86 hardware**
Hardware emulators can be surprisingly efficient.
Hardware emulators can be surprisingly efficient.
Hardware emulators can be surprisingly efficient.
Hardware emulators can be surprisingly efficient.
Hardware emulators can be surprisingly efficient.
Hardware emulators can be surprisingly efficient.
Hardware emulators can be surprisingly efficient.
Hardware emulators can be surprisingly efficient.
Types of virtual machines

- Emulators
  - Can emulate a different hardware than the host machine (QEMU, Simics).
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).
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).
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).
... but I never installed a Hypervisor?

VirtualBox etc also installs a kernel module that turns your regular operating system into a hypervisor.
... but I never installed a Hypervisor?

VirtualBox etc also installs a kernel module that turns your regular operating system into a hypervisor.
Multiple operating systems running on the same machine.
Multiple operating systems running on the same machine.

Each operating system provided a virtual hardware.
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.
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.
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.