Virtualisation

Johan Montelius

KTH

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

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

user space

code (.text)
data
heap

kernel space

stack
The kernel

User space:
- code (.text)
- data
- heap
- stack

Kernel space:
- MMU
- IDTR
the kernel

user space

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

kernel space

- segm. table
- page table

MMU
IDTR
the kernel

- Code (.text)
- Data
- Heap
- Stack

User space

Kernel space

- Segment table
- Page table
- Interrupt 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
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.
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,
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,
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:

Asynchronous interrupts:

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

The kernel is interrupt driven.
Interrupts

Synchronous interrupts - exceptions:

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

Asynchronous interrupts:

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

The kernel is interrupt driven.
Interrupts

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

Asynchronous interrupts:
Interrupts

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, ...
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

hardware
Virtualisation

- operating system
- hardware
Virtualisation

- Hardware
- Operating system
- Process
Virtualisation

- process
- operating system
- hardware
Virtualisation

- process
- process
- process
- process
- process
- process
- operating system
- operating system
- hypervisor - virtual machine manager (VMM)
- hardware
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?
Utilisation of hardware.
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.
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?
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.
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
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?
the virtual IDT

Hypervisor

Guest Operating system

set up IDT
The virtual IDT

Hypervisor

Guest Operating system

set up IDT
pass control to OS
The operating system is running in non-privileged mode.
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.
The operating system is running in non-privileged mode.

- set up IDT
- pass control to OS
- initialize OS
- set up IDT
- handle interrupt
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 operating system is running in non-privileged mode.
The virtual IDT

The operating system is running in non-privileged mode.

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Guest Operating system</th>
</tr>
</thead>
<tbody>
<tr>
<td>set up IDT</td>
<td>initialize OS</td>
</tr>
<tr>
<td>pass control to OS</td>
<td>set up IDT</td>
</tr>
<tr>
<td>handle interrupt</td>
<td></td>
</tr>
<tr>
<td>save ref to IDT of OS</td>
<td></td>
</tr>
<tr>
<td>pass control to OS</td>
<td></td>
</tr>
</tbody>
</table>
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

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

Hypervisor          Guest operating system          Application

running
Hypervisor | Guest operating system | Application
---|---|---
running system call
<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Guest operating system</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>running system call</td>
<td>INT 0x80</td>
<td></td>
</tr>
</tbody>
</table>
a system call

Hypervisor     Guest operating system     Application

running system call

INT 0x80
A system call involves a sequence of actions between the Hypervisor, Guest operating system, and Application. The process begins with the Application running system call INT 0x80. Upon receiving this interrupt, the Guest operating system handles the interrupt. The Hypervisor monitors these interactions, ensuring proper execution and security.
A system call

Hypervisor -> Guest operating system -> Application

running system call
INT 0x80

handle interrupt
check OS IDT
a system call

Hypervisor  Guest operating system  Application

handle interrupt
check OS IDT
call OS procedure

running system call
INT 0x80
a system call

Hypervisor     Guest operating system     Application

running system call
INT 0x80

handle interrupt
check OS IDT
call OS procedure
a system call

Hypervisor  Guest operating system  Application

running system call

INT 0x80

handle interrupt
check OS IDT
call OS procedure

handle interrupt
a system call

Hypervisor  Guest operating system  Application

handle interrupt  running system call
check OS IDT  INT 0x80
call OS procedure  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
return to user

11 / 24
A system call involves the following steps:

1. The application running a system call
2. INT 0x80
3. Handling the interrupt
4. Checking the OS IDT
5. Calling the OS procedure
6. Handling the interrupt
7. Returning to the user
A system call involves several steps:

1. The Application running INT 0x80 to initiate a system call.
2. The Hypervisor handles the interrupt.
3. It checks the Guest operating system's IDT to determine which procedure to call.
4. The Hypervisor calls the appropriate OS procedure.
5. The OS procedure handles the interrupt.
6. The Hypervisor returns to the Application, allowing it to continue running.
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?

- process
- guest operating system
- hypervisor
- hardware

This will be expensive!
What about virtual memory?

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

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

- regular translation tables

This will be expensive!
What about virtual memory?

- virtual addresses
- physical addresses
- machine addresses
- guest operating system
- hypervisor
- regular translation tables
- second level translation
- hardware
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.
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

<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></td>
<td></td>
<td>running</td>
</tr>
<tr>
<td></td>
<td></td>
<td>page fault</td>
</tr>
</tbody>
</table>
Hypervisor | Guest operating system | Application

running page fault

handle interrupt
second level paging

Hypervisor | Guest operating system | Application

handle interrupt

call OS procedure

running

page fault
second level paging

Hypervisor | Guest operating system | Application

handle interrupt

call OS procedure

running page fault
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  call OS procedure  running page fault

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

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Guest Operating System</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Handle interrupt
- Call OS procedure
- Map missing page
- Update page table
- Modify page table
- Return to OS

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></td>
<td>running page fault</td>
</tr>
<tr>
<td>call OS procedure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>map missing page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>update page table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>modify page table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return to OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return to user</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
second level paging

Hypervisor  Guest operating system  Application

- handle interrupt
- call OS procedure  map missing page
- modify page table  update page table
- return to OS  return to user

running page fault
second level paging

Hypervisor          Guest operating system          Application

running
page fault

handle interrupt

map missing page

update page table

modify page table

return to OS

return to user

return to user

call OS procedure

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 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 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

Hypervisor  Guest operating system  Application

kernel space  user space  in user mode

in user mode
system call revisited

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

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

→ in user mode
system call

INT 0x80
system call revisited

Hypervisor       Guest operating system       Application

kernel space     user space

in user mode
system call
INT 0x80

change tables
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

16 / 24
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
Hypervisor

kernel space

Guest operating system

user space

change tables

OS now in user space

Application

in user mode

system call

INT 0x80

in user mode

16 / 24
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

in user mode
handle interrupt
system call revisited

Hypervisor

kernel space
close 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
system call revisited

Hypervisor

kernel space

change tables

OS now in user space

Guest operating system

user space

in user mode

handle interrupt

return to user

Application

in user mode

system call

INT 0x80
system call revisited

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

INT 0x80

change tables
OS now in user space

in user mode
handle interrupt
return to user

change tables
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

change tables
OS in kernel space

return to user
system call revisited

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

OS now in user space → in user mode
handle interrupt

← change tables
return to user

OS in kernel space →
system call revisited

Hypervisor | Guest operating system | Application
---|---|---
| kernel space | user space | in user mode
| OS now in user space | handle interrupt | system call
| change tables | return to user | INT 0x80
| OS in kernel space | resume execution |
Hardware support needed - available in both AMD and Intel x86 processors.
Hardware support needed - available in both AMD an Intel x86 processors.

With hardware support, hypervisors can provide near “bare metal” performance.
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.
the original goal

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.

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.
operating system

hardware
containers

process

operating system

hardware
containers

process

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.
the original goal

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?
Hardware emulators can be surprisingly efficient.

x86 hardware
Hardware emulators can be surprisingly efficient.

- **emulating hardware**
  - **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.
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).
Multiple operating systems running on the same machine.
Summary

- 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.
Summary

- 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.