50004 - Operating Systems - Lecture 18

Oliver Killane

28/12/21

Lecture Recording

Lecture recording is available here

Benefit of Virtualization

- **Consolidation** RUnning many servers on one.
  An organisation may want to run many dedicated servers (e.g for mail, webserver). However each will not use an entire machine's resources all the time.

  Hence can save money by running multiple dedicated servers on the same physical server, dynamically allocating resources on demand.

- **Legacy Software** Some software is OS & version specific.
  Despite large efforts by companies to ensure backward compatability, software produced for older OSes may be incompatible/buggy on newer ones.

  Applications may use drivers that are no longer supported for legacy hardware that is no longer commercially available.

  e.g Windows 8, 7, Vista are not truly backwards compatible with XP

- **Software Development & Testing** Test more & more easily
  - Can test multiple instances of an Operating System simultaneously on a machine.
  - Can test system crashing errors (e.g kernel panics) without brining the machine down.

- **Security** Can isolate virtualized system.
  - Only the **VMM** runs in kernel mode, hence if a guest OS is compromised, the whole system & other guests are not.
  - **VMM**s are typically significantly smaller than OS kernels. Less code → less potential for exploitable bugs.
  - Intrusion detection via introspection on guest OSes (monitor resources, e.g scan memory) for suspicious activity.

- **Software Management** Simpler system administration.
  - Can take snapshots of the state (memory, CPU, etc) of a guest OS, and roll back to snapshots.
  - Can migrate between hosts (switch guest OS between machines, take snapshot then transfer and resume). We can use this for load balancing (move to higher spec machine when it is required).
  - Applications depend on OS, compiler, library versions which sometimes conflict between applications. when we want to run a specific application, we can create a VM with all its requirements and then distribute this.

Implementing a Virtual Machine Monitor (VMM)

Technique	Description	Slowdown	
		CPU-Bound	IO-Bound
Emulation	Intercept all instructions. The emulate their execution on hardware.	$\approx 100\times$	$\approx 2\times$
Modern VMs	Use hardware to accelerate virtualization.	$\approx 5\%$	$\approx 30\%$

Instructions

Most instructions can be run directly on the CPU. We may need to perform some checks (e.g to not access other guests' memory).

```
; MASM (Microsoft macro ASseMbler) syntax for convenient lstlisting highlighting
mov eax, [ebp + 16]         ; eax =  *(ebp + 16)
lea eax, [ebx + 8]          ; eax = ebx + 8
call printf                 ; call to printf
add eax, DWORD PTR 12[ebp]  ; eax += 4 bytes below
inc eax                     ; eax++
```

Only sensitive instructions need to be intercepted & emulated. For example with:

```
cli ; Clear Interrupt Flag (maskable interrupts are ignores - e.g timer)
sti ; Set Interrupt Flag (enables interrupts after next instruction)

;When recived, the VMM uses a trap:
;VM #3:
;1. cli
; - [trap, cli in user mode!] VMM stops delivering interrupts to VM #3
; - VMM remembers IF (interrupt flag) state
; - VM#3 continues
;2. interrupts disabled
;3. ...
```

When the instruction is run, it traps, allowing the **VMM** to take over and run relevant routines on the guest OS that caused the trap.

However there are some obstacles:

- **Some instrtuctions do not trap**
  For example on the intel i386 **popf** can be used to replace flags. Including the **IF** (Interrupt Flag) bit.

  However if running in user mode, the **IF** flag is left unchanged, with no trap.

  Hence as the guest OS runs in user space, this instruction could silently fail (not informing VM of attempt to access **IF**)

- **Visibility of privilege level**
  The **CS** (code segment) register informs a process of its privilege level. If we do not attempt to hide this from the guest OS, it will be able to determine if it is in a virtual machine as when running in a vm it will not have kernel privilege on the hardware.

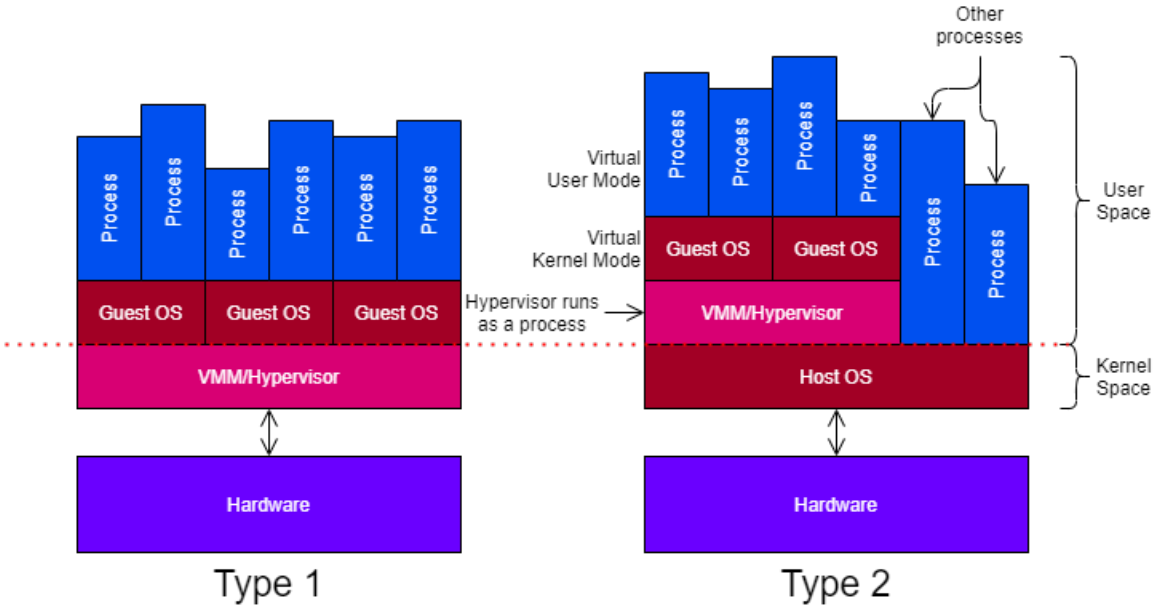
A CPU is considered **virtualizable** if all sensitive instructions trap, and hence when a sensitive instruction is executed the **VMM** can take over.

x86 has been **virtualizable** since 2005:

- Intel Virtualization Technology **VT-x**
- AMD Virtualization **AMD-V**
- And many more extensions such as **VT-d** and **APIC-v** (**APIC** - Advanced Programmable Interrupt Controller)

Hypervisors

Hypervisor Types



- **Type 1** Runs on bare metal - Better performance.
  Has access directly to hardware and runs in kernel mode.

  Requires hardware to support virtualization (trap privileged/sensitive instruction)

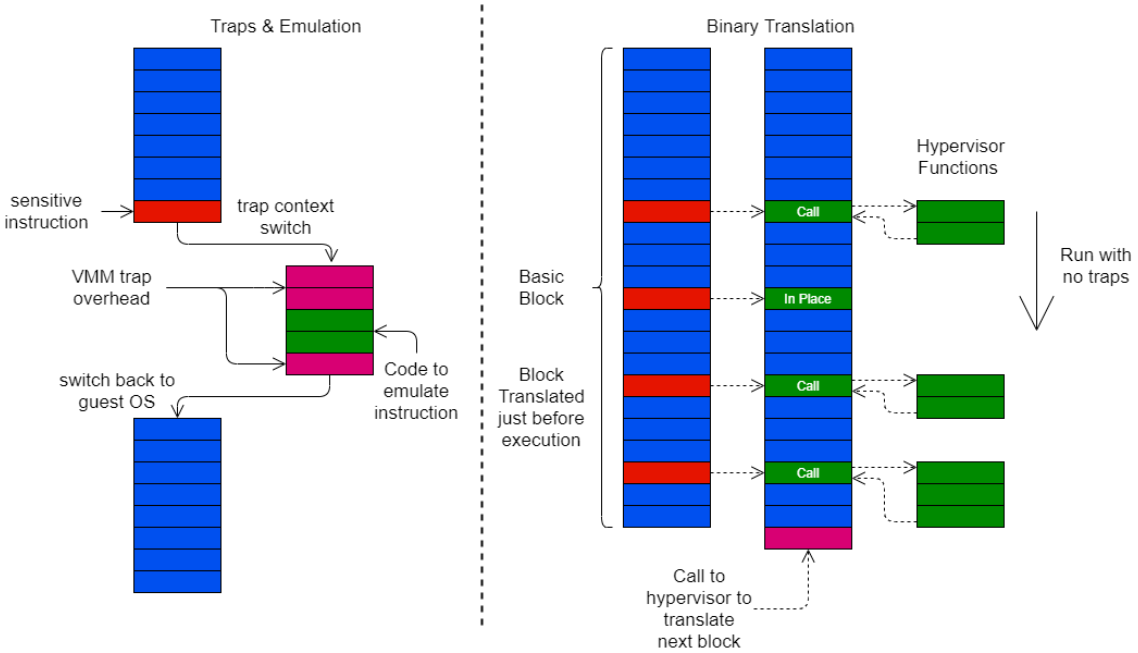
- **Type 2** Runs inside host OS Hypervisor is a program that can be run as a process.
  - Any performance or security issues in the host OS can affect the guest OSes.
  - Easy to install (like a regular application).
  - Can use drivers, scheduling and other services from host OS.

Most systems use a hybrid of both approaches.

Binary Translation

Frequent traps incur significant performance overhead (must clear **TLB**, other caches get obliterated, reduces effectiveness of branch prediction).

In order to eliminate traps, we can instead dynamically translate trap-inducing instructions into the code to resolve them. This is faster than trapping, then emulating, and then switching back to the guest OS.



This technique was originally developed for the **Type 2** hypervisor VMWare Workstation (though has been used outside of virtualization for decades, first paper I could find here.) It can also be used to speed up **Type 1** hypervisors.

1. Scan each **Basic Block** just before execution.
2. If the block contains sensitive/privileged instructions, replace them with calls to hypervisor functions, or instructions in place.
3. Replace last instruction with a call to the hypervisor (to translate next block of instructions and then resume there).

There are several possible optimisations:

- Cache translated blocks (do not need to redo translation when executing code again).
- Once the successor basic block has been translated, replace the hypervisor call with a normal jump.
- Do not translate user code (no privileged instructions so no need to translate, though may need address translation)

With these optimisations we must be careful for cases such as evicting cached blocks (may need to undo some hypervisor call $\rightarrow$ jump optimisations).

This can also slow down non-privileged instructions. For example user-level calls now need to be handled by the **VMM** to ensure only translated blocks are executed, and the **VMM** must be careful with saving pointers (e.g in call to **VMM** function) to ensure the guest OS cannot discover they are being virtualized through stack inspection.

Paravirtualization

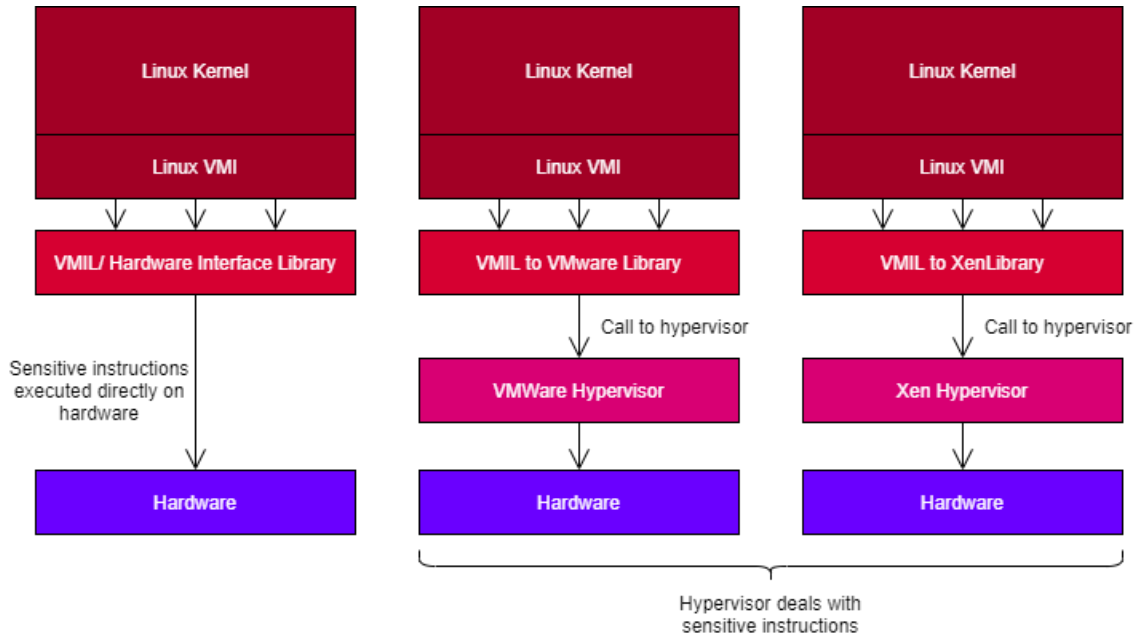
Another technique is to change the guest operating system's kernel source code to replace sensitive instructions with hypervisor calls.

- Can handle unvirtualizable hardware.
- Can achieve better performance.
- Need to make many changes to the guest OS source (not always possible). An OS must support it, as well as native hardware, as well as other hypervisors.

Virtual Machine Interface (VMI)

A single hypervisor **API** that can interface with multiple hypervisors & hardware. This **API** can connect with hypervisor specific libraries, or directly to hardware.

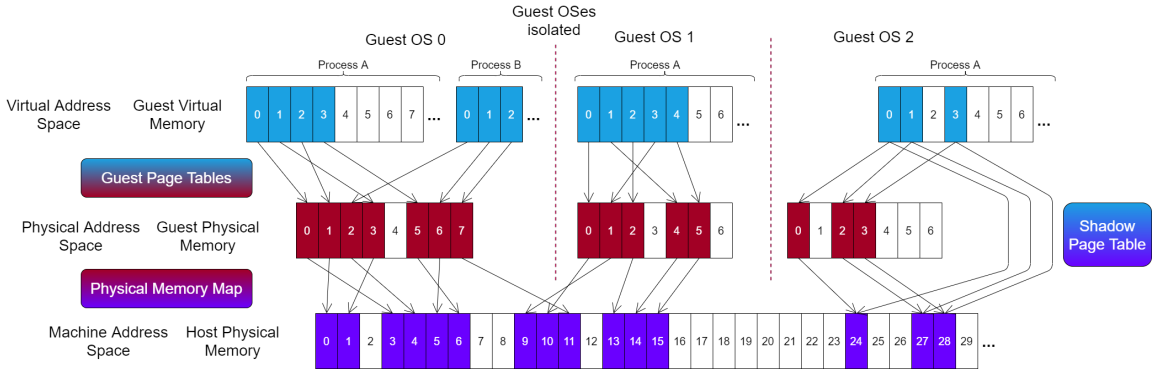
This can be achieved through function pointers. sensitive instructions run through function pointers, if virtualized, the pointer goes to a hypervisor specific function to resolve sensitive operation.



Memory Virtualization

We must prevent conflicts between guest OSes (where two guest OSes use the same frame in the host's physical memory). Hence we add an extra layer of indirection to translate:

Guest Virtual Addresses → **Guest Physical Addresses** → **Host Physical Addresses**



- Virtual Address Space uses Virtual Page Numbers (VPNs)
- Guest Page Tables translate VPNs → PPNs
- Physical Address Space uses Physical Page Numbers (PPNs)
- Physical Memory Map (**pmap**) translates PPNs → MPNs
- Machine Address Space uses Machine Page Numbers (MPNs)

Shadow Page Tables

Shadow Page Tables combine the two stage process of translating through the guest page tables and the physical memory map into one. The Guest OS is unaware of the shadow page table (managed by the hypervisor), which contains direct mappings to speed up address translation. When updating the shadow page table, the guest page tables & physical memory map are used.

- Hardware **MMU** (Memory Management Unit) uses shadow Page Tables.
- Hardware **TLB** maps VPNs → MPNs

On a **TLB** miss:

1. **Search Shadow Page Table**
   If the mapping found, update TLB & restart the instruction.
2. **Page Fault Handled by the VMM**
   (a) VMM attempts to find the **VPN** → **PPN** mapping in the guest OS page table.
   (b) If not there, then it is a **true page fault** to be handled by the Guest OS.
   (c) If it is found, it is a **hidden page fault** (Guest OS unaware). **VMM** updates the **pmap** and the shadow page table.

In order to use **Shadow Page Tables** they must be kept in sync with the Guest OS's Page Tables. This can be achieved in two ways:

Software

Mark page tables as read only, this way when the Guest OS writes to a page table, it traps and the **VMM** can take over to update both it & the Shadow Page table.

Traps are costly!

Hardware

Newer CPUs have hardware Support. The hardware itself does the required lookup on TLB miss, removing the need to use a shadow Page Table. (hardware implements the behaviour of a shadow page table).

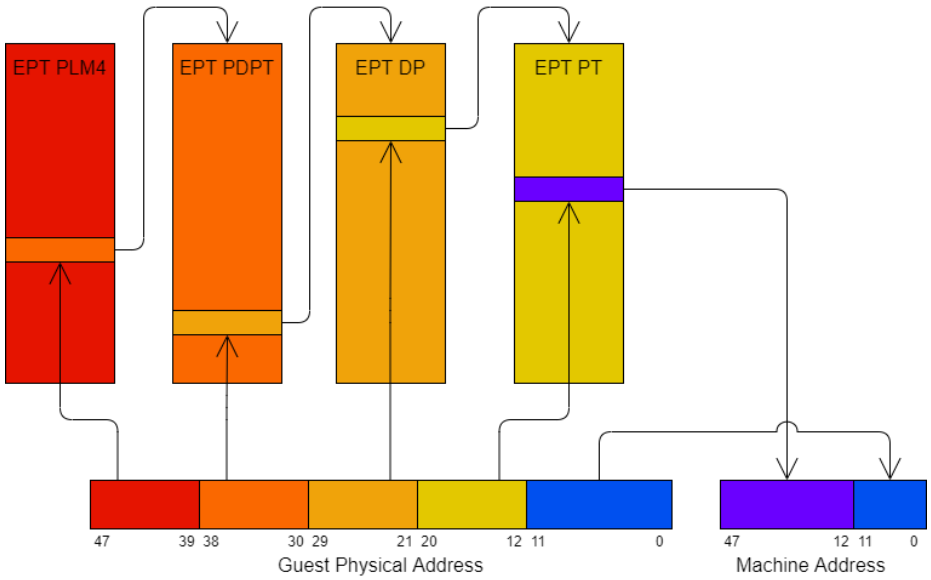
- AMD's Nested Page Tables (NPTs)
- Intel's extended Page Tables (EPTs)

On a **TLB** miss:

1. **MMU** searches for mapping in guest page table.
2. If not mapping found, it is a **true page fault**, hand control back to guest OS to resolve.
3. If mapping found:
   (a) MMU searches for mapping in **pmap**
   (b) If mapping found, update TLB & restart instruction.
   (c) else it is a **hidden page fault**, hand control to **VMM** to resolve it.

Intel EPT

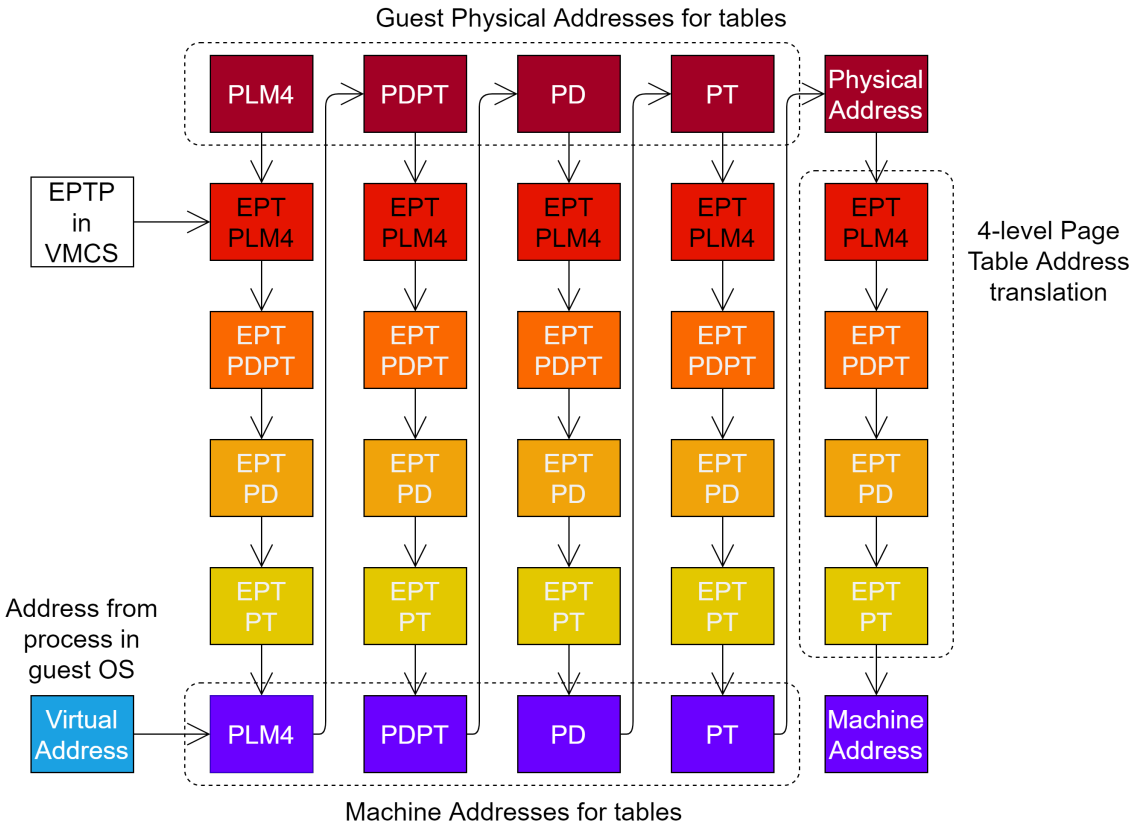
Intel Extended Page Table make use of a 4-Level page table as below:



In a **TLB** cache miss on a physical address (PPN) (e.g the hypervisor's page table), only 4 memory

accesses are required (4 level page table).

However when translating from virtual addresses (from the guest OS) far more memory accesses are required as translations are required for the 4 tables (24 accesses).



Paging Problems

The combined guest OS's physical memory can be larger than the Host's physical memory. In order to ensure the physical memory of Guest OSes is usable, we must swap guest physical pages to and from host physical memory.

- Hypervisor has little information on which pages the OS is using, Guest OS knows more about active pages.
- **Double Paging Problem**

  1. **VMM** select page P to be evicted/paged out.
  2. Guest OS selects page P to be evicted/paged out, accesses P brining it back from swap.
  3. Guest OS writes P to virtual disk.

  Here we have brought back a page, only for it to be immediately written to virtual disk.