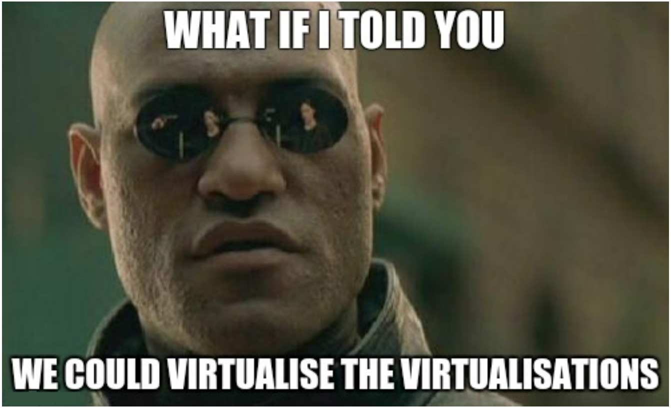


http://memegenerator.net

Definition (Popek-Goldberg [8, Section 1]: **Virtual Machine (VM)**)

A **Virtual Machine (VM)** is an efficient, isolated duplicate of some physical machine.

Definition (Popek-Goldberg [8, Section 1]: **Virtual Machine Monitor (VMM)**)

A **Virtual Machine Monitor (VMM)** is a software component that manages VMs, offering

1. **equivalence** (or **fidelity**) : instructions executed on a virtual machine have identical behaviour to doing so on the physical machine (bar resource availability and timing differences)
2. **isolation** (or **safety**) : VMM has complete control of all resources provided by the physical machine
3. **efficiency** : a statistically dominant proportion of instructions are executed without intervention by VMM

- If you accept the idea that

$$\text{VM} \;\simeq\; \text{execution environment},$$

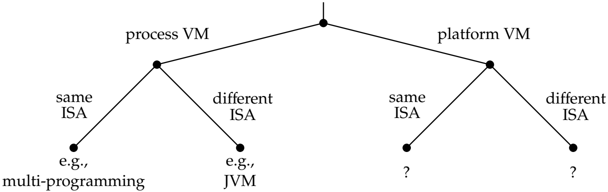
then, depending on your perspective,

$$\begin{array}{rcll} \text{process} & \Rightarrow & \text{VM} \simeq \text{ABI} & \text{VMM} = \text{kernel} \\ \text{kernel} & \Rightarrow & \text{VM} \simeq \text{ISA} & \text{VMM} = \;\;\; ? \end{array}$$

- That is,

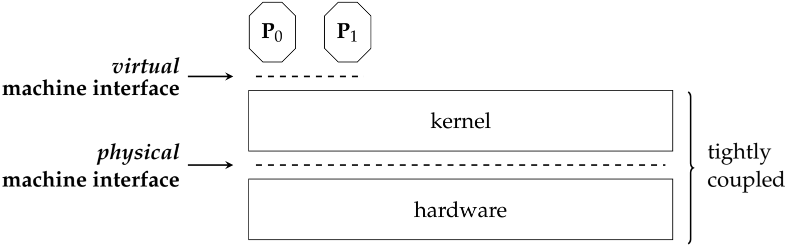
$$\begin{array}{rcl} \textbf{process VM} & = & \text{execution environment for a process} \\ \textbf{platform VM} & = & \text{execution environment for a kernel} \end{array}$$

and so per [12, Figure 1.13]



meaning a platform VMM is a sort of "kernel for kernels".

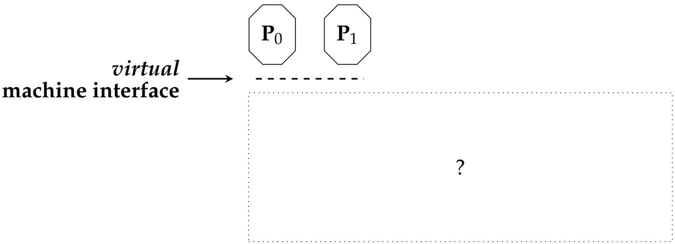
- Motivation: **utility computing $\subset$ cloud computing**.
  - Consider some user who wants to execute some processes (e.g., web-servers).
  - Option #1: they purchase a dedicated, physical platform, i.e.,



which means

+ dedicated access to resources,
– sole burden wrt. capital investment,
– sole burden wrt. maintenance,
– static provisioning of resources.

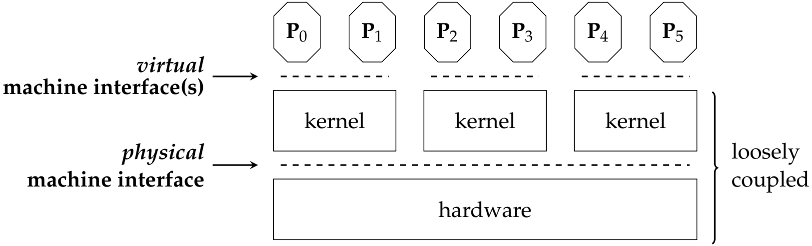
- Motivation: **utility computing** $\subset$ **cloud computing**.
  - Consider some user who wants to execute some processes (e.g., web-servers).
  - From their perspective, the platform is more like



st.

- provided the virtual machine interface is consistent their processes can still execute,
- i.e., what the (now abstract) platform *is* will have limited importance.

- Motivation: **utility computing $\subset$ cloud computing**.
  - Consider some user who wants to execute some processes (e.g., web-servers).
  - Option #2: they lease a shared, virtual platform, i.e.,



which means

– shared access to resources,
+ shared burden wrt. capital investment,
+ shared burden wrt. maintenance,
+ dynamic provisioning of resources

but, to support this option we need a kernel (i.e., VMM) to manage the kernels (i.e., VMs) ...

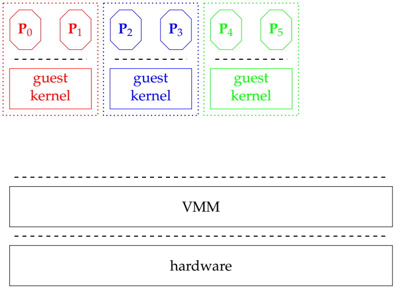
Definition (Goldberg [5, Page 22]: **VMM architectures**)

There are two classical platform VMM architectures, namely:

A so-called

$$\begin{array}{lcr} \textbf{type-}1\ \textbf{VMM} & \simeq & \text{native VMM} \\ & \simeq & \text{infrastructure VMM} \end{array}$$

which executes on bare-metal hardware, i.e.,



examples of which include VMware ESX.

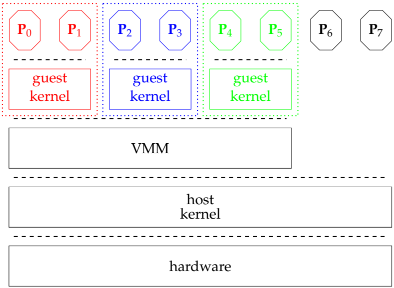
Definition (Goldberg [5, Page 22]: **VMM architectures**)

There are two classical platform VMM architectures, namely:

A so-called

type-2 VMM $\simeq$ hosted VMM
$\simeq$ personal VMM

which executes on (or in) another kernel i.e.,



examples of which include VMware Workstation.

Definition (Popek-Goldberg [8, Section 2]: "third-generation" **physical machine**)

The state of a physical machine is a tuple

$$S = (E, M, P, R)$$

where

- E is the q-element executable storage (or memory) whose j-th element is denoted $E[j]$,
- $M \in \{s, u\}$ is the processor mode (s for supervisor or kernel mode, u for user mode),
- P is the program counter, and
- $R = (l, b)$ is the relocation register st. an access to address $0 \leq x < b$ is relocated per $E[x + l]$

st. (M, P, R) is the **Program Status Word (PSW)**. If C is the space of all such states, an instruction is then a function $i : C \rightarrow C$ that allows us to write

$$S_1 = i(S_0)$$

or

$$(E_1, M_1, P_1, R_1) = i((E_0, M_0, P_0, R_0))$$

for example.

Definition (Popek-Goldberg [8, Section 2]: **traps**)

An instruction i is said to **trap** (i.e., raise an interrupt) if

$$(E_1, M_1, P_1, R_1) = i((E_0, M_0, P_0, R_0))$$

where

1. $E_1[j] = E_0[j]$ for $0 < j < q$,
2. $E_1[0] = (M_0, P_0, R_0)$, and
3. $(M_1, P_1, R_1) = E_0[1]$

i.e., the executable storage is unchanged, bar the 0-th element where the PSW *before* the trap is stored; the PSW *after* the trap is loaded from the 1-st element. Normally we'd expect $M_1 = s$ and $R_1 = (0, q-1)$.

Definition (Popek-Goldberg [8, Section 2]: **instruction behaviours**)

Execution of a

- **privileged instruction** will trap if the processor is in user mode, but do not trap if it is in kernel mode,
- **control sensitive instruction** will attempt to change the configuration of resources,
- **behaviour sensitive instruction** will depend on the configuration of resources.

Theorem (Popek-Goldberg [8, Theorem 1])

An effective VMM may be constructed for a physical machine, if the set of sensitive instructions is a sub-set of the privileged instructions.

Theorem (Popek-Goldberg [8, Theorem 2])

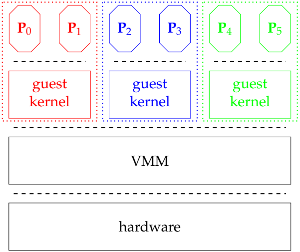
A physical machine is *recursively* virtualisable if a) it is virtualisable, and b) a VMM without any timing dependencies can be constructed for it.

Theorem (Popek-Goldberg [8, Theorem 1])

An effective VMM may be constructed for a physical machine, if the set of sensitive instructions is a sub-set of the privileged instructions.

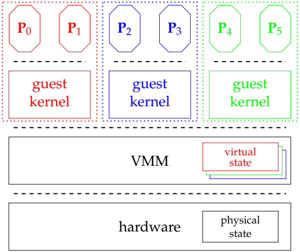
Concept: platform virtualisation (4)
The practice

- Idea:



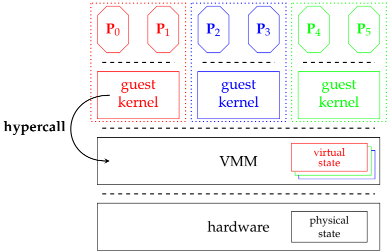
Concept: platform virtualisation (4)
The practice

- Idea:



1. maintain some virtual state.

- **Idea**:

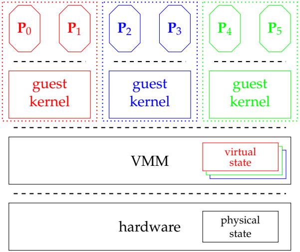


1. maintain some virtual state,
2. *optionally* allow interaction between VM and VMM yielding either

full virtualisation	:	kernel is totally unmodified
para-virtualisation [4]	:	kernel is selectively modified

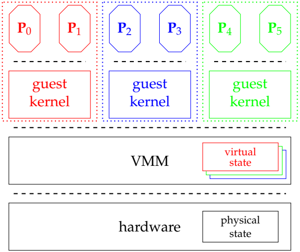
Concept: platform virtualisation (4)
The practice

- Idea:



1. maintain some virtual state,
2. *optionally* allow interaction between VM and VMM,
3. implement the VMM using option #1: **emulate**.
   - – close to no instructions executed directly on physical platform,
   - + low(er) efficiency,
   - + isolation of VMs is by default (and perfect).

- Idea:

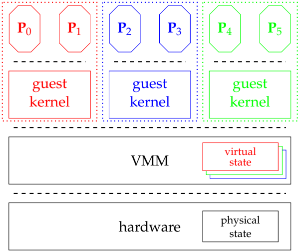


1. maintain some virtual state,
2. *optionally* allow interaction between VM and VMM,
3. implement the VMM using option #2: **trap+emulate** (or **classic virtualisation**).

   + close to all instructions executed directly on physical platform,
   + high(er) efficiency,
   – isolation of VMs needs careful attention.

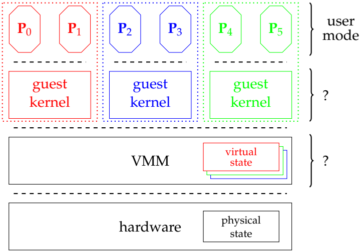
Concept: platform virtualisation (5)
The practice

- Idea: **trap+emulate**.



Concept: platform virtualisation (5)
The practice

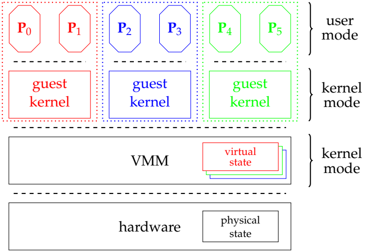
- Idea: **trap+emulate**.



1. De-privilege kernel relative to VMM, st.
    - a non-sensitive instruction will execute as is, whereas
    - a sensitive instruction will raise an interrupt

   thus ensuring efficiency.

- Idea: **trap+emulate**.

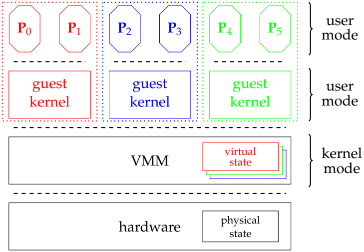


1. De-privilege kernel relative to VMM, st.
    - a non-sensitive instruction will execute as is, whereas
    - a sensitive instruction will raise an interrupt

   thus ensuring efficiency.

Concept: platform virtualisation (5)
The practice

- Idea: **trap+emulate**.

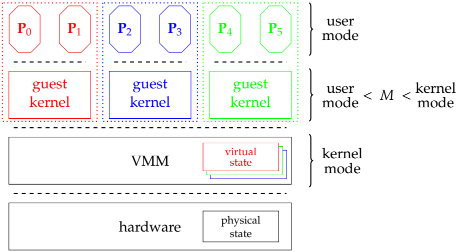


1. De-privilege kernel relative to VMM, st.
   - a non-sensitive instruction will execute as is, whereas
   - a sensitive instruction will raise an interrupt

   thus ensuring efficiency.

Concept: platform virtualisation (5)
The practice

- Idea: **trap+emulate**.

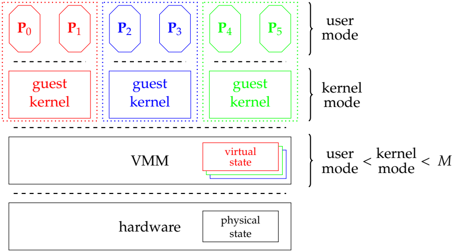


1. De-privilege kernel relative to VMM, st.
   - a non-sensitive instruction will execute as is, whereas
   - a sensitive instruction will raise an interrupt

   thus ensuring efficiency.

Concept: platform virtualisation (5)
The practice

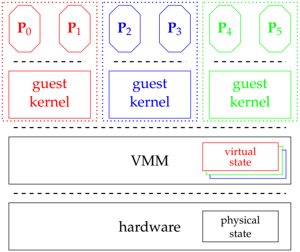
- Idea: **trap+emulate**.



1. De-privilege kernel relative to VMM, st.
   - a non-sensitive instruction will execute as is, whereas
   - a sensitive instruction will raise an interrupt

   thus ensuring efficiency.

- Idea: **trap+emulate**.

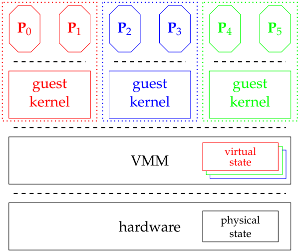


1. De-privilege kernel relative to VMM, st.
   - a non-sensitive instruction will execute as is, whereas
   - a sensitive instruction will raise an interrupt

   thus ensuring efficiency.

Concept: platform virtualisation (5)
The practice

- Idea: **trap+emulate**.



2. VMM handles interrupts, e.g., by
   - updating physical state,
   - updating virtual state, or
   - delegating to VM, performing a virtual interrupt

   thus ensuring equivalence and isolation.

- Example: $\mathbf{P}_0$ executes a system call.

	should handle	does handle
type-1		
type-2		

- Example: $\mathbf{P}_0$ executes a system call.

	should handle	does handle
type-1	guest kernel	
type-2	guest kernel	

Concept: platform virtualisation (6)
The practice

- Example: $\mathbf{P}_0$ executes a system call.

	should handle	does handle
type-1	guest kernel	VMM
type-2	guest kernel	host kernel $\rightsquigarrow$ VMM

so, the interrupt handling mechanism becomes

1. (non-privileged) `syscall` execution causes interrupt,
2. interrupt handled by host kernel,
3. host kernel delegates to VMM,
4. VMM delegates to VM and hence guest kernel,
5. ...
6. (privileged) `rfe` execution causes interrupt,
7. interrupt handled by host kernel,
8. host kernel delegates to VMM,
9. VMM emulates wrt. virtual state

where some steps are merged for the type-1 case.

- Example: **D** raises an interrupt.

	should handle	does handle
type-1	guest kernel	VMM
type-2	guest kernel	host kernel $\rightsquigarrow$ VMM

so, the interrupt handling mechanism becomes

1. device raises interrupt,
2. interrupt handled by host kernel,
3. host kernel delegates to VMM,
4. VMM delegates to VM and hence guest kernel via *virtual interrupt*,
5. ...
6. (privileged) `rfe` execution causes interrupt,
7. interrupt handled by host kernel,
8. host kernel delegates to VMM,
9. VMM emulates wrt. virtual state

where some steps are merged for the type-1 case.

- Example: $\mathbf{P}_0$ causes a TLB fault.

	should handle	does handle
type-1	guest kernel	VMM
type-2	guest kernel	host kernel $\rightsquigarrow$ VMM

so, the interrupt handling mechanism becomes

1. TLB fault causes interrupt,
2. interrupt handled by host kernel,
3. host kernel delegates to VMM,
4. VMM delegates to VM and hence guest kernel,
5. ...
6. (privileged) TLB update causes interrupt,
7. interrupt handled by host kernel,
8. host kernel delegates to VMM,
9. VMM emulates wrt. virtual state, *using physical TLB*

where some steps are merged for the type-1 case.

Concept: platform virtualisation (8)

The gap between theory and practice

- **Problem**: an ISA may be imperfect wrt. the Popek-Goldberg requirements, e.g.,
  1. x86 [9]: `popf` pops from the stack into a PSW-like register
  2. ARM [7]: `msr` transfers a general-purpose register into a PSW-like register

  but neither raises an interrupt in user mode!

Concept: platform virtualisation (8)

The gap between theory and practice

- Problem: an ISA may be imperfect wrt. the Popek-Goldberg requirements, e.g.,

  1. x86 [9]: `popf` pops from the stack into a PSW-like register
  2. ARM [7]: `msr` transfers a general-purpose register into a PSW-like register

  but neither raises an interrupt in user mode!

- Solution #1: **dynamic binary translation** (cf. JIT compilation).
  - Starting at kernel entry point, scan through machine code.
  - Re-write (or "patch") sensitive instructions into interrupt-generating alternatives, e.g.,

```
...
popf
...
```

$\rightsquigarrow$

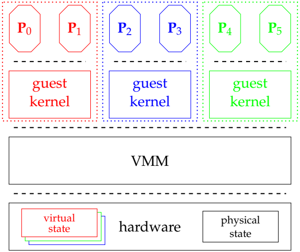
```
...
int 0xff
...
```

  - Now *all* sensitive instructions are protected, so one can use trap+emulate as before.

- Problem: an ISA may be imperfect wrt. the Popek-Goldberg requirements, e.g.,
  1. x86 [9]: `popf` pops from the stack into a PSW-like register
  2. ARM [7]: `msr` transfers a general-purpose register into a PSW-like register

  but neither raises an interrupt in user mode!
- Solution #2: **VM assist** (i.e., hardware support) to
  - reduce overhead of per instruction emulation, and/or
  - offer an **Interpretive Execution Facility (IEF)** [6, Part 2],

  e.g., for the latter



Intel VT-x (or "Vanderpool") [12, Section 8.7]:

- add a VMX mode at ring −1 in which VMM executes,
- support a Virtual Machine Control Structure (VMCS),
- support a range of special-purpose instructions (e.g., `vmlaunch`, `vmresume`),
- VMM selects when control is transferred (e.g., can constrain system calls *inside* VM).

Conclusions

- Take away points: this topic is
  - important, in the sense it explains how concepts like cloud computing are realised,
  - useful, in the sense it integrates with other topics (cf. language engineering),
  - interesting, in the sense it is underpinned by some *really* neat theoretical CS

  but, the technical challenge wrt.

  - processor (e.g., scheduling VMs)
  - memory (e.g., shadow page tables) and
  - I/O device

  virtualisation are significant: we've only touched the surface!

References

[1] Wikipedia: Cloud computing.
https://en.wikipedia.org/wiki/Cloud_computing.

[2] Wikipedia: Utility compiting.
https://en.wikipedia.org/wiki/Utility_computing.

[3] Wikipedia: Virtualisation.
https://en.wikipedia.org/wiki/Virtualization.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield.
Xen and the art of virtuaization.
In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[5] R.P. Goldberg.
Architectural Principles for Virtual Computer Systems.
PhD thesis, Harvard University, 1973.

[6] P.H. Gum.
System/370 extended architecture: Facilities for virtual machines.
IBM Journal of Research and Development, 27(6):530–544, 1983.

[7] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere.
Formal virtualization requirements for the ARM architecture.
Journal of Systems Architecture, 59(3):144–154, 2013.

References

[8] G.J. Popek and R.P. Goldberg.
Formal requirements for virtualizable third generation architectures.
Communications of the ACM (CACM), 17(7):412–421, 1974.

[9] J.S. Robin and C.E. Irvine.
Analysis of the Intel Pentium's ability to support a secure virtual machine monitor.
In *USENIX Security Symposium*, 2000.

[10] J.E. Smith and R. Nair.
Chapter 3: Process virtual machines.
In *Virtual Machines: Versatile Platforms for Systems and Processes* [12].

[11] J.E. Smith and R. Nair.
Chapter 3: System virtual machines.
In *Virtual Machines: Versatile Platforms for Systems and Processes* [12].

[12] J.E. Smith and R. Nair.
Virtual Machines: Versatile Platforms for Systems and Processes.
Elsevier, 2005.

[13] A.S. Tanenbaum and H. Bos.
Chapter 7: Virtualization and the cloud.
In *Modern Operating Systems*. Pearson, 4th edition, 2015.