Virtualization

Azza Abouzied

A Recap of everything

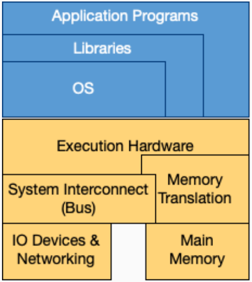
Central OS Concept: Abstraction

We build systems on levels of abstractions. Higher levels hide lower level details.

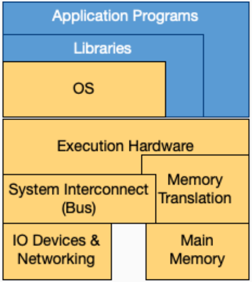
Abstractions so far:

1. Processes: abstract CPU, multiple programs.
2. Device Drivers: hide details of hardware
3. Virtual Memory: abstract memory

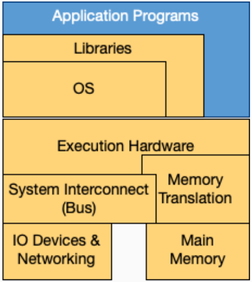
The right level of Abstraction



Instruction Set Architecture
Hardware / Software Divide
OS Developers



Application Binary Interface
ISA Calls & System Calls
Compiler Developers



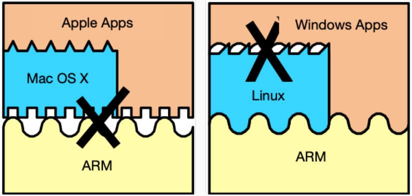
Application Program Interface
Library Calls
Application Developers

Why do we abstract?

1. Decouple problems
2. Hardware and software development out of sync
3. Run software on any machine

But the reality is:

1. Software for one ISA will not run on hardware with different ISA e.g. ARM vs. x86
2. Same ISAs different OS



Focused Only on Hardware Abstraction

OS manages hardware e.g. memory, or interfaces with device drivers
$\rightarrow$ **Can't share hardware without OS**

If you want to use the hardware, you have to stick with the OS & its design choices

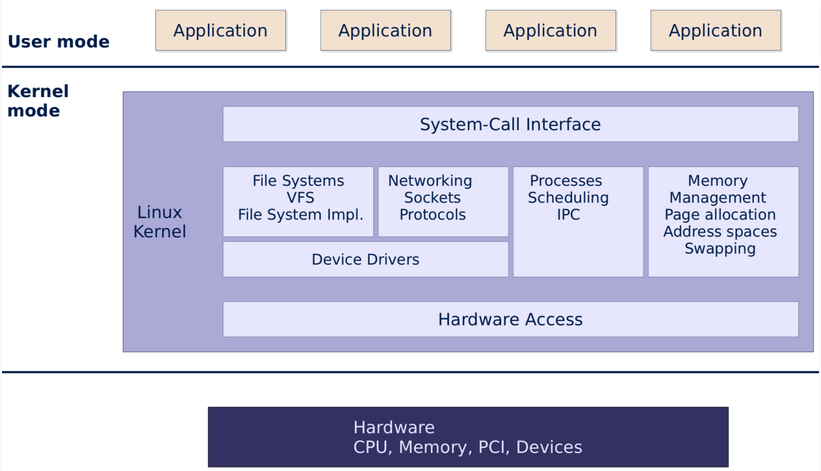
We need Modularity / Plug-n-Play Services

If you are using an OS, you are vulnerable to attacks because of users sharing the OS!

We also need Isolation

An OS Goal: Strong Isolation

Monolithic Kernels: Linux



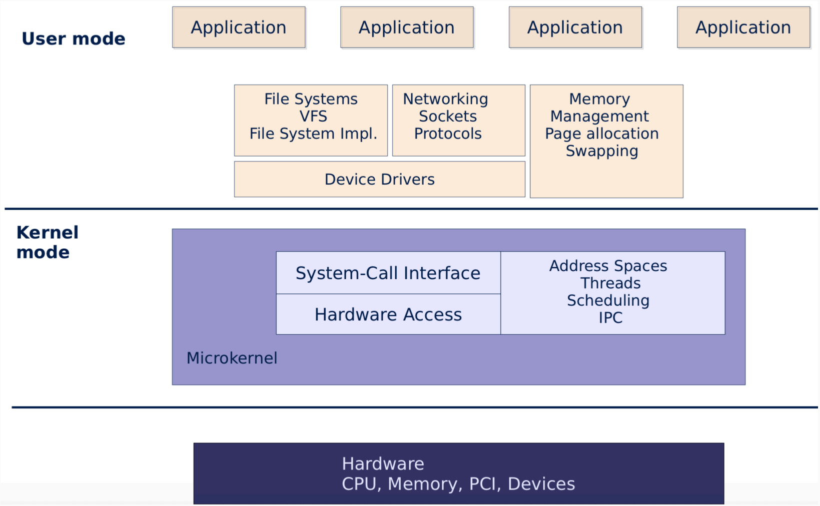
What’s the problem?

1. Security issues
    - Everything within the kernel runs in privileged mode
    - Direct access to all kernel-level data
    - Haven for rootkits
2. Resilience issues
    - A faulty device can crash the whole system
    - Today’s kernels have lots of drivers (more than 50% of the codebase)
3. Software Complexity

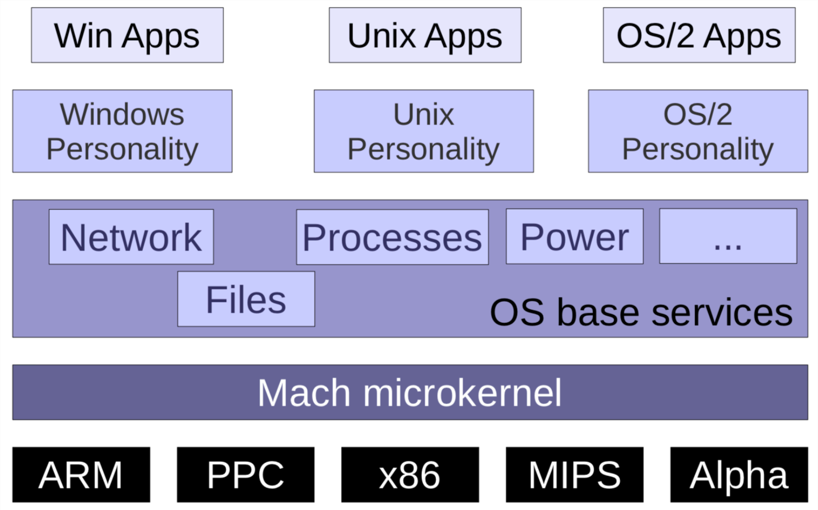
A Microkernel

1. Minimal OS kernel
   - Small Trusted Computing Base
   - Can be verified (formally)
2. User-level services
   - Flexible & Extensible
3. Protection between components
   - More resilient: crashing component does not (necessarily...) crash the whole system
   - More secure

The Microkernel Vision



A Microkernel Case-study: IBM Workplace OS



A Microkernel Case-study: IBM Workplace OS

Never finished (but spent 1 billion $), Why?

- Underestimated difficulties in creating OS personalities
- Forced divisions to adopt new system without having a system
- Second System Effect: too many fancy features
- Slow & Somewhat still complex

What are a μKernel's main advantages?

There are always research μKernels popping up: Minix, L4, Singularity (MSR), etc.

- Subsystem protection / isolation
- Small code size
- Can be adapted to embedded, real-time, secure systems, etc.

"A microkernel does no real work!" — Jochen Liedtke

It only provides **inevitable** mechanisms: Abstractions such as threads and address spaces and Mechanisms such as communication, resource mapping, and maybe scheduling.

Virtual Machines

Isolation++

Microkernel: Isolated Processes & OS services

Virtual Machines: Isolate Complete Operating Systems
Side-effect: Balance Isolation with Compatability

How to implement a VMM?: Emulation

Pure emulation (e.g. QEMU, Bochs): *VMM interprets every guest instruction*

```
for(;;){
    read_instruction();
    switch(decode_instruction_opcode()){
        case OPCODE_ADD:
            int src = decode_src_reg();
            int dst = decode_dst_reg();
            regs[dst] = regs[dst] + regs[src];
            break;
        case OPCODE_SUB:
            ...
            regs[dst] = regs[dst] - regs[src];
            break;
        ...
    }
    eip += instruction_length;
}
```

Challenges

Pure Emulation is very slow! Every instruction needs to be interpreted.

Some workarounds: Patch guest instructions directly to hardware. Generally works for user code.

1. But what about privileged instructions?
2. What hardware state should we virtualize?

Principles of a VM Architecutre

Equivalence

VM indistinguishable from the underlying hardware

Resource Control

VM in complete control of any virtualized resources

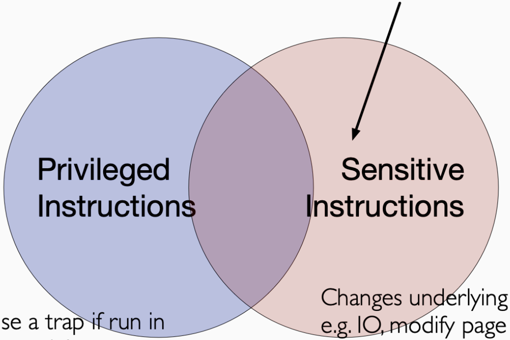
Efficiency

Most instructions should be executed directly on the underlying CPU without involving the Hypervisor

Popek & Goldberg 1974

Equivalence: Breaking the Illusion

IA32: Read Descriptor Tables (e.g. interupt vectors - if a hypervisor is lying about where interrupts point to the GuestOS can find this out!



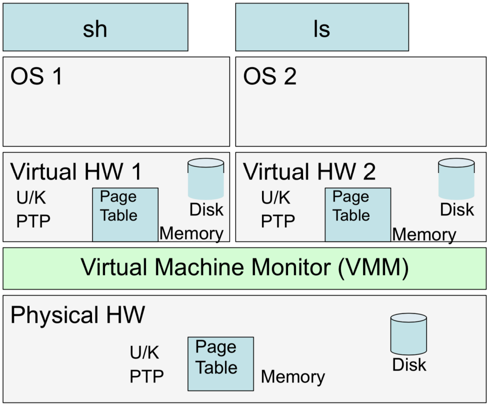
Cause a trap if run in user mode!

Changes underlying resources e.g. IO, modify page tables, determine privilege (e.g. U/K bit)

Equivalence: Keeping the Illusion

1. **Use CPU's breakpoint mechanism:** Scan code to figure out where to put breakpoints → Overhead
2. **Use code-rewrite:** Replace critical instructions with system call to hypervisor → breaks illsuion → *Trick: mark rewritten pages as non-readable, trap and give it original page.*
3. **Paravirtualization:** Guest OS rewrites itself to run on VMM → breaks illsuion, compatiblity
4. **Use CPU support/accelerated virtualization/hardware-assisted:** two new modes: host mode, guest kernel mode, and user mode the same.

What and how to virtualize

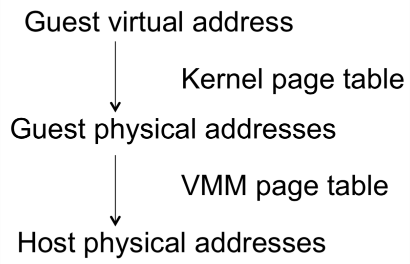


Each guestOS assumes it manages:

1. Physical memory
2. Page-table pointer
3. U/K bit
4. Interrupts, registers, etc.

How to virtualize memory

Add another level of indirection



VMM must translate guest OS addresses into actual memory addresses. e.g. if guest VM has 1GB of memory, it will access memory addrs [0 - 1GB], but VMM may map these to some other place in physical memory.

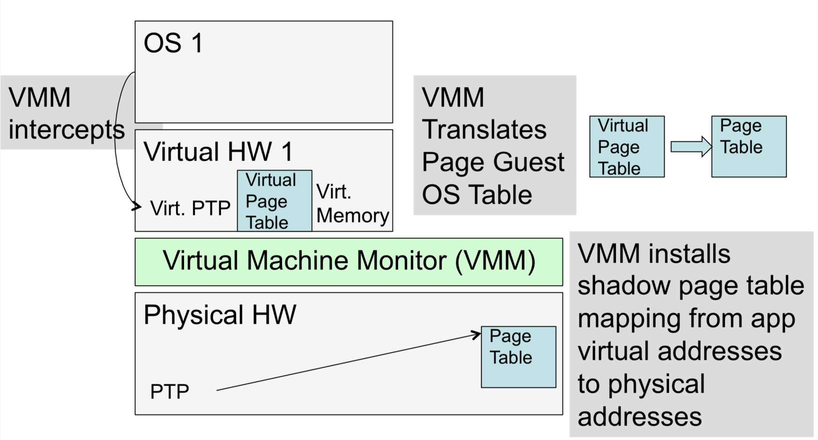
How to virtualize PTP?

- Guest VM's page table maps from guest VAs to guest PAs.
- Hardware page table must point to host PAs (actual DRAM locations).
- Setting hardware PTP register to point to guest PT would not work, since that would allow guest OS to choose which PAs it wants to access.

Process:

1. VMM intercepts guest OS loading PTP.
2. VMM iterates over guest PT and constructs shadow PT: Replacing guest physical addresses with corresponding host physical addresses
3. VMM loads host physical address of shadow PT into PTP

The Process in a Diagram



Translating from guest address to physical address

VMM Page Table

Guest PA	PA
0xA1	0xC0
0xA2	0xC1
0xA3	0xC4

Maps from guest physical address to real physical addresses

Guest OS Page Table

VA	Guest PA
0x01	0xA2
0x02	0xA3

Maps app virtual addresses to guest physical addresses

Real Page Table

VA	PA
0x01	0xC1
0x02	0xC4

Maps app virtual addresses to real physical addresses

Trap & Emulate: PT modification

1. Host maps guest PT read-only
2. Guest modifies its PT
3. If guest modifies, hardware generates page fault
4. Page fault handled by host: Update shadow page
5. Restart guest

Virtualizing the U/K bit

1. Hardware U/K bit must be U when guest OS runs otherwise guest OS can do whatever it wants $\rightarrow$ Strong isolation
2. Behavior affected by U/K bit
3. Execute privileged instructions: e.g. load PTP
4. Whether pages marked “read only” in page table can be modified.

Trap & Emulate: U/K bit

1. VMM stores guest U/K bit in some location
2. VMM runs guest kernel with U set
3. Privileged instructions will cause an exception, and VMM emulates privileged instructions. For example:
   - Set or read virtual U/K
   - if load PTP in virtual K mode, load shadow page table
4. Or raise exception in guest OS

Hardware Support Can Simplify Things Alot

AMD and Intel added hardware support

VMM operating mode, in addition to U/K

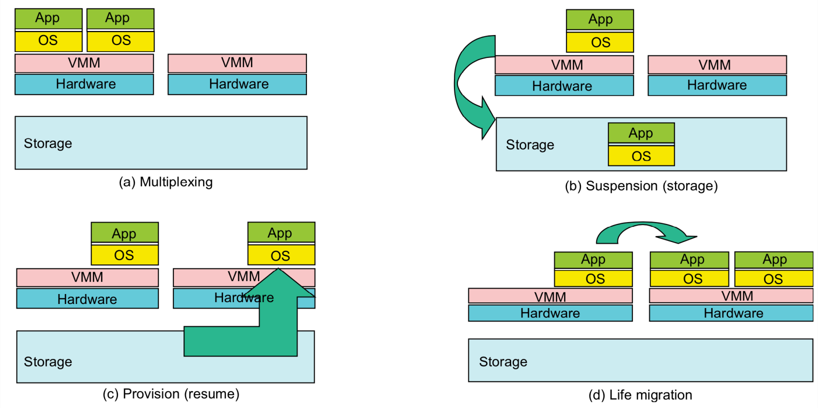
Two levels of page tables

Simplifies job of VMM implementer:

- Let the guest VM manipulate the U/K bit, as long as VMM bit is cleared.
- Let the guest VM manipulate the guest PT, as long as host PT is set.

Why does Amazon use/provide VMMs?

VMM Functions



VMM Benefits

Manageability

Ease maintenance, administration, provisioning

Performance

Overhead of virtualization should be small

Power Savings

Server Consolidation

Isolation

Activity of one VM should not impact other active VMs

Data of one VM is inaccessible by another

Scalability

Minimize cost per VM

Questions?