

Introduction to the Course

Luca Abeni

luca.abeni@santannapisa.it

February 21, 2020

Why this Course?

- Today, virtualization is very used in computing systems
 - From “low-power” devices (traditionally considered as embedded)...
 - ...To big clouds!
- Used for various reasons:
 - Security / safety
 - Running multiple OSs on the same machine
 - Server consolidation
 - “On-demand” allocation of computing resources
- Lots of very different technologies...

Virtualization?

- “Virtualization” is often used as a generic term, to indicate lots of different things
 - Different kinds of requirements and features...
 - ...Different levels of performance, security, flexibility
- People sometimes consider virtualization solutions and VMs as “black boxes”
 - Without caring about the used mechanisms or implementation details
- As a result, it becomes difficult to exactly understand the security and performance implications
- Need for better understanding of the various virtualization technologies and solutions!

Overview — 1

- Introduction to virtualization
- Various levels of virtualization
 - Hardware virtualization
 - OS-level virtualization
 - Containers
 - language-level virtualization, ...
- Virtualizing computing and non-computing resources
 - CPU virtualization
 - Emulation
 - Trap and emulate
 - Hardware-assisted virtualization
 - I/O devices virtualization, ...

Overview — 2

- Virtual abstractions
 - VMs vs containers
 - Containers as an abstraction (as opposite to containers as a technology)
- Virtualization architectures
- Real-Time virtualization
 - Latencies
 - Hierarchical scheduling
- Modern technologies
 - Lightweight VMs / MicroVMs
 - Unikernels, specialized/reduced monitors

Virtualization

- Virtualization: creation of a *virtual instance* of a computing system
 - Computer (PC, server, embedded board, ...)
 - Operating System
 - Storage device / other
- Separate / independent from the physical system(s) hosting it
- This mainly requires two activities:
 1. **Pooling**: consolidating possibly distributed resources into a single logical entity
 2. **Isolation**: creating the impression that the virtualized application has a private copy of the resources

Resource Pooling

- Set of multiple, possibly distributed, resources
- Single “virtual resource”, that can be used to transparently access them
 - Pool of physical servers hosting VMs in a cloud; accessed by starting a VM \Leftarrow load balancing
 - Pool of storage devices (disks, databases, ...) accessed as a single virtual storage device \Leftarrow I do not know where data are really stored...
 - ...
- Used for automatically distributing the load, for building powerful machines based on less powerful ones, for making computation independent on data placement, ...

Resource Isolation

- The usage of virtual resources must be controlled by the virtualization software
 - Example: applications running in a VM should not be able to access resources outside of the VM...
 - ...Nor to directly access physical resources!
 - Or: applications using a virtual storage device should not be able to even see the physical storage devices
 - ...
- Virtual resources should not even be distinguishable from physical ones
 - Example: applications running in a VM should have the impression to run on a physical machine...

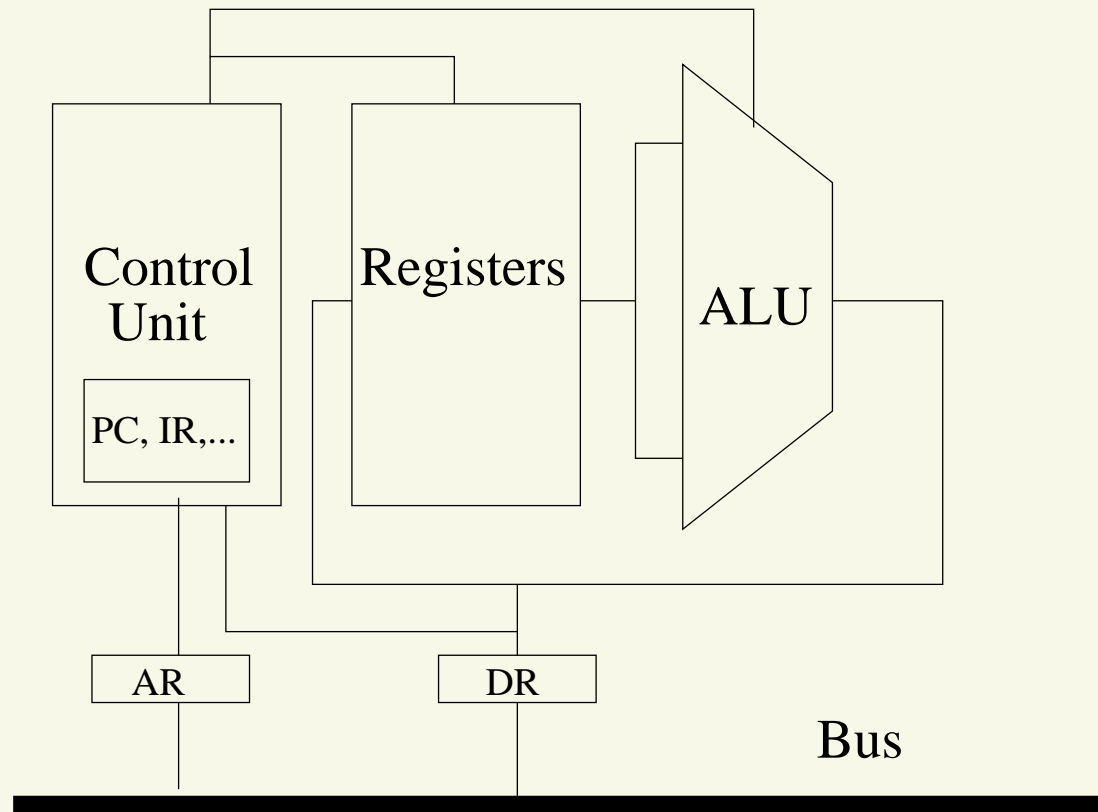
Different Kinds of Isolation...

- Resource isolation can be used for different reasons
 - Security ← reduce the impact of compromised subsystems
 - Application sandboxing ← execute non-trusted software
 - Performance guarantees ← isolate the performance of a component from interference of other components
 - ...
- Different kinds of requirements

...And Different Kinds of Virtualization

- Example: Compute virtualization ← virtualization of a computing element
- Different kinds of abstractions, ranging from Virtual Machine (virtual instance of a PC) to “virtual language runtime”
 - Abstraction: virtual PC → probably strongest type of isolation, but some overhead
 - Abstraction: JVM → much less isolation (but the virtualization overhead can be reduced)
- Common concept: **Abstract Machine**... Let’s look at it
 - Starting from the very beginning: physical machine

Example of (Toy) CPU

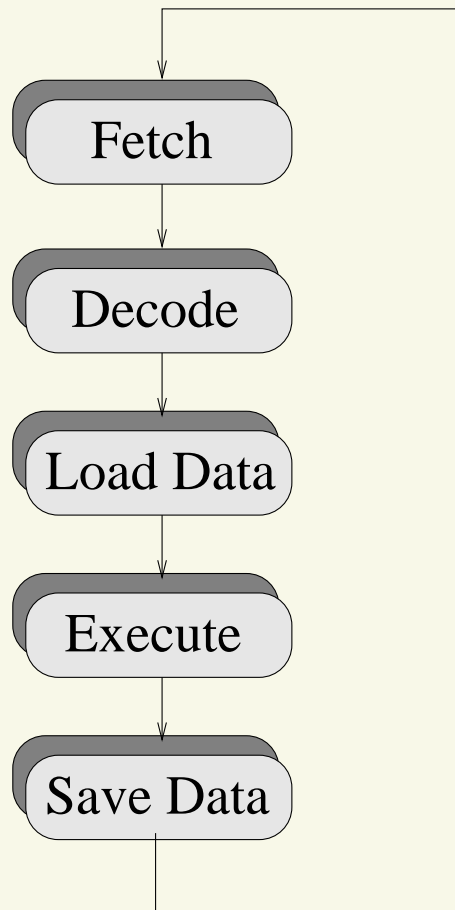


- Toy CPU: just an example with many simplifications
- Modern (real) CPUs are much more complex!
 - Pipeline
 - Parallel execution

CPUs, Programs, & Friends

- CPU → executes programs
 - Stored in main memory
 - Use data from main memory
- Program: formal description of an algorithm
 - Using a programming language
- Sequence of machine instructions
 - **Actions** having **effects** on some **objects**
 - “Object”: data stored in main memory
- Instance of program in execution: sequence of actions on objects
 - Example: `int mcd(int a, int b)` and its execution

Executing a Program



- CPU: cyclical execution (fetch / decode / load / execute / save)
 - Machine instructions are executed (mainly) sequentially
- Machine designed to execute its own language!
 - Machine Language

Physical Machines...

- Computer: (physical) machine designed to execute programs
- Every machine executes programs written in **its own language**
- Relationship between **machine** and **language**
 - A machine has its own language (the language it can parse and execute)
 - A language can be “understood” (parsed and executed) by multiple different machines
- Program execution: (infinite) cycle
fetch/decode/load/execute/save
 - CPU: hw implementation of this cycle

...And Abstract Machines!

- The fetch/decode/load/execute/save cycle can be implemented in hw or in sw...
- Software Implementation: **Abstract Machine**
 - Algorithms and data structures used to **store** and **execute** programs
- Once upon a time referred as “*Virtual Machine*”
 - Today, the term “Virtual Machine” (VM) is used with a slightly different meaning

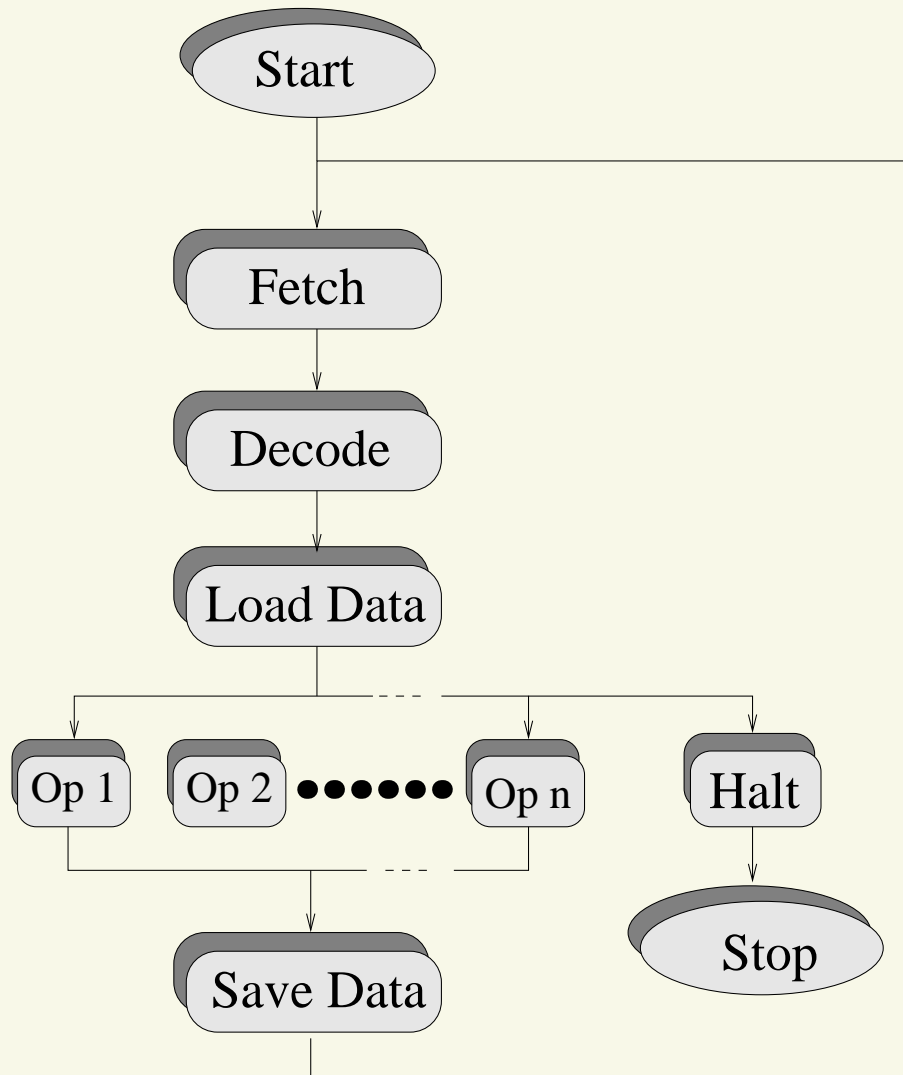
Abstract Machines and Languages

- Similarly to physical machines (CPUs), each abstract machine has its own machine language
 - Machine language for a CPU: sequence of 0 / 1
 - Assembly makes it more readable
 - Abstract machines generally have higher level machine languages (C, Java, etc...)
- $\mathcal{M}_{\mathcal{L}}$: abstract machine understanding language \mathcal{L}
 - \mathcal{L} is the *machine language* of $\mathcal{M}_{\mathcal{L}}$
 - Program: sequence of instructions written in \mathcal{L}
- $\mathcal{M}_{\mathcal{L}}$ is just a possible way to describe \mathcal{L}

Abstract Machines Behaviour

- To execute a program written in \mathcal{L} , $\mathcal{M}_{\mathcal{L}}$ has to:
 1. Execute some “elementary operations”
 - In hw, ALU
 2. Manage the execution flow
 - Execution is not only sequential (jumps, loops, etc...)
 - In hw, PC handling
 3. Move data from / to memory
 - Addressing modes, ...
 4. Take care of memory management
 - Dynamic allocation, stack management, etc...

Abstract Machine Example



- Execution cycle: very similar to a CPU...
- ... But it is implemented in software!

Virtualized Resources

- Virtual Machine: efficient, isolated duplicate of a **physical machine**
 - Why focusing on *physical* machines?
 - What about abstract machines?
- Software stack: hierarchy of abstract machines
 - ...
 - Abstract machine: **language runtime**
 - Abstract machine: **OS** (hardware + system library calls)
 - Abstract machine: **OS kernel** (hardware + syscalls)
 - **Physical machine** (hardware)

Hardware Virtualization

- Can be full hardware virtualization or paravirtualization
 - Paravirtualization requires modifications to guest OS (kernel)
- Can be based on trap and emulate
- Can use special CPU features (hardware assisted virtualization)
- **In any case, the hardware (whole machine) is virtualized!**
 - Guests can provide their own OS kernel
 - Guests can execute at various privilege levels

OS-Level Virtualization

- The OS kernel (or the whole OS) is virtualized
 - Guests can provide the user-space part of the OS (system libraries + binaries, boot scripts, ...) or just an application...
 - ...But continue to use the host OS kernel!
- One single OS kernel (the host kernel) in the system
 - The kernel virtualizes all (or part) of its services
- OS kernel virtualization: container-based virtualization
- Example of OS virtualization: wine

Virtualization at Language Level

- The language runtime is virtualized
 - Often used to achieve independence from hardware architecture
- Example: Java Virtual Machine
- Often implemented by using emulation techniques
 - Interpreter or just-in-time compiler

Virtual Machines

- Virtual Machine: efficient, isolated duplicate of a physical machine
 - Execution environment essentially identical to the physical machine
 - Programs only see a small decrease in speed
 - A “monitor” or “hypervisor” is in full control of physical resources
- Programs running in a VM should not see differences respect to real hw
- Virtualization should be efficient
- Programs should not be able to access resources outside of the VM

VMs and OSs

- How is an OS related to Virtual Machines?
 - The OS should provide support for the Virtual Machine Monitor / hypervisor
 - The OS *could* be optimized to run inside a VM
- OS support for virtualization (as host or as guest)
 - Impact on resource management
 - Impact on the exposed features
 - Impact on the I/O devices support
- Impact on the OS architecture?
 - Host: type-1 hypervisors, μ -kernel systems
 - Guest: library OSs, unikernels, vertically structured OSs

CPU Virtualization

- First idea: simulate the CPU hw in software
 - Software implementation of an abstract machine implementing the fetch-decode-execute-(write) cycle
 - **Fails** the **efficiency** requirement!!!
- Other idea: directly execute the virtualized instructions on the CPU
 - Virtual ISA: exact copy of the host ISA
 - Might fail the third (VMM is in control) requirement
 - Limited to unprivileged instructions (with VMM executing at a high privilege level)
 - What to do for privileged instructions?

Virtualizable CPU Architectures

- The monitor should be able to “intercept” some machine instructions
 - Some kind of trap / exception / software interrupt must be generated
 - Not always possible (think about x86 ring 0)
- The CPU must provide some support for full virtualization
 - “More than supervisor” mode → hypervisor mode
 - Introduce two operating modes: “root mode” and “non-root mode”; non-root mode can only modify a shadow copy of the CPU privileged state
 - ...

OSs for Virtualizable Architectures

- Virtualizable ISA: how to use it?
 - VMM or hypervisor responsible for managing VMs and other resources
 - Re-invent an OS, or using an existing one?
- OS support for hypervisors
 - Hosted hypervisor
 - Dom0
 - ...
- Difference between a hypervisor and a μ -kernel???
 - Are we reinventing an old idea?

ParaVirtualization

- So, CPU virtualization can be easy and efficient
 - Provided that the ISA is virtualizable
 - Provided host OS support / hypervisor
- What about I/O devices?
 - Virtualizing real hardware can be complex and inefficient
 - Idea: device passthrough
 - Other possibility: paravirtualization
- Paravirtualization: the guest knows that it is running in a VM
 - Memory buffers can be (securely) shared between guest and host
 - ...

Different Kinds of VMMs/Hypervisors

- Software implementation of $\mathcal{M}_{\mathcal{L}}$ (hardware machine): executes on a **Host Machine** $\mathcal{M}h_{\mathcal{L}h}$
- $\mathcal{M}h_{\mathcal{L}h}$ is of course an abstract machine...
 - Is it the hardware machine?
 - Is it a higher-level abstract machine (implemented by kernel, OS, ...)?
- In other words: **where does the hypervisor run?**
 - Does it run directly on the hardware?
 - Does it run on an OS kernel?
 - Does it uses other services provided by a host OS?

Bare Metal Hypervisors

- The host machine $\mathcal{M}h_{\mathcal{L}h}$ is the hardware machine, without extensions
 - The hypervisor directly accesses the hardware
 - Loaded by some kind of bootloader, controls the execution of guest kernels/OSs
 - No functionalities provided by a “host kernel”
- Examples: Xen, but also Jailhouse (funny detail: Linux is the bootloader!)
- How to access the physical devices?
 - Must implement device drivers...
 - ...Or rely on a “special” guest OS that provides the drivers
 - Generally uses pass-through techniques

Hosted Hypervisors

- The host machine is the hardware machine + a host kernel (or even a host OS!)
 - The hypervisor can rely on functionalities provided by the host kernel
 - No need to re-implement memory management, drivers, CPU scheduler, ...
- Examples: KVM, VirtualBox, ...
- The hypervisor can be started only after the guest OS is booted
 - There generally is a kernel-level hypervisor + a user-level management tool (sometimes named VMM)

OS-Level Virtual Machines

- Do not virtualise the whole hardware
 - Only OS services are virtualised
 - Host kernel: virtualise its services to provide isolation among guests
- Container: isolated execution environment to encapsulate one or more processes/tasks
 - Sort of “chroot on steroids”
- Two aspects: resource control (scheduling) and visibility
 - Control/monitor how much resources a VM is using
 - Make sure that virtual resources of a VM are not visible in other VMs

More on “Containers”

- Banga and others, 99: “Resource Containers: A New Facility for Resource Management in Server Systems”
 - Operating system abstraction containing all the resources used by an application to achieve a particular independent activity
- Today, “container” == execution environment
 - Used to run a whole OS → VM (with OS-level virtualization)
 - Used to run a single application / micro-service

Linux Containers

- The Linux kernel does not directly provide the “container” abstraction
- Containers can be built based on lower-level mechanisms: *control groups* (`cgroups`) and *namespaces*
 - **namespaces**: isolate and virtualise system resources
 - **cgroups**: limit, control, or monitor resources used by groups of tasks
- Namespaces are concerned with resources’ visibility, cgroups are concerned with scheduling

Linux Namespaces

- Used to isolate and virtualise system resources
 - Processes executing in a namespace have the illusion to use a dedicated copy of the namespace resources
 - Processes in a namespace cannot use (or even see) resources outside of the namespace
- Processes in a network namespace only see network interfaces that are assigned to the namespace
 - Same for routing table, etc...
- Processes in a PID namespace only see processes from the same namespace
 - PIDs can be “private to the namespace”

Linux Control Groups

- Used to restrict (limit, control) or monitor the amount of resources used by “groups of processes”
 - Processes can be organized in groups, to control their accesses to resources
- Example: CPU control groups for scheduling
 - Limit the amount of CPU time that processes can use, etc...
- Similar cgroups for other resources
 - memory, IO, pids, network, ...

Building a Container

- Namespaces and control group give fine-grained control on processes and resources
 - Per-resource control groups and/or namespaces
 - Lower level abstractions respect to other OSs (for example, FreeBSD jails)
- More powerful than other mechanisms, but more difficult to use
- To build a container, it is necessary to:
 - Setup all the needed namespaces and control groups
 - Create a “disk image” for the container (directory containing the container’s fs)

Running in a Container

- Chroot to the container fs
 - Must contain the whole OS, or the libraries/files needed to execute the program to containerize
- Start init, or the program to containerize
 - Thanks to the PID namespace, it will have PID 1 in the container!
- Note: init can mount procfs or other pseudo-file systems
 - Namespaces allow to control the information exported in those pseudofilesystems!

Example: Networking in Containers

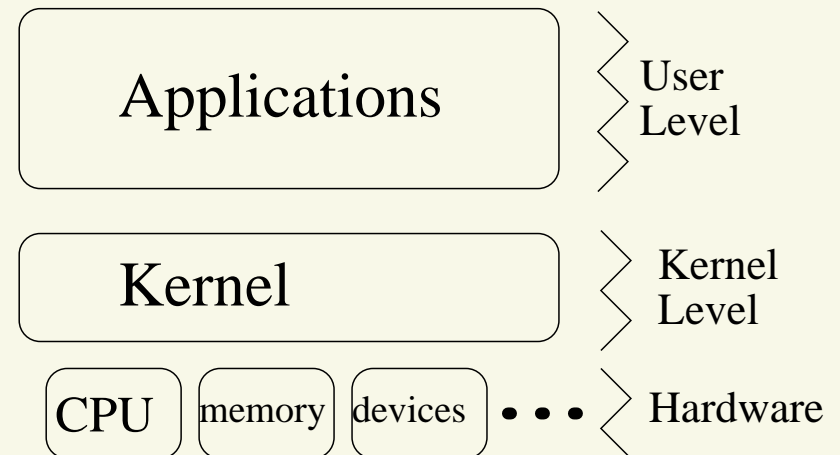
- Thanks to the network namespace, processes running in a container do not see the host's network interfaces
 - How to do networking, then?
- Create a *virtual ethernet pair*
 - Two virtual ethernet interfaces, connected point-to-point
 - Packets sent on one interface are received on the other, and vice-versa
- Associate one of the two virtual ethernet interfaces to the network namespace of the container
- Bind the other one to a software bridge

Virtual Operating Systems

- Instead of virtualizing kernel services, it is possible to virtualize the whole OS!
 - With containers, software implementation of an abstract machine $\mathcal{M}_{\mathcal{L}}$ corresponding to the OS kernel (understanding machine language + system calls)
 - Possible to implement an abstract machine $\mathcal{M}_{\mathcal{L}}$ understanding machine language + standard OS runtime (libc, ...)
- Possible to execute (for example) Windows applications on Linux (or vice-versa!)
 - How can this be implemented in practice?
 - Re-implement the system libraries...

The Kernel

- Kernel → OS component interacting with hardware
 - Runs in privileged mode (Kernel Space → KS)
 - User Level ⇔ Kernel Level switch through special CPU instructions (INT, TRAP, ...)
 - User Level invokes *system calls* or IPCs
- Kernel Responsibilities
 - Process management
 - Memory management
 - Device management
 - System Calls



System Libraries

- Applications generally don't invoke system calls directly
- They generally use *system libraries* (like glibc), which
 - Provide a more advanced user interface (example: `fopen()` vs `open()`)
 - Hide the US \Leftrightarrow KS switches
 - Provide some kind of stable ABI (application binary interface)

Static vs Shared Libraries - 1

- Libraries can be *static* or *dynamic*
 - `<libname>.a` **VS** `<libname>.so`
- Static libraries (`.a`)
 - Collections of object files (`.o`)
 - Application linked to a static library \Rightarrow the needed objects are included into the executable
 - Only needed to compile the application

Static vs Shared Libraries - 2

- Dynamic libraries (`.so`, shared objects)
 - Are not included in the executable
 - Application linked to a dynamic library \Rightarrow only the library symbols names are written in the executable
 - Actual linking is performed at loading time
 - `.so` files are needed to execute the application
- Linking static libraries produces larger executables...
- ...But these executables are “self contained”

OS ABI Virtualization

- A modified executable loader load some “special” system libraries instead of the standard ones
 - On Windows, when loading an ELF Linux executable dynamically link a “libc.so” implementing the Linux ABI on the windows kernel
 - On Linux, when loading a PE Windows executable dynamically link some special “win32.dll”/“win64.dll” implementing the Windows ABI on Linux
 - ...
- Of course, the devil is in the details (look at Wine!)
- What to do for statically linked applications / applications directly invoking syscalls?