Perfect Security

Ok, Not *Completely* Isolated...

The notion of **isolation** is useful for security, even if it doesn’t mean completely disconnected from the world

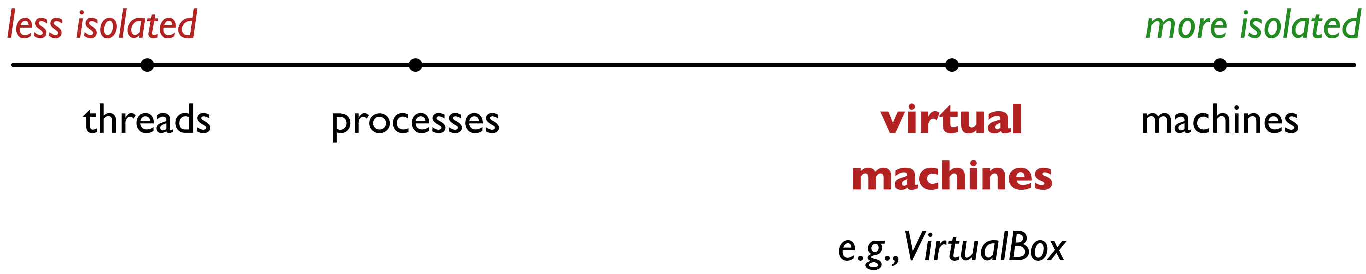
The **principle of least privilege** means that actors should have only the capabilities and connectivity that they need

- Implemented in part with access control
- Implemented in part with isolation

Isolation as a kind of capability: If two actors don’t share a thing, then misuse of the thing by one (whether malicious or accidental) can’t break a use of the thing by the other

Good for maintenance and deployment as well as security

Representing Tasks



less isolated

more isolated

threads

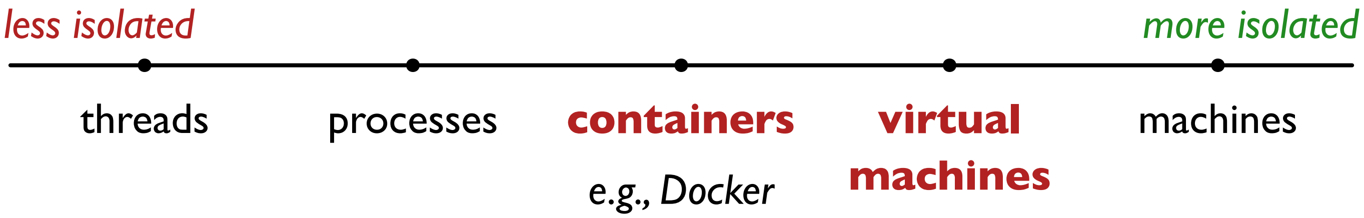
processes

virtual machines

e.g., VirtualBox

machines

Representing Tasks



less isolated

more isolated

threads

processes

containers

e.g., *Docker*

virtual machines

machines

Representing Tasks

less isolated ——— *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝️				
address space	🏙️				
file descriptors	🏙️				
filesystem	🏙️				
devices	🏙️				
kernel	🏙️				
processor	🏙️				

Threads can easily interfere with each other via shared objects

Representing Tasks

less isolated → *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝️	🏝️			🏝️
address space	🏙️	🏝️			🏝️
file descriptors	🏙️	🏝️/🏙️			🏝️
filesystem	🏙️	🏙️			🏝️
devices	🏙️	🏙️			🏝️
kernel	🏙️	🏙️			🏝️
processor	🏙️	🏙️			🏝️

Processes mostly have separate file descriptors, but can share them via `fork`

Representing Tasks

less isolated → *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝	🏝		🏝	🏝
address space	🏙	🏝		🏝	🏝
file descriptors	🏙	🏝/🏙		🏝	🏝
filesystem	🏙	🏙		🏝	🏝
devices	🏙	🏙		🏝	🏝
kernel	🏙	🏙		🏝	🏝
processor	🏙	🏙		🏙	🏝

Virtual machines pretend to be real machines, but run within some OS

Representing Tasks

less isolated ——— *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝	🏝	🏝	🏝	🏝
address space	🏙	🏝	🏝	🏝	🏝
file descriptors	🏙	🏝/🏙	🏝	🏝	🏝
filesystem	🏙	🏙	🏝/🏙		
devices	🏙	🏙	🏝/🏙	🏝	🏝
kernel	🏙	🏙	🏙	🏝	🏝
processor	🏙	🏙	🏙	🏙	🏝

Containers mostly have their own filesystems, but can be made to share with the host

Representing Tasks

less isolated — *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝️	🏝️	🏝️	🏝️	🏝️
address space	🏙️	🏝️	🏝️	🏝️	🏝️
file descriptors	🏙️	🏝️/🏙️	🏝️	🏝️	🏝️
filesystem	🏙️	🏙️	🏝️/🏙️		
devices	🏙️	🏙️	🏝️/🏙️		
kernel	🏙️	🏙️	🏙️		
processor	🏙️	🏙️	🏙️	🏙️	🏝️

Containers use host devices, but virtualized, so that access can be limited and adressing can be separate

Representing Tasks

less isolated → *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝️	🏝️	🏝️	🏝️	🏝️
address space	🏙️	🏝️	🏝️	🏝️	🏝️
file descriptors			🏝️	🏝️	🏝️
filesystem			🏝️/🏙️	🏝️	🏝️
devices			🏝️/🏙️	🏝️	🏝️
kernel	🏙️	🏙️	🏙️	🏝️	🏝️
processor	🏙️	🏙️	🏙️	🏙️	🏝️

Unlike a virtual machine, a container uses the same kernel as its host—but the rest of the OS in a container can be different, because the filesystem is separate

Representing Tasks

less isolated *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝️	🏝️	🏝️	🏝️	🏝️
address space	🏙️	🏝️	🏝️	🏝️	🏝️
file descriptors	🏙️	🏝️/🏙️	🏝️	🏝️	🏝️
filesystem	🏙️	🏙️	🏝️/🏙️	🏝️	🏝️
devices	🏙️	🏙️	🏝️/🏙️	🏝️	🏝️
kernel	🏙️	🏙️	🏙️	🏝️	🏝️
processor	🏙️	🏙️	🏙️	🏙️	🏝️

Representing Tasks

Especially helpful for deployment

less isolated → *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝	🏝	🏝	🏝	🏝
address space	🏙	🏝	🏝	🏝	🏝
file descriptors	🏙	🏝/🏙	🏝	🏝	🏝
filesystem	🏙	🏙	🏝/🏙	🏝	🏝
devices	🏙	🏙	🏝/🏙	🏝	🏝
kernel	🏙	🏙	🏙	🏝	🏝
processor	🏙	🏙	🏙	🏙	🏝

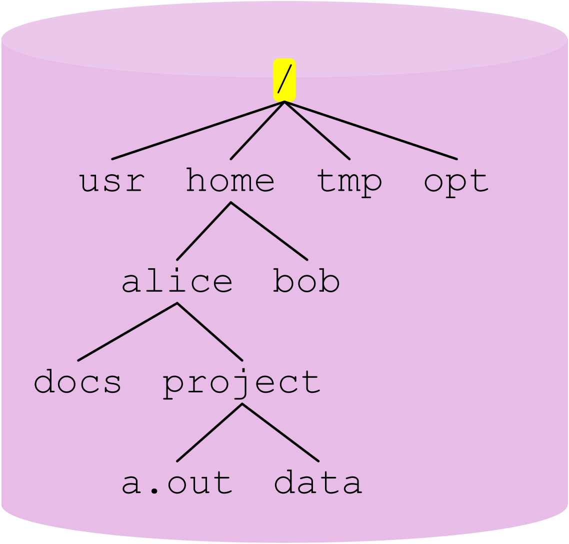
Representing Tasks

More points in between: `chroot` and namespaces

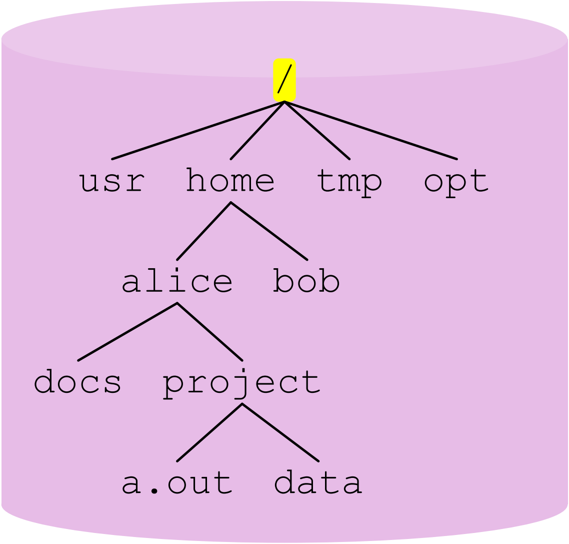
less isolated *more isolated*

	threads	processes	**containers**	**virtual machines**	machines
stack	🏝	🏝	🏝	🏝	🏝
address space	🏙	🏝	🏝	🏝	🏝
file descriptors	🏙	🏝/🏙	🏝	🏝	🏝
filesystem	🏙	🏙	🏝/🏙	🏝	🏝
devices	🏙	🏙	🏝/🏙	🏝	🏝
kernel	🏙	🏙	🏙	🏝	🏝
processor	🏙	🏙	🏙	🏙	🏝

Toward Containers: `chroot`

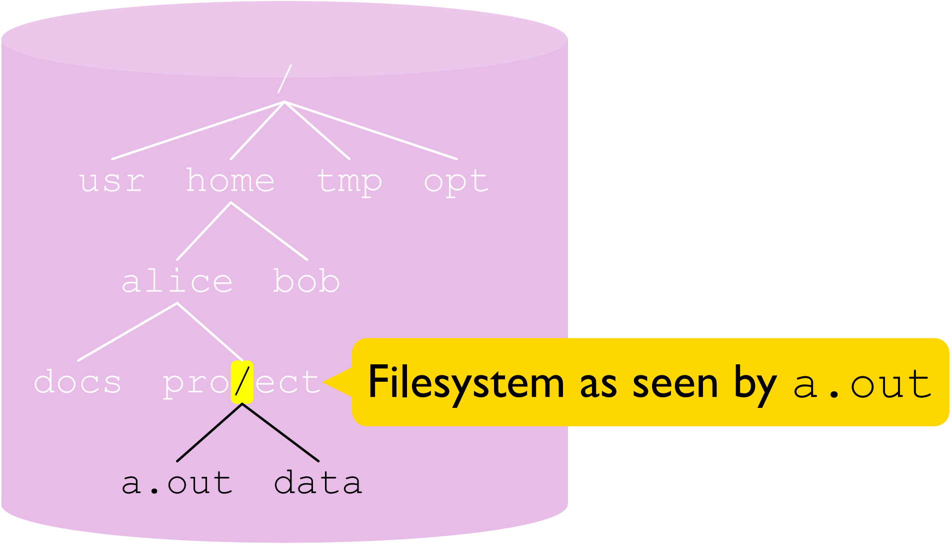


Toward Containers: `chroot`



```
$ chroot /home/alice/project /a.out
```

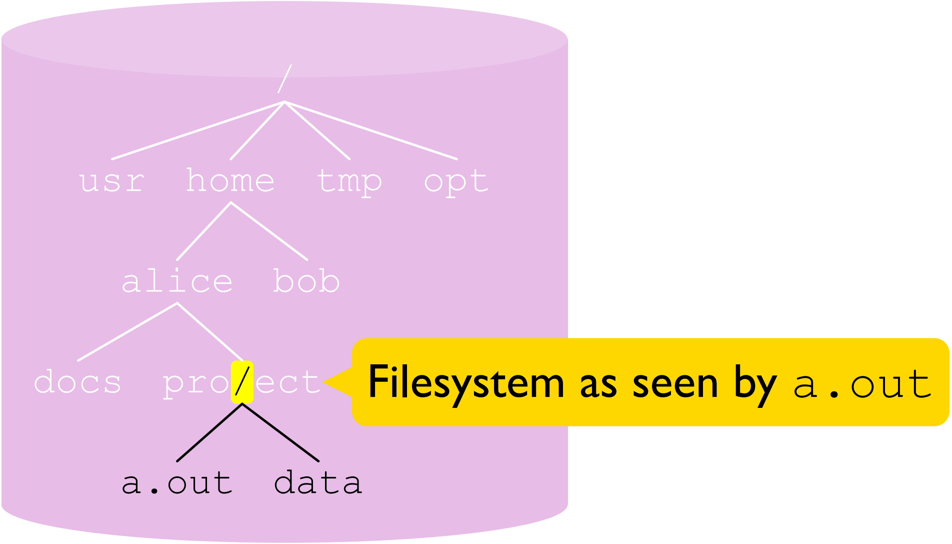
Toward Containers: `chroot`



Filesystem as seen by `a.out`

```
$ chroot /home/alice/project /a.out
```

Toward Containers: `chroot`

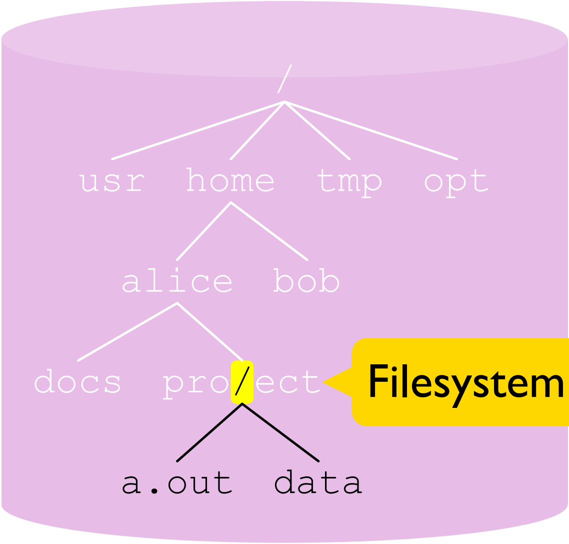


Filesystem as seen by `a.out`

```
$ chroot /home/alice/project /a.out
```

`chroot` is tricky to use directly, because executables need shared libraries that are provided by the operating system

Toward Containers: `chroot`



Filesystem as seen by `a.out`

```
$ chroot /home/alice/project /a.out
```

Isolates only the filesystem
— and not, for example, process IDs

Linux Namespaces

A **namespace** in Linux is a generalization of `chroot`

- filesystem
- process IDs
- network interfaces (and therefore addresses)
- interprocess communication
- hostname
- users and groups
- time

Related concept: a **sandbox** is the same kind of functionality more generally, sometimes based on runtime support in a programming language

A **container** system is a manageable API for namespaces

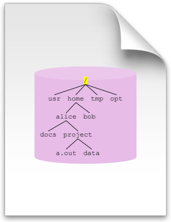
Docker

and similar container systems

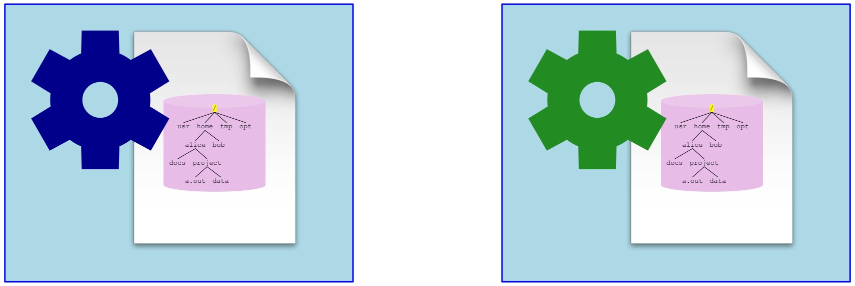
and similar in other OSes

Docker builds on Linux namespaces:

- An **image** contains a filesystem, normally with a copy of an OS



- A **container** starts with a copy of an image plus a configuration



Docker

Typical uses:

- different OS distribution (capatible with host kernel)
- different set of installed libraries
- sandboxing to restrict network access, limit computation time, etc.
- reproducible builds

Docker

```
$ docker image ls
REPOSITORY     TAG         IMAGE ID       CREATED          SIZE
debian         testing     0713af5d6328   8 months ago     117MB
ubuntu         18.04       8d5df41c547b   20 months ago    63.1MB
ubuntu         20.04       ba6acccedd29   2 years ago      72.8MB
archlinux      latest      481b70173ad4   2 years ago      387MB
racket/racket  latest      1ca0bea7d02d   4 months ago     244MB
pkg-build      latest      c6a6792dec0a   2 years ago      1.96GB
```

```
$ docker container ls -a
CONTAINER ID   IMAGE            COMMAND       CREATED        ....
8f476a83a297   debian:testing   "bash"        8 months ago   ....
7052d25067bd   racket/racket    "/bin/bash"   2 years ago    ....
d88cb393d42f   racket/racket    "/bin/bash"   2 years ago    ....
```

Dockerfiles

Docker images are created by **Dockerfile** scripts

```
FROM debian:stable-slim

RUN apt-get update -y \
    && apt-get install -y --no-install-recommends ca-certificates curl sqlite3 \
    && apt-get clean

RUN curl --retry 5 -Ls "${RACKET_INSTALLER_URL}" > racket-install.sh \
    && echo "yes\n1\n" | sh racket-install.sh --create-dir --unix-style --dest /usr/ \
    && rm racket-install.sh

ENV SSL_CERT_FILE="/etc/ssl/certs/ca-certificates.crt"
ENV SSL_CERT_DIR="/etc/ssl/certs"

RUN raco setup
RUN raco pkg config --set catalogs \
    "https://download.racket-lang.org/releases/${RACKET_VERSION}/catalog/" \
    "https://pkg-build.racket-lang.org/server/built/catalog/" \
    "https://pkgs.racket-lang.org" \
    "https://planet-compats.racket-lang.org"
```

Fetched from DockerHub

Smart sharing of data among images and containers makes them relatively lightweight

Creating Docker Containers

Create and start a container with `docker run` *image*

```
$ docker run -it debian:testing
```

Creating Docker Containers

Create and start a container with `docker run` *image*

```
$ docker run -it debian:testing
```

Start an existing container with `docker start` *container_id*

```
$ docker start -ia 8f476a83a297
```

Creating Docker Containers

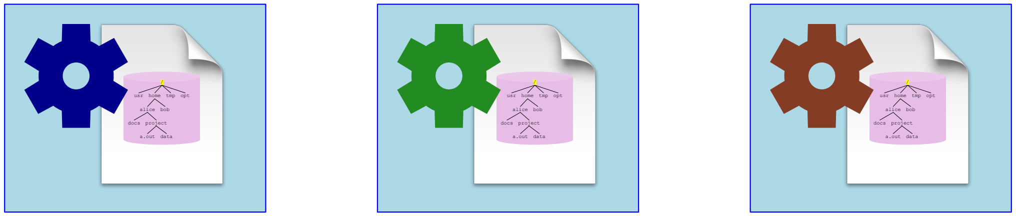
Create and start a container with `docker run` *image*

```
$ docker run -it debian:testing
```

Start an existing container with `docker start` *container_id*

```
$ docker start -ia 8f476a83a297
```

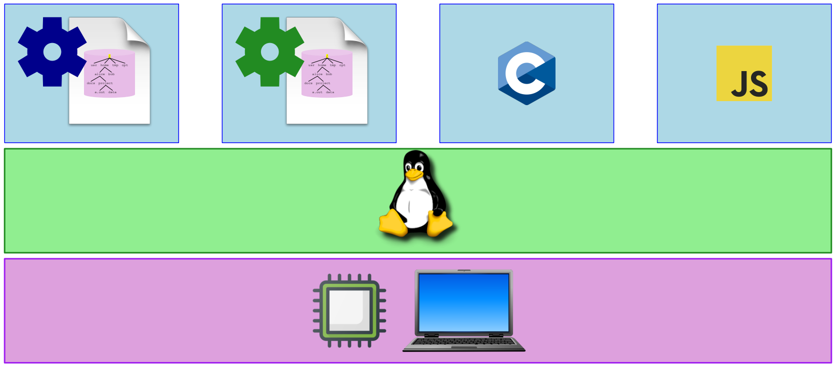
Different containers from the same image have separate filesystem state



Containers and Isolation

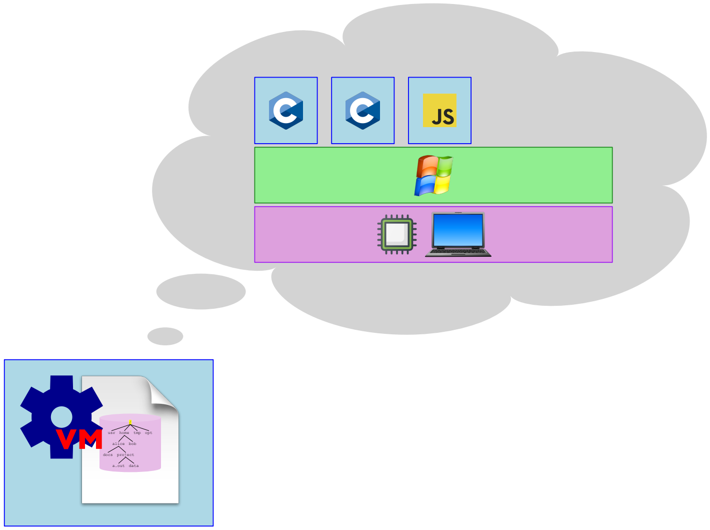
A container can be well isolated from its environment, but it still uses the same kernel as the host operating system

A kernel bug could allow an exploit to escape a container



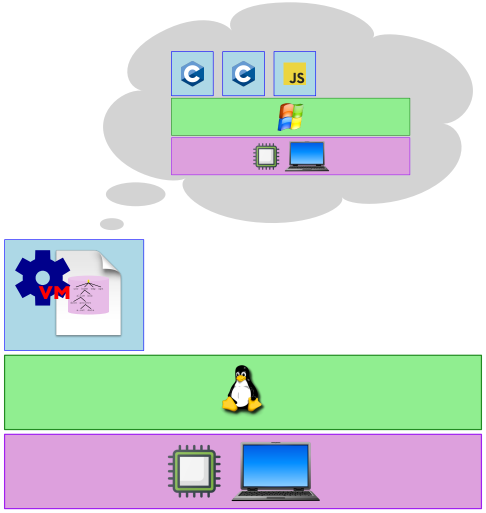
Virtual Machines

A **virtual machine (VM)** abstracts hardware instead of abstracting an operating system



Virtual Machines

A **virtual machine (VM)** abstracts hardware instead of abstracting an operating system



Kernel in a VM can be unrelated to the host OS running the VM

Machine's interface is even simpler than kernel's interface

Virtual Machines

Two kinds of virtual machines

Docker on macOS uses a VM to run Linux to run containers, and it can use QEMU

Emulation uses an interpreter machine code

The emulated processor can be unrelated to host processor

example: QEMU

Virtualization uses hardware to interpret directly

Mostly just intercept system calls, must be the same processor

example: VirtualBox

Virtual Machines

Two kinds of virtual machines

Emulation uses an interpreter machine code

The emulated processor can be unrelated to host processor

example: QEMU

Virtualization uses hardware to interpret directly

Mostly just intercept system calls, must be the same processor

example: VirtualBox

Direct device access is an issue, so recent processors help by offering specific virtualization support, such as **VT-x**

Virtual Machines

Two kinds of virtual machines

Emulation uses an interpreter machine code

The emulated processor can be unrelated to host processor

example: QEMU

Virtualization uses hardware to interpr

... but this is a hardware resource, so it should be managed by the kernel

Mostly just intercept system calls, must be the same proce sor

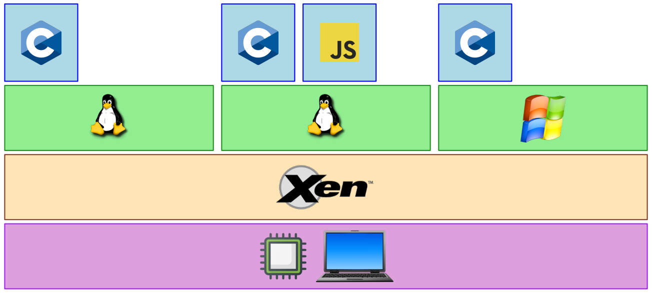
example: VirtualBox

Direct device access is an issue, so recent processors help by offering specific virtualization support, such as **VT-x**

Hypervisor

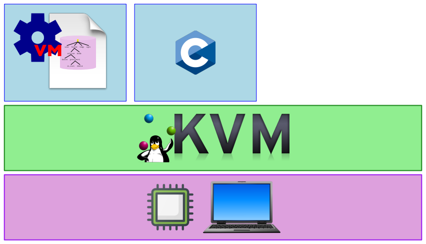
A kernel supervises programs; a **hypervisor** supervises kernels

Can be between the hardware and OSes, like **Xen**:



Either form of hypervisor may take advantage of hardware support for virtualization

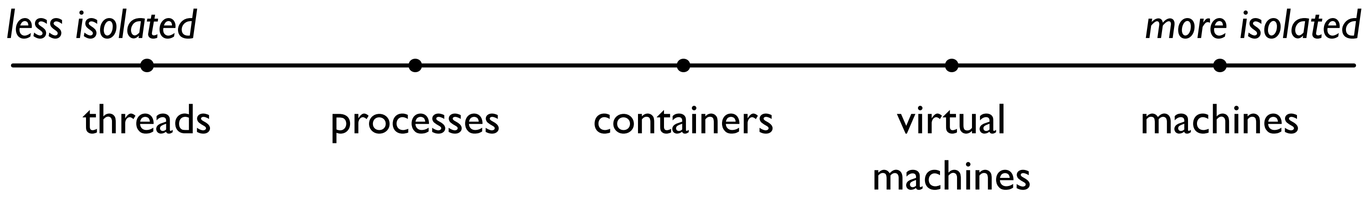
Can be capable OS, like Linux with **KVM**:



This is the main technology behind **cloud services**

Mainframes have been doing this since the 1960s

Tradeoffs



less isolated

more isolated

threads

processes

containers

virtual machines

machines

⇐ faster and easier

slower and more secure ⇒

Threads sharing data is very fast, and
and starting a thread is easy, but
any thread failure also takes down other threads

Machines sharing data is slow, and
maintaining machines is difficult, but
machines are completely autonomous

Containers a great compromise for many purposes

Summary

Isolation is good for software architecture, maintenance, and security

Containers and **virtual machines** provide useful degrees of isolation in between mere processes and completely separate machines

Any layer of a system can be virtualized, and that creates many possibilities to trade isolation, convenience, and performance