Aalto University

Platform Security: Lecture 1: Sandboxing

You will be learning:

Processes

- Virtual memory

Containers

- Docker
- Linux namespacing
- Overlay filesystems

Virtual Machines

- Virtualisation techniques
- DMA and IOMMUs
- Memory encryption
- Application virtual machines

System call filtering

Isolation

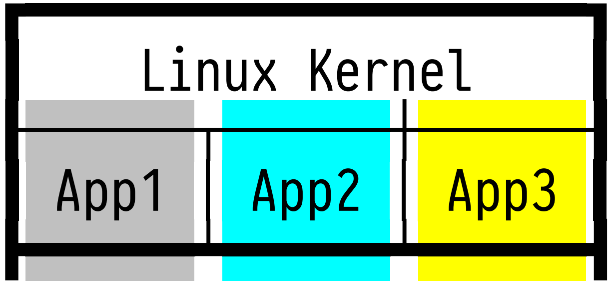
Problem: Developers make mistakes

- Real applications will contain bugs and vulnerabilities
- How can we limit the damage caused by faulty code?

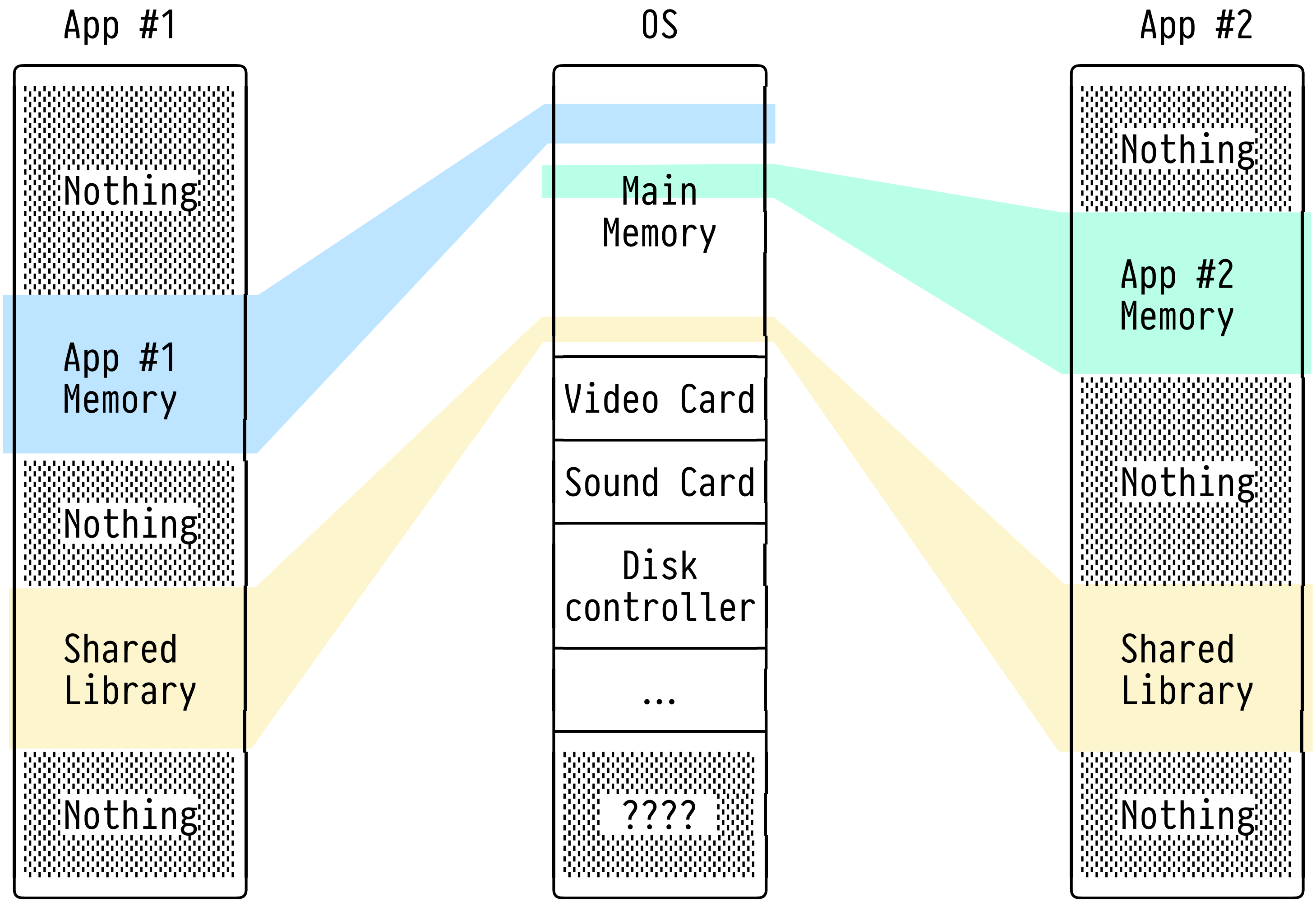
Solution: Insert isolation barriers

First approach: put each application process into a separate address space

- Processes can't write to regions of memory not mapped into their address space



Applications each have their own view of memory



The OS controls the current view by manipulating page tables

- Page = unit of memory by the memory management unit (usually 4KiB)

The page tables map virtual addresses to physical addresses

- How? See next slide

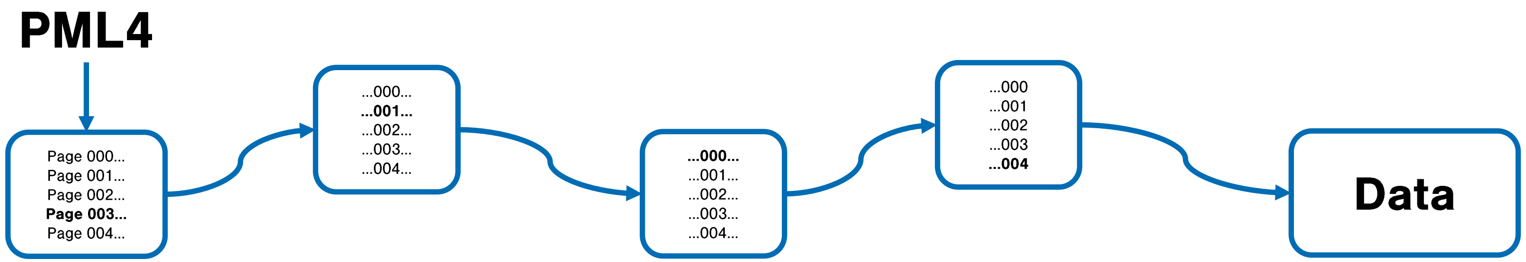
Mappings include access control: what can current process do with each page?

- Read
- Write
- Execute

Virtual addresses are translated to physical address using a page table walker

Example: Read address 0x000064a32b15e660, assuming 4kiB pages on x86-64

1. Top nine bits (011001001) point to an entry in the fourth-level page table
   - Each entry in the top-level page table contains the location of the third-level table
2. Next nine bits (010001100) point to an entry in the third-level page table
3. Next nine bits (101011000) point to an entry in the second-level page table
4. Next nine bits (101011110) point to an entry in the first-level page table
   - Each entry in the first-level page table contains the page number of the data in question
5. Lowest twelve bits (0x660) point to an offset in data page



Virtual memory can be used for sandboxing

- Switching the top-level page table changes software’s view of memory

This approach provides separation between processes

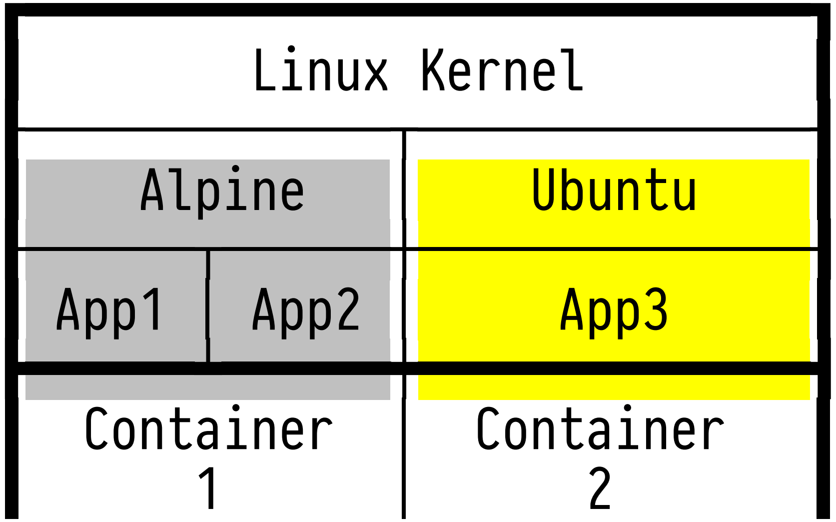
- Each process sees only a small part of memory
- Shared libraries can be mapped read-only into multiple processes
- Processes can communicate via shared pages

See an Operating Systems course for more information; many terrifying details

- Caching
- Swapping (keeping unused memory on disk)
- Other techniques to keep performance reasonable

Containers

Multiple operating systems share one kernel



High-level system to create and manage containers

```
Host $ docker pull ubuntu:bionic
Host $ docker run –it ubuntu:bionic
root@b0f077d3cee1:/#
```

Docker images

Containers have their own filesystems

- Host may have a different operating system!

Docker approach: Dockerfile describes how to build a container

- Which base OS image?
- How to install software?
- What to run at startup?

Example: Ubuntu environment with compiler, linker, etc.

```
FROM ubuntu:bionic
RUN apt-get –yyq update && apt-get –yyq install build-essential
CMD gcc –v && /bin/bash
```

Docker images

Each command in the Dockerfile that changes filesystem creates a new layer

```
FROM ubuntu:latest
```
Which OS image to use

```
RUN apt-get update && apt-get install -yyq build-essential
```
Install software packages

```
ADD ./src /src
```
Copy source code into the container

```
RUN cd /src/ && ./configure --prefix=/usr && make && make install && rm -rf /src
```
Build & install the application

Application binaries
Application source code
build-essential package files
ubuntu:latest

How do Linux containers work?

Linux supports “namespaces” for processes

- man page: namespaces(7)

Each namespace has its own copy of global resources

- Control groups
- IPC
- Network
- Mount points
- Process ID namespace
- Clocks
- User and group IDs
- Hostname

Background: Linux system call interface

Mechanism for applications to call into the kernel

- Used by C library to implement e.g. printf, fwrite, etc.

System calls (syscalls) often denoted syscall_name(2)

- Documented in section 2 of the “manual”
- Access documentation with

```
$ man 2 syscall_name
```

Most kernel objects represented by an integer id

- `int   file_descriptor = open(“/path/to/file”, O_RDONLY);`
- `pid_t process_id      = getpid();`
- …

Background: Linux system call interface

Examples:

- Print “Hello, World”

```
const char hello[] = “Hello, World!\n”;
write(STDOUT_FILENO, hello, sizeof(hello));
```

- Create a new process, and wait for it to finish

```
pid_t child_pid = clone(child_main, child_stack_top,
                        SIGCHLD, NULL);
if (child_pid > 0) {
    waitpid(child_pid, NULL, 0);
}
```

Creating a namespace

Namespaces are created using clone(2) and unshare(2)

Create a new process For the current process

Flags are in the clone(2) man page for those interested

Examples

- CLONE_NEWNET
- CLONE_NEWPID
- CLONE_NEWNS
- ...

Filesystems on Linux

Filesystems need to be mounted

- Connection directory ↔ filesystem

Run mount to see current mounts

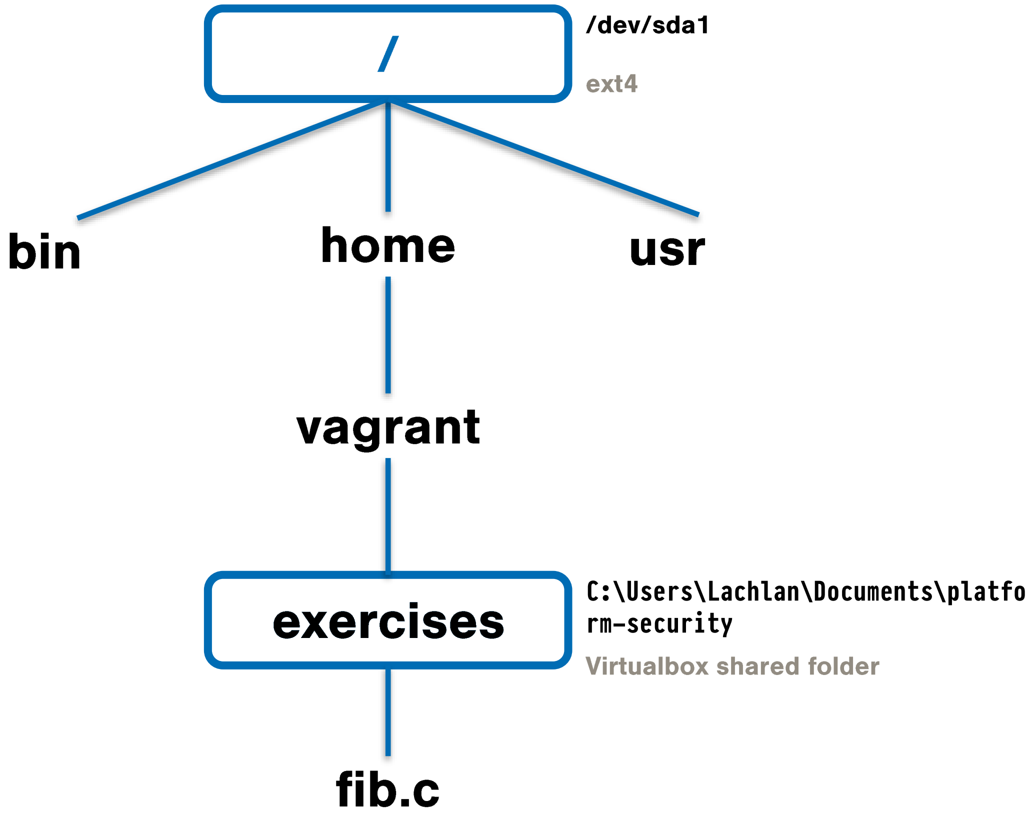
```
$ mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,noexec,relatime,size=484972k,nr_inodes=121243,m
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=00
tmpfs on /run type tmpfs (rw,nosuid,nodev,noexec,relatime,size=100460k,mode=755)
/dev/sda1 on / type ext4 (rw,relatime)

home_vagrant_exercises on /home/vagrant/exercises type vboxsf (rw,nodev,relatime,iocha
tmpfs on /run/user/1000 type tmpfs (rw,nosuid,nodev,relatime,size=100456k,mode=700,ui
```

During boot, root filesystem mounted at /

~~Virtualbox shared folders use vboxsf filesystem~~

- *Don't worry about this yet*

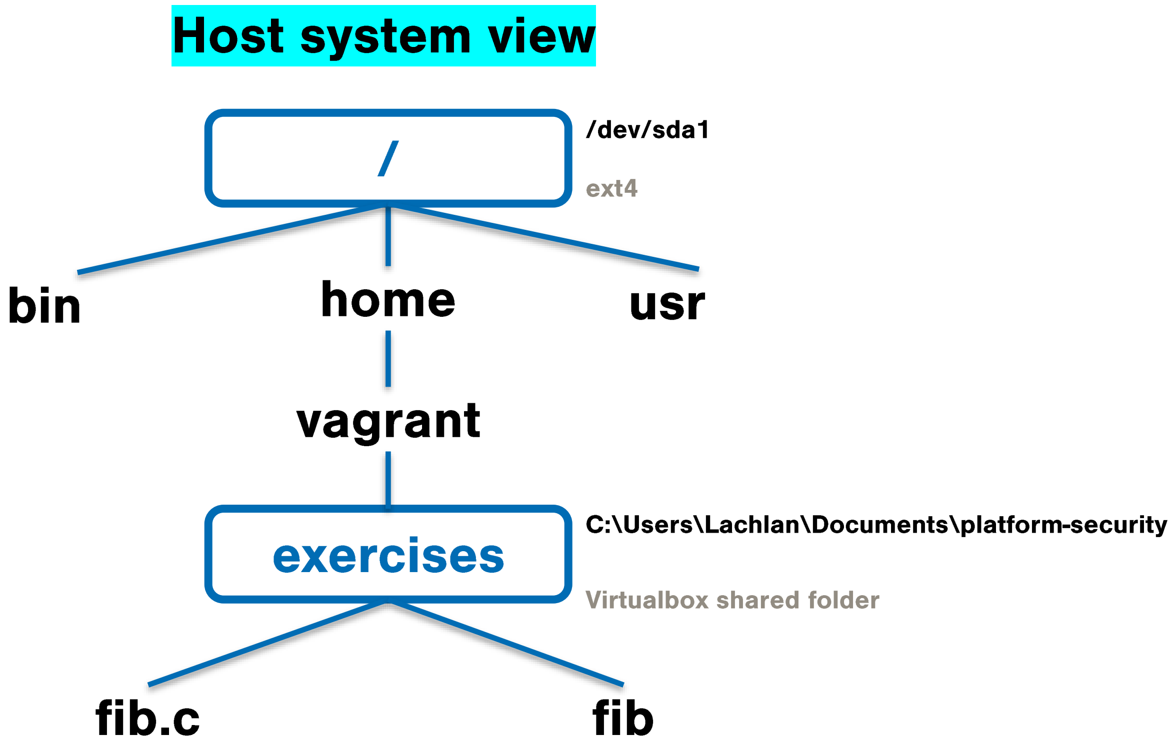


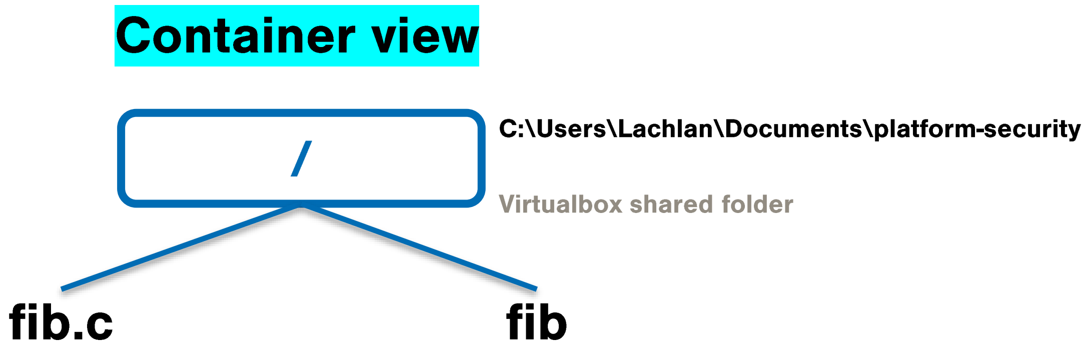
Mount namespaces

Processes in separate mount namespaces have separate mount points

How? Pass CLONE_NEWNS flag to clone(2) or unshare(2)

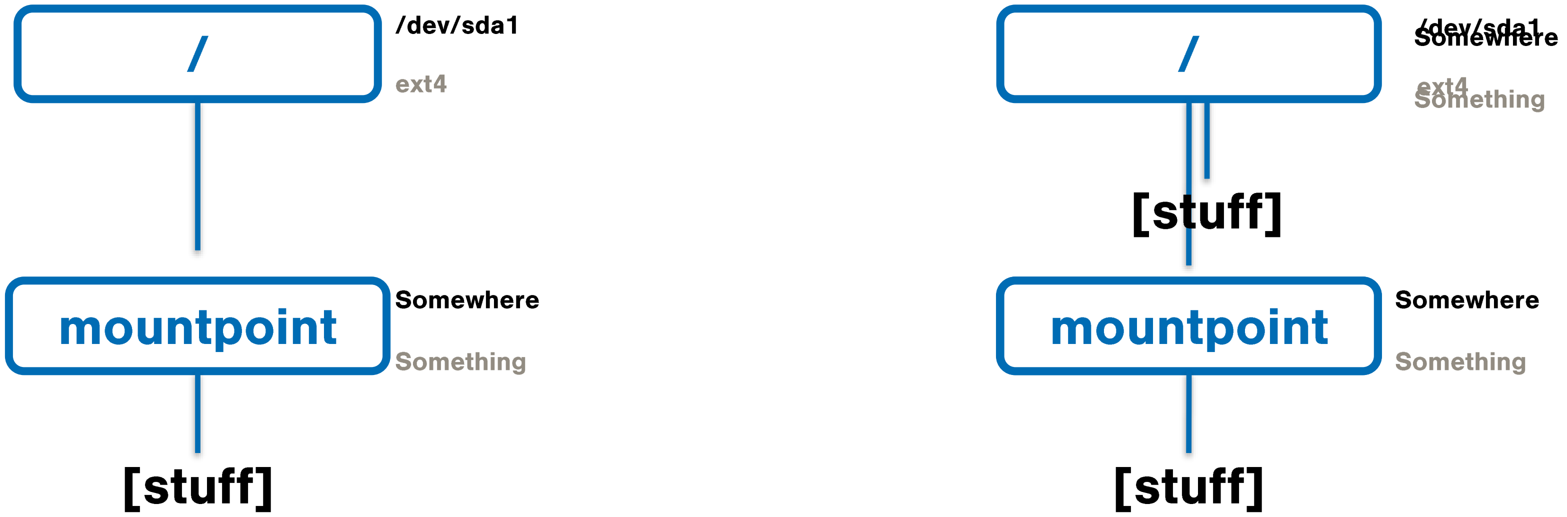
Result: Containerized process can see different filesystem layout





Setting this all up

1. **Create a directory** (“mountpoint”)
2. **Mount the container’s filesystem there**
3. **Use clone(2) or unshare(2) to create a new mount namespace**
4. **Use pivot_root(2) to make** mountpoint **the new root**
5. **Unmount the old root**



See [libcontainer/rootfs_linux.go]() for the code Docker uses

Image structure

Container filesystems share a lot of data

- Operating system: shared with many containers
- Applications: shared with instances of the same container

How can we share this between containers?

Solution: Layers

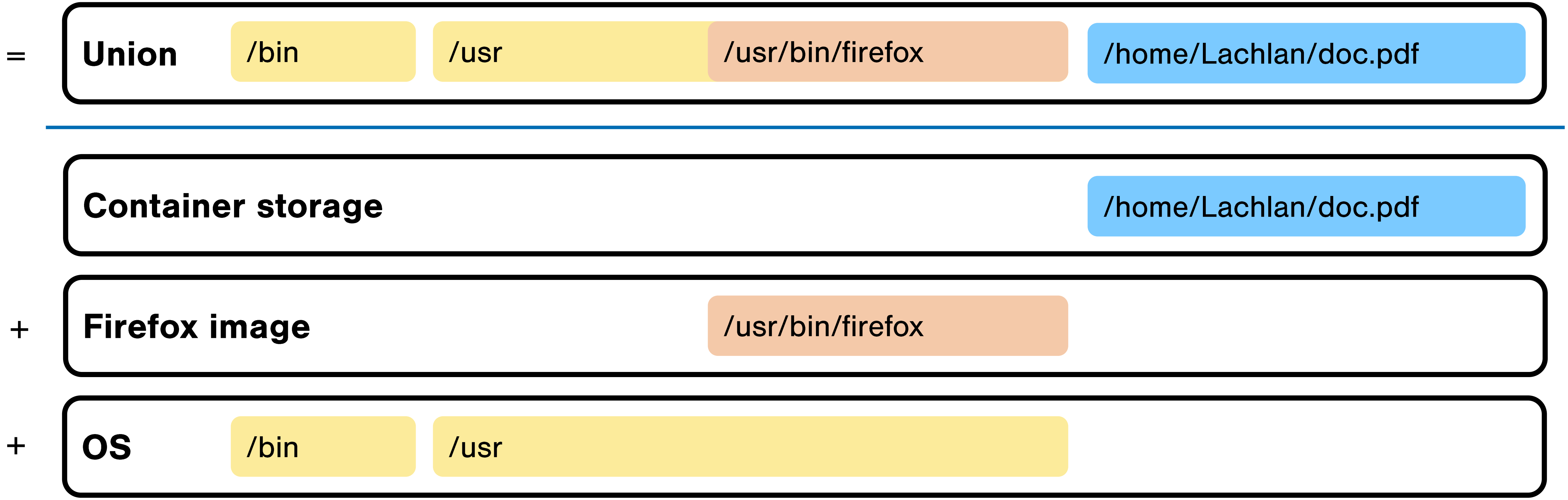
- Each step in building an image creates a new layer
- For each layer, store only changed files

Container filesystem = OS + Step1 + ... + StepN + Data

Overlay filesystems

Combine several directories together

- Upper (one): Added/modified files stored here
- Lower (many): Read-only



PID namespaces

PID namespaces have separate lists of processes

How? Pass CLONE_NEWPID flag to clone(2) or unshare(2)

Result: Container processes can’t see/signal host processes

```
$ ps xh | wc -l
142
$ docker run -it ubuntu:bionic sh -c ‘ps xh | wc -l’
3
```

Network namespaces

Prevents direct access to physical network interfaces

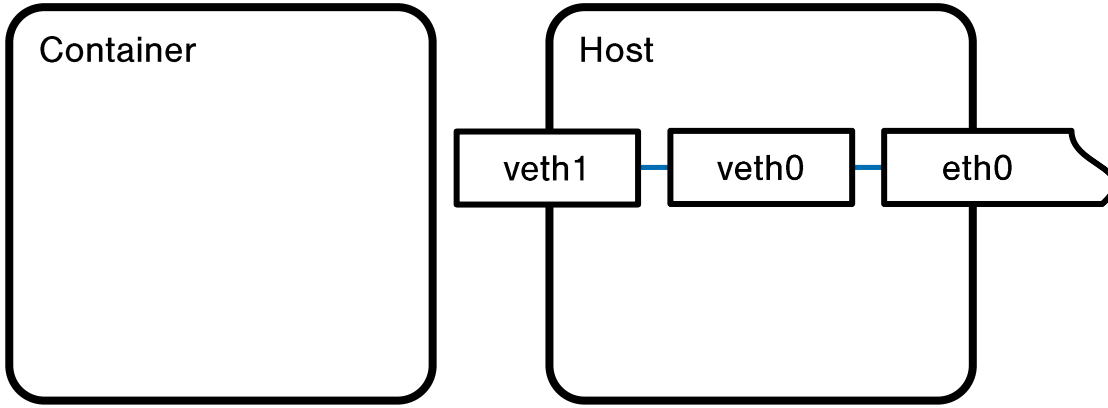
╰ Network interface can be in just one network namespace

How? Pass CLONE_NEWNET flag to clone(2) or unshare(2)

Result: All network interfaces disappear.

For container network access:

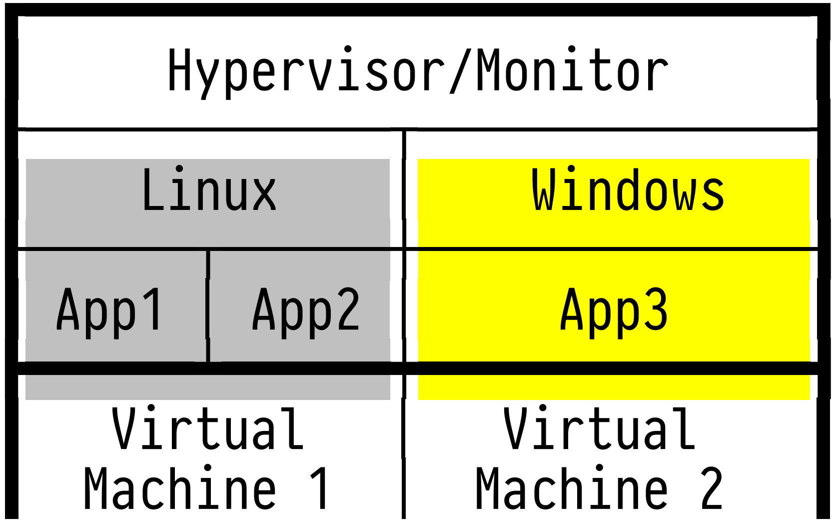
1. Create virtual ethernet device
2. Move one end into container
3. Bridge host end to the world



Other things are possible (e.g. NAT)

Virtual machines

Multiple kernels running on one physical machine



Why and what?

Several purposes:

- Run software from other OSs (e.g. Linux on Windows)
- Provide security boundary between applications
- Separate applications from hardware (e.g. cloud computing)

Virtual machine monitor goals:

- Equivalence...........Virtual environment looks like real hardware
- Efficiency................Almost all instructions run “natively”
- Resource control..VMM controls all hardware resources

Popek & Goldberg, “Formal requirements for virtualizable third generation architectures” (1974)

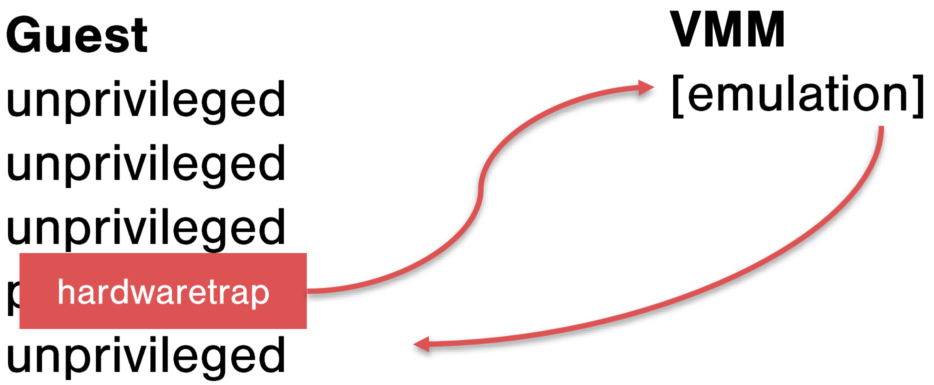
Trap-and-Trace

Virtualized software should run without modification

⟶ Virtual hardware should look like real hardware

Fundamental goal: run privileged software in unprivileged mode

- Privileged instructions “trapped” by hardware
- Emulated by Virtual Machine Monitor, “tracing” virtual state



Dynamic recompilation

Challenge: x86 can't trap all privileged state interactions

Examples:

- Privilege level stored in unprivileged register
- Some instructions behave differently for privileged mode

One solution: replace these instructions with jump to emulator

⟶ Used by early x86 virtual machine managers (e.g. VMWare)

Guest	**VMM**
unprivileged	[emulation]
unprivileged	
unprivileged	
privileged	
unprivileged	

Adams & Agesen, “A comparison of Software and Hardware Techniques for x86 Virtualization” (2006)

Paravirtualisation

Another solution: Sacrifice some equivalence for efficiency

⟶ Guest is modified to interact with VMM without traps

Examples:

- Xen (VMM calls needed for all privileged functionality)
- Most desktop VMMs (display, mouse, shared folder drivers)

x86 Hardware extensions

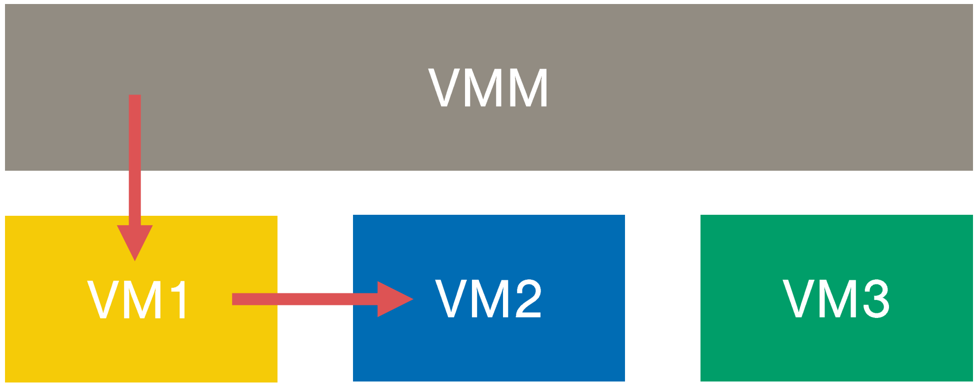
x86 processor vendors extended hardware to allow trap-and-trace

- Intel: VT-x (2005)
- AMD: AMD-V (2006)

Dynamic recompilation still useful

⟶ Jumping is sometimes faster than trapping

Security issues

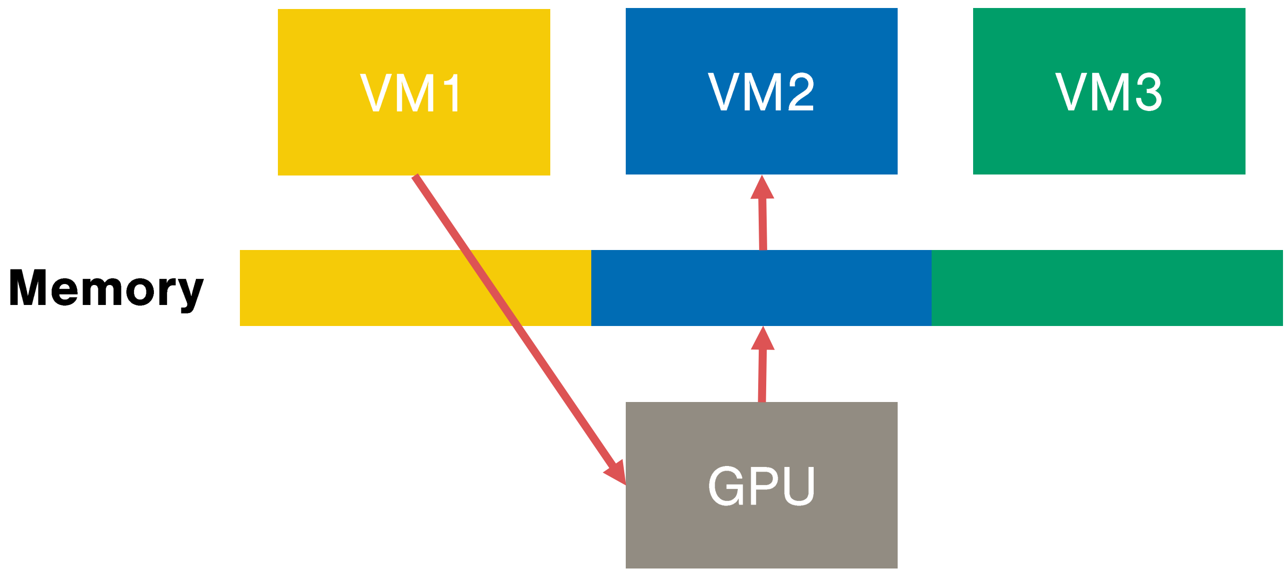


Direct memory access

Devices communicate with CPU by direct memory access (DMA)

⟶ Devices can write directly to main memory

Challenge: VMs can ask a device to overwrite another VM's memory

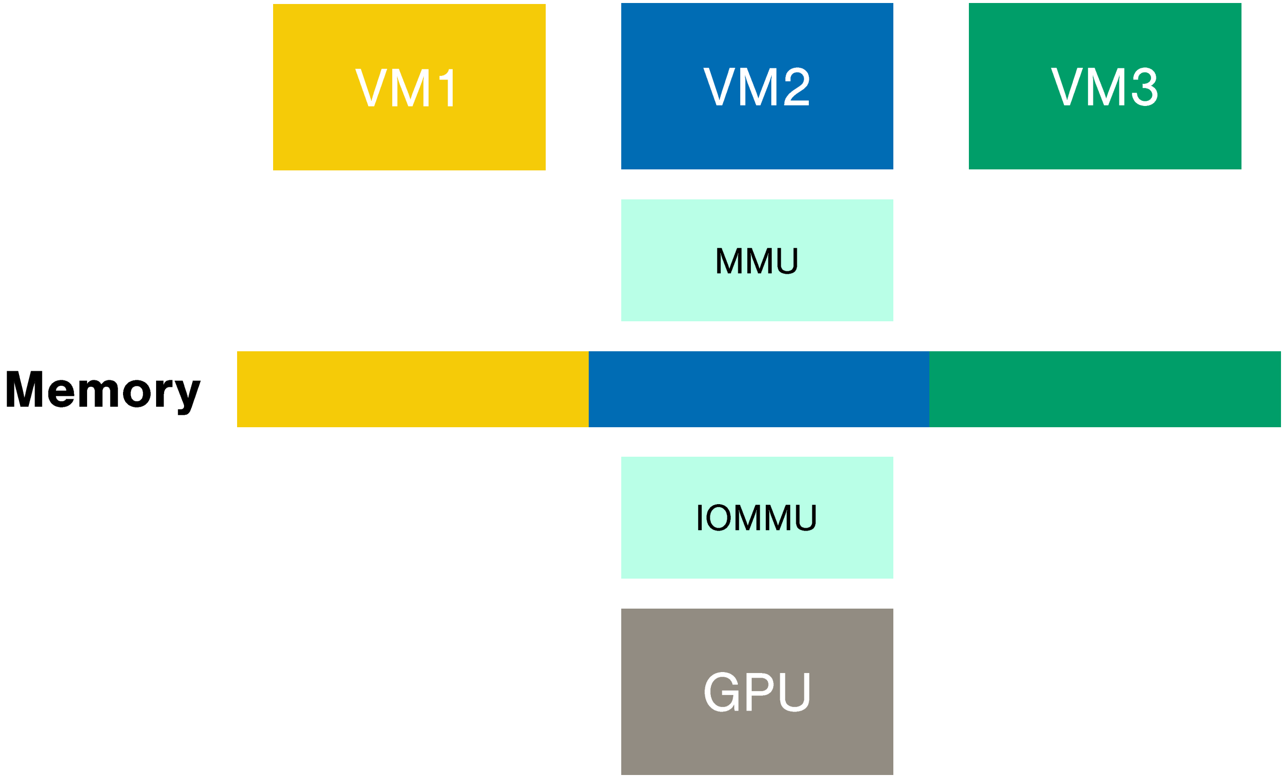


Direct memory access

Solution: add an I/O Memory Management Unit (IOMMU)

- e.g. Intel VT-d

Applies access control to DMA



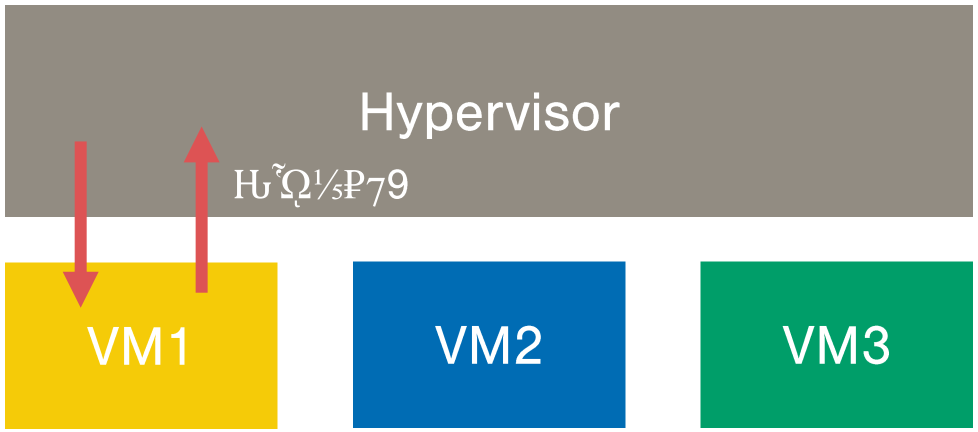
Memory encryption

Challenge: VMM can read virtual machine memory

Solution: Transparent memory encryption/authentication

- e.g. AMD SEV + SEV-SNP

Result: VMM can't read virtual machine memory



Virtual machines don't need to emulate a "normal" machine

Java Virtual Machine (JVM)

- Runs Java bytecode compiled from Java/Scala/Kotlin/...
- *Slogan: "Write Once, Run Anywhere"*
  - *Alternative view: "Write Once, Debug Everywhere"*
- Instruction set knows about classes, methods, etc.

WebAssembly (Wasm)

- Instruction format for C/C++/Rust/etc. code
- Originally designed to run in web browsers
- Designed to run untrusted code: strong sandboxing requirements

Example: WebAssembly

Applications compiled into WebAssembly modules

Modules have (non-exhaustive)

- An isolated memory space
- Exports: functionality exposed to the VM (e.g. program entry point)
- Imports: functionality requested from the VM (e.g. storage operations)

WebAssembly instruction set is different from underlying CPU

- Requires dynamic recompilation (example: WasmTime)
- *or an interpreter: simpler but slower* (example: Wasmi)

System call filtering

Applications call into the kernel using a system call instruction

- 64-bit x86: SYSCALL instruction
- Others: see syscall(2) manpage

Linux seccomp restricts system calls on a per-thread basis

- Opt-in: filters applied to application by itself

Example:

```
// Enable seccomp
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
// Application can now only read, write, _exit, sigreturn
```

Advanced system call filtering

Filters can be specified using Berkeley Packet Filter (BPF)

- Safe bytecode language, originally for network filters
- Adapted by seccomp to filter syscalls

Commonly used for sandboxing

- Docker
- Chrome on Linux

Safety requires many limitations in the filter language

- No pointer dereferencing (so can’t use paths, etc.)
- Generally not Turing-complete

Runs very early in the system call process

- Very little code that can contain vulnerabilities

Did you learn:

Processes

- Virtual memory

Containers

- Docker
- Linux namespacing
- Overlay filesystems

Virtual Machines

- Virtualisation techniques
- DMA and IOMMUs
- Memory encryption
- Application virtual machines

System call filtering