OS Containers – Introduction: the building blocks

Renaud Lachaize

Univ. Grenoble Alpes

M2 MoSIG & Phelma-SEOC

December 2025

Acknowledgments

Many parts of this lecture are heavily inspired by:

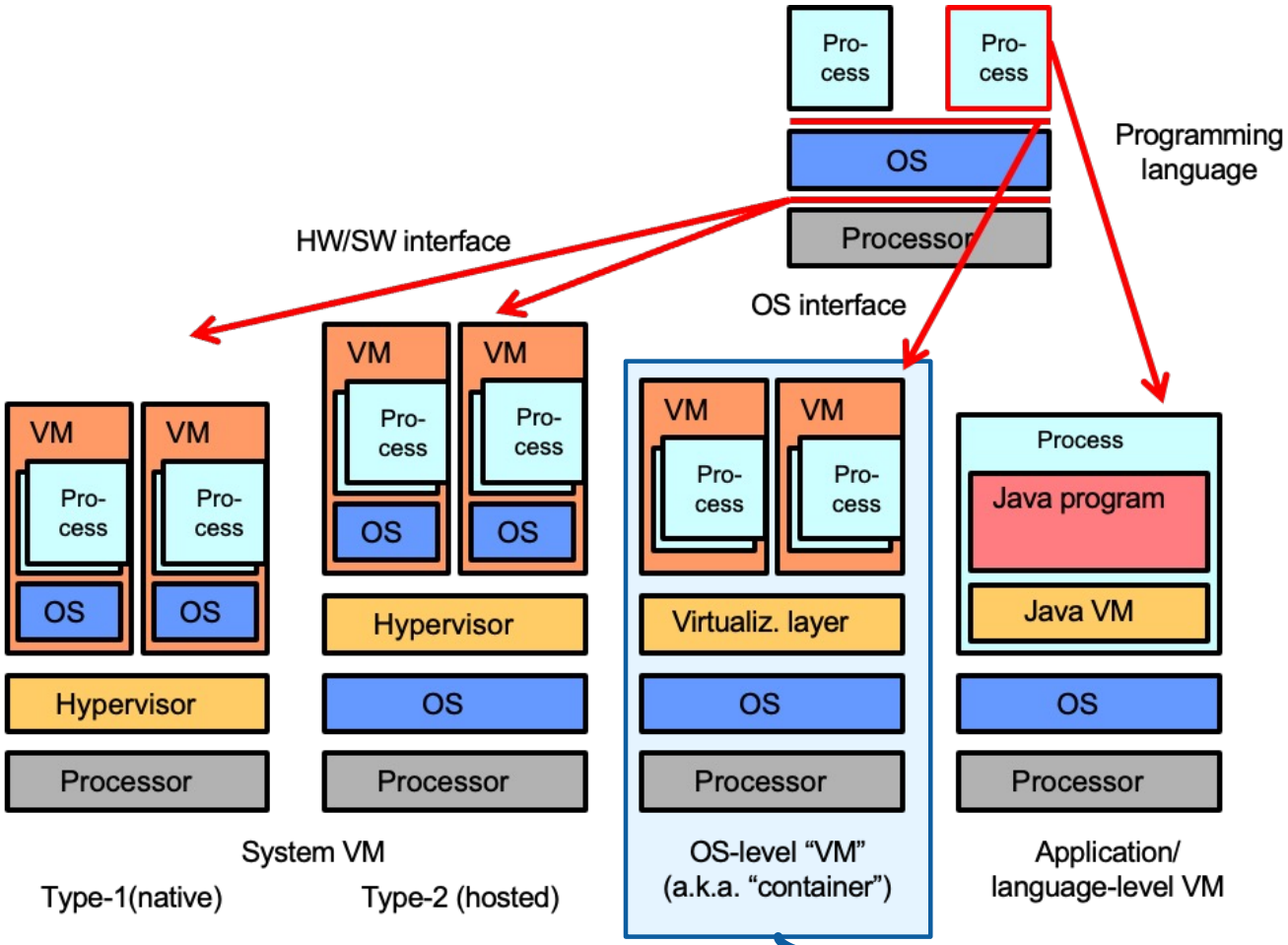
- Several talks, lectures and articles by Michael Kerrisk, including:
  - M. Kerrisk. **Containers as an illusion**. 2022. https://man7.org/conf/ndcsecurity2022/containers_as_an_illusion-NDC-Security-2022-Kerrisk.pdf
  - M. Kerrisk. **Linux containers in (less than) 100 lines of shell**. 2025. https://man7.org/conf/ndcsecurity2025/Containers_in_100_lines_of_shell--NDC-Security-2025-Kerrisk.pdf
- The text of the man pages for the different Linux facilities covered in the slides.

See also the end of this slide deck for additional references and acknowledgements.

Outline

- Introduction
- Linux capabilities
- Linux namespaces
- Linux cgroups
- Linux seccomp
- Misc. Linux storage-related facilities

Reminder: Types of virtualization



Our focus in this lecture:
Containers

(Figure courtesy of Gernot Heiser, UNSW Sydney)

Goals & motivation

Reminder: **What is the purpose of OS containers?**

Answer: Each container corresponds to **a set of processes** for which we provide a **virtualized view of the OS ABI**. Thus, it is possible to run several/many containers above the same OS kernel, while enforcing several types of **isolation guarantees**:

- **Isolation regarding software configuration**, including:
  - OS settings, daemons, libraries, application binaries, application files, …
  - OS logical resources (e.g., available PIDs, TCP port numbers, …)
  - File system structure, permissions and contents
- **Isolation regarding performance interferences**
- **Isolation regarding security attacks**

Goals & motivation [continued]

Within each container, **the running processes have the illusion that they are running alone** on the machine. In particular:

- They are **not aware** that they are (possibly) running concurrently with many other containers … and that each container provides a potentially different view of the current OS state.

- They are **not aware** that they are running concurrently with non-containerized processes.

- They are not (or at least not necessarily) aware that they are running inside a container.

Goals & motivation [continued]

Each container is a form of "mini system" instance (but without its own kernel – although it is not aware of that).

This notion of "mini system" implies that **each container should have its own logical resources and superuser**.

- Some examples of **logical resources**:
  - Init process (PID 1) instance
  - Set of mounted file systems (mount points)
  - Hostname
  - Network resources and configuration
- By **"superuser"**, we mean that a container should include some user/process(es) with administrative powers equivalent to the "root" user … but only valid inside the container.

Linux containers

In our lectures, we will focus on Linux containers as a concrete case study.

This choice is motivated by diverse reasons:

- Linux is currently one of the most used server operating systems.
- Linux was the main operating system impacted by the renewed interest and advanced in OS containers in the past 20 years.
- The Linux ecosystem provides rich support for containerization, including in-kernel facilities and external tools, and plays a central role in the Open Container Initiative (OCI).

That said, note that:

- Linux was not among the pioneering OS kernels in terms of container support (e.g., see FreeBSD jails & Solaris zones)
- **The Linux design internals/interfaces studied in these lectures are not necessarily/directly transposable to other operating systems.**

Linux containers – What is it exactly?

On Linux, the concept of containers is not a unified facility provided by the operating system. Rather, the management of containers is achieved by **combining a set of independent facilities** provided by distinct Linux kernel subsystems and tools, including:

- **Capabilities:** for controlling the operations that a given process/user is allowed to perform on various kinds of resources
- **Namespaces**: for configuring the system resources that are visible by a given process (network interfaces/ports, users, PIDs, …)
- **Control groups ("cgroups"):** for enforcing resource allocation limits
- **Seccomp:** for filtering the legitimate system calls that a process can make
- **Various storage-related facilities:** for example, overlay file systems, which allow stacking/combining file system hierarchies/images
- **Various networking-related facilities** [Not covered in this lecture due to lack of time]
- **Note that different container management tools/settings can combine/configure/use (all or some of) these facilities in different ways.**

Outline

- Introduction
- **Linux capabilities**
- Linux namespaces
- Linux cgroups
- Linux seccomp
- Misc. Linux storage-related facilities

Linux capabilities

- **The traditional Unix model essentially defines two categories of users regarding system privileges:**
  - the regular users
  - the superuser (corresponding to effective UID 0)
- This coarse granularity can introduce security/safety risks and lacks flexibility.

- The concept of Linux capabilities is aimed at addressing these issues, by following the **"principle of least privilege"**.

- **Capabilities divide the full power of the superuser into many smaller pieces** in order to have more flexible/granular control over the privileges granted to a given user/program/process.

Linux capabilities [continued]

- Recent versions of Linux (~kernel v6.10) define approximately 40 distinct capabilities. (For full list and details, see `man 7 capabilities`)
- (A process running with traditional superuser privileges has the full set of capabilities.)
- Some examples of capabilities:

Capability name	Corresponding privileges
`CAP_DAC_OVERRIDE`	Bypass file RWX permission checks
`CAP_KILL`	Bypass permission checks for sending signals
`CAP_NET_ADMIN`	Perform various network-related operations
`CAP_SYS_BOOT`	Reboot the system
`CAP_SYS_NICE`	Modify the scheduling priority and policy for processes
`CAP_SYS_TIME`	Modify the system clock

Linux capabilities [continued]

- **Each process can have zero, one, several or all of the distinct capability types.**
  - (Capabilities are actually managed as a per-thread attribute, but we will not get into such details. In this lecture, we will assume that capabilities are managed at a per-process granularity.)
  - The capabilities owned by a process may change during its execution. (See details below/later.)
- **Each process actually has several capability sets.**
  - We will skip most of these details and focus only on two type of set: the so-called **"permitted set"** and **"effective set"**.
  - **Permitted set:** Corresponds to the largest set of capabilities that a process may employ.
  - **Effective set:** Corresponds to the capabilities that are currently in effect for a process.
  - **In this lecture, we will mostly consider the effective set.**
- **A process can drop some capabilities**
  - **In permitted set:** dropping a capability is irrevocable (and only possible if it is not in effect).
  - **In effective set:** reacquiring the capability is possible, if it is still in the permitted set.

Linux capabilities [continued]

- **It is also possible to attach capabilities to executable files.**
  - (Like a process, an executable file has several capability sets, but we will skip such details.)
  - The capabilities are stored in the file metadata (as an extended attribute named `security.capability`).

  - **This allows providing (additional) capabilities to a process when it executes a given program/file.**
  - This concept is **a generalization of the classic "set-UID"/"set-GID" technique in Unix systems.**
    - Allows program/process launched by a non-privileged user to obtain the privileges/permissions of another user/group (possibly the superuser) by modifying the effective user ID (eUID) and/or effective group ID (eGID) of the running process.

Outline

- Introduction
- Linux capabilities
- **Linux namespaces**
- Linux cgroups
- Linux seccomp
- Misc. Linux storage-related facilities

Linux namespaces

This facility is one of the main mechanisms enabling the illusion of "running alone on the machine" to the group of processes encapsulated in a container.

- **The purpose of a namespace is:**
  - to offer **a private/customized view of a global system resource**
  - … to a **group of processes**
  - ... by providing them with a **virtual instance** of that resource.

- **Linux supports several types of namespaces**
  - Each namespace type corresponds to a distinct type of resource. (See examples in the next slides.)
  - **In the case of Linux containers, all (or most of) the namespace types are used in combination.**

Linux namespaces [continued]

- **For each namespace type T:**
  - At system startup time, there is exactly one namespace instance of type T ("initial namespace instance").
  - Afterwards, multiple instances of that namespace type T may coexist on the system.
  - **At any moment, a process belongs to (exactly) one of the existing namespace instances of type T.**
  - **Creating a new namespace instance of type T requires some administrator privileges, except for the "user" type.** (More details later.)

- **Warning:** The word "namespace" (alone) is often used with the meaning of "namespace instance".

Linux namespaces [continued]

Recent versions of Linux (~kernel v6.10) define **8 namespace types**:

Namespace	Flag (for syscalls/APIs)	Isolates resources such as:
Cgroup	`CLONE_NEWCGROUP`	Cgroup root directory
IPC	`CLONE_NEW_IPC`	System V IPC, POSIX message queues
Network	`CLONE_NEWNET`	Network devices, stacks, ports, etc.
Mount	`CLONE_NEWNS`	Mount points
PID	`CLONE_NEWPID`	Process IDs
Time	`CLONE_NEWTIME`	Boot and monotonic clocks
User	`CLONE_NEWUSER`	User and group IDs
UTS	`CLONE_NEWUTS`	Hostname

See also `man 7 namespaces` for an overview of the documentation.

Linux namespaces [continued]

Some key system calls for the management of namespaces:

- **`clone()`** (see **`man 2 clone`**)
  - **Warning:** This system call also implements a number of features unrelated to namespaces.
  - This system call can be used (among other features) to **create a new process**.
  - If the **`flags`** argument of the call specifies one or more of the **`CLONE_NEW*`** flags listed previously, then new namespaces are created for each flag, and the child (i.e., newly created) process is made a member of those namespaces.

- **`setns()`** (see **`man 2 setns`**)
  - Allows the calling process to join an existing namespace.
  - The namespace to join is specified via a file descriptor that typically refers to one of the **`/proc/pid/ns`** files described in the documentation (see **`man 7 namespaces`**).

Linux namespaces [continued]

Some key system calls for the management of namespaces [continued]:

- **`unshare()`** (see **`man 2 unshare`**)
  - **Warning:** This system call also implements a number of features unrelated to namespaces.
  - This system call allows a process (or thread) to disassociate parts of its execution context that are currently being shared with other processes (or threads).
  - **In particular, it allows moving the calling process to a new namespace (or set of namespaces).**
  - If the **`flags`** argument of the call specifies one or more of the **`CLONE_NEW*`** flags listed previously, then new namespaces are created for each flag, and the calling process is made a member of those namespaces.
- **`ioctl()`** (see **`man 2 ioctl_nsfs`**)
  - Various **`ioctl()`** operations can be used to discover information about namespaces.

Linux namespaces [continued]

Some key commands/utilities for the management of namespaces:

- **`unshare [options] [`*`program`* **`[`***`arguments`***`]]`**
  - In brief: run program in new namespaces (see **`man 1 unshare`**)
  - The **`unshare`** command creates new namespaces (as specified by the command-line options) and then executes the specified program (within the new namespaces).
  - If ***`program`*** is not given, then **`"${SHELL}"`** is run.

- **`nsenter [options] [`*`program`* **`[`***`arguments`***`]]`**
  - In brief: run program in different namespaces (see **`man 1 nsenter`**)
  - The nsenter command executes a program in the namespace(s) that are specified in the command-line options.
  - If ***`program`*** is not given, then **`"${SHELL}"`** is run.

Linux namespaces [continued]

Regarding the **lifecycle of namespaces:**

- **By default,** a namespace persists (i.e., exists) only as long as it has member processes. **A namespace is automatically destroyed when the last process in the namespace terminates or leaves the namespace.**

- However, there are **several factors that may preserve the existence of a namespace,** even though it has no member processes. Among them:
  - An open file descriptor or a "bind mount" exists for the `/proc/pid/ns/*` file associated to the namespace.
  - The namespace is hierarchical (i.e., a PID or user namespace), and has a child namespace.
  - (see `man 7 namespaces` for a full list)

Linux PID namespaces

PID namespaces have some specific aspects:

- **A process's PID namespace membership is determined when the process is created and cannot be changed thereafter.**
  - (Otherwise, this could lead to dynamically changing the PID of a process, which may break existing code.)
- **PID namespaces can be nested.**
  - Each PID namespace has a parent, except for the initial ("root") PID namespace.
  - A process **can see only** processes contained in its own PID namespace and in descendants of that namespace. **Not** processes in parent namespace (or ancestors).
  - **A given process typically appears with a different PID in different PID namespaces.**
    - For example: Process P_x can have PID 548 in the *initial PID namespace* N_0, PID 5 in a *child PID namespace* N_1, and PID 1 in a *grandchild PID namespace* N_2.

Linux PID namespaces [continued]

- **The first process created inside a new PID namespace:**
  - **has PID 1** (in that namespace)
  - **plays a specific and essential role:** it **acts as the "init" process,** quite similarly to the traditional init process used in Unix systems (https://en.wikipedia.org/wiki/Init).
- In particular, this "init" process **is in charge of:**
  - **performing container initialization** and creation of other processes
  - **becoming the parent of "orphaned" processes** in the container
- **If this "init" process terminates:**
  - **The kernel terminates all of the processes** in the namespace (via a `SIGKILL` signal).
  - **The namespace becomes unusable.**
  - This behavior reflects the fact that the "init" process is essential for the correct operation of a PID namespace.
- (see `man 7 pid_namespaces` for more details)

Linux user namespaces

- **The purpose of the "user" namespace type is to support the virtualization of user IDs ("UIDs") and group IDs ("GIDs").**
- In other words: **the UID (and GID) of a process running in a (user) namespace may be different from the UID (and GID) of the same process as seen from the outside** of that namespace.
- In particular, this can be used to achieve the following scenario:
  - A process with a nonzero UID outside the user namespace and with UID 0 (= superuser) inside the namespace.
  - Thus, a process can obtain superuser privileges for operations inside the namespace. (More details later.)
- **Reminder:** creating a new namespace instance of type "user" does **not** require any specific privilege, unlike all the other namespace types (which require capability **`CAP_SYS_ADMIN`**).
- (see **`man 7 user_namespaces`** for more details)

Linux user namespaces [continued]

- **User namespaces have a hierarchical relationship:**
  - A user namespace can have zero, one or several child user namespaces.
  - Each user namespace has a parent namespace (up to the initial user namespace, i.e., the only user namespace instance existing at system startup).
  - The parent of a user namespace N_x is the user namepace N_p of the process that created N_x using **`clone()`** or **`unshare()`**.

- The parent-child relationship determines some rules for the precise operation/management of capabilities within namespaces. (See details later.)

Linux user namespaces [continued]

User namespaces and capabilities:

- When a process enters a new/target user namespace N_{new}, this process gains the full set of capabilities within N_{new}.
  - However, the process has no capabilities in the parent/previous user namespace $N_{previous}$ from which it originates
  - … even if it was running with superuser power in $N_{previous}$.

- When using **`setns()`** to join an existing namespace, a process must have the **`CAP_SYS_ADMIN`** capability in the target namespace.
- The above rules imply that **it is only possible to join a descendant user namespace**.
- It is not permitted to use **`setns()`** to reenter the caller's current user namespace. This prevents a caller that has dropped capabilities from regaining those capabilities.

Linux user namespaces [continued]

- **After the creation of a new user namespace, an important step/requirement consists in defining a set of UID and GID mappings for this new namespace.**
- These mappings define how to map the UIDs (and GIDs) of the new namespace to the ones of the parent namespace.
  - More precisely: which range of “inside IDs” to be mapped to which range of “outside IDs”.
- The mappings are stored in two files in the `/proc` pseudo-filesystem (`/proc/PID/uid_map` and `/proc/PID/uid_map`).
  - Each process running in the user namespace has these files. Updating these files in only of these processes is sufficient.
  - These files are initially empty.
- There are many rules (not detailed here) defining how these files can (or cannot) be updated, in order to ensure that capabilities cannot leak (i.e., that a process inside a namespace cannot obtain more privileges than it should outside the namespace.

Linux user namespaces [continued]

Combining user namespaces and other types of namespaces:

- As we have seen:
  - Creating a user namespace instance requires no specific privilege, whereas creating another type of namespace instance requires the **`CAP_SYS_ADMIN`** capability.
  - When a process enters a new/target user namespace N_{U_new}, this process gains the full set of capabilities within N_{U_new}.
- **Therefore, it is possible to create and enter within a new user namespace instance N_{U_new} … and then create instances of other namespace types from within N_{U_new}.**
- Actually, when performing a call to **`clone()`** or **`unshare()`** with an argument specifying several types of namespaces (including the user type – for example: **`CLONE_NEWUSER`** | **`CLONE_NEWUTS`**), the kernel implements the above-described behavior:
  - The kernel creates the user namespace first, and then the other namespace types.
  - The other namespaces are owned by the user namespace.

Linux user namespaces [continued]

Rules about capabilities in user namespaces:

- **A process P has a capability C in a user namespace N if:**
  - P is a member of N
  - **and** C is present in its effective set
  - Note that this rule does not grant C to P in N's parent namespace.
- **A process P that has a capability C in a user namespace N also has C in all the descendants (user namespaces) of N.**
  - This means that the processes inside a given user namespace M are not isolated from the effects of a privileged process running in another user namespace that is the parent (or another ancestor) of M.
- **Regarding the creator/owner of a user namespace:**
  - When a user namespace N is created, the kernel records the eUID (effective UID) of the creator process a being the "owner" of N.
  - A process residing in the parent of N and whose eUID matches N's owner has all the capabilities in N, and also in all the user namespaces that are descendants of N.

Linux user namespaces [continued]

Regarding the privileges in a user namespace:

- **Having a capability inside a user namespace permits a process to perform operations** (that require privilege) **only on resources governed by that user namespace**.

- **Interplay between user namespaces and non-user namespaces**
  - Vocabulary: By “non-user namespace”, we mean a namespace whose type is not “user” (for example, a PID namespace or a mount namespace).
  - The kernel associates each non-user namespace instance with a specific user namespace instance. In other words, **each non-user namespace is owned by a given user namespace**.
  - When process P creates a new non-user namespace N, the user namespace of P becomes the owner of N.

Linux user namespaces [continued]

Regarding the privileges in a user namespace: [continued]

What happens when a process P wants to perform privileged operations (requiring capability C) on resources that are governed by a non-user namespace N?

These privileged operations are **only allowed if P indeed has capability C in the user namespace that owns N**.

Thus, it is possible to obtain a flexible, safe and secure nesting of privileges. In particular:

- **A process P can have a full set of capabilities inside a given user namespace U and no capabilities in the outer user namespace(s) than contain U. And P can have full power over the resources governed by the non-user namespaces owned by U.**
- Depending on the hierarchy of (user and non-user) namespaces and owner relationships, a given process can potentially be granted privileges for certain operations on some types of resources (e.g., time settings) but not on others (e.g., network settings).

Linux user namespaces [continued]

Interaction with resources that are not governed by namespaces:

- **Some privileged operations manipulate resources that are not associated with any namespace type** (at least not in the current Linux versions).
- **For example:**
  - Changing the system (calendar) time (governed by **`CAP_SYS_TIME`**)
  - Loading a kernel module (governed by **`CAP_SYS_MODULE`**)
  - Creating a device (governed by **`CAP_MKNOD`**)
  - Modifying the scheduling priority/settings of a process (governed by **`CAP_SYS_NICE`**)
- **Only a process with the appropriate capabilities in the initial user namespace can perform such operations.**
  - **In other words: Having all capabilities in a non-intial user namespace does not grant any specific privilege over such resources.**
  - For example: A process attempting to change the system time only succeeds if it has the **`CAP_SYS_TIME`** capability **in the initial user user namespace**.

Outline

- Introduction
- Linux capabilities
- Linux namespaces
- **Linux cgroups**
- Linux seccomp
- Misc. Linux storage-related facilities

Linux Cgroups

- **The purpose of a cgroup is to measure and limit the usage of various resource types by a given set/group of processes.**
  - This allows guaranteeing that a given group does not overwhelm the system (and does not impact other groups) with excessive resource demands.
  - The name “cgroup” stands for **“control group”**.

- Cgroups are managed in a **hierarchical** manner.
  - For example: the limits placed on a cgroup at a higher level in the hierarchy cannot be exceeded by descendant cgroups.

- (See `man 7 cgroups` for a detailed documentation.)

Linux Cgroups [continued]

- The **user/kernel interface** for the management of cgroups is provided **via a pseudo-filesystem called cgroupfs**.

- In the kernel, the implementation of the cgroups logic is split in different parts:
  - **Group management:** in the core cgroup kernel code
  - **Resource tracking/limits:** in **per-resource-type subsystems** (a.k.a. **"resource controllers"** or **"controllers"**)

- **Currently, Linux provides two distinct/orthogonal implementations of the cgroups facility, named version 1 and version 2.**

Linux Cgroups [continued]

Important warning about cgroup versions:

- **Cgroups v1** was initially released in Linux kernel v2.6.24 (2008). However, the **uncoordinated development of the different controllers resulted in inconsistencies and issues.**
- **Cgroups v2** is intended as a replacement and became usable around Linux kernel v4.18 (2018).

- By ~2022, many projects (Linux distributions, tools, …) have moved to cgroups v2
  - … but v1 remains available (and is unlikely to be removed, due to compatibility reasons for existing code bases).
  - Controllers for v1 and v2 can coexist in the same system (e.g., because some types of controllers are not available for version 2). A given controller type cannot be simultaneously used by both versions.
- **In this lecture, we only consider cgroups version 2.**

Linux Cgroups [continued]

Cgroups pseudo-filesystem:

- For cgroups v2: mounted at **`/sys/fs/cgroup`**
- Creating a directory allows creating a new cgroup.
- The directory hierarchy defines the hierarchy of cgroups.
- Creating and writing to different text files (for each controller) allow configuring the properties of a given cgroup. Examples:
  - (max CPU usage per 100ms) **`echo '50000 100000' > /sys/fs/cgroup/mygroup/cpu.max`**
  - (max number of tasks) **`echo 5 > /sys/fs/cgroup/mygroup/pids.max`**
- Writing a PID in a specific file puts the corresponding process in a cgroup.
  - Example: **`echo 12567 > /sys/fs/cgroup/mygroup/cgroup.procs`**
- Reading specific files returns information about the current cgroup state/statistics.
  - Example: **`cat /sys/fs/cgroup/mygroup/pids.current`**

Linux Cgroups [continued]

Some examples of resource controllers:

- **CPU usage**
- **CPU set** (for binding processes to a specified set of CPU cores and NUMA nodes)
- **Memory usage** (including process memory, kernel memory, and swap usage)
- **PIDs** (limits the number of processes/threads that can be created)
- **Freezer** (suspend/resume all the processes in a cgroup)
- **Disk/block I/O bandwidth**
- **Network traffic** (requires eBPF programs/filters for cgroups v2)
- **Access to devices** (requires eBPF programs/filters for cgroups v2)

Linux Cgroups [continued]

Some additional details about Cgroups v2:

- **Delegation**
  - By default, managing cgroups requires administrator privileges.
  - Cgroup delegation allows passing the management of some subtree of the cgroup hierarchy to a nonprivileged user/process, with containment guarantees.
- **The “no internal processes” rule**
  - (Except for the root/initial cgroup) Processes may reside only in leaf nodes of the cgroup hierarchy (i.e., in cgroups that do not themselves contain child cgroups).
  - This avoids the need to decide how to partition resources between processes at different levels.
- **No thread-granularity control**
  - All the threads of a process must be in the same cgroup.

Linux Cgroups [continued]

Some additional details about Cgroups v2: [continued]

- **Thread mode**
  - **This mode allows relaxing the previously discussed restrictions imposed by the initial cgroups v2 design:** no thread-granularity control & no internal processes.
  - This mode requires an explicit declaration of a **cgroup type** (“threaded” type, as opposed to the default “domain” type), allowing to define **“threaded subtrees”** in the cgroup hierarchy.
  - This mode requires using **“threaded controllers”** (as opposed to **“domain controllers”**), which support the management of threaded subtrees.

Outline

- Introduction
- Linux capabilities
- Linux namespaces
- Linux cgroups
- **Linux seccomp**
- Misc. Linux storage-related facilities

Linux Seccomp

Motivation:

- In a general-purpose (i.e., feature-rich) OS based on a monolithic-kernel design such as Linux, **the user/kernel interface constitutes a major attack surface.**
- Today, the Linux kernel provides approx. 400 syscalls.
- **Security attacks**
  - **Origin:** Attacks performed by **malicious applications** or by **compromised applications**
  - **Target:** Attacks **against the OS itself and/or the other applications**
  - **Vectors:** Attacks exploiting various vulnerabilities of the kernel interface, such as:
    - API design flaws
    - kernel implementation bugs
    - incorrect/weak system configuration
    - …

Linux Seccomp [continued]

Motivation: [continued]

- Each syscall offered by the user/kernel interface potentially provides additional attack opportunities
- … but **most programs only need to use a small fraction of the full set of available syscalls**.

- **The usage of certain syscalls (and/or their arguments) by an application could hint at the fact that this application is malicious (or has been compromised).**

- Also, **according to the "principle of least privilege", sandboxing an application can help contain the potential damages of an attack**.
- Sandboxing example: restricting accesses to the network and to the file system.

Linux Seccomp [continued]

- In essence, **seccomp allows restricting/inspecting/auditing the syscalls that a given application (or set of processes) can make**.
- More precisely, seccomp **allows installing a filter program that can make decisions** about all the syscalls issued by an application.
- **The filter program returns a decision to the kernel** indicating how the syscall should be handled:
  - Allow (i.e., perform the syscall)
  - Kill the calling process
  - Make it look like the syscall failed (return a specified error code without actually executing the syscall)
  - Send a notification to supervisor process (which might perform actions on behalf of the caller process)
  - …

Linux Seccomp [continued]

- The **most flexible and advanced features of seccomp** (especially filter programs) **are actually provided by seccomp-bpf** (introduced around 2012), which is an extension of the base seccomp facility (introduced around 2005).

- **Seccomp-bpf relies itself on the eBPF framework**, which allows injecting user-provided code into the Linux kernel.
  - eBPF is also known as “BPF” for short (although it is somewhat different from the original technology named BPF).
  - BPF programs are compiled into bytecode.
  - BPF uses an in-kernel virtual machine to check and execute the bytecode.
  - This approach allows protecting the kernel from issues introduced by bugs in the user-supplied code, which could otherwise jeopardize the safety of the kernel (e.g., due to crashes, hangs, kernel data corruption).

Outline

- Introduction
- Linux capabilities
- Linux namespaces
- Linux cgroups
- Linux seccomp
- **Misc. Linux storage-related facilities**

Linux bind mounts

- **A ”bind mount” is a particular type of mountpoint that allows remounting a part of the filesystem hierarchy somewhere else in that same hierarchy.**

- **In other words, a bind mount provides another access point to an existing directory (sub)tree.**
  - Any modification performed via one of the paths leading to the impacted files/directories is also (and immediately) reflected to the other side.
  - (Similarly to traditional mounts, once a bind mount becomes effective, the initial content of target directory is not accessible anymore.)

- **Note that a bind mount is not the same thing as a “symbolic link” or a “hard link” in a file system.** It has distinct properties and behaves differently.

Linux bind mounts [continued]

Some use cases:

The usage of a bind mount is typically aimed at providing an alternate view of an existing directory tree.

This is **motivated by various reasons** such as: convenience, safety/security guarantees, compatibility/functional requirements, file visibility, multi-tenancy and virtualization.

For example:

- Providing a read-only view of an existing FS
- Grafting various subtrees in the customized view of the FS tree of a container
- Accessing files that are currently hidden behind a mount point
- Working around certain issues with symbolic links

Linux bind mounts [continued]

Some simple examples:

- **Example 1:**
  - `mount --bind /my/directory /my/new/mount/point`
  - Both paths must correspond to existing directories
  - Note that, with the above version, if there are mount points under /my/directory, their contents are are not visible under /my/new/mount/point.
  - Replicating the mount points is possible using rbind:

    ```
    mount --rbind /my/directory /my/new/mount/point
    ```

- **Example 2:**
  - This is also possible with files:
  - `mount --bind /some/file /another/file`
- (For a longer introduction/motivation, see also https://unix.stackexchange.com/questions/198590/what-is-a-bind-mount)
- (For additional details and options, see `man 2 mount` and `man 8 mount`)

Linux pivot_root

- **Here "root" means the root of the filesystem tree**, i.e., the top-level directory serving as the starting point of the file/directory hierarchy.
- Various scenarios require to change the view of the filesystem tree that is provided to a process (and its descendants).
- **The `pivot_root()` syscall allows changing the root mount in the mount namespace of the calling process.**
  - More precisely, it moves the root mount (and its descendant mounts) to the directory **`put_old`** and makes **`new_root`** the new root directory.
  - The calling process must have the **`CAP_SYS_ADMIN`** capability in the user namespace that owns the mount namespace of the caller.
  - Note that **`pivot_root()`** is not equivalent to the traditional **`chroot()`** syscall and is considered more robust/secure.

- There is also a **`pivot_root`** command (based on the above syscall) that can be used in a shell.
- (For additional details and rules/restrictions, see **`man 2 pivot_root`** and **`man 8 pivot_root`**)

Linux union/overlay file systems

- Various Unix/Linux use cases (not only about containers) rely on the possibility to **combine/"stack" several file-system (FS) hierarchies/"trees"** (sometimes also called "branches", "mounts" or "layers"), in order to **export a single, unified view** of them as a virtual FS, accessible via a new mount point.
- In certain configurations, **this idea also allows providing a writable FS on top of a read-only base layer**, by leveraging a **"Copy-on-Write" (CoW)** approach.

- **Special FS types**, generally referred to as **"Union FS", "Overlay FS"** or **"Union mount (FS)"**, have been specifically designed to address these scenarios. **These three generic names are often used interchangeably.**
- **Some typical use cases:**
  - "Live" Linux distributions (on a USB stick or previously on a CD/DVD)
  - Embedded/IoT devices (for quick factory reset and/or for flash storage with limited write cycles)
  - Managing container image layers (see details in another lecture)

Linux union/overlay file systems [continued]

General principles:

- **When combining two FS layers:**
  - For directories with the same path: contents are merged
  - For a file with the same name (i.e., full-path name) in both layers, one of the layers gets priority over the other. This priority rule is clearly specified and sometimes configurable. Often, the version in the uppermost layer has priority.
  - Each layer can be either read-only or read-write (with clear priority rules for writes).
- The above principles can be applied several times: **a complete union/overlay can combine more than two FS layers**.
- **In practice, for many setups:**
  - **Only the top FS layer is in read-write mode.**
  - **All the other ones are in read-only mode.** (Thus, they can be shared/reused safely.)
  - The modifications made to the (writable) top FS layer are volatile (and often stored in main memory).

Linux union/overlay file systems [continued]

- **There are currently several union/overlay filesystems for Linux:**
  - **Main ones:** UnionFS, Aufs, OverlayFS
  - Also: mhddfs, mergerfs
- Although they are share many high-level ideas, they have some differences in design choices, features and semantics.
- In our examples/discussions for the remainder of this course, **we will mainly consider OverlayFS.**

- **OverlayFS:**
  - Implemented as a kernel module
  - Merged into the Linux mainline kernel since 2014
  - Now often used as the one of the main options for many container-related tools

Linux OverlayFS

Main characteristics:

- Relatively simple design (fewer features/options than some other union filesystems).
- Allows combining two or more layers into a merged FS view.
- Supports at most one writable layer (the uppermost one).
- Each FS layer is provided as a directory tree. The trees of the different layers may be stored on the same FS or on different ones.
- After a file is opened, all the operations on that file go directly to the underlying (lower or upper) filesystems.
- The lower FS can be of any FS type supported by Linux, and can even be another overlayFS.
- When writable, the upper FS introduces some requirements (e.g., must support the creation of trusted.* extended attributes) so NFS is not suitable.

Linux OverlayFS [continued]

Additional details

- In order to support file/directory deletion without impact the lower FS, overlayFS needs to record in the upper FS that such deletions have occurred. This is achieved via so-called “opaque” directories and file whiteouts (i.e., using modified extended attributes and special files).
- An overlay mount may use the same lower layer path as another overlay mount and it may use a lower layer path that is beneath or above the path of another overlay lower layer path.
- **Warning:** Using an upper layer path and/or a workdir path that are already used by another overlay mount is not allowed.
- **Warning:** Changes to the underlying filesystems while part of a mounted overlayFS are not allowed. If the underlying FS is changed, the behavior of the overlay is undefined, though it will not result in a crash or deadlock.
- Offline changes, when the overlay is not mounted, are allowed to the upper tree. Offline changes to the lower tree are only allowed in some cases (depending on the configured overlay options).
- When the underlying filesystems supports NFS export and the “nfs_export” feature is enabled, an overlay filesystem may be exported to NFS.

Linux OverlayFS [continued]

Mount command:

```
mount -t overlay overlay \
      ./path/to/merged/dir \
      -o lowerdir=/path/to/lower/dir1,upperdir=/path/to/upper/dir,workdir=/path/to/work/dir
```

Indicates that the FS is of type overlay. (The second occurrence is of "overlay" is just a placeholder because we do not mount a device.)

Mount point

Path to the directory corresponding to the lower FS layer.
Does not need to be on a writable FS.

Note: It is example to provide an ordered list of several directories, each acting as a distinct FS layer. (See example later.)

Path to the directory corresponding to the upper FS layer.
It is normally stored on a writable FS.

workdir is a directory used internally by OverlayFS.
(For instance, to prepare files before they are atomically switched to upperdir).
workdir must be an empty directory stored on the same FS as upperdir.

Linux OverlayFS [continued]

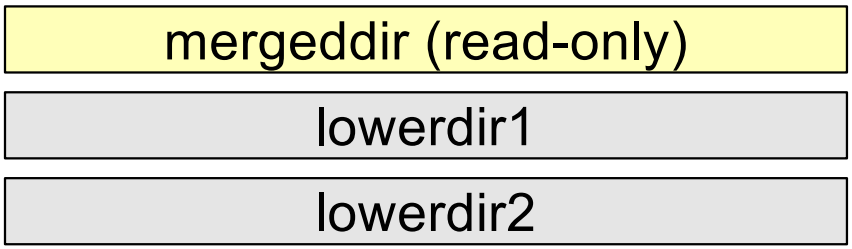
Usage example #1: Read-only overlay

In this case, we want to combine two layers to obtain merged view as a read-only mountpoint.

```
mount -t overlay overlay \
      ./path/to/mergeddir \
      -o lowerdir=/path/to/lowerdir1:/path/to/lowerdir2
```

When upperdir is not specified, the overlay is mounted as read-only.

Note that the order of the list matters: the rightmost directory corresponds to the lowest layer.

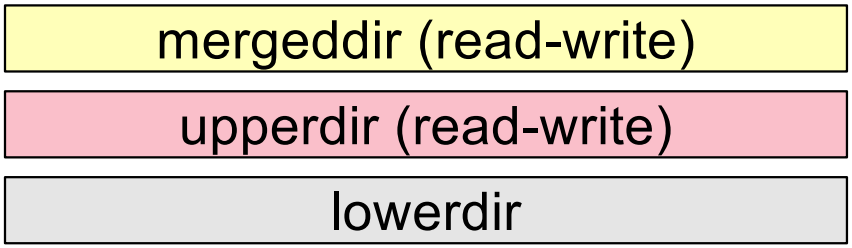


Linux OverlayFS [continued]

Usage example #2: Writable overlay with 2 layers

In this case, we want to combine two layers to obtain a merged view as a read-write mountpoint.

```
mount -t overlay overlay \
      ./path/to/mergeddir \
      -o lowerdir=/path/to/lowerdir,upperdir=/path/to/upperdir,workdir=/path/to/workdir
```



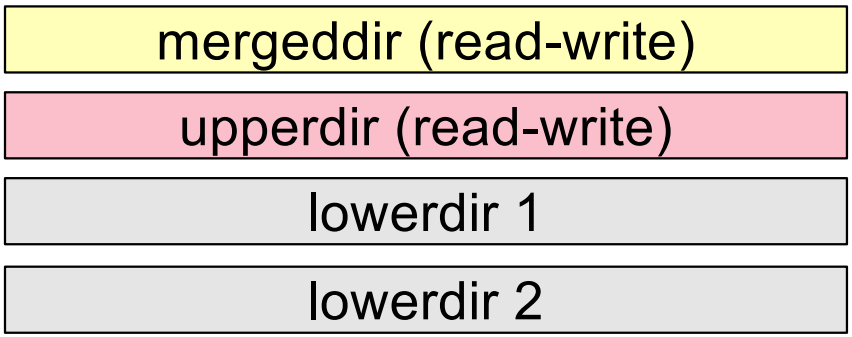
Linux OverlayFS [continued]

Usage example #2: Writable overlay with 3 layers or more

In this case, we want to combine three layers (or more) to obtain a merged view as a write mountpoint.

```
mount -t overlay overlay \
      ./path/to/mergeddir \
      -o lowerdir=/lowerdir1:/lowerdir2,upperdir=/upperdir,workdir=/workdir
```

Note that the order of the list matters: the rightmost directory corresponds to the lowest layer.



Linux OverlayFS [continued]

- For more details, see documentations:
  - Usage overview: https://wiki.archlinux.org/title/Overlay_filesystem
  - Mount command (see section “mount options for overlay”): https://man7.org/linux/man-pages/man8/mount.8.html
  - Linux kernel (advanced details): https://docs.kernel.org/filesystems/overlayfs.html

A note about nesting

- **Remember the initial goals for containers:**
  - "Within each container, **the set of running processes have the illusion that they are running alone on the machine**. […] **They are not (or at least not necessarily) aware that they are running inside a container**."
- **So it should be possible to run a container inside another container.**
- This is supported by various design aspects of the facilities that we have studied. In particular, many of these facilities support nesting. For example:
  - PID namespaces are hierarchical
  - User namespaces are hierarchical
  - Cgroups are hierarchical
  - Bind mounts are hierarchical
  - Overlay filesystems are hierarchical

References & Acknowledgments

General references:

- M. Kerrisk. **Containers as an illusion**. 2022. https://man7.org/conf/ndcsecurity2022/containers_as_an_illusion-NDC-Security-2022-Kerrisk.pdf

- M. Kerrisk. **Linux containers in (less than) 100 lines of shell**. 2025. https://man7.org/conf/ndcsecurity2025/Containers_in_100_lines_of_shell--NDC-Security-2025-Kerrisk.pdf

- Michael Kerrisk. **Linux security and isolation APIs essentials**. October 2025. https://man7.org/training/download/Linux_Security_and_Isolation_APIs_Essentials-mkerrisk_man7.org.pdf

References & Acknowledgments [continued]

References about Linux capabilities:

- **Linux man pages:**
  - capabilities(7): https://man7.org/linux/man-pages/man7/capabilities.7.html
  - libcap(3): https://man7.org/linux/man-pages/man3/libcap.3.html
  - capsh(1): https://man7.org/linux/man-pages/man1/capsh.1.html
  - getcap(8): https://man7.org/linux/man-pages/man8/getcap.8.html
  - getcaps(8): https://man7.org/linux/man-pages/man8/getpcaps.8.html
  - captree(8): https://man7.org/linux/man-pages/man8/captree.8.html

- Sergen Hallyn, Andrew Morgan. **Linux Capabilities: making them work.** 2008. https://www.kernel.org/doc/ols/2008/ols2008v1-pages-163-172.pdf

- Michael Boelen. **Linux capabilities 101.** https://linux-audit.com/kernel/capabilities/linux-capabilities-101/

References & Acknowledgments [continued]

References about Linux cgroups:

- **Linux man pages:**
  - cgroups(7): https://man7.org/linux/man-pages/man7/cgroups.7.html
  - cgroups_namespaces(7): https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html
- **Linux kernel documentation:**
  - Cgroups v1: https://docs.kernel.org/admin-guide/cgroup-v1/index.html#cgroup-v1
  - Cgroups v2: https://docs.kernel.org/admin-guide/cgroup-v2.html
- Ivan Velichko. **Controlling Process Resources with Linux Control Groups.** https://labs.iximiuz.com/tutorials/controlling-process-resources-with-cgroups
- ArchLinux wiki. **Cgroups documentation.** https://wiki.archlinux.org/title/Cgroups
- Red Hat. **Resource management guide.** https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/resource_management_guide/index

References & Acknowledgments [continued]

References about Linux namespaces:

- **Linux man pages:**
  - namespaces(7): https://man7.org/linux/man-pages/man7/namespaces.7.html
  - user_namespaces(7): https://man7.org/linux/man-pages/man7/user_namespaces.7.html

- Michael Kerrisk. **Namespaces in operations [article series in Linux Weekly News]**, 2013. https://lwn.net/Articles/531114/

- Steve Ovens. **The 7 most used Linux namespaces.** 2021. https://www.redhat.com/en/blog/7-linux-namespaces

- Mikail Kirov. **Digging into Linux namespaces.** 2021.
  - Part 1: https://blog.quarkslab.com/digging-into-linux-namespaces-part-1.html
  - Part 2: https://blog.quarkslab.com/digging-into-linux-namespaces-part-2.html

References & Acknowledgments [continued]

- **References about sandboxing with Linux seccomp:**
  - **Seccomp-BPF documentation in the Linux kernel:** https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html
  - seccomp(2) – Linux man page: https://man7.org/linux/man-pages/man2/seccomp.2.html
  - Terence Kelly and Edison Fuh. **Sandboxing: Foolproof Boundaries vs. Unbounded Foolishness.** ACM Queue. 2025. https://queue.acm.org/detail.cfm?id=3733699

References & Acknowledgments [continued]

References about union/overlay file systems:

- **A series of articles by Valerie Aurora on the history and main aspects of Unix union filesystems:**
  - **Unioning file systems: Architecture, features, and design choices**. Linux Weekly News. 2009. https://lwn.net/Articles/324291/
  - **Union file systems: Implementations, part I**. Linux Weekly News. 2009. https://lwn.net/Articles/325369/
  - **Unioning file systems: Implementations, part 2**. Linux Weekly News. 2009. https://lwn.net/Articles/327738/
  - **A brief history of union mounts**. Linux Weekly News. 2010. https://lwn.net/Articles/396020/
- **Some documents about the Linux OverlayFS filesystem:**
  - Jake Edge. **Another union filesystem approach**. Linuw Weekly News. 2010. https://lwn.net/Articles/403012/
  - OverlayFS – Linux kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html