Introduction to Containers

Prof. Shih-Wei Li

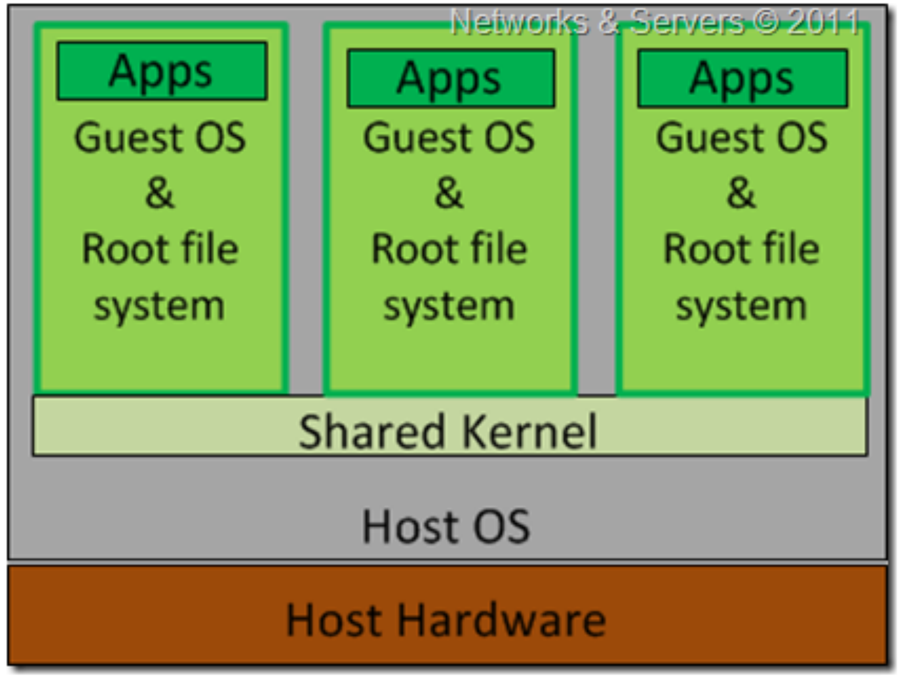
Department of Computer Science and Information Engineering
National Taiwan University

Agenda

- Introduction of Containers
- Mechanisms for Supporting Containers

OS-level Virtualization

- Also called containerization
- A shared OS supports multiple isolated user-space instances
  - Leverages low-level OS services to enforce the isolation; ex: docker containers use cgroup and namespace!
- Analogous to that the base shared kernel forked itself into multiple copies to provide different OS environments:
  - What is an OS environment?
- Benefits: security and portability

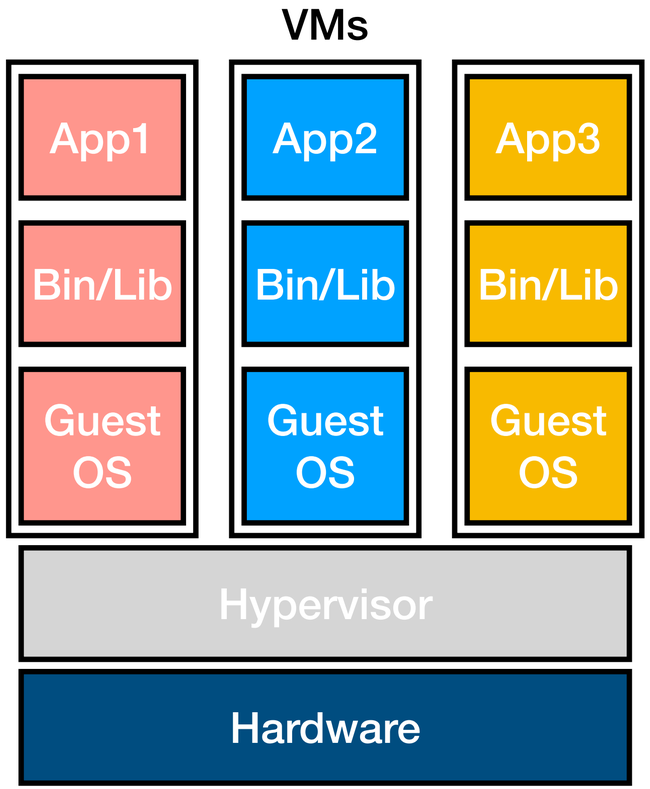


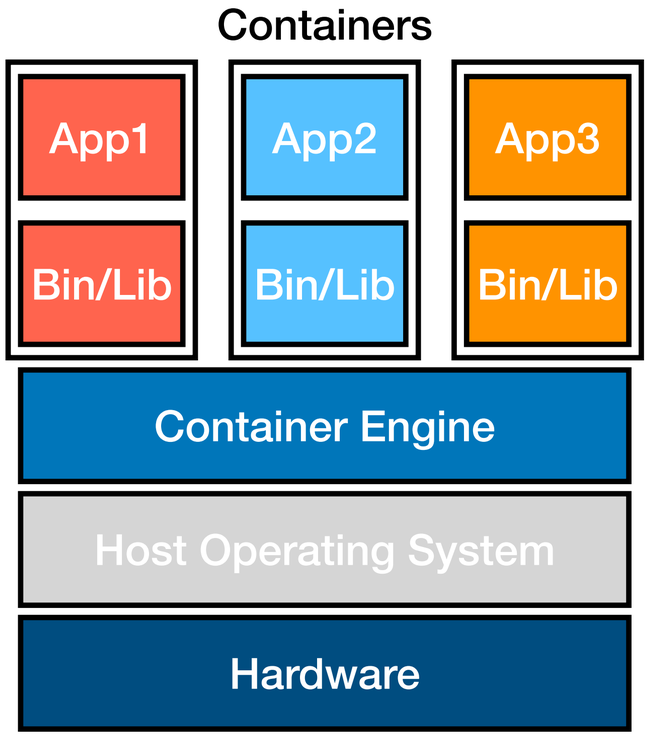
https://forum.huawei.com/enterprise/en/data/attachment/forum/202111/11/143452s32ecrayco2uwjre.png

OS-level Virtualization

- Container: An isolated OS environment with file system, processes, network, and block I/O space
  - LXC:
    - Introduced in 2008; uses exiting Linux mechanisms; requires no Linux patches
  - Docker Containers:
    - Introduced in 2013; initially uses ***LXC*** but mainly uses ***libcontainer*** later
- Chroot; FreeBSD Jail
- Linux VServer:
  - Introduced in 2001 for resource (file systems, network addresses, memory) partitioning

Containers v.s. VMs





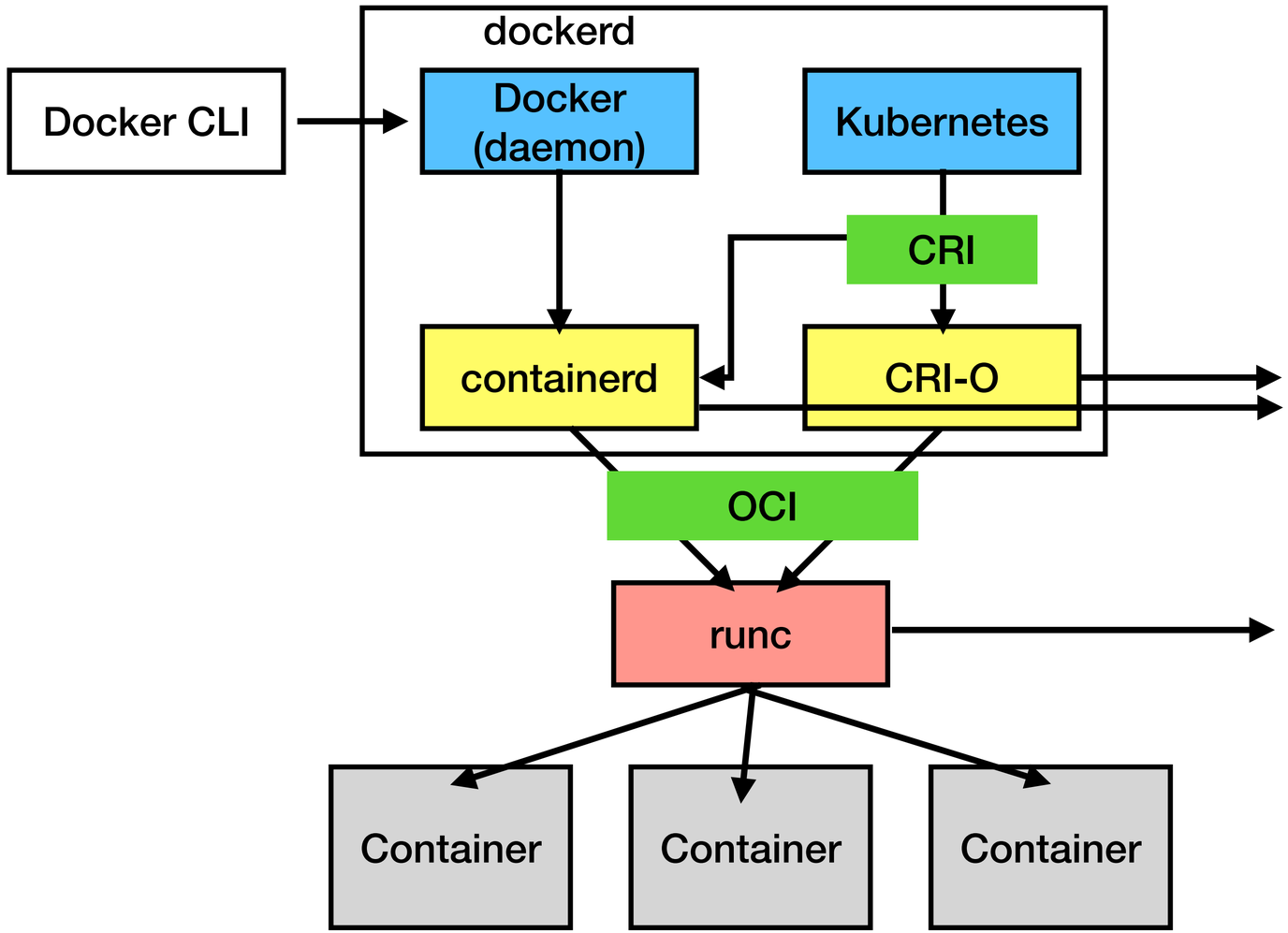
Containers v.s. VMs

	Virtual Machine	Container
Operating System	Runs an different isolated guest OSes	Runs on the same OS as the host (shared)
Isolation	Isolated from other VMs and the hypervisor/host OS	Rely on mechanisms from the host OS to enforce isolation
Size	Large (in gigabytes)	Small (in megabytes)
Startup Time	Slow (in seconds to minutes)	Fast (in seconds)
Security	?	?

Container Ecosystem

- You may have heard of the jargons: Kubernetes, Docker, OCI, CRI, runc
- What is their role in a container ecosystem?

Container Ecosystem



<u>High-Level Runtimes:</u> handle image mgmt, unpacking, etc.

<u>Low-Level (OCI) Runtimes:</u> creates and run containers

Container Ecosystem

- Tools to run containers: Docker, Kubernetes, etc.
- Two primary standards in the container ecosystem:
  - ***Open Container Initiative (OCI):*** a set of standards for how images are build and stored, and how container runtimes should execute containers
  - ***Container Runtime Interface (CRI):*** defines the API for container orchestration (e.g., Kubernetes) for different container runtimes
    - Container orchestration: management of the lifecycle of containers in a distributed environment — deployment on servers, scaling based on demands, monitoring execution status (health), manage communication methods, etc.

Container Ecosystem

- CRI compliant (high-level) container runtimes:
  - ***containerd:*** high-level container runtime from Docker that implements the CRI specification; is installed when Docker is installed
  - ***CRI-O:*** a CRI implementation developed by IBM/RedHat/SUSE/etc
- Container runtimes call the OCI compliant runtimes (ex: ***runc***) to create and run containers

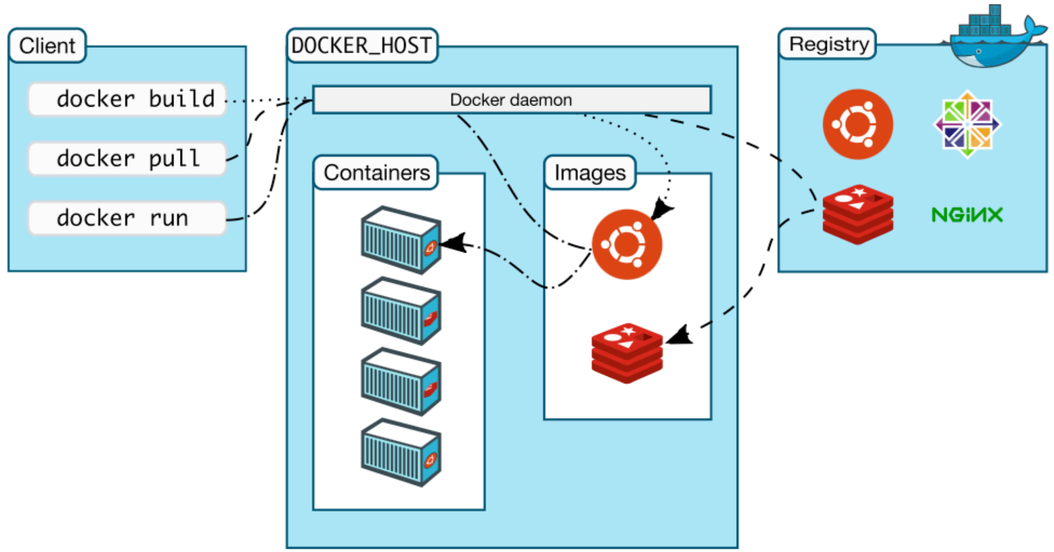
Container Ecosystem

- OCI is backed by a group of tech companies that maintain a specification for the container image format and how containers should be run
  - May have different OCI-compliant container implementations
- ***runc*** implements the OCI specification
  - Introduced by Docker to be lightweight and portable
  - Includes “libcontainer” (Go library for running containers)
  - Supports low-level functionality for containers by leveraging existing Linux kernel features like namespaces and cgroups (will discuss later)

Container Ecosystem

- There are several OCI-compliant alternatives to runc:
  - ***crun:*** C-based, similar to runc
  - ***Kata-runtime:*** from the kata-containers project, implements the OCI specification as individual lightweight VMs
  - **runsc*:*** implemented by Google to support containerized environment ***gVisor***; implements 200+ Linux system calls in the user space OS kernel to improve security (will discuss next week)

Docker Containers



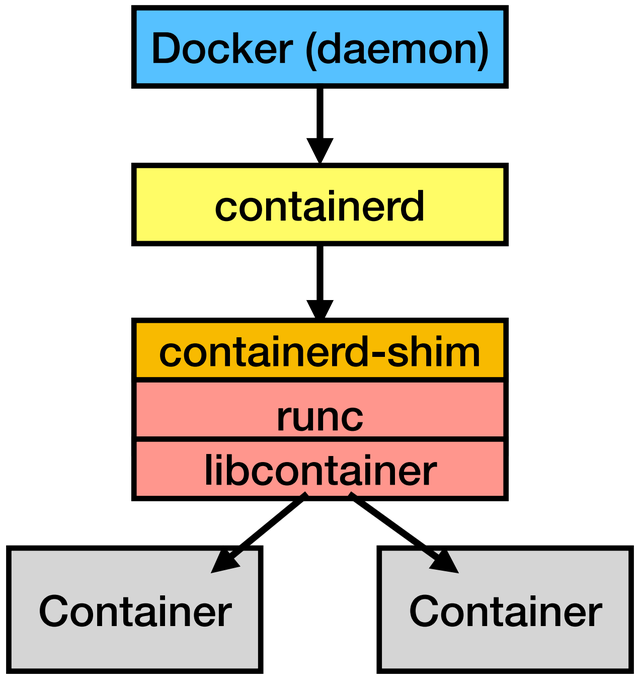
- Docker is a tool to run and manage containers
- Docker uses a client-server architecture
  - The docker client either connects to the docker engine (daemon)
  - The docker engine runs either locally or remote
  - Client-server communications via REST API

Docker Containers

- Docker engine/daemon
  - Listens for Docker API requests and manages docker images, containers, networks, and disk volumes
- Client
  - Users interact with the docker engine by executing commands via Docker Command Line Interface (CLI); performing “docker run”, “docker pull”, etc.
  - Users can use the Docker compose tool for running multi-container applications
- Docker registries
  - Stores docker images; Docker Hub is a public registry for storing images

Docker Containers

- containerd calls containerd-shim, which it then calls runc
- runc builds on the libcontainer to interface with the Linux kernel



Docker Containers

- Runs an ubuntu container, attaches to command line session, and runs /bin/sh
  - Docker pulls the ubuntu image from the registry if the image does not exist locally
  - Docker creates a new container
  - Docker allocates a read-write filesystem to the container
  - Docker creates a network interface to the container, assign an IP address
  - Docker starts the container (calls containerd then runc) and executes /bin/bash

```
docker run -i -t ubuntu /bin/bash
```

Docker Containers

- Runs an ubuntu container, attaches to command line session, and runs /bin/sh
  - Docker pulls the ubuntu image from the registry if the image does not exist locally
  - Docker creates a new container
  - Docker allocates a read-write filesystem to the container
  - Docker creates a network interface to the container, assign an IP address
  - Docker starts the container (calls containerd then runc) and executes /bin/bash

```
docker run -i -t ubuntu /bin/bash
```

Docker Containers

- Runs an ubuntu container, attaches to command line session, and runs /bin/sh
  - Docker pulls the ubuntu image from the registry if the image does not exist locally
  - Docker creates a new container
  - Docker allocates a read-write filesystem to the container
  - Docker creates a network interface to the container, assign an IP address
  - Docker starts the container (calls containerd then runc) and executes /bin/bash

```
docker run -i -t ubuntu /bin/bash
```

Docker Containers

- Runs an ubuntu container, attaches to command line session, and runs /bin/sh
  - Docker pulls the ubuntu image from the registry if the image does not exist locally
  - Docker creates a new container
  - Docker allocates a read-write filesystem to the container
  - Docker creates a network interface to the container, assign an IP address
  - Docker starts the container (calls containerd then runc) and executes /bin/bash

```
docker run -i -t ubuntu /bin/bash
```

Docker Containers

- Runs an ubuntu container, attaches to command line session, and runs /bin/sh
  - Docker pulls the ubuntu image from the registry if the image does not exist locally
  - Docker creates a new container
  - Docker allocates a read-write filesystem to the container
  - Docker creates a network interface to the container, assign an IP address
  - Docker starts the container (calls containerd then runc) and executes /bin/bash

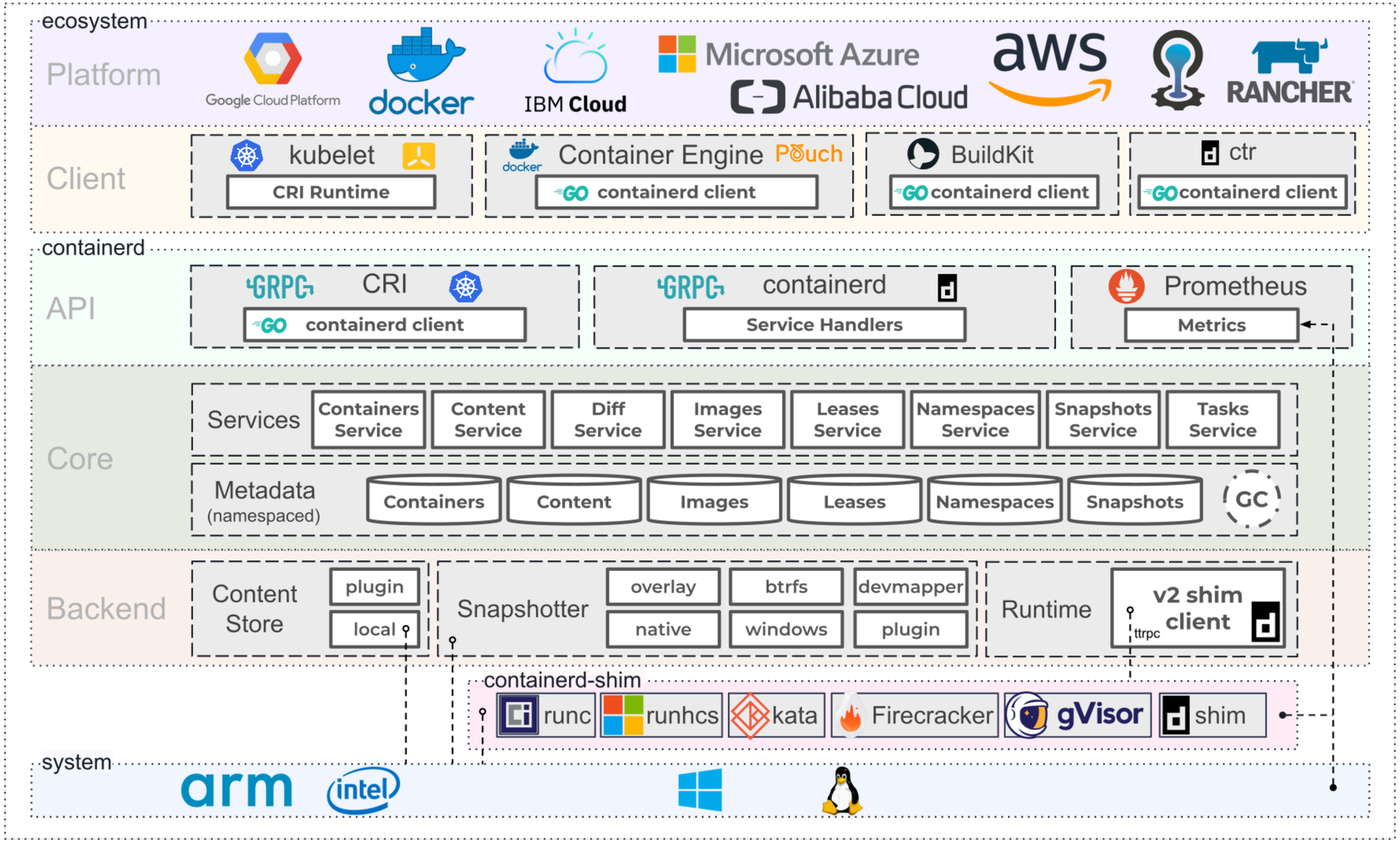
```
docker run -i -t ubuntu /bin/bash
```

Docker Containers

- Runs an ubuntu container, attaches to command line session, and runs /bin/sh
  - Docker pulls the ubuntu image from the registry if the image does not exist locally
  - Docker creates a new container
  - Docker allocates a read-write filesystem to the container
  - Docker creates a network interface to the container, assign an IP address
  - Docker starts the container (calls containerd then runc) and executes /bin/bash

```
docker run -i -t ubuntu /bin/bash
```

Container Ecosystem (from https://containerd.io)



Agenda

- Introduction of Containers
- **Mechanisms for Supporting Containers**

Kernel Mechanisms used by Docker Containers

- Docker containers build on Linux kernel features:
  - **Capabilities**
  - **seccomp**
  - Control Groups (cgroups)
  - Namespaces
  - Union Filesystems

Capabilities

- Capabilities turn the binary “root/non-root” dichotomy into a fine-grained access control system
  - Example: a process does not need full root privileges to bind a port; all it needs is to be granted with the “*net_bind_service*” capability!
- Linux capabilities were introduced in the v2.2 kernel
  - https://man7.org/linux/man-pages/man7/capabilities.7.html
- Each capability represents a distinct unit or a set of privilege(s)
  - Can be independently enabled and disabled
  - Capabilities are a per-thread attribute in Linux

Capabilities

- Linux capabilities (prefixed by CAP_) example:
  - CAP_CHOWN
  - CAP_SETUID
  - CAP_NET_ADMIN
  - CAP_SYS_ADMIN

```
CAP_SYS_ADMIN
       Note: this capability is overloaded; see Notes to kernel
       developers, below.

       * Perform a range of system administration operations
         including: quotactl(2), mount(2), umount(2),
         pivot_root(2), swapon(2), swapoff(2), sethostname(2),
         and setdomainname(2);
```

From: https://man7.org/linux/man-pages/man7/capabilities.7.html

Capabilities: Docker Containers

- The following Linux capability options are allowed by Docker by default and can be dropped by configuration

Capability Key	Capability Description
AUDIT_WRITE	Write records to kernel auditing log.
CHOWN	Make arbitrary changes to file UIDs and GIDs (see chown(2)).
DAC_OVERRIDE	Bypass file read, write, and execute permission checks.
FOWNER	Bypass permission checks on operations that normally require the file system UID of the process to match the UID of the file.
FSETID	Don't clear set-user-ID and set-group-ID permission bits when a file is modified.
KILL	Bypass permission checks for sending signals.
MKNOD	Create special files using mknod(2).
NET_BIND_SERVICE	Bind a socket to internet domain privileged ports (port numbers less than 1024).
NET_RAW	Use RAW and PACKET sockets.
SETFCAP	Set file capabilities.
SETGID	Make arbitrary manipulations of process GIDs and supplementary GID list.
SETPCAP	Modify process capabilities.
SETUID	Make arbitrary manipulations of process UIDs.
SYS_CHROOT	Use chroot(2), change root directory.

https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities

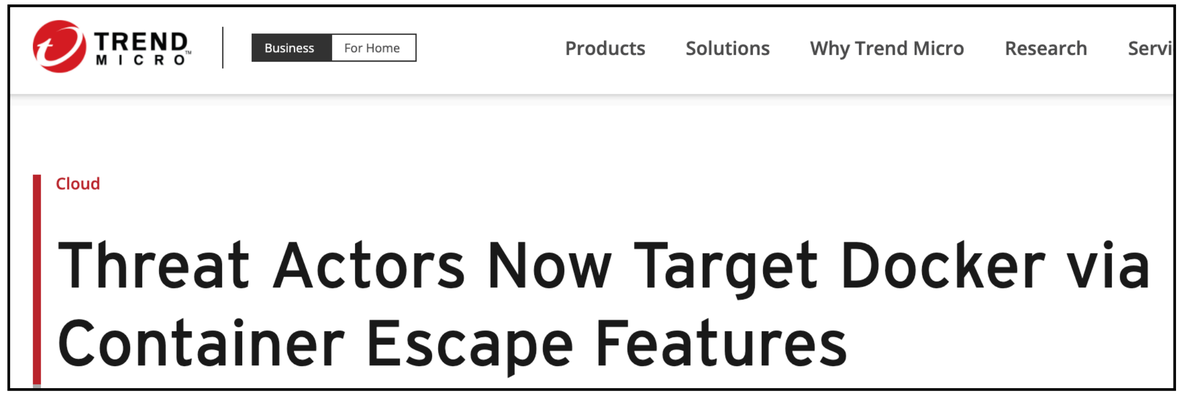
Capabilities: Docker Containers

- Check: https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities
- Use "*docker run --privileged*" to give all capabilities to the container
- Use "*--cap-add"* or *"--cap-drop*" flags to have fine-grained control over the capabilities given to the container
  - Example: "*docker run --cap-add=ALL --cap-drop=MKNOD ..*"

Seccomp

- Processes on the host and containers share the same OS kernel
  - Both can make system calls to the shared OS kernel
  - Attackers can make system calls to perform container escape exploits

Abusing Privileged and Unprivileged Linux Containers



Seccomp

- Seccomp is a Linux kernel feature called “secure computing mode”
  - Limit the number of system calls that a process is allowed to make
  - If non-permitted system calls were made, Linux kills the process via a SIGKILL signal
  - Modern browsers like Chrome and Firefox also employ seccomp to secure their executions

Seccomp

- Containers like Docker and LXC employ seccomp to restrict the system call interface for containers to those required to carry out its function
- Docker by default blocks a set of dangerous system calls:
  - Example: Docker disallows “reboot”, “ptrace”, etc.
  - Full list of syscalls blocked by Docker by default: https://docs.docker.com/engine/security/seccomp/

Seccomp

- In seccomp's most restrictive mode (SECCOMP_MODE_STRICT), the process cannot make any system calls other than read(), write(), exit(), and sigreturn() to *already-open* file descriptors

- Guess what's the output like?

```c
#include <linux/seccomp.h>
#include <sys/prctl.h>


int main(int argc, char **argv)
{
        int output = open("output.txt", O_WRONLY);
        const char *val = "test";

        printf("Calling prctl() to set seccomp strict mc
        prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);

        printf("Writing to an already open file...\n");
        write(output, val, strlen(val)+1);

        printf("Trying to open file for reading...\n");
        int input = open("output.txt", O_RDONLY);

        printf("You will not see this message--the proce
}
```

From: https://gist.github.com/mstemm/3e29df625052616fffcd667ff59bf32a#file-seccomp_strict-c

Seccomp

- In seccomp's most restrictive mode, the process cannot make any system calls other than read(), write(), exit(), and sigreturn() to *already-open* file descriptors
- Guess what's the output like?

```c
#include <linux/seccomp.h>
#include <sys/prctl.h>


int main(int argc, char **argv)
{
        int output = open("output.txt", O_WRONLY);
        const char *val = "test";

        printf("Calling prctl() to set seccomp strict mc
        prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);

        pr                                              n");
        wr

        printf("Trying to open file for reading...\n");
        int input = open("output.txt", O_RDONLY);

        printf("You will not see this message--the proce
}
```

The open is rejected, and the program is killed!

From: https://gist.github.com/mstemm/3e29df625052616fffcd667ff59bf32a#file-seccomp_strict-c

Seccomp

- prctl() also supports seccomp's filter mode (SECCOMP_MODE_FILTER): to fine-grained control what system calls can be made

```
void install_syscall_filter()
{
        struct sock_filter filter[] = {
                /* Validate architecture. */
                VALIDATE_ARCHITECTURE,
                /* Grab the system call number. */
                EXAMINE_SYSCALL,
                /* List allowed syscalls. We add open() to the set of
                   allowed syscalls by the strict policy, but not
                   close(). */
                ALLOW_SYSCALL(rt_sigreturn),
#ifdef __NR_sigreturn
                ALLOW_SYSCALL(sigreturn),
#endif
                ALLOW_SYSCALL(exit_group),
                ALLOW_SYSCALL(exit),
                ALLOW_SYSCALL(read),
                ALLOW_SYSCALL(write),
                ALLOW_SYSCALL(open),
                KILL_PROCESS,
        };
        struct sock_fprog prog = {
                .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
                .filter = filter,
        };

        assert(prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) == 0);

        assert(prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) == 0);
}
```

https://gist.github.com/mstemm/1bc06c52abb7b6b4feef79d7bfff5815

Kernel Mechanisms used by Docker Containers

- Docker containers build on Linux kernel features:
  - Capabilities
  - seccomp
  - **Control Groups (cgroups)**
  - **Namespaces**
  - Union Filesystems

Control Groups (cgroups)

- Linux kernel feature (merged to Linux v2.6.24 in 2008) that limits, accounts for, and isolates resource usage of a *single or a collection of processes*
  - ***Resource limiting:*** groups can be set to not exceed a configured processor usage, memory limit, and device usage
  - ***Accounting:*** monitor resource usage
  - and many more…

Control Groups (cgroups)

- Cgroups categorization from Red Hat's documentation (Linux also calls them *controllers*)
  - CPU related: cpu, cpuacct, cpuset
  - Memory related: memory, hugetlb
  - I/O related: blkio, devices, net_cls, net_prio
  - Misc: perf_event, freezer, etc.

Control Groups (cgroups)

- Use ***cgroup-tools*** to administer cgroups

套件：cgroup–tools（0.41–11）

control and monitor control groups (tools)

Control Groups (cgroups) provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.

libcgroup allows one to manipulate, control, administrate and monitor cgroups and the associated controllers.

Create memory controller cgroup

```
parallels@ubuntu-linux-20-04-desktop:~$ sudo cgcreate -g memory:test
parallels@ubuntu-linux-20-04-desktop:~$ sudo cgset -r memory.limit_in_bytes=1500K test
parallels@ubuntu-linux-20-04-desktop:~$ cgget -r memory.limit_in_bytes test
test:
memory.limit_in_bytes: 1536000

parallels@ubuntu-linux-20-04-desktop:~$ cat /sys/fs/cgroup/memory/test/memory.limit_in_bytes
1536000
```

Demo: cgroups

gcc mem.c -o memes

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main() {
    char *ptr;
    while(1) {
        ptr = (char *)malloc(4096);
        memset(ptr, 0, 4096);
    }
    return 0;
}
```

memes was killed by Linux's Out-of-Memory Killer

```
18823 Killed                  sudo cgexec -g memory:test ./memes
```

```
parallels@ubuntu-linux-20-04-desktop:~$ cat /sys/fs/cgroup/memory/test/memory.oom_control

oom_kill_disable 0
under_oom 0
oom_kill 2
parallels@ubuntu-linux-20-04-desktop:~$ ./cgroup.sh
1536000
./cgroup.sh: line 5: 19244 Killed                  sudo cgexec -g memory:test ./memes
parallels@ubuntu-linux-20-04-desktop:~$ cat /sys/fs/cgroup/memory/test/memory.oom_control

oom_kill_disable 0
under_oom 0
oom_kill 3
```

Put the process in the cgroup

Namespaces

- Linux kernel feature that partitions and isolates kernel resources
- Wrap global kernel resources in abstractions for a set of processes
    - Processes in the namespace think they have their own isolated copy of the global resources
- Added to Linux kernel between 2002 to 2006

Namespaces

- Linux 5.6 includes 8 types of namespaces:
  - Mount (mnt): isolate mount points (CLONE_NEWNS)
  - Process ID (pid): isolate processes (CLONE_NEWPID)
  - Network (net): isolate network stack (CLONE_NEWNET)
  - IPC: isolate IPC/message queues (CLONE_NEWIPC)
  - UTS: isolate hostnames (CLONE_NEWUTS)
  - User ID (uid): isolate user/group ID (CLONE_NEWUSER)
  - Control group (cgroup): isolate cgroup sysfs (CLONE_NEWCGROUP)
  - Time: isolates system time (CLONE_NEWTIME)

Namespaces

- Linux creates a single default namespace of each type
- Processes can create new namespaces and join the new namespaces
- Each process is associated with one namespace
  - A process only sees or uses the resources in the associated namespace
  - Namespaces are hierarchical: parent namespaces could observe descendant namespaces

Namespaces: system calls

- Three Linux system calls to manage namespaces
  - ***clone():*** creates a new process and namespace (specified by the flag); attaches the process to the new namespace
  - ***unshare():*** creates a new namespace and attaches the current process to it
  - ***setns():*** joins an existing namespace (specified by file descriptor)

Namespaces: struct nsproxy

```
struct task_struct {

 /* Namespaces: */
 struct nsproxy                  *nsproxy;

 };
```

```
/*
 * A structure to contain pointers to all per-process
 * namespaces - fs (mount), uts, network, sysvipc, etc.
 *
 * The pid namespace is an exception -- it's accessed using
 * task_active_pid_ns.  The pid namespace here is the
 * namespace that children will use.
 *
 * 'count' is the number of tasks holding a reference.
 * The count for each namespace, then, will be the number
 * of nsproxies pointing to it, not the number of tasks.
 *
 * The nsproxy is shared by tasks which share all namespaces.
 * As soon as a single namespace is cloned or unshared, the
 * nsproxy is copied.
 */
```

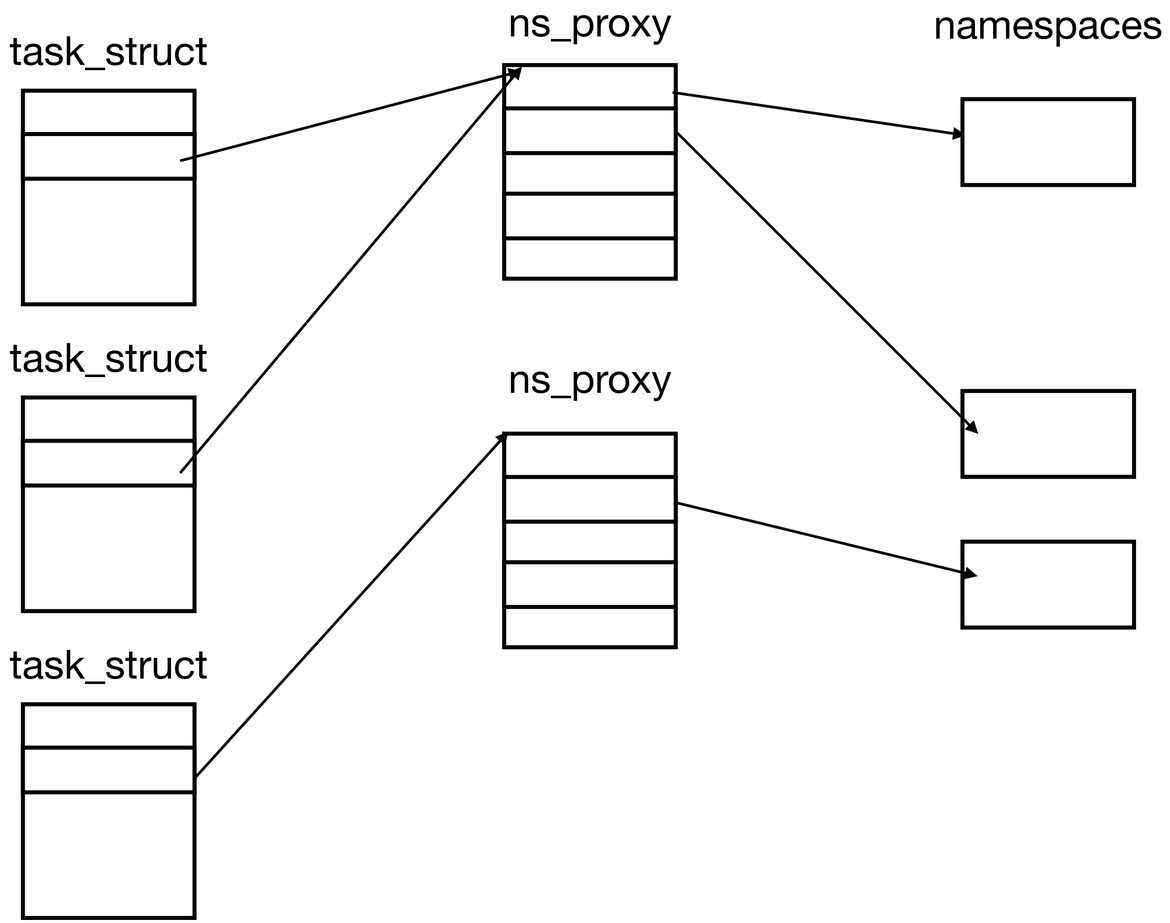
```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
        struct time_namespace *time_ns;
        struct time_namespace *time_ns_for_children;
        struct cgroup_namespace *cgroup_ns;
};
```

Namespaces: struct nsproxy

- A C struct named *nsproxy* was added to Linux's process descriptor: "struct task_struct"
  - The instance *init_nsproxy* specifies the default namespace for each namespace type

```
struct nsproxy init_nsproxy = {
        .count                  = ATOMIC_INIT(1),
        .uts_ns                 = &init_uts_ns,
#if defined(CONFIG_POSIX_MQUEUE) || defined(CONFIG_SYSVIPC)
        .ipc_ns                 = &init_ipc_ns,
#endif
        .mnt_ns                 = NULL,
        .pid_ns_for_children    = &init_pid_ns,
#ifdef CONFIG_NET
        .net_ns                 = &init_net,
#endif
#ifdef CONFIG_CGROUPS
        .cgroup_ns              = &init_cgroup_ns,
#endif
#ifdef CONFIG_TIME_NS
        .time_ns                = &init_time_ns,
        .time_ns_for_children   = &init_time_ns,
#endif
};
```

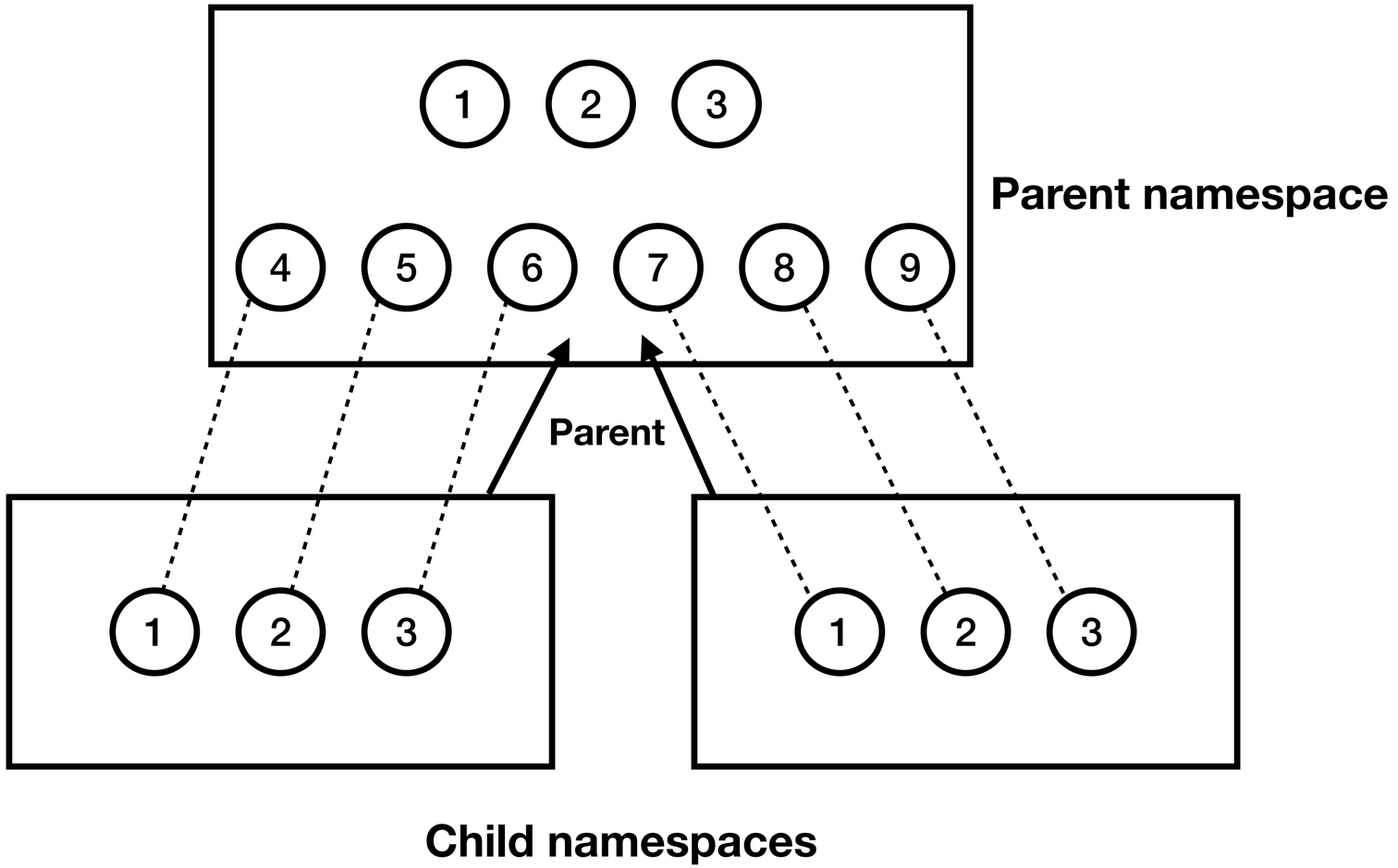
Namespaces: struct nsproxy



PID namespace

- From a given PID namespace, a process could only see those processes contained the namespace or *descendent* namespaces
  - The process cannot see the PIDs from the parent — *getppid()* returns 0!
  - PIDs of processes from the child namespace are visible to the parent namespace
- Processes in different PID namespaces could share the same process ID

PID namespace



PID namespace

- The process IDs from a PID namespace are unique and start with PID 1
  - Can be leveraged by containers to support migration; why?
  - First process who created in the new namespace has PID 1 (Init process)
- The “Init process” in the PID namespace performs management tasks:
  - Change orphan child processes’ PID to the init process (child reaping)
  - Cannot be killed by SIGKILL by any of the namespaces (either the initial or new)

Namespaces: system calls

```
int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...
          /* pid_t *parent_tid, void *tls, pid_t *child_tid */ );
```

CLONE_NEWPID (since Linux 2.6.24)
If **CLONE_NEWPID** is set, then create the process in a new PID namespace. If this flag is not set, then (as with fork(2)) the process is created in the same PID namespace as the calling process.

For further information on PID namespaces, see namespaces(7) and pid_namespaces(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWPID**. This flag can't be specified in conjunction with **CLONE_THREAD** or **CLONE_PARENT**.

PID namespace

```c
#define _GNU_SOURCE
#include <linux/sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

static char child_stack[40960];

static int child_fn() {
  printf("PID: %ld\n", (long)getpid());
  printf("PPID: %ld\n", (long)getppid());
  return 0;
}

int main() {
  pid_t child_pid = clone(child_fn, child_stack+40960, CLONE_NEWPID | SIGCHLD, NULL);
  printf("clone() = %ld errno %d\n", (long)child_pid, errno);

  waitpid(child_pid, NULL, 0);
  return 0;
}
```

What is the PID and PPID printed?

https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

PID namespace

```c
#define _GNU_SOURCE
#include <linux/sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

static char child_stack[40960];

static int child_fn() {
  printf("PID: %ld\n", (long)getpid());
  printf("PPID: %ld\n", (long)getppid());
  return 0;
}

int main() {
  pid_t child_pid = clone(child_fn, child_stack+40960, CLONE_NEWPID | SIGCHLD, NULL);
  printf("clone() = %ld errno %d\n", (long)child_pid, errno);

  waitpid(child_pid, NULL, 0);
  return 0;
}
```

What is the PID and PPID printed?

```
ubuntu@ubuntu:~$ sudo ./a.out
clone() = 43096 errno 0
PID: 1
PPID: 0
```

https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

UTS namespace

- Provides namespace-specific hostnames
- Implementation in Linux is straightforward
  - Added a new “struct new_utsname” to store hostname

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
        struct time_namespace *time_ns;
        struct time_namespace *time_ns_for_children;
        struct cgroup_namespace *cgroup_ns;
};
```

```
struct new_utsname {
        char sysname[__NEW_UTS_LEN + 1];
        char nodename[__NEW_UTS_LEN + 1];
        char release[__NEW_UTS_LEN + 1];
        char version[__NEW_UTS_LEN + 1];
        char machine[__NEW_UTS_LEN + 1];
        char domainname[__NEW_UTS_LEN + 1];
};
```

UTS namespace

```
int gethostname(char *name, size_t len);
```

gethostname() returns the null-terminated hostname in the character array *name*, which has a length of *len* bytes. If the null-terminated hostname is too large to fit, then the name is truncated, and no error is returned (but see NOTES below). POSIX.1 says that if such truncation occurs, then it is unspecified whether the returned buffer includes a terminating null byte.

These system calls are used to access or to change the system hostname. More precisely, they operate on the hostname associated with the calling process's UTS namespace.

UTS namespace

- Implementation of gethostname in Linux v5.15.6

```
static inline struct new_utsname *utsname(void)
{
        return &current->nsproxy->uts_ns->name;
}
```

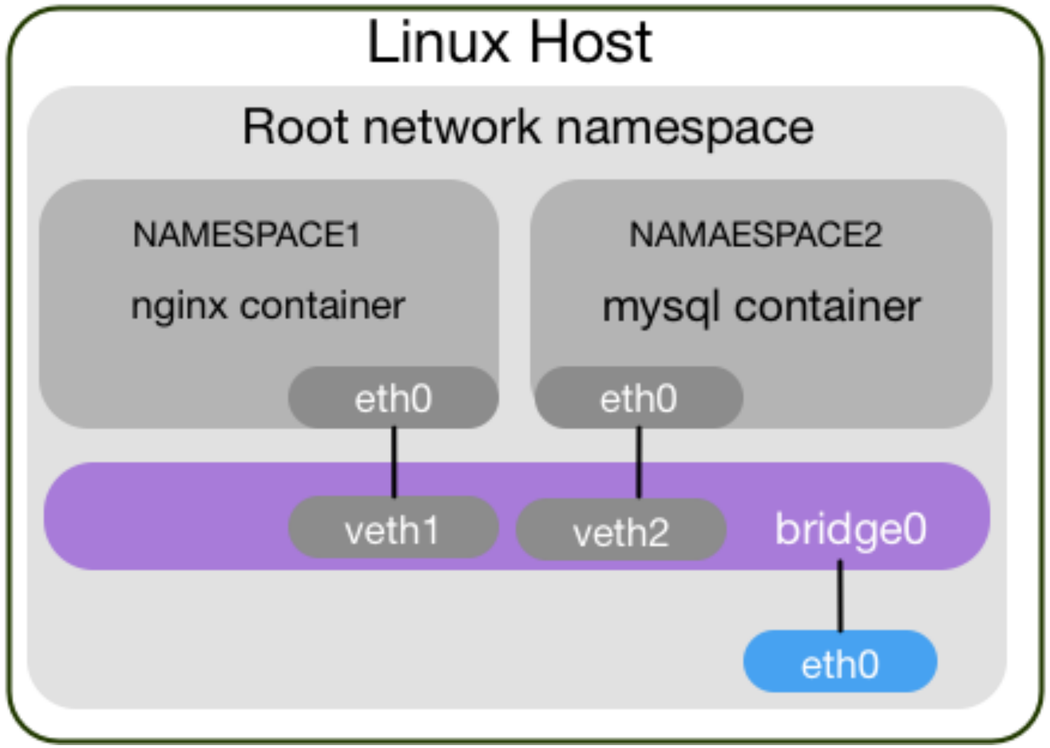
```
SYSCALL_DEFINE2(gethostname, char __user *, name, int, len)
{
        int i;
        struct new_utsname *u;
        char tmp[__NEW_UTS_LEN + 1];

        if (len < 0)
                return -EINVAL;
        down_read(&uts_sem);
        u = utsname();
        i = 1 + strlen(u->nodename);
        if (i > len)
                i = len;
        memcpy(tmp, u->nodename, i);
        up_read(&uts_sem);
        if (copy_to_user(name, tmp, i))
                return -EFAULT;
        return 0;
}
```

Network namespace

- Container tools set up virtual network interfaces for namespaces
- Routing must be done in the global network namespace to access the physical network interface — multiplex traffic for different namespaces
- Example: Docker creates bridge interface for containers running in different namespaces and routes packets for these containers

Network namespace



https://logingood.github.io/images/NS-intro.png

User namespace

- A Linux process has a distinct set of security-related IDs and attributes
  - User IDs (UIDs) and Group IDs (GIDs)
- Processes within the user namespace can have IDs and privileges different than are permitted outside
- Used to virtualize privileges for containers: a normal user acts as root (ID 0) inside a container without being root on the host outside

User namespace: Case Study

Create a new user namespace, map “root” in the namespace to the user “ubuntu” outside

```
ubuntu@ubuntu:~$ unshare --user -r /bin/bash
root@ubuntu:~#
```

```
root@ubuntu:~# readlink /proc/$$/ns/user
user:[4026532631]
```

https://manpages.ubuntu.com/manpages/jammy/man1/unshare.1.html

The root user has full capability, allowing it to create more namespaces

```
root@ubuntu:~# cat /proc/$$/status | egrep 'Cap'
CapInh: 0000000000000000
CapPrm: 000001ffffffffff
CapEff: 000001ffffffffff
CapBnd: 000001ffffffffff
CapAmb: 0000000000000000
```

The root user creates a new uts namespace

```
root@ubuntu:~# readlink /proc/$$/ns/uts
uts:[4026531838]
root@ubuntu:~# unshare --uts /bin/bash
root@ubuntu:~# readlink /proc/$$/ns/uts
uts:[4026532641]
```

User namespace: Case Study

The “root” inside the user namespace cannot change resources owned by the “real root”, WHY?

```
root@ubuntu:~# echo $$
32734
root@ubuntu:~# hostname hello
hostname: you must be root to change the host name
root@ubuntu:~# ls /root/
ls: cannot open directory '/root/': Permission denied
root@ubuntu:~#
```

User namespace: Case Study

The “root” inside the user namespace cannot change resources owned by the “real root”, WHY?

```
root@ubuntu:~# echo $$
32734
root@ubuntu:~# hostname hello
hostname: you must be root to change the host name
root@ubuntu:~# ls /root/
ls: cannot open directory '/root/': Permission denied
root@ubuntu:~#
```

Because the root in the namespace corresponds to the user ubuntu!

Namespace: Container Case Study

- Namespace is essential for containers
- Consider a web server runs in the container (with namespaces):
  - The server has its own view of hostname, the process tree, users, etc.
  - Much harder for a malicious process spawned from the web server to affect other processes on the host

Cgroups v.s. Namespaces

- Cgroups provide an interface to limit and control resource utilization
- Namespaces provide a mechanism to limit the resource view
- Cgroups and namespaces are not dependent!
  - You can build cgroups *without* kernel support for namespaces

Kernel Mechanisms used by Docker Containers

- Docker containers build on Linux kernel features:
  - Capabilities
  - seccomp
  - Control Groups (cgroups)
  - Namespaces
  - **Union Filesystems**

Union Filesystem

- Docker images are large (~100Mb)
  - Do not want to allocate storage space for the images each time when creating a container
  - Long init time if full image copy is required for container creation

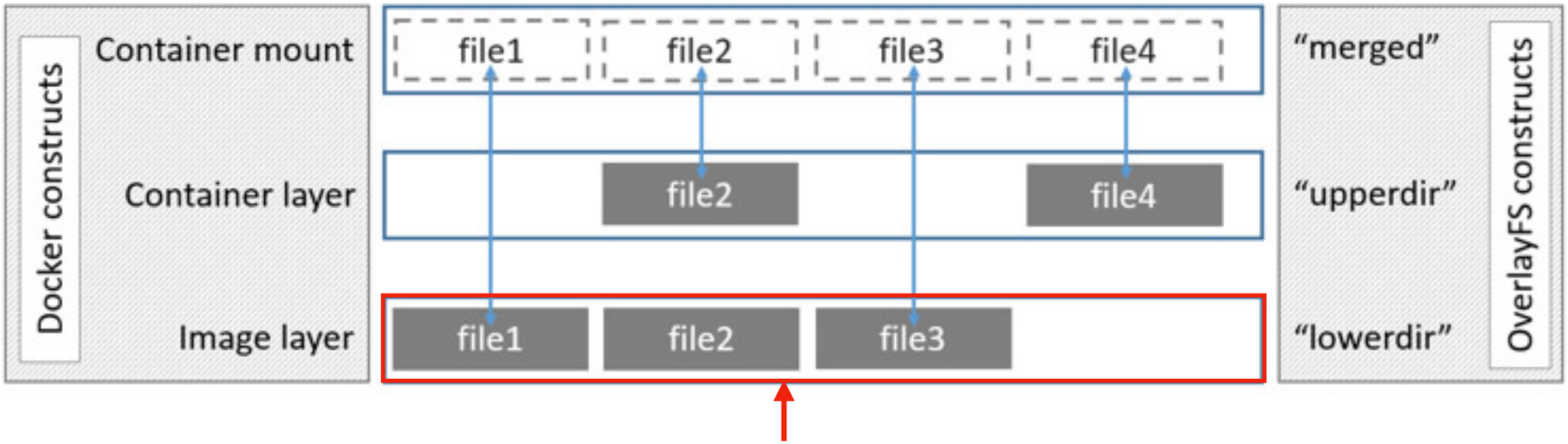
Union Filesystem

- Union filesystem supports merging of different file system contents to provide a single merged view
- Docker uses the “OverlayFS” union filesystem implementation in Linux
  - OverlayFS was merged to mainline in Linux v3.18

Union Filesystem

- OverlayFS merges two layers to provide a merged view
  - A lower layer: read-only directory (lowerdir)
  - A higher layer: read-write directory (upperdir)
- OverlayFS employs a copy-on-write (CoW) approach:
  - The lowerdir contains shared but read-only files
  - For file writes, the OverlayFS first copies files from lowerdir to upperdir then allows writes to the same files in upperdir

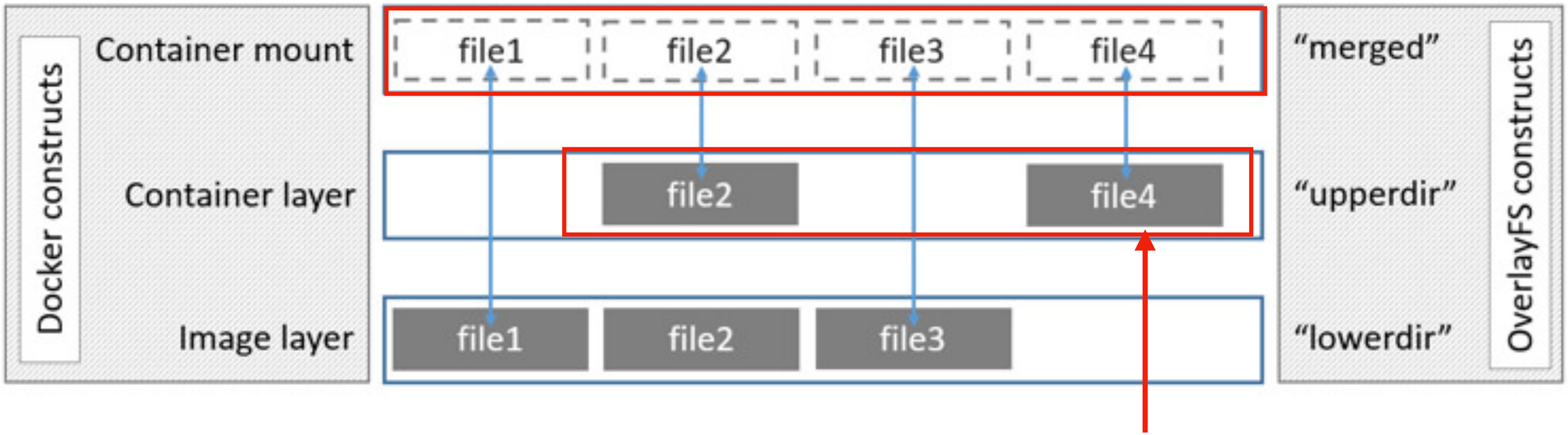
Union Filesystem



Container image pulled from the registry, RO

Union Filesystem

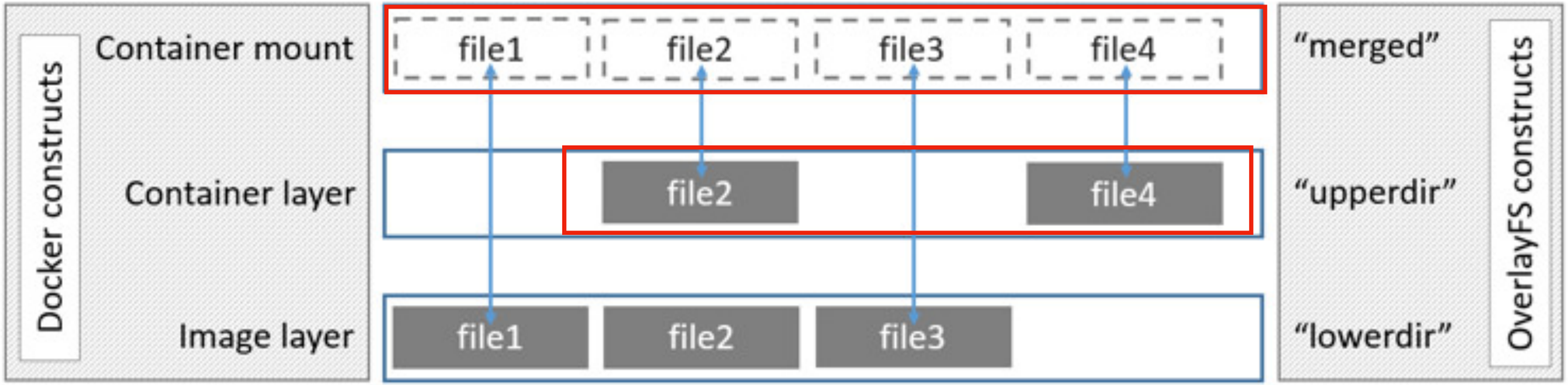
UnionFS resents the merged view to container



Files updated by the container

Union Filesystem

UnionFS resents the merged view to container



Question: How to support file deletion?