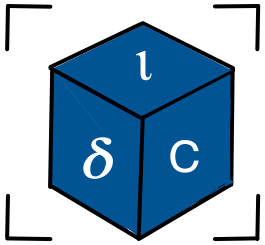
Resource Virtualization with Containers

Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi



PID Namespaces in Docker

- docker run –d –name server1 busybox sh –c “nc -l –p 0.0.0.0:7070”
- docker run –d –name server2 busybox sh –c “nc -l –p 0.0.0.0:8080”

OR

- docker run –t –d busybox sh
- docker run –t –d busybox sh
- docker ps -a

- docker exec server1 ps –ef

Will see two processes

- docker exec server2 ps –ef

Will see two processes

- ps –ef

<Stop and delete server1>

- docker run –d **–pid host** –name server1 busybox sh –c “nc -1 –p 0.0.0.0:7070”
- docker exec server1 ps –ef

/proc from the host is mounted

User Namespace

- Allow per-namespace mappings of user and group IDs.
  - A process's user and group IDs can be different inside and outside a user namespace.
- A process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace.
  - This means that the process has full root privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

Creating User Namespace

```
child_pid = clone(childFunc, child_stack +
STACK_SIZE, CLONE_NEWUSER | SIGCHLD, argv[1]);
```

- Unshare(`CLONE_NEWUSER`)

- No privilege is required to create a user namespace.

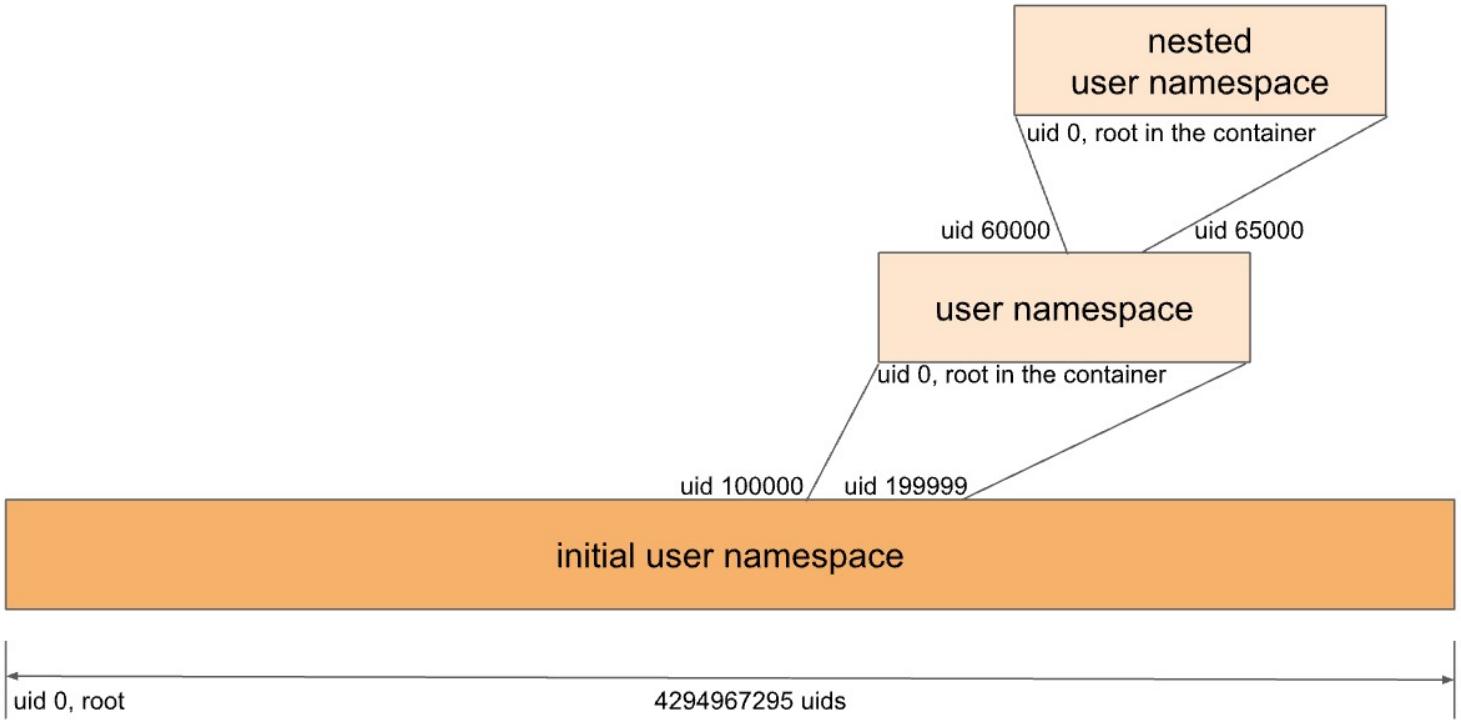
UID and GID Mappings

- Records written to/read from /proc/*PID*/uid_map and /proc/*PID*/gid_map have this form:
  - ID-inside-ns ID-outside-ns length
- *ID-inside-ns* and *length* define range of IDs inside user NS that are to be mapped
- *ID-outside-ns* defines start of corresponding mapped range in "outside" user NS
- E.g., following says that IDs 0...9 inside user NS map to IDs 1000...1009 in outside user NS
  - 0 1000 10

Who sets the mapping?

- Parent process sets the mapping of child process by writing two files available via /proc
  - /proc/*PID*/uid_map and /proc/*PID*/gid_map

User namespaces can be nested



Example

- **./demo_userns x**

- Determine PID of cloned child
- **ps -C demo_userns -o 'pid uid comm’**

- **echo '0 1000 1' > /proc/4713/uid_map**

- user ID 1000 in the parent user namespace (earlier mapped to 65534) has been mapped to user ID 0 in the user namespace created by demo_userns.

"Root privileges inside a user NS”

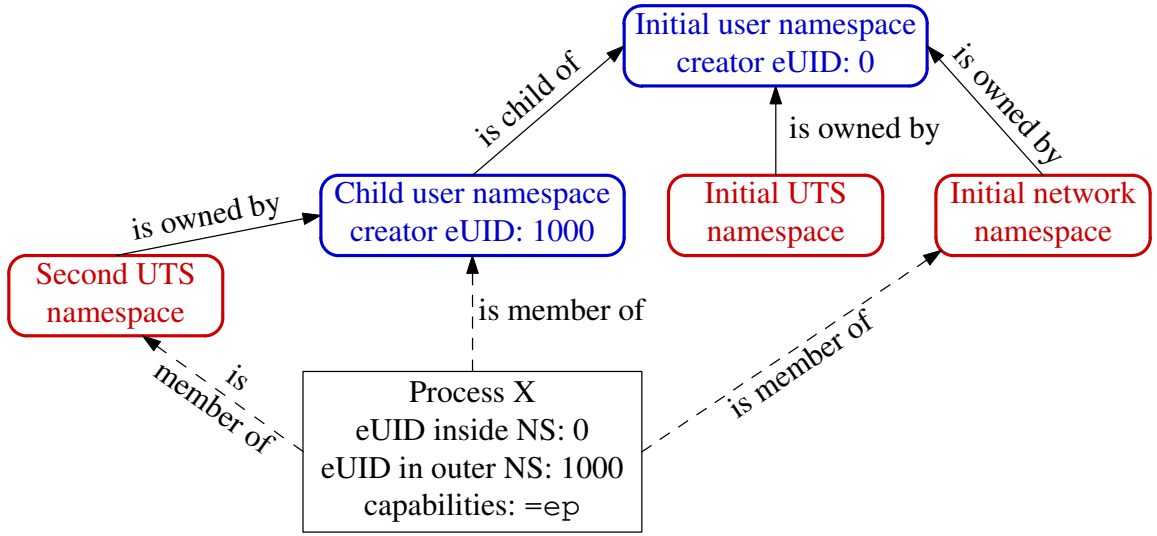
- What does “root privileges in a user NS” mean?

- There are a number of NS types
- Each NS type governs some global resource(s); e.g.:
  - UTS: hostname, NIS domain name
  - Mount: set of mount point
  - Network: IP routing tables, port numbers, /proc/net, ...

- There is an ownership relationship between user NSs and non-user NSs such that **each non-user NS is “owned” by a particular user NS**
  - When creating a new nonuser NS, kernel marks that NS as owned by the **user NS of process creating the new NS**
- If a process operates on resources governed by nonuser NS:
  - Permission checks are done according to **process’s capabiliites in user NS that owns the nonuser NS that governs the resources**

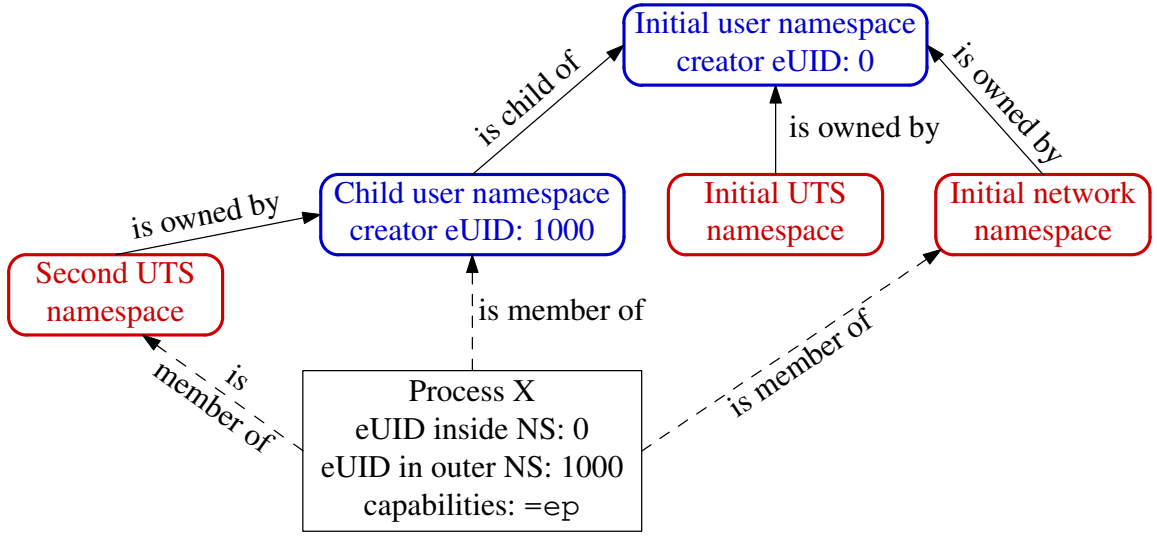
User namespaces “govern” other namespace types

- X is created with Unshare –Ur –u <prog>
- X is in new user NS, with root mappings and has all capabilities
- X is in a new UTS NS, which is owned by new user NS
- X is in initial instance of all other NS types (e.g network NS)



Changing hostname

- Suppose X tries to change hostname (CAP_SYS_ADMIN)
- X is in second UTS NS
- Permissions checked according to X's capabilities in user NS that owns that UTS NS => succeeds (X has capabilities in that user NS)



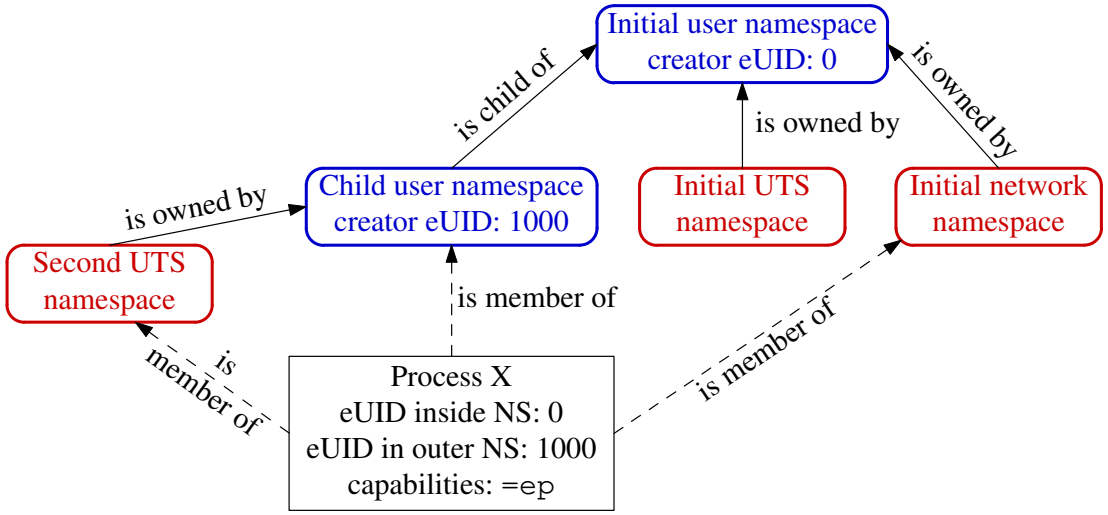
Changing hostname

- Suppose X tries to bind to reserved socket port (CAP_NET_BIND_SERVICE)
- X is in initial NET NS
- Permissions checked according to X's

capabilities in user NS that owns

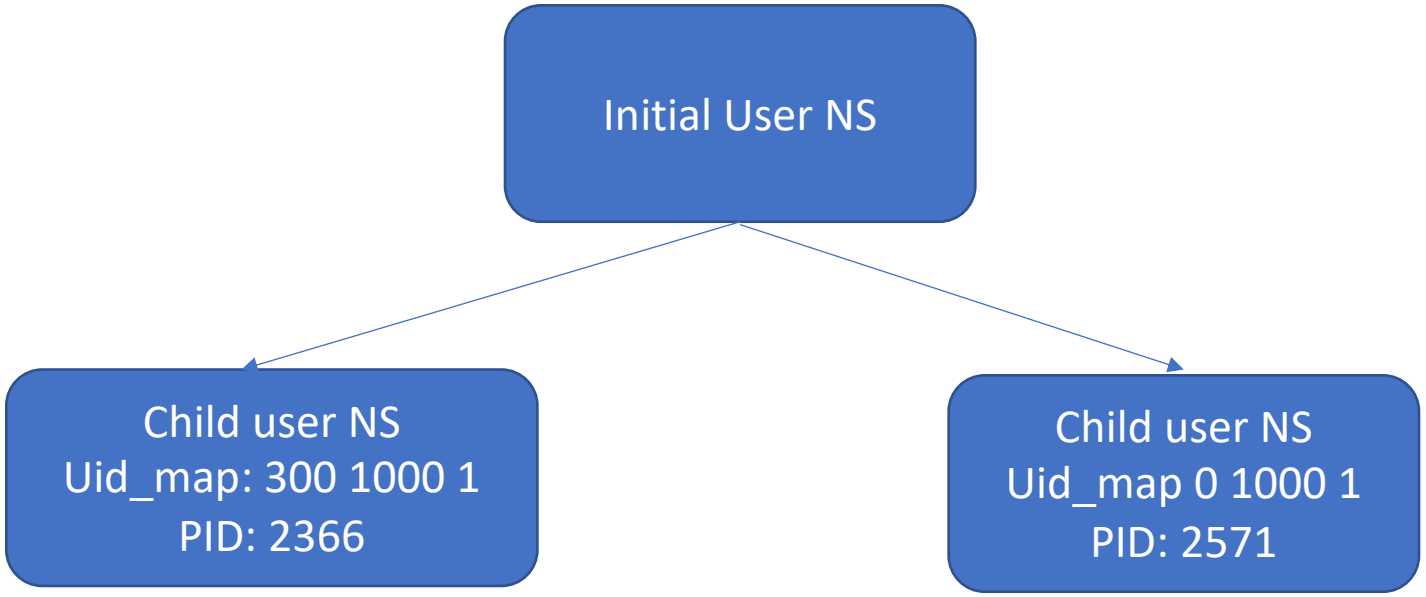
that network NS => fails

(X has no capabilities in initial user NS)

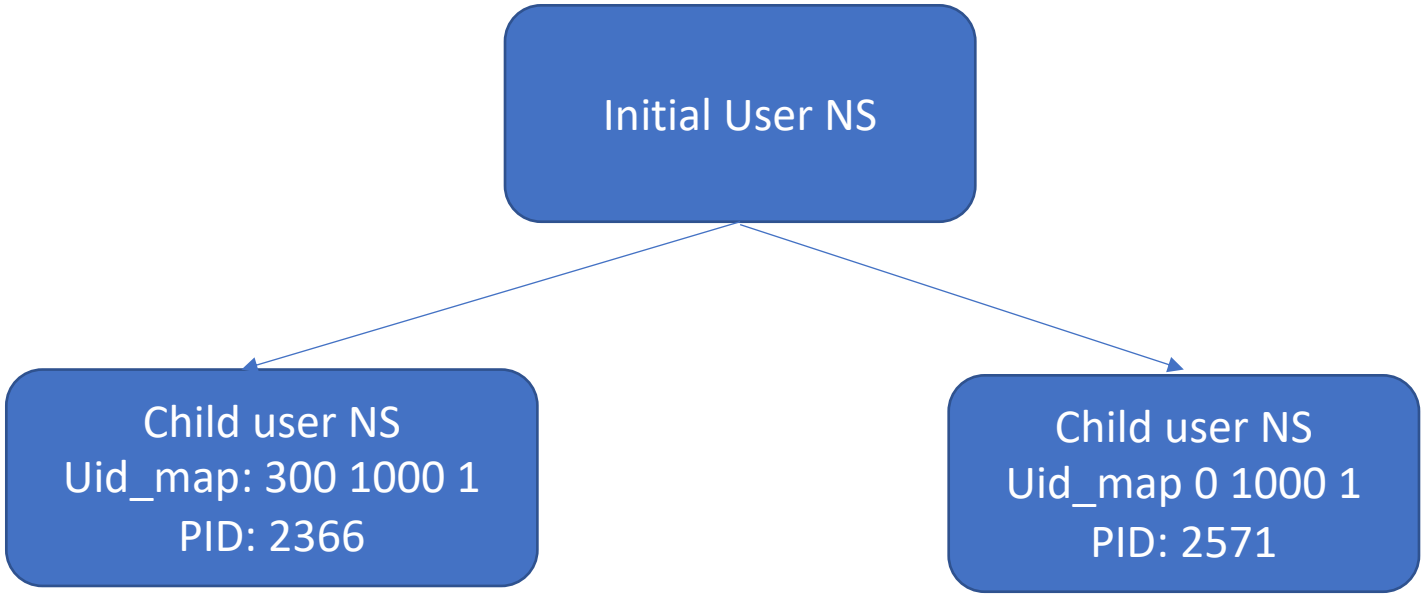


Interpretation of ID-outside-ns

- Interpretation of *ID-outside-ns depends on whether process opening uid_map/gid_map is in the same*
- If “opener” and *PID* are in **same user NS**:
  - *ID-outside-ns* interpreted as **ID in parent user NS** of *PID*
  - Common case: process is writing its own mapping file
- If “opener” and *PID* are in **different user NSs**:
  - *ID-outside-ns* interpreted as **ID in opener’s user NS**
  - Equivalent to previous case, if “opener” is (parent) process that created user NS using *clone()*
- (Above rules make sense, when we consider how these two cases could be rationally conceived)



- If PID 2366 reads /proc/2571/uid_map, what should it see? 0 300 1
- If PID 2571 reads /proc/2366/uid_map, what should it see? 300 0 1

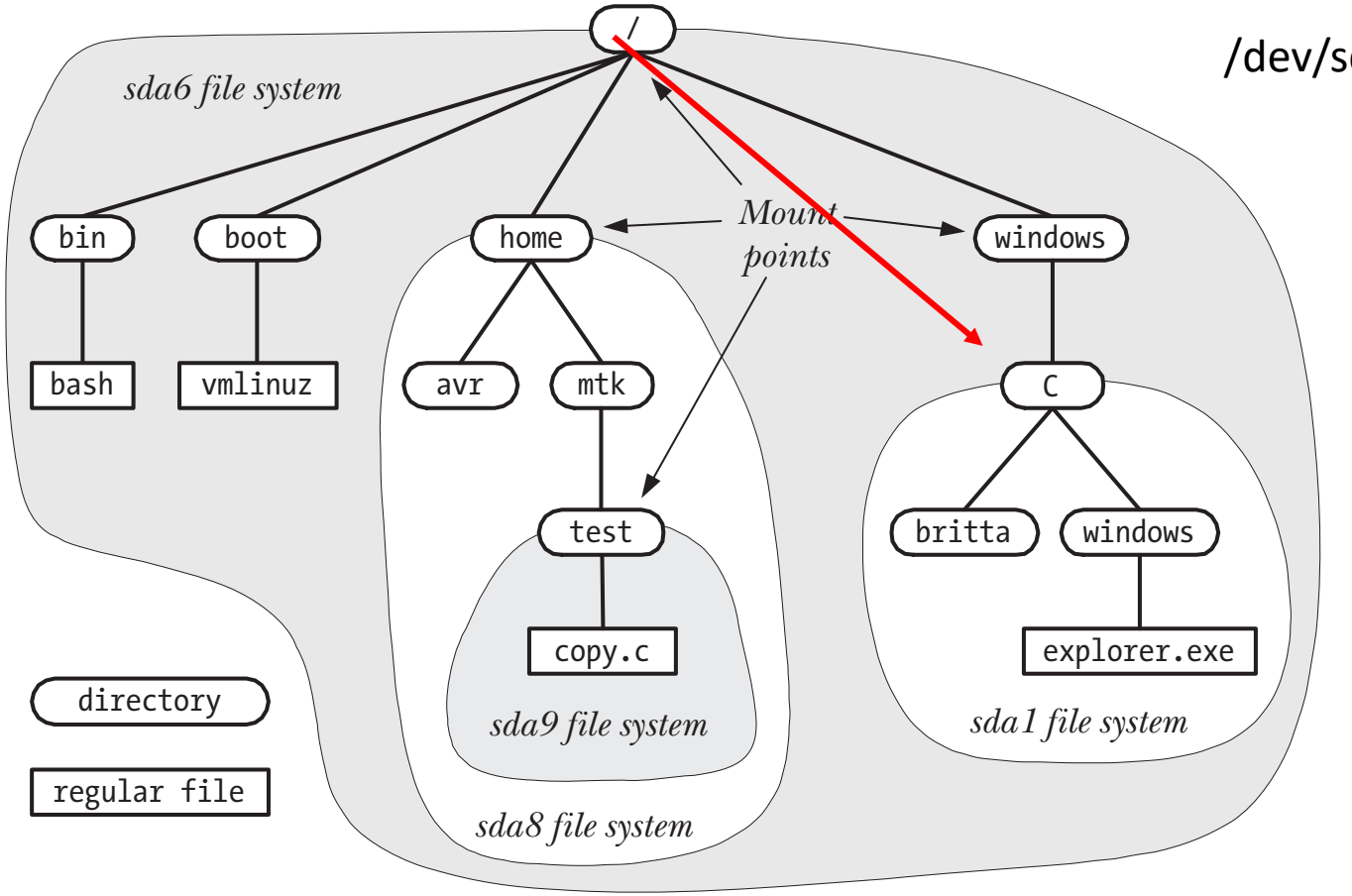


- If PID 2366 reads /proc/2571/uid_map, what should it see? 0 300 1
- If PID 2571 reads /proc/2366/uid_map, what should it see? 300 0 1

Mounting a filesystem

- On Linux, as on other UNIX systems, all files from all file systems reside under a single **directory** tree.
- Root of this tree is the root directory / (slash)
- Other storage devices/file systems are *mounted* under the root directory and appear as subtrees within the overall hierarchy
- The superuser uses a command of the following form to mount a device/file system at the specified directory:
- $ **mount *device directory***
  - The directory becomes a mount point
- A file system must be mounted before it can be used by the operating system

Mount points



- To list the currently mounted file systems, we can use the command *mount* with no arguments

$ **mount**
/dev/sda6 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda1 on /windows/C type vfat (rw,noexec,nosuid,nodev)
/dev/sda9 on /home/mtk/test type reiserfs (rw)

Mount() system call

- #include <sys/mount.h>
  int **mount**(const char **source*, const char **target*, const char **fstype*,
- unsigned long *mountflags*, const void **data*);
  Returns 0 on success, or –1 on error

- *Source* specifies the file system contained on the device
- *Target* specifies the directory (the *mount point*)
- *fstype* argument is a string identifying the type of file system
- *mountflags* argument is a bit mask that modify the operation of *mount()*.

Unmount() system call

- The umount() system call unmounts a mounted file system.
- int umount(const char *target);
- The target argument specifies the mount point of the file system to be unmounted.
- It is not possible to unmount a file system that is busy; that is, if there are open files on the file system, or a process’s current working directory is somewhere in the file system.

Maintaining mount information

- The *mount(8)* and *umount(8)* **commands** automatically maintain the file /etc/mtab
  - Similar to /proc/mounts but more detailed
  - Corresponding system calls do not maintain this file. The developer must write to them.
- Format: /dev/sda9 /boot ext3 rw 0 0
- This line contains six fields:
  - The name of the mounted device.
  - The mount point for the device.
  - The file-system type.
  - Mount flags. In the above example, *rw* indicates that the file system was mounted read-write.
  - A number used to control the operation of file-system backups by *dump(8)*. This field and the next are used only in the /etc/fstab file; for /proc/mounts and /etc/mtab, these fields are always 0.
  - A number used to control the order in which *fsck(8)* checks file systems at system boot time.

Mounting a filesystem at multiple mount points

- A file system can be mounted at multiple locations within the file system.
- Because each of the mount points shows the same subtree, changes made via one mount point are visible through the other(s)
- **# mkdir /testfs**
  # mkdir /demo
  # mount /dev/sda12 /testfs
  # mount /dev/sda12 /demo
  # mount | grep sda12
  /dev/sda12 on /testfs type ext3 (rw)

  /dev/sda12 on /demo type ext3 (rw)
  # touch /testfs/myfile
  # ls /demo

myfile

Stacking multiple mount points at the same mount point

- Multiple mounts to be stacked on a single mount point.
  - Each new mount hides the directory subtree previously visible at that mount point.
  - When the mount at the top of the stack is unmounted, the previously hidden mount becomes visible once more

mount /dev/sda12 /testfs *Create first mount on* /testfs
touch /testfs/myfile *Make a file in this subtree*

mount /dev/sda13 /testfs *Stack a second mount on* /testfs
mount | grep testfs *Verify the setup*

touch /testfs/newfile *Create a file in this subtree*

ls /testfs *View files in this subtree*
newfile
umount /testfs *Pop a mount from the stack*

mount | grep testfs *Now only one mount on* /testfs
/dev/sda12 on /testfs type ext3 (rw)

ls /testfs *Previous mount is now visible*
lost+found myfile

Bind mounts

- A bind mount (using MS_BIND flag) allows a directory or a file to be mounted at some other location in the file-system hierarchy.
- This results in the directory or file being visible in both locations.
- A bind mount is somewhat like a hard link, but differs in two respects:
  - A bind mount can cross file-system mount points (and even chroot jails).
  - It is possible to make a bind mount for a directory.

 # mkdir d1

 # touch d1/x

 # mkdir d2

 # mount –bind d1 d2

 # ls d2

x

 # touch d2/y

 # ls d1

x y

Bind mounts on a file

- # **cat > f1** *Create file to be bound to another location*

  Chance is always powerful. Let your hook be always cast.
  Type Control-D

  # **touch f2** *This is the new mount point*
  # **mount --bind f1 f2** *Bind* f1 *as* f2
  # **mount | egrep '(d1|f1)'** *See how mount points look*
  /testfs/d1 on /testfs/d2 type none (rw,bind)
  /testfs/f1 on /testfs/f2 type none (rw,bind)
  # **cat >> f2** *Change* f2
  In the pool where you least expect it, will be a fish.
  # **cat f1** *The change is visible via original file* f1

  Chance is always powerful. Let your hook be always cast.
  In the pool where you least expect it, will be a fish.
  # **rm f2** *Can't do this because it is a mount point*

  rm: cannot unlink `f2': Device or resource busy
  # **umount f2** *So unmount*
  # **rm f2** *Now we can remove* f2

Why bind mounts?

- creation of a *chroot* jail.
- Rather than replicating various standard directories (such as /lib) in the jail, we can simply create bind mounts for these directories within the jail.
- These directories should possibly be mounted read-only

Recursive Bind Mounts

- Recursive bind mount: submounts under the source directory are replicated under mount target.
- How: MS_REC flag ORed with MS_BIND
- Example:

```
# mkdir top
# mkdir src1
# touch src1/aaa
# mount --bind src1 top
# mkdir top/sub
# mkdir src2
# touch src2/bbb
# mount --bind src2 top/sub
# find top
```

Non-recursive

```
# mkdir dir1
# mount --bind top dir1
# find dir1
```

Recursive

```
# mkdir dir2
# mount --rbind top dir2
# find dir2
```

Mount Moves

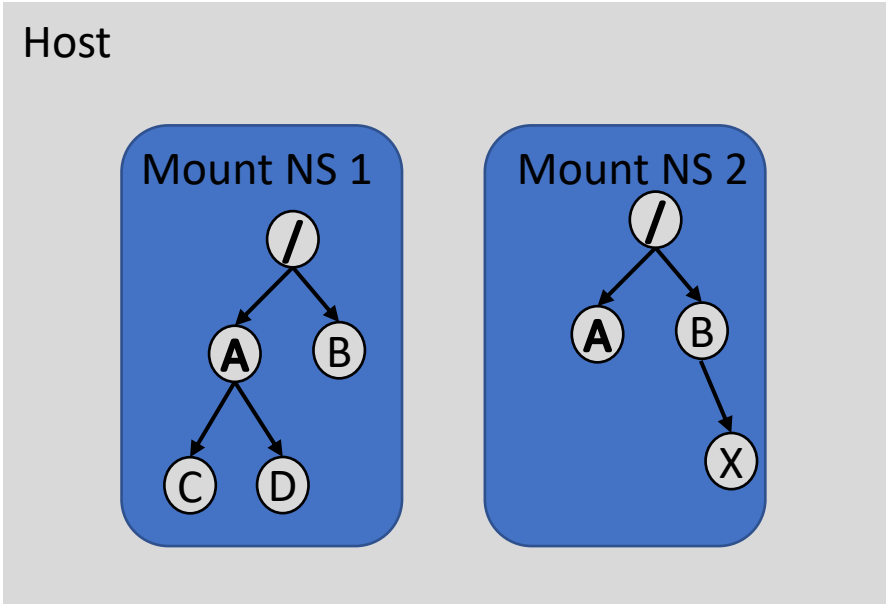
- move a subtree: *source* specifies an existing mount and *target* specifies the new location to which that mount is to be relocated.
- The move is atomic: at no point is the subtree unmounted.

Mount Namespace

- isolate the set of **filesystem mount points** seen by a group of processes.
- processes in different mount namespaces can have different views of the filesystem hierarchy.
- Use: Create environments that are similar to chroot jails but more secure.

Mount namespaces

- Creating a separate mount namespace allows each isolated process to have a completely different view of the entire system's mountpoint structure from the original one.
- Allows a different root to be specified for each group of isolated processes



Mount Namespace

- CLONE_NEWNS puts cloned process in new mount namespace OR make the calling process enter the new namespace.
- Child process can unmount/mount filesystems without affecting anything outside
- Can setup an entirely new filesystem for container
- newly created namespace **initially** receives all mount points replicated from the caller's namespace.
- Later mount points?
  - Depends on mount propagation types

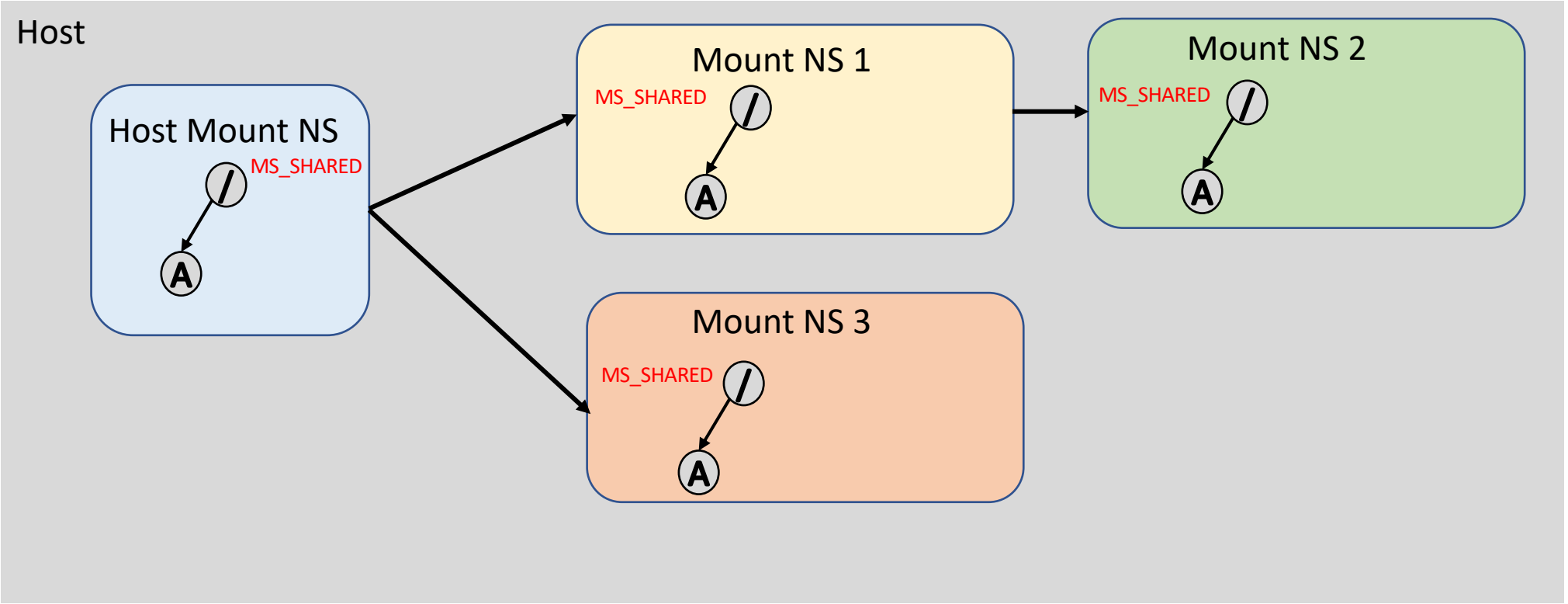
MS_SHARED

- MS_SHARED: When changes are made **under** a mount point of this type in one namespace, the change will be propagated to other namespaces in the same peer group.

Peer group

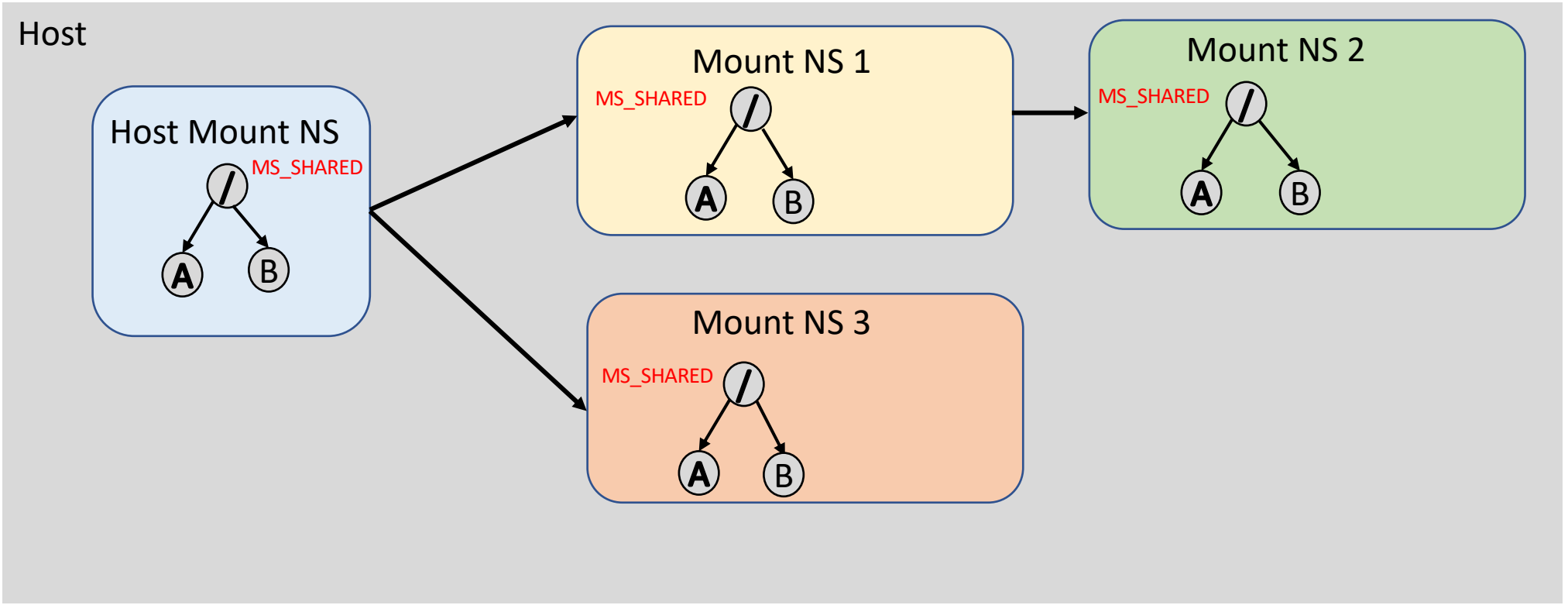
- A peer group is a set of mount points that propagate mount and unmount events to one another.
- A peer group acquires new members when
  - a mount point whose propagation type is shared is either replicated during the creation of a new namespace
  - A mount point is used as the source for a bind mount.
- In both cases, the new mount point is made a member of the same peer group as the existing mount point.
- A mount point ceases to be a member of a peer group when
  - it is unmounted, either explicitly, or
  - implicitly when a mount namespace is torn down because the last member process terminates or moves to another namespace.

MS_SHARED

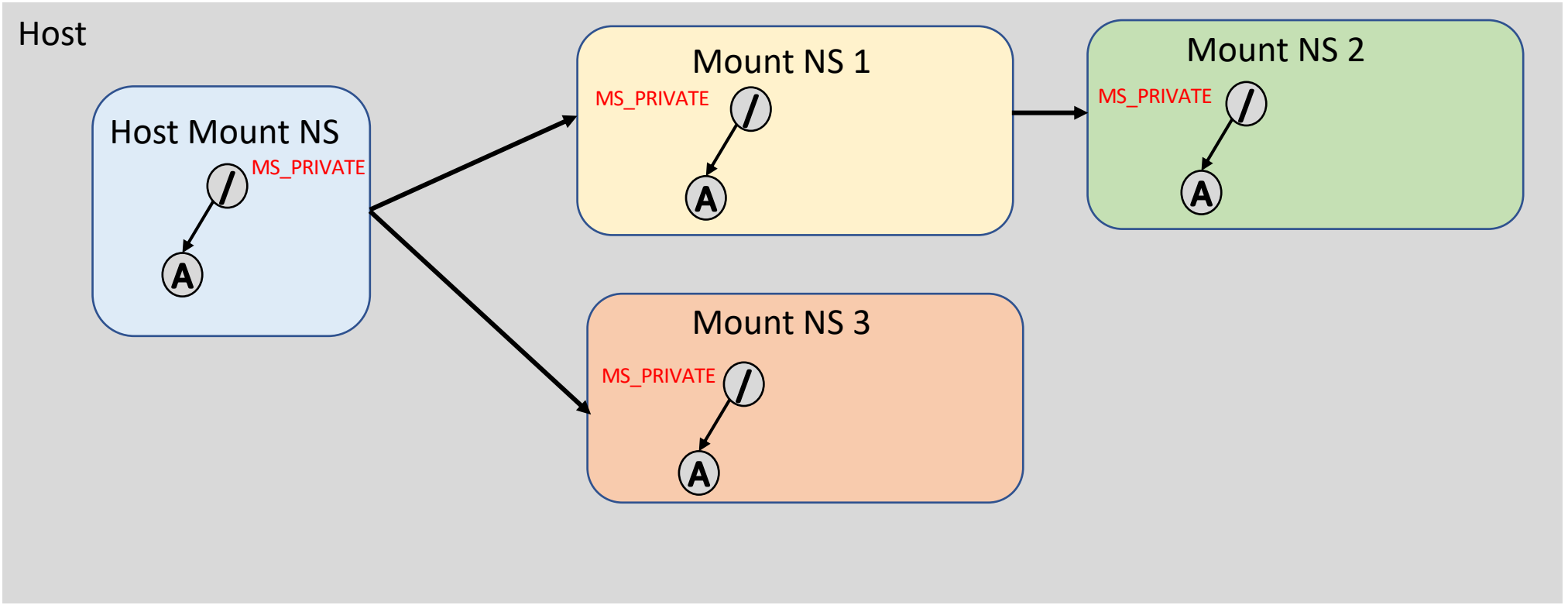


- NS 1 and NS 3 are cloned from Host Mount NS.
- NS 2 is cloned form NS 1

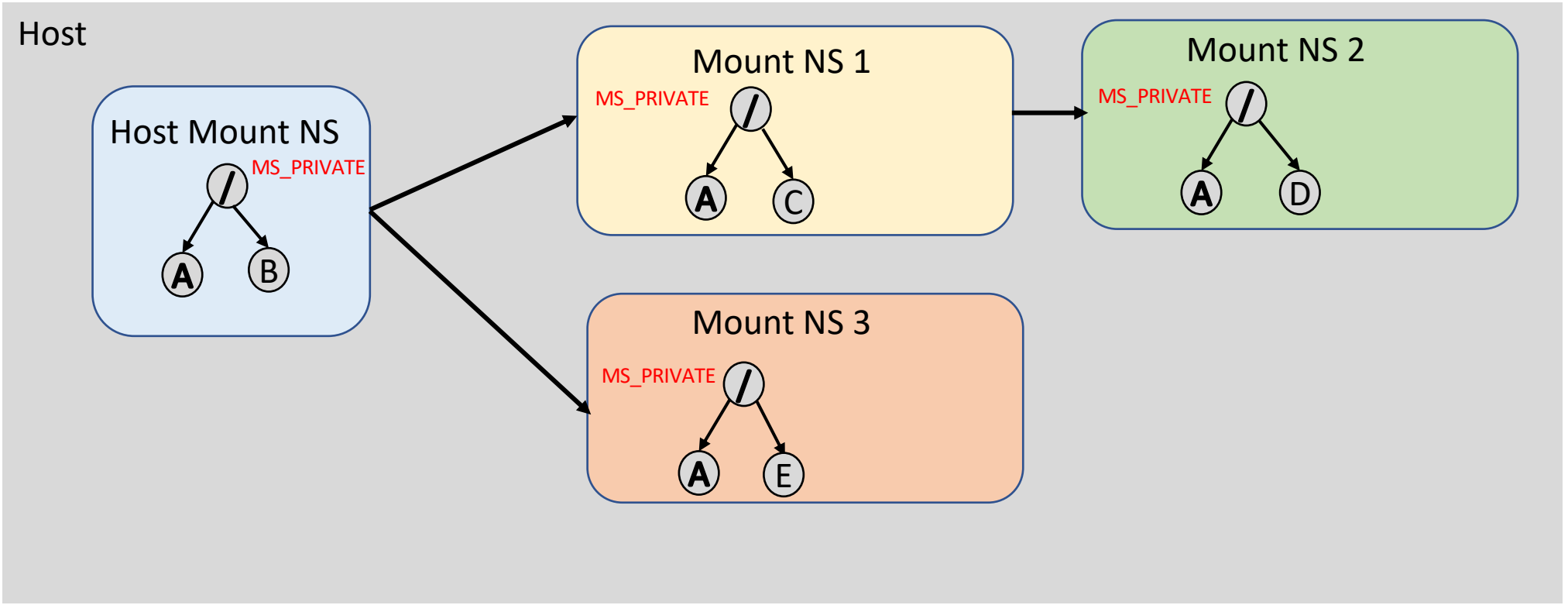
MS_SHARED


Host
Host Mount NS
MS_SHARED
/
A
B
Mount NS 1
MS_SHARED
/
A
B
Mount NS 2
MS_SHARED
/
A
B
Mount NS 3
MS_SHARED
/
A
B

MS_PRIVATE


Host
Host Mount NS
MS_PRIVATE
/
A
Mount NS 1
MS_PRIVATE
/
A
Mount NS 2
MS_PRIVATE
/
A
Mount NS 3
MS_PRIVATE
/
A

MS_PRIVATE

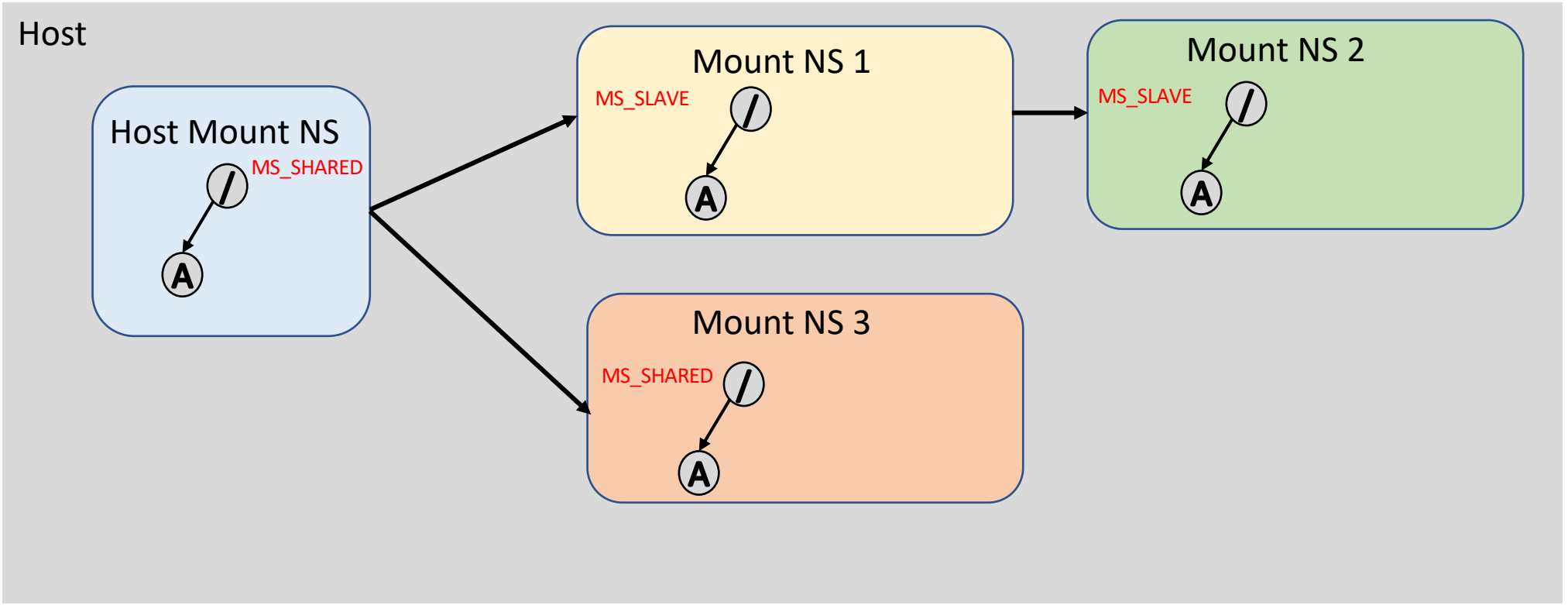


- Changes in either NS are not shared across containers.

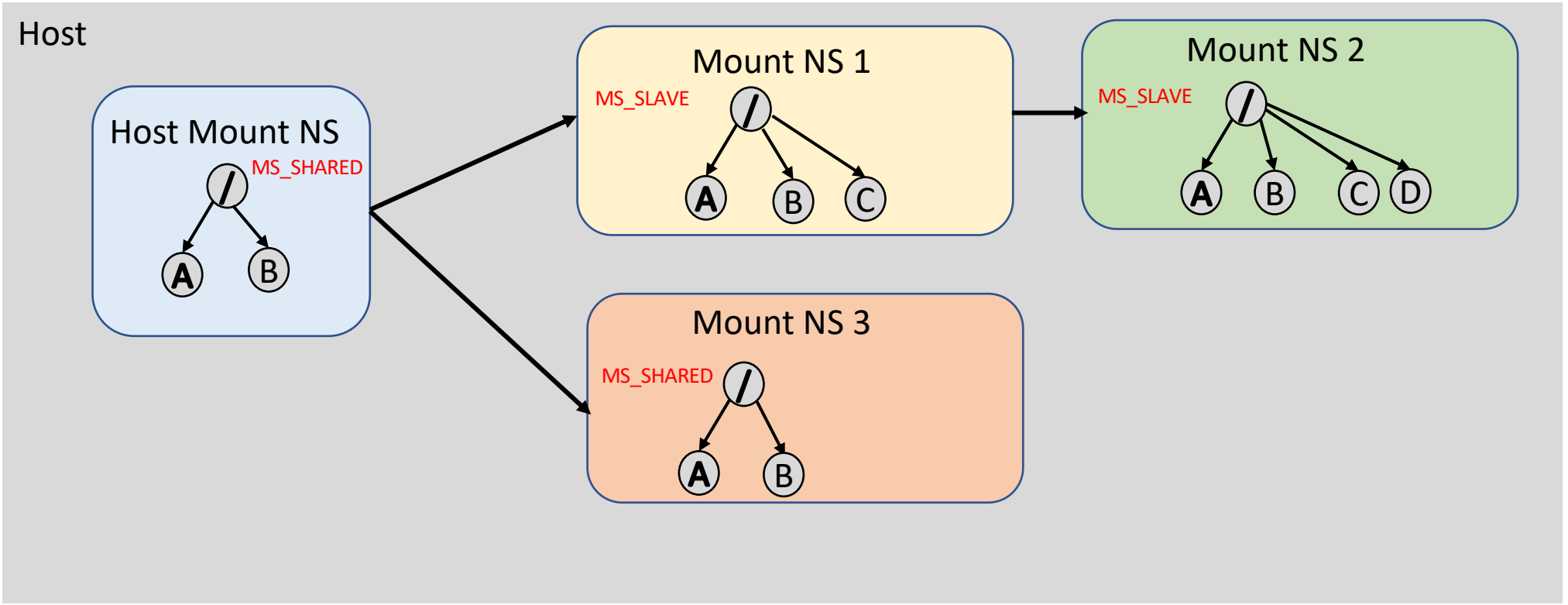
MS_SLAVE

- MS_SLAVE: A mount point of this type receives changes from patent but does not propagate changes to peers.

MS_SLAVE


Host
Host Mount NS
MS_SHARED
/
A
Mount NS 1
MS_SLAVE
/
A
Mount NS 2
MS_SLAVE
/
A
Mount NS 3
MS_SHARED
/
A

MS_SLAVE



- Changes in NS 1 and NS 2 are not seen by peers or host.

MS_UNBINDABLE

- MS_UNBINDABLE: A mount point of this type can’t be the source of a bind mount operation. And similar to MS_PRIVATE, changes under this mount point does not propagate/receive changes to/from peers.

Mount Point Details

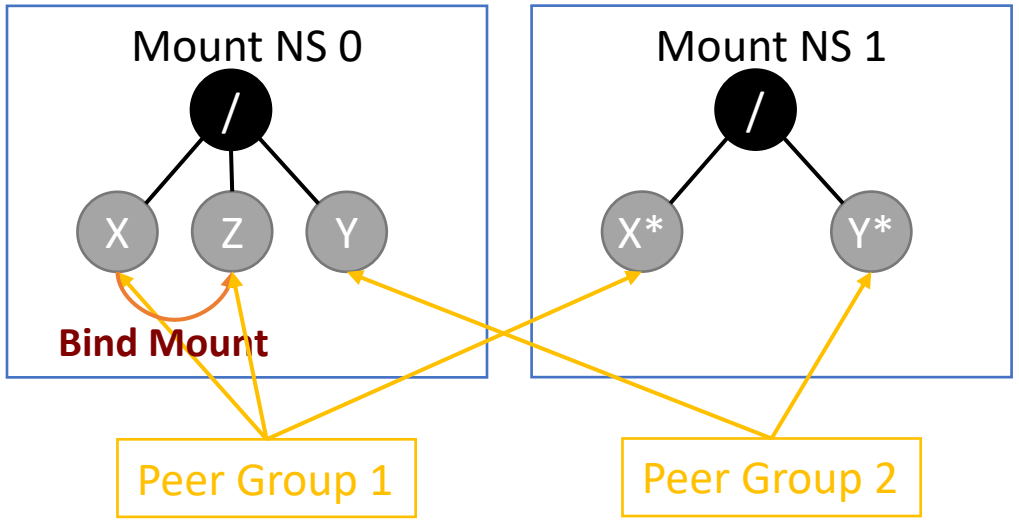
- The first is that the propagation type is a per-mount-point setting. Within a namespace, some mount points might be marked shared, while others are marked private (or slave or unbindable).
- Propagation type determines the propagation of mount and unmount events **immediately under** the mount point.
- Possible for a mount to be both the slave of a master peer group as well as sharing events with a set of peers of its own—a so-called slave-and-shared mount.
- Event propagation does not imply some sort of message passing between mount points.

Example

- **# mount --make-private /**
- **# mount --make-shared /dev/sda3 /X**
- **# mount --make-shared /dev/sda5 /Y**
- **# unshare -m --propagation unchanged sh**
- **# mkdir /Z**
- **# mount --bind /X /Z**

Example

- **# mount --make-private /**
- **# mount --make-shared /dev/sda3 /X**
- **# mount --make-shared /dev/sda5 /Y**
- **# unshare –m --propagation unchanged sh**
- **# mkdir /Z**
- **# mount --bind /X /Z**



Defaults for / and unshare()

Default propagation type is for a new mount point:

- If the mount point has a parent (i.e., it is a non-root mount point) and the propagation type of the parent is MS_SHARED, then the propagation type of the new mount is also MS_SHARED.
- Otherwise, the propagation type of the new mount is MS_PRIVATE.

- What is default for root?

Defaults for / and unshare()

Default propagation type is for a new mount point:

- If the mount point has a parent (i.e., it is a non-root mount point) and the propagation type of the parent is MS_SHARED, then the propagation type of the new mount is also MS_SHARED.
- Otherwise, the propagation type of the new mount is MS_PRIVATE.

- What is default for root?
  - systemd sets the propagation type of all mount points to MS_SHARED.
- What does unshare() assume as default?
  - Opposite behavior. Why?
  - mount --make-rprivate /

  - To prevent: unshare -m --propagation unchanged <cmd>

Creating a basic container

```
int main(int argc, char *argv[]) {
 int cpid = fork();
if (cpid == -1)
{ errExit("fork"); }
if (cpid == 0)
{ unshare(CLONE_NEWNS); // (1) Create a a new mount namespace.
mount("", "/", NULL, MS_SLAVE | MS_REC, NULL); // (2)  Why SLAVE?
mount(rootfs, rootfs, NULL, MS_BIND | MS_REC, NULL); // (3)  Why bind mount to itself?
chdir(rootfs); // (4) Enter the rootfs directory.
mount(rootfs, "/", NULL, MS_MOVE, NULL); // (5) Move mount point rootfs from itself to "/"
chroot("."); // (6) Change the root directory to rootfs.
chdir("/"); // (7)  Safe practice
mount("", "/", NULL, MS_SHARED | MS_REC, NULL); // (8) changes in the container will be propagated to its children if any
mount("proc", "/proc", "proc", MS_NOSUID | MS_NOEXEC | MS_NODEV, NULL); // (9) Mount procfs for the container.
execv(argv[1], &argv[1]); }
else {
 if (waitpid(cpid, NULL, 0) == -1) { errExit("waitpid"); } }
 return 0;
}
```

Do we still need chroot()?

- pivot_root(SYS_pivot_root, const char *new_root, const char *put_old) changes the root mount in the mount namespace of the calling process.
  - Moves the root mount to the directory put_old and makes new_root the new root mount.
  - pivot_root() does not change the caller's current working directory (unless it is on the old root directory), and thus it should be followed by a chdir("/") call.

- MS_MOVE + chroot() = pivot_root()

Example

chdir(new_root);

pivot_root(".", ".");

umount2(".", MNT_DETACH);

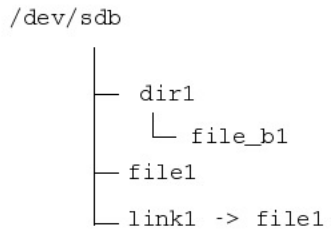
- **pivot_root**() call stacks the old root mount point on top of the new root mount point at /.
  - At that point, the calling process's root directory and current working directory refer to the new root mount point (*new_root*).
- During the subsequent **umount**() call, resolution of *"."* starts with *new_root* and then moves up the list of mounts stacked at /, with the result that old root mount point is unmounted.

Verify container and parent in different namespace

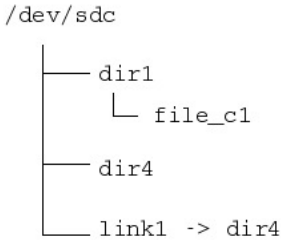
- sudo readlink /proc/$$/ns/mnt
- sudo readlink /proc/<PID>/ns/mnt

Union Mounts

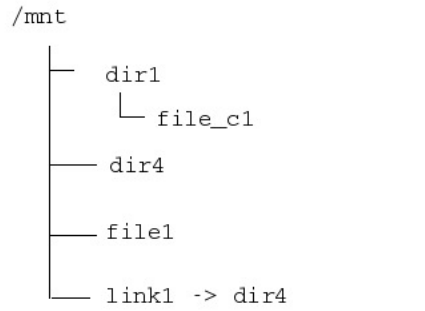
Filesystem on /dev/sdb



Filesystem on /dev/sdc



Resultant filesystem after union mounts



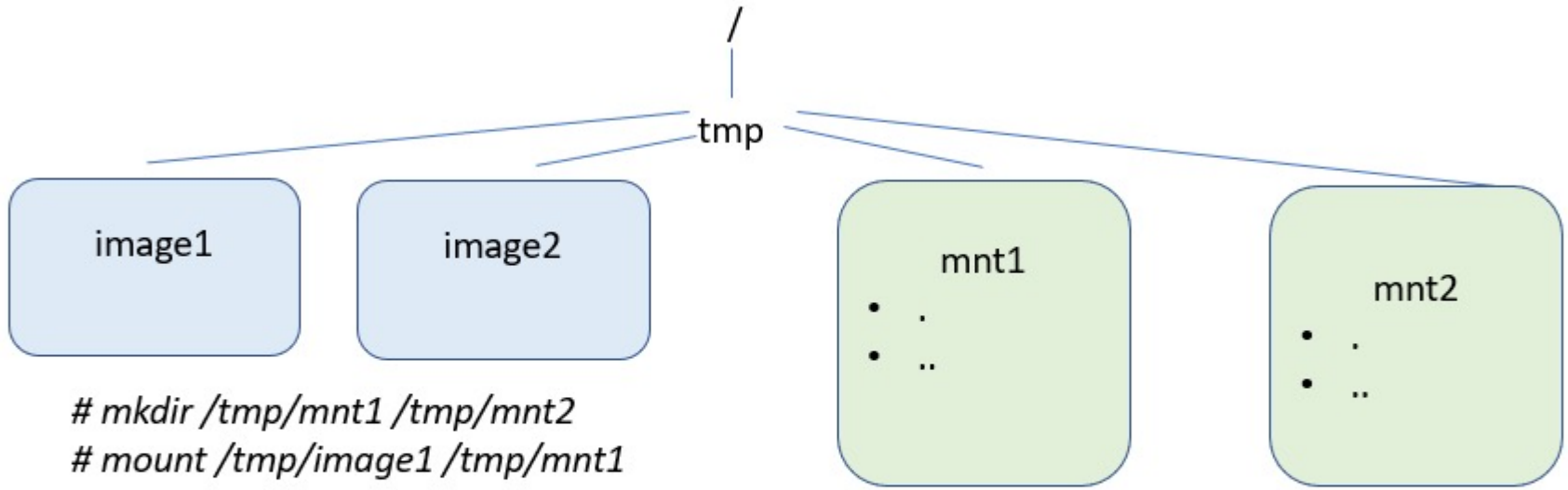

```
# mount /dev/sdb /mnt
# ls /mnt dir1 file1 link1
# mount --union /dev/sdc /mnt
# ls /mnt dir1 dir4 file1 link1
# umount /mnt
# ls /mnt dir1 file1 link1
```

Mounting multiple filesystems on the same mount point

Here are 2 files “image1” & “image2” formatted as a filesystems

 # dd if=/dev/zero of=/tmp/image1 bs=1024 count=1024
 # dd if=/dev/zero of=/tmp/image2 bs=1024 count=1024
 # mkfs –t ext4 /tmp/image1
 # mkfs –t ext4 /tmp/image2

Accessing filesystems on these “image1” & “image2”

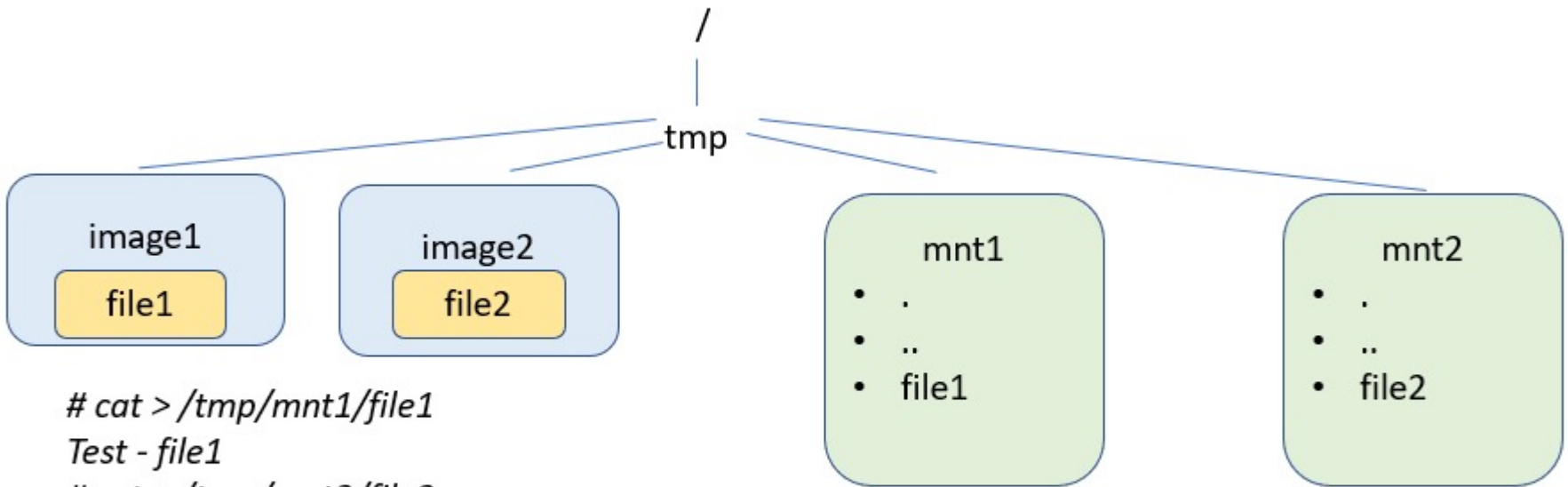


 # mkdir /tmp/mnt1 /tmp/mnt2
 # mount /tmp/image1 /tmp/mnt1
 # mount /tmp/image2 /tmp/mnt2

Mounting filesystems (files image1 and image2) on mount points mnt1 and mnt2 in read-write mode allows files created and accessed via respective mount points.

** Directories are lists of files and subdirectories*

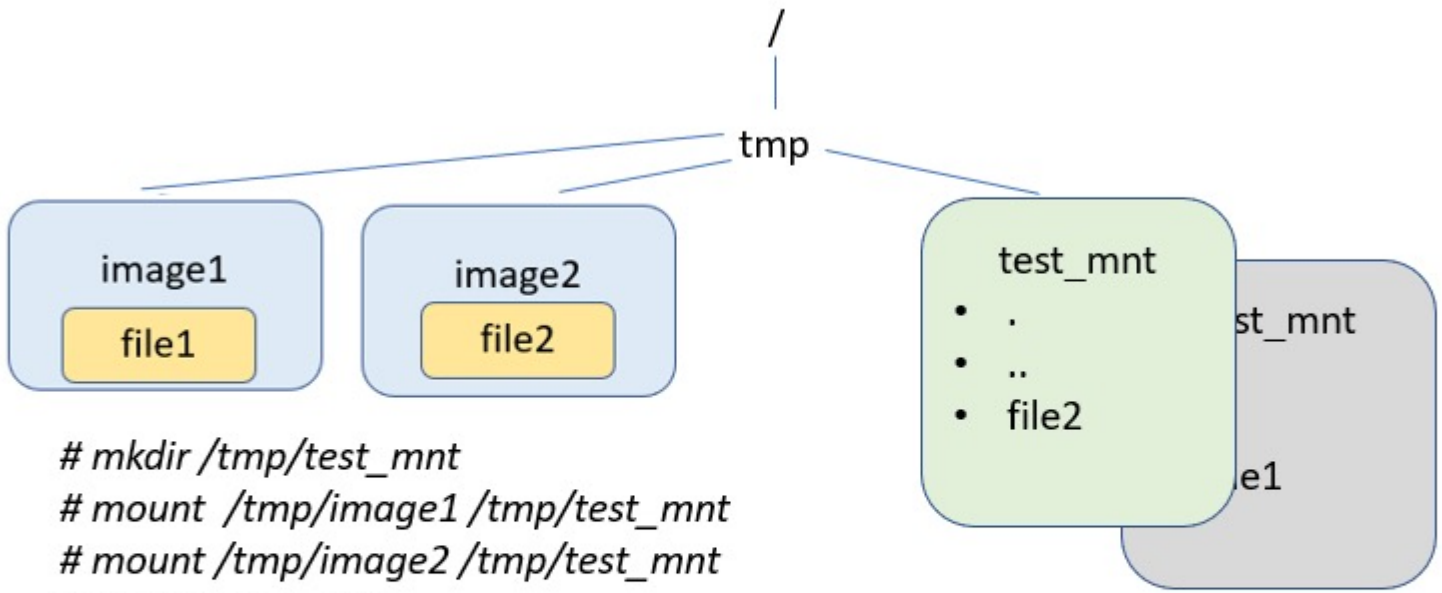
Creating files in filesystems (“image1” & “image2”)



 # cat > /tmp/mnt1/file1
Test - file1
 # cat > /tmp/mnt2/file2
Test - file2
 # umount /tmp/mnt1 ; umount /tmp/mnt2

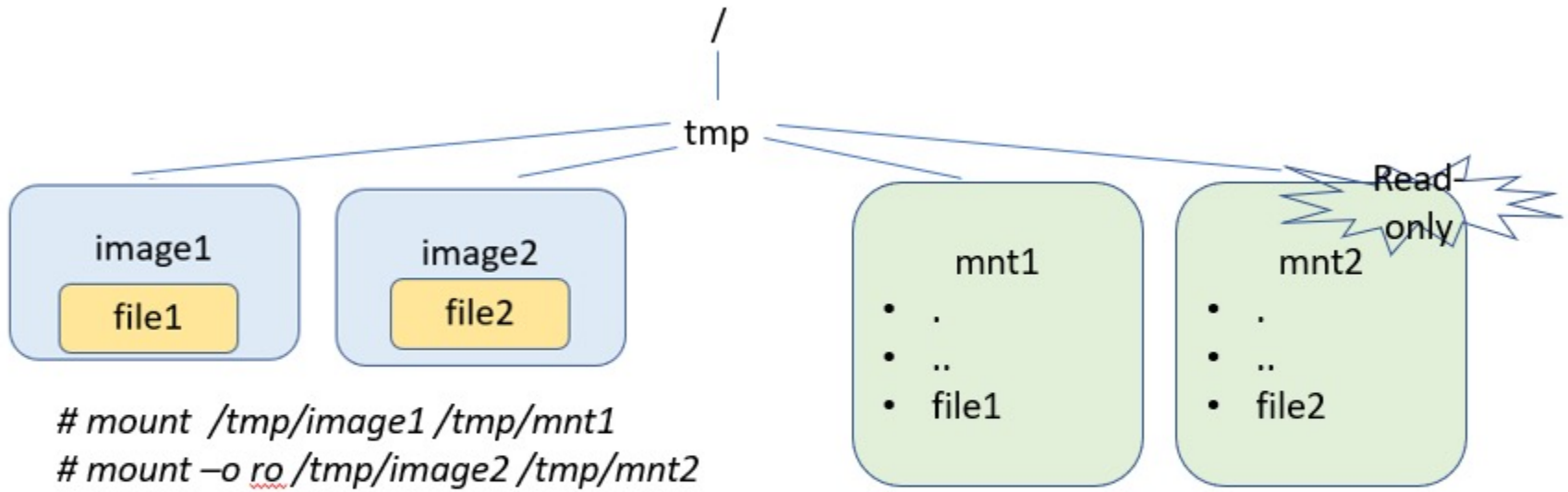
Files file1, and file2 are created on file systems image1, and image2 respectively

Mounting multiple filesystems on single mount point

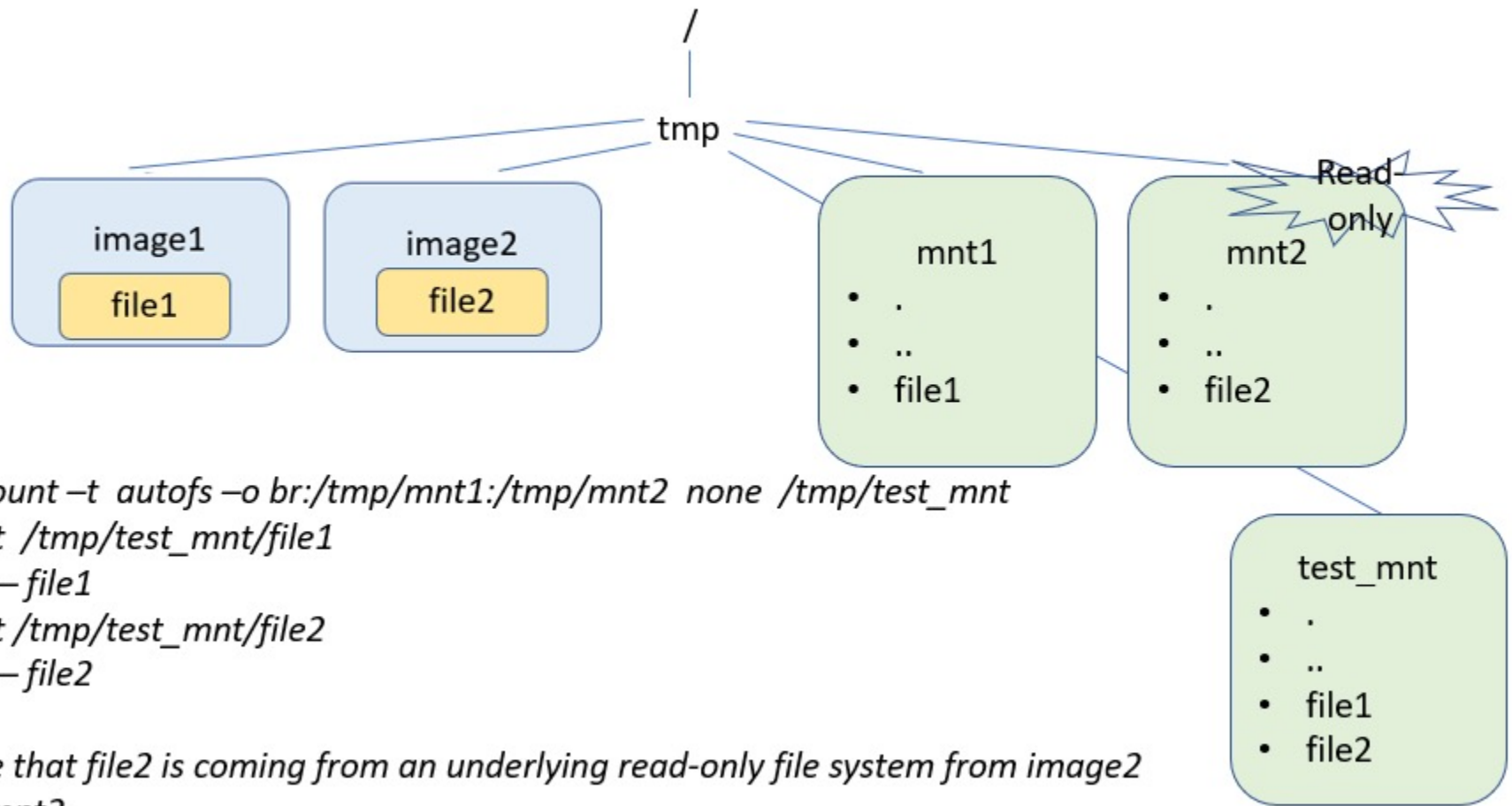


 # mkdir /tmp/test_mnt
 # mount /tmp/image1 /tmp/test_mnt
 # mount /tmp/image2 /tmp/test_mnt
 # ls /tmp/test_mnt
file2
 # umount /tmp/test_mnt
 # ls
file1
 # umount /tmp/test_mnt

When two filesystems are mounted on a single mount point one after the other, only files from the filesystem mounted last remain accessible via this mount point.


/
tmp
image1
file1
image2
file2
mnt1
.
..
file1
mnt2
.
..
file2
Read
only
mount /tmp/image1 /tmp/mnt1
mount –o ro /tmp/image2 /tmp/mnt2

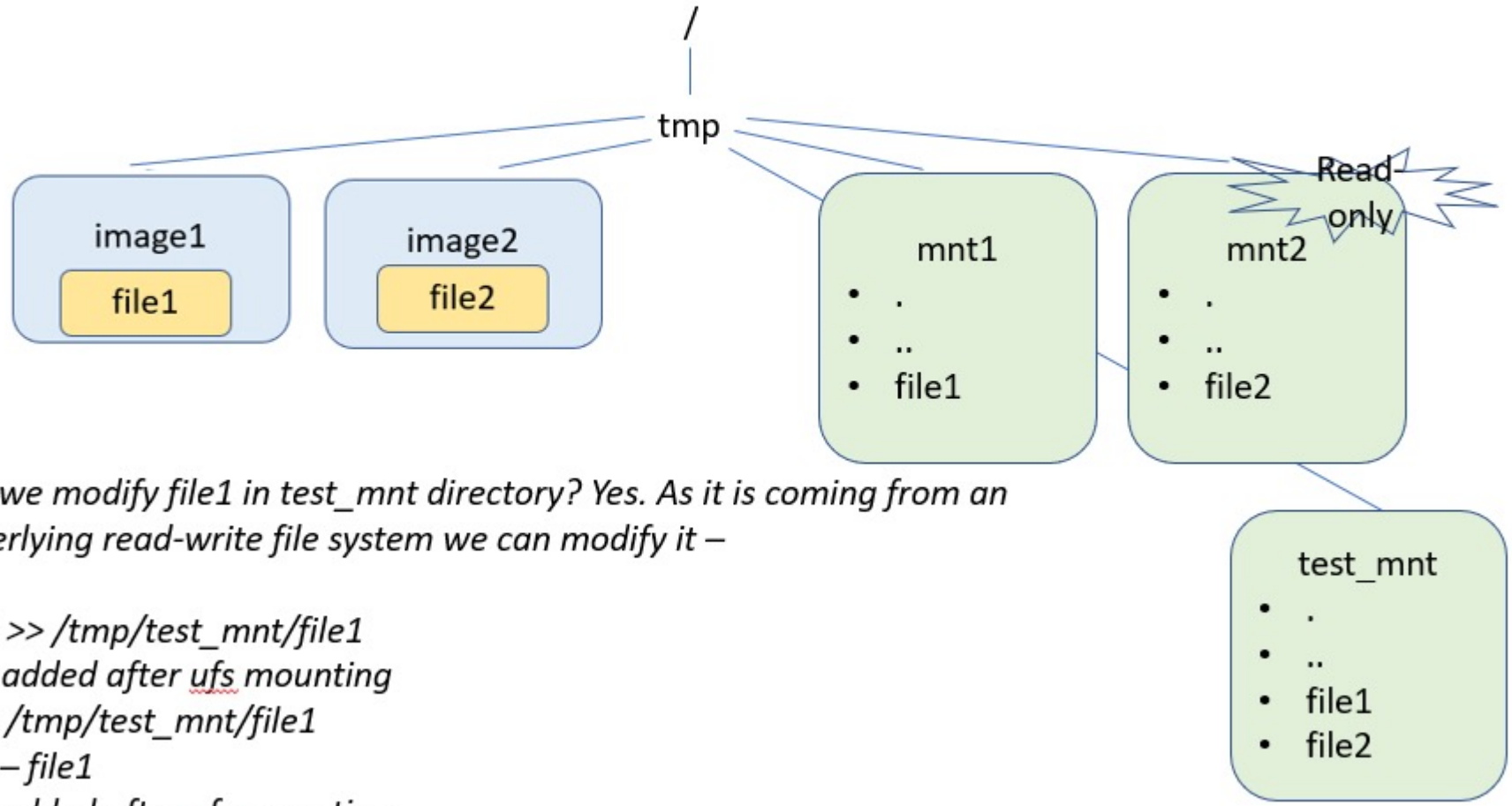
Union of multiple filesystems on single mount point



 # mount –t autofs –o br:/tmp/mnt1:/tmp/mnt2 none /tmp/test_mnt
 # cat /tmp/test_mnt/file1
Test – file1
 # cat /tmp/test_mnt/file2
Test – file2

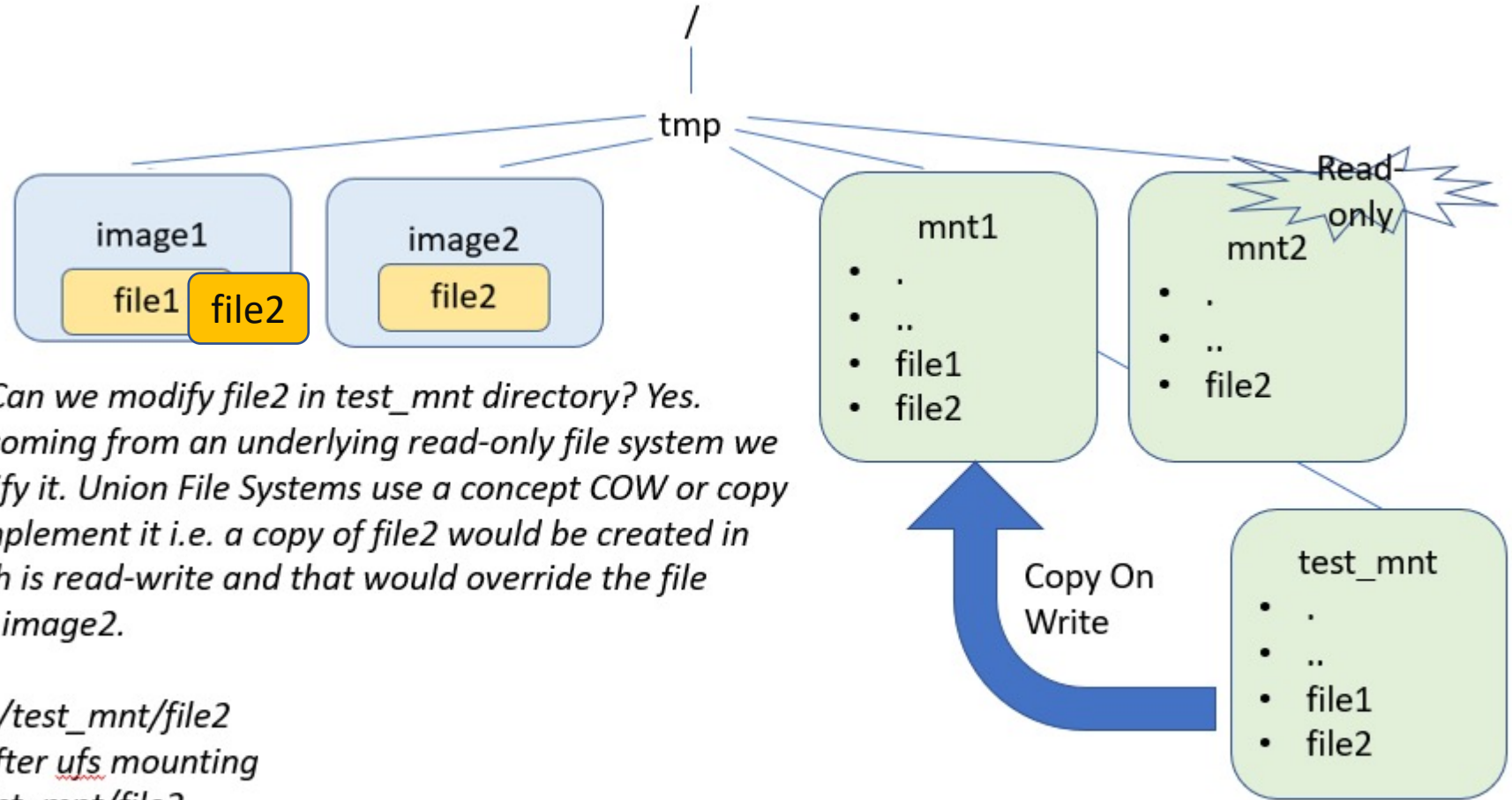
Note that file2 is coming from an underlying read-only file system from image2 OR mnt2

Union of multiple filesystems on single mount point



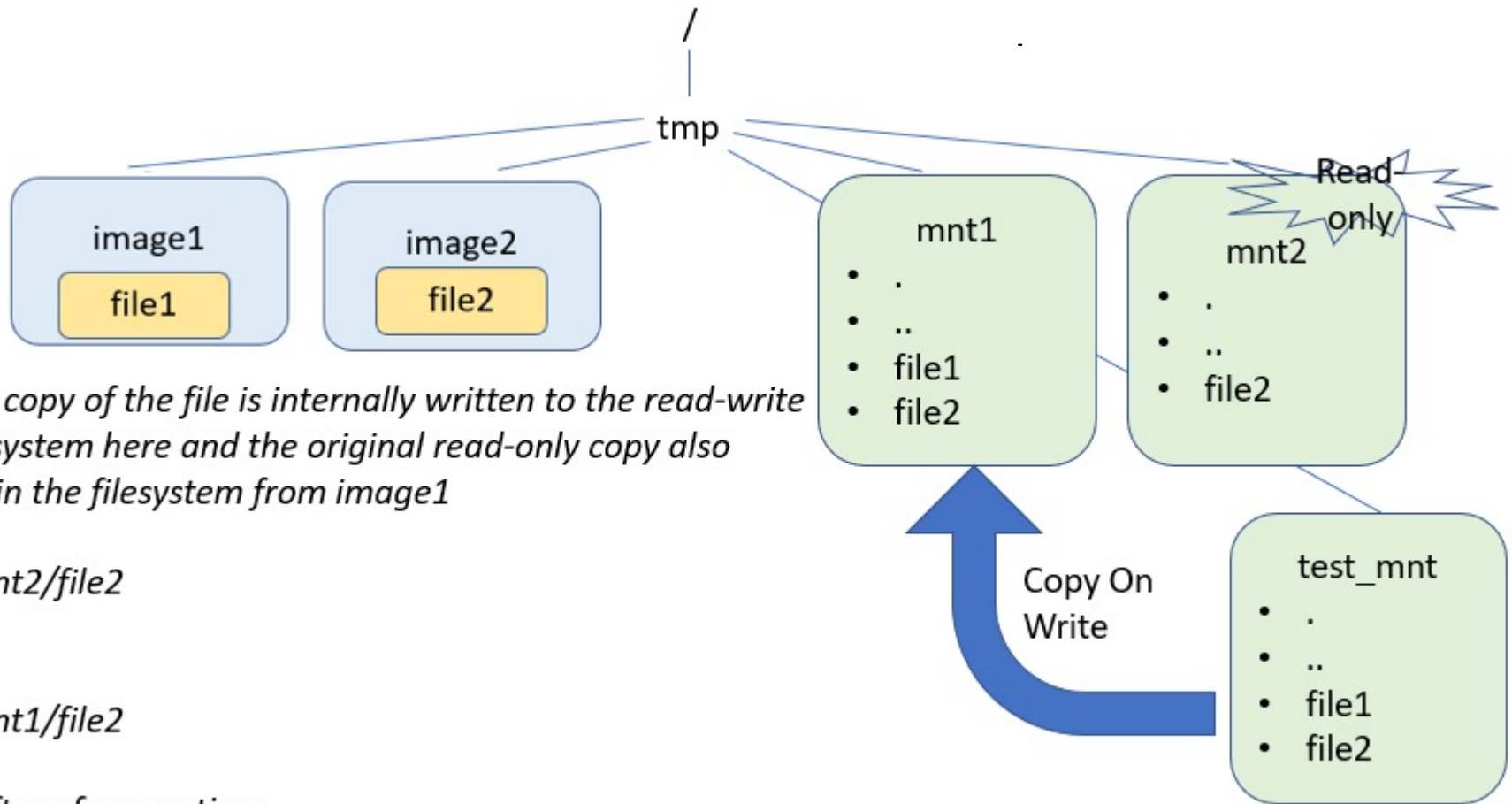
Can we modify file1 in test_mnt directory? Yes. As it is coming from an underlying read-write file system we can modify it –

#cat >> /tmp/test_mnt/file1
Text added after ufs mounting
#cat /tmp/test_mnt/file1
Test – file1
Test added after ufs mounting



Important *- Can we modify file2 in test_mnt directory? Yes. Though it is coming from an underlying read-only file system we can still modify it. Union File Systems use a concept COW or copy of write to implement it i.e. a copy of file2 would be created in image1 which is read-write and that would override the file coming from image2.*

#cat >> /tmp/test_mnt/file2
Text added after ufs mounting
#cat /tmp/test_mnt/file2
Test – file2
Test added after ufs mounting



The modified copy of the file is internally written to the read-write mounted filesystem here and the original read-only copy also remains within the filesystem from image1

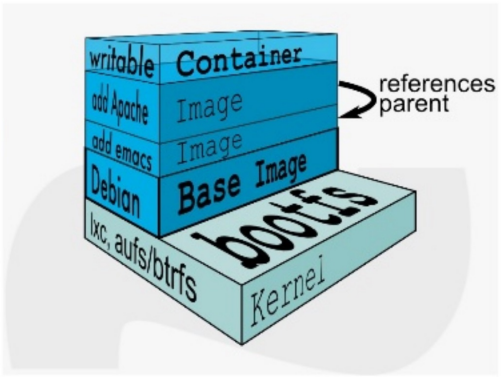
#cat /tmp/mnt2/file2
Test – file2

#cat /tmp/mnt1/file2
Test – file2
Test added after ufs mounting

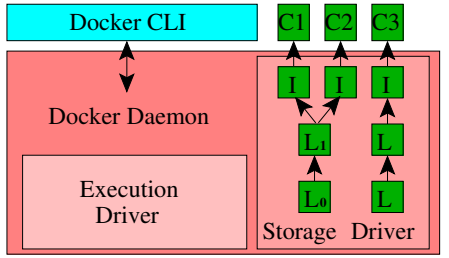
Storage in Containers

How are data & containers stored?

- **AUFS** *Another Union Filesystem*
  - possibly other snapshotting fs (zfs) / block device (LVM)
- Layered approach
  - *rootfs* → kernel layer
  - *bootfs* → a Linux distribution
  - *emacs*
  - *apache*
  - application
- *Copy-on-Write* approach – à la subversion (SVN)



Docker design



- *images* are similar to VM images, except that they consist of a series of *layers*.
- Every layer is a set of files. The layers get stacked with files in the upper layers superseding files in the layers below them.
  - The number of layers in a single image ranges from one to several dozens.
- Similarly to git, the layers are identified by fingerprints of their content.
- Different images often share layers, which provides significant space and I/O transfer savings.
- A layer in a Docker image often represents a layer in the corresponding software stack.
  - For example, an image could consist of a Linux distribution layer, a libraries layer, a middleware layer, and an application layer.

Images

- **A container image is read-only**, with changes to its file system during execution stored separately.
- To create a container from an image, Docker creates an additional *writeable* layer on top of the image with which the container interacts.
- When the container updates a file, the file is copied to the writable layer and only the copy is updated (copy-on-write).
- Unless the user saves the changes as a new layer (and hence a new image), the changes are discarded when the container is removed.

Storage Drivers

- Storage drivers are sometimes also called *graphdrivers* because they maintain the graph (tree) of Docker layers and images.
- A storage driver is responsible for preparing a file system for a container.
- Several:
  - VFS
  - AUFS
  - Overlay
  - BtrFS

Which storage driver to use?

Source: Tarasov, Vasily, et al. "In search of the ideal storage configuration for Docker containers."
2017 IEEE 2nd International Workshops on Foundations and Applications of Self Systems (FAS* W)*. IEEE, 2017.

Storage Driver Comparison

VFS

- This simple driver does not save file updates separately from an image via CoW, but instead creates a complete copy of the image for each newly started container. It can therefore run on top of any file system.
- + stable
- - inefficient

Aufs (Another Union file system)

- Takes multiple directories and stacks them on top of each other to provide a single unified view at a single mount point. Aufs performs file-level CoW, storing updated versions of files in upper branches. To support Docker, each branch maps to an image layer
- - Not so much stable
- + Efficient but depends on multiple factors

OverlayFS

- Yet another implementation of a union file system
- Available for Linux distributions
- OK on efficiency and stability

Btrfs

- Modern CoW file system based on a *CoW-friendly* version of a B-tree
- Natively supports CoW and does not require an underlying file system
- + IO performance
- + Space efficiency
- - not so stable