Project 5: Container

CSE 130 Principles of Computer Systems Design

Spring 2023

Project 5 is out!

- Sum up all the things we learned in this class
- Focus on container and file system
- Implement a container runtime that can run a program in a sandboxed environment
- GitHub Classroom to create a repo, submit on Gradescope

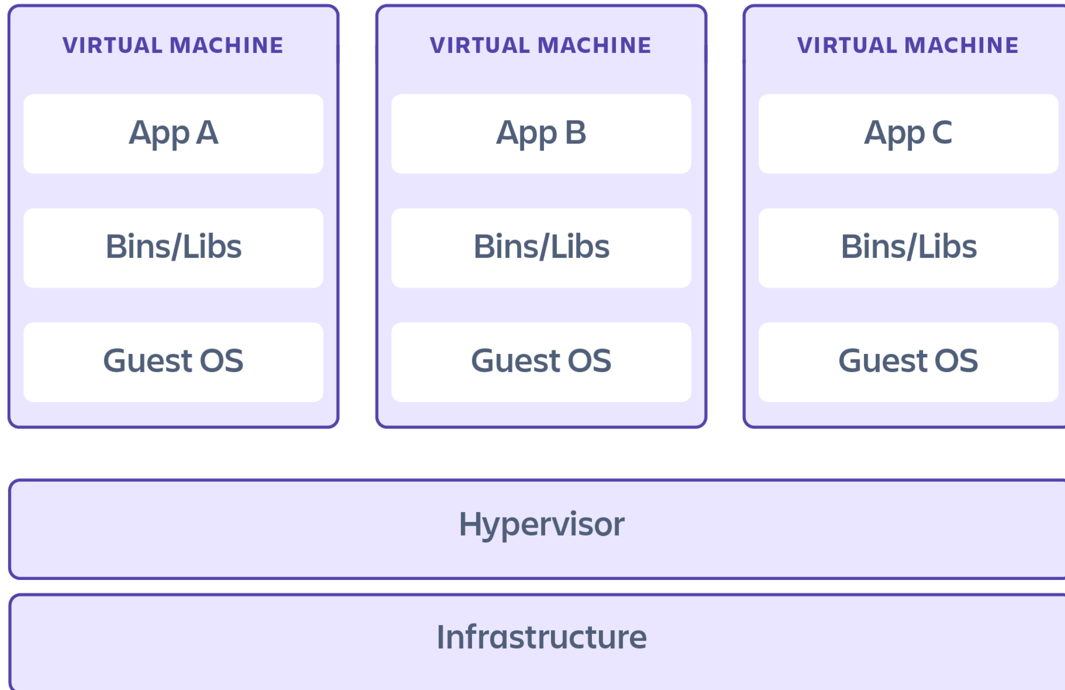
Q. We've been using Docker and Dev Container throughout the quarter. But what is a container?

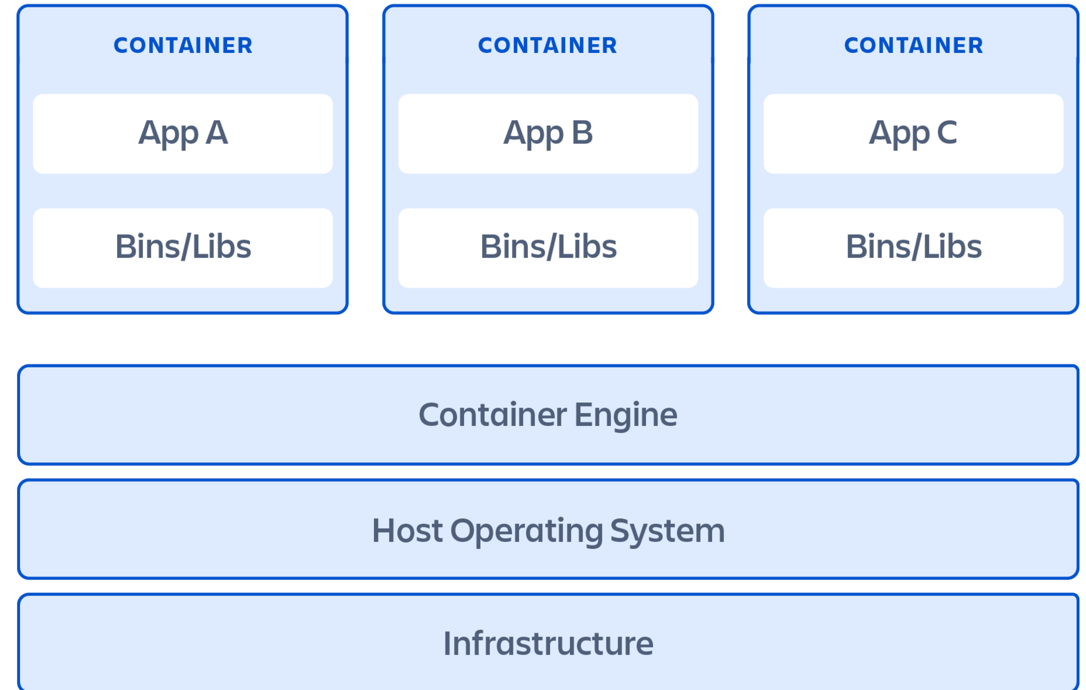
Container

- A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
- A container is created from an image, which is a package of all the dependencies needed to run an application.
- A container runs natively on Linux and shares the kernel of the host machine with other containers.

Container vs. VM

- A virtual machine uses a hypervisor to virtualize the underlying hardware.
  - Each VM includes a full copy of an operating system
  - VMs run on top of a physical machine using a hypervisor (e.g. VirtualBox, VMWare)
- A container runs natively on Linux and shares the kernel of the host machine with other containers.
  - Each container shares the host OS kernel and, usually, the binaries and libraries, too.
  - Containers are isolated from each other and from the host machine using kernel features.





Credit: Containers vs Virtual Machines | Atlassian

Isolation

- Containers are isolated from each other and from the host machine using kernel features.
    - Whatever happens inside a container stays inside the container.
- Modern container runtime provides isolation for many resources such as CPU, Memory, Network, File system, etc.
- In this project, we focus on two aspects of isolation: **Processes** and **Filesystem**.

Isolating Processes: PID Namespace

- Processes in a container should not be able to see processes outside the container.
- PID namespace provides isolation for processes.
  - Each container has its own PID namespace.
  - Processes inside a container can only see processes inside the same container.

Isolating Filesystem: Mount Namespace

- Processes in a container should not be able to see files outside the container.
- Mount namespace provides isolation for filesystem.
  - Each container has its own mount namespace.
  - Processes inside a container can only see files inside the same container.

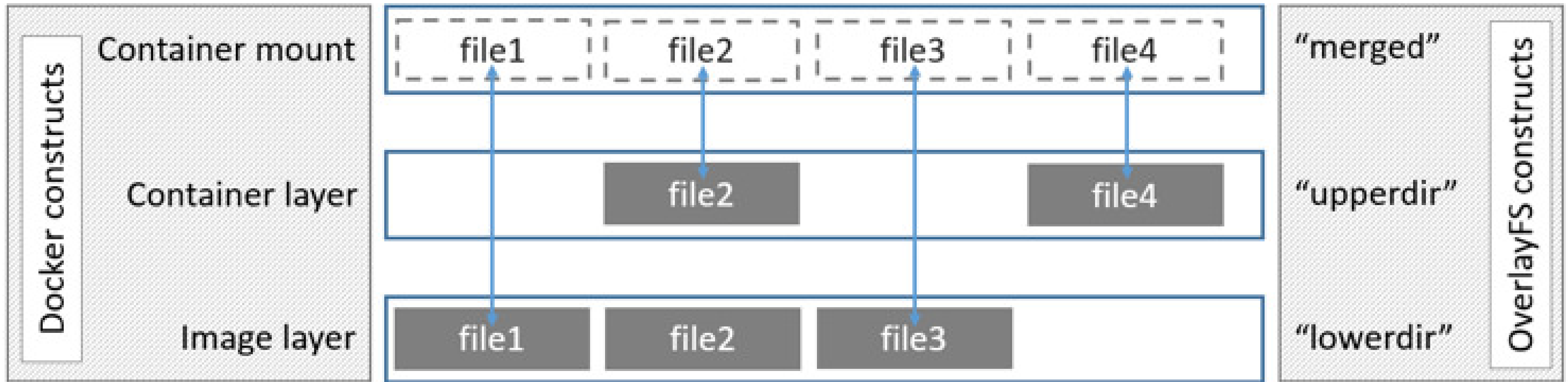
Creating a new namespace: `clone`

- `clone` is a system call that creates a new process.
  - Similar to `fork` but more flexible.
  - Specify the functions to run in the child process and an argument to pass, just like `pthread_create`.
- Specify the flags to create a new PID namespace and mount namespace.

```
container_t container;
int clone_flags = SIGCHLD | CLONE_NEWNS | CLONE_NEWPID;
int pid = clone(container_exec, &child_stack, clone_flags, &container);
```

Isolating Filesystem: OverlayFS

- Container images are read-only.
  - We want to reuse the same image for multiple containers.
- However, containers need to be able to write to the filesystem.
- OverlayFS is a filesystem that allows us to mount a read-only filesystem on top of a writable filesystem.
  - The read-only filesystem is called the lower layer.
  - The writable filesystem is called the upper layer.
  - The combined filesystem is called the overlay.



Credit: Use the OverlayFS storage driver | Docker Documentation

Testing Overlay FS

Note: You need a non-docker Linux environment to test this.

```
# create directories
mkdir lower upper work merged
# create a file in the lower directory
echo "this is in lower" > lower/foo
# create overlayFS
mount -t overlay overlay -o lowerdir=lower,upperdir=upper,workdir=work merged
# check the content of the merged directory
ls merged
cat merged/foo
# create changes in the merged directory
echo "new foo" > merged/foo
echo "bar" > merged/bar
# check the content of the upper
ls upper
```

`container.c`

- `container.c` implements a small container runtime.
- It creates a new PID and mount namespace, set up the overlayFS, and run the program.
- The program is specified by the command line arguments.

Container Image

- A container image is a directory that contains the root filesystem of the container.
  - Think of taking a snapshot of the entire filesystem.
- Docker uses a layered filesystem to store container images.
  - Each layer is saved as a tarball.
- In this project, we use a directory to represent a container image.
  - Images are directories under the `./images` directory

Creating an image directory

The easiest way to create an image directory is to use `docker export`.

```
docker run --rm -it alpine sh
# on a different terminal
docker ps
docker export [CONTAINER ID] > alpine.tar
mkdir images/alpine
tar -xf alpine.tar -C images/alpine
```

Command-Line Interface

```
./container [ID] [IMAGE] [CMD]...
```

- `ID` is the ID of the container.
  - Docker uses a random string as the ID. Here, the user must provide one.
- `IMAGE` is the name of the image directory.
- `CMD` is the command to run inside the container.

Example

```
# prints hello world from container
sudo ./container c1 alpine echo hello world
# runs a shell inside the container
sudo ./container c2 alpine sh
```

Finishing `container.c`

- `container.c` is missing a few pieces, and your job is to finish it.
- `main`
  - Parse the command-line arguments and pass them to the child process.
- `container_exec`
  - Set up the overlayFS.
  - Call `change_root` to change the root directory of the container.
  - Call `execvp` to run the command.

`main`

- `main` calls `clone` to create a child process and execute `container_exec`.
- Modify `container_t` and `main` to pass all necessary information to `container_exec`.

Creating OverlayFS

```
int mount(const char *source, const char *target, const char *filesystemtype,
  unsigned long mountflags, const void *data);
```

- For `source`, use the dummy string `"overlay"`
- `target` specifies the directory at which the overlayFS will be mounted.
  - Use the `merged` directory: `/tmp/container/{id}/merged`
- `filesystemtype` specifies the type of the filesystem: use `"overlay"`.
- `mountflags` specifies options that are independent of the type of the filesystem: use `MS_RELATIVE`
- `data` specifies options for OverlayFS.
  - `"lowerdir={lower},upperdir={upper},workdir={work}"`
  - `{lower}` is the path to the image directory.
  - `upper` and `work` directories must be created inside `/tmp/container/{id}`

Changing Root Directory

Next, the container needs to change its root directory to the `merged` directory. This is done using the `pivot_root` system call.

Because calling `pivot_root` is a bit complicated, we provide a helper function: `change_root`.

```
void change_root(const char* path);
```

This function will change the root directory to the specified path, as well as doing some other things to make the container work properly.

Running the Command

Finally, the container needs to run the command specified by the user. This is done using the `execvp` system call.

```
int execvp(const char *file, char *const argv[]);
```

- `file` is the path to the executable.
- `argv` is an array of strings that contains the command-line arguments.

Testing with `alpine`

The `alpine` image we created earlier can be used to test your container runtime.

```
# check process isolation
sudo ./container my-container alpine ps -A
# check filesystem isolation
sudo ./container my-container alpine sh
echo foo > foo.txt
```

Testing with other images

Surprisingly, our container runtime is capable of running many images. For example, here is how you can run JavaScript programs using the `node` image.

```
# create a container image
docker pull node:18-alpine
docker run --rm -it node:18-alpine sh
# on a different terminal
docker ps
docker export {container-id} > node.tar
mkdir images/node
tar -xf node.tar -C images/node
# run the image
sudo ./container node-container node node
```

Try running your favorite image!