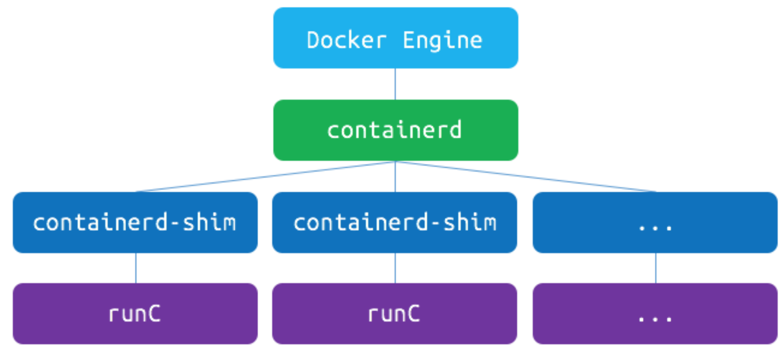
Docker

Docker

- Broadly speaking, Docker is a platform for developing, shipping and running applications using container technology
    - Founded in 2009.
    - Formerly dotCloud Inc.
    - Released Docker in 2013.
- It consists of a bunch of products/tools
    - Docker Engine – i.e., to start container instances
    - Docker Hub – i.e., like github to host public container images
    - Docker Trusted Registry – i.e., store container images
    - Docker Machine - i.e., create a (virtual) machine that support docker contaienr
    - Docker Compose – i.e., to build container images
    - Docker for Windows/Mac
    - Docker Datacenter (swarm) – i.e., similar to Kubernetes

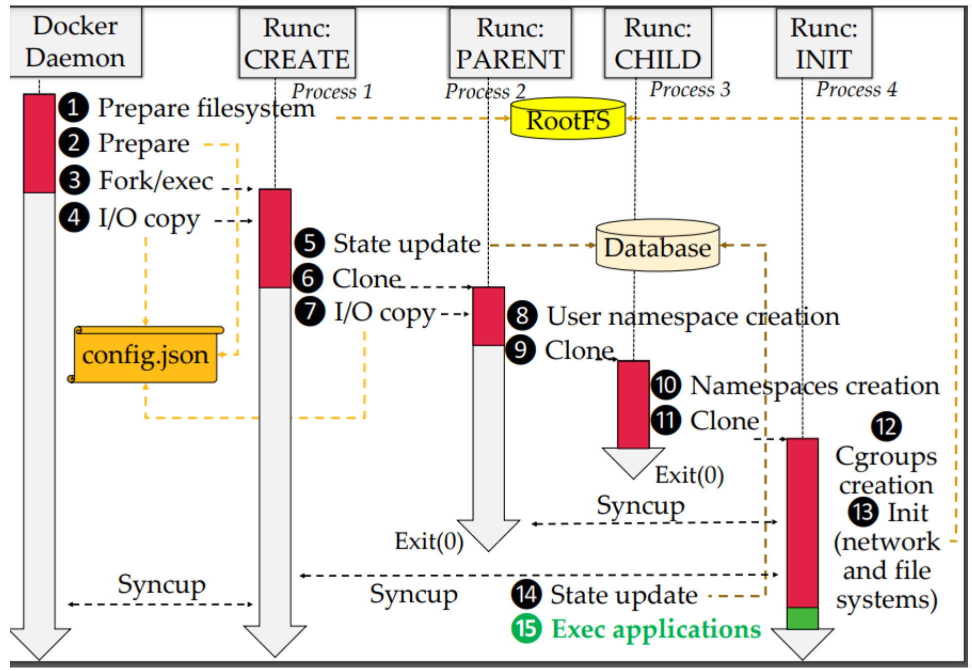
Docker architecture

- Docker Engine receives requests from upstream clients
- Containerd manages the complete container lifecycle
    - Create, pause, termination, deletion
- runC is a lightweight tool that does one thing, it creates a container instance (name spaces and cgroups): https://github.com/opencontainers/runc



Long startup latency

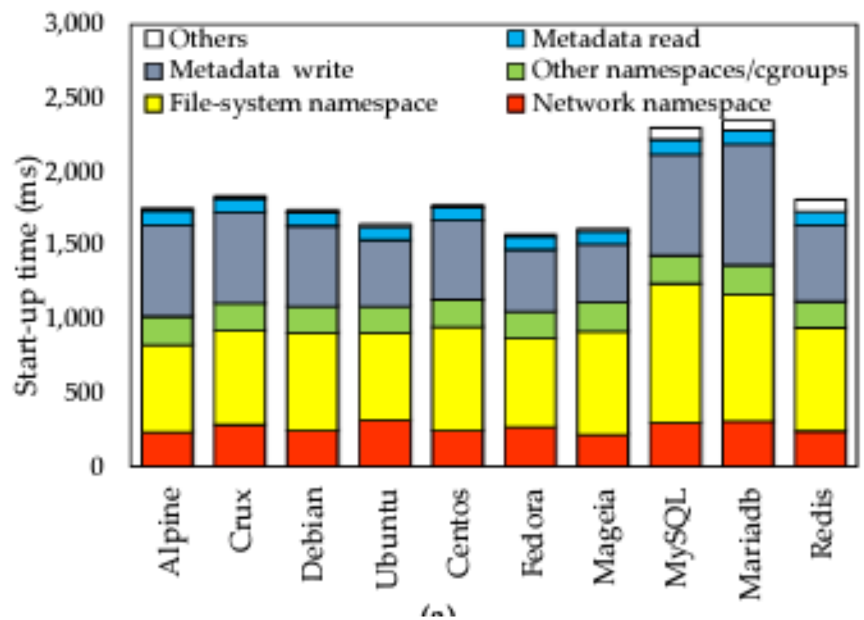
- The construction of a Docker container goes through a long, serialized pipeline involving multiple processes
- These processes need frequent (and slow) synchronization to coordinate the different initialization stages
- E.g., allocating storage and network resources, isolating allocated resources, and filtering system calls.



[FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes]
Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, Hui Lu
In Proc. of the 11th Workshop on Programming Languages and Operating Systems (***PLOS' 21***), online, October 2021.

Long startup latency

- It could take up to 2.4 seconds tc complete a single Linux containe initialization before its encapsulat function code executes.
- Creating various isolation components for a container insta contributes more than 50% to the total cold-start latency.

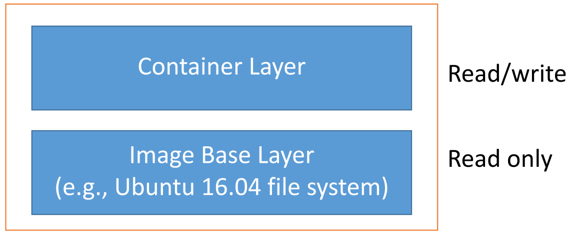


Container Images

- A container instance is launched from a container image.
- A container image is a root file system (mini OS) that includes everything needed to run an application(s)
    - The application code, a runtime, libraries, environment variables, and configuration files.
    - Consisting of folders and files just like a Linux file system (i.e., file organization)
- When we launch a container, a container instance is a runtime instance of an image
    - like binary code vs. processes

Layout of a Docker Container Image

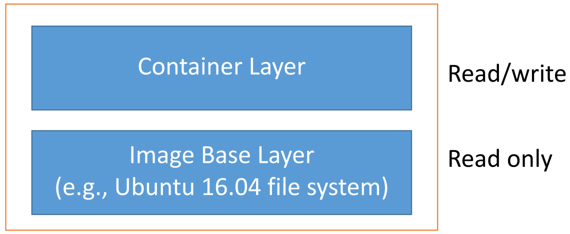
- A container image contains an image base layer, including a base file system.
- When you launch a container instance from the container image, another layer is created on top of the base image layer, called container layer
- Container layer are initially empty and will be discarded when the container instance is terminated
- So all modifications during container execution will be forgotten



- The image base is read only
  - Multiple containers can share the same image base layer
  - Just like a shared library
- The container layer can both read and write
  - Per-instance, private layer

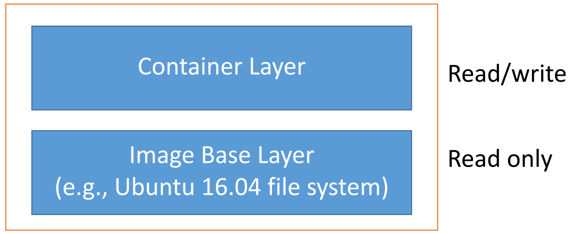
Write policy

- Writing to a file for the first time and the file exists in the image layer
    - copy_up: copy files from the base layer to the container layer, and write changes to it.
- Deleting a file
    - A "whiteout" file is created in the container layer marking that the file with the same name in the image layer is invalid



Read Policies

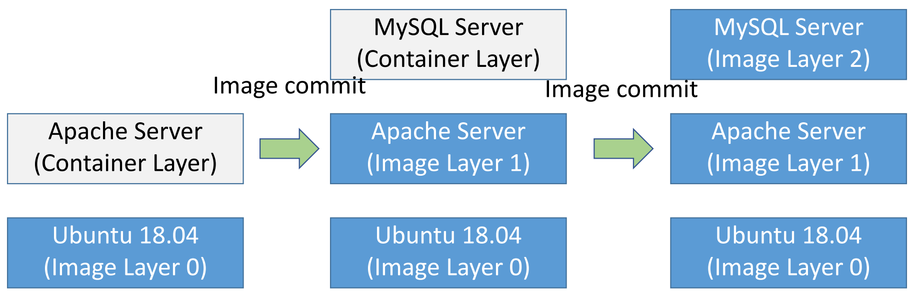
- Files only exist in image layer, it is read from image layer
- Files only exist in container layer, it is read from container layer
- Files exist in both layers, it is read from container
- Files in the container layer obscure files with the same name in the image layer.



Pros/cons of overlay file systems

- Cons
  - Overhead
- Pros
  - Many container instances share the same base images
  - Saving space
  - Container image can be stackable
  - Easy to build new images

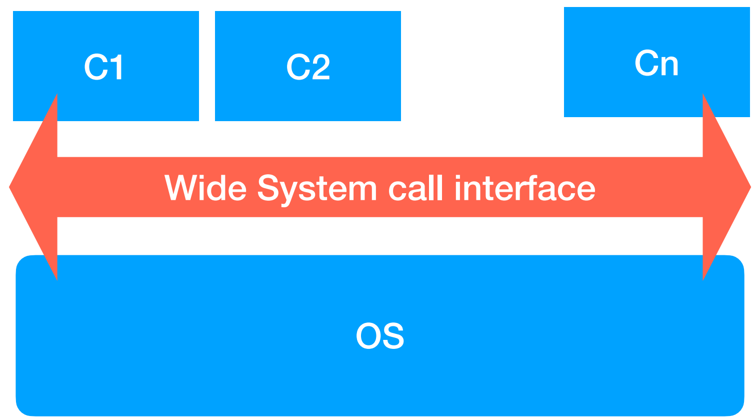
Stackable Container images



gVisor

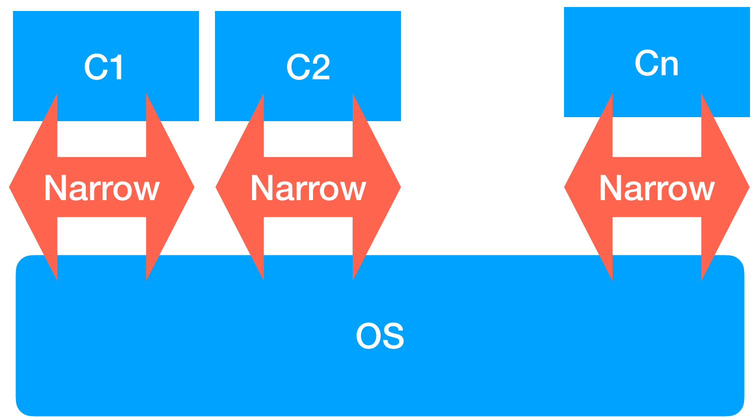
Improving container security by intercepting system calls

Security concern of containers



- Traditional containers have a wide system call interface
    - E.g. Linux has over 400 system calls
- Vulnerable system calls can allow unauthorized access across containers, hosts or data centers etc., thus affecting all the containers on the Host OS.

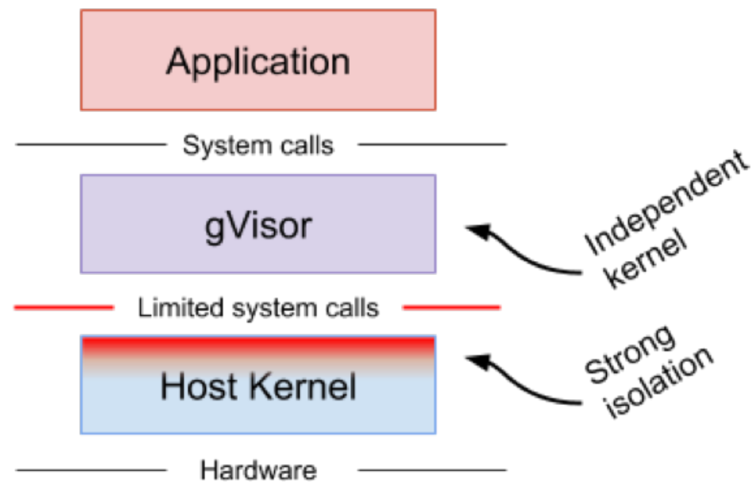
Option 1



- Allow only limited system calls that the container needs.
- Rule-based execution allows the specification of a fine-grained security policy for an application or container. (e.g., Linux's seccomp)
- In practice, not easy. It may break unknown applications whose system call profile is not known accurately.

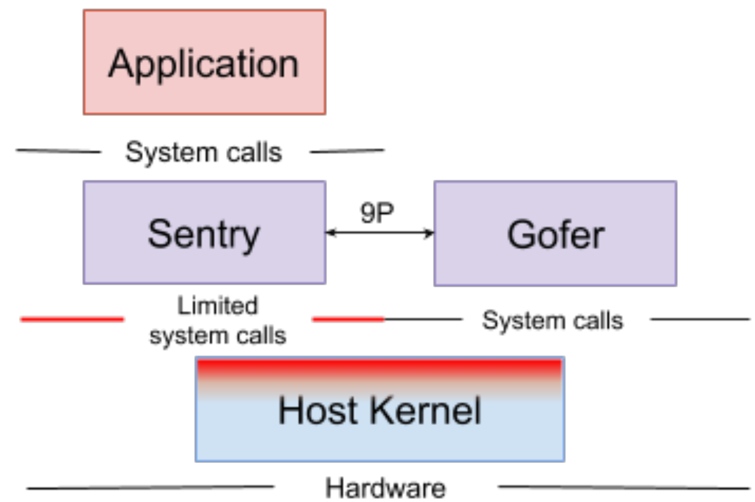
gVisor -- Google's Secure Containerization

- gVisor intercepts application system calls and acts as a guest kernel
- It implements a substantial portion of the Linux system surface
- The isolation boundary between the application and the host kernel is maintained
- Drawback: High per-system call overhead



gVisor Architecture

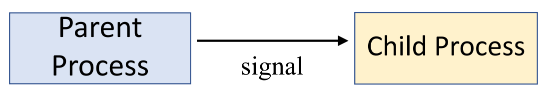
- Sentry: the largest component of gVisor
- Can be thought of as a <u>userspace OS kernel</u>, implementing all the kernel functionality needed by the untrusted application
- System calls are redirected to Sentry
- Sentry will make some host system calls to support its operation, but it will not allow the application to directly control the system calls it makes.



gVisor - Sentry

- gVisor requires a way to implement interception of syscalls
- The ptrace() system call provides a mechanism by which a parent process may observe and control the execution of another process.

long ptrace(enum __ptrace_request *request*, pid_t *pid*, void * *addr*, void * *data*);



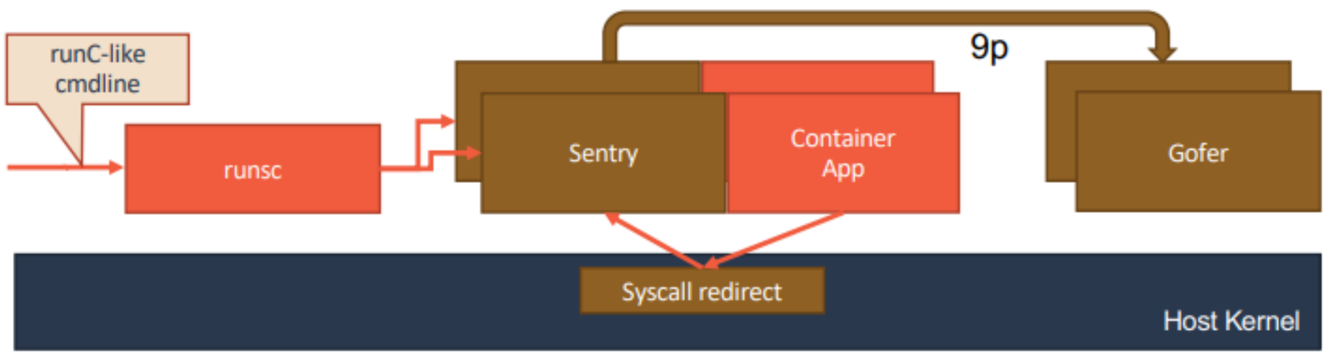
- PTRACE_SYSEMU request
  - causes the traced process to stop on entry to the next syscall

gVisor - Sentry

```
for (;;) {
   ptrace(PTRACE_SYSEMU, pid, 0, 0);
   waitpid(pid, 0, 0);

   struct user_regs_struct regs;
   ptrace(PTRACE_GETREGS, pid, 0, &regs);

   switch (regs.orig_rax) {
      case OS_read:
         /* ... */

      case OS_write:
         /* ... */

      case OS_open:
         /* ... */

      case OS_exit:
         /* ... */

      /* ... and so on ... */
   }
}
```

gVisor - runsc

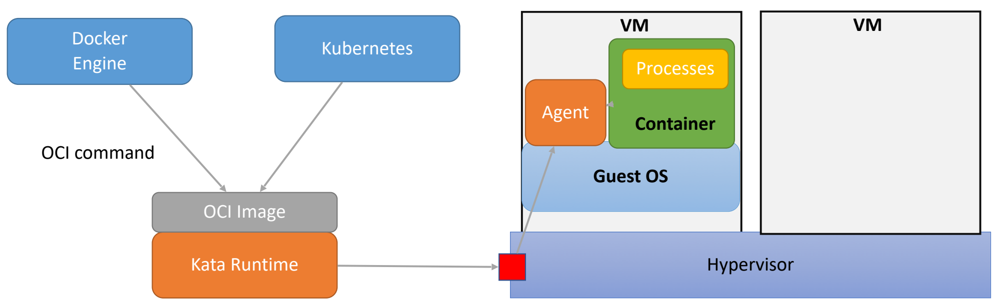
- runsc – the entrypoint to running a sandboxed container
- Implements an OCI runtime specification including:
    - A config.json file contains container configurations
    - A root filesystem



Kata Containers

- Basic Ideas: It’s possible to run containers inside of virtual machines
  - Pretty common deployment method (think about the cloud)
  - Introduces another layer of protection: the hypervisor
  - But notice that Hypervisors can have security bugs as well
- Kata Container
  - Introduced in 2017 from the merger of Intel’s Clear Containers and Hyper’s runV
  - “Wraps” containers into dedicated virtual machines
  - OCI runtime implementation: can be plugged into the container engine (e.g., Docker)
  - Supports existing container images

Kata containers - architecture



References

- 1. How to Implement Secure Containers Using Google's gVisor https://thenewstack.io/how-to-implement-secure-containers-using-googles-gvisor/
- 2. Intercepting and Emulating Linux System Calls with Ptrace: https://nullprogram.com/blog/2018/06/23/
- 3. The True Cost of Containing: A gVisor Case Study: https://www.usenix.org/system/files/hotcloud19-paper-young.pdf
- 4. kata containers & gVisor: https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/presentation-media/kata-containers-and-gvisor-a-quantitave-comparison.pdf
- 5. Kata Containers The way to run virtualized containers: https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/presentation-media/Kata-Containers-The-way-to-run-virtualized-containers.pdf
- 6. Bringing container security to the next level using Kata Containers: https://www.suse.com/media/presentation/TUT1201_Bringing_Container_Security_to_the_Next_Level_Using_Kata_Containers.pdf