Tutorial 12

Gidon Rosalki

2025-06-18

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Containers

1.1 Recap

1.1.1 What are containers / Virtual machines

Virtualisation does not only speak of virtual memory, but also of other concepts. These include when creating storage platforms (e.g. RAID), virtual computer hardware, computer network resources, and so on. Considering RAID, this was virtualised where if we use RAID0, we may have 10TB of physical memory, but the computer thinks we only have 5TB, since all the data is copied. A virtual machine is an emulation of the computer system. The OS thinks that it has direct access to the hardware, but it in fact only has virtualised access to the hardware. There can be many VMs on a single physical system, and each VM may have its own ”virtual” resources and OS.

One can run several VMs at once, on the same hardware, each with its own OS, and all needing to share the RAM, CPU, hard disk, and so on. However, this uses a large number of CPU cycles, with a huge amount of overhead. For this we have containers. A container is an OS level virtualisation. A program that runs inside of a container can only see the container's contents, and devices assigned to the container. Multiple containers can run on the same machine, and they share the OS kernel with other containers.

1.1.2 Linux Containers - LXC

LXCs are used to run multiple isolated Linux systems, on a single host running a single Linux kernel. These have the important elements of

- Namespaces = What the containers see (resource isolation)
- Cgroups (Container groups) - What the containers can use (resource limitation). If we want to limit the amount of memory, or CPU cycles, or something, then we use cgroups.
- What containers can do (permissions). These can be used to limit access to files, and how they may access the files, such as read write, and so on.

Containers are just processes with isolated views, and limits.

1.2 Namespaces

1.2.1 Introduction

Namespaces define what of the kernel resources the process inside the container can see. Consider running the command `ps -ef` on your host machine (Linux master race). You will be able to see all the processes running on your machine when you do this. However, should this be done inside a container, then you will only see the process ids inside the container. This also applies to hostnames, user ids, file systems, network access, IPC access, and so on. The term namespace is often used for a single type of resource, but can also be used for a set of resources. A container can be limited to only view a subset of the resouces.

1.2.2 Isolation

One uses the functions `clone()` or `unshare()` to create new namespaces.

```c
int clone(typeof(int (void *_Nullable)) *fn,
              void *stack,
              int flags,
              void *_Nullable arg, ...
```

This creates a new process running the function `fn`.

`int unshare(int flags);` - This changes the permissions of an existing thread / process.
In both of these functions, `flags` are a list of context to share / unshare.
Common types of namespace isolation is the process id, the unix timesharing system, and the network.

Processes can create new namespaces, and join different existing namespaces. The host Linux system starts out with a single namespace of each type, used by all processes.

We can also control the filesystem, and hostname isolation. We may use the functions `chroot` or `pivot_root` to isolate the root filesystem. This way, we do not share our root file system with every container we might want to run on our machine. We can also mount /proc manually inside containers, and set unique hostnames with `sethostname()`

1.2.3 Setting a new process ID's namespace

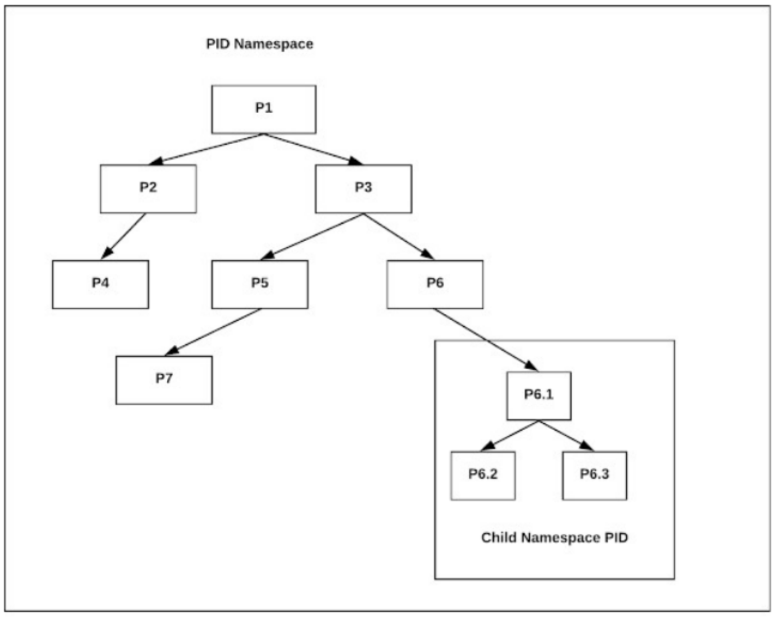

Figure 1: Namespaces

Here we see that all the processes have a shared PID namespace, aside from those descended from P6, which have been assigned their own namespace system.

1.3 Cgroups

Control groups (cgroups) define what the container can use. For example, we might limit the number of processes inside the container to 100, or the maximum memory usage to 1GB. We can also limit how much CPU it can use, or perhaps how much network bandwidth, or even what parts of the network it can access (can it access the internet or no), and so on.

1.4 Capabilities

These define what the container can do. Perhaps we want to change the host's network settings (CAP_NET_ADMIN), or mount filesystems on the host (CAP_SYS_ADMIN), or read/write protected files (CAP_DAC_OVERRIDE), and so on. We drop these capabilities in our container before starting it, and so define what it can, and cannot access.

1.5 Key takeaways

The overhead of a container is significantly lower than that of a VM, VMs are unusable on weak computers, but containers will work excellently. Containers can also be modelled as a type of process, and a minimal container is simply the namespaces, cgroups, filesystem, and capabilities. The tools `runc`, `nsjail` both build containers this way.

2 IPC - Inter Process Communication

Communication between processes is not straightforward. Each process only sees its own allocated memory. So to enable this the OS provides mechanisms to allow processes to communicate with each other.

We have already seen signals, but signals are inefficient, since they only transmit binary bits, perhaps we want to send more information. We also saw sockets, and one can use them to transmit much information, and they are also used in a more general setting, but generally today, one uses pipes, file systems, and shared memory.

2.1 Pipes

This is a stream based IPC channel (unidirectional, or bidirectional). It allows the output of one program to be redirected to the input of another program. The use case is streaming data between processes (for example, pipes in the shell). The has the benefits of it being simple, an OS managed method of synchronisation, and good for message passing. However, this comes with the downside of not being persistent, if you miss some information, you can never get it back, and having a limited buffer, which can easily have too much information in it, and thus be lost. Finally, there is no random access, suppose 1GB of information is transmitted through a pipe, but we only need the information at the `0x1000` mark, we have to read all the information before it first.

2.2 Shared file

This is a disk file that is accessed by multiple processes. This is used for **persistent** data sharing, even after a process finishes, it will remain there. This is often used for configuration files, and logs. This is useful for logs, since even when a process fails, we can see how far it got before it failed. However, this comes with the downside that disk IO is **much** slower, and it will also need file locks, so that two programs cannot try and write, or one read, and one write, simultaneously. Since mutexes do not exist for files, we can create something very similar. We can create a shared variable called `allow_write`, wrap it inside a mutex, and only write when it is able to be taken.

2.3 Shared memory

This is a memory region that is shared by multiple processes. This is useful for high speed, real time IPC on the same machine. This has the benefits of being incredibly fast, due to being in the RAM. It is also efficient for large data, since we have random access to it. However, this comes with the downside of being volatile, should my computer suddenly shut down, I will lose all the information therein. It also needs manual synchronisation, we have to ensure that two processes do not try and write to the same bytes, at the same time. Finally, this system is local only, this will only work on a single computer, in contrast to a shared file, which can be shared across a network. Another downside of shared memory is that, if we do a lot of `mallocs` in a traditional program, or `new` in cpp, when this program closes, and we have not called free, then the OS knows to free this memory automatically. However, if we do this in shared memory, then this memory can become a zombie, since the OS does not know that it can now be freed.

2.4 Summary

In conclusion, shared memory is fast, and volatile, but requires manual synchronisation which is difficult. A shared file is persistent, but slower, however has a very simple setup. Pipes are stream based, meaning all the output of one program will become the input of another, are not persistent, and are not particularly efficient, but are very easy to use.

3 Exercise 5

We will measure communication in overheads in different settings. We will create 2 processes on a single machine, and measure communication speed between them, using pipes, or shared memory, or shared files, or every other method that we have learnt. We will then contrast this by creating two containers, and measuring the communication speeds between them. Finally, we will have a container open in Azure, and one on our local machine, and then measure communication speeds between them. We will get to see what permissions this requires, and how significant the slowdown is due to the communication over the input.