Image taken borrowed from here

Understanding Containerization By Recreating Docker

Daniel Mitre Follow
Jun 14, 2019 · 11 min read

· · ·

Let's suppose we have an online store with some server-stuff that you host on *Amazon Web Services* or maybe on *Google Cloud Platform*. We don't want to spend a lot of money when just a few people are accessing our site, but our site may became viral few times a year (let's say at black friday), how many resources we should buy extra to our site still available at this access peak?

If you're familiar with these cloud technologies, you already know they have an option of auto-scaling that just works for our purpose, but how it works? Maybe it uses **virtual machines, but they are slow** to configure/replicate and actually spend a lot of recurses to make all the hardware virtualization, etc. So this is probably done with containers, which are lighter, faster and much easier to manage.

TL;DR: Containers are a way to isolate programs from the real system that it's in. It's relies on linux features that provide isolation and let us manage resources. In this article we will try to understand how it works while creating our containerization program.

. . .

Scope and prerequisites

In this article we will show the steps that we took to create our containerization program doqueru-kun. Unfortunately took me much time to write this, so may have some inconsistency between topics, please let me know if you find any.

We will give an introduction of the linux namespaces we will use and show how it is possible to isolate it from the rest of the system, but you may find (as we do) LiveOverflow's approach far more entertainer.

We choose to implement in C/C++ to call the system calls directly, without the abstractions that some language or library may do. In this fashion, we learned a lot about some some namespaces and system calls linux implementations.

A lot can be done to isolate a system. In this article we are dealing with the isolation of: **process**, **hostname**, **filesystem**.

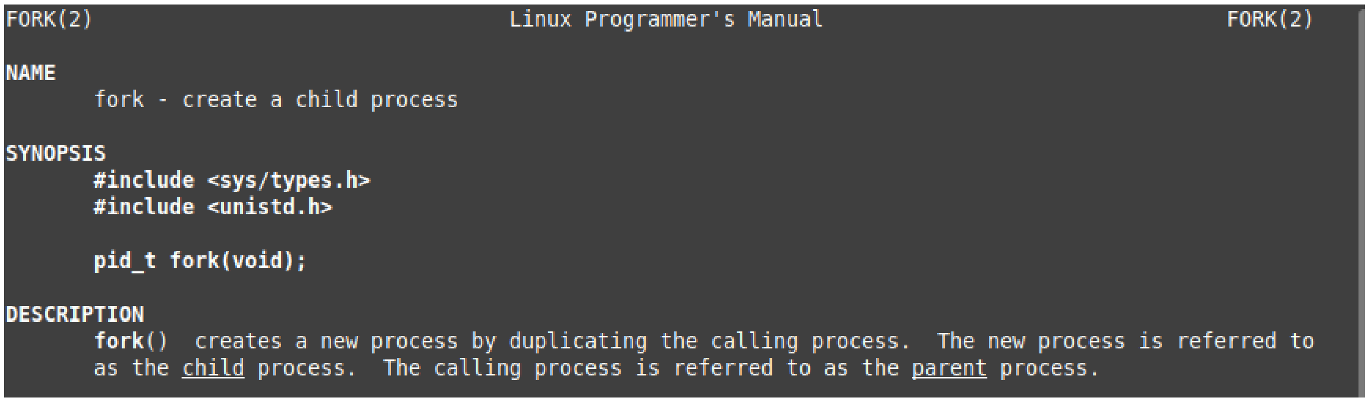
We are **NOT** dealing with the isolation/bounding of: **network**, **cpu usage**, **memory usage**, **I/O rate** limits, **disk usage**, etc. But most of those bounds can be controlled by using cgroups. We already use cgroup in doqueru-kun, so maybe I'll post about them later.

> *Note: The network isolation is a problem that we are discussing how to deal with. If you have an opinion about it, please open an issue on our git.*

. . .

fork()

fork and **exec** are the most important system calls (syscall for short) in the *process* genesis.

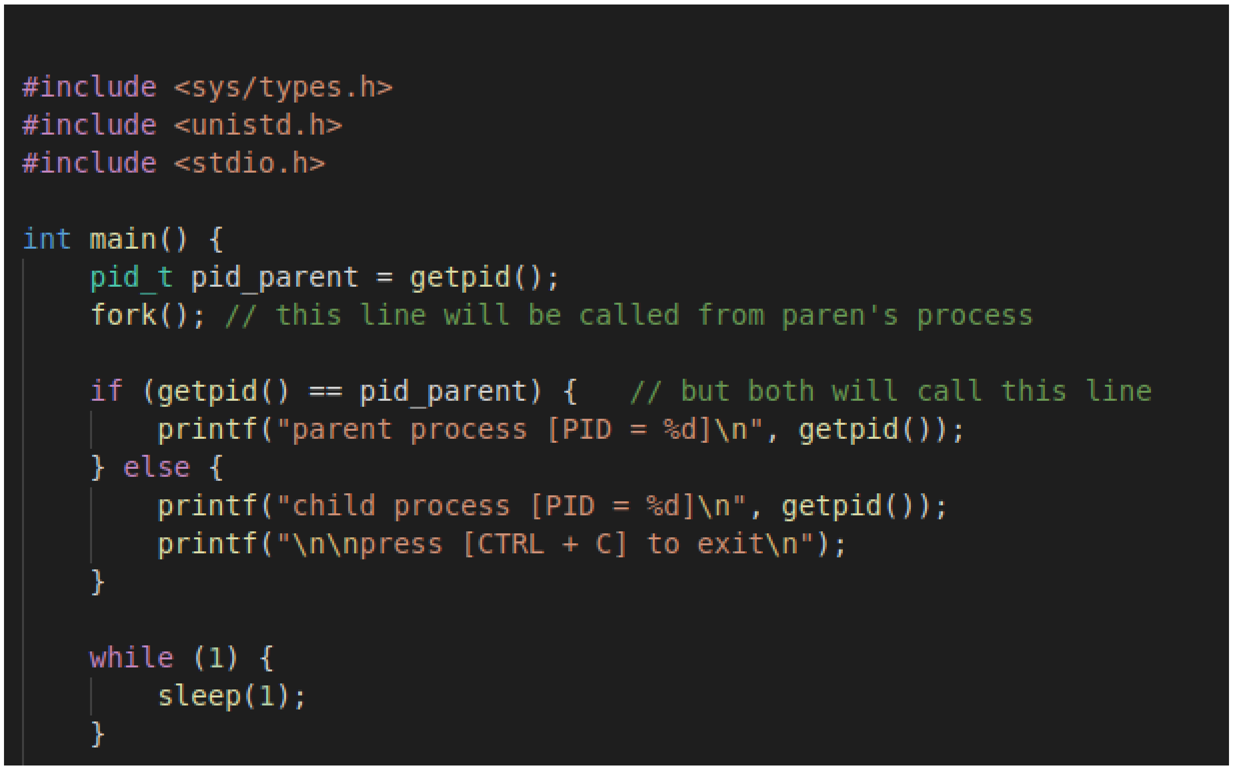


```
FORK(2)                          Linux Programmer's Manual                          FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates a new process by duplicating the calling process.  The new process is referred to
       as the child process.  The calling process is referred to as the parent process.
```

fork(2) on **Linux Programmer's Manual**

fork(): it creates a new process which will be *the image and likeness of his caller*, it will run the very same code starting at the point the caller program was after calling fork(). Very small changes are made to this new process, the important ones to us is that it has its own PID and his caller will be associated as his parent (and so, it's one child of the parent process).

With just this difference, we can make two different programs using the same code:

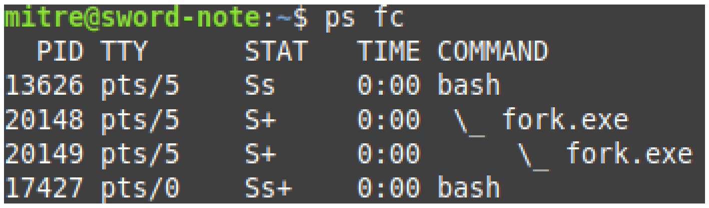


```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid_parent = getpid();
    fork(); // this line will be called from paren's process

    if (getpid() == pid_parent) {   // but both will call this line
        printf("parent process [PID = %d]\n", getpid());
    } else {
        printf("child process [PID = %d]\n", getpid());
        printf("\n\npress [CTRL + C] to exit\n");
    }

    while (1) {
        sleep(1);
    }
```

```
    return 0;
}
```

fork() example usage

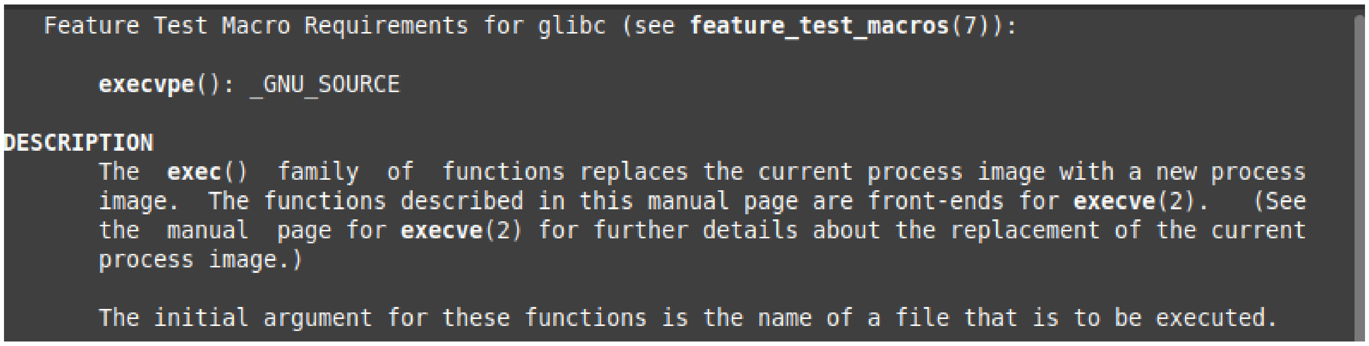
Before closing this program, we can visualize the hierarchy of our processes with **ps fc**, it will show that our program has another one with the same name, which is his child.



```
mitre@sword-note:~$ ps fc
  PID TTY      STAT   TIME COMMAND
13626 pts/5    Ss     0:00 bash
20148 pts/5    S+     0:00  \_ fork.exe
20149 pts/5    S+     0:00      \_ fork.exe
17427 pts/0    Ss+    0:00 bash
```

With these knowledge in mind and with the image aside you can guess that bash is also calling **fork()** to initiate a new process. but how it would be possible if the actual code of the programs are not in the bash's code?

exec()



```
   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       execvpe(): _GNU_SOURCE

DESCRIPTION
       The  exec()  family  of  functions replaces the current process image with a new process
       image.  The functions described in this manual page are front-ends for execve(2).   (See
       the  manual  page for execve(2) for further details about the replacement of the current
       process image.)

       The initial argument for these functions is the name of a file that is to be executed.
```

exec(3) on **Linux Programmer's Manual**

While the **fork** is called once and returns twice (**0** on child and **child's PID** on parent) the **exec** actually returns not once (actually, both returns once if an error has occurred).

It simply substitutes the executing code of the calling process with the code an specified executable.

> *Note: The **exec** is a family of functions that does the same thing of different ways or with different parameters, so we will using **execvp** from this project. **execvp** expects an string with executable's path and an pointer to the args.*

To test quickly and advance things a little bit, let's make an file called test.cpp that just prints the process PID before dies and let's compile it to **test.exe**, we will use this file to check our container progress as we develop it.

And after cleaning some code we have an program that calls another program (sounds like making a terminal shouldn't be so hard after all) and then turns itself into the same program.

So, now we know how to **fork()** and **exec()** to create another programs. This program we wrote will became our container and every program we call inside it should not affect our real system.

. . .

Defensive programming

With our current program we can run any executable by passing his location, so we can run **/bin/bash** for example and execute several commands and mess around (type **exit** or press **CTRL+D** to close it).

But inside the program we still are in our real system, so whatever happens still reflects in our system and this is why we need to proceed carefully at each step.

> *To put a real example: at some moment we will totally replace our "/" from the system after we isolate the process in many ways this doesn't affect the real workspace. If any step of this process have not worked, we have to know that before the next step.*

Linux provides an header called **<errno.h>** and all syscall we invoke have a way of telling if things gone wrong. Generally, it will return -1 if have some error, and we can check in **errno** the error code.

So, after writing some C macros and our own assert(), after we call any syscall we call, let's assert that everything is working as expected before move on.

. . .

unshare() and namespaces

In the unshare's manual page we have a list of unshareable namespaces (we can read about namespaces on clone's manual page). Let's stick with the UTS, PID and MNT namespaces for now.

- **UTS (UNIX Time Sharing)**: This namespace isolates the properties we can get from uname syscall like operating system's name, version, user's and domain's name, etc.

- **PID (Process ID)**: This one isolates the PID tree, that one we visualizate with ps -auxf, [spoiler] this will be a very interesting one [/spoiler].
- **MNT (Mount)**: So we have also a namespace with all mount points, cool. I actually did not care to much about it before, until I found out how many linux mechanisms relies on this… I will talk deeper about it, but let's not put the cart before the horse.

UTS namespace

As mentioned above, the **UTS namespace** isolates some properties. It's a very simple namespace and will help us to dive deeper into linux documentations and programer's manual.

Let's explore what we can do by creating an new UTS inside the container. On the clone's manual is mentioned **CLONE_NEWUTS** flag, and the **sethostname()** function. In the end of the section, is also mentioned that only **CAP_SYS_ADMIN** can create a new UTS. We are not familiar to this command yet, so we are going to cover each of this keywords.

The **clone()** syscall have a lot of flags meant to create a process with the ability of isolate it's namespaces and these flags are used also in **unshare()** syscall, **CLONE_NEWUTS** is one of them. The flags are integers where each bit has a different meaning, so we can bitwise-or flags to combine them.

sethostname() is just a system call that changes the hostname of the processor (similar to hostname command), we will use this to set our container's hostname. We can visualize this trough the **uname -n** command or **gethostname()** function.

Function that creates a new UTS namespace and set a hostname to "doqueru".

So, adding this little function on our program we isolated our real UTS namespace. If any process inside it changes something about it, it will have no effects in our computer.

But… if you tried to run the code as before, you can get an **EPERM** error. This is what **CAP_SYS_ADMIN** requirement was warning us about, calling our program with sudo privileges is needed from now on.

> *Tip: if you get an error like this running a syscall, it's a good idea start looking for the reference of the error in the man page of the syscall. Seeking for the ERRORS section on* ***man 2 unshare*** *you will find that EPERM error will be throwed if "the calling process did not have the required privileges for this operation". Using* ***errno*** *terminal command to see the name or description of the error may help you too.*

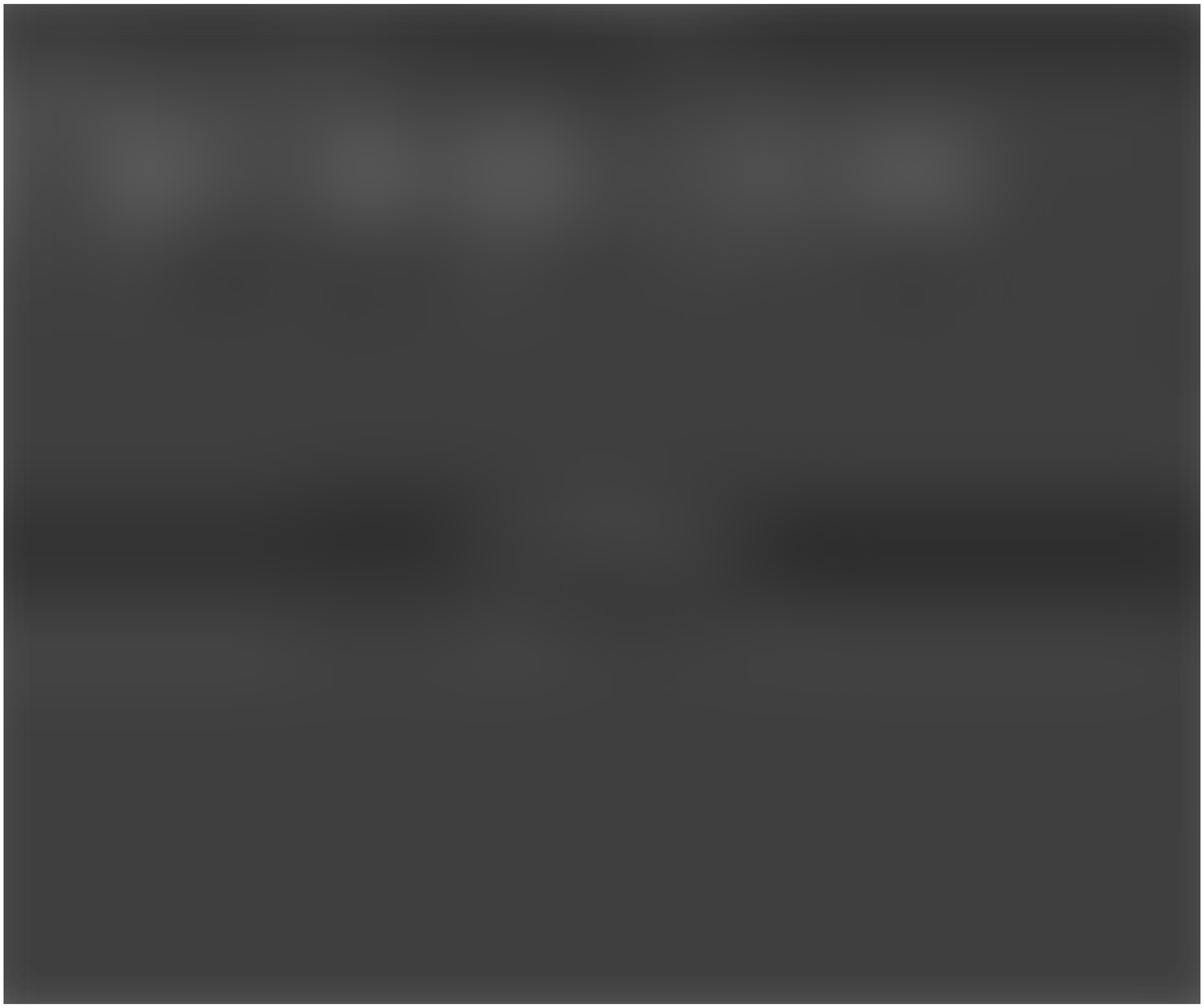
PID namespace

Inside our container, we still have access to all others processes and can interact with them, by creating file descriptors using their PID or sending signals to them (with kill for example). Now we will avoid things like this by creating a new PID namespace.

Manipulating linux's process tree may be tricky, but seems very clever and simple after reading the documentation.

Every process must have a parent, thus we have a process tree where the root is the process with PID 1 (let's forget about the one with PID 0). The root is called **init** and have an special role on PID namespace: this process will adopt all orphan children processes (happens when his parents ends before them) and is essential for the operation of creating new processes. If it dies, then no processes can be created and all processes of namespace will end with an **SIGKILL** signal sent by the kernel.

When unsharing with **CLONE_NEWPID** flag the calling process will remain in it's current namespace, but its children created after the **unshare** will be into a new **PID namespace**.



process tree and it's unshare visualization

In the *gif* above we can visualize that a process with the **PID namespace** unshared (bottom terminal) cannot see processes from outside, but the entire namespace is visible from the parent's **PID namespace** (top terminal). Is also notable that the same processes are seen with different PID in different namespaces. I used the *mount-proc* argument at **unshare** command, this unshares **mount namespace** and mounts the *procfs* at **/proc**, we will get there.

Backing to code: we will need to call **unshare(CLONE_NEWUTS | CLONE_NEWPID)** to combine the flags. Our first child after this could be the executable we get as parameter, but this will be the **init** process and we shouldn't trust arbitrary code to hold this responsibility. If we **unshare** before **forking**, our fork() will already produce the **init**. Let's then call fork+exec inside our "**init**" to preserve it and make it wait all children die (we already has done this with **wait** syscall on the **main** function).

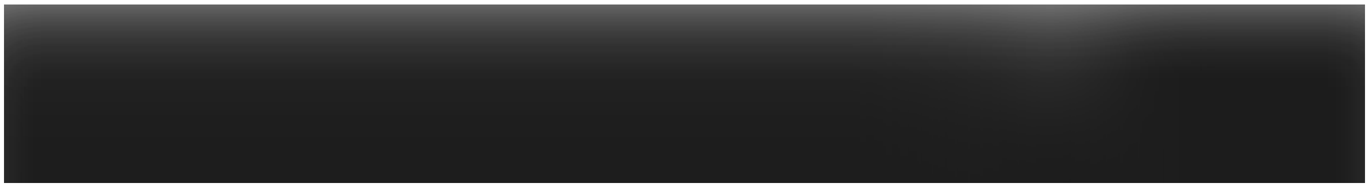
MNT Namespace

The mount namespaces are a way to link devices, filesystems and directories in a way that we can reach them from the filesystem. Mounts are actually pretty tricky, but it's mainly about this.

So now we learnt enough to just straight-forward here, right? We should add **CLONE_NEWNS** flag (yeah, they didn't expect that would have more namespaces in the future) on **unshare** and we're just good. Or aren't we?

Mounting on shared subtree

Whooops, we used mount to create a link into to / in the **./link** directory, but when we exited the container the link persisted. The problem here is that even unsharing, our subtrees are shared. We should so make them private mounting with **MS_REC** and **MS_PRIVATE** flags.



Mounting on private subtree

Good enough, moving forward. You may have read that the PID namespace's manual says:

> *"After creating a new PID namespace, it is useful for the child to change its root directory and mount a new procfs instance at* /proc *so that tools such as ps(1) work correctly".*

We needed that mount to not affect our main system, we already accomplished this, so we can already do this (inside the new PID namespace), but instead of change root with **chroot**, let's do some overengineering it with our final boss: El Pivoto-Rooterino.

Pivot root

Well, we had chroot thing which archives the same purpose of course, but that would be too easy (and unsafe!). Our choice then was to use the pivot_root() syscall and we regret that for a long time, until we exhausted all information sources and finally made a code that works properly! After you see the code and our explanation we hope this to be a piece of cake to you, because we really struggled with the lack of information about this procedure.

entry of **man 2 pivot_root**

First of all, the signature of pivot_root is **int pivot_root(const char *new_root, const char *put_old)**, but there's no wrapper for it, so we should call it as **syscall(SYS_pivot_root, new_root, put_old)**.

Pre-conditions to both **new_root** and **put_old**: they must be directories, they must not be in the same filesystem of the current /, **put_old** must be underneath **new_root**.

Pivot rooting:

- We will download a small linux (Alpine was our choice as well as Docker's choice) and put it unzip it at our **new_root**
- We will make **new_root** to be the mount point of a new filesystem (this is done by bind mounting the folder in itself)
- After that, let's create **put_old** somewhere inside **new_root**
- We can now **pivot_root(new_root, put_old)**
- Let's and the script precisely by **chdir** into /
- And finally, we will lazily umount (with **MNT_DETACH**) the put_old directory to leave no way of escaping from the container restricted filesystem.

That's it. This sequence seems quite trivial seeing now, but we've failed a lot in our tries, some of the failures was actually because our mounted filesystem weren't private and the semantics of using **mount** isn't trivial at first glance.

. . .

Thanks for the reading (or intention of reading) feel free to contribute with us at github, and to contact me if you found something wrong in this article.

. . .

EDIT 26 feb 2020: LiveOverflow just posted a video mentioning this article. I highly recommend this series of his for the educational content, but I'm also very happy and proud so this recommendation is also a little *show-off*.

Namespaces Linux Kernel Containerization Containers Docker