

## Tecnologias para Desenvolvimento de Sistemas

# Implementação de Sistemas em Containers

## RunC, LXC, Docker e Podman

Vitor Mayorca Camargo

<https://www.youtube.com/watch?v=85k8se4Zo70>

<https://www.youtube.com/watch?v=-94T26MxQXM>

### 1. Sistema Operacional Utilizado Para Suportar os Contêineres:

O sistema operacional utilizado durante a maior parte do trabalho foi o MX-Linux 23 “*Libretto*” (baseado em *Debian*), instalado dentro do software “VirtualBox”, com a extensão “Oracle VM”. A ISO foi instalada diretamente do site oficial da distribuição (<https://mxlinux.org/download-links/>).

Para executar o docker, foi utilizado o sistema operacional Windows 11.

### 2. Pacotes Instalados Durante o Trabalho:

- Git;
- RunC;
- debootstrap
- docker.io
- docker
- lxc
- uidmap
- dbus-user-session
- podman

## Parte 1 – Implementação de Containers Nativos Usando RunC

Objetivo: Implementar um container usando a tecnologia “RunC”. Configurar um rootfs e explorar um config.json.

Inicialmente, o pacote “RunC” foi instalado no terminal do MX-Linux, usando o comando “sudo apt install runc”.

```
radajaa@roboroto:~$ sudo apt install runc
```

Agora, iremos criar um diretório que usaremos para armazenar os arquivos do runc. Farei isso dentro da pasta home do meu usuário. Para isso, usei dois comandos: “`cd /home/radajaa`” para acessar a minha pasta /home, e “`mkdir runc`”:

```
radajaa@roboroto:~$ cd /home/radajaa
radajaa@roboroto:~$ mkdir runc
radajaa@roboroto:~$ ls
assemblyteste Desktop Downloads Music Public Templates
assembly.zip Documents GitHubDesktop-linux-3.1.1-linux1.deb Pictures runc Videos
radajaa@roboroto:~$
```

E dentro de `./runc/` adicionaremos outro diretório chamado `rootfs`, aonde configuraremos nosso `rootfs`. Para o presente trabalho, usei uma `busybox` exportada via Docker. A `busybox` foi exportada com o comando “`docker export $(docker create busybox) | tar -C rootfs -xvf -`”

Agora, temos um diretório rootfs já configurado. Ainda no diretório runc, criamos o arquivo config.json usando o comando “runc spec”. Dentro de config.json, coleí um código de configuração genérico obtido na internet. Também fiz algumas alterações nele: em hostname, alterei o valor para “vitor”:

```
"root": {
  "path": "",
  "readonly": false
},
"hostname": "vitor",
"mounts": [
```

Também alterei as permissões octais do contêiner, para que qualquer usuário tenha permissões completas sobre o arquivo.

Agora, o contêiner RunC está pronto para ser usado! Para executá-lo, usamos o comando “sudo runc run mycontainerid”. “mycontainerid” é o nome do contêiner busybox que exportei do docker.

Como o busybox não vem com um instalador de pacotes padrão, não é possível usá-lo para fazer tarefas mais complexas. Porém podemos testar as suas funcionalidades: podemos criar pastas com “mkdir”:

```
root@roboroto:/home/radajaaj/TrabalhoTDS/runc# runc run mycontainerid
/# mkdir teste
/# ls: "proc"
bin  dev  etc  home  lib  lib64  proc  root  sys  teste  tmp  usr  var
/# s
```

Ou até deletar todos os arquivos com “rm -rf /”:

```
/ # ls
sh: ls: not found
/ # mkdir test
sh: mkdir: not found
/# cd "proc",
sh: cd: can't cd to /root: No such file or directory
```

## Parte 2 – Implementação de Containers LXC Usando Virt-Manager e LXD

Objetivo: Configurar um sistema para suporte a LXC usando virt-manager, com as templates busybox e debian; configurar um sistema para suporte a LXC usando LXD com a imagem debian/12.

Inicialmente, instalei os pacotes lxc, lxcctl, lxc-templates e virt-manager. Primeiro vamos criar o container debian. Para isso executamos o comando “sudo lxc-create -n container-debian -t download”, aonde “container-debian” é o nome do container. Durante a instalação, o lxc irá pedir para que sejam inseridas algumas informações básicas do sistema operacional a ser instalado, como a distribuição, a versão e a arquitetura da máquina. Aqui, optei por instalar a versão mais recente do debian.

```
---
You just created a Debian bookworm amd64 (20231207_05:24) container.
type: "proc",
To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
```

Também criei um container busybox com o comando “sudo lxc-create -t busybox -n busyboxlxc”.

Ao tentar rodar os dois contêineres com o comando “sudo lxc-start -n nome\_do\_container”, obtive o seguinte erro:

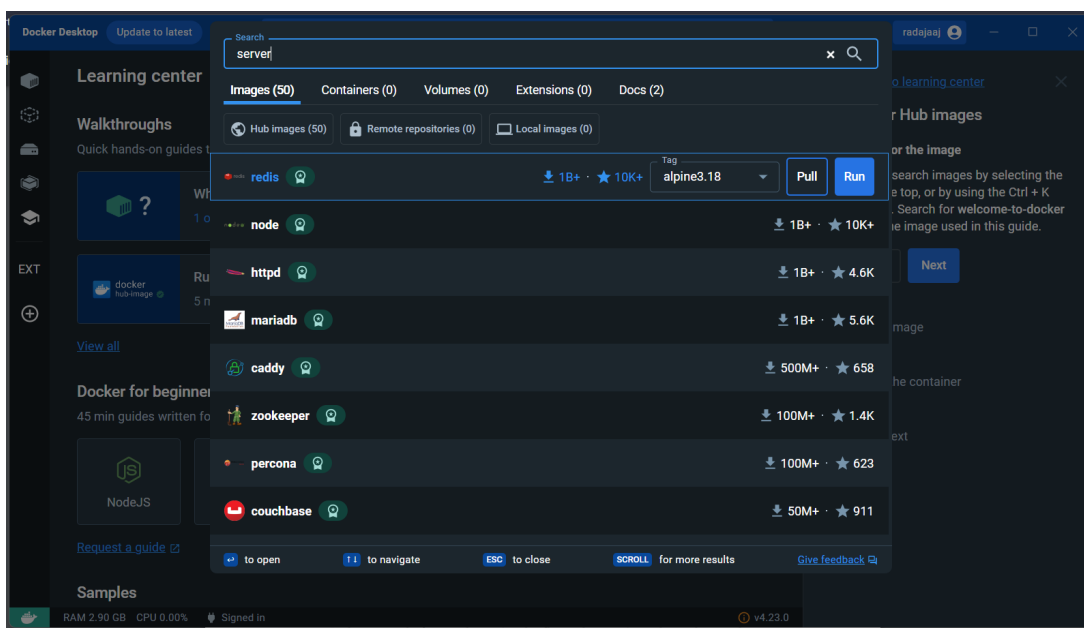
```
lxc-start: container_debian: ../src/lxc/tools/lxc_start.c: main: 266 No container config specified
```

O qual não fui capaz de resolver.

## Parte 3 – Implementação de Containers usando Docker

Objetivo: Configurar o sistema para que ele suporte docker, e explore um dockerfile. Configure o sistema para suporte a Docker rootless e explore um container rootless.

O docker engine, docker server e docker desktop foram instalados diretamente do site oficial do docker, dentro do sistema operacional Windows 11. Dentro do docker hub, temos acesso a uma imensa biblioteca de imagens de contêineres, prontos para serem instalados e usados:



Optei por criar um container contendo um servidor genérico de NodeJS usando um dockerfile. Para isso, criei uma pasta chamada “testedocker” dentro da Área de Trabalho. Nela, criei um arquivo sem extensão chamado “Dockerfile”, e nele escrevi:

```
Dockerfile
Arquivo  Editar  Exibir

# Use an official Node.js runtime as a parent image
FROM node:14

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json to the working directory
COPY package*.json ./

# Install app dependencies
RUN npm install

# Copy the application code to the container
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Define the command to run your app
CMD ["npm", "start"]
```

Depois, abri o prompt de comando dentro dessa pasta, e executei o comando “docker build -t nodesrv:test .”, para criar a imagem do container. Para executá-lo, usei “docker run -d --restart always -p 3000:3000 nodesrv:test”. A flag “--restart always” foi necessária pois o container estava tendo um tempo de vida muito curto. Aqui podemos ver ele em funcionamento:

```
C:\Users\lvito\OneDrive\Documentos\Área de Trabalho\testedocker>docker run -d --restart always -p 3000:3000 nodesrv:test
8ba810923cbc15352ecd8b0096c49f434c6b5ce933dd45a08c576c4c521e6266

C:\Users\lvito\OneDrive\Documentos\Área de Trabalho\testedocker>docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS              PORTS                               NAMES
8ba810923cbc   nodesrv:test "docker-entrypoint.s..." 5 seconds ago  Up Less than a second  0.0.0.0:3000->3000/tcp             jovial_cray

C:\Users\lvito\OneDrive\Documentos\Área de Trabalho\testedocker>
```

Para a criação do docker rootless, precisei retornar ao sistema operacional MX Linux. Dentro dele, precisei instalar o docker com o seguinte comando: “sudo apt install docker.io”. Para facilitar o trabalho, adicionei o meu usuário ao grupo “docker”, para que eu pudesse manusear dockers sem precisar de sudo: “sudo usermod -aG docker \$USER”. Agora, podemos iniciar o serviço docker com o comando “service start docker”.

Seguindo a documentação do docker, instalei os pacotes “uidmap”, “dbus-user-session”, “fuse-overlayfs”, “slirp4netns”, “docker-ce-rootless-extras”. Por fim, foi necessário rodar o comando “dockerd-rootless-setup.sh install” como um usuário comum.

Infelizmente, não fui capaz de executar o docker no modo rootless dentro do MX Linux. Isso muito provavelmente aconteceu pois o sistema operacional está sendo hospedado dentro de uma máquina virtual (Oracle VM VirtualBox) e, por conta disso, muitos serviços se tornam inacessíveis, e o erro a seguir aparece com frequência:

```
$ systemctl
System has not been booted with systemd as init system (PID 1). Can't operate.
Failed to connect to bus: Host is down
```

Até o momento, não fui capaz de resolver o problema.

## Parte 4 – Implementando Containers Podman

Objetivo: Configure o sistema para suporte a Podman e explore um container rootless.

Primeiro, instalei o pacote podman com o comando “sudo apt install podman”. Podman suporta contêineres rootless por padrão. Porém, decidi criar um usuário novo com permissões rootless para o podman seguindo o tutorial fornecido pela Red Hat:

[developers.redhat.com/blog/2020/09/25/rootless-containers-with-podman-the-basics#example\\_using\\_rootless\\_containers](https://developers.redhat.com/blog/2020/09/25/rootless-containers-with-podman-the-basics#example_using_rootless_containers).

Agora, vamos buscar uma imagem de linux alpine para executar no podman:

```
$ podman pull alpine
Resolved "alpine" as an alias (/etc/containers/registries.conf.d/shortnames.conf)
Trying to pull docker.io/library/alpine:latest...
Getting image source signatures
Copying blob c926b61bad3b done
Copying config b541f20801 done
Writing manifest to image destination
Storing signatures
b541f2080109ab7b6bf2c06b28184fb750cdd17836c809211127717f48809858
```

E por fim, tentei executar o container com “podman run --rm -it alpine”. Porém, obtive o seguinte erro:

```
Error: OCI runtime error: runc: runc create failed: mountpoint for devices not found
```

Aparentemente, o runc entrou em conflito com o podman. Isso foi resolvido depois que removi o pacote do runc com “sudo apt remove runc”.

Tentando de novo...

```
$ podman run --rm -it alpine
/# ls
bin    etc    lib    mnt    proc  run    srv    tmp    var
dev    home  media opt    root  sbin   sys    usr
```

Agora o container funcionou! Para explorar as suas capacidades, irei instalar os pacotes “nano”, “nasm” e “binutils”, com os comandos “apk add nano”, “apk add nasm” e “apk add binutils”. Com o comando “touch hello.asm”, criei um arquivo



‘.asm’. Nele, irei escrever um código em assembly x86\_64 e tentar executar ele no container:

```

#define maxChars 20
section .data
    str0la : db "Hello?", 10, 0
    str0lal: equ $ - str0la

    strBye : db "Voce disse ", 0
    strByel: equ $ - strBye

section .bss
    strLido : resb maxChars
    strLidoL: resd 1

section .text
    global _start

_start:
    mov rdi, 1
    mov rax, 1
    lea rsi, [str0la]
    mov edx, str0lal
    syscall                ;mensagem de oi

    mov rax, 0
    lea rsi, [strLido]
    mov edx, maxChars
    syscall                ;leitura do input do usuario

    mov [strLidoL], eax

    mov rax, 1
    lea rsi, [strBye]
    mov edx, strByel
    syscall                ;mensagem de saida

    mov rax, 1
    lea rsi, [strLido]
    mov edx, strLidoL
    syscall

fim:
    mov rax, 60
    mov rdi, 0
    syscall
```

Montamos o código:

```

/ # nasm -f elf64 hello.asm
/ # ld hello.o -o hello.x
```

E agora executamos ele:

```

/ # ./hello.x
```

Mas o que ele faz? Basicamente, ele vai pegar um input do usuário e jogar de volta no terminal:

```

/ # ./hello.x
Hello?
I love podman
Voce disse I love podman
/ #
```

## Parte 5 – Upload no CodeBerg

Por fim, todos os códigos foram enviados para a plataforma CodeBerg por meio de um repositório do git. O git foi instalado nas máquinas com o comando “sudo apt install git”. Depois, o repositório foi clonado com “git clone <https://codeberg.org/radajaaj/TrabalhoTDS.git>”. Por fim, cada diretório foi inserido no repositório com uma commit diferente.

Link do repositório: <https://codeberg.org/radajaaj/TrabalhoTDS>