UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Implementações de Containers em Sistemas Operacionais GNU/Linux

Marcos Vinicius Barros de Lima Andrade Junqueira

São Carlos – SP

Implementações de Containers em Sistemas Operacionais GNU/Linux

Marcos Vinicius Barros de Lima Andrade Junqueira

Orientador: **Prof. Dr. Francisco José Monaco**

Monografia final de conclusão de curso apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como requisito parcial para obtenção do título de Bacharel em Ciências de Computação.
Área de Concentração: Sistemas Operacionais

USP – São Carlos
Novembro de 2019

Junqueira, Marcos Vinicius Barros de Lima Andrade
Implementações de Containers em Sistemas Operacionais GNU/Linux / Marcos Vinicius Barros de Lima Andrade Junqueira. - São Carlos - SP, 2019.
39 p.; 29,7 cm.

Orientador: Francisco José Monaco .
Monografia (Graduação) - Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos - SP, 2019.

1. Sistemas Operacionais. 2. Virtualização. 3. Virtualização A Nível de Sistema Operacional. I. , Francisco José Monaco. II. Instituto de Ciências Matemáticas e de Computação (ICMC/USP). III. Título.

À minha mãe, a mulher mais forte que eu conheço

AGRADECIMENTOS

Os principais agradecimentos são direcionados ao Prof. Dr. Francisco José Monaco pela oportunidade de fazer um trabalho num tema que eu considero interessante e desafiador.

O correr da vida embrulha tudo.
A vida é assim: esquenta e esfria,
aperta e daí afrouxa,
sossega e depois desinquieta.
O que ela quer da gente é coragem
- Guimarães Rosa

RESUMO

JUNQUEIRA, M. V. B. L. A.. **Implementações de Containers em Sistemas Operacionais GNU/Linux**. 2019. 39 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Containers é uma método de virtualização de processos e aplicações do qual tem sido cada vez mais utilizado em computação em nuvem. A entrega dessas aplicações se tornou um quesito obrigatório e cultural, além da orquestração desses containers em larga escala. Este trabalho apresenta a implementação de um container minimalista e para isso, são empregadas funcionalidades disponíveis no kernel Linux que fornecem isolamento e controle sobre os recursos necessário para alcançar tais objetivos.

Palavras-chave: Sistemas Operacionais, Virtualização, Virtualização A Nível de Sistema Operacional.

ABSTRACT

JUNQUEIRA, M. V. B. L. A.. **Implementações de Containers em Sistemas Operacionais GNU/Linux**. 2019. 39 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Containers is a method of virtualization which processes and their applications that has been popular in cloud computing. The deployment of these applications became mandatory and cultural issue, as well as the orchestration of these containers on a large scale. This paper presentes the implementation of a minimalist container system and for this, features available in the Linux kernel are employed that provides isolation and control over the resources need to achieve these goals.

Key-words: Operating Systems, Virtualization, Operating System-level Virtualization.

LISTA DE ILUSTRAÇÕES

Figura 1 – Comparação entre máquinas virtuais com *hipervisores*, *containers* e arquitetura híbrida 22
Figura 2 – Comparação entre os tipos de *hipervisores* 26

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Hello World em Go 29
Código-fonte 2 – Rotina main 30
Código-fonte 3 – Rotina *runContainerSelf* 31
Código-fonte 4 – Rotina runContainerChild 31
Código-fonte 5 – Rotinas setCGroups e createCGroup 32
Código-fonte 6 – Rotina pullImage 34

LISTA DE ABREVIATURAS E SIGLAS

CSP ..... Communicating Sequential Processes
GPU ..... Graphic Processing Unit
IaaS ..... Infrastructure as a Service
IBM ..... International Business Machines
IPC ..... Inter-process Communication
OCI ..... Open Containers Iniciative
PaaS ..... Plataform as a Service
PID ..... Process Identifier
UTS ..... UNIX Time Sharing
VM ..... Virtual Machine
VMM .... Virtual Machine Monitor

SUMÁRIO

1 INTRODUÇÃO 21
1.1 Motivação e Contextualização 21
1.2 Objetivos 22
1.3 Organização 22

2 MÉTODOS, TÉCNICAS E TECNOLOGIAS USADAS 25
2.1 Virtualização: conceitos e técnicas 25
2.1.1 Hipervisores *25*
2.1.2 Requerimentos para Virtualização de Popek Go *26*
2.1.3 Virtualização Total *26*
2.1.4 Virtualização a nível de Sistema Operacional *27*
2.2 Chamadas de Sistema 27
2.2.1 Chroot e pivot root *27*
2.2.2 Namespaces *28*
2.2.3 Cgroups *28*

3 DESENVOLVIMENTO 29
3.1 Linguagem de Programação Go 29
3.2 Implementação de um container usando Go 29
3.2.1 Detalhes de Implementação *29*
3.3 Criando um container 35
3.4 Resultados 35
3.5 Dificuldade e limitações 35

4 CONCLUSÃO 37
4.1 Conclusão 37
4.1.1 Considerações sobre o Curso de Graduação *37*

REFERÊNCIAS 39

Capítulo 1

INTRODUÇÃO

1.1 Motivação e Contextualização

Virtualização é uma das tecnologias fundamentais para provedores e serviços de infraestrutura, contudo, esses serviços se veem cada vez mais na necessidade de acompanhar a transformação digital de seus clientes. Essa transformação acontece pelo fenômeno *Devops*[1], cultura que prega entregas contínuas e rápidas iterações, além de uma automatização constante dos processo (STåHL; MåRTENSSON; BOSCH, 2017).

Na cultura *Devops*, existe uma relação estrita entre métodos ágeis e entrega contínua (LWAKATARE; KUVAJA; OIVO, 2016). O termo *Devops* tem sua origem na discussão de entrega do software[2]. Entre suas práticas estão: automatização de processos, ferramentas de gerenciamento de configuração, lançamento de versões frequentes, iterações ágeis entre desenvolvedores e infraestrutura e entre outros. Exemplo de ferramentas utilizadas: Jenkins[3] e Puppet[4] para automatização, Docker[5] e Kubernetes [6] para containers e orquestração, Terraform[7] e Ansible[8] para infraestrutura como código.

Mesmo sendo ferramenta essencial em Infrastructure as a Service (IaaS) e Plataform as a Service (PaaS), existe um custo na virtualização[9]. Esse custo aparece em vários aspectos: acesso custoso a hardware dedicado, como Graphic Processing Unit (GPU); necessidade de virtualização de todo hardware mesmo que a aplicação nesta Virtual Machine (VM) utilize parte desse hardware; impossibilidade de otimização de hardware para aplicações virtualizadas;. Além disso, virtualização surgiu para remediar várias deficiências presentes no sistemas operacional[10], sendo containers uma das possíveis soluções. Segundo Claus Pahl(PAHL, 2015), virtualização por *containers* é uma tecnologia relevante em computação na nuvem, não só em Iaas mas também em PaaS.

1 <https://queue.acm.org/detail.cfm?id=3338532>
2 <https://www.youtube.com/watch?v=LdOe18KhtT4>
3 <https://jenkins.io/>
4 <https://puppet.com>
5 <https://www.docker.com/>
6 <https://kubernetes.io/pt/>
7 <https://www.terraform.io/>
8 <https://www.ansible.com/>
9 <https://queue.acm.org/detail.cfm?id=1348591>
10 <https://lwn.net/Articles/524952/>

Virtualização por *containers* ou Virtualização a nível de Sistema Operacional surge de ideias já presentes no mercado. Através de técnicas de isolamento e gerencia de processos fornecidas pelo *kernel*, os processos são isolados do sistema operacional e se comunicam através de chamadas de sistema (LAADAN; NIEH, 2010). Além disso, as tecnologias de virtualização por *containers* apresentam funcionalidades que facilitam vários processos, por exemplo *Docker*, disponibiliza um sistema de imagens através de um sistema de arquivos imutável. Dado uma camada desse sistema, que pode conter bibliotecas de distribuição do sistema operacional, é possível construir outras camadas. Essa camada serve de base para a construção de outras imagens, isso permite que *containers* compartilhem a mesma imagem, diminuindo o espaço em disco consumido, como pode ser visto na Figura 1.

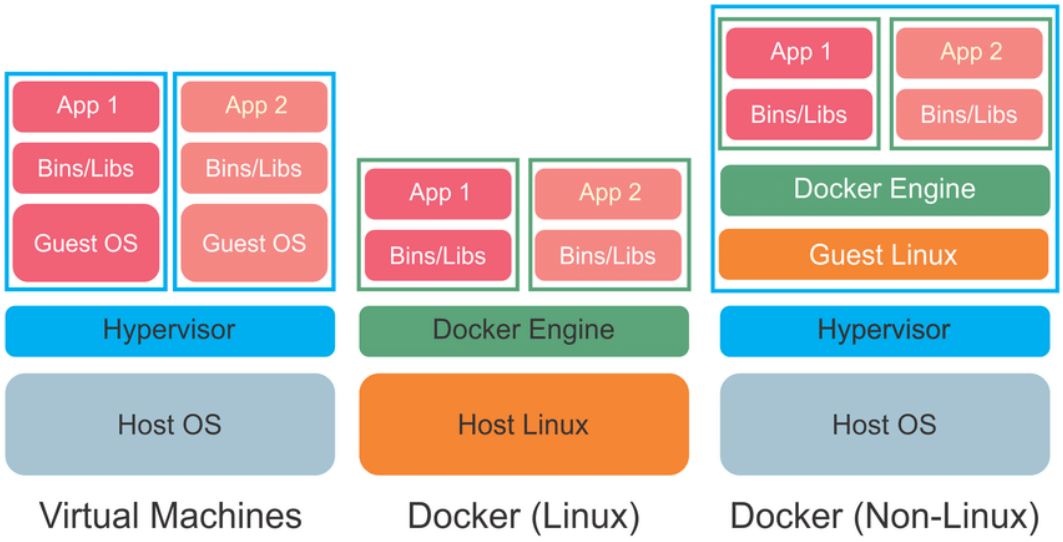

Figura 1 – Comparação entre máquinas virtuais com *hipervisores*, *containers* e arquitetura híbrida.

Fonte: (HUNG *et al.*, 2016)

1.2 Objetivos

Desenvolver um *container* mínimo em código portátil Go, utilizando funcionalidades disponíveis em sistemas operacionais Linux. De forma a demonstrar a facilidade de interação e implementação de um *container* básico através do isolamento de um processo a partir de chamadas de sistemas.

1.3 Organização

No capítulo 2 são apresentados os conceitos e técnicas empregado em máquinas virtuais, além da diferença entre os diversos tipos de virtualização. No mesmo capítulo, caracterizamos as chamadas de sistemas e funcionalidades do *kernel* utilizadas num sistema de *container*. No capítulo 3 é descrita a implementação de um *container* minimalista utilizando essas técnicas. No

capítulo 4 são sintetizadas as conclusões alcançadas pelo trabalho, como também uma crítica e considerações sobre o curso de graduação.

Capítulo 2

MÉTODOS, TÉCNICAS E TECNOLOGIAS USADAS

2.1 Virtualização: conceitos e técnicas

Computação em nuvem é possível graças a evolução e aperfeiçoamento nas técnicas de virtualização. Serviços de *IaaS* e *PaaS* existem graças a essa tecnologia (OLUDELE *et al.*, 2014). Surgiu entre a década de 60 e 70, no contexto de mainframes. Nesta época, a International Business Machines (IBM) investiu recursos massivos em soluções de tempo compartilhado (tradução de *time sharing*). Outras necessidades da época: criação de sistemas multi usuários, maior reaproveitamento dos recursos, isolamento e segurança estão entre as motivações para o nascimento de tecnologias de virtualização. Todas essas necessidades decidiram que de fato, virtualização era uma solução.

Virtualização cria uma plataforma entre o sistema operacional virtual e o real (ZAHARIA, 2010). Essa abstração é chamada de Virtual Machine Monitor (VMM) ou *hipervisor* e é responsável por lidar com o gerenciamento e mapeamento entre o hardware virtual e o físico. Além disso, máquinas virtuais (traduzido de *Virtual Machine*) podem ser classificadas de duas formas: de sistema ou de aplicação. Na primeira, suportam sistemas operacionais completos, na segunda, aplicações específicas.

2.1.1 Hipervisores

Hipervisores ou VMM são considerado a camada responsável por virtualizar todos os recursos da máquina física, assim definindo e dando suporte a execução de múltiplas máquinas virtuais. Existem dois tipos de *hipervisores*: tipo 1 (nativos) e tipo 2 (convidados). No tipo 1, o *hipervisor* é instalado ou implementado direto no hardware, sem a necessidade de interações com o sistema operacional para isso. Já o tipo 2, o *hipervisor* é executado na forma de um processo, gerenciado pelo *kernel* do sistema. Enquanto que o tipo 2 precisa ser executado para cada máquina virtual em execução, o tipo 1 permite gerenciar várias máquinas ao mesmo tempo como visto na figura 2.

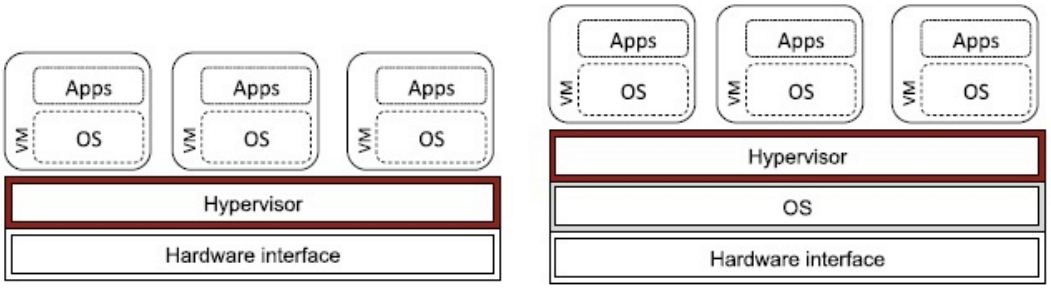

Figura 2 – Comparação entre os tipos de *hipervisores*, da esquerda é o tipo 1, da direita do tipo 2

Fonte: (BASHIR, 2015)

2.1.2 Requerimentos para Virtualização de Popek Go

Popek e Goldberg (1974) teorizam um *hipervisor* pode ser utilizado como camada de abstração numa arquitetura de computadores. Eles classificaram as instruções de máquina de duas formas: *sensíveis* e *privilegiadas*. Assumindo que o processador possa ter operações com permissões, como modo de usuário e modo privilegiado, uma instrução é considerada *privilegiada* se for executada em modo usuário e gerar uma exceção, ou seja, há uma interrupção e o sistema operacional fica sabendo disto. Já as *sensíveis* são aquelas que alteram informações do processador, ou seja, podem manipular os registradores que possuem o estado atual de execução.

Além disso, Popek e Goldberg (1974) descrevem como um *hipervisor* pode ser implementado. Para isso acontecer, é necessário que nesta arquitetura as instruções *sensíveis* sejam um subconjunto das instruções *privilegiadas*. Isso permite que uma instrução não privilegiada cause uma interrupção e seja interceptada pelo *hipervisor*. Com essa interferência, o *hipervisor* consegue emular o efeito da instrução sensível.

2.1.3 Virtualização Total

Virtualização total ou completa, o *hipervisor* emula o hardware físico, logo toda a interface, instruções, dispositivos, operações são emulações. Pelo fato do *hipervisor* ficar no mesmo nível que o sistema operacional, essa virtualização é também chamada de virtualização por hardware(LI; LI; JIANG, 2010). Pelo fato do *hipervisor* interceptar as instruções da máquina virtual, ele consegue traduzi-las para o hardware real, dado que a aplicação na máquina virtual acessam um hardware virtualizado, as instruções para esse não representam as instruções da máquina física.

Uma vantagens dessa técnicas de virtualização é o que hipervisor não interfere no sistema operacional. Porém, a necessidade de captar toda instrução da máquina virtual e traduzi-la reduz o desempenho do do sistema operacional virtualizado e sobrecarrega o hardware físico.

2.1.4 Virtualização a nível de Sistema Operacional

Virtualização a nível de sistema operacional, também chamada de virtualização de processos, é uma técnica aonde usuário tem a ilusão de que o processo executa um sistema operacional completo. Entretanto, todas esses processos vão compartilhar o mesmo *kernel* e bibliotecas, diferente de soluções baseadas em *hipervisores*. Esses processos também pode ser chamados de containers, quando se trata dessa tecnologia, as preocupações são como orquestrar e gerenciar containers em arquiteturas em nuvem(PAHL, 2015).

O conceito de *container* não é recentes. Tecnologias que fornecem isolamentos de processos já existem, como por exemplo jails[1] no FreeBSD e Zones no Solaris[2]. Das vantagens de *containers*, podemos citar:

1. Menos recursos computacionais: ocupa e exige menos recursos (como memória).
2. Densidade: é possível instalar mais *containers* num *host* do que máquinas virtuais.
3. Elasticidade: tempo de execução e término são menores.

A execução e isolamento destes containers acontece através de funcionalidades disponíveis no *kernel*, ou seja, nesta virtualização é feito um acesso direto ao hardware. Tecnologias como Docker e LXC [3], são as mais populares e utilizam funcionalidades do Linux como chroot, *namespaces* e *cgroups* para alcançar o isolamento necessário.

2.2 Chamadas de Sistema

Chamadas de sistemas ou *syscall* são chamadas do qual o processo ou programa faz ao *kernel* do sistema operacional. Essas chamadas podem ser desde operações relacionadas aos dispositivos instalados, como acesso ao disco, ou criação de novos processos. A seguir, vemos as chamadas utilizadas por trás das implementações de *containers* mais populares(SILVA; KIRIKOVA; ALKSNIS, 2018).

2.2.1 Chroot e pivot root

Chroot ou *change root* é um mecanismo que permite mudar o diretório raiz de um processo e todos seus processos filhos. *Chroot* é utilizado para restringir o acesso ao sistema de arquivo, entretanto, ele não é suficiente para que haja um isolamento seguro destes processos no sistema. Um processo que tenha permissões de usuário *raiz* pode executar o *chroot* para sair

[1] <https://www.freebsd.org/cgi/man.cgi?query=jail&sektion=&n=1>
[2] <https://docs.oracle.com/cd/E38904_01/html/820-2978/zones.intro-1.html>
[3] <https://linuxcontainers.org/pt_br/>

desse diretório restrito. Dado essa deficiência, outra chamada do sistema é utilizada para mudar o diretório raiz do processo, *pivot root*[4].

2.2.2 Namespaces

No Linux é implementado vários tipos de *namespaces*[5]. Sendo que o propósito para cada *namespace* é encapsular aquele recurso de forma que apenas os processos que estejam no mesmo *namespaces* possam acessá-lo. Por exemplo, *namespaces* de montagem (tradução de *mount namespaces*) lidam com os sistemas de arquivos. Outros *namespaces* relevantes são o de Process Identifier (PID), Inter-process Communication (IPC), UNIX Time Sharing (UTS). A criação de novos *namespaces* acontece através de três chamadas de sistema:

- **clone**: que cria um novo processo e um novo *namespace*. Além disso, *fork* e *exit* também lidam durante sua execução.
- **unshare**: não cria um processo novo, apenas um novo *namespace* e anexa um processo existente a este.
- **setns**: chamada que anexa um *thread* a um *namespace* existente.

2.2.3 Cgroups

Cgroups ou grupos de controle (traduzido de *control groups*) é um subsistema do kernel que fornece uma hierarquia entre os processos. Com tal, um controlador fica responsável por distribuir os recursos do sistema para cada processo, dada as regras e limitações. Isso acontece através de ganchos nos métodos de criação de processos já citados, *fork e exit*.

[4] <https://linux.die.net/man/8/pivot_root>
[5] <http://man7.org/linux/man-pages/man7/namespaces.7.html>

Capítulo 3

DESENVOLVIMENTO

3.1 Linguagem de Programação Go

Go é uma linguagem de programação de código aberto, estaticamente tipada e compilável criada por Rob Pike e Ken Thompson do Google[1]. Ela possui coletor de lixo, tipagem estrutural e concorrência no estilo de Communicating Sequential Processes (CSP). Influenciada por C, com foco em simplicidade e segurança. Um exemplo de *Hello World* em Go pode ser vista no código abaixo.

Código-fonte 1: Hello World em Go

```
1 package main
2
3 import "fmt"
4
5 func main() {
6   fmt.Println("Hello, World")
7 }
```

3.2 Implementação de um container usando Go

Dado as técnicas e chamadas do sistema detalhadas no capítulo 2, vamos implementar um *container* usando a linguagem Go e executar a distribuição Linux *Alpine*[2] dentro desse. Essa distribuição é uma das mais utilizadas por *containers* pois seu sistema base tem em torno de 4-5 MB sem contar o kernel. Como no caso desse tipo de virtualização, não precisamos de um kernel dado que usaremos o do sistema hóspede.

3.2.1 Detalhes de Implementação

É necessário importar algumas bibliotecas na implementação. Entre as mais importantes estão ao *os/exec* que executa comandos nativos do sistema operacional e *syscall* responsável por lidar com chamadas do sistema. No código abaixo, definimos algumas informações como

[1] <https://golang.org/>
[2] <https://alpinelinux.org/>

diretório do *chroot* e nome do *cgropus*. Além disso, a rotina *main* fica responsável por lidar com os parametros de entrada do programa e chamar as respectivas funções.

Código-fonte 2: Rotina main

```
1 // +build linux
2 package main
3
4 // bibliotecas utilizadas
5 // similar ao #include <...> de C
6 import (
7   "os/exec"     // executa comandos (comandos bash por ex.)
8   "strconv"   // conversão de string
9   "syscall"   // lida com chamadas do sistema
10  "fmt"     // operações de output e input
11  "io/ioutil"   // utilizatrios de output/input (em disco por ex)
12  "log"     // logging
13  "net/http"    // mini servidor web
14  "os"      // utilitarios do sistema operacional
15 )
16
17 // constantes utilizadas durante o programa
18 const (
19  CHROOT_PATH = "/tmp/con/rootfs" // utilizamos o diretório /tmp como
      arquivo de sistema para nossa imagem
20  CONTROLG_NAME     = "con"
21  CONTROLG_ROOT     = "/sys/fs/cgroup"
22  ALPINE_URL  = "http://dl-cdn.alpinelinux.org/alpine/v3.6/releases/
     x86_64/alpine-minirootfs-3.6.2-x86_64.tar.gz"
23 )
24
25 // funçaõ main, serve de função de entrada no compilador
26 func main() {
27
28  // os.Args possui os argumentos que o programa atual recebeu em
     linha de comando
29  switch os.Args[1] {
30  case "run":   // executa um comando dentro do container
31    runContainerSelf(os.Args[2:])
32  case "rmi":   // remove a imagem / diretório do sistema
33    err := removeImageDir()
34    if err != nil {
35      log.Fatal(err)
36    }
37  case "-":   // spawn num processo filho
38    runContainerChild(os.Args[2:])
39  case "pull":    // faz download da imagem localizada no ALPINE
```

```
40      err := pullImage()
41      // tratamento de erro
42      if err != nil {
43        log.Fatal(" error in pulling the Alpine image ", err)
44      }
45      log.Println("image is pulled and in process of installation")
46    }
47 }
```

A função *runContainerSelf* fica responsável pelo *clone* do processo com os novos *namespaces*:

Código-fonte 3: Rotina *runContainerSelf*

```
1
2 // o processo do program atual fica responsável por redirecionar
      algumas coisas pro processo novo
3 // que é criado via clone com  um novo namespace para pid e uts
4 func runContainerSelf(args []string) {
5
6   args = append([]string{"-"}, args...)
7   cmd := exec.Command("/proc/self/exe", args...)
8   cmd.Stdin = os.Stdin
9   cmd.Stdout = os.Stdout
10  cmd.Stderr = os.Stderr
11  cmd.SysProcAttr = &syscall.SysProcAttr{
12    Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall
      .CLONE_NEWNS,
13    // /proc/self/exe will not be found
14    // if we chroot here
15    //Chroot: CHROOT_PATH,
16  }
17  if err := cmd.Run(); err != nil {
18    log.Fatal(err)
19  }
20 }
```

A função abaixo é responsável por criar o novo diretório raiz (via *chroot*) e criar configurações do *cgroups*. É bom salientar que em ambientes Linux tudo é considerado um arquivo, inclusive recursos, sendo assim a configuração dessas duas funcionalidades acontecem através de criação de arquivos em diretórios específicos.

Código-fonte 4: Rotina runContainerChild

```
1 // executa o container num processo filho
2 func runContainerChild(args []string) {
```

```
3   // configura o cgroups do processo
4   if err := setCGroups(&args); err != nil {
5     log.Fatal("error in creating cgroups: ", err)
6   }
7   // cria o diretório raiz onde a imagem vai ser descompactada via
      chroot
8   err := os.MkdirAll(CHROOT_PATH, 0755)
9   if err != nil {
10    log.Fatal("error in creating chroot path: ", err)
11  }
12  // chamamos a rotina chroot para mudar o diretório / para o
      CHROOT_PATH
13  if err = syscall.Chroot(CHROOT_PATH); err != nil {
14    log.Fatal("chroot: ", err)
15  }
16  // precisamos mudar para esse diretório raiz novo,
17  // caso naõ mudemos, procesos ainda terá referencia ao antigo diret
      ório raiz
18  if err := os.Chdir("/"); err != nil {
19    log.Fatal("error in change dir: ", err)
20  }
21  if err := syscall.Mount("proc", "/proc", "proc", 0, ""); err != nil
      {
22    log.Fatal("mounting: ", err)
23  }
24  if err := syscall.Sethostname([]byte("my-container")); err != nil {
25    log.Fatal("setting hostname: ", err)
26  }
27  cmd := exec.Command(args[0], args[1:]...)
28  cmd.Stdin = os.Stdin
29  cmd.Stdout = os.Stdout
30  cmd.Stderr = os.Stderr
31  if err := cmd.Run(); err != nil {
32    log.Fatal(err)
33  }
34  if err := syscall.Unmount("/proc", 0); err != nil {
35    log.Fatal("umount: ", err)
36  }
37 }
```

Nas rotinas *setCGroups*, configuramos o grupo de controle responsável por gerenciar e limitar o recursos do processo. Para o *container* criado dois novos *cgroups* são feitos: *PID* e o de memória.

Código-fonte 5: Rotinas setCGroups e createCGroup

```
1
```

```
2 // configura o conjunto do control groups
3 func setCGroups(args *[]string) error {
4   // avoid PID exhaustion
5   // kernel/Documentation/cgroups/pids.txt
6   if err := createCGroup("pids", "10"); err != nil {
7     return err
8   }
9   if (*args)[0] == "-m" {
10    if err := createCGroup("memory", (*args)[1]); err != nil {
11      return err
12    }
13    *args = (*args)[2:]
14  }
15  return nil
16 }
17
18 //configuramos o control groups do processo
19 // definimodos pid, memoria disponivel (e seus limites)
20 func createCGroup(resource, limit string) error {
21  // dicionário com o que queremos controlar do processo
22  resourceDict := map[string]string{
23    "pids":   "/pids.max",
24    "memory": "/memory.limit_in_bytes",
25  }
26  // construimos um caminho do sistema
27  // como quase tudo no linux é um arquivo, configurações de control
     groups também são
28  resourceDir := fmt.Sprintf("%s/%s/%s", CONTROLG_ROOT, resource,
     CONTROLG_NAME)
29  err := os.MkdirAll(resourceDir, 0755) // criamos o diretório e
     arquivos com as configurações
30  if err != nil {
31    return err
32  }
33  // escrevemos de fato o que queremos controlar no processo dentro
     desse diretório do cgroups
34  err = ioutil.WriteFile(resourceDir+resourceDict[resource], []byte(
     limit), 0644)
35  if err != nil {
36    return err
37  }
38  // se o recurso solicitado é memória
39  // configuramos memória swap
40  if resource == "memory" {
41    err = ioutil.WriteFile(resourceDir+"/memory.swappiness", []byte
     ("0"), 0644)
42    if err != nil {
```

```
43        return err
44      }
45    }
46    // pegamos o pid do processo atual, convertemos para inteiro e
       depois para string
47    p := strconv.Itoa(os.Getpid())
48    // anexamos esse PID ao diretório /tasks, onde o control groups
       trabalha
49    // o pid tambem é gerenciado como um arquivo no sistema,
50    err = ioutil.WriteFile(resourceDir+"/tasks", []byte(p), 0644)
51    if err != nil {
52      return err
53    }
54    return nil
55 }
```

A função abaixo fica responsável por fazer o *downloads* dos arquivos da distribuição *Alpine*

Código-fonte 6: Rotina pullImage

```
1 func pullImage() error {
2   // pega o conteudo da ALPINE_URL (que é um tar)
3   res, err := http.Get(ALPINE_URL)
4   if err != nil {
5     return err
6   }
7   defer res.Body.Close() // fecha a conexão http no final deste
     escopo
8   // cria um diretório com CHROOT_PATH dado e permissões
9   // 0755(lida e executa por outros, porém apenas o dono pode
10  // escrever nela)
11  err = os.MkdirAll(CHROOT_PATH, 0755)
12  if err != nil {
13    return err
14  }
15  // como a imagem vem um tar, precisamos descompacta-la
16  // fazemos via comando do sistema
17  cmd := exec.Command("tar", "-xzf", "-", "-C", CHROOT_PATH)
18  cmd.Stdin = res.Body
19  cmd.Stderr = os.Stderr
20  err = cmd.Run()
21  if err != nil {
22    return err
23  }
24  return nil
25 }
26
```

```
27 func removeImageDir() error {
28   return os.RemoveAll(CHROOT_PATH)
29 }
```

3.3 Criando um container

Para criar um container com a implementação acima, precisamos compilar o código através do comando a seguir:

```
1 go build containers.go
```

No diretório onde o código se encontra, podemos executa-lo de tal forma (em alguns kernels, o programa precisa de permissões de usuário *root*):

```
1 # puxa a imagem do Alpine LInux
2 ./containers pull
3
4 # executa o shell dentro do container com limite de memória
5 ./containers run -m 10000000 /bin/sh
6 # depois desse comando, o processo atual se torna o container, tendo
      acesso as funcionalidades
7 # da imagem
8
9 # remove o diretório e imagem
10 ./containers rmi
```

3.4 Resultados

Como visto, em poucas de linhas de código é possível lidar com *containers* programaticamente. Graças as funcionalidades do kernel, *chroot*, *namespaces* e *cgroups*, implementamos um *container* mínimo, limitando sua memória e controlado seu acesso. A partir de tal implementação, é realizável a construção de sistemas maiores utilizando *containers* ou lidando com a orquestração desses.

3.5 Dificuldade e limitações

Uma das maiores dificuldades é definir de fato o escopo do que vai ser tratado no trabalho. Por exemplo, não foi tratado o padrão Open Containers Iniciative (OCI) que padroniza vários elementos de sistemas de *containers*, como sua implementação, interação com o sistema de arquivo e imagens. Isto acontece dado que tal padrão se torna complexo e além do escopo do

trabalho pelo fato de interagir com vários outros sistemas, como orquestração e repositório. Das limitações que a implementação tratada nesse trabalho possui, podemos citar o fato de que a implementação funciona apenas para imagens da distribuição *Alpine*, além de que outros tipos de controles de recursos como redes e interface *IPC* não são restringidas pelo *cgroup*.

Capítulo 4

CONCLUSÃO

4.1 Conclusão

Neste trabalho foi descrito funcionalidades do sistemas operacionais Linux que podem ser utilizadas para implementar um sistema mínimo de *containers*. Através de chamadas de sistemas, em poucas linhas de código, é possível isolar um processo de forma que ele utilize uma imagem mínima e um diretório diferente do sistema de arquivo. Dadas chamadas de função, um sistema de *containers* é implementado utilizando a linguagem Go. Pelo fato dessa linguagem fornecer bibliotecas de interação com o kernel, torna fácil a interação com o sistema operacional.

4.1.1 Considerações sobre o Curso de Graduação

O curso de graduação forneceu ferramentas essenciais para a pessoa que eu sou hoje. Graças as dificuldades que passei e as aulas ruins que tive, construí uma mentalidade de sempre estar aprendendo, de ser curioso e questionar como as coisas funcionam. Não acreditar em axiomas e em conhecimento "magia negra", aquele que todos tem medo de entender pelo preconceito de considerar difícil. Com isso, adquiri uma base sólida de conhecimento e fundamentais que me ajudaram no desenvolvimento pessoal e profissional.

As disciplinas de matemática oferecidas, poucas foram no contexto computacional. A estrutura de disciplinas fornece uma sólida base matemática, entretanto, estão acumuladas nos dois primeiros anos e as disciplinas de computação nos anos seguintes, fazem pouco uso dessa base. O que tange às disciplinas de computação, é visível o esforço em mantê-las atualizadas. Porém acredito que disciplinas fundamentais, como Programação Orientada aos Objetos e Sistemas Operacionais, precisam ser reestruturadas e atualizadas.

A infraestrutura e suporte dado aos alunos destaco como um das maiores qualidades. Seja a infraestrutura física como laboratórios ou seções de serviços, como Seção de Eventos. Tal ambiente privilegia o conhecimento e o crescimento de vários grupos de extensões. Posso afirmar que sem grupos de extensão, eu não teria continuado no curso.

Complementando com minha autocrítica, demorei demais para tomar atitude em relação aos problemas que a graduação me trouxe. O fato de ter vindo com uma base de conhecimento deficitária fez com que meus primeiros anos fossem difíceis, dado a alta carga de disciplinas de matemática. Dificuldades que carrego até hoje.

REFERÊNCIAS

BASHIR, S. **Handling Elephant Flows in a Multi-tenant Data Center Network**. Tese (Doutorado), 09 2015. Citado na página 26.

HUNG, L.-H.; KRISTIYANTO, D.; LEE, S.; YEUNG, K. Y. Guidock: Using docker containers with a common graphics user interface to address the reproducibility of research. **PloS one**, v. 11, p. e0152686, 04 2016. Citado na página 22.

LAADAN, O.; NIEH, J. Operating system virtualization: Practice and experience. In: . [S.l.: s.n.], 2010. Citado na página 22.

LI, Y.; LI, W.; JIANG, C. A survey of virtual machine system: Current technology and future trends. **Electronic Commerce and Security, International Symposium**, v. 0, p. 332–336, 07 2010. Citado na página 26.

LWAKATARE, L. E.; KUVAJA, P.; OIVO, M. Relationship of devops to agile, lean and continuous deployment. In: . [S.l.: s.n.], 2016. p. 399–415. ISBN 978-3-319-49093-9. Citado na página 21.

OLUDELE, A.; OGU, E.; KUYORO, S.; UMEZURUIKE, C. On the evolution of virtualization and cloud computing: A review. **Journal of Computer Science and Applications**, v. 2, p. 40–43, 12 2014. Citado na página 25.

PAHL, C. Containerisation and the paas cloud. **IEEE Cloud Computing**, v. 2, p. 24–31, 06 2015. Citado 2 vezes nas páginas 21 e 27.

POPEK, G.; GOLDBERG, R. Formal requirements for virtualizable third generation architectures. **Commun. ACM**, v. 17, p. 412–421, 01 1974. Citado na página 26.

SILVA, V.; KIRIKOVA, M.; ALKSNIS, G. Containers for virtualization: An overview. **Applied Computer Systems**, v. 23, p. 21–27, 05 2018. Citado na página 27.

STåHL, D.; MåRTENSSON, T.; BOSCH, J. Continuous practices and devops: beyond the buzz, what does it all mean? In: . [S.l.: s.n.], 2017. p. 440–448. Citado na página 21.

ZAHARIA, T. Virtualization as the evolution of operating systems. v. 7 (XXIV), 06 2010. Citado na página 25.