UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

BERNARDO L. GONÇALVES

AVALIAÇÃO DOS RECURSOS DO *KERNEL* LINUX PARA A CRIAÇÃO DE *CONTAINERS*

RIO DE JANEIRO
2019

BERNARDO L. GONÇALVES

AVALIAÇÃO DOS RECURSOS DO *KERNEL* LINUX PARA A CRIAÇÃO DE *CONTAINERS*

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Valeria Menezes Bastos

RIO DE JANEIRO
2019

CIP - Catalogação na Publicação

Gonçalves, Bernardo Lins
G635a Avaliação dos recursos do Kernel Linux para a criação de containers / Bernardo Lins Gonçalves. -- Rio de Janeiro, 2019.
58 f.

Orientadora: Valeria Menezes Bastos.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2019.

1. Container. 2. Linux. 3. Sistemas operacionais. I. Bastos, Valeria Menezes, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

BERNARDO L. GONÇALVES

AVALIAÇÃO DOS RECURSOS DO *KERNEL* LINUX PARA A CRIAÇÃO DE *CONTAINERS*

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 13 de dezembro de 2019

BANCA EXAMINADORA:

Valeria Menezes Bastos
D.Sc. (UFRJ)

Nelson Quilula Vasconcelos
M.Sc. (UFRJ)

Vanessa Quadros Gondim Leite
M.Sc. (IME)

Dedico esta conquista à minha esposa Nathalia, que com suas palavras de apoio e carinho me inspirou durante todo o curso da graduação e também aos meus pais, que não mediram esforços para que eu chegasse nesta etapa da minha vida.

AGRADECIMENTOS

Agradeço ao Departamento de Ciência da Computação pela oportunidade de aplicar na prática os conhecimentos que inspiraram esta pesquisa. Agradeço também à Professora Valeria pelo apoio e orientação durante a escrita do texto bem como à minha esposa por toda a compreensão e suporte. Por fim, meus agradecimentos aos amigos que me acompanharam durante o curso e fizeram parte da minha formação.

"Quando aceitamos nossos limites, conseguimos ir além deles."

Albert Einstein

RESUMO

Este trabalho apresenta um estudo sobre as ferramentas disponíveis no Linux que podem ser usadas para criar e executar processos em *containers*. Para se ter uma visão clara dos diferentes aspectos que definem um *container*, foram usados os padrões definidos pela Open Containers Initiative. A partir da premissa de que a maioria das ferramentas usadas para criação de um *container* estão presentes no próprio *kernel*, foi feito um estudo do modelo de processos do Linux, assim como das ferramentas que permitem modificar o seu comportamento. Com o objetivo de validar o funcionamento dessas ferramentas, foi desenvolvida uma aplicação de linha de comando que cria e executa containers seguindo a especificação da Open Containers Initiative. Diferentes cenários de uso desta aplicação foram executados, com o objetivo de mostrar o impacto das diferentes ferramentas estudadas no funcionamento de um processo. Através desses cenários foi possível verificar como diferenças no uso das ferramentas estudadas modificam o comportamento dos processos em execução no *container*.

Palavras-chave: container. linux. sistemas operacionais.

ABSTRACT

This work presents a study about the resources available on Linux that can be used to create and execute containers. To clarify the definition of a container, the standards defined by the Open Containers Initiative were used. Starting from the premise that most of the tools are available on the kernel itself, a study was made about the Linux process model and the tools that can be used to modify its behavior. To validate how those tools work, a command-line application that creates and executes containers were developed, following the Open Containers Initiative spec. Several scenarios were executed with this application to show how different uses of the tools change how the process works. Through them, it was possible to verify how those different tools modify the behavior of the process running inside the container.

Keywords: containers. linux. operating systems.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programing Interface
CLI	Command Line Interface
CPU	Central Processing Unit
PID	Process Identifier
UID	User Identifier
GID	Group Identifier
ELF	Executable and Linkable Format
CFS	Completely Fair Scheduler
PC	Program Counter
JSON	JavaScript Object Notation
OCI	Open Containers Initiative
LCI	Laboratório de Computação de Informática

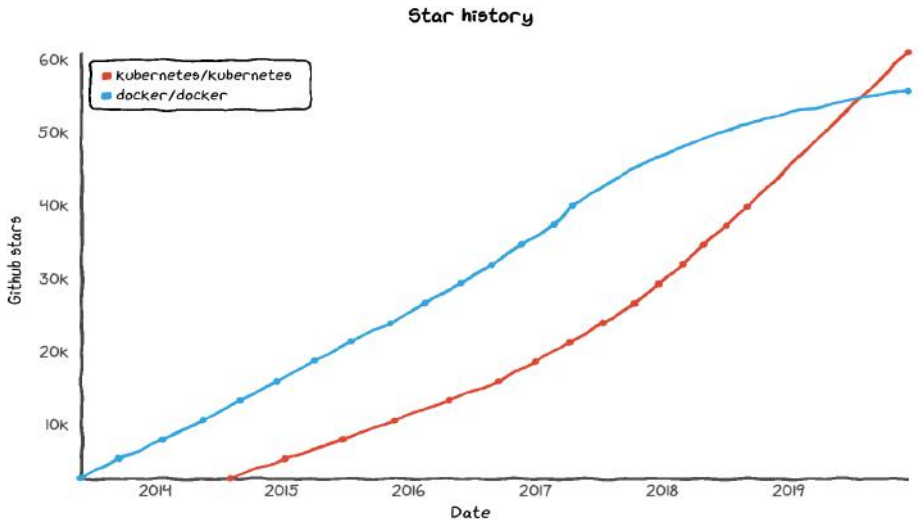
SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

O uso de *containers* ganhou popularidade recentemente, com muitas empresas adotando este tipo de tecnologia para distribuição e lançamento de aplicações. O desenvolvimento de aplicações *cloud native*, com foco em resiliência e escalabilidade cresceu com o surgimento dos chamados orquestradores de *container*. O Kubernetes[1] é um dos exemplos mais famosos de orquestrador. Na figura 1 é possível verificar a variação da quantidade de estrelas recebidas por esse projeto no Github[2], o que ilustra esse crescimento de popularidade.

Figura 1 – Variação da popularidade do Kubernetes e Docker



Fonte: https://star-history.t9t.io/

Ferramentas como Kubernetes são chamadas de orquestradores de *container*, pois permitem instanciar um conjunto de *containers* para executar uma aplicação – que pode estar distribuída em diferentes máquinas de um cluster – movê-los entre máquinas, reiniciá-los caso sofram algum tipo de erro, aumentar ou diminuir sua quantidade de acordo com a demanda e assim por diante. Para realizar estas tarefas, o orquestrador deve se comunicar com um *container runtime*, ou seja, um programa que, entre outras coisas, cria e destrói os *containers*. O Docker[3] é um *container runtime* com grande popularidade no momento da escrita desse texto. A figura 1 também mostra a variação da sua quantidade de estrelas

[1] <https://kubernetes.io>
[2] <https://github.com/kubernetes/kubernetes>
[3] <https://www.docker.com>

no Github[4] ao longo dos anos.

Um *container runtime* deve ser capaz de efetuar uma série de operações, lendo seus parâmetros de entrada e traduzindo para chamadas de sistema feitas ao sistema operacional. Os *containers* não são objetos internos do *container runtime*, mas sim processos do sistema operacional. Sendo assim, quando um *runtime* – como o Docker – recebe o comando para a criação de um novo *container*, internamente, uma chamada de sistema é feita para que se crie um novo processo. Este novo processo se difere dos demais apenas por uma camada de isolamento que modifica a forma que os recursos do sistema se mostram disponíveis para ele. De fato, é possível visualizar normalmente um processo em execução dentro de um *container*, juntamente com os outros processos do sistema, através de qualquer ferramenta que liste os processos em execução no sistema operacional, como o ps[5]. As operações e o procedimento de criação de containers são descritos na especificação da OCI[6], mostrada com mais detalhes no capítulo 3.

(RANDAL, 2019) diz que o *kernel* do Linux possui vários componentes que permitem a criação dos *containers*, como *namespaces* e *control groups*. Esses componentes estão disponíveis, em alguns casos, há muitos anos, mas o surgimento dos projetos em *containers* facilitou sua adoção por parte dos usuários. Seu uso permite uma utilização melhor de recursos quando comparados aos mecanismos de virtualização tradicionais (como o KVM), uma vez que fazem uso dos componentes do próprio sistema hospedeiro. Entretanto, garantir a segurança dos *containers* é uma tarefa mais desafiadora, uma vez que a camada de isolamento entre eles e o sistema operacional é menor. Configurações incorretas na criação do *container* ou vulnerabilidades no sistema podem comprometer o isolamento, tornando possível o acesso aos recursos que não deveriam estar disponíveis (HAYDEN, 2015). Existem maneiras de aumentar a segurança com módulos como SELinux[7] e AppArmor[8]. O uso desses módulos e os problemas que podem comprometer a segurança dos *containers* não são o foco deste trabalho.

O estudo sobre a tecnologia de *containers* foi motivado devido ao trabalho de melhoria na disponibilidade dos serviços do LCI, no Departamento de Ciência da Computação da UFRJ. Este trabalho consistiu em modificar a arquitetura dos diferentes serviços oferecidos aos alunos e professores do departamento – como a página *web*, *e-mail* e armazenamento de arquivos pessoais.

Inicialmente, os serviços eram distribuídos em um grupo de máquinas, sendo cada uma responsável por um ou mais serviços. Essa arquitetura dificultava manutenções programadas e causava grandes períodos de indisponibilidade de um grupo de serviços em caso de falhas no software ou hardware de algum servidor. Com a ajuda de um

[4] <https://github.com/docker/docker>
[5] <http://man7.org/linux/man-pages/man1/ps.1.html>
[6] <https://www.opencontainers.org/>
[7] <https://selinuxproject.org/page/Main_Page>
[8] <https://www.kernel.org/doc/html/latest/admin-guide/LSM/apparmor.html>

container runtime (Docker) e um orquestrador de *containers* (Docker Swarm Mode[9]), foram criadas imagens para a maior parte dos serviços, que passaram a ser executados como *containers*. O orquestrador então garante que o serviço permanecerá disponível enquanto houver máquinas suficientes. Maiores detalhes deste projeto de migração estão além do escopo deste estudo, mas o caso ilustra bem uma aplicação prática do uso de *containers*.

O objetivo deste trabalho é mostrar as ferramentas do Linux que podem ser usadas para criar e executar processos em *containers*. O escopo inclui ainda o desenvolvimento de uma aplicação que faz uso destes recursos para criar e executar *containers*. O desenvolvimento do trabalho teve início com um estudo do modelo de processo no Linux, mostrando como é sua implementação, como podem ser criados e como um desenvolvedor pode interagir com eles. Em seguida foram abordados os parâmetros e ferramentas que limitam e controlam a execução dos processos no Linux. Por fim, foi feita a implementação de uma aplicação de linha de comando, com o intuito de demonstrar o uso desses parâmetros e ferramentas. O estudo partiu da hipótese de que as principais ferramentas usadas para criar um container estão presentes no próprio kernel do Linux.

1.2 ESTRUTURA DO DOCUMENTO

O capítulo 2 faz referência aos recursos disponíveis no sistema operacional, que são usados para a criação de um *container*.

A seção 2.1 se refere ao modelo de processos no Linux. Ela começa descrevendo a diferença entre um programa e um processo, apresenta as estruturas de dados implementadas no código do *kernel*, relacionadas a eles, e os seus principais campos, e explica como é o conceito de *threads* no Linux. Por fim, é tratado o sistema de arquivos `/proc` e os arquivos e diretórios mais importantes para o escopo deste trabalho.

A seção 2.2 trata da execução de um programa e como é feita a criação de um novo processo. Também é explicado como funciona a hierarquia entre processos e a relação entre processo pai e filho. A seção apresenta detalhes sobre as chamadas *fork* e *clone*. Também é mostrado como modificar a imagem de um processo através das chamadas de sistema da família "exec". Por fim, são descritas as chamadas de sistema *wait* e *exit*.

A seção 2.3 descreve os mecanismos internos do *kernel* para o isolamento de processos. Explica o conceito de usuários e grupos e como funcionam as "capabilities"do Linux que limitam o uso de algumas operações por parte dos processos.

A seção 2.4 fala sobre o diretório raiz do processo e como é possível modificá-lo com a chamada de sistema "chroot". Na seção 2.5 é dada a definição de namespaces e as chamadas de sistema relacionadas, como unshare e "setns". Por fim, são descritos todos

[9] <https://docs.docker.com/engine/swarm/>

os tipos de *namespace* existentes no *kernel* até a versão 4.20.10, última versão estável até o lançamento deste trabalho.

A seção 2.7 trata da limitação de recursos disponíveis aos processos no Linux e mostra como é possível modificar esse limites para um determinado recurso (como CPU ou memória) com os chamados *control groups*.

O capítulo 3 descreve o projeto de código que acompanha o trabalho e como alguns dos recursos descritos no capítulo 2 são usados em uma CLI, capaz de criar e gerenciar o ciclo de vida de um *container*.

O capítulo 4 apresenta as conclusões deste trabalho e desenvolvimentos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Para (RANDAL, 2019), os *container* modernos não possuem uma característica principal, mas combinam diferentes tecnologias capazes de prover controle de recursos e isolamento. Muitas dessas tecnologias foram criadas e introduzidas ao Linux ao longo dos anos, antes mesmo do termo *container* ser usado com o propósito atual. Este capítulo descreve as principais tecnologias presentes no Linux que permitem a criação e execução de um *container*.

2.1 MODELO DE UM PROCESSO

Processos são responsáveis por executar tarefas no sistema operacional. Enquanto um programa é um arquivo executável armazenado no disco, um processo é a instância de um programa em execução. O programa que um processo executa é chamado de imagem do processo (ARPACI-DUSSEAU, 2015; TANENBAUM, 2014).

2.1.1 Programas e Processos

O programa é um arquivo estático, armazenado em disco, que contém uma série de informações sobre como construir o processo no momento de sua execução. Todo programa inclui metainformações sobre o formato do arquivo executável, indicando ao *kernel* como ele deve interpretar o restante do arquivo. Atualmente, o Linux faz uso do formato ELF. Os programas também contém as instruções em código de máquina necessárias para a sua execução, além de dados estáticos e valores de inicialização de variáveis globais, informações sobre a tabela de símbolos (que armazena o nome e localização das funções e variáveis no programa), a lista de todas as bibliotecas compartilhadas que o programa faz uso e muitas outras informações (KERRISK, 2010; LOVE, 2010).

O processo, por sua vez, é dinâmico e suas informações mudam conforme a execução das instruções de máquina do programa. Além das instruções do programa, um processo contém o contador de programa (*Program Counter* ou PC) e valores armazenados em outros registradores. O processo também possui um espaço de endereçamento, onde ficam armazenadas as instruções do programa, variáveis temporárias, argumentos de linha de comando e retornos de função. Um processo é a instância de um programa em execução, sendo que o mesmo pode ser executado por diferentes processos simultaneamente (ARPACI-DUSSEAU, 2015; KERRISK, 2010; TANENBAUM, 2014). Cada processo é uma tarefa separada no sistema operacional e, mesmo que dois processos estejam executando o mesmo programa, uma falha durante a execução de um deles não afetará a execução do outro. Cada processo executa em seu próprio espaço de endereçamento vir-

tual e não consegue se comunicar com outros processos, a não ser por mecanismos seguros disponibilizados pelo *kernel* (RUSLING, 1999).

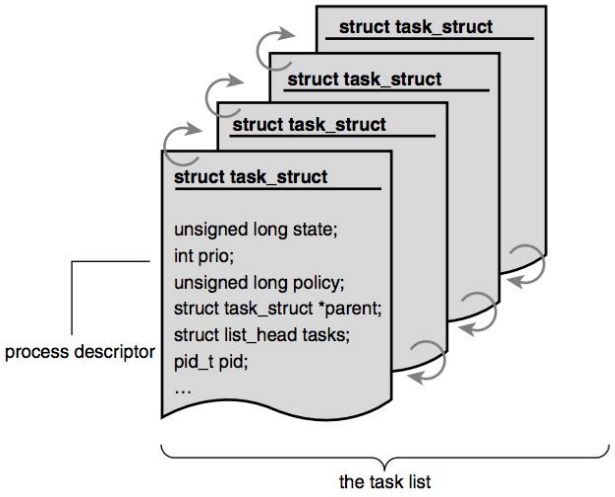
Durante seu ciclo de vida, um processo consome e manipula diferentes recursos do sistema, como a CPU, memória física e arquivos do sistema de arquivos. O recurso mais importante para um processo é a CPU. O Linux executa um processo em cada CPU disponível e, caso existam mais processos que CPUs, o restante dos processos deve aguardar seu momento de execução. Quando uma CPU estiver disponível, é dever do escalonador decidir qual processo deve ser executado em seguida. O Linux adota várias estratégias para fazer o compartilhamento de CPU o mais justo possível. De fato, o escalonador padrão do Linux, inserido a partir da versão 2.6.23, é chamado de *Completely Fair Scheduler* (CFS) (ISHKOV, 2015).

2.1.2 Representação Interna dos Processos

No Linux, cada processo possui um descritor, representado por uma estrutura de dados interna do *kernel* chamada `task_struct` (LOVE, 2010; RUSLING, 1999). O descritor contém todas as informações de um determinado processo que o *kernel* necessita. Os descritores de processo são armazenados em uma lista circular duplamente encadeada chamada *task list* ou *task array* (LOVE, 2010). A figura 2 ilustra essa lista.

Apesar de ser uma estrutura de dados grande e complexa, (LOVE, 2010) explica que é possível dividir os campos da task_struct em diferentes áreas:

Figura 2 – Representação da task list



Fonte: Love (2010, p. 25)

- **Estado**: Enquanto o processo é executado, seu estado é atualizado de acordo com as circunstâncias. Na estrutura `task_struct` o estado atual do processo é armazenado

no campo *state*. O código do kernel representa cada estado por macros com os nomes se iniciando em `TASK_*`. A figura 3 mostra um diagrama com as possíveis mudanças de estado de um processo. Durante sua execução, todo processo no Linux está em um dos seguintes estados:

  - **Running**: O processo está em execução ou está pronto para ser executado pela CPU.
  - **Waiting**: O processo está esperando um sinal ou um recurso ficar disponível. Processos no estado *waiting* podem ser diferenciados entre *interruptible* e *uninterruptible*.
  - **Stopped**: O processo terminou. Um processo sendo analisado por alguma ferramenta de *debug* pode estar neste estado.
  - **Zombie**: O processo morreu mas, por algum motivo, ainda existe um descritor alocado para ele na *task list*. Mais informações sobre processos zumbis podem ser vistas em 2.2.5.

- **Informações de escalonamento**: Estas informações são usadas pelo escalonador de processos para decidir qual processo deve ser executado.

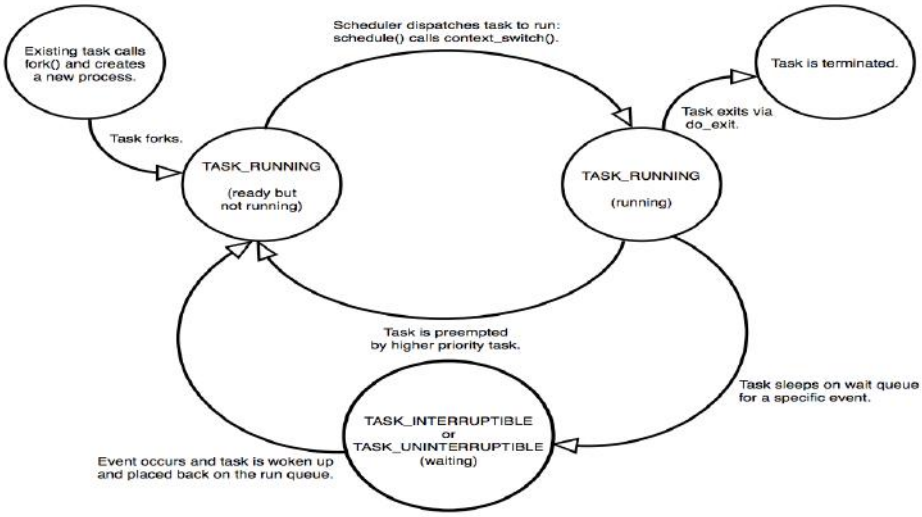
- **Identificadores**: Os identificadores são campos que identificam o processo (PID), seu usuário (UID) e grupo (GID). Estes últimos são usados para controlar permissões de acesso do processo a arquivos e diretórios do sistema. Detalhes sobre o PID do processo se encontram na seção 2.2.1, e as informações sobre o UID e GID são apresentadas em 2.3.

- **Sistema de Arquivos**: O descritor do processo contém ponteiros para os descritores de todos os arquivos abertos pelo processo, além de informações sobre o diretório corrente e o diretório raiz do processo.

- **Memória virtual**: Muitos processos possuem um espaço de endereçamento virtual, e o descritor do processo guarda informações sobre o mapeamento da sua memória virtual no espaço de endereçamento real na Memória Principal.

- **Outras informações**: O descritor é responsável por armazenar outros tipos de informação sobre o processo, como ponteiros para o processo pai e seus irmãos (na seção 2.2.1 é detalhada a hierarquia de processos), informações sobre os tempos de execução e o tempo consumido desde sua criação, o contexto do processo (como valores nos registradores) e muito mais.

Figura 3 – Diagrama de estados do processo



Fonte: Love (2010, p. 28)

2.1.3 *Threads*

Threads são uma abstração muito popular nos sistemas modernos, uma vez que permitem o desenvolvimento de programas concorrentes. Caso múltiplos processadores estejam disponíveis, o uso de *threads* em um programa permite atingir o paralelismo verdadeiro, onde diferentes partes do programa – cada uma em uma *thread* diferente, são executadas simultaneamente (LOVE, 2010).

O Linux possui uma implementação única do modelo de *threads*. Internamente, não existe uma estrutura de dados que represente uma thread ou alguma política de escalonamento específica para elas. Dentro do kernel, cada thread é simplesmente um processo que compartilha alguns recursos com o processo pai. Cada thread recebe seu próprio descritor, ou seja, existe uma `task_struct` alocada na *task list* para cada thread no sistema (LOVE, 2010). Elas podem ser criadas com a chamada de sistema clone, detalhada na seção 2.2.3.

2.1.4 Organização da Memória de um Processo

O espaço de endereçamento de um processo é dividido em partes, constantemente referidas como segmentos. Em sistemas UNIX, um segmento é uma divisão lógica da memória virtual de um processo.

O segmento de texto, chamado *text segment*, é onde ficam as instruções em código de máquina do programa. Como vários processos podem estar executando o mesmo programa, este segmento é compartilhado. Para evitar que um processo possa modificar suas próprias instruções devido a um acesso indevido a esta área de memória, o segmento de texto somente pode ser acessado para leitura (KERRISK, 2010).

O segmento de dados inicializados, chamado *initialized data segment*, contém variáveis globais e estáticas que foram explicitamente inicializadas no código do programa. Essas valores são lidos do executável e carregados para essa área de memória (KERRISK, 2010).

O segmento de dados não-inicializados, chamado *uninitialized data segment*, contém as variáveis globais e estáticas que não foram inicializadas. Antes da execução do programa o sistema inicializa todas as variáveis neste segmento com o valor 0. A razão para esta área de memória estar separada do segmento de dados inicializados, é que o programa, quando armazenado em disco, não precisa reservar espaço para essas variáveis. Em vez disso, ele guarda a localização e nome das variáveis e o tamanho necessário para este segmento. Quando o programa for executado, o espaço necessário será alocado na memória (KERRISK, 2010).

A *stack* é onde ficam informações sobre chamadas de função, argumentos passados pela linha de comando e variáveis locais das funções. O tamanho da *stack* pode crescer ou diminuir de acordo com as instruções que estão sendo executadas. A *stack* armazena os chamados *stack frames*. Para cada chamada de função um *stack frame* é alocado neste segmento (ARPACI-DUSSEAU, 2015; KERRISK, 2010).

A *heap* é o segmento reservado para alocação de estruturas de dados de tamanho dinâmico, como listas encadeadas, por exemplo. Assim como a stack, o tamanho da *heap* também pode variar. Na linguagem C, por exemplo, é possível alocar espaço na *heap* usando a função `malloc` e liberar espaço usando a função `free` (ARPACI-DUSSEAU, 2015; KERRISK, 2010).

2.1.5 Diretório com Informações do Processo

Cada processo possui seu próprio subdiretório de `</proc>` localizado em `</proc/PID>`, sendo PID o identificador único do processo no sistema operacional. A seção 2.2.1 possui mais detalhes sobre PIDs e a hierarquia de processos. Os arquivos presentes neste diretório pertencem ao mesmo usuário e grupo deste processo e somente os processos que pertencem a este usuário podem manipular estes arquivos – salvo aqueles que tenham privilégio para tal (mais detalhes em 2.3.1).

A maior parte desses arquivos estão acessíveis somente para leitura, o que significa que eles estão disponíveis para se obter informações sobre o *kernel* e não podem ser modificados. Existe uma grande lista de arquivos e diretórios que podem ser encontrados em `</proc/PID>`. A lista completa e atualizada pode ser consultada no manual sobre `</proc>`[1]. Algumas entradas importantes para o escopo deste trabalho estão listadas a seguir. A documentação completa de cada item também pode ser consultada no manual de `</proc>`.

[1] <http://man7.org/linux/man-pages/man5/proc.5.html>

- `</proc/PID/attr/>`
  Este diretório contém arquivos usados para a configuração de atributos relacionados com segurança. Estes arquivos formam uma API criada para o SELinux, mas que é genérica o suficiente para ser usada por outros módulos de segurança. Para que ele esteja presente, o *kernel* deve estar configurado com o parâmetro `CONFIG_SECURITY`.

- `</proc/PID/cgroup>`
  Este arquivo contém os *control groups* que o processo em questão faz parte. O conteúdo do arquivo difere de acordo com a versão dos cgroups. Para cada hierarquia que o processo faz parte, existe uma entrada no arquivo contendo três campos, separados pelo caractere ':' (dois pontos). Cada entrada é uma *string* no formato `<hierarquia:controllers:path>` (mais detalhes em 2.7).

- `</proc/PID/limits>`
  Este arquivo contém os limites inferiores e superiores e a unidade de medida de cada um dos recursos do processo (mais detalhes em 2.6). Até a versão 2.6.35 do Linux (inclusive), este arquivo só poderia ser lido pelo usuário dono do processo, mas da versão 2.6.36 em diante, qualquer usuário do sistema pode acessar estes arquivos.

- `</proc/PID/ns/>`
  Este diretório contém um arquivo para cada *namespace* que o processo faz parte. Estes *namespaces* podem ser manipulados pela chamada de sistema setns, abordada em maiores detalhes na seção 2.5.3. A partir da versão 3.8 do Linux os arquivos presentes neste diretório passaram a ser exibidos como *links* simbólicos. O conteúdo do *link* simbólico é o tipo do namespace e o número do `inode` deste *link*. Se dois processos estiverem no mesmo namespace, o número do `inode` do *link* simbólico de ambos será igual. Enquanto um processo estiver manipulando um arquivo deste diretório, o *namespace* correspondente continuará existindo até que o descritor de arquivo seja fechado, mesmo que todos os processos dentro do *namespace* terminem. Cada um dos arquivos presentes neste diretório representa um tipo diferente de *namespace* ao qual o processo faz parte.

- `</proc/PID/status>`
  Contém informações a respeito do processo em um formato legível para seres humanos. Este arquivo contém basicamente as mesmas informações dos arquivos `</proc/PID/stat>` e `</proc/pid/statm>`.

- `</proc/PID/task/>`
  Este diretório contém um subdiretório para cada thread do processo. O nome de cada diretório é o número do thread ID (TID). Dentro de cada subdiretório, existem os mesmos arquivos e diretório contidos em `</proc/PID>`. Estes arquivos terão o

mesmo conteúdo quando as threads compartilharem um determinado recurso. Caso o recurso seja exclusivo da thread, o arquivo correspondente pode conter informações diferentes.

2.2 GERENCIAMENTO DE PROCESSOS

Processos são as unidades básicas de alocação de recursos do sistema operacional. Conforme dito em 2.1, um processo executa um programa. É possível que o mesmo programa esteja sendo executado por diferentes processos simultaneamente, mas cada processo terá sua própria cópia do programa em seu espaço de endereçamento e o executará de maneira isolada dos demais (TANENBAUM, 2014).

Os processos são organizados de forma hierárquica dentro do sistema operacional. Todo processo possui um pai, que é o processo que o criou. Quando um processo cria outros, estes são seus filhos. O processo pai também possui um pai e isso se repete até o processo de inicialização do sistema (LOOSEMORE RICHARD M. STALLMAN; DREPPER, 2018).

Esta seção descreve como é a criação, o gerenciamento e a finalização de um processo, através das chamadas de sistema disponíveis no Linux.

2.2.1 Executando programas

Três operações são necessárias para que seja possível executar um programa: a criação de um novo processo, determinar que o novo processo execute o programa e coordenar o término deste programa (LOOSEMORE RICHARD M. STALLMAN; DREPPER, 2018).

Um identificador, chamado de PID, é alocado para cada processo no momento em que ele é criado. Segundo a norma IEEE Std 1003.1-2008 (IEEE, 2017), este valor deve ser um número inteiro positivo, único por processo. O PID de um processo pode ser reutilizado, desde que o seu ciclo de vida tenha terminado.

O ciclo de vida de um processo chega ao fim quando seu término é reportado para seu pai. Neste momento, todos os recursos do processo são liberados, inclusive o número de seu PID, e seu valor poderá ser usado novamente na criação de um novo processo, caso seja necessário (LOOSEMORE RICHARD M. STALLMAN; DREPPER, 2018). O procedimento de encerramento de um processo é descrito em 2.2.5.

A norma IEEE Std 1003.1-2008 determina que os processos sejam criados a partir da chamada de sistema *fork*. No Linux, *fork* simplesmente realiza uma chamada para clone com um conjunto determinado de parâmetros (mais detalhes na seção 2.2.3), e é ela quem efetivamente cria o processo.

Quando criado a partir de *fork*, o novo processo é uma cópia do original, exceto por alguns detalhes, como o fato dele possuir seu próprio PID. Após a criação do novo processo, tanto o processo pai quanto o processo filho continuam a sua execução de forma

independente. Se um programa necessitar esperar o término de um de seus filhos para continuar sua execução, deve realizar explicitamente uma das chamadas da família *wait* (veja a seção 2.2.5 para mais informações).

Um processo recém criado por *fork* executa o mesmo programa de seu pai e continua sua execução do mesmo ponto que ele se encontra, ou seja, no retorno de *fork*. O processo filho pode executar um novo programa usando uma das funções da família *exec*. Conforme dito na seção 2.1, o programa em execução é chamado de imagem do processo. Quando a imagem do processo é modificada, toda informação que ele possui sobre a imagem anterior é perdida. Quando o novo programa terminar, o processo é encerrado e não retorna à imagem antiga.

As seções a seguir descrevem cada etapa do processo de execução de um programa.

2.2.2 Criação do processo

A função para a criação de novos processos segundo a norma IEEE Std 1003.1-2017 (IEEE, 2017) é chamada de fork. Segundo (KERRISK, 2010), quando uma chamada a fork é feita, um novo processo é criado como uma cópia quase idêntica do original. Os dois processos executam o mesmo programa, mas o processo recém criado tem sua própria *heap*, *stack*, registradores e segmento de dados separados do processo original. Inicialmente, os dados contidos na memória do filho são idênticos aos do pai. O filho também tem seu próprio contador de programas e cópia dos registradores.

Pai e filho criados a partir de *fork* se diferem em alguns aspectos específicos, dentre eles:

- O filho tem seu próprio PID;
- O identificador do processo pai é o PID do processo que o criou;
- O tempo de uso de CPU do processo recém criado é zero;
- O filho não herda os sinais pendentes do pai.

O processo filho não começa a execução a partir do início da *main*, mas do mesmo ponto logo após o retorno de *fork* (ARPACI-DUSSEAU, 2015; LOOSEMORE RICHARD M. STALLMAN; DREPPER, 2018). A norma IEEE Std 1003.1-2017(IEEE, 2017) determina que é possível distinguir quem é o processo pai e quem é o processo filho através do valor retornado pela *fork*: no processo pai, a chamada retorna o PID do novo processo e no processo filho, *fork* retorna 0. Algumas bibliotecas, como a biblioteca nix, usada no desenvolvimento do projeto descrito em 3, fazem uso desta informação para criar uma abstração alto nível, onde o programador pode identificar qual processo está em execução

de forma mais clara. Segundo o manual da chamada *fork*[2], as condições de erro que podem ocorrer ao executá-la são:

- `EAGAIN`: Ocorre quando não há recursos disponíveis no sistema para a criação de um novo processo ou o usuário atingiu o limite de processos criados;
- `ENOMEM`: Ocorre quando o processo está requisitando mais espaço do que o disponível no sistema.

A partir do momento que os dois processos passam a existir, a ordem de execução não pode ser determinada, ficando essa decisão a cargo do escalonador de processos do sistema operacional. Um programa que fizer uso de *fork* deve realizar os tratamentos necessários para evitar condições de corrida entre os dois processos. A seção 2.2.5 explica como a chamada de sistema *wait* pode ser útil nesta tarefa.

2.2.2.1 Descritores de arquivos

Segundo (KERRISK, 2010), descritores de arquivos contém informações sobre arquivos usados pelo processo, como o deslocamento atual (*offset*) e informações a respeito do estado do arquivo. Quando criado, o processo filho recebe cópias de todos os descritores de arquivos do pai. As cópias são feitas de maneira que os descritores do pai e do filho referenciam os mesmos arquivos.

(KERRISK, 2010) explica que o compartilhamento dos descritores garante que modificações feitas nos descritores de arquivos (como a atualização do *offset* em um deles, por exemplo) estarão visíveis tanto no pai quanto no filho. Caso dois processos estejam escrevendo no mesmo arquivo, o compartilhamento dos descritores evita que um processo sobrescreva o trabalho do outro.

2.2.2.2 Páginas na memória

Para evitar desperdício de recursos, os sistemas operacionais modernos, entre eles o Linux, não fazem a cópia de todas as páginas da memória do processo pai imediatamente, mas implementam uma técnica chamada *copy on write* (BACH, 1986). Após a execução do fork, as entradas na tabela de páginas dos dois processos são marcadas como somente leitura e passam a apontar para a mesma região da memória física. Qualquer tentativa de modificação em uma dessas páginas, seja pelo processo pai ou pelo processo filho, resultará na criação de uma cópia desta página. A tabela de páginas do processo que tentou realizar a modificação é atualizada de maneira que passe a apontar para a cópia recém criada. A partir deste momento, ambos os processos possuem sua versão privada da página na memória e podem realizar qualquer modificação sem que a mesma esteja visível ao outro (KERRISK, 2010).

[2] http://man7.org/linux/man-pages/man2/fork.2.html

2.2.3 Clone

A chamada de sistema *clone*, assim como *fork*, cria um novo processo. Entretanto, diferentemente de *fork*, *clone* permite que partes do contexto de execução (como o espaço de endereçamento de memória) do novo processo sejam compartilhadas com o processo pai. Ao contrário de *fork*, *clone* não está definida na norma IEEE Std 1003.1-2017 (IEEE, 2017), portanto não deve ser usada diretamente em programas que possuem o requisito de serem portáveis.

Uma das aplicações de *clone* é a criação de *threads*. O código 2.1 mostra o esboço de uma chamada a *clone* com os parâmetros necessários para a criação de uma *thread*.(LOVE, 2010). O efeito de cada parâmetro, assim como a lista completa e atualizada de parâmetros, pode ser visto no manual de *clone*[3].

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

Código 2.1 – Criação de uma *thread* com clone

Segundo (LOVE, 2010), a implementação de *fork* no Linux realiza uma chamada a *clone* com um conjunto específico de parâmetros, mostrados no esboço de código 2.2.

```
clone(SIGCHLD, 0);
```

Código 2.2 – Chamada a *clone* equivalente a fork

(LOVE, 2010) enumera os procedimentos que são executados pelo *kernel* quando é feita uma chamada a *clone*:

1. Uma nova `task_struct` é criada para o novo processo. Neste momento o descritor do novo processo é idêntico ao descritor do processo original – aquele que chamou *clone*.

2. É feita uma verificação nos limites de recurso do usuário que está criando o novo processo, para garantir que o mesmo não irá extrapolar o número máximo de processos permitido para ele

3. Para diferenciar o processo pai do processo filho, vários campos do novo descritor são configurados para o valor inicial. Estes campos, em sua maioria, contêm informações estatísticas sobre o processo.

4. O estado do processo é modificado para `TASK_UNINTERRUPTIBLE` para garantir que ele não executará ainda.

[3] <http://man7.org/linux/man-pages/man2/clone.2.html>

5. As *flags* do processo são atualizadas.

6. Um novo PID é alocado para o processo.

7. Dependendo das *flags* passadas pra a chamada *clone*, alguns recursos podem ser compartilhados – como arquivos abertos, sinais pendentes e espaço de endereçamento. Os recursos que não forem compartilhados são copiados neste momento.

8. Por fim, as limpezas necessárias são feitas e a função retorna para quem a chamou.

Vários parâmetros podem ser passados como argumento para a chamada de *clone*. A lista completa e atualizada de parâmetros aceitos na chamada *clone* pode ser consultada em seu manual. Alguns parâmetros importantes para o desenvolvimento deste trabalho são detalhados em 2.5.

Para que um *container* seja criado, o *container runtime* precisa configurar um ambiente onde seu processo de inicialização executará de forma isolada do restante do sistema operacional. A seção 3.3 mostra como um *clone* é usado para se criar um novo *container*.

2.2.4 Modificando a imagem de um processo

Um processo pode modificar o programa em execução através de alguma função da família *exec*. Estas funções são implementadas sobre a chamada de sistema execve (KERRISK, 2010).

A norma IEEE Std 1003.1-2008 (IEEE, 2017) determina que um dos parâmetros das funções *exec* seja um caminho para o programa que será executado. Dependendo da função, este caminho pode ou não ser absoluto. A exceção é a função fexecve, que possui a mesma funcionalidade que execve, mas recebe um descritor de arquivo no lugar do caminho do programa no sistema de arquivos.

A chamada execve e suas derivadas carregam um programa novo para a memória do processo que a executar. O programa antigo é descartado e a *heap*, *stack* e os segmentos de dados do processo são substituídos pelo que estiver definido no novo programa, que iniciará a execução em sua *main*. Esta chamada geralmente é executada logo após um novo processo ser criado, mas também pode ser usada em programas que não criam novos processos.

As funções exec nunca retornam quando bem sucedidas, uma vez que um novo programa é carregado para a memória e inicia sua execução. Se uma chamada a exec retornar, significa, portanto, que um erro aconteceu durante a chamada. A lista completa de erros pode ser consultada no manual de execve, disponível na página <http://man7.org/linux/man-pages/man2/execve.2.html>.

Todas as funções da família exec tem suas implementações baseadas na chamada de sistema execve. No geral, elas representam outras interfaces para a mesma funcionalidade

(KERRISK, 2010). A parte final do nome de cada função é baseada em quais argumentos são aceitos. Por exemplo, execve e execle esperam que a lista de variáveis de ambiente do programa seja passada como argumento.

2.2.5 Finalizando processos

Quando um processo se encerra, seus recursos são liberados e seu término é notificado para seu pai. O término do processo é feito quando ele executa a chamada de sistema *exit*. Segundo (LOVE, 2010), o compilador do C insere uma chamada a *exit* logo após o retorno da função *main*.

O procedimento de encerramento é feito da mesma maneira, não importando se o processo se encerrou normalmente ou de forma involuntária – através de um sinal ou um erro inesperado. Após o término do processo, todos os recursos que estavam em uso por ele são liberados – juntamente com seu espaço de endereçamento – e ele não poderá mais ser executado. Seu descritor continua existindo para que seu pai seja capaz de recuperar alguma informação sobre ele. Neste momento, ele está no estado de zumbi e seu pai deve usar essa informação ou avisar ao *kernel* que não se interessa por ela, para só então seu descritor ser liberado (LOVE, 2010).

Um processo pode requisitar informações sobre um filho terminado através das chamadas de sistema da família *wait*. As funções chamadas estão disponíveis na família *wait* são *wait* e *waitpid*. O comportamento padrão das chamadas *wait* é suspender a execução do processo pai até que um dos seus filhos se encerre. Quando a mudança ocorrer, a chamada a *wait* retornará e o pai poderá seguir sua execução. É possível que o pai se encerre antes de seus filhos. Nesse caso, os processos devem ser adotados por outro processo no sistema, geralmente de inicialização (LOVE, 2010).

2.3 USUÁRIOS E GRUPOS

Todo usuário possui um *login* no sistema e um número identificador único, chamado de *user* ID. Usuários podem pertencer a um ou mais grupos, que também possuem um nome e número identificador único, chamado de *group* ID (KERRISK, 2010).

A razão para a existência de usuários e grupos é determinar o dono de certos recursos do sistema, para ajudar no controle de acesso por parte dos processos a estes recursos. Arquivos e diretórios, por exemplo, possuem um *user ID* associado, representando o usuário dono do arquivo, e um *group ID*, representando o grupo que o arquivo pertence. O usuário com ID 0 possui todos os privilégios de acesso a estes recursos e, por este motivo, é chamado de superusuário ou *root* (KERRISK, 2010).

Cada processo possui IDs de grupos e usuários associados, que são identificadores usados para determinar o dono do processo. Estes IDs são muitas vezes referidos como

credenciais do processo. Um processo pode possuir os seguintes identificadores (KERRISK, 2010):

- *Real User ID e Group ID*: Representam o usuário e grupo donos do processo. Quando um processo é criado, ele irá possuir os mesmos donos que seu processo pai.
- *Effective User ID e Group ID*: Estes identificadores são usados para conferir as permissões de um processo quando ele tenta efetuar alguns tipos de operação – como chamadas de sistema, por exemplo. Geralmente este conjunto de identificadores é igual ao conjunto de *real IDs*, mas pode ser diferente em duas situações: através do uso de algumas chamadas de sistema ou da execução dos programas que modificam os bits de permissão `set-user-ID` e `set-group-ID` de programas executáveis, como o `chown`, por exemplo[4].
- *File-System User ID e Group ID*: Especificamente no Linux, estes identificadores são usados para verificar permissões para o processo efetuar operações no sistema de arquivos, como abrir arquivos ou modificar o dono de um diretório. Assim como os *effective IDs*, estes identificadores geralmente são iguais aos *real IDs*, mas podem ser diferentes pelos mesmos motivos.

2.3.1 *Capabilities*

Em sistemas UNIX tradicionais, os processos eram divididos em duas categorias: privilegiados e não privilegiados. Processos privilegiados (*privileged*) são processos que pertencem ao usuário *root*, e não são submetidos a nenhuma checagem de permissão por parte do *kernel* quando tentam efetuar uma operação privilegiada. Os processos não privilegiados (*unprivileged*) são submetidos às checagens de permissão de acordo com seus *ids* de usuário e grupo, conforme descrito em 2.3.

Devido a essa falta de granularidade, um processo só consegue efetuar qualquer operação privilegiada caso seu dono seja o superusuário. O problema é que, quando o superusuário executa um processo, não é feita qualquer checagem de permissão, independente da operação. Dessa forma, o processo em questão tem permissão de efetuar uma série de outras operações privilegiadas como, por exemplo, ter acesso a arquivos que ele normalmente não teria. Executar um processo com o superusuário pode ser inseguro se o processo se comportar de maneira imprevisível, seja por alguma condição inesperada – como um bug – ou por estar executando um programa malicioso (KERRISK, 2010).

No Linux, em vez de verificar se o processo é privilegiado ou não, as permissões do processo são divididas em unidades chamadas *capabilities*. Cada uma das operações

[4] Quando um programa com esses bits modificados for executado, os valores do *effective user ID* e *effective group ID* será o mesmo valor configurado no arquivo executável do programa

privilegiadas do sistema está associadas com uma *capability* e o processo poderá efetuar esta operação apenas se possuir a *capability* necessária para tal – independente do ID de seu usuário[5]. A lista atualizada de *capabilities* disponíveis no Linux pode ser conferida no manual sobre capabilities[6].

As *capabilities* pertencem a uma *thread*. A representação hexadecimal das *capabilities* da *thread* principal de um processo pode ser consultada no arquivo `</proc/PID/status>`, mais especificamente nos campos `CapPrm`, `CapEff` e `CapInh`. Se o programa possuir mais de uma *thread*, é possível consultar as *capabilities* de cada *thread* no arquivo `</proc/PID/task/TID/status>`. Quando um processo é criado através de *fork*, o novo processo herdará as *capabilities* do processo pai (maiores detalhes sobre fork na seção 2.2.2).

2.3.1.1 *Capabilities* do Processo

Cada processo possui três conjuntos, que podem conter zero ou mais capabilities. Kerrisk (KERRISK, 2010) lista os conjuntos como sendo os seguintes:

- *Permitted*: São as *capabilities* que o processo tem o direito de empregar, caso seja necessário. Este conjunto é usado para determinar quais *capabilities* podem aparece no conjunto *effective*. Quando um processo perde uma dessas *capabilities*, não poderá readquiri-la, a menos que execute um programa (através de uma das chamadas *exec*), que lhe confere esta capability (ver 2.3.1.2).

- *Effective*: É o conjunto de *capabilities* que o *kernel* usa para checar as as permissões do processo no momento que ele efetua uma operação privilegiada. Apenas *capabilities* do conjunto *permitted* podem estar neste conjunto.

- *Inheritable*: Quando um novo programa for executado (através de uma chamada `exec`, as *capabilities* deste conjunto serão adicionadas ao conjunto *permitted*.

2.3.1.2 *Capabilities* de Arquivos

Se um arquivo executável contém conjuntos de *capabilities* associados, os mesmos serão usados para determinar as *capabilities* do processo que executá-lo. Assim como as *capabilities* do processo – descritas em 2.3.1.1, existem três conjuntos de *capabilities* de arquivos (KERRISK, 2010):

- *Permitted*: É o conjunto de *capabilities* que será adicionada ao conjunto *permitted* do processo, durante a chamada a uma das chamadas a `exec`.

[5] Quando um processo é executado pelo superusuário, todo o conjunto de *capabilities* é associado ao processo.

[6] <http://man7.org/linux/man-pages/man7/capabilities.7.html>.

- *Effective*: Este campo possui apenas um *bit*. Quando habilitado, durante a chamada a *exec*, as *capabilities* do novo conjunto *permitted*, serão também adicionadas ao conjunto *effective* do processo. Se estiver desabilitado, o conjunto *effective* do processo ficará inicialmente vazio.

- *Inheritable*: Este conjunto, juntamente com o conjunto *inheritable* do processo, determina quais campos devem ser adicionados ao conjunto *permitted* do processo após uma chamada a *exec*.

2.3.1.3 Modificando *Capabilities* de Arquivos e Processos

É possível ler e modificar as *capabilities* de um arquivo diretamente pelo *shell*, através dos comandos `setcap`[7] e `getcap`[8]. Kerrisk (KERRISK, 2010) exemplifica o uso destes comandos mostrando como usá-los para adicionar a *capability* `CAP_SYS_TIME` ao binário `date`, permitindo que um usuário comum (com ID diferente de 0) modifique o horário do sistema.

Também é possível modificar as *capabilities* de um processo programaticamente, através da chamada de sistema `capset`[9] ou através da API *libcap* (KERRISK, 2010).

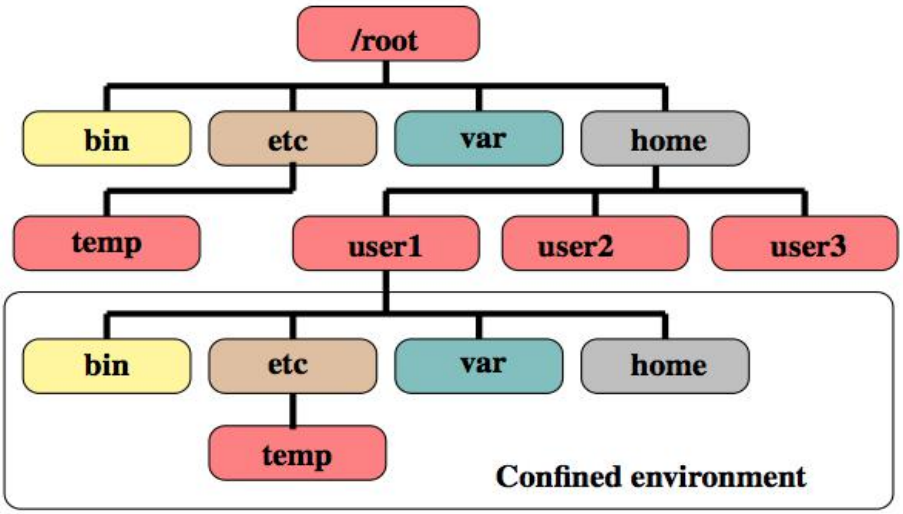
2.4 DIRETÓRIO RAIZ DO PROCESSO

Quando um *container* é criado, é preciso fornecer a ele um sistema de arquivos, contendo os binários e bibliotecas necessários para os processos que serão executados no *container* (BORATE, 2016). Estes processos fazem uso das dependências locais ao sistema de arquivos, e não devem ser capazes de acessar arquivos e diretórios do sistema hospedeiro. A figura 4 mostra uma estrutura de diretórios que pode ser usada como sistema de arquivos do *container*.

A chamada de sistema *chroot* modifica o processo que a executou, trocando a raiz da sua árvore de diretórios (e por consequência a de seus filhos) para o diretório especificado na chamada. Segundo o manual de *chroot*[10], esta chamada não modifica o diretório de trabalho do processo, portanto é possível que, após sua execução, o diretório de trabalho esteja fora da nova árvore de diretórios. É comum observar o termo *chroot jail* sendo usado como referência a esta chamada, uma vez que ela mantém um programa confinado a uma determinada área do sistema de arquivos. Entretanto, *chroot* não é considerado completamente seguro. Kerrisk (KERRISK, 2010) diz que existem diversas maneiras de um processo privilegiado acessar diretórios fora do confinamento. Ele segue mostrando

[7] <http://man7.org/linux/man-pages/man8/getcap.8.html>
[8] <http://man7.org/linux/man-pages/man8/setcap.8.html>
[9] <http://man7.org/linux/man-pages/man2/capget.2.html>
[10] <http://man7.org/linux/man-pages/man2/chroot.2.html>

Figura 4 – Estrutura de diretórios de um *chroot*



Fonte: Borate, Chavan (2016, p. 3)

que até mesmo processos não privilegiados podem explorar determinadas situações para quebrar o confinamento.

No Linux existe a chamada de sistema *pivot_ root*[11], que modifica o ponto de montagem do *root* de todos os processos que compartilham o mesmo *mount namespace* (2.5.4). Os *container runtime's* modernos fazem uso desta chamada de sistema, conforme mostrado na documentação[12] do runc[13]. A implementação atual do projeto de código (detalhado no capítulo 3) faz uso apenas de chroot.

2.5 *NAMESPACES*

Namespaces são abstrações que dão a um conjunto de processos a visão de que determinados recursos globais do sistema operacional são de seu uso exclusivo. As modificação que eventualmente forem feitas ao recurso permanecerão visíveis aos processos em execução dentro do *namespace*, mas não serão refletidas aos outros processos do sistema. A ideia principal dos *namespaces* é particionar recursos entre grupos de processos, permitindo que cada grupo tenha uma visão diferente dos recursos disponíveis no sistema operacional (ROSEN, 2014). O uso de *namespaces* por processo surgiu no sistema operacional Plan9 da Bell Labs (PIKE DAVE PRESOTTO, 1993; BHATTIPROLU ERIC W. BIEDERMAN, 2008). Como o Plan9 possuía o conceito de tudo ser um arquivo ("everything is a file"), sua implementação consistia basicamente de *namespaces* do sistema de arquivos. O Linux, por sua vez, implementa *namespaces* por processo para diversos tipos de recurso.

[11] <http://man7.org/linux/man-pages/man2/pivot_root.2.html>
[12] <https://github.com/opencontainers/runc/blob/master/libcontainer/SPEC.md>
[13] <https://github.com/opencontainers/runc>

Cada processo pode fazer parte de um conjunto de *namespaces*.

Em *clusters* de computadores, é a comum a necessidade de se mover aplicações entre os diferentes nós. Essa necessidade pode existir por diferentes motivos, como aliviar um nó sobrecarregado, efetuar manutenções programadas ou até mesmo recuperar um nó defeituoso (BIEDERMAN, 2006; BHATTIPROLU ERIC W. BIEDERMAN, 2008).

Enquanto estiver em execução, uma aplicação fará uso de diferentes recursos globais do sistema – como seu PID, identificadores IPC, arquivos e diretórios, entre outros. Caso esta aplicação seja movida para um sistema diferente, não é garantia de que os mesmos recursos estejam disponíveis no novo nó. É possível executar a aplicação em um ambiente isolado através de ferramentas de virtualização baseadas em *Hypervisors*, como o Xen, mas isso gera impacto na performance, uma vez que cada máquina virtual possui seu próprio kernel. O objetivo dos namespaces é resolver este problema sem a necessidade de múltiplos sistemas operacionais em uma única máquina (BIEDERMAN, 2006; BHATTIPROLU ERIC W. BIEDERMAN, 2008). Os objetivos e requerimentos para a implementação de *namespaces* no *kernel* do Linux são (BHATTIPROLU ERIC W. BIEDERMAN, 2008):

- As aplicações devem executar em *namespaces* como se estivessem em execução no sistema hospedeiro
- *Namespaces* não devem adicionar nenhum custo de performance significativo
- As ferramentas administrativas e utilitários do sistema operacional devem funcionar do mesmo jeito dentro e fora de um namespace
- *Namespaces* devem ser parte do *kernel* e não um módulo ou fazer parte apenas de uma versão customizada do Linux.

No momento da implementação dos *namespaces*, foram identificados dez recursos diferentes do *kernel* passíveis de isolamento (BIEDERMAN, 2006). Entretanto, até a versão 4.18 do Linux, apenas sete destes tipos haviam sido implementados. São eles: *mount* (2.5.4), *ipc*, *network*, *pid* (2.5.5), *userid* (2.5.6), *hostinfo* (2.5.7) e *cgroup*[14]. Os outros tipos de *namespaces* identificados por Bierderman mas ainda não implementados no *kernel* são *security*, *device* e *time* (BIEDERMAN, 2006).

2.5.1 Estrutura nsproxy

Conforme dito em 2.2.3, no Linux, todo processo é criado através da chamada de sistema *clone*. Ela se difere de *fork* porque permite que partes do contexto do novo processo sejam compartilhados ou duplicados. Cada tipo de recurso que será compartilhado ou duplicado é especificado através de parâmetros da chamada de sistema. Quando um recurso

[14] Este *namespace* não fazia parte do conjunto inicial proposto por Biederman (BIEDERMAN, 2006) uma vez que no momento da implementação dos *namespaces*, não existia o conceito de *control groups*.

do processo pai é duplicado durante uma chamada a *clone*, se diz que recurso está sendo clonado (BHATTIPROLU ERIC W. BIEDERMAN, 2008). Também é possível clonar um recurso depois do processo ter sido criado, através da chamada de sistema *unshare* (mais detalhes em 2.5.2).

Em 2.1.2 é mostrado que cada processo no Linux é representado por uma estrutura de dados chamada `task_struct`. Uma das responsabilidades da `task_struct` é armazenar informações sobre *namespaces*. Ao invés de adicionar um ponteiro para cada *namespace* diretamente na `task_struct`, a estrutura `nsproxy` é que guarda essas informações. Ela está definida no cabeçalho `<include/linux/nsproxy.h>` no código fonte do Linux e seus campos podem ser vistos no esboço de código 2.3.

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
        struct cgroup_namespace *cgroup_ns;
};
```

Código 2.3 – Definição da estrutura `nsproxy`

Segundo (ROSEN, 2014), cada estrutura `nsproxy` é referenciada por uma ou mais `task_struct`. Isso evita que cada `task_struct` tenha que referenciar vários ponteiros para os diferentes tipos de *namespace*, uma vez que apenas o ponteiro para `nsproxy` é necessário. Além disso, no momento da criação de um novo processo, é necessário apenas incrementar o campo *count* da estrutura, que guarda o número de processos referenciando um `nsproxy`. Quando um processo termina, esse campo é decrementado.

Quando um novo tipo de *namespace* é adicionado ao *kernel*, é criado um novo campo na estrutura `nsproxy`. Um novo parâmetro de clonagem é adicionado, para que processos possam clonar o novo tipo de *namespace* através das chamadas de sistema *clone* e *unshare* (ROSEN, 2014).

Quando o namespace de um determinado tipo é clonado, uma nova estrutura `nsproxy` é alocada. Todos os ponteiros para cada tipo de namespace na estrutura são copiados, exceto aquele do tipo que está sendo criado. O ponteiro do tipo de *namespace* sendo clonado passará a referenciar um novo objeto com o tipo do *namespace* em questão, que será uma cópia idêntica do *namespace* original. A partir deste momento, mudanças neste novo *namespace* não afetarão o original (ROSEN, 2014).

Quando não existirem mais *tasks* referenciando um `nsproxy`, o `nsproxy` correspondente é liberado. Da mesma maneira, se não existir mais nenhum `nsproxy` referenciando um *namespace*, este *namespace* também é liberado (ROSEN, 2014).

2.5.2 unshare

A partir da versão 2.6.16 do *kernel*, foi a adicionada uma nova chamada de sistema chamada *unshare*. Assim como *clone* – detalhada em 2.2.3, ela permite que partes do contexto de execução de um processo compartilhados com outros processos sejam desassociados. Entretanto, ao contrário de *clone*, *unshare* permite que isso seja feito sem a criação de um novo processo. A chamada de sistema *unshare* é exclusiva do Linux.

Partes do contexto de execução de um processo são compartilhados automaticamente quando um novo processo é criado através de *fork* (2.2.2). Quando um processo é criado através de *clone* (2.2.3), é possível determinar quais partes devem ser compartilhadas, mas, na maioria dos casos, é necessário que isso esteja explícito no momento da chamada. A chamada *unshare* permite que um processo (ou *thread*) crie novos recursos – dentre eles *namespaces* – sem a necessidade da criação de novos processos.

A lista de parâmetros aceitos e os possíveis cenários de erro podem ser consultados no manual de *unshare*[15].

2.5.3 setns

Conforme mostrado em 2.1.5, o *kernel* disponibiliza um diretório em `</proc/[pid]/ns>`, contendo arquivos que representam os *namespaces* que um determinado processo faz parte. Estes arquivos podem ser manipulados através de chamadas comuns para leitura e escrita em arquivos. Além do diretório `</proc/pid/ns>`, a API de *namespaces* contém chamadas de sistema para o gerenciamento de *namespaces*.

Um descritor de arquivo é retornado quando um desses arquivos é aberto. Esse descritor pode ser passado como parâmetro para a chamada de sistema setns. Quando um processo faz a chamada a setns, ele se junta ao *namespace* identificado pelo descritor passado como parâmetro. Enquanto o arquivo de um *namespace* permanecer aberto o *namespace* continuará existindo, mesmo que nenhum processo esteja usando este *namespace*. É possível listar todos os namespaces disponíveis no sistema usando o utilitário lsns.

2.5.4 Mount *Namespaces*

Da versão 2.4.19 em diante o Linux possui o conceito de *mount namespaces* por processo. Este *namespace* isola a lista de pontos de montagem[16] do sistema, permitindo aos

[15] <http://man7.org/linux/man-pages/man2/unshare.2.html>

[16] Um ponto de montagem é um diretório no sistema onde um determinado dispositivo, como um *driver* de CD-ROM por exemplo, está montado (WARD, 2015).

processos em execução dentro de *namespaces* diferentes terem visões diferentes da lista de pontos de montagem. Uma modificação feita ao sistema de arquivos por um processo afetará os processos em execução no mesmo *namespace*. As modificações à lista de pontos de montagem são feitas através das chamadas de sistema *mount* e *umount*.

Por padrão, um processo recém criado irá compartilhar o *mount namespace* de seu pai. Sendo assim, este processo enxergará a mesma lista de pontos de montagem que o processo que o criou (KERRISK, 2010). As informações mostradas nos arquivos do diretório do processo em /proc, (descritos em 2.1.5), são referentes ao *namespace* que o processo faz parte, ou seja, o conteúdo destes arquivos será o mesmo para todos os processos que compartilhem o mesmo *namespace*.

Quando um processo cria um novo *mount namespace* via *clone* (2.2.3) ou *unshare* (2.5.2), a lista de pontos de montagem do novo *namespace* será uma cópia da lista do processo que o criou. Subsequentes modificações à lista de pontos de montagem no novo *namespace* via *mount* ou *umount* não irão afetar, por padrão, a lista original. A criação de um novo *mount namespace* requer que o processo possua a *capability* `CAP_SYS_ADMIN` (*capabilities* são detalhadas em 2.3.1).

2.5.5 PID *Namespaces*

PID *namespaces* isolam os números disponíveis de PID, o que significa que processos em *namespaces* diferentes podem possuir o mesmo número como identificador. Com isso é possível suspender um processo dentro de um *container* e movê-lo para outro *host* sem que seu PID precise ser modificado.

O primeiro processo que entrar no novo PID namespace (criado por uma chamada a clone com o parâmetro `CLONE_NEWPID` ou o primeiro filho criado após uma chamada a *unshare* com a mesmo parâmetro), será o processo de inicialização no novo *namespace*. O PID deste processo será 1 no novo *namespace*. Caso exista algum processo órfão (veja 2.2.1) no *namespace*, o processo de inicialização do *namespace* – ou seja, o primeiro processo que entrar no *namespace*, será o seu pai, a menos que exista algum ancestral do processo órfão responsável pela sua limpeza.

Se o processo de inicialização de um *namespace* terminar, o *kernel* enviará um sinal `SIGKILL` para todos os processos dentro do *namespace*. Caso seja feita uma chamada a *fork* após o término do processo de inicialização do *namespace*, um erro `ERRNOMEM` será retornado, uma vez que não é possível criar novos processos em um *namespace* cujo processo de inicialização tenha terminado.

Apenas sinais que o processo de inicialização tenha estabelecido um *signal handler* podem ser enviados para ele. Essa restrição é válida até mesmo para processos privilegiados e serve para prevenir que processos dentro do *namespace* matem acidentalmente o processo de inicialização. Os processos em *namespaces* ancestrais possuem as mesmas restrições, exceto para os sinais `SIGKILL` e `SIGSTOP`, que, quando partem de um processo

em um namespace ancestral, são enviados e não podem ser ignorados pelo processo de inicialização do *namespace*.

2.5.6 *User Namespaces*

Os *user namespaces* isolam as credenciais de um processo: uid e gid, o diretório raiz e *capabilities* (mais detalhes sobre credenciais de um processo em 2.3.1). O manual sobre *user namespaces*[17] explica que um processo pode possuir um usuário dentro de um *user namespace* e outro fora. Isso permite que o processo tenha todos os privilégios dentro de um *namespace*, mas tenha acesso restrito fora dele.

User namespaces podem ser encadeados, ou seja, um *user namespace* terá sempre um ancestral e zero ou mais filhos. O primeiro namespace do sistema é chamado de *namespace raiz* e é o único que não possui qualquer ancestral. O pai de um *namespace* é o *namespace* do processo que o criou – ou seja, que fez a chamada a *clone* ou a *unshare* com o parâmetro `CLONE_NEWUSER` – faz parte.

Todo processo pertence a um único *user namespace*. Se um processo foi criado via *fork* ou *clone* (sem o parâmetro `CLONE_NEWUSER`), ele fará parte do *namespace* do seu processo pai. Caso `CLONE_NEWUSER` seja usada no momento da chamada a *clone*, o processo criado será membro de um novo *user namespace*. Da mesma maneira, se um processo fizer uma chamada unshare com a mesmo parâmetro, ele passará a fazer parte do *novo namespace*.

2.5.7 UTS *Namespaces*

UTS namespaces isolam o *hostname* e o NIS *domain name* do sistema. Esses identificadores são retornados através da chamadas de sistema *uname*, *gethostname* e *getdomainname*. É possível configurá-los com as chamadas de sistema *sethostname* e *setdomainname*. Realizar uma destas chamadas dentro de um UTS *namespace* modificará esses identificadores do sistema apenas para os processos em execução dentro dele.

2.6 *RESOURCE LIMITS*

É possível limitar os recursos disponíveis para os programas em execução no Linux. Os limites impostos podem ser físicos (como limites no consumo de memória), podem ser relacionados às políticas do sistema (como tempo de CPU que um processo tem disponível) ou ainda de implementação (como tamanho máximo permitido para nomes de arquivo) (STONES, 2008).

Os limites representam o maior valor possível para o uso de um determinado recursos. A maioria desses limites é configurado individualmente por processo, exceto o limite do número máximo de processos, que é uma configuração do usuário (EMELIANOV

[17] <http://man7.org/linux/man-pages/man7/user_namespaces.7.html>

DENIS LUNEV, 2007). O principal objetivo desses limites é proteger o sistema contra processos mau comportados (que consomem uma grande quantidade de memória, por exemplo).

É possível ler e configurar os limites para um determinado recursos através das chamadas de sistema getrlimit e setrlimit. Cada tipo de recurso possui dois tipos de limite: *soft limit* e *hard limit*. O primeiro argumento de ambas as chamadas é um inteiro que representa o recurso para o qual o limite configurado será aplicado ou consultado. A lista de recursos disponíveis pode ser consultada no manual dessas funções[18].

O segundo parâmetro dessas chamadas é uma estrutura do tipo `rlimit`, que possui a configuração desses valores. A definição dos campos dessa estrutura pode ser consultada no esboço de código código 2.4

```
struct rlimit {
    rlim_t rlim_cur;   /* soft limit */
    rlim_t rlim_max;   /* hard limit */
};
```

Código 2.4 – Estrutura `struct rlimit`

O *soft limit* é um valor que será aplicado para o recurso correspondente. Já o *hard limit* é o maior valor possível que um recurso pode ter. Um processo sem os privilégios necessários pode configurar apenas o seu *soft limit* para um valor entre 0 e o *hard limit* ou reduzir (de maneira irreversível) seu *hard limit*. Processos que possuem a capability `CAP_SYS_RESOURCE` podem modificar arbitrariamente ambos os valores. Configurar um recurso com o valor `RLIM_INFINITY` significa que este recurso não possui nenhum limite.

2.7 *CONTROL GROUPS*

Segundo (MENAGE, 2007), embora existissem mecanismos eficientes para monitorar e controlar o consumo de recursos por parte de um único processo, o suporte para essas operações era limitado em se tratando de grupos de processos relacionados. Essa funcionalidade foi adicionada ao *kernel* na versão 2.6.24 e recebeu o nome de *control groups* ou simplesmente cgroups.

O manual[19] descreve um cgroup como uma coleção de processos onde uma série de restrições ou parâmetros são aplicados através de um sistema de arquivos chamado "cgroup filesystem".

O comportamento de um processo em um cgroup é modificado por componentes do *kernel* chamados de controladores (*controllers*). Existem vários controladores diferentes,

[18] <http://man7.org/linux/man-pages/man2/getrlimit.2.html>
[19] <http://man7.org/linux/man-pages/man7/cgroups.7.html>

cada um responsável por uma tarefa, como limitar o tempo de CPU ou o total de memória disponível.

O manual de cgroups explica que desde a versão 2.6.24, vários controladores foram adicionados ao *kernel*, muitas vezes de forma descoordenada, causando inconsistências entre os controladores e tornando seu uso complexo. A partir da versão 3.10 do Linux, foi iniciado o desenvolvimento de uma nova implementação dos cgroups, que visava corrigir os problemas da versão original. Essa nova versão foi chamada de "cgroups version 2", e a original de "cgroups version 1". Por motivos de compatibilidade, a versão original ainda existe e não há planos para removê-la.

2.8 PREOCUPAÇÕES COM SEGURANÇA

Soluções tradicionais de virtualização fazem uso de *hypervisors* para prover o isolamento e compartilhamento de recursos de uma máquina física entre diferentes máquinas virtuais. Cada uma delas possui seu próprio sistema operacional e todas as suas instruções são executadas de forma isolada entre elas e o sistema hospedeiro (NICK CAILLIAU ENIAS, 2016).

Quando comparados com as tecnologias de *container*, os *hypervisors* oferecem um isolamento mais completo, uma vez que cada máquina virtual executa em seu próprio sistema operacional. Esse isolamento tem um custo, uma vez que, ao emular um sistema inteiro, podem ocorrer impactos na performance das aplicações e nos tempos que inicialização dessas máquinas (NICK CAILLIAU ENIAS, 2016).

A virtualização baseada em *containers*, por sua vez, utiliza recursos do próprio sistema hospedeiro, sem a necessidade de se emular um novo sistema operacional completo. Isso traz uma melhoria na performance das aplicações enquanto ainda oferece o isolamento. A melhoria se deve ao fato das instruções estarem sendo executadas diretamente no sistema hospedeiro. Os tempos de inicialização dos containers, em geral, são menores, uma vez que não é necessário realizar o *boot* de todo o sistema, mas apenas alocar os recursos que serão utilizados pelos processos em execução dentro dele (NICK CAILLIAU ENIAS, 2016).

Como o container executa no próprio sistema hospedeiro, algumas preocupações com segurança são necessárias. (HAYDEN, 2015) cita que, caso o usuário seja *root* dentro do container (ou seja, um usuário com UID 0), ele também será *root* no sistema hospedeiro. Isso possibilita que um invasor que tenha encontrado uma maneira de escapar do confinamento, tenha acessos privilegiados a todas as operações no sistema. Hayden mostra que, através das *capabilities*, é possível limitar o conjunto de operações privilegiadas que um processo pode executar. *Runtimes* como o Docker configuram por padrão um conjunto mínimo de *capabilities* no momento da criação de seus containers. Ele também mostra que é possível, com a ajuda dos *user namespaces*, garantir que um usuário *root* dentro do

container não seja, necessariamente, um usuário privilegiado no sistema hospedeiro.

(NICK CAILLIAU ENIAS, 2016) e (KERRISK, 2010) mostram possíveis problemas com a chamada de sistema *chroot*, usada para modificar a raiz da árvore de diretórios, mostrando que um usuário privilegiado conseguiria subverter o confinamento. O LXC[20] faz uso da chamada *pivot_root*, que, segundo Nick, abstrai completamente a visão do sistema de arquivos por parte dos processos em execução no container.

Por fim, (HAYDEN, 2015) explica que como os containers fazem uso dos recursos do sistema, vulnerabilidades no *kernel* podem comprometer a sua segurança. Por isso, é essencial que ele seja constantemente atualizado, garantindo as correções para as falhas de segurança conhecidas.

[20] <https://linuxcontainers.org/>

3 PROJETO DE CÓDIGO

O escopo deste trabalho incluiu a implementação de uma aplicação de linha de comando capaz de criar e executar *containers* seguindo a especificação descrita pela *Open Containers Initiative*[1]. Alguns detalhes da especificação podem ser vistos em 3.1. O projeto recebeu o nome de Plankton e seu código foi hospedado no Github, podendo se acessado em <https://github.com/bernardolins/plankton>

3.1 ESPECIFICAÇÃO

Para criar um *container* segundo a especificação é necessário um pacote, chamado de *bundle* na especificação. O pacote deve conter apenas um arquivo no formato JSON e um subdiretório. Um mesmo *bundle* pode ser usado para a criação de vários *containers*. O quadro 3.1 mostra um exemplo da estrutura de diretórios de um pacote chamado `my_bundle`.

```
$ tree -L 2 my_bundle/
my_bundle/
|-- config.json
|-- rootfs/
|---- bin/
|---- dev/
|---- etc/
|---- home/
|---- proc/
|---- root/
|---- sys/
|---- tmp/
|---- usr/
|---- var/
```

Código 3.1 – Estrutura de diretórios de um *bundle*

A especificação determina que o arquivo JSON se chame `config.json`. Este arquivo contém diversos parâmetros que serão usados para a criação do *container*, como o programa que será executado, diretórios a serem montados e variáveis de ambiente. O quadro 3.2 mostra um exemplo básico de arquivo de configuração. O *container runtime* deve ler

[1] https://github.com/opencontainers/runtime-spec/blob/master/spec.md

este arquivo e criar um novo processo com todos os objetos descritos nele. Caso a configuração contenha valores inválidos ou campos obrigatórios ausentes, um erro deve ser retornado.

```
{
    "ociVersion": "1.0.0-dev",
    "process": {
        "user": {"uid": 0, "gid": 0},
        "args": ["ls", "-la"],
        "env": ["TERM=xterm"],
        "cwd": "/"
    },
    "root": {"path": "rootfs", "readonly": true},
    "hostname": "my-container",
    "mounts": [
        {
            "destination": "/proc",
            "type": "proc",
            "source": "proc"
        }
    ],
    "linux": {
        "namespaces": [
            {"type": "pid"},
            {"type": "mount"},
            {"type": "uts"}
        ]
    }
}
```

Código 3.2 – Exemplo de arquivo de configuração válido

A implementação do Plankton contém uma estrutura chamada *Environment*, que é criada a partir da leitura do arquivo de configuração. Essa estrutura é um dos parâmetros usados na criação do processo de inicialização do *container*. Em 3.3 é mostrado como o arquivo de configuração é transformado nesta estrutura.

Os campos da configuração podem ser diferentes dependendo do sistema operacional. O escopo deste projeto trata apenas da configuração para sistemas Linux. Os campos e seus valores possíveis estão descritos na especificação (OCI, 2018).

Além do arquivo de configuração, o pacote deve conter um subdiretório que será montado como raiz do processo de inicialização do *container*. A seção 2.4 explica como funciona o processo de mudança da árvore de diretórios de um processo e 3.3 mostra como o diretório raiz é modificado quando um *container* é criado. Esse diretório também é chamado de imagem do *container*. Segundo a especificação, o nome do diretório deve

ser especificado no campo `root.path` do arquivo de configuração. Caso o diretório não exista ou contenha um nome diferente do especificado no arquivo, um erro será retornado.

Conforme descrito em 2.4, o processo de inicialização do *container* só terá acesso a arquivos e diretórios dentro da imagem. Portanto, a imagem do *container* deve possuir todas as dependências necessárias para executar o programa definido na configuração, como binários, arquivos de configuração e bibliotecas.

A OCI possui uma especificação para formatos de imagem, mas a mesma está além do escopo deste trabalho. As imagens usadas durante o desenvolvimento foram geradas com o Docker, uma vez que o formato de imagens geradas por ele é compatível com o padrão definido pela OCI[2]. O comando mostrado no exemplo 3.3 gerou um diretório chamado rootfs a partir de uma imagem Docker chamada *busybox*. A documentação[3] da CLI do Docker mostra detalhes sobre os comandos usados neste exemplo. A imagem usada no exemplo está disponível no repositório público chamado Dockerhub[4].

```
$ docker export \
        $(docker create busybox) | \
        tar -C rootfs -xvf -
```

Código 3.3 – Comando docker para gerar um rootfs

3.2 AMBIENTE DE DESENVOLVIMENTO

O desenvolvimento foi feito na linguagem Rust[5], que possui um ecossistema robusto para a escrita de aplicações de linha de comando. A escolha da linguagem se deve, principalmente, pela facilidade de gerenciamento do projeto através de utilitários como o Cargo[6], uma ferramenta que ajuda a criar e compilar um projeto Rust, executar seus testes e gerenciar suas dependências. A versão do Rust usada no desenvolvimento foi a 1.34. O sistema usado foi o Fedora 29 com o Linux na versão 4.18. Contudo, o projeto deve ser capaz de executar em qualquer distribuição com Linux 4.6 ou superior.

Programas escritos em Rust podem fazer uso de códigos de outras linguagens através de um mecanismo chamado FFI (foreign function interface). O pacote libc[7] disponibiliza as ferramentas necessárias para que seja possível o uso de funções e tipos de dados escritos em C em um programa Rust, através de uma abstração que não causa qualquer impacto de performance[8].

2 <https://github.com/opencontainers/image-spec>
3 <https://docs.docker.com/engine/reference/commandline/cli/>
4 <https://hub.docker.com/>
5 <https://www.rust-lang.org/>
6 <https://doc.rust-lang.org/cargo/guide/>
7 <https://github.com/rust-lang/libc>
8 <https://blog.rust-lang.org/2015/04/24/Rust-Once-Run-Everywhere.html>

O pacote nix[9] faz uso da libc para disponibilizar uma interface mais simples para programas que executam em sistemas Unix, garantindo um tratamento de erros de acordo com a linguagem e provendo tipos e estruturas que facilitam o desenvolvimento.

As dependências de um projeto Rust são gerenciadas pelo Cargo. Elas são definidas em um arquivo chamado Cargo.toml, que é lido no momento da compilação do programa. Quando uma nova dependência é definida neste arquivo, o Cargo se encarregará do *download* e da instalação na próxima compilação. Além disso, este arquivo pode conter outras informações sobre o programa, como nome, versão e seu mantenedor. O quadro 3.4 mostra um exemplo de arquivo Cargo.toml.

```
[package]
name = "plankton"
version = "0.1.0"

[dependencies]
clap = {version = "2.32", features = ["yaml"]}
serde_json = "1.0"
nix = "0.13.0"
libc = "0.2.51"
```

Código 3.4 – Exemplo de arquivo Cargo.toml

3.3 CRIANDO UM *CONTAINER*

O Plankton possui um comando para a criação de um *container*. Este comando aceita dois parâmetros como entrada: um identificador único para o *container* e o caminho no sistema de arquivos onde está localizado o pacote. Caso o pacote não possua o formato descrito em 3.1, um erro é retornado. Se o `container_id` já estiver em uso, o programa é encerrado e uma mensagem de erro é mostrada para o usuário.

O exemplo 3.5 mostra os comandos necessários para compilar o projeto e, em seguida, usá-lo para criar um *container* chamado **my-container**. No exemplo foi usado o arquivo de configuração mostrado em 3.2 e o rootfs gerado a partir do comando mostrado em 3.3.

O comando **cargo build** irá verificar as dependências listadas no arquivo Cargo.toml e baixá-las caso seja necessário. Em seguida, o projeto será compilado juntamente com suas dependências. Por padrão será gerado um binário no diretório `<./target/debug/plankton>`.

O comando **plankton run** cria e executa um novo *container*. O primeiro parâmetro é obrigatório e representa o nome do *container* que será criado. Esse nome deve ser

[9] <https://github.com/nix-rust/nix>

único dentro do sistema. A opção **-b** permite ao usuário especificar um caminho para um diretório *bundle* e, caso não seja definido, o diretório local será usado como caminho para o *bundle*.

```
$ cargo build
$ sudo target/debug/plankton run my-container \
           -b /opt/containers/mycontainer/
total 40
drwxr-xr-x    2 root     root         12288 Jul 16 01:13 bin
drwxr-xr-x    2 root     root            40 Dec  2 11:59 dev
drwxr-xr-x    3 root     root          4096 Jul 28 01:49 etc
drwxr-xr-x    2 nobody   nogroup       4096 Jul 16 01:13 home
dr-xr-xr-x   97 root     root             0 Dec  2 17:24 proc
drwx------    2 root     root          4096 Aug  3 21:01 root
drwxr-xr-x    3 root     root          4096 Sep  7 20:18 sys
drwxrwxrwt    2 root     root          4096 Jul 16 01:13 tmp
drwxr-xr-x    3 root     root          4096 Jul 16 01:13 usr
drwxr-xr-x    4 root     root          4096 Jul 16 01:13 var
```

Código 3.5 – Exemplo do comando plankton run

Se os parâmetros passados no comando estiverem corretos, o arquivo de configuração é lido e validado e o estado atual do *container* é salvo em um arquivo. A seção 3.4.4 mostra o formato desse arquivo. Caso exista algum problema na sintaxe, algum valor inválido ou campo obrigatório ausente, um erro é retornado e o programa se encerra. Se o arquivo de configuração for válido, ele é transformado em uma estrutura interna chamada *Environment*. Essa estrutura é criada a partir do arquivo de configuração, e é responsável por definir os valores padrão que não foram definidos na configuração. O *Environment* evita que a lógica da criação do *container* esteja acoplada ao formato da configuração, que pode mudar ao longo do tempo.

Conforme dito em 3.1, os campos da configuração mudam de acordo com o sistema usado. Atualmente, o projeto suporta apenas o formato de configuração do Linux, portanto, existe apenas o *Environment* para este sistema. Caso novos sistemas sejam adicionados, novas estruturas deste tipo devem ser criadas. Nem todos os campos da especificação foram implementados pelo *Environment*. O capítulo 4 mostra projetos sobre implementações futuras. Os campos presentes no *Environment* Linux são:

- `argv`: Lista de *strings* contendo o programa que será executado pelo processo de inicialização e seus argumentos. O primeiro elemento dessa lista é o nome do programa.

- `rootfs`: *String* que representa o caminho no sistema de arquivos para o diretório que será usado como a raiz do processo de inicialização do *container*, conforme explicado em 2.4.

- `working_dir`: *String* que representa o caminho absoluto para o diretório de trabalho do processo de inicialização do *container*. Esse diretório deve existir na imagem do *container*, caso contrário, um erro será retornado para o usuário. Se este campo não estiver definido na configuração, o valor `/` será adotado como padrão.

- `hostname`: *String* que representa o valor mostrado como *hostname* do processo de inicialização do *container*. Esse campo só pode ser definido caso o arquivo de configuração também tenha definido um *namespace* UTS privado para o *container*. Mais informações podem ser vistas em 2.5.7.

- `namespaces`: Lista de objetos contendo os *namespaces* que o processo de inicialização se juntará. Esse objeto pode conter uma *string* com um tipo de *namespace* ou um caminho para um arquivo no sistema de arquivos </proc/[PID]/ns>. Caso o primeiro esteja definido, um novo *namespace* será criado para o *container*. No caso do segundo, o processo de inicialização do *container* se juntará ao mesmo *namespace* do processo com identificador [PID]. Para mais detalhes veja 2.1.5, 2.5.2 e 2.5.3.

- `mount_list`: Lista de objetos com informações sobre os diretórios do sistema hospedeiro que serão montados no *container*.

- `env_vars`: Lista de *strings* contendo variáveis de ambiente que serão definidas no *container*.

- `rlimits`: Lista de objetos contendo as configurações dos limites que serão aplicados para os processos do *container*. Para mais informações veja 2.6.

- `user`: Objeto com informações de usuários e grupos a serem definidas no *container*.

A estrutura *Environment* é a responsável por criar o processo de inicialização do *container*. Esse procedimento é feito no método `spawn_process`, que tem sua implementação no arquivo <src/libcontainer/linux/environment/mod.rs>. A lista a seguir enumera todos os passos realizados por ele:

1. Antes mesmo de criar o processo de inicialização do *container*, spawn_process cria os novos namespaces descritos no *Environment* e faz uma chamada a *unshare* para se associar a eles. A seção 2.5.2 mostra como essa chamada funciona.

2. É feita uma chamada a "clone", que cria o processo de inicialização do *container*. Conforme dito em 2.2.3, nesse momento, o novo processo é uma cópia quase idêntica ao seu pai e já faz parte dos novos *namespaces* criados.

3. É feita uma chamada a "chroot", que modifica a raiz do processo para o diretório configurado no campo "rootfs"do *Environment*. Essa chamada é descrita em 2.4.

4. O diretório de trabalho desse novo processo é modificado para o caminho especificado no campo *working_dir* do *Environment*. Caso este caminho não exista no "rootfs", um erro é retornado.

5. Outros recursos são configurados, como os pontos de montagem, *hostname* e variáveis de ambiente.

6. Os limites especificados no campo rlimits do *Environment* são aplicados, conforme descrito em 2.6.

7. Por fim, é feita uma chamada a execvp, uma das funções da família exec, que modifica a imagem do processo para o programa salvo no campo argv do *Environment*. Conforme descrito em 2.2.4, as funções da família exec nunca retornam em caso de sucesso.

Neste momento o *container* está criado e segue sua execução separado do pai. No pai, o método spawn_process retorna o PID do processo de inicialização do *container*. Caso não seja possível criar o novo processo, um erro é retornado. O *status* do *container* é atualizado para "Running"e o PID do processo de inicialização é salvo no estado. Por fim, é feita uma chamada a wait que, conforme descrito em 2.2.5, causará a suspensão do processo atual até que o *container* se encerre. Quando isso acontecer, seu *status* é atualizado para "Stopped"e o processo do *container runtime* termina.

3.4 CENÁRIOS DE USO

Diferentes cenários foram pensados para demonstrar como mudanças na configuração impactam na criação de um *container plankton* e, consequentemente, nos recursos disponíveis para os processos em execução dentro dele.

3.4.1 Cenário 1: Executar programas

O objetivo deste cenário é demonstrar como executar um programa em um *container*. O comando que foi executado para este cenário pode ser visto no código 3.6. O diretório usado como *bundle* no exemplo contém um arquivo de configuração – cujo conteúdo é mostrado no código 3.7 – e um subdiretório chamado *rootfs* – cujo conteúdo foi gerado a partir da imagem *busybox* do Docker, conforme mostrado no código 3.3.

Segundo a especificação mostrada em 3.1, o programa que será executado no *container* deve estar definido no campo `process.args`. Este campo é uma lista onde o primeiro

item é o caminho para o binário e os outros itens são seus argumentos. O programa usado no exemplo foi o *sleep*[10].

Seguindo o procedimento descrito em 3.3, quando o comando 3.6 foi executado, uma chamada a *clone* foi feita, criando um novo processo. Em seguida o diretório *rootfs* foi montado como raiz do novo processo com a ajuda da chamada *chroot* (detalhada em 2.4) e uma chamada a *exec* foi feita, trocando o programa em execução para *sleep*, conforme mostrado em 2.2.4. O processo pai (*plankton*) esperou o término de seu filho (*sleep*) para, enfim, se encerrar. A seção 2.2.5 detalha esse processo de espera.

O utilitário *pstree*[11] pode ser usado para exibir a árvore de processos no sistema. A imagem 5 mostra a saída exibida por ele durante a execução do container. O PID do processo é exibido entre parênteses, ao lado do nome do programa. Com isso é possível verificar que o processo *plankton* de fato criou um novo processo, que passou a executar o programa *sleep*. Também é possível verificar que o processo *plankton* não se encerrou antes de seu filho.

```
$ sudo plankton run mycontainer -b /opt/containers/mycontainer/
```

Código 3.6 – Comando usado no cenário 1

```
{
    "ociVersion": "1.0.0-dev",
    "process": {
        "user": {"uid": 0, "gid": 0},
        "args": ["/bin/sleep", "60"],
        "env": ["TERM=xterm"],
        "cwd": "/",
    },
    "root": {"path": "rootfs", "readonly": true},
    "mounts": [],
    "linux": {
        "namespaces": []
    }
}
```

Código 3.7 – Configuração do cenário 1

3.4.2 Cenário 2: Limitar o número de arquivos abertos

O objetivo deste cenário é demonstrar como é possível limitar a quantidade de arquivos abertos por um processo que será executado por um container, através da configuração

[10] <http://man7.org/linux/man-pages/man1/sleep.1.html>
[11] <http://man7.org/linux/man-pages/man1/pstree.1.html>

Figura 5 – Cenário 1: Saída do comando pstree

```
$ sudo pstree -p
systemd(1)─┬─NetworkManager(1989)─dhclient(2009)
           │
           ├─agetty(537)
           ├─auditd(17542)───{auditd}(17543)
           ├─bash(18420)
           ├─chronyd(442)
           ├─gssproxy(445)─┬─{gssproxy}(455)
           ├─rpc.statd(17586)
           ├─rpcbind(17582)
           └─sshd(17483)─sshd(2097)──sshd(2100)──bash(2101)──sudo(584)──plankton(586)──sleep(587)
```

de *rlimits*. O comando que foi executado para este cenário pode ser visto no código 3.8. A configuração usada neste exemplo é mostrada no código 3.9. Assim como no exemplo 1, mostrado em 3.4.1, o *rootfs* foi gerado a partir da imagem *busybox* do Docker.

Neste cenário foi usado o comando *ulimit*[12], que mostra todos os limites configurados para o *shell* e os programas criados por ele. Segundo a especificação mostrada em 3.1, para que seja possível modificar os limites dos processos em execução no container, o campo `process.rlimits` deve receber uma lista com as configurações dos *rlimits* que serão aplicados. Mais detalhes sobre a configuração de limites são mostrados na seção 2.6.

O procedimento de criação do container foi parecido com aquele mostrado no exemplo 1 (3.4.1), mas dessa vez, um um passo adicional foi realizado, que foi a aplicação dos limites definidos na configuração. O procedimento completo de criação de containers é detalhado em 3.3.

O único tipo de *rlimit* definido na configuração do exemplo foi `RLIMIT_NOFILE`, que limita o número máximo de arquivos abertos para um processo. Seus valores *soft* e *hard* foram configurados para 512. A saída do comando *ulimit*, exibida na imagem 6, mostra que esse valor foi configurado corretamente para os processos do container.

```
$ sudo plankton run mycontainer -b /opt/containers/mycontainer/
```

Código 3.8 – Comando usado no cenário 2

```
{
    "ociVersion": "1.0.0-dev",
    "process": {
        "user": {"uid": 0, "gid": 0},
        "args": ["/bin/sh", "-c", "ulimit -a"],
```

[12] <https://linux.die.net/man/1/sh>

```
        "env": ["TERM=xterm"],
        "cwd": "/",
        "rlimits": [{"type": "RLIMIT_NOFILE", "hard": 512, "soft": 512}]
    },
    "root": {"path": "rootfs", "readonly": true},
    "hostname": "my-container",
    "mounts": [
        {"destination": "/proc", "type": "proc", "source": "proc"}
    ],
    "linux": {
        "namespaces": [
            {"type": "pid"},
            {"type": "mount"},
            {"type": "uts"}
        ]
    }
}
```

Código 3.9 – Configuração do cenário 2

Figura 6 – Cenário 2: Saída do comando *ulimit*

```
$ sudo plankton run mycontainer -b /opt/containers/mycontainer/
core file size (blocks)         (-c) unlimited
data seg size (kb)              (-d) unlimited
scheduling priority             (-e) 0
file size (blocks)              (-f) unlimited
pending signals                 (-i) 7866
max locked memory (kb)          (-l) 16384
max memory size (kb)            (-m) unlimited
open files                      (-n) 512
POSIX message queues (bytes)    (-q) 819200
real-time priority              (-r) 0
stack size (kb)                 (-s) 8192
cpu time (seconds)              (-t) unlimited
max user processes              (-u) 7866
virtual memory (kb)             (-v) unlimited
file locks                      (-x) unlimited
```

3.4.3 Cenário 3: Configurar *namespaces*

O objetivo deste cenário é demonstrar como funciona a configuração de *namespaces* em um *container* e como ela afeta o comportamento dos processos em execução dentro dele. Para isso, foram criadas duas configurações diferentes: em uma delas foi usado um *namespace* de PID na criação do *container* e na outra não. O comando que foi executado

para criar o container pode ser visto no código 3.10. Assim como nos exemplos 1 e 2, mostrados respectivamente em 3.4.1 e 3.4.2, o *rootfs* foi gerado a partir da imagem *busybox* do Docker.

O programa usado neste exemplo foi o *top*[13], que mostra, entre outras coisas, a lista de processos em execução no sistema. Este comando lê informações presentes no sistema de arquivos `</proc>`, portanto, é necessário que o container monte este diretório para seu funcionamento correto. Segundo a especificação mostrada em 3.1, os *namespaces* devem ser configurados no campo `linux.namespaces`. O valor deste campo deve ser uma lista contendo objetos, onde cada objeto pode possuir os campos *type* e *path*. O campo *type* é obrigatório e guarda o tipo de *namespace* que será criado. Os tipos de *namespaces* são mostrados na seção 2.5.

Na primeira configuração – mostrada no código 3.11 – os *namespaces* do tipo PID e *mount* foram definidos, para que, através do comando *top*, seja possível verificar a lista de processos em execução no *container*. A segunda configuração – mostrada no código 3.12 – difere da primeira apenas pelo fato de não configurar o *namespace* de PID. As saídas da primeira e da segunda execução podem ser vistas nas imagens 7 e 8, respectivamente.

```
$ sudo plankton run mycontainer -b /opt/containers/mycontainer/
```

Código 3.10 – Comando usado no cenário 3

```
{
    "ociVersion": "1.0.0-dev",
    "process": {
        "user": {"uid": 0, "gid": 0},
        "args": ["/bin/top"],
        "env": ["TERM=xterm"],
        "cwd": "/",
        "rlimits": []
    },
    "root": {"path": "rootfs", "readonly": true},
    "mounts": [
        {"destination": "/proc", "type": "proc", "source": "proc"}
    ],
    "linux": {
        "namespaces": [{"type": "pid"}, {"type": "mount"}]
    }
}
```

Código 3.11 – Configuração do cenário 3 (com namespace)

[13] <https://linux.die.net/man/1/top>

```
{
    "ociVersion": "1.0.0-dev",
    "process": {
        "user": {"uid": 0, "gid": 0},
        "args": ["/bin/proc"],
        "env": ["TERM=xterm"],
        "cwd": "/",
        "rlimits": []
    },
    "root": {"path": "rootfs", "readonly": true},
    "mounts": [
        {"destination": "/proc", "type": "proc", "source": "proc"}
    ],
    "linux": {
        "namespaces": [{"type": "mount"}]
    }
}
```

Código 3.12 – Configuração do cenário 3 (sem namespace)

Figura 7 – Cenário 3: Saída do comando top com namespaces

```
Mem: 1836232K used, 204012K free, 560K shrd, 228144K buff, 1315656K cached
CPU: 95.5% usr  0.4% sys  0.0% nic  0.0% idle  0.0% io  3.5% irq  0.4% sirq
Load average: 1.00 1.00 1.00 3/139 1
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 root     R     1308  0.0   0  0.0 /bin/top
```

3.4.4 Inspecionando um *container*

Quando um *container* é criado, seu estado é salvo em um arquivo no formato JSON. O nome do arquivo será o identificador do *container*, passado como parâmetro de linha de comando no momento de sua criação. No Linux, o caminho onde o arquivo será salvo é `</run/plankton/[CONTAINER_ID].json>`. Este arquivo guarda informações importantes para que seja possível consultar o estado do *container* durante sua execução ou reiniciá-lo após seu término. Os campos armazenados no estado são:

- `bundle`: Caminho no sistema de arquivos que contém o *bundle* que foi usado na criação do *container*.

Figura 8 – Cenário 3: Saída do comando top sem namespaces

```
Mem: 1836488K used, 203756K free, 568K shrd, 228224K buff, 1315668K cached
CPU: 97.2% usr  0.4% sys  0.0% nic  0.0% idle  0.0% io  1.8% irq  0.4% sirq
Load average: 1.00 1.00 1.00 2/139 952
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
13009     1 root     R    11820  0.5   0 99.5 ./target/debug/plankton
 2100  2097 1000     S    39600  1.9   0  0.1 sshd: vagrant@pts/0
  952   951 root     R     1312  0.0   0  0.1 /bin/top
17562     1 998      S    1667m 83.3   0  0.0 /usr/lib/polkit-1/polkitd --no-debug
 1989     1 root     S     331m 16.5   0  0.0 /usr/sbin/NetworkManager --no-daemon
  619     1 root     S     290m 14.5   0  0.0 /usr/sbin/VBoxService --pidfile /var/run/vboxadd-service.sh
  365     1 root     S     199m  9.9   0  0.0 /usr/lib/systemd/systemd-journald
 3454  3452 1000     S     123m  6.2   0  0.0 (sd-pam)
11849 11847 root     S     113m  5.6   0  0.0 (sd-pam)
  442     1 996      S    88740  4.3   0  0.0 /usr/sbin/chronyd
```

- `id`: Identificador do *container*, passado como parâmetro de linha de comando no momento da sua criação.
- `pid`: PID do processo de inicialização do *container* no sistema operacional. Conforme dito em 2.5.5, caso o processo seja o primeiro criado em um *pid namespace*, os processos dentro do namespace o enxergarão com o PID 1. Entretanto, para processos em execução fora deste *namespace* seu PID será diferente (geralmente um valor um pouco maior que o PID de seu pai). É este segundo valor que é armazenado neste campo.
- `status`: O *status* atual da execução do *container*. Os possíveis valores para este campo estão descritos na especificação(OCI, 2018).

É possível inspecionar o estado atual da execução de um *container* verificando o conteúdo deste arquivo. Para isso, é preciso passar o identificador do *container* como parâmetro de linha de comando. Caso não exista um arquivo com este identificador, um erro é retornado e o programa se encerra. Se o arquivo existir, as informações contidas nele serão exibidas. A CLI do Plankton disponibiliza o comando **plankton query** para que o usuário consulte o estado de um *container* a partir do seu identificador. O quadro 3.13 mostra um exemplo de consulta pelo estado de um *container* chamado **my-container**.

O comando `plankton query` não aceita qualquer *flag* ou opção e o único parâmetro obrigatório é o nome do *container*. Mais informações podem ser vistas com o comando `plankton help query`

```
$ sudo target/debug/plankton query my-container
{
  "bundle": "/opt/containers/mycontainer/",
  "id": "my-container",
```

```
  "pid": 2517,
  "status": "Stopped"
}
```

Código 3.13 – Exemplo do comando plankton query

Quando a execução de um *container* se encerra sem erros, o arquivo permanece salvo para que seja possível executá-lo novamente. Neste caso, o campo status é atualizado para stopped, e o valor do campo pid é removido, uma vez que o processo de inicialização do *container* terminou.

Só é possível iniciar um *container* que foi criado mas não executou ainda (status "creating") ou que já terminou sua execução (status "stopped"). Caso o usuário tente iniciar um *container* que possui um status diferente um erro é retornado.

4 CONSIDERAÇÕES FINAIS

O desenvolvimento do presente estudo possibilitou uma análise das ferramentas que, quando usadas em conjunto, permitem que um processo Linux (ou ainda um grupo de processos) seja executado de maneira isolada e controlada. Durante seu tempo de vida, esses processos possuem uma visão particular do sistema, tendo acesso limitado a uma série de recursos, como CPU, memória, arquivos e diretórios e até mesmo chamadas de sistema. Embora o termo container seja usado para representar este ambiente controlado de execução, um container não é um objeto do sistema, sendo, na verdade, um conjunto de configurações e parâmetros que foram aplicados no momento da criação do processo. O texto incluiu uma breve análise sobre a especificação para container runtimes criada pela OCI, que determina que esses parâmetros estejam definidos em um arquivo do tipo JSON, que deve ser lido e interpretado pelo runtime.

A pesquisa incluiu o desenvolvimento de uma aplicação de linha de comando, que recebeu o nome de Plankton. Uma parte da especificação para containers Linux foi implementada neste projeto, ficando o restante a cargo de trabalhos futuros. O uso da aplicação desenvolvida ajudou a ilustrar a relação entre as diferentes ferramentas e a especificação. Foi possível verificar como mudanças no arquivo de configuração modificam a maneira que um container é criado.

A pesquisa para o desenvolvimento do trabalho permitiu aprofundar um tema pouco explorado na graduação. Seu objetivo foi mostrar que é possível executar um processo de forma controlada sem a necessidade de um hypervisor, como é o caso em máquinas virtuais, ou mesmo qualquer módulo que não esteja presente em uma instalação padrão do Linux. Com isso é possível replicar o mesmo programa em máquinas diferentes, criando, em cada uma, ambientes idênticos para sua execução. Isso permite ao administrador de um sistema ter cópias da mesma aplicação em várias máquinas diferentes, o que pode melhorar a performance e aumentar sua disponibilidade. O uso de containers permite que aplicações com dependências diferentes sejam executadas na mesma máquina, sem a necessidade da sua instalação no sistema hospedeiro. Isso evita problemas com aplicações diferentes que dependem de versões distintas da mesma biblioteca, por exemplo.

Como trabalho futuro é necessário foco maior na forma de uso de ferramentas importantes que foram abordadas de forma superficial nesta pesquisa, como control groups. Algumas preocupações de segurança são necessárias, e grande parte dos container runtimes do mercado fazem uso de ferramentas como o SELinux e seccomp, que não foram abordados na pesquisa. No projeto de código existem vários pontos de evolução, principalmente para a implementação de toda a especificação da OCI para sistemas Linux. Atualmente apenas alguns recursos são suportados, como montagem de arquivos e diretórios, namespaces e Rlimits. Novos comandos devem ser adicionados, como a remoção

de um container e o envio de sinais para os processos em execução dentro dele. Conforme mostrado em 2.4, é preciso evoluir a forma que o rootfs é montado no container, uma vez que a chamada usada atualmente não fornece o isolamento necessário para a execução segura do container.

REFERÊNCIAS

ARPACI-DUSSEAU, A. C. A.-D. R. H. **Operating Systems: Three Easy Pieces**. 0.91. ed. [S.l.]: Arpaci-Dusseau Books, 2015.

BACH, M. J. **The Design of the UNIX Operating System**. 1st. ed. [S.l.]: Prentice Hall, 1986.

BHATTIPROLU ERIC W. BIEDERMAN, S. H.-D. L. S. Virtual servers and checkpoint/restart in mainstream linux. **ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel**, v. 42, Julho 2008.

BIEDERMAN, E. W. Multiple instances of the global linux namespaces. **Proceedings of the Linux Symposium**, v. 1, Julho 2006.

BORATE, R. K. C. I. Sandboxing in linux: From smartphone to cloud. **International Journal of Computer Applications (0975 - 8887)**, v. 48, Agosto 2016.

EMELIANOV DENIS LUNEV, K. K. P. Resource management: Beancounters. **Proceedings of the Linux Symposium**, v. 1, Junho 2007.

HAYDEN, M. Securing linux containers. Julho 2015.

IEEE. Ieee standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7. **IEEE Std. 1003.1-2017**, 2017.

ISHKOV, N. A complete guide to linux process scheduling. Fevereiro 2015.

KERRISK, M. **The Linux Programming Interface**. 1st. ed. [S.l.]: No Starch Press, 2010.

LOOSEMORE RICHARD M. STALLMAN, R. M.-A. O. S.; DREPPER, U. **The GNU C Library Reference Manual**. 2018.

LOVE, R. **Linux Kernel Development: A thorough guide to the design and implementation of the Linux kernel**. 3rd. ed. [S.l.]: Addison-Wesley Professional, 2010.

MENAGE, P. B. Adding generic process containers to the linux kernel. **Proceedings of the Linux Symposium**, v. 1, Junho 2007.

NICK CAILLIAU ENIAS, D. G. L.-N. L. A. A comparative study on containers and related technologies. Novembro 2016.

OCI. **Open Container Initiative Runtime Specification**. 2018.

PIKE DAVE PRESOTTO, K. T. H. T. P. W. R. The use of name spaces in plan 9. **ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel**, v. 27, Abril 1993.

RANDAL, A. The ideal versus the real: Revisiting the history of virtual machines and containers. Abril 2019.

ROSEN, R. **Linux Kernel Networking Implementation and Theory**. 1. ed. [S.l.]: Apress, 2014.

RUSLING, D. A. **The Linux Kernel**. 0.8-3. ed. [S.l.]: Linux Documentation Project, 1999.

STONES, N. M. R. **Beginning Linux Programming**. 4th. ed. [S.l.]: Wiley Publishing, Inc., 2008.

TANENBAUM, H. B. A. S. **Modern Operating Systems**. 4th. ed. [S.l.]: Prentice Hall, 2014.

WARD, B. **How Linux Works: What Every Superuser Should Know**. 2nd. ed. [S.l.]: No Starch Press, 2015.