Trabalho de Sistemas Operacionais

Módulo de prioridade do serviço Docker no kernel do Linux

Arthur Gomes

1. Introdução

Unix é talvez o mais importante sistema operacional já criado. Isso porque ele introduziu conceitos fundamentais como multitarefa e multiusuário, que estão presentes em praticamente todos os SOs modernos. Ele abriu o caminho para iniciativas como a do Linux, que é um sistema livre, de código aberto e mantido por uma comunidade ativa e engajada. Tudo isso contribuiu para que os sistemas *Unix-like* se tornassem referência em estabilidade, segurança e alta performance. Por isso, eles são amplamente utilizados em aplicações de grande importância científica e comercial.

No campo da pesquisa em computação distribuída, é crescente a adoção de paralelização para controle de hardware. Entretanto, é imprescindível analisar uma alternativa viável a esta tecnologia.

Nas grandes corporações, por sua vez, técnicas de *conteinerização* têm sido utilizadas em larga escala. Os *containers*, que são ambientes isolados hospedados em um SO, podem ser escaláveis de forma dinâmica de acordo com a demanda, gerando melhor economia e realocação de recursos não utilizados.

Na via da modularização dos serviços e da distribuição da computação através de plataformas *cloud*, é necessário que haja uma prioridade na execução de processos vitais para fornecimento do produto.

2. Proposta

Diante do apresentado, a proposta de trabalho é a implementação de um módulo de prioridade do serviço *docker* no *kernel* do *Linux*, visando melhorar o desempenho da plataforma *Docker*.

2.1. Kernel e módulos

O *kernel* do sistema operacional é o responsável por controlar os dispositivos e demais periféricos do sistema como: memória, placas de som, discos rígidos e outros recursos. Para que esse controle seja possível é necessário que o *kernel* ofereça suporte para o dispositivo que está sendo instalado. O tamanho que um *kernel* ficaria para torná-lo capaz de suportar todos dispositivos e periféricos existentes o tornaria inviável. Diante desse cenário, a solução encontrada foi a utilização de módulos.

Módulos são códigos que atuam junto ao *kernel* e que são carregados na memória a medida que são solicitados manualmente ou por algum aplicativo ou dispositivo. Quando não são mais necessários, os módulos são descarregados. Essa construção evita a criação de um *kernel* original muito grande, onde boa parte das funcionalidades seriam pouco utilizadas, e permite a criação de um *kernel* mais simples e eficiente onde as partes mais complexas são alocadas dinamicamente.

2.2. Prioridade de processos

Quando um processo é criado, ele recebe uma valor de prioridade que no Linux varia entre -20 e 19. Essa escala trabalha de forma inversamente proporcional, quanto menor o valor maior a prioridade do processo.

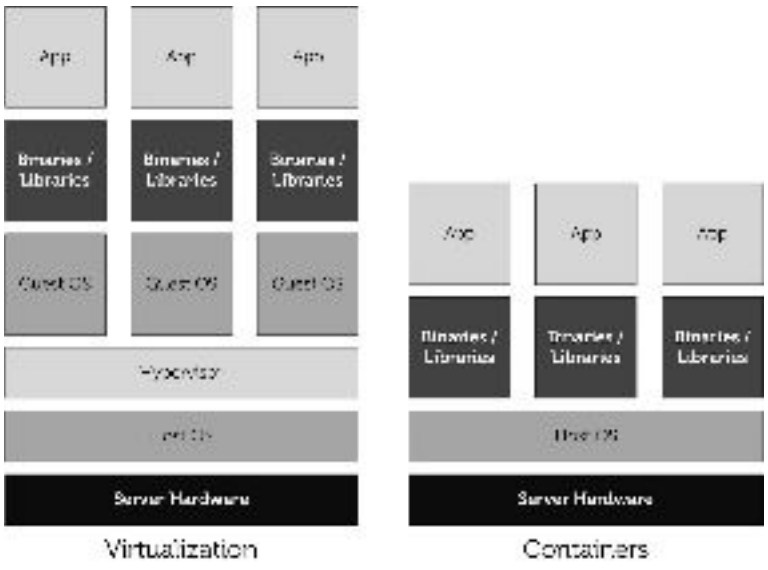
Quando o processo chega ao escalonador o nível de prioridade do processo que será executado é consultado. Esse valor afeta o tempo que a tarefa terá para ser executada e a ordem em que ela será executada. Por medida de segurança, somente superusuários podem atribuir valores negativos para a prioridade de um processo.

2.3. *Docker*

O *Docker* é uma plataforma de virtualização *Open Source* escrita na linguagem *GO* e mantida pela *Docker Inc*. Ela permite a criação e o gerenciamento de ambientes isolados hospedados em um sistema operacional.

Ao contrário de um ambiente de virtualização tradicional, onde se tem sistemas operacionais completos e isolados, dentro do *Docker* existem recursos isolados que utilizam bibliotecas de um *kernel* em comum, resultando em ganho de desempenho.

A estrutura da plataforma é baseada em *containers* que são executados em um *host*. Eles comportam desde uma aplicação até um ambiente inteiro e podem ser facilmente portados para qualquer outro host que tenha o *Docker* instalado.



Essa portabilidade é essencial no mundo distribuído, pois torna extremamente simples e fácil a replicação de ambientes idênticos, aumentando a disponibilidade do sistema, desde que aplicadas políticas de compartilhamento de informações de forma devida.

2.4. Quais vantagens pode trazer?

Dentro da plataforma *Docker,* é possível transformar um *container* em um arquivo de imagem, que pode ser distribuído e hospedado remota ou localmente. O nível global de arquivamentos é chamado de *Docker Hub*. Muitas vezes, imagens grandes (maiores que 20GB) não são utilizadas devido a demora da compactação, algo que pode de certa forma ser acelerado através de uma prioridade maior do serviço. Ao implementar uma mudança como essa, é esperado que os serviços baseados em *Docker* consigam realizar *backups* e recuperação de grande volume de dados de forma mais rápida e eficiente, principalmente em horários de menor utilização do sistema. Com isso, é possível aumentar a confiabilidade dos sistemas distribuídos uma vez que seria sabido que os dados armazenados nos arquivos de *backup* estarão sempre atualizados.

É importante ressaltar que a alteração não pretende modificar o escalonamento dos serviços executados dentro de um *container*, e sim acelerar e otimizar os comandos da própria plataforma, gerando e clonando contêineres de forma mais rápida.

Ao implementar em nível tão baixo a mudança, se torna fácil a propagação da mesma, já que a imagem base dos servidores e *desktops* já pode vir com o módulo habilitado.

3. Implementação

A implementação pode ser dividida em duas partes: identificar os processos de interesse e modificar os valores de prioridades.

Um processo dentro do kernel é representado por uma grande estrutura que tem o nome de *task_struct.* Essa estrutura contém os dados necessário para representar o processo - como nome e identificador do processo - e para manter os relacionamentos com outros processos.

Dentro dessa estrutura existe uma campo do tipo *struct list_head* com o nome de *tasks*. Este campo fornece um ponteiro para a tarefa anterior e outro ponteiro para a próxima tarefa, transformando o campo *tasks* em uma lista duplamente encadeada e circular.

Na maioria dos casos os processos são criados dinamicamente e passam a ser representados por uma estrutura *task_struct* que também é alocada de forma dinâmica. Uma das exceções é o processo *init*, o primeiro processo a ser executado, que é o responsável por carregar os outros processos durante a inicialização. Ele é alocado de forma estática e representado como *init_task,* o que torna possível a utilização dele como referência para interagir com o lista de processos.

Após a busca pelo processo do *Docker*, a alteração da prioridade pode ser feita com as funções *nice* ou *renice*. A função *nice* ,com o escopo nice -n [nice value] [command]*,* recebe o valor da prioridade para um processo que ainda não foi iniciado, podendo ser utilizada nos casos onde não for encontrado nenhum processo de nome *Docker.* Já a função *renice,* com o renice [nice value] -p [process id], altera a prioridade de um processo que já está sendo executado, o que torna ela útil para diminuir a prioridade de outros processo e aumentar, caso já esteja em execução, o processo *Docker.*

Algo interessante a ser implementado em conjunto ao programa principal é um *switch* de quando o processo deve atuar com prioridade máxima, já que nem sempre o mesmo será utilizado de forma crítica. Ao controlar este quesito, além de permitir o funcionamento normal do escalonador do sistema, é permitido também que o servidor rode com maior estabilidade e consiga exercer sua função primária de forma mais fluida.

3.1. Versão primária (0.1)

Analisando a necessidade de intervenção humana no processo, foi feita uma versão de testes em que a mudança pode ser realizada através de comandos a nível do usuário, manualmente indicando a necessidade de modificação da prioridade do processo. O código

fonte de comandos para nível usuário está disponível para *download* no link https://github.com/tutaarthur/trabalho-so/tree/master/c_simple.

O programa foi escrito em C, linguagem nativa do kernel e já é um passo em direção à implementação em módulo do mesmo. Para executá-lo, é necessário informar a prioridade que se deve colocar no processo do *Docker*. É recomendável que se varie entre prioridade máxima e normal, já que uma prioridade muito baixa pode gerar travamentos e lentidão na plataforma. Para isso, é recebido um argumento que tem três variações: *max, nor e min*, indicando o nível de *niceness* que deve ser adotado.

O programa então analisa os processos que estão executando e procura pelo ID do *Docker*. Logo que encontra, inicia o processo de mudança, montando uma *string* a ser passada como comando para o sistema, contendo todas as informações necessárias para a alteração da prioridade. Importante mencionar que o processo já deve estar rodando, uma vez que só é possível enviar comandos para a plataforma caso o serviço principal esteja sendo executado.

Por exemplo: ao listar os contêineres disponíveis no dispositivo local, o comando *docker ps* retorna uma tabela contendo todas as informações relevantes sobre cada um. Caso o serviço principal não esteja rodando, uma aviso parecido com com a imagem 3.1.1 é mostrado.

Imagem 3.1.1 - *Cannot connect to the Docker_daemon at unix:///var/run/docker.sock. Is the docker daemon running?*

E caso o serviço principal esteja rodando, a saída pode se parecer com a demonstrada na imagem 3.1.2.

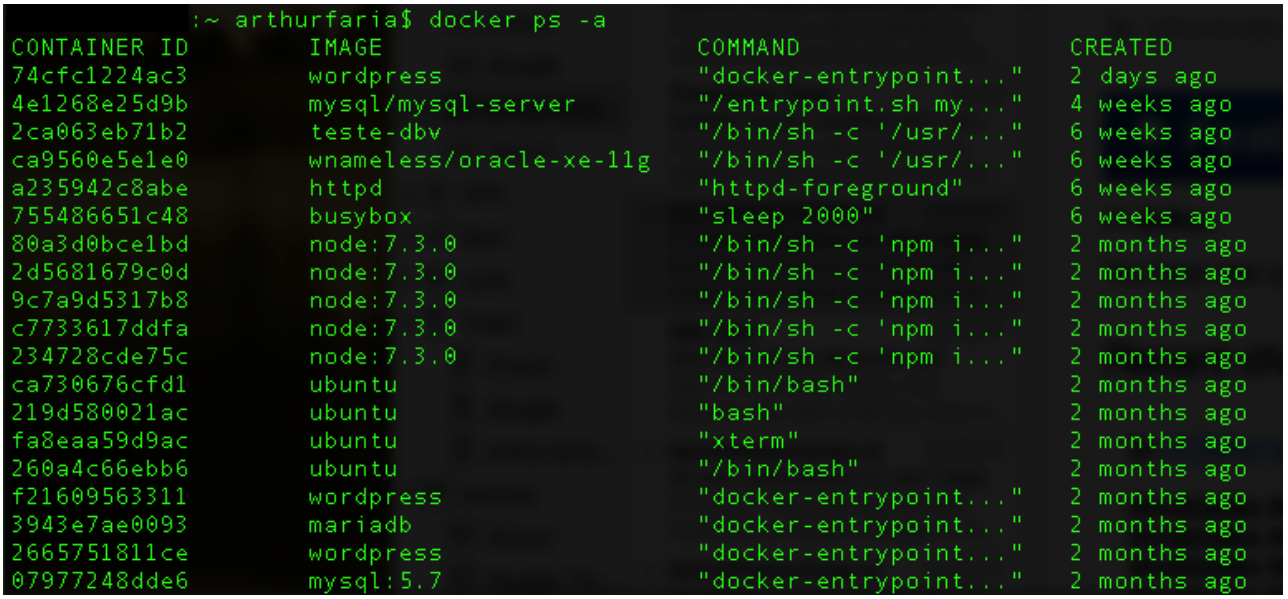


Imagem 3.1.2 - *Lista de contêineres salvos na máquina local*

Futuramente serão feitos testes utilizando esse programa a fim de analisar a viabilidade de implementar tal rotina internamente no kernel através de um módulo.

4. Testes

A fim de garantir a replicabilidade dos testes e a análise geral da performance com e sem a modificação da prioridade, serão utilizadas máquinas virtuais rodadas diretamente da mídia de instalação do sistema Debian 9.2 Live (https://www.debian.org/). Por ser considerado leve pela comunidade e utilizar uma interface gráfica que não requer muitos recursos, é ideal por permitir análises interativas e ao mesmo tempo não consumir tanto poder de processamento.

Serão executadas máquinas através do programa Virtual Box 5.2.0 (https://www.virtualbox.org/) sem a adição de discos rígidos, executando uma interface totalmente ao vivo. Para isso, a máquina host conta com 32GB de memória RAM e é equipada com 512GB de Solid State Drive para minimizar os impactos da paginação do sistema e maximizar a velocidade de execução.

Serão criadas instâncias do banco de dados relacional MySQL 8.0.3 através da imagem pública disponível em https://hub.docker.com/_/mysql/ e preenchido com tipos variáveis de dados até atingir um tamanho superior a 2 GB. Os tamanhos então serão aumentados até 15GB e os tempos cronometrados de compressão do container serão analisados com todas as prioridades. Exemplo: um contêiner com 8 GB de dados será

submetido a compressão pela plataforma no mínimo três vezes, uma para cada prioridade setada.

Os volumes de dados utilizados serão: 2, 5, 8, 12 e 15 GB, cada um então submetido a três testes. Para criar grandes quantidades de dados a ser importados, será utilizado o script Generate Data versão 3.2.8, disponível no GitHub através do link https://github.com/benkeen/generatedata, que permite gerar de forma aleatória os dados necessários para os testes. Para medir o tempo de execução do comando, será utilizado o *time*, comando que realiza as medições e mostra os resultados.

5. Resultados

Para cada tamanho de imagem, como descrito na seção 4, foram feitos três testes dependentes do comando *nice*. Os resultados são exibidos na tabela 5.1.1, em que o tempo é registrado da forma xxmyyyyys, onde x representa os minutos e y os segundos.

Tempo de commit (segundos)					
	2GB	5GB	8GB	12GB	15GB
nice 19	0m18,445s	0m38,470s	1m1,209s	1m35,758s	1m54,235s
	0m15,396s	0m40,890s	1m0,827s	1m33,423s	1m57,176s
	0m18,135s	0m37,763s	1m1,857s	1m35,295s	2m2,725s
nice 0	0m18,674s	0m40,675s	1m2,541s	1m31,770s	1m55,182s
	0m15,130s	0m38,411s	1m2,092s	1m32,774s	1m55,044s
	0m17,771s	0m41,259s	1m2,159s	1m35,736s	1m53,711s
nice -20	0m18,304s	0m43,038s	1m1,124s	1m39,047s	1m56,553s
	0m14,871s	0m45,099s	1m1,596s	1m37,806s	2m1,924s
	0m18,155s	0m40,979s	1m5,391s	1m33,122s	2m1,208s

Tabela 5.1.1 -Tempo de commit (segundos) - Servidor sem carga

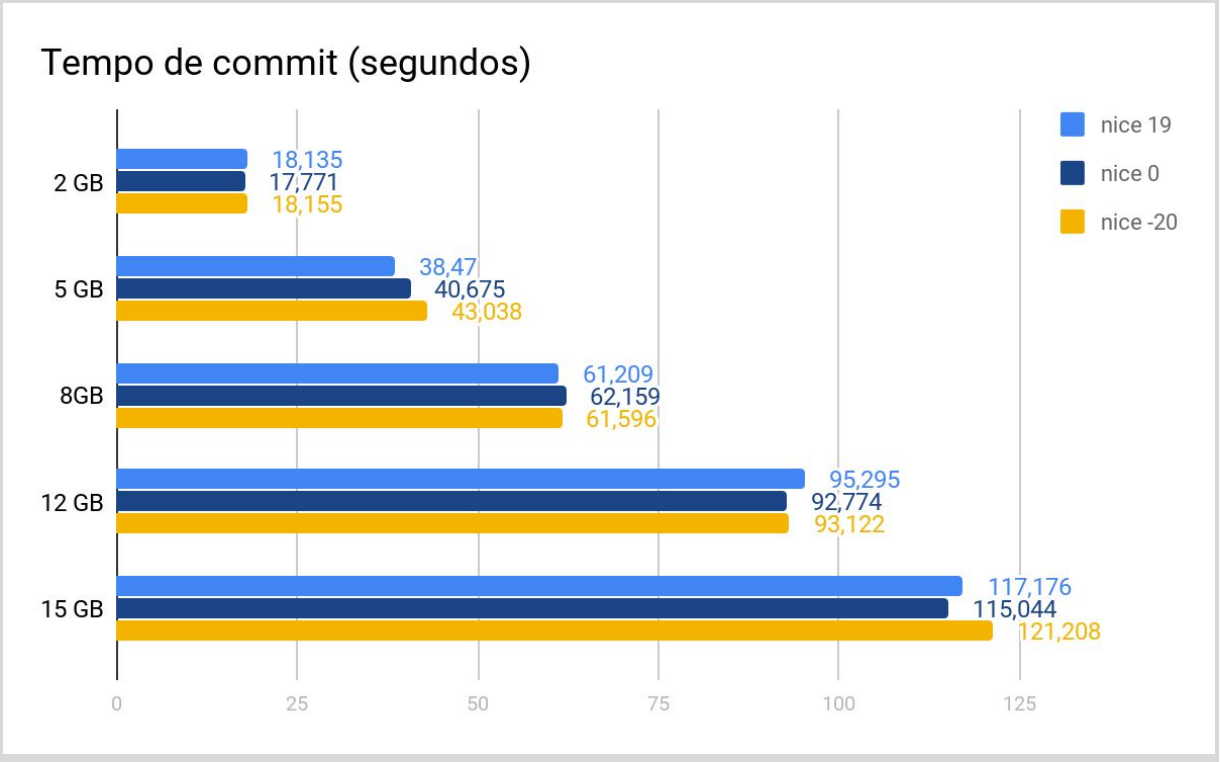


Gráfico 5.1.2 -Tempo de commit (segundos) - Servidor sem carga

Ao analisar os resultados, é possível perceber a proximidade do tempo entre os diferentes comandos e os diferentes níveis de *nice*. A hipótese levantada foi a seguinte - devido ao fato de todo o computador está dedicado aos testes, diferenças de prioridade não terão um impacto significativo, já que não há outro processo de tal importância concorrendo no escalonador. Levando isso em consideração, foi realizado outro teste, desta vez utilizando em paralelo um programa chamado *stress* (http://people.seas.harvard.edu/~apw/stress/) que realiza carga nos processadores, aproximando um pouco de uma situação real de usabilidade, onde o servidor ativo não usaria somente o comando *docker commit* e estaria sujeito à carga operacional de rotina.

Gerando carga nos 8 processadores disponíveis através do comando *stress --cpu 8,* foi coletada outra série de dados, disponíveis na tabela abaixo:

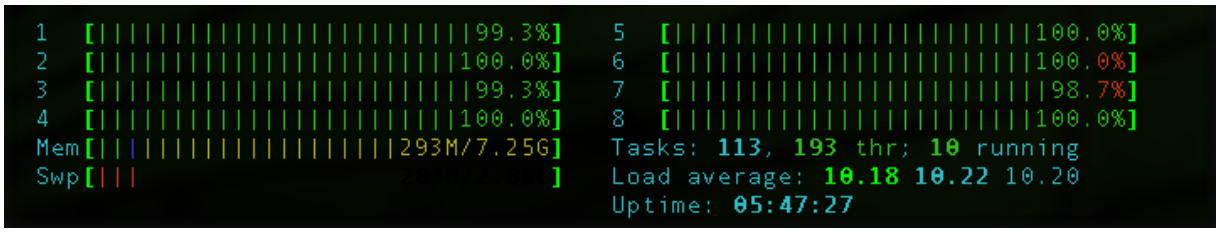


Imagem 5.1.3 - *Situação dos processadores - programa htop*

Tempo de commit (segundos)					
STRESSED	2GB	5GB	8GB	12GB	15GB
nice 19	0m27,903s	1m21,612s	2m3,633s	3m8,959s	3m52,088s
	0m30,105s	1m16,218s	2m2,592s	3m3,929s	3m50,584s
	0m30,036s	1m18,675s	2m3,172s	3m0,339s	3m47,881s
nice 0	0m30,127s	1m15,264s	2m1,617s	2m58,664s	3m55,137s
	0m27,820s	1m19,088s	2m5,532s	3m8,224s	3m51,508s
	0m29,331s	1m16,090s	2m6,977s	3m9,903s	3m48,627s
nice -20	0m28,608s	1m14,036s	2m1,858s	3m3,894s	3m52,794s
	0m30,169s	1m13,460s	2m2,442s	3m6,217s	3m54,229s
	0m27,772s	1m14,037s	2m3,240s	3m0,498s	3m54,692s

Tabela 5.1.4 -Tempo de commit (segundos) - Servidor com carga

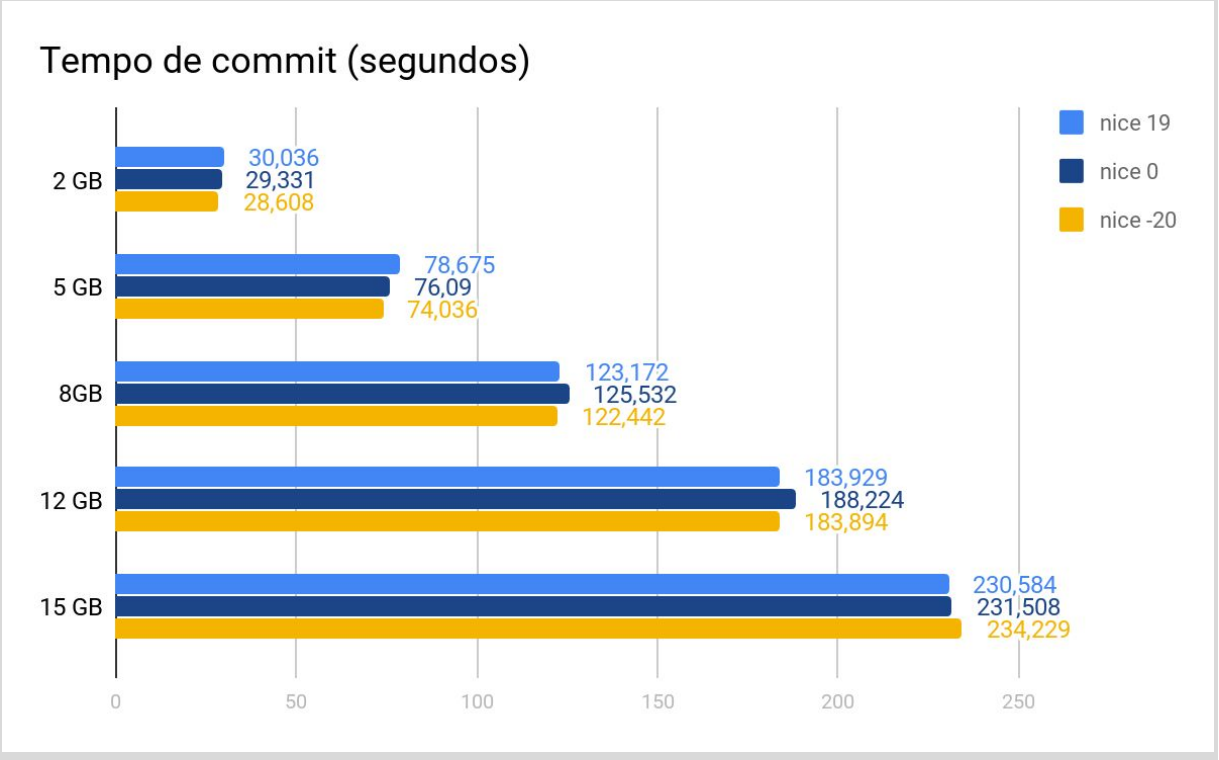


Gráfico 5.1.5 -Tempo de commit (segundos) - Servidor com carga

Analisando tanto o gráfico 5.1.5 quanto a tabela 5.1.4, é possível verificar uma pequena diferença entre os tempos resultantes da execução do comando em prioridade normal e máxima. O que não foi constatado durante a execução sem carga, ficou devidamente explícita quando carga foi aplicada ao computador, a diferença entre os diferentes níveis de prioridade. Embora em imagens maiores o resultado possa ter alguma relação à variáveis externas, em curtos períodos a mudança é clara.

6. Conclusão

Como visto, o trabalho apresenta a ferramenta de virtualização *Docker*, que permite a criação e o gerenciamento de ambientes isolados hospedados em um sistema operacional e apresenta uma forma para melhorar o seu desempenho por meio da utilização de um módulo de priorização do serviço.

Amplamente aplicável em escala de produção onde a disponibilidade de serviços é vital para o funcionamento total da plataforma e, o *backup* e recuperação de dados devem acontecer em tempo mais próximo do real, ao diminuir o tempo nas duas últimas operações, mesmo segundos a menos pode resultar em uma melhora significativa da performance geral do sistema.

Tentando atingir a máxima eficiência ao abordar uma mudança na prioridade do serviço *docker* no *linux*, a proposta, através de vários testes e medições, mostrou-se válida já que houve diferença entre os tempos de *commit*. Ao testar o caso extremo em que um servidor tem sua carga elevada ao máximo, a distância temporal de execução dos comandos se mostrou insignificante. Como eram esperados resultados mais destoantes entre os níveis de prioridade no caso crítico, foi concluído então que a diferença ínfima que a adoção de tal recurso possa trazer pode ser tornada irrelevante simplesmente ao realizar o *backup* dos servidores em tempos de menor carga.

6. Referências Bibliográficas

CORRÊIA, Iago. O Docker e seus containers: a nova era da virtualização!. Disponível em: http://coral.ufsm.br/pet-si/index.php/o-docker-e-seus-containers-a-nova-era-da-virtualizacao/

Dock Inc, What is Docker. Disponível em: https://www.docker.com/what-docker

DA SILVA, Gleydson Mazioli. Guia Foca GNU/Linux. Disponível em: http://www.guiafoca.org/?page_id=14

DE MEIRA, Marcos Vinícius. Política de Escalonamentos de Processos em Linux. Disponível em: http://revistas.bvs-vet.org.br/campodigital/article/view/30958

DIEDRICH, Cristiano. O que é Docker. Disponível em: https://www.mundodocker.com.br/o-que-e-docker/

JONES, M. Tim. Anatomia do Gerenciamento de Processos Linux. Disponível em: https://www.ibm.com/developerworks/br/library/l-linux-process-management/index.html

Changing the default scheduler. Disponível em: https://lists.kernelnewbies.org/pipermail/kernelnewbies/2016-May/016332.html

https://www.debian.org/

https://www.virtualbox.org/wiki/Downloads

https://hub.docker.com/_/mysql/