Aprendendo mais sobre o Linux

Cláudio Torres Júnior

12/11/2018

Resumo

Muito trabalho e diversos empecilhos como falta de recursos computacionais tentaram barrar o surgimento do sistema *UNIX*. Mas os programadores por trás desse projeto não deixaram isso atrapalhar o desenvolvimento do sistema. Diversas outras tecnologias foram criadas no decorrer do seu surgimento, como interpretadores de comandos, o *Shell*, que faz a ponte entre usuário e o sistema operacional. Vários anos se passaram, e cada vez mais novos modelos foram criados e aperfeiçoados, tornando a vida dos usuários e a conversa homem/máquina cada vez mais fácil. Arquivos são a base do *UNIX*. Tudo é transformado em um no sistema. Desde *drivers* até os *softwares*. Tendo um sistema hierárquico de diretórios, sua manipulação é bem mais intuitiva e prática. Existem também diversos comandos para facilitar a manipulação de arquivos. Desde os mais simples, até comandos mais complexos que são combinados com outros para uma maior experiência do usuário. Será mostrado alguns usos desses comandos em contextos comuns do cotidiano, como verificar a evolução de matriculados em um curso, por exemplo.

1 Introdução

UNIX é um sistema operacional portável que suporta multiusuários e multitarefas. Suporta alterações tanto por linha de comando quanto pela interface gráfica, uma opção bem mais amigável e menos trabalhosa. Por isso trabalha em conjunto com um interpretador de comandos, usualmente chamado de *Shell*, que faz a ligação entre usuários e o *kernel* do sistema. Existem os que recebem textos na linha de comando, retornando respostas em formato textual, e aqueles que utilizam uma interface gráfica, que retornam tanto textos quanto imagens.

Tudo começou em 1965, quando um grupo de programadores, Ken Thompson, Denis Ritchie, Douglas Mcllroy e Peter Weiner, juntos com a *AT&T* (companhia americana de telecomunicações), *GE (General Electric)* e do *MIT* (Instituto de Tecnologia de Massachusetts) se uniram para a produção de um sistema operacional chamado *Multics*.

Os recursos computacionais disponíveis na época eram insuficientes para a finalização do projeto, por ter conceitos bem avançados para a época. Então, em 1969, Ken Thompson reescreveu o *Multics* num conceito menos ambicioso, que fez com que pudesse ser terminado. Mais tarde, o projeto foi rebatizado para *UNIX* por Brian Kernighan.

Em 1973 o projeto foi novamente reescrito pelos programadores Dennis Ritchie e Ken Thompson, mas agora utilizando a linguagem C. A partir daí, ainda nos anos 70, começaram a ser desenvolvidas as primeiras distribuições de grande dimensão do sistema operacional, e foi em 1977 que a *AT&T* começou a fornecer o *UNIX* para instituições comerciais.

Com a crescente oferta de microcomputadores, diversas empresas buscaram o *UNIX* para suas máquinas. Devido à disponibilidade e à sua simplicidade, muitos fabricantes alteravam o sistema e geravam sistemas personalizados.

Em 1977/1981, *AT&T* criou várias variantes do primeiro *UNIX* comercial, chamado de *System III*. Em 1983, após várias melhorias no sistema, a *AT&T* apresentava o novo *UNIX* comercial, o *System V*. Hoje, esse sistema é reconhecido como padrão internacional. Ainda neste ano, pensando em um sistema operativo de *software* livre, Richard Stallman começou o projeto *GNU* tendo como base o *UNIX*. No final da década de 80, Andrew S. Tanenbaum criou um sistema para uso acadêmico, o *MINIX*, sendo mais barato e simples que o *UNIX*.

Em 1991, Linus Torvalds começou um projeto de um sistema operacional tendo como base o *MINIX*. De acordo com [7], a intenção de Torvalds era a de fazer o projeto rodar especificamente em sua máquina 80386, e por isso não esperava que seu projeto fosse crescer tanto e se tornar o *Linux*, sistema tão importante para o advento da computação e da tecnologia da informação. Passou então a ser incorporado em várias outras plataformas, o que contribuiu para o seu sucesso. O *Linux* é um *kernel*, por isso não tem muita utilidade sozinho. Por isso, para contornar esse problema, incorporou o software *GNU* (explicado anteriormente).

Por ser um *Kernel* disponibilizado de maneira gratuita e com o código-fonte aberto, qualquer pessoa ou empresa pode criar um sistema operacional customizado. Por isso há várias distribuições *Linux* no mercado.

Mas como o usuário se comunica com o sistema? De acordo com [23], para fazer essa comunicação foi desenvolvido o *Shell*, que é um interpretador de linhas de comando. Na próxima seção entraremos em mais detalhes sobre os diversos interpretadores existentes e algumas de suas particularidades, utilizando-se de exemplos que demonstram sua facilidade para a manipulação de arquivos.

2 Shell

A seguir, antes de ser explicado algumas particularidades do *Shell* e do sistema de arquivos, iremos ver um pouco de sua história.

2.1 História

Em 1971, Ken Thompson criou a primeira *Shell* para *UNIX*, chamada *Shell V6*, que era independente do usuário e executada fora do *kernel*. Em 1977 surgiu a *Bourne Shell* (sh), criada pelo Stephen Bourne para *UNIX V7*. Tinha como objetivo servir como interpretador de comandos. Introduziu fluxo de controle, *loops* e variáveis nos *scripts*, criando uma linguagem mais funcional para comunicação com o sistema operacional. Foi o início para diversas *Shells*.

Em 1978 foi desenvolvida a *C Shell* (csh) para *UNIX BSD* por Bill Joy. Dentre seus principais objetivos, era a criação de uma linguagem semelhante a C. Um recurso útil foi o histórico de comandos. Nessa mesma época foi projetada por David Korn a *Korn Shell* (ksh) que era compatível com a *Bourne Shell* original. Até 2000 era um *software* proprietário. A partir daí foi lançada como *software* livre.

Para substituir a *Bourne Shell*, foi criada a *Bourne-Again Shell* (bash), um projeto *GNU* de *software* livre. *Bash* foi desenvolvida por Brian Fox e se tornou um dos interpretadores de maior disponibilidade. Além de compatibilidade com versões antigas, incorporou recursos das *Korn e C Shells*. A seguir, temos uma tabela com os seus principais recursos e suas disponibilidades nas diferentes versões, retiradas de [1].

Característica	**sh**	**csh**	**bash**
Renomear comandos complexos com nomes simples	Não	Sim	Sim
Histórico de comandos	Não	Sim	Sim
Corrigir um nome de comando incorreto em um comando complicado usando as teclas de seta e o backspace	Não	Não	Sim
Nomes completos de arquivos longos com uma única tecla (tab)	Não	Sim	Sim
Alterar o prompt para exibir o diretório de trabalho atual	Não	Não	Sim
Pode lidar com grandes listas de argumentos	Sim	Não	Sim
Disponível gratis: baixe o shell e possivelmente o código fonte	Não	Não	Sim

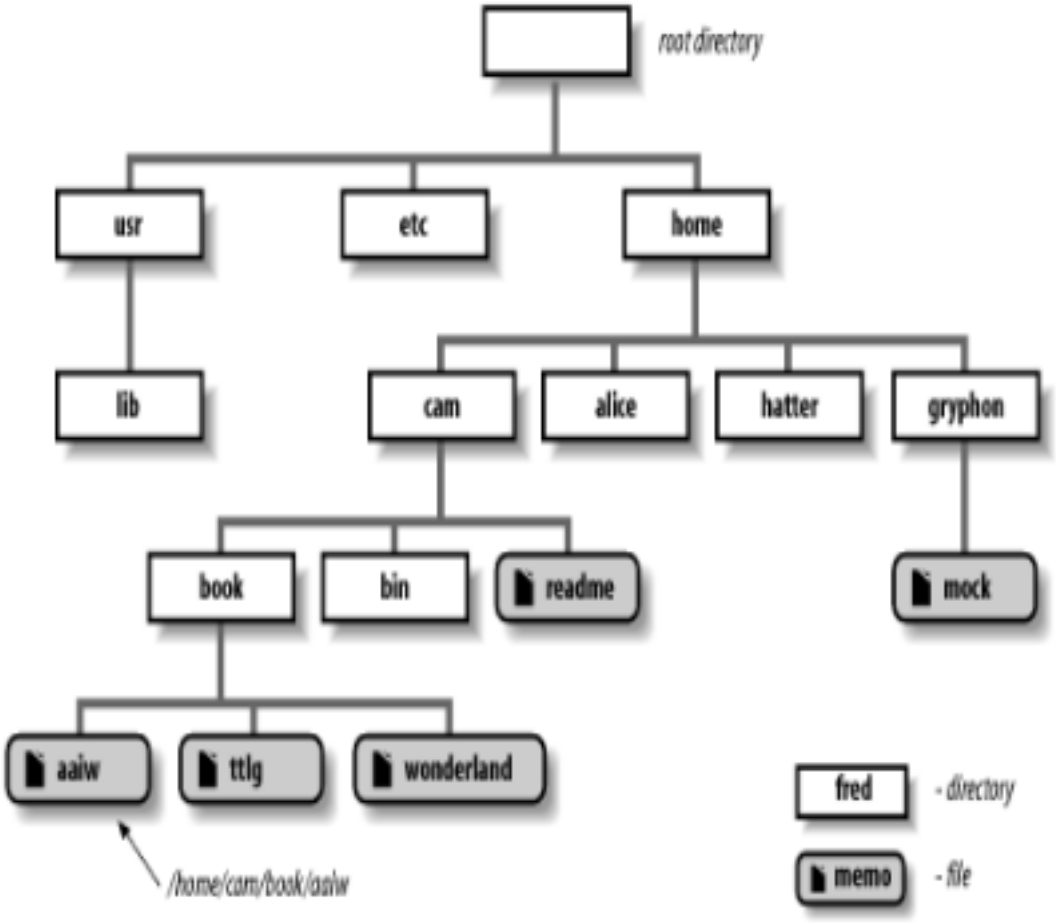
3 Sistema de Arquivos

Arquivos são super importantes em qualquer sistema *UNIX* e podem ter qualquer tipo de informação. Segundo [16] há três tipos importantes: Arquivos regulares, que podem ser chamados de arquivos de texto. Neles há caracteres que são possíveis de serem lidos pelos usuários; Arquivos executáveis, que são os programas. Alguns deles não podem ser compreendidos (lidos) pelas pessoas. Os *scripts* em *Shell* são um tipo especial de arquivos de texto que não são caracteres possíveis de serem lidos por nós e é um arquivo executável chamado *bash*; E por último mas não menos importante, temos os diretórios. São como pastas que podem ter outros tipos de arquivos nele ou até mesmo subdiretórios. A seguir, será posto em mais detalhe este último tipo.

3.1 Diretórios

Como visto no item anterior, diretórios são arquivos que podem "guardar" outros tipos de arquivos e até mesmo outros diretórios, os subdiretórios. Esse fato de poder ter outros do mesmo tipo, implica numa estrutura hierárquica. Abaixo temos uma representação da estrutura hierárquica de um diretório.

Figura 1: FONTE: [16]



O primeiro diretório é chamado de raiz. A partir dele, será distribuído todos os outros. Para referenciar um deles, segue uma linha desde a raiz, até o diretório que se quer informação, separando-os por barra (/). Por exemplo, temos um diretório chamado *home* que há um outro chamado *claudio* que possui o *IBM* que há *disciplinas*. Para acessar o diretório das disciplinas é só seguir o caminho */home/claudio/IBM/disciplinas*.

Para não precisar ficar sempre escrevendo o caminho desde a raiz até o que você quer, há uma grande facilidade em referenciar outro diretório a partir do seu atual. Se o que você quer "visitar" está dentro do seu atual, basta acessá-lo com o nome. Por exemplo estamos dentro do diretório *disciplinas* visto ante-

riormente (*/home/claudio/IBM/disciplinas*). Vamos supor que nele há várias disciplinas divididas em subdiretórios. Se quisermos acessar uma delas, como a *CI320*, não precisamos fazer o caminho novamente (*/home/claudio/IBM/disciplinas/CI320*), basta somente acessar pelo nome *CI320/*. (Não necessariamente precisa-se da barra (/) após o nome, o sistema se encarrega disso). Desde o início deste tópico estamos vendo sobre acessar diretórios. Mas como realmente podemos acessá-los via shell? Veremos a seguir alguns comandos que ajudam na navegação e manipulação de arquivos e diretórios.

4 Alguns comandos do Linux

Vários comandos nos ajudam a manipular arquivos. Seguindo o exemplo anterior onde vimos o caminho dos subdiretórios, como poderíamos acessar as *disciplinas*? Primeiramente temos que acessar um terminal. Pressione as teclas **ctrl+alt+t** e você terá um terminal.

Antes de ir para o diretório que queremos, temos que saber onde estamos. Para isso, basta digitar o comando **pwd** que o *Shell* irá te mostrar onde você está. Acabou de ser mostrado nosso caminho atual. Temos que acessar o */home/claudio/IBM/disciplinas*. Para essa finalidade, temos um comando que acessa o caminho desejado, o **cd**. Como já estamos no */home/claudio/*, basta digitar o complemento do caminho, que iremos até o que queremos. Mas vamos aos poucos para exemplificar melhor o que acontece. digitando **cd IBM** no terminal, iremos ser deslocados para o caminho */home/claudio/IBM*. Para saber se realmente estamos nesse caminho, basta realizar o comando **pwd** novamente. Agora temos que acessas as *disciplinas*. Simplesmente digitaríamos **cd disciplinas**, mas se você escrever **cd di** e apertar **tab**, o próprio sistema irá terminar a palavra pra você se somente existir um diretório com este nome, ou irá mostrar todas as possibilidades se houver mais de um com as primeiras letras sendo *di*. Pode-se também acessar diretamente a partir do */home/claudio/* escrevendo *IBM/disciplinas*. Pronto, estamos nas *disciplinas*. Antes de continuar, há várias particularidades no comando **cd**. Se você quiser voltar ao caminho anterior ao que você estava, basta realizar o comando **cd -**; para voltar ao diretório raiz, basta digitar **cd ...** Voltando ao exemplo, neste diretório há vários outros relacionados à algumas disciplinas. Se for digitado no terminal o comando **ls**, será listado todos os arquivos que está no teu caminho atual. Caso deseje ver mais detalhes, faça **ls -l** e será mostrado tamanho, hora da última modificação, e outras informações. E o **ls -a** irá mostrar os arquivos ocultos. E **ls (nome do diretório)** irá mostrar o que esta dentro do *(nome do diretório)*.

As vezes o comando **ls** não é o suficiente. Como vimos, existem vários códigos de algumas disciplinas. Para facilitar a procura por um determinado tipo de arquivo, mas que não nos lembramos como realmente é o nome, existem os coringas, que são caracteres que junto aos comandos desempenham funções que ajudam a encontrar padrões.

4.1 Coringas

Vamos supor que não saibamos qual o último número do código que estamos querendo achar. Se executarmos somente o **ls**, teremos muitos arquivos para procurar. Mas ao realizar o comando **ls CI32?** será mostrado todos os arquivos que começam com *CI32*.

O coringa **?** não funciona caso após ele tenha mais que um caractere. Se você não soubesse os números do código e tentasse **ls CI?** não iria funcionar. Para esses casos basta fazer **ls CI*** que irá mostrar somente resultados que começam por CI e terminam com zero ou mais caracteres. Caso saiba somente os números e queira ver todos que possuam o mesmo número do código, é só fazer **ls *320**.

É possível também pegar todos os códigos que começam com letras entre A e C. Basta fazer **ls [ABC]*** ou **ls [A-C]***. Funciona também com algarismos. **ls *[0-2]** mostra os códigos que tenham 0, 1 ou 2 neles.

Caso o comando tenha **!** precedendo os caracteres, implica em uma negação. Então **ls *[!0-2]** não irá mostrar nenhum código que tenha números entre 0 e 2.

O uso de coringas são considerados expansões de caminho, pois somente um * pode expandir todo o caminho existente no diretório. Existem outros tipos de expansão, como as de colchetes e a de chaves.

4.1.1 Expansão de colchetes

Esse é um conceito próximo à expansão anterior. Enquanto as realizadas pelos coringas, as de colchetes expandem à uma *string* arbitrária. É formada simplesmente por um prefixo (opcional), *strings* separadas por vírgulas entre colchetes e um sufixo. Exemplo: **echo c{obr, ubr}a**, que irão formar *cobra e cubra*. O comando **echo** é usado para escrever *strings* no terminal. Esse tipo de expansão é útil para criação de vários arquivos que possuem o mesmo sufixo e prefixo. Por exemplo levando em conta nossos códigos de disciplinas o comando **mkdir CI{32, 33, 34}0** irá criar três novos diretórios no caminho atual: *CI320, CI330 e CI340*. O comando **mkdir** é usado para criar novos diretórios e o **touch (nome arquivo)** cria novos arquivos. Caso deseje remover diretórios ou arquivos, use o comando **rm**. Realizando o **rm (nome do arquivo)** exclui o *nome do arquivo*. Caso queira excluir um diretório, utiliza-se o comando **rm -rf (nome diretorio)**. Mas deve-se tomar cuidado com esse comando. O parâmetro **f** irá excluir todos os arquivos e subdiretórios que estiverem dentro do diretório a ser excluído. Caso não queira excluir os arquivos/diretórios, pode-se movê-los. Utilizando o comando **mv (origem) (destino)** você pode mover *origem* para *destino*. Se *destino* for um arquivo, você irá sobrescrevê-lo; se for diretório, o *origem* vai pra dentro dele; *origem* pode ser um conjunto de arquivos. Além desses comandos, existe também o **cp (origem) (destino)**, que copia o arquivo *origem* para o diretório *destino*.

Muitos comandos foram citados anteriormente e alguns parâmetros não foram ditos, o que nos leva a confusão. Para resolver isso, existe um comando

que funciona como um manual para os demais comandos, que é o **man**. Para utilizá-lo, você faz **man (nome comando)** e irá ser mostrado ao usuário o funcionamento do *nome comando* com todos os parâmetros que podem ser usados com ele.

5 Entrada e Saída (I/O - Input/Output)

Tudo no *UNIX* é tratado como arquivo, como é dito em [16]. Cada programa aceita uma entrada padrão (*STDIN*) que pode ser o teclado, a saída de outro programa ou até mesmo a leitura de um arquivo (os dois últimos exemplos utilizam-se de *pipeline*, que será abordado posteriormente), uma saída padrão (*STDOUT*) que é a tela do terminal e/ou gravado em um arquivo, e uma mensagem de erro padrão que avisam de possíveis problemas ocorridos. O *pipeline* é muito utilizado, pois a maioria dos programas realizam tarefas específicas, que em conjunto com outros programas formam um mais complexo.

Existem vários comandos que manipulam arquivos, como o **cat**. Para usá-lo, deve-se digitar como parâmetro o nome do arquivo que quer acessar, exemplo: **cat exemplo.c** irá mostrar na tela do terminal o conteúdo do *exemplo.c*. Se o argumento não for um arquivo, irá ser mostrado o próprio parâmetro no terminal.

Também há o **grep**, que procura um certo trecho (padrão) dentro de arquivos retornando onde foi encontrado e em qual linha se encontra. Seu uso é **grep "trecho a procurar" exemplo.txt** ou **grep "trecho a procurar" ***. No primeiro exemplo, o padrão é procurado no *exemplo.txt* e no segundo é buscado em todos os arquivos/diretórios do diretório atual.

Há vários outros comandos muito utilizados que serão listados a seguir. Para uma maior compreensão de sua utilização, basta usar o comando **man** que já vimos anteriormente.

sort	organiza as linhas do arquivo em ordem numérica ou alfabética
last	mostra as últimas N linhas de um arquivo
head	mostra as primeiras N linhas de um arquivo
cut	mostra apenas determinadas colunas da entrada na saída
tr	traduz a entrada padrão

5.1 Redirecionamento de entrada/saída

Muitos dos comandos citados não utilizam necessariamente o teclado como *input* padrão. Arquivos podem ser usados como tal. O comando **comando** < **nomearquivo** realiza esta tarefa. O *comando* recebe como entrada o *nomearquivo*. O contrário também pode ser realizado, fazendo **comando** > **nomearquivo**. Nesse caso a saída do *comando* é salva/utilizada no *nomearquivo*.

5.2 Pipelines

As entradas/saídas não são somente utilizadas de arquivos. Os *output* dos comandos servem como o *input* de outros. O que faz isso é o caractere |, o *pipe*. O *pipeline* é quando dois ou mais comandos são conectados na linha de comando do terminal pelo *pipe*. E para demonstrar sua grande utilidade, iremos seguir passo a passo vários exemplos de utilização tanto do *pipeline* quanto dos comandos citados anteriormente. Mas antes, será mostrado alguns atalhos do *Shell* para ajudar na manipulação do terminal e de telas secundárias, o que ajudará em um melhor aproveitamento.

6 Explorando atalhos do *Shell* e do *Linux*

Primeiramente abriremos um terminal. Nossos exemplos irão utilizá-lo, portanto essa etapa é fundamental. Para isso, basta apertar as teclas **ctrl** + **alt** + **t** ([25]). Há também o **alt** + **F2** ([11]), mas dessa forma irá se abrir uma janela que você deve escrever o nome do aplicativo a ser executado (serve tanto para o terminal quanto para qualquer outro aplicativo). Após aberto, podemos aumentar a fonte das letras, realizando o atalho **ctrl** + **shift** + **(+)** e para diminuir, **ctrl** + **(-)** (há também a possibilidade de voltar ao tamanho original fazendo **ctrl** + **0**).

Uma outra facilidade do terminal, é poder abrir várias outras abas. Para isso, faça **ctrl** + **shift** + **t**. Para alternar entre abas abertas, basta fazer **alt** + **(numero respectivo à aba)** (exemplo: **alt** + **1** alterna para a primeira aba). Para fechá-las, faça **ctrl** + **shift** + **w**.

Há a possibilidade de abrir um aplicativo com o terminal. Para isso, utiliza-se o comando **(nome programa)** + **&**. Quando desejar encerrar o programa, basta ir ao terminal novamente e executar o comando **ctrl** + **c**. Mas como mudar para a janela do terminal? Basta segurar a tecla **ctrl** e ir apertando a tecla **tab**. Com isso, você irá alternar entre todas as janelas abertas. Pode-se também minimizar a tela do aplicativo até chegar à janela do terminal com o atalho **alt** + **F9** [11] (para maximizar, **alt** + **F10**).

Quando usamos dois monitores, podemos espelhá-los e ter a mesma tela comum aos dois. Mas com isso temos um problema. Se quisermos que um programa seja mostrado somente em um deles, devemos tirar o espelhamento. Ao ligar os monitores, faça **windows** + **p** e irá aparecer opções para as telas. Escolha juntar telas. Com isso, terá duas áreas de trabalho diferentes. Para trocar um programa aberto de uma tela para outra, faça **windows** + **shift** + **(seta direcional para esquerda/direita)**. Com isso, somente uma das telas apresentará o programa.

Ao finalizar a execução de alguma tarefa, há a necessidade de fechá-la. Para isso, faça **alt** + **F4** ou **ctrl** + **shift** + **q** para encerrar a execução da janela aberta. Isso funciona para o terminal também, mas para ele há a possibilidade de digitar **exit** na linha de comando para fechá-lo. Para encerrar uma seção, **ctrl** + **alt** + **backspace** realiza o desejado.

Agora que sabemos abrir um terminal e executar alguns atalhos que facilita o seu uso, vamos ver algumas técnicas utilizadas para manipular arquivos.

7 Processos

Cada código executável desses que foram explicados anteriormente, necessita de recursos para executar, como área de memória para armazenar o código por exemplo. Esses recursos alocados à uma tarefa é denominado processo. [14]

Não entraremos em detalhes mais profundos sobre os processos, pois não é o escopo deste trabalho. Temos somente que saber que cada processo possui um proprietário, estado (executando, em espera, etc), prioridade de execução e recursos de memória [8]. Além de um número que o identifica, o **PID** (identificador do processo).

Como cada processo possui um proprietário, o sistema sabe quem pode o executar ou não. Por isso devemos mudar a permissão de nossos *scripts* com o comando **chmod** [3]. Para dar permissão para todos os usuários, faça o comando **chmod +x script.sh** ou **chmod 0755 script.sh**. Para só que o usuário atual (quem criou o script) o execute, faça **chmod u+x script.sh** ou **chmod 0700 script.sh**. Para visualizar as permissões de um *script* faça **ls -l script.sh**. Para mais informações sobre o comando que seta as permissões, é só realizar **man chmod**.

Quando é criado, não implica que ele será imediatamente executado. O *Linux* que irá saber qual a situação cada processo terá. O comando **ps** é muito usado para o gerenciamento de processos [8]. Com ele, pode-se saber quais processos estão em execução. É uma informação do momento, não irá ser atualizada. Existem também várias outras opções que combinadas com esse comando nos dá mais detalhes sobre processos. Para ter uma visão atualizada (visão mais real) do que está acontecendo, usamos o comando **top**, que mostrará informações atualizadas a cada 10 segundos (geralmente). Como o comando **ps**, o **top** também pode ser usado junto com outras opções.

Além desses comandos, podemos destruir (matar) processos. Usamos para isso o comando **kill** junto com um sinal (parâmetro). Vamos levar como exemplo que desejamos parar temporariamente com o processo de *PID 4496*. Para isso, fazemos **kill -STOP 4496**. (Nos nossos *scripts*, quando os executamos em um terminal, podemos interrompê-los com o atalho *ctrl+z*). Para retornar o processo à execução, fazemos **kill -CONT 4496**. Para parar temporariamente com todos os processos: **kill -STOP -1**. Para acabar (matar) um processo, usamos o **kill -9 PID**. O sinal utilizado é 9 porque não queremos que o processo ignore o **kill**. Vamos supor que tenha esquecido o *PID* do processo que queira "matar". Podemos realizar o processo com o próprio nome do processo: **killall -(sinal) (processo)**. Por exemplo, queremos "matar"o *Google Chrome*. Fazemos então: **killall -9 chrome**.

Com isso, terminamos com a apresentação do básico que se deve saber sobre manipulação de arquivos e processos. Como vimos mais a essência de cada comando, uma boa abordagem seria avançar em mais pesquisas e descobrir

outros comandos que se encaixem nas tarefas que iremos realizar.

8 Exemplos do uso do Shell e seus comandos para manipulação de arquivos

Todos os códigos mostrados serão explicados por partes. Ao final de cada exemplo, haverá uma subseção com o código respectivo completo. Todos foram salvos como um *script em Sehll* e foram executados em um terminal. Para salvar um *script*, deve-se dar o nome do arquivo com a extensão **.sh**. Exemplo: **meuscript.sh**. No início do arquivo, deve estar comentado qual *Shell* será utilizada. No nosso caso, será o *Bash Shell*: **#!/bin/bash**. Isso será comum para todos os códigos. Agora vamos começar.

9 Exemplo 1 - Dados dos patrimônios de um local

Temos em mãos um arquivo que possui os dados dos patrimônios de um determinado departamento. Cada item possui um código, nome do objeto e suas características, além de um código que indica onde está localizado. Cada linha do arquivo tem um separador, que é o ponto e vírgula (**;**) que separa cada característica do objeto e um delimitador que são aspas duplas (**"**) que possui as características. Por exemplo: **"12748";"CADEIRA GIRATORIA MADEIRA ESTOFADA";"ESTOFADA, COM RODIZIOS, SEM BRACOS SALA 03 ";;"2100.13.03.02 ";** Queremos dividi-lo em vários outros arquivos que serão nomeados de acordo com cada local que os objetos estão. Este item citado acima deverá ser colocado em um arquivo chamado **2100.13.03.02.csv**. Como isso será realizado? A seguir iremos ver algumas ferramentas que nos auxiliam nessa tarefa.

9.1 Etapa 1 - Verificando possíveis empecilhos

Antes de começar a divisão em vários outros arquivos, temos que verificar os separadores. Como dito anteriormente, cada linha possui o (**;**) como separador. Cada linha possui 5 campos de características, facilitando nosso trabalho. Mas e se houver algum ítem que possua o (**;**) sem ser o delimitador? Por exemplo: *"cadeira; plastico"*. Neste exemplo, o **;** está dentro do separador dos itens. Isto faria esta linha ter 6 **;** podendo nos confundir como tendo uma coluna a mais. Para isso, vamos utilizar o comando **grep** [24]. Com ele iremos localizar possíveis linhas fora da formatação desejada (5 colunas divididas por **;**).

Abrimos o terminal no diretório onde está nosso arquivo. Para não ter o perigo de estragar o original, vamos criar uma cópia. No nosso exemplo, nosso arquivo se chama *patrimonio.csv*. Realizando (**wc -l patrimonio.csv**) no terminal temos o número de linhas (2603). O comando **wc** de acordo com [13], é usado para determinar número de linhas, caracteres e palavras. Usamos

o comando (**cp patrimonio.csv patrimoniocp.csv**) e assim temos uma cópia chamada *patrimoniocp.csv*. Agora fazemos o comando:

```
grep -n "[a-zA-Z0-9 ];[a-zA-Z0-9 ]" patrimoniocp.csv
```

Vamos por partes: **grep** é um comando que procura um certo padrão (frase, palavras, ...) em um arquivo; **-n** é um parâmetro que mostra a linha em que está localizado o padrão; **"[a-zA-Z0-9];[a-zA-Z0-9]"** é o nosso padrão. Nele usamos o coringa **[a-zA-Z0-9]**, que procura frases ou palavras que tenham as letras de A a Z tanto maiúsculas e minúsculas, os algarismos de 0 a 9 e espaço em branco que estão antes e depois do **;** e entre as aspas duplas; **patrimoniocp.csv** é o nome do arquivo que estamos procurando tal ocorrência.

Temos como resultado a linha que possui **;** entre as palavras, e como esperávamos com o uso do **-i** temos as linhas que estão nosso problema. Agora usando o comando **sed** que serve para editar textos [10], iremos deletar essas linhas do nosso arquivo. Usamos o comando:

```
sed -i '175d;734d;735d' patrimoniocp.csv
```

O **-i** indica que o arquivo será modificado; **175d;734d;735d** significa deletar a linha 175, 734 e 735 (que foram as que apareceram no resultado do **grep**) ; e **patrimoniocp.csv** é o nome do arquivo; Para verificar se tudo ocorreu certo, comparamos o número de linhas novamente com o comando (**wc -l patrimoniocp.csv**) e vemos que possui 2600 linhas, 3 a menos que o original e o comando realizado anteriormente para encontrar as falhas para ter certeza de que foi essas linhas que sumiram. Agora que todas as linhas possuem somente 5 colunas cada, podemos remover as aspas duplas para "limpar"um pouco nossos itens. Para isso utilizaremos o comando de edição de texto novamente.

```
sed -i 's/"/ /g' patrimoniocp.csv
```

irá tirar todos os **"** e substituir por espaço em branco. O **s** indica substituição e o **g** é para que a substituição seja para todas encontradas no arquivo.

9.2 Etapa 2 - Criando um arquivo com os locais

Após tirar as linhas que estavam atrapalhando a formatação das outras, podemos realizar a segunda etapa, que consiste em criar um arquivo que contenha todos os códigos dos locais de forma única e ordenada. Sabendo que o *patrimoniocp.csv* tem 5 campos divididos por **;** e o código que queremos encontra-se no último campo, utilizaremos um método que corta todo o texto anterior à um determinado padrão que queremos. De acordo com [2], o comando utilizado para esse feito é o **cut**, que combinando os seus diversos parâmetros, conseguimos imprimir na saída padrão uma parte de uma *string* de um arquivo. Logo, fazemos:

```
cut -s -d";" -f5 patrimoniocp.csv
```

No documento feito por [2], vemos que o parâmetro **-s** faz com que não seja impressa nenhuma linha que não tenha o delimitador especificado; **-d** especifica o delimitador que procuramos (**;** no caso). Por isso o **-d";"**); **-f** indica o

campo a ser usado. No nosso exemplo queremos a 5ª coluna, por isso **-f5**; E **patrimoniocp.csv** é o nome do arquivo.

Vemos que isolamos os códigos dos locais. Os espaços em branco são as linhas que possuem a 5ª coluna em branco. Queremos passar todos esse locais para um arquivo separado, tirando as ocorrências repetidas e ordenando-os. Para isso utilizaremos o *pipeline*. Vamos usar a saída do **cut** como a entrada de um outro comando, o **sort** que é utilizado para ordenar linhas de arquivos. O **sort** com o parâmetro **-u** irá rearranjar as linhas tirando todos as ocorrências repetida em ordem crescente. Logo usamos a linha de comando:

```
cut -s -d';' -f5 patrimoniocp.csv | sort -u > locais.txt
```

Nesse comando utilizamos de um *pipeline*, em que os *output* dos comandos servem como o *input* de outros. A saída do **cut** é a entrada do **sort -u**. O sinal $>$ redireciona a saída do último comando para o arquivo **locais.txt**. Basta realizar (**cat locais.txt**) e será mostrado no terminal o nosso novo arquivo com os locais em ordem crescente e sem repetição.

9.3 Etapa 3 - Criando arquivos com os itens respectivos

Agora que já temos os locais, temos que criar para cada um deles um arquivo com o seu nome e salvar as linhas respectivas à esses códigos. Exemplo: O local 2100.13.01.02 terá um arquivo com seu nome com extensão *.csv* (2100.13.01.02.csv) que terá todas as linhas que contém o código "2100.13.01.02". Primeiramente vamos criar um diretório *locais* para não "sujar"o atual com vários outros arquivos. Para isso usamos o comando:

```
mkdir locais
```

Agora movemos o *locais.txt* e o *patrimonio.csv* para esse novo diretório com:

```
mv patrimonio.csv locais.txt locais
cd locais
```

Utilizaremos o **loop** do *Shell* para realizar nossa próxima tarefa. Sua sintaxe é:

```
for locais in $(cat locais.txt)
do
    grep $locais patrimonio.csv > \$locais.csv
done
```

Em [4], vemos que o **for** indica o *loop*; o **local** é a variável que irá receber o nome de cada local; **$(cat locais.txt)** é a coleção de itens que você deseja percorrer; **do** significa que a partir deste ponto o *loop* começa, **grep $locais patrimonio.csv > $locais.csv** é o comando que irá se repetir, que é procurar com o **grep** a ocorrência de **$locais** (que é o conteúdo da variável locais: cada linha do arquivo **locais.txt**) no arquivo **patrimonio.csv** e ter sua saída salva pelo comando $>$ no arquivo **$locais.csv** (que é cada local com a extensão *.csv*);

$ indica que queremos o conteúdo da variável e não o seu nome; **done** termina o *loop*.

Após realizar o comando, damos um **ls** no diretório corrente para mostrar os arquivos que estão nele. Agora damos um **cat** em cada arquivo criado e veremos que cada um possui os itens respectivos a cada local.

9.4 Script

```
#!/bin/bash

grep -n "[a-zA-Z0-9 ];[a-zA-Z0-9 ]" patrimoniocp.csv

sed -i '175d;734d;735d' patrimoniocp.csv
sed -i 's/"/ /g' patrimoniocp.csv

cut -s -d";" -f5 patrimoniocp.csv
cut -s -d';' -f5 patrimoniocp.csv | sort -u > locais.txt

mkdir locais
mv patrimonio.csv locais.txt locais
cd locais

for locais in $(cat locais.txt)
do
    grep $locais patrimonio.csv > $locais.csv
done
```

10 Exemplo 2 - Tabela de evolução de matrículas em diversas disciplinas em vários semestres

Temos um diretório com vários subdiretórios que representam as disciplinas de um curso. Em cada um deles, há vários arquivos que representam as pessoas que estão matriculadas e seus respectivos cursos divididos em semestres (do 1º semestre de 1988 até o 2º semestre de 2002). No arquivo *19882.dados* temos os dados relacionados com o 2º semestre de 1988 divididos da seguinte forma: **curso:GRR**. Exemplo: **13:198701326**.

Iremos fazer um *script* que terá como saída um arquivo que contém o total de matrículas semestre a semestre considerando que os *GRR's* não se repetem. (Um aluno mesmo se matriculando em várias disciplinas deve ser contado como uma matrícula nessa etapa). Após essa etapa, vamos acompanhar a evolução do número de matrículas em cada disciplina semestre a semestre, tendo como saída uma tabela que tem na vertical os semestres e na horizontal cada curso matriculado no período estudado.

10.1 Explicando o Script

Primeiramente criamos uma cópia do diretório que está os dados que precisamos para não correr o risco de perder informações importantes. Usamos o comando **cp**. [26].

```
cp –R DadosMatricula/ copia/
```

Após criar o nosso *backup*, vamos mudar para o novo caminho criado utilizando o **cd**. [21].

```
cd copia/
```

Nossos arquivos possuem a extensão *.dados* e no início de cada um possui um cabeçalho que não servirá para nós. Para isso, iremos removê-lo com **sed**. [10]. O comando abaixo temos uso de coringas que facilitam o acesso à vários diretórios ao mesmo tempo. O ***/*.dados** acessa todos os diretórios e tudo que possui neles com a extensão *.dados*.

```
sed -i '1d' */*.dados
```

Usamos novamente o coringa (*) com o comando **ls**, [19]. Com isso ele irá listar todos os diretórios e tudo que se encontra neles. A saída desse comando é utilizada pelo **grep '.dados'** ([24]) que faz um "filtro"do **ls** deixando somente o que tem a *string .dados*. Após isso, o resultado de **grep** é passado para o **sort -u** ([6]) que tira todas ocorrências repetidas (não queremos datas repetidas) e com o > salva em um arquivo **datas.txt**. Com isso, temos todas as datas relativas aos arquivos.

```
ls * | grep '.dados' | sort -u > datas.txt
```

Utilizando do **for** ([4]), iremos percorrer data por data e contar o numero de matriculados total. Com o **sort -u */$datas** procuramos em todos os diretórios arquivos que tenham o conteúdo da nossa variável **datas** e tiramos as ocorrências repetidas (queremos uma pessoa matriculada no semestre, não contando quantas matriculas teve um aluno no total) e contamos com o **wc -l** ([13]). Salvamos o resultado em um arquivo e usamos » pois queremos complementar o que foi salvo anteriormente. (O > iria reescrever). Ao mesmo tempo, com o **cut** ([2]) procuramos nosso separador (:) e pegamos somente a primeira coluna (**-f1**) e salvamos em **cursos.txt**. (Essa etapa salva todos os códigos dos cursos existentes em nossos dados).

```
for datas in $(cat datas.txt)
do
    sort -u */$datas | wc -l >> numeromatricula.txt
    cut -s -d":" -f1 */$datas >> cursos.txt
done
```

Usamos novamente o **sort -u** e retiramos todas as ocorrências repetidas no **cursos.txt** e salvamos em outro.

```
sort -u cursos.txt > cursossr.txt
```

Iremos preparar nossos arquivos para nossa tabela. Com o **printf** ([5]) que irá escrever o resultado do **tr** ([22]) no arquivo **temp.** O comando **tr** ira substituir todos os **\n** (quebra de linhas) do **cursossr.txt** (arquivo dos cursos sem repetição) por um espaço em branco. Com isso, todo o arquivo terá somente uma linha.

```
printf "s $(tr '\n' ' ' < cursossr.txt)" '.' >> temp
```

Nosso ultimo **for** terminará nossa tabela. Primeiramente salvamos cada data em uma variável.

```
for datas in $(cat datas.txt)
do
```

Para cada data, escrevemos no **temp** as datas precedidas de um quebra linhas. (Estamos fazendo nossa coluna de datas da nossa tabela).

```
printf "\n$datas" >> temp
```

Para cada curso, salvamos em uma variável.

```
for cursos in $(cat cursossr.txt)
do
```

Com o **grep "$cursos:"*/$datas** procuramos em todos os diretórios arquivos que possuam a extensão com a variável **datas** a expressão **$cursos:** (o conteúdo da variável cursos, que é cada código dos cursos). Com o **wc** contamos o número de linhas da saída do **grep**, que é a quantidade de matrículas em tal semestre e somamos na variável **qntd**.

```
qntd=$(grep "$cursos:" */$datas | wc -l)
```

Se o conteúdo da variável for maior que 0, escrevemos o valor no arquivo **temp**. Se não for, colocamos um 'X' no lugar.

```
if [ "$qntd" -gt 0 ]
        then
                printf " $qntd" >> temp
        else
                printf "%s" 'X' >> temp
        fi
done
```

No nosso arquivo final da tabela escrevemos nosso cabeçalho:

```
printf "\t%s\n\n" 'Numero de matriculas em cada semestre (vertical: semestres,
horizontal: codigo do curso)' > ../tabela.txt
```

No nosso arquivo **temp** temos cada ano/semestre precedidos da quantidade de matrículas por cada curso. Com o **awk** ([18]) iremos colocar cada linha desse arquivo (**temp**) na **tabela.txt**. O **"%7s"** significa que entre cada palavra (números no nosso caso) será colocado 7 espaçamentos simples.

```
awk '{ for(i=1;i<=NF;i++) printf "%7s", \$i; printf "\\n"\}' temp >>
                                                              ../tabela.txt
```

Temos nossa tabela finalizada. Agora para terminar nosso arquivo com o total de matriculas por semestre sem repetição de GRR's, usamos o **paste** ([12]) para concatenar nossas datas com o total de matriculados e salvamos no **matriculaSemestre.txt**.

```
paste datas.txt numeromatricula.txt > ../matriculaSemestre.txt
```

Para tirar o 'X' que substituímos no arquivo onde **qntd** não era maior que 0:

```
sed -i "s/X/ /g" ../tabela.txt
```

Para tirar a extensão **.dados** dos nossos arquivos e substituir por **:** em todos os arquivos com extensão **.txt** e removemos o arquivo que usamos como *backup* (copia):

```
sed -i "s/.dados/ :/g" ../*.txt
rm -rf ../copia/
```

10.2 Script

```
#!/bin/bash

cp -R DadosMatricula/ copia/

cd copia/

sed -i '1d' */*.dados

ls * | grep '.dados' | sort -u > datas.txt

for datas in $(cat datas.txt)
do
    sort -u */$datas | wc -l >> numeromatricula.txt
    cut -s -d":" -f1 */$datas >> cursos.txt
done

sort -u cursos.txt > cursossr.txt

printf "%s $(tr "\n" " " < cursossr.txt)" "." >> temp

for datas in $(cat datas.txt)
do
        printf "\n$datas" >> temp
        for cursos in $(cat cursossr.txt)
        do
```

```
                qntd=$(grep "$cursos:" */$datas | wc -l)
                if [ "$qntd" -gt 0 ]
                then
                        printf " $qntd" >> temp
                else
                        printf " %s" 'X' >> temp
                fi
        done
done

printf "\t%s\n\n" 'Numero de matriculas em cada semestre (vertical: semestres,
horizontal: codigo do curso)' > ../tabela.txt
awk '{ for(i=1;i<=NF;i++) printf "%7s", $i; printf "\n"}' temp >>
                                                                ../tabela.txt

paste datas.txt numeromatricula.txt > ../matriculaSemestre.txt

sed -i "s/X/ /g" ../tabela.txt
sed -i "s/.dados/ :/g" ../*.txt

rm -rf ../copia/
```

11 Exemplo 3 - Contabilizando ocorrências de regras em um log de Firewall

Neste último exemplo iremos criar um código para mostrar o total de pacotes bloqueados por tipo de regras de filtragem de um *Firewall*. No nosso arquivo de exemplo temos informações de pacotes barrados. Na coluna 6 do arquivo encontramos o nome da regra de filtragem utilizada, sendo divididas entre *IPv4* e *IPv6*. Por isso, nosso resultado final além de mostrar a quantidade de bloqueios por regra de filtragem, irá dividir entre os dois protocolos. Também iremos levar em consideração que há um limite normal de bloqueios (20 mil) e que caso haja uma regra com um valor superior a esse, iremos mandar um e-mail para nós mesmos dizendo que isto aconteceu.

11.1 Explicando o Script

Iremos criar 3 variáveis para referenciar a arquivos que iremos usar.

Usada para criar o arquivo onde está as regras que tiveram quantidade de bloqueios superior a 20 mil.

```
mail="mail.temp"
```

Arquivo do log do *Firewall*

```
log="log-firewall"
```

Arquivo das regras utilizadas para filtragem

```
regras="tipo-regras"
```

Agora vamos criar 3 vetores associativos. Esse tipo de *array*, também chamado de *hash* em outras linguagens, consegue indexar seus elementos usando *strings* ao invés dos usuais números inteiros [9]. Como iremos separar em protocolos, criamos um vetor para cada IP.

```
declare -A ipv4
declare -A ipv6
```

Vamos agora criar um vetor associativo dos logs com o **awk** [17]. Um comando poderoso na manipulação de arquivos.

```
qntdR=$(awk '{qntdR[$6]++} END {for (regras in qntdR) print regras,
                                                    qntdR[word]}' $log)
```

O primeiro elemento após o **awk** cria uma variável **qntdR** indexada pela sexta coluna do arquivo (**$6**). Sempre que aparecer uma palavra igual, ela não é atribuída novamente e o conteúdo naquela posição do vetor é incrementado (por isso o ++ logo após). O **for (regras in qntdR)** cria uma variável **regras** que pega os índices do vetor e em cada iteração, escreve o valor dela e o conteúdo relacionado. Tudo isso é salvo na variável **qntdR**. (A atribuição do resultado é feita no início do comando).

Nossa etapa principal está dentro desse **for** [4]. O **awk** está retornando cada linha do arquivo que está na variável **regras** (que possui as regras de filtragem), que foi declarada no início e atribuindo à variável **i**.

```
for i in $(awk '{print $1}' $regras)
do
```

Com o **grep** [24], verificamos se a regra da iteração atual está na variável que possui o resultado do primeiro **awk** (primeira coluna é a regra e segunda coluna é a quantidade de ocorrência de bloqueios). Observação: Quando usamos arquivos, usamos $<$ para redirecionar o arquivo para o comando. Como a variável não é um arquivo mas sim uma *string*, usamos $<<<$ que redireciona a *string* para ser usada. O resultado é a linha que possui a regra.

```
q=$(grep "^$i:" <<< $qntdR)
```

Se o resultado do **grep** não for nulo (**!** no **if** é uma negação e **-z** verifica se há conteúdo), então **qntd** recebe a segunda coluna da linha (quantidade). Se a quantidade for maior (**-gt**) 20 mil, escrevemos a regra e a quantidade sem atribuir a nenhuma variável ou arquivo. Com isso, ao final do **for** sera redirecionado todas as escritas para o **mail**. Se não houver linha, é atribuído zero (não houve ocorrência de bloqueio para aquela regra).

```
if [ ! -z "$q" ]
then
```

```
        qntd=$(awk '{print $2}' <<< $q)
        if [ $qntd -gt 20000 ]
        then
                printf "$i: $qntd\n"
        fi
else
        qntd=0
fi
```

Agora precisamos saber se tal regra é *IPv4* ou *IPv6*. Quando é *IPv6*, há **V6-** no início da palavra. Para isso usamos o **cut** [2] para pegar as letras que estão antes de '-'. Se for igual a **V6**, atribui a quantidade ao vetor associativo **ipv6** com índice sendo a regra que teve o número de bloqueios calculado. Se não for, atribui ao vetor **ipv4**. No final, redirecionamos a saída do **for** (que no caso é o **printf** dos valores superiores a 20 mil) no arquivo **mail**.

```
if [ "$(cut -d"-" -f1 <<< $i)" == "V6" ]
then
        ipv6[$i]=$qntd
else
        ipv4[$i]=$qntd
fi
done > $mail
```

Agora basta escrever cada vetor no arquivo. Abrimos e fechamos chaves para que tudo q esteja dentro possa ser redirecionado juntamente para o mesmo arquivo. Com isso não precisamos redirecionar em cada **print**. Temos um **for** para cada vetor associativo. O **i** recebe **$!ipv4[@]** que são todos os índices do vetor. (O **@** indica o vetor inteiro) [9]. Para cada **i**, é escrito no arquivo final a regra e a quantidade de ocorrências (**$ipv4[@]**). Veja que diferente do comando anterior, esse não possui a exclamação (**!**), indicando que agora queremos o conteúdo e não o índice. Como queremos que o arquivo final tenha como nome a data do dia da execução do *script*, usamos o comando **date** com a formatação em seguida que indica respectivamente Dia, Mês, Ano, Hora, Minutos e Segundos. Esse comando tem várias outras formatações e outros usos como vemos em [20].

```
{
        printf "\n*** V4 ***\n"
        for i in "${!ipv4[@]}"
        do
                printf "%s %s\n" "$i:" "${ipv4[$i]}"
        done

        printf "\n*** V6 ***\n"
        for i in "${!ipv6[@]}"
        do
                printf "%s %s\n" "$i:" "${ipv6[$i]}"
        done
```

```
} > $( date '+%d-%m-%Y_%H-%M-%S' )
```

Usamos agora o comando **mail** que envia um e-mail para o endereço ao final da linha de comando. Com o parâmetro **-E** o e-mail só é enviado se o conteúdo de **mail** não for nulo; Após o parâmetro **-s** colocamos o assunto da mensagem; e o arquivo redirecionado no final é o corpo da mensagem, que no caso são as ocorrências superiores a 20 mil. Após enviar o email, removemos os arquivos temporários com o comando **rm**.

```
mail -E -s "Alerta de seguranca de Firewall" "(seu email)" < $mail

rm *.temp
```

11.2 Script

```
#!/bin/bash

mail="mail.temp"
log="log-firewall"
regras="tipo-regras"

declare -A ipv4
declare -A ipv6

qntdR=$(awk '{qntdR[$6]++} END {for (regras in qntdR) print regras,
                                                    qntdR[word]}' $log)

for i in $(awk '{print $1}' $regras)
do
    q=$(grep "^$i:" <<< $qntdR)
    if [ ! -z "$q" ]
    then
            qntd=$(awk '{print $2}' <<< $q)
        if [ $qntd -gt 20000 ]
        then
                printf "$i: $qntd\n"
        fi
    else
        qntd=0
    fi
    if [ "$(cut -d"-" -f1 <<< $i)" == "V6" ]
    then
        ipv6[$i]=$qntd
    else
        ipv4[$i]=$qntd
    fi
```

```
    done > $mail
{
        printf "\n*** V4 ***\n"
        for i in "${!ipv4[@]}"
        do
                printf "%s %s\n" "$i:" "${ipv4[$i]}"
        done

        printf "\n*** V6 ***\n"
        for i in "${!ipv6[@]}"
        do
                printf "%s %s\n" "$i:" "${ipv6[$i]}"
        done
} > $( date '+%d-%m-%Y_%H-%M-%S' )

mail -E -s "Alerta de seguranca de Firewall" "(seu email)" < $mail

rm *.temp
```

12 Exercícios

Pergunta 1) Procure ler o livro [15] para aprender mais sobre este interpretador de comandos.

Pergunta 2) Descubra um jeito de contar o numero de arquivos/-subdiretorios do diretório Pergunta2

Resposta:

```
ls Pergunta2 | wc -l
```

Pergunta 3) Temos um arquivo texto, notas.txt, com várias notas de alunos. Estão fora de ordem e precisamos criar dois novos arquivos: um com as notas abaixo da média (7) e outro com notas superior ou igual à média. Leve em conta que o arquivo está formatado com o nome sem espaços em branco, seguido de : e a nota. Exemplo: CláudioTorresJúnior:10 Observação: as notas são arredondadas, ou seja, números inteiros.

Resposta:

```
for nota in $(cat notas.txt)
do
    if [ $(cut -d":" -f2 <<< $nota) -lt 7 ]
    then
        printf " $nota \n" >> abaixo.txt
    else
                printf " $nota \n" >> acima.txt
        fi
done
```

Conclusão

Vimos que o *Shell* foi criado para facilitar nossas vidas. Desde o início de seu desenvolvimento, o esperado era algo que ajudasse na interação homem-computador. Várias versões surgiram e com elas a necessidade de sempre estarem evoluindo para atenderem os mais diversos públicos. E com os atalhos para a sua manipulação (tanto do *Shell* em si quanto do *Linux*) seu uso se torna mais rápido e fácil, tornando possível a realização dos trabalhos com o sistema sem a utilização de um mouse. Com os exemplos utilizados vimos também como tarefas que pareciam gigantes e demoradas foram realizadas em questão de segundos. Em uma simples linha de comando pode-se unir vários comandos tendo a saída de um como a entrada do comando adjacente, otimizando o trabalho e evitando o uso repetido de operações. Saber lidar com processos é muito importante para manter o bom funcionamento do computador. Como dito anteriormente, o foco aqui não é explicar completamente o funcionamento de processos e como o sistema operacional os manipulam, e sim dar uma ideia de como o usuário pode ter um controle maior sobre o que está sendo executado em seu computador. Lógico que o controle não se resume ao que foi falado anteriormente, mas saber o básico de alguns comandos e de seu funcionamento é uma porta de entrada para o bom entendimento do funcionamento do sistema *Linux*.

Referências

[1] The University of British Columbia. *Differences Between Unix Shells*. URL: https://www.ugrad.cs.ubc.ca/~cs219/CourseNotes/Unix/shell-Differences.html.

[2] Cleiton Bueno. *Linux – Dominando o comando cut*. URL: https://cleitonbueno.com/linux-o-comando-cut/.

[3] cyberciti. *Setting up permissions on a script*. URL: https://bash.cyberciti.biz/guide/Setting_up_permissions_on_a_script.

[4] MATTIAS GENIAR. *Bash for Loop, o primeiro passo na automação no Linux*. URL: https://imasters.com.br/desenvolvimento/bash-for-loop-primeiro-passo-na-automacao-no-linux.

[5] Carlos Affonso Henriques. *CONHECENDO O PRINTF*. URL: http://www.dltec.com.br/blog/linux/exemplos-de-uso-do-comando-tr-no-linux/.

[6] Computer Hope. *Linux sort commandx*. URL: https://www.computerhope.com/unix/usort.htm.

[7] Infowester. *O que é Linux e qual a sua história?* URL: https://www.infowester.com/historia_linux.php.

[8] Infowester. *Processos no Linux*. URL: https://www.infowester.com/linprocessos.php.

[9] Dæmonio Labs. *Arrays Associativos (Hash) no Bash*. URL: https://daemoniolabs.wordpress.com/tag/array-vetor-associativo-shell-script-bash-hash/.

[10] likegeeks.com. *Sed Linux Command*. URL: https://likegeeks.com/sed-linux/.

[11] Xerxes Lins. *USE O GNOME SEM MOUSE*. URL: https://www.vivaolinux.com.br/dica/Use-o-Gnome-sem-mouse.

[12] Viva o Linux. *O comando paste*. URL: https://www.vivaolinux.com.br/dica/O-comando-paste.

[13] Diego Rodrigo Machado. *O comando wc*. URL: https://www.vivaolinux.com.br/dica/O-comando-wc.

[14] Carlos A. Maziero. *Sistemas Operacionais: Conceitos e Mecanismos*. 2017.

[15] Cameron Newham. *Learning the bash Shell*. O'Reilly Media, Inc., 2005.

[16] Cameron Newhan e Bill Rosenblatt. *Learning the bash Shell, 3rd Edition*. O'Reilly Media, Inc., 2009.

[17] Arnold Robbins. *Arrays in awk*. URL: http://kirste.userpage.fu-berlin.de/chemnet/use/info/gawk/gawk_12.html.

[18] SASIKALA. *8 Powerful Awk Built-in Variables*. URL: https://www.thegeekstuff.com/2010/01/8-powerful-awk-built-in-variables-fs-ofs-rs-ors-nr-nf-filename-fnr/?ref=binfind.com/web.

[19] José Cleydson Ferreira da Silva. *O comando LS de A a Z*. URL: https://www.vivaolinux.com.br/artigo/O-comando-LS-de-A-a-Z.

[20] Rafael Brianezi da Silva. *DESVENDANDO O COMANDO DATE*. URL: https://www.vivaolinux.com.br/dica/Desvendando-o-comando-DATE.

[21] ALEXEI C TAVARES. *Dicas de Uso para o Comando cd do Linux*. URL: http://www.dltec.com.br/blog/linux/dicas-de-uso-para-o-comando-cd-do-linux/.

[22] ALEXEI C TAVARES. *Exemplos de uso do comando tr no Linux*. URL: https://www.vivaolinux.com.br/dica/Conhecendo-o-printf.

[23] Ticursos.net. *Linux Shell Essentials*. URL: https://www.devmedia.com.br/linux-shell-essentials/20158.

[24] Weslley Wellington. *Usando grep e egrep*. URL: https://www.vivaolinux.com.br/artigo/Usando-grep-e-egrep.

[25] wikihow. *Como Abrir uma Janela do Terminal no Ubuntu*. URL: https://pt.wikihow.com/Abrir-uma-Janela-do-Terminal-no-Ubuntu.

[26] Wikilivros. *Comandos para manipulação de Arquivos/cp*. URL: https://pt.wikibooks.org/wiki/Guia_do_Linux/Iniciante%5C%2BIntermedi%5C%C3%5C%A1rio/Comandos_para_manipula%5C%C3%5C%A7%5C%C%5C%A3o_de_Arquivos/cp.