Shell, Relatório Geral

Gabriel M. Segatti

03/12/2018

Resumo

Este texto tem como objetivo abordar a história do UNIX e do Shell, como se deu o desenvolvimento e a progressão dessas tecnologias de forma técnica. Além disso, sobre o Shell será exposto seu uso e suas funcionalidades, desde os componentes básicos, como simples comando a customização do ambiente e outros.

1 Introdução

Unix é uma família de sistemas operacionais desenvolvida para ser uma plataforma conveniente para programadores. Shell é uma interface disponível ao usuário pelo qual pode-se realizar pedidos de funcionalidades pertencentes ao sistema operacional.

Enquanto ferramenta para programadores acostumados ao uso do UNIX, ou similares, o Shell é excepcional, proporcionando comodidade e muita produtividade dos mesmos.

No entanto, para usuários mais novos (*newbies*), tal ferramenta pode aparentar ser algo de complexidade inexorável. Visando auxiliar tais usuários, este documento foi concebido. Estarão expostos na próxima sessão informações pertinentes ao UNIX e o Shell, assim como a história que ocasionou o desenvolvimento de ambos e derivados. Serão expostos, ainda, em maiores detalhes, elementos pertinentes a utilização do Shell, como comandos, variáveis e outros, expondo também casos de uso.

2 História do Unix

2.1 Primórdios

Mediante à necessidade de criar um ambiente adequado para programadores nos anos 60, um esforço conjunto entre o MIT, AT&T Bell Labs e a General Eletric levaram ao desenvolvimento de um sistema operacional chamado Multics. Repleto de conceitos inovadores que permitiam aos engenheiros maior poder sobre a performance e usabilidade do sistema, o Multics foi a fundação sobre o qual os SO's seguintes teriam como influência durante seu desenvolvimento.

Surgem, então, os Shells, que tem como objetivo proporcionar maior facilidade para o usuário interagir com o sistema e manusear arquivos, programas e demais documentos presentes.

2.2 Anos 70

Diante o fracasso do Multics, um dos programadores que havia trabalhado no projeto, Ken Thompson, decide criar seu próprio sistema operacional. Inicialmente denominado Unics, o fruto dessa iniciativa consistiu num produto que tinha como componente um shell demasiado simples, denominado "Thompson Shell". O sufixo do nome, "cs" consistia numa forma de relembrar o projeto do Multics, de certa forma com ironia. Finalmente, a denominação do sistema foi alterada para "Unix".

Com o passar do tempo e o crescimento do Unix, foi criado o Manual do Programador Unix, que referenciava comandos a partir do comando "man", recurso existente até os dias de hoje. Ocorreu também a reescrita do sistema para a linguagem C, o que, posteriormente, viria a contribuir com a portabilidade do sistema. Apesar dessa alteração ocasionar um aumento de um terço no tamanho do sistema, os lados positivos, como maior manutenibilidade, mais

funcionalidades e outros, eram mais significativos. Pouco antes dessas alterações o Shell incorporava Pipes a seus recursos.

2.3 Anos 80

Nesse momento, milhares de pessoas são usuárias do Unix e sua capacidade de tornar-se um sistema portátil para a maior gama de máquinas possíveis. No final de 83 ao menos 16 processadores e arquiteturas diferentes o traziam consigo. A AT&T começou então, a comercialização do Unix, o que, por sua vez, resultou na fundação do GNU Project.

Esse projeto tinha como objetivo prover liberdade e controle aos usuários de computadores por meio do desenvolvimento de Software Livre. Unindo o Kernel feito por Linus Torvals às iniciativas do projeto, surgiu o primeiro sistema operacional que também era software livre, o Linux. Dentre às iniciativas citadas encontram-se o GCC, a GNU Emacs e algumas funcionalidades do Unix, como os comandos de Shell ls, awk, grep e outros.

No final desta década inicia-se a Unix Wars. Ocasionada pelas inúmeras implementações diferentes do Unix, deu-se origem a uma rivalidade entre os criadores dessas partições. Não muito depois, os pedidos por padronização do sistema começaram.

2.4 Anos 90

A Unix Wars teve continuidade durante o princípio da década de 90, embora com menor intensidade. Um dos fatores para tanto foi a criação do Common Open Software Environment (COSE), uma iniciativa cujo objetivo era a implementação de sistemas operacionais unificados. O linux começa a lutar por espaço no mercado e torna-se um dos possíveis competidores do Unix para o futuro.

2.5 Anos 2000

Em 2000, todos os ativos relacionados ao Unix são vendidos ao SCO Group. Esse grupo, por sua vez, entrou na justiça com ações contra vários usuários do Linux alegando que o sistema continha códigos pertencentes ao Unix, e, portanto, pertencentes à corporação. Desde os anos 2000, o Linux tornou-se a principal partição de sistemas similares ao Unix.

3 Interfaces com Shell

Enquanto usuário de um sistema, pode vir a ser útil saber navegar pelo mesmo sem o uso do mouse. Tal hábito pode acarretar em ganhos de produtividade e são até mesmo necessários em sistemas que funcionam somente pela linha de comando. Como fazer isso é o assunto abordado nesta sessão.

3.1 Essenciais

O primeiro comando mais importante para o escopo do que será explicitado durante o decorrer deste documento é como abrir o terminal. Existem duas formas de o fazer. A primeira e mais direta é **CTRL ALT T**. Se, no entanto, o leitor preferir uma abordagem ainda mais simples, **SUPER** abre o *dash* do ubuntu e então basta pesquisar pelo terminal.

Limpar a tela do terminal: **CTRL L** ou comando **clear**. Para fechar o mesmo enquanto não é executado alguma tarefa: **CTRL D** ou digitar **exit** na linha de comando. Parar uma execução é feito por **CTRL C**. Aumentar o zoom corresponde a **CTRL SHIFT** +, já diminuí-lo **CTRL -** e retornar ao padrão **CTRL 0**. Bloquear sessão, pode ser tanto pelo atalho **CTRL ALT L** ou pelo comando **gnome-screensaver-command -l**. Desligar a máquina é feito por **shutdown**, que, após um minuto a desligará. Caso deseje desligar no exato momento o argumento **now** pode ser adicionado ao comando. O comando **gnome-session-quit** encerra a sessão atual.

3.2 Espaços de Trabalho

Existem maneiras de chavear entre os espaços de trabalho usando somente atalhos no teclado de maneira cômoda e rápida. **CTRL ALT <seta direcional>** permite que o usuário vá para o espaço de trabalho superior, inferior ou vizinho. No entanto, sabendo somente isso, nos encontramos limitados a fazer somente essa operação, tendo que recorrer ao mouse e interface gráfica para demais funcionalidades como deletar um dos espaços de trabalho, por exemplo.

O comando **wmctrl** interage com dados respectivos ao X Window Manager que por sua vez controla a interação das janelas com o sistema. Por exemplo, **wmctrl -d** lista no terminal todos desktops administrados pelo window manager, o nome e número de cada um deles, assim como outras informações, como qual pixel delimita o início de um desktop. Com o argumento '-s' conseguimos escolher para qual espaço desejamos ir, basta informar o nome ou número do mesmo (wmctrl -s 2 nos levaria ao espaço de trabalho 2, claro). '-o' permite ao usuário mudar de workspace pelo valor do pixel, indicado em **wmctrl -d**. Outra opção importante é '-n' que permite ao usuário escolher o número de desktops a sua disposição.

3.3 Aplicações

Para executarmos alguma aplicação a partir da linha de comando, basta digitar o nome da mesma (supondo que esteja instalada) no terminal. O problema de fazer isso é que o funcionamento do programa fica aliado ao terminal usado para o abrir. Logo, se digitarmos "firefox" no Shell e posteriormente fechá-lo, a janela do navegador também será imediatamente fechada. Visando menor dependabilidade pode-se optar pelo nome da aplicação a ser executada seguida do "&". Isso indica ao Shell que o comando atual deve ser executado no background,

e o terminal atualmente aberto pode continuar a receber outros comandos normalmente.

3.4 Calendários

Existem dois comandos relacionados a calendários no linux, **calendar** e **cal**. Enquanto **cal** dispõe o calendário em um formato tradicional, com o dia atual marcado, podendo dispor os dias desde o ano 1 até 9999, ou somente o mês atual, **calendar** exibe os acontecimentos importantes que aconteceram no dia de hoje e amanhã.

3.5 Calculadoras

Além de podermos usar a calculadora padrão do sistema por meio da entrada **gnome-calculator** na linha de comando, o usuário pode recorrer ao próprio terminal para realizar contas se desejar interação ainda menor com interfaces gráficas. Por meio do comando **gcalccmd** ou **bc** o terminal passa a aceitar expressões numéricas e retornar o resultado como entrada.

4 Recursos do Shell

4.1 Comandos

A linha de comando do Shell é operada por um conjunto de palavras separadas por espaços dentre as quais a primeira constitui um comando e as demais são argumentos. Estes argumentos variam seus tipos, podendo determinar de que forma, ou, sobre o que o comando deve agir. Chamamos um argumento de opção quando este informa qual deve ser o comportamento do comando. Caso o argumento não seja uma opção, temos de fato, um parâmetro sobre o qual o comando irá agir.

exemplo: echo -n essa frase não tem quebra de linha

No exemplo exibido "-n" corresponde a opção que delimita que não ocorrerá uma nova linha quando ocorrer a execução da instrução. Já "echo" corresponde ao comando em si e o restante ao parâmetro sobre o qual será executado.

4.2 Arquivos e Diretórios

Existem três tipos de arquivos mais importantes visto o escopo deste artigo em um sistema UNIX: arquivos de leitura, arquivos de execução e diretórios. Dos tipos citados acima os diretórios se destacam por conter uma estrutura hierárquica que permite a existência de uma árvore que mantém a disposição de todos os arquivos do sistema. Se seguirmos desde a raiz da árvore até o endereço de um arquivo temos o full pathname do mesmo. Se um arquivo encontra-se no diretório atual temos o relative pathname. Para descobrir o relative pathname de arquivos do diretório atual basta executar o comando pwd. A instrução pwd indica o diretório atual mostrando seu full pathname, já cd é utilizado para

"caminhar" entre os diretórios. O comando "cd ~" sempre retorna ao diretório designado como "home".

4.3 Wildcards (Coringas)

Wildcards são utilizadas quando desejamos utilizar um comando sobre um determinado conjunto de arquivos. Os "coringas" essenciais e suas correspondentes interpretações são:

Wildcard	Matches
?	Any single character
*	Any string of characters
[*set*]	Any character in *set*
[!*set*]	Any character *not* in *set*

O processo de utilizar wildcards para encontrar arquivos no sistema é chamado de wildcard expansion.

4.3.1 Wildcard ?

Suponha, por exemplo, que estamos em um diretório onde existem vários documentos e queremos encontrar aqueles cujos nomes são: "trabalho.pdf", "trabalho.txt", "trabalho.cpp" e assim por diante. O comando "ls trabalho.???" nos daria todas ocorrências destes arquivos, independente de qual formato estes sejam, contanto que após o ponto tenhamos somente três caracteres. Este coringa então, pode corresponder a qualquer char existente.

4.3.2 Wildcard *

Se quisermos encontrar todas as ocorrências de arquivos cujo nome é "trabalho.", mas de qualquer formato que possua abreviação de tamanho não só de três caracteres, como "trabalho.c" ou "trabalho.html" poderíamos usar "ls trabalho.*".

Ou se é necessário encontrar arquivos com o nome "virgil" em qualquer posição que seja, usamos "ls *virgil*". Logo "*" corresponde a qualquer string existente, de qualquer tamanho possível.

4.3.3 Wildcard [ ]

Esta wildcard específica um relacionamento de "or" entre os conteúdos dentro dos colchetes de tal forma que, se quisermos efetuar novamente um comando respectivo ao da seção 1.3.1, digitamos "ls trabalho.[a-z][a-z][a-z]". [a-z] significa, a grosso modo, qualquer letra de a até z. Tanto "[a-z]" como "[abcdef...z]" possuem o mesmo significado.

Da mesma forma funcionam números, se digitarmos "ls [1-6].pdf" buscaremos os arquivos que se chamam "1.pdf", "2.pdf" até o número 6.

O caractere "!" corresponde à negação do que está contido nos colchetes. Se então executarmos "ls [!1-6].pdf" teremos os arquivos "7.pdf", "8.pdf" e "9.pdf".

4.3.4 Expansão de Chaves

Enquanto os exemplos anteriormente citados são expansões de caminho, existe ainda a expansão de chaves. A expansão de chave é, basicamente, um mecanismo de gerar strings. Desta forma: "echo a{d,c,b}e" gera "ade ace abe".

Podemos usar essa expansão para os usos citados nas seções anteriores. Se procuramos por documentos em pdf das disciplinas CI320, CI166 e CI058, por exemplo, basta o comando "ls CI{320,166,058}.pdf" e seriam gerados os comandos "ls CI320.pdf", "ls CI166.pdf" e "ls CI058.pdf".

4.4 Entrada e Saída

Arquivos de I/O tem como formato de seus conteúdos sequências de caracteres que são bytes. Tudo no sistema que produz ou aceita dados é tratado como um arquivo. O funcionamento de entrada e saída no UNIX é baseado nestes dois simples conceitos.

4.4.1 Redirecionamento de Entrada e Saída

O redirecionamento de dados se dá pelos operandos "$>$" e "$<$". Logo, comando $<$ arquivo faz com que o comando receba como entrada os dados de um arquivo. Já o contrário, comando $>$ arquivo faz que a saída do comando seja redirecionada ao arquivo.

Enquanto lidando com arquivos e redirecionando a saída de um comando para um arquivo o operador "$>$" sobrescreve qualquer dado anteriormente existente à execução. Portanto, para concatenar a saída atual ao conteúdo já existente usamos o operador "$»$".

Desta forma, ao executarmos "echo 12 $>$ file", o arquivo terá como conteúdo o número 12. Ao realizarmos "echo 123 $>$ file", 12 será sobrescrito e 123 será o único conteúdo de file. Se fizermos "echo 123 » file", no entanto, o arquivo terá tanto 12 como 123 guardado dentro de si.

4.4.2 Pipeline

O redirecionamento de dados entre comandos e arquivos acontece por meio dos operandos citados na seção anterior. No entanto, se pretendemos redirecionar dados de entrada e saída entre comandos, nos valemos do uso de pipes, escritas como " | ". Desta forma, suponha que exista um arquivo chamado "file" que foi preenchido com números aleatórios, por meio dos comandos apresentados anteriormente. Sabemos que "cat $<$ file" tem como saída o conteúdo do arquivo, ou seja, uma sequência não ordenada de números. Para redirecionarmos a saída deste comando para o comando sort e, então, ordenarmos o arquivo, basta usarmos "cat $<$ file | sort -n" (a opção "-n" indica que o comando sort ordenará os

dados com base em seus valores numéricos). Assim, a saída do comando "cat" funciona como entrada para "sort".

5 Progressão do Shell

Esta sessão trata de explicitar as diferenças e evoluções do Shell ao longo dos anos. Desta forma, torna-se claro como os assuntos tratados na sessão anterior surgiram ao longo dos anos.

5.1 Thompson Shell

Influenciada pelo Shell do Multics, esse foi a primeira ferramenta do tipo a ser desenvolvida para um sistema Unix. Apesar de ser inerentemente simples, apresentava inovações quando comparada aos shell's anteriores, tendo, em destaque, uma sintaxe simplificada tratando-se de redirecionamento de entrada e saída. Enquanto no Multics comandos literais realizavam essa funcionalidade, aqui bastava usar os símbolos '>' ou '<'. Posteriormente os Pipes foram introduzidos e ambos símbolos de redirecionamento são utilizados até hoje.

Um shell derivada a partir desta trouxe outras inovações que vieram a tornar shell scripting mais adequado para programadores. Chamada de Masey Shell, essa versão da Thompson incorporou comandos como if-else, goto, switch e while. Variáveis surgiram, embora seus nomes só podiam ser limitados a uma letra, e o dígito "$" tornou-se um desreferenciador das mesmas.

5.2 Bourne Shell

Codificado após seu criador, Stephen Bourne, participar da construção de um compilador ALGOL68, esse shell é reconhecível por ter sintaxe similar a linguagem ALGOL (embora tenha sido feito em C). O Bourne Shell, por sua vez, apresentava ferramentas de controle de fluxo e variáveis, diferentemente do shell de Thompson. Existe o argumento de que o shell de Masey (citado anteriormente), pode ter sido o primeiro shell a incorporar tais funções, no entanto, como era um produto derivado de outro shell, esses comandos só foram incorporados "por cima". Logo, o Bourne shell tornou-se o primeiro a ter, desde sua primeira implementação, tais funcionalidades.

Esse shell serviu de influência para o desenvolvimento de outros shell's, tais como o Korn Shell e o Bash, vide sua importância.

5.3 C Shell

Esse shell tinha como característica de design uma sintaxe similar e seja compatível ao da Linguagem C, por isso tal nome. Um recurso notável introduzido foi o comando history. Graças a isso os comandos previamente executados permaneciam na memória para serem recuperados, se necessário. Logo as setas do teclado serviam para iterar pela história de comandos já digitados, funcionalidade que persiste até hoje.

5.4 Bash (Bourne-Again Shell)

Outro fruto do GNU Project, o Bash tinha como intuito substituir o Bourne Shell, apresentando maior versatilidade. Era o shell por default de sistemas Linux e MacOS. Suportando controle de fluxo, wildcards, variáveis, pipes e o comando history, o Bash reuniu todos recursos previamente apresentados que se mostraram úteis, tomando recursos básicos como keywords e sintaxe do Bourne Shell.

6 Customização da Linha de Comando

Nessa sessão, abordaremos algumas das customizações que podem ser efetuadas no Shell. Isso permite aos programadores maior agilidade e produtividade enquanto realizando tarefas.

6.1 Aliases

Existem comandos que possuem um certo grau de dificuldade para executá-los, seja por causa de seu nome complicado, os argumentos necessários para realizar determinado efeito ou as inúmeras options utilizadas. Para evitar a repetição e a dificuldade causadas pela necessidade de os digitar, existem as Aliases.

Trata-se de uma espécie de apelido ao comando que pode ser usado para executá-lo. Desta forma, se temos:

```
grep palavra rel1.tex
```

Embora não seja um comando complicado, podemos fazer:

```
alias listar='grep palavra rel1.tex'
```

E, toda vez que digitarmos "listar" na linha de comando, o grep acima será executado.

Se queremos "trocar" somente o nome de um determinado comando, sem argumentos ou options basta fazer:

```
alias buscar=grep
```

Essa funcionalidade também torna-se útil quando queremos evitar perder tempo com erros de digitação:

```
alias grpe=grep

alias sde=sed
```

Para ver a lista de todas aliases que basta digitar alias. Para remover um alias, digite-o após o comando unalias.

6.2 Shopt

Shopt é um comando que permite determinar o comportamento do Shell, habilitando, ou não, o funcionamento, de variáveis de configuração.

Enquanto shopt -s funciona como habilitador, -u faz o contrário. Se habilitarmos “mailwarm”, por exemplo, enquanto checando um e-mail que já foi acessado, o shell mostrará uma mensagem dizendo que este já foi acessado. Enquanto shopt -s funciona como habilitador, -u faz o contrário. Se habilitarmos “mailwarm”, por exemplo, enquanto checando um e-mail que já foi acessado, o shell mostrará uma mensagem dizendo que este já foi acessado.

6.3 Variáveis Internas

6.3.1 Variáveis de Prompt

Existem algumas variáveis que controlam a aparência do prompt de comando, chamadas de PS1, PS2, PS3 e PS4, respectivamente, que guardam consigo dados no formato de uma string. Abordaremos PS1 e PS2, visto que as demais são relacionadas a debugging.

PS1 é disposta toda vez que o terminal é aberto, expondo o que seria o “nome” do user. Como padrão essa guarda os caracteres “\u”, “\h”, “\w” e "\$”. O primeiro corresponde ao username do usuário atual, seguido do nome do host até o primeiro “.”, o diretório atual e uma hashtag ou o sinal $. Ao invés de editar o arquivo .bashrc para mudar o valor dessas variáveis, basta fazer PS1=”\u-> “, se quisermos alterar o nome do user, PS1=”\w-> “ para o diretório, e assim em diante.

PS2 surge quando algo é digitado de forma incompleta e o shell aguarda por mais dados de entrada. Logo, se ***grep “la*** é enviado, por default o shell faz uma quebra de linha e aponta a não completude do comando por meio do caractere “>”. Logo, é possível editar o valor de PS2 para que uma mensagem mais clara seja enunciada.

6.3.2 Caminho de Busca de Comando

Para todo comando disponibilizado para o shell existe um arquivo que contém o código sobre qual o mesmo é executado. A variável PATH guarda consigo a lista de diretórios que o shell procura quando tenta-se executar algo.

A relevância desse fato encontra-se na possibilidade de que, eventualmente, quando o usuário desejar escrever seus próprios programas em shell e os manter prontos para serem executados, o diretório a qual eles pertencem deve estar contido em PATH.

No entanto, nem sempre o shell itera por todos diretórios possíveis para encontrar os comando necessários. Uma hash table é mantida para encontrar o que é desejado e evitar demasiadas buscas nos conteúdos de PATH.

7 Uso do vi

Considerado o editor padrão em relação ao UNIX, o vi fornece ao seu usuário produtividade e economia de tempo. No entanto, para obter-se o mesmo, é imprescindível controle sobre a forma de o usar, logo, essa seção apresenta os essenciais.

Como básicos temos:

dh	deleta o caractere anterior
db	deleta a palavra anterior
ESC	entra em modo de controle
h	move um caractere para à esquerda
l	move um caractere para à direita
w	move uma palavra para à direita
b	move uma palavra para à esquerda
O	move para o início da linha
$	move para o fim da linha
d0	deleta até o início da linha
d$	deleta até o fim da linha
dw	deleta uma palavra para frente
db	deleta uma palavra para trás

É possível realizar muito mais com o vi, visto que é uma ferramenta muito poderosa. Para tanto, consulte o manual criado por membros da Universidade de Stanford[1].

8 Processos

Ao executar qualquer programa no UNIX, inicia-se então um processo, criado pela chamada do sistema fork. Isso se chama background process, ou seja, tudo que acontece invisível para o usuário. Para todo processo criado, denomina-se uma identidade para o identificar ao meio de outras tarefas realizadas. Diante a amplitude que processos representam para com o funcionamento das atividades desempenhadas por um computador, torna-se clara a importância de abordar tal assunto.

Saber e dominar como se dá o funcionamento de programas "embaixo" da camada apresentada ao programador permitem ao mesmo maior flexibilidade, velocidade e segurança: ele irá saber quando paralelizar/executar vários processos simultaneamente ou quando matar certos processos sem danificar arquivos importantes, por exemplo.

8.1 ID's e Números de Job

Enquanto executando programas o UNIX atribui aos mesmos ID's, números que funcionam como uma identidade. Por exemplo, o comando firefox & faz com que o Shell dê como saída algo similar a **[1] 2997**. Neste caso, **2997** corresponde à identidade do processo, já **[1]** diz respeito ao número do job, que, por sua vez é dado pelo Shell.

Estes números (de job) são respectivos a quantidade de processos executando sob o Shell, em contrapartida, as ID's são respectivas a todos processos em funcionamento sob seu sistema operacional. Conforme mais programas rodam, maior todos estes números tornam-se.

8.2 Primeiro Plano e Planos de Fundo

O comando **fg** (foreground) traz jobs rodando em background no Shell para o primeiro plano do mesmo, ou seja, a execução se dará da mesma forma que se o comando tivesse sido digitado sem o **&**.

Usando **jobs** é listado todos programas executados em background. Logo, usando **fg %<número do job>** faz-se o mencionado no parágrafo anterior para o job respectivo ao número. Pode-se utilizar também o nome do job para efetuar tal comando. Se executado sozinho, sem parâmetros, **fg** age sobre o job mais recente.

Inversamente, **bg** traz jobs realizados em primeiro plano para o background do sistema. Seus parâmetros funcionam da mesma forma que **fg**.

Alternativo a ambos comandos citados, se o programador não deseja mudar o plano de execução de um job, e deseja simplesmente o pausar **CTRL Z** fará o truque.

8.3 Sinais

Um sinal e uma notificação assíncrona enviada para um processo. Ao ser enviado, o sistema operacional interrompe o fluxo de execução a ser seguido para entregar o sinal. **CTRL Z** é, no fundo, o envio do sinal *TSTP* (terminal stop). Da mesma forma funciona **CTRL C**, por exemplo, que envia o *INT* (interrupt).

O comando **stty** permite que o programador crie seus próprios atalhos para enviar diferentes sinais. Visto que a sintaxe muda de acordo com diferentes sistemas operacionais, recomenda-se consultar à manpage do comando em seu sistema.

Têm-se também **kill**, que por sua vez envia SIGTERM que requer de um processo a chamada de sistema textitexit.

8.4 Visualização de Processos

Existem maneiras mais intuitivas de visualizar os processos executados em determinado momento, assim como as implicações das execuções dos mesmos,

como uso de memória, qual usuário o roda, dentre outros. Isso pode ser feito a partir dos comandos **top**/**htop**. Ambos são gerenciadores de processos interativos que atualizam os dados de uso do sistema com determinada frequência, embora a interface do **htop** seja mais *user-friendly*.

Mais simples que os comandos citados acima, **ps** (process status) também mostra quais processos estão em execução durante sua chamada, embora, se passado sem nenhum parâmetro, apresenta dados de maneira bem menos rica que os comandos citados anteriormente.

8.5 Traps

Traps permitem aos programadores usarem os sinais que comunicam-se com os processos a seu favor em seus programas. A sintaxe do comando **trap** se dá da seguinte forma: **trap <comando> sinal1 sinal2...**, onde "<comando>" é respectivo a qualquer coisa que o programador deseje fazer caso qualquer um dos sinais a seguir sejam capturados pela trap.

Suponha que um programador deseje fazer um script "invencível" contra o atalho **CTRL C**, que envia o sinal *INT* (interrupt). Usando **trap <comando> INT** o shell simplesmente realizará o que for especificado em "<comando>" (podendo inclusive ser uma string nula, visando ignorar o sinal) e a execução continuará.

Logo, neste cenário, para acabar com a execução de uma vez por todas, **kill** seria o ideal.

8.6 Co-rotinas

Se existem dois processos que executam simultaneamente e, existe a possibilidade de comunicação entre ambos, os chamamos de co-rotinas. Co-rotinas permitem ao programador a melhor utilização dos recursos disponíveis para o mesmo. Então, se desejamos rodar dois programas, onde um realiza diversas operações em disco e o outro utiliza muito a CPU, seria beneficente rodá-los concomitantemente.

9 Casos de Uso do Shell

9.1 Shell em Documentos .csv

Dada uma planilha que contém dados sobre patrimônios pertencentes ao DINF e suas respectivas localizações, três problemas necessitam ser solucionados: eliminar pontos e vírgulas que não pertencem a separação de campos, obter, em um único arquivo, a lista de todos locais no departamento de forma ordenada e, por fim, para todo local, criar uma planilha que contenha os patrimônios pertences somente ao mesmo.

9.1.1 Primeira Etapa

A resolução dessa etapa consiste em eliminar pontos e virgulas em posições erradas em um documento chamado "patrimonio.csv". Um campo sem erros seria:

"476255";"MESA";"TIPO GRANDE, COR BEGE";;"2100.13.02.17;"

Enquanto um campo com erros seria um que contém ";"entre as palavras que descrevem um determinado objeto, como:

476255";"MESA";"**TIPO GRANDE;** COR BEGE";;"2100.13.02.17;"

Sabendo que os erros encontram-se entre palavras temos quatro formatos sob quais os pontos podem estar dispostos:

1-palavra;outraPalavra
2-palavra; outraPalavra
3-palavra ;outraPalavra
4-palavra ; outraPalavra

Para encontrá-los iremos usar o comando grep que encontra conjuntos de dados em texto que correspondem a expressão regular passada por parâmetro. Digitamos, então, na linha de comando:

```
cat patrimonio.csv | grep ".*;[A-Z]"
```

Primeiro, a saída do comando cat funciona como entrada para grep, por meio do pipe. A expressão regular informada específica que procuramos por um ponto e vírgula, precedido por qualquer tipo de caractere (“.*”), seguido de qualquer letra de a até z em maiúsculo (“[A-Z]”). Logo, este comando procura por ocorrências do número 1 e 3 e encontra somente uma pontuação mal colocada, do tipo 1.

Resta descobrir as ocorrências de erros tipo 2 e 4. Usando:

```
cat patrimonio.csv | grep ".*;[^;\textbackslash"0-9A-Z]"
```

Queremos, especificamente, encontrar pontos e vírgulas sucedidos por um espaço em branco, logo temos [^;\"0-9A-Z] onde “^” é uma negação, o que corresponderia a qualquer ponto e vírgula que não é sucedido por qualquer número, letra, aspas ou ponto e vírgula.

Por fim, encontramos duas ocorrências do tipo 2, somente. Temos então 3 erros que devemos consertar. Eles se dão da forma:

1-palavra;outraPalavra
2-palavra; outraPalavra

Usando o comando sed, que tem como uma de suas funcionalidades a substituição de texto, fazemos:

```
sed -i -r 's/[A-Z];[^\"0-9]/,/' patrimonio.csv
```

Usamos as options -i para habilitar que sed faça mudanças no arquivo e -r para usar expressões regulares. Logo, [A-Z];[^\"0-9] une todos erros que encontramos e /,/ os substitui por vírgulas.

9.1.2 Segunda Etapa

Agora é necessário encontrar todos os locais existentes em patrimonio.csv, ordená-los e dispô-los em um único arquivo, sem repetições.

Observando, é possível perceber que todos locais começam com a string "2100", logo um simples grep "2100" basta para os encontrar. Como grep retorna toda a linha em que algo correspondente foi achado, usamos -o", para retornar somente aqui que de fato, corresponde ao parâmetro do comando.

Todos locais tem como sufixo ponto e vírgula seguido de aspas, então usamos uma expressão regular [^\";] para os mesmos não serem pegos durante a execução, e direcionamos tudo aquilo que encontrarmos para um arquivo chamado locais:

```
cat patrimonio.csv | grep -o "2100.*[^\";]" > locais\\
```

Por fim, basta ordenar o arquivo e garantir a unicidade de seus conteúdos. "sort -n -u locais -o locais" os ordena sendo que: "-n" diz ao comando para o fazer de acordo com o valor numérico das strings, "-u" elimina repetições e "-o" escreve o resultado do comando ao próprio arquivo.

9.1.3 Terceira Etapa

A parte final do problema consiste em, para cada local existente, criar um arquivo com o nome do local que guarda todos patrimônios nele presentes.

Temos em "locais" a lista de todas possíveis localizações, logo, iteramos por elas, e as usamos para "fazer um grep" no documento original. Desta forma encontraremos todos os patrimônios necessários bastando redirecionar a saída do grep para um novo arquivo, cujo nome é local em si:

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

9.2 Buscando Dados em Diretórios Diversos com Shell

É necessário resolver dois problemas. Primeiro, criar um documento que mostra a evolução do número total de matrículas no DINF ao longo dos anos. Depois, queremos acompanhar a evolução do número de matrículas por curso para cada disciplina no departamento ao longo dos anos, dispondo-os em colunas.

Em um diretório chamado "DadosMatriculas" existem diversos diretórios respectivos às disciplinas ofertadas no DINF. Dentro dessas pastas existem cujos nomes são um determinado ano, seguido pelo semestre e o sufixo ".dadhttp://www.inf.ufpr.br/ci208/os". Logo o arquivo "20001.dados" contém dentro de si informações relativas ao ano 2000, primeiro semestre. Os dados de cada curso estão dispostos a cada linha no formato: códigoDoCurso:GRR.

9.2.1 Buscando o Número Total de Matrículas por Semestre

Para contarmos a quantidade total de matrículas do primeiro semestre disponível, 19881, precisamos entrar no diretório de cada disciplina e pegar somente os dados do primeiro documento. Da mesma forma, para 1988.2, precisamos entrar em cada diretório e pegar os dados do segundo documento, assim em diante.

A dificuldade em fazer isso encontra-se no fato de que temos que sistematicamente entrar em um diretório, pegar o n-ésimo documento e repetir, sendo que todos comandos que temos para lidar com a manipulação de dados de diretórios (ls, por exemplo) não nos provém tamanha especificidade em suas funcionalidades.

Uma forma de realizar tal iteração é pelo uso de loops aninhados. Logo, estando no diretório "mãe" dos demais, fazemos:

```
for (( i = 1988; i < 2003; i++ )); do
for (( j = 1; j < 3; j++ )); do
for dir in $(ls); do
```

Agora, temos três loops em que i = 1988, 1989, 1990... 2002, j = 1, 2 e dir = CI048, CI055, CI056. Para acessarmos os arquivos em sequência basta desreferenciar i e j em cada diretório contido em dir. Logo, na primeira iteração, ao realizarmos cd $dir, "$i$j.dados" é equivalente a "19881.dados" no diretório CI048.

Para contarmos as matrículas totais de um determinado semestre precisamos saber os GRR's matriculados para não contarmos o mesmo aluno. Sabemos que os dados estão por exemplo, assim: "13:198701326". Logo, fazendo:

soma=$soma' '$(grep ':[0-9].*" ij.dados | cut -f2 -d ":")

Guardamos na variável "soma" a busca de "grep :[0-9].*" que corresponde, no exemplo acima, a ":198701326" pipelined a "cut -f2 -d ":" que nos dá a segunda coluna de tal saída, ou seja, somente o GRR "198701326'. O grep que procura

por um número é importante pois ignora a primeira linha de cada arquivo, que não tem significado para o objetivo em questão e poderia atrapalhar na contagem.

Em um arquivo temporário "temp" salvamos "soma". A maneira que iremos usar para contar as matrículas totais será usando **"wc -l"** para contar a quantidade de linhas totais de temp. Para tanto, é necessário retirar as repetições de GRR's e separá-los por linha.

Primeiro **"sed -i -e 's/\s\+/\n/g' temp"** para retirar uma ou mais ocorrências de espaços em branco (\s \+/) e substitui-los por quebras de linha, depois **"sort -u -o temp temp"** para ordenarmos os GRR's, sem repetições. Agora que o arquivo temp contém à quantidade de linhas igual à quantidade de matrículas de um determinado ano, basta redirecionar a saída de **wc -l** para o arquivo final. O script como um todo pode ser escrito da forma:

```
#!/bin/bash


for (( i = 1988; i < 2003; i++ )); do
for (( j = 1; j < 3; j++ )); do
for dir in $(ls); do
if [ -d $dir ]; then
cd $dir
soma=$soma' '$(grep ':[0-9].*' $i$j.dados | cut -f2
-d ":")
cd ..
fi
done
echo $soma > temp
sed -i -e 's/\s\+/\n /g' temp
sort -u -o temp temp
echo $i$j: $(wc -l < temp) >> evol
unset soma
> temp
done
done

rm temp
```

9.2.2 Evolução do Número de Matrículas por Curso a Cada Semestre

A resolução desse problema dividiremos o script em duas subseções, a primeira consistindo em descobrir, para cada matéria, todos os cursos que já tiveram alunos nelas matriculadas. Após esse passo iremos buscar o número de matrículas que cada curso obteve em todos semestres e dispor tais dados em colunas.

9.2.3 Gerando a Lista de Cursos para Cada Disciplina

Esse passo é essencial para, posteriormente, sabermos por quais cursos estamos procurando em cada disciplina. A execução dos comandos expostos nesta seção acarretarão na criação de um documento chamado "cursos" (cada disciplina terá um), que por sua vez, guarda a lista que desejados.

Para cada diretório corresponde à disciplinas, entraremos no mesmo, buscaremos em cada arquivo todas ocorrências de cursos, e iremos guardá-las sem repetições. Dois loops aninhados cumprem a parte de iterar pelas disciplinas e pelos seus documentos que guardam informações sobre os semestres. Anteriormente, quando queríamos os GRR's usamos **"cut -f2 -d ":"**, agora, como desejamos extrair a primeira coluna da saída do grep, basta mudar o **"-f2"** para **"-f1"**.

Cada Grep | Cut tem sua saída direcionada para o arquivo cursos que é ordenado sem repetições por sort -u.

```
for materia in $(ls); do
if [ -d $materia ]; then
cd $materia
for files in $(ls); do
if [ "$files" != "cursos" ]; then
grep [^a-z:] $files | cut -f1 -d ":" >> cursos
fi
done
sort -u -o cursos cursos
cd ..
fi
done
```

9.2.4 Gerando Dados Finais

Por fim, nessa seção será abordado como conseguir, para cada curso, a evolução do número de alunos matriculado em uma determinada disciplina do DINF por semestre.

Para cada disciplina existente, procuraremos em cada documento o número de ocorrências do código de um curso (no documento "cursos") e guardaremos em um arquivo temporário. Supondo que estamos iterando pelos arquivos (por exemplo 19811.dados) e anos guarda o nome deste arquivo, o loop:

```
for linha in $(cat cursos); do
saida_mat=$(cat $anos | grep -c "$linha:")
echo $saida_mat >> temp
done
```

É pega a saída de 19811.dados e, pelo pipe, procura o número de ocorrências da linha do documento cursos (que guarda, em cada linha, o código de um curso já matriculado naquela disciplina) e redireciona para um arquivo temporário.

Agora, para cada diretório de uma disciplina, basta imprimir os códigos dos cursos nele já registrados e os conteúdos do arquivo temporário com pipe para column e os dados são dispostos em colunas.

9.3 Tratando Dados de um Firewall com Shell

O problema consiste em apresentar de forma simples o total de bloqueios por tipos de regras de filtragem, para que, desta forma, uma pessoa encarregada de administrar o sistema possa inferir se a situação atual é normal ou não. Além disso, estamos interessados em separar os bloqueios pelos protocolos que estes pertencem, tudo isso de forma eficiente, dado que os documentos usados para tanto são demasiado grandes.

Ao final deste processo, o script criado deve enviar a algum email um aviso, caso o número de bloqueios seja maior que o esperado.

9.3.1 Número Total de Ocorrências de Um Erro

Para contarmos a quantidade total de erros usamos o comando **awk**. Ao executarmos tal comando será criado um vetor, que, para cada ocorrência dos erros presentes na sexta coluna do arquivo, será incrementado o valor 1. Ou seja, uma contagem de quantas vezes uma palavra aparece. Fazemos:

```
awk '{count[$6]++} END {
for (word in count)
print word, count[word]}' log-firewall > temp
```

Agora, temos o arquivo "temp"guardando o número de ocorrências (quando diferente de 0) e os respectivos erros.

Para adicionarmos os erros que não ocorreram, iteramos pelo arquivo "tipos-bloqueios"e, para cada linha nele, verificamos se a linha existe no arquivo "temp". Se existe, ótimo, não fazemos nada. Caso contrário:

```
echo $linha: 0 >> temp
```

9.3.2 Separando Erros IPv4 e IPv6

Agora que sabemos para todo erro, a quantidade de vezes que o mesmo acontece, mesmo se igual a 0, desejamos separar em um arquivo os erros IPv6 daqueles que são IPv4. Supondo a criação de dois arquivos, "bloqueios-V6"e "bloqueios-V4":

```
echo "***V6***" > bloqueios-V6
echo "***V4***" > bloqueios-V4

for tipos_bloq in $(cat temp); do
if [[ "$tipos_bloq" = V6* ]]; then
echo $tipos_bloq >> bloqueios-V6
```

```
else
echo $tipos_bloq >> bloqueios-V4
fi
done
```

Assim, para cada linha cujo nome do bloqueio inicia-se com o prefixo "V6"seguido por qualquer coisa, a escrevemos no final do arquivo "bloqueios-V6". Caso contrário, escrevemos no fim de "bloqueios-v4". Agora, possuímos dois arquivos distintos para os diferentes erros. Para os concatenar em um único arquivo faça:

```
cat bloqueios_V6 bloqueios_V4 > saida
```

9.3.3 Enviando Aviso

Precisamos, para todo bloqueio que ultrapassar 20000 ocorrências, enviar ao administrador do sistema, um arquivo mostrando quais eles são. Temos duas opções, iterar a saída final e procurar por quais bloqueios passam de 20000, ou, modificamos o **awk** anterior, redirecionando previamente o que desejamos para um arquivo "warning". Desta forma:

```
awk -v warning='warning' '{count[$6]++} END {
for (word in count){
if(count[word] > 20000) {
print word,count[word] >> warning
}
print word,count[word]
}
}' log-firewall > temp
```

Inicializamos a variável "warning", e, a cada execução da iteração no awk, se o número de ocorrências for maior que 20000, escrevemos no final do arquivo "warning"o bloqueio e seu valor de ocorrências.

Usando o comando **mail**:

```
echo "Bloqueios:" | mail -s "Alerta" admin@gmail.com -A warning.txt
```

Onde '-s' especifica o assunto do email e '-A' anexa o arquivo.

Exercícios

1. Recorra a um livro e estude os assuntos aqui tratados por meio do mesmo.
2. Estude assuntos além do escopo deste artigo.
3. Crie seus próprios documentos e resolva casos de uso similares aos aqui expostos.

10 Conclusão

Durante este artigo foi explanado as fundações do desenvolvimento do UNIX e do Shell, demonstrando sucintamente como, e por quais motivos, se deram seus processos evolutivos.

Foi explicitado como é possível lidar com diversos problemas a serem resolvidos de maneiras diferentes, devido a versatilidade do Shell, mesmo limitando o escopo dos conceitos aqui ministrados. Além disso, tornou-se evidente o leque de possibilidades que o Shell oferece para tornarmos o ambiente mais adequado e simplificado para o desenvolvimento de tais soluções.

Espera-se que, ao ser confrontado com problemas resolvíveis por meio de funcionalidades do Shell, a leitura aqui realizada sirva de referência para o leitor saber por onde iniciar a resolução do problema.

11 Referências

[1]https://cs.stanford.edu/~miles/vi.html

Cameron Newham; Bill Rosenblatt. Learning the Bash: 3. ed. O'REILLY, fevereiro, 2009.

GNU Bash Manual https://www.gnu.org/software/bash/manual/bash.pdf, 16 de setembro, 2016.

IBM developerWorks https://www.ibm.com/developerworks/library/l-linux-shells/index.html Dennis M. Ritchie; Ken Thompson. The UNIX Time-sharing System. Julho, 1974.

https://people.eecs.berkeley.edu/ brewer/cs262/unix.pdf History of Unix, Wikipedia https://en.wikipedia.org/wiki/History_of_Unix.

R. Schoenacher, S. Freitas, C. Mourthé, Desenvolvimento ergonômico e testes de teclado e mo use acoplados para profissionais da área de Tecnologia da Informação.

Ubuntu Desktop Guide <https://help.ubuntu.com/stable/ubuntu-help/index.html.en>.