BASH Shell

Matheus R. de Camargo[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
04 de Dezembro de 2018

mrc13@inf.ufpr.br

Resumo. *Este artigo apresenta, assim como uma introdução aos sistemas UNIX e ao Shell, um pequeno aprofundamento nestes conceitos. Tópicos como linhas de comando, argumentos, redirecionamento, curingas e ambiente serão abordados, todos estes sendo utilizados em conjunto para proporcionar toda a experiência do shell. Ao final serão apresentados exemplos de aplicações práticas.*

1. UNIX e Shell

1.1. História do UNIX

O UNIX começou a ser desenvolvido no inicio dos anos 70 nos Laboratórios Bell da AT&T, e o seu sucesso resultou em diferentes versões. Pessoas que tinham acesso ao sistema UNIX começaram a fazer suas próprias modificações, o que levou universidades, institutos de pesquisa, órgãos governamentais e companhias de tecnologia a usarem o sistema UNIX para desenvolver várias das tecnologias que hoje são parte do sistema UNIX.

Ao final dos anos 70, todas estas pessoas que tiveram acesso ao UNIX e desenvolveram suas próprias tecnologias começaram a divulgar suas diferentes versões do UNIX que eram otimizadas para suas arquiteturas específicas, cada qual com suas vantagens e desvantagens.

No início dos anos 80 o mercado de sistemas UNIX havia crescido a ponto de chamar a atenção de analistas e pesquisadores. Discussões sobre pontos fortes e fracos dos sistemas UNIX se tornaram constantes, fazendo com que os desenvolvedores constantemente adicionassem novas funcionalidades às suas versões do UNIX.

Apesar do UNIX ainda pertencer à AT&T, a companhia não fez muito até o meio dos anos 80, até que o interesse por um sistema UNIX padronizado se tornou interesse da industria, porém qual versão deveria ser padronizada?

Em meio a esta ”guerra”, em 1991 Linus Trovalds desenvolveu o **Linux**, baseado no UNIX. O Linux foi desenvolvido com a proposta de, não apenas ser um software livre, mas também ser fácil de utilizar e de graça. O Linux é mantido pelo Linus e por uma comunidade de desenvolvedores que auxilia no desenvolvimento de novas versões.

1.2. História dos Shells UNIX

A partir do momento em que o shell se tornou independente de sistemas UNIX vários shells foram desenvolvidos, porém apenas alguns se tornaram amplamente utilizados.

1.2.1. Os Grandes Shells

O primeiro grande shell desenvolvido é conhecido como **shell Bourne**, que foi desenvolvido inteiramente por Stephen Bourne nos Laboratórios Bell e foi distribuido em sistemas UNIX a partir de 1979. Alguns de seus objetivos eram permitir que scripts shell fossem usados como filtros, promover controle sobre a entrada e saída de descritores de arquivo e permitir que strings de qualquer tamanho fossem interpretadas por scripts shell. Este shell é conhecido como *sh* e ainda pode ser encontrado em vários sistemas UNIX no caminho */bin/sh*.

O primeiro shell a ser amplamente utilizado é conhecido como *shell C* (ou *csh*). Este shell foi criado por Bill Joy na Universidade da Califórnia no final dos anos 70.

O *csh* foi modelado na linguagem C, e introduziu várias novas funcionalidades para o trabalho interativo, como mecanismos de histórico e edição. estas funcionalidades do *csh* foram copiadas na maioria dos shells atuais.

Recentemente vários outros shells vêm se tornando populares, dentre eles o **shell Korn**. Este shell incorpora as melhores funcionalidades do *sh* e do *csh*, além de implementar novas funcionalidades. Porém este shell é um produto comercial, ou seja, é pago.

1.3. O Bash

O bash (*Bourne Again Shell*) foi criado para uso no projeto GNU, projeto este que foi criado por Richard Stallman. O intuito deste projeto era criar um sistema operacional compatível com o UNIX, substituindo todas as funcionalidades comerciais do UNIX por novas que fossem distribuídas gratuitamente.

Segundo próprio projeto GNU, "o nome "GNU" foi escolhido porque atende a alguns requisitos; em primeiro lugar, é um acrônimo recursivo para "GNU's Not Unix", depois, porque é uma palavra real e, finalmente, é divertido de falar.".

1.3.1. Funcionalidades

Apesar do **shell Bourne** ser considerado o shell padrão, o *bash* vêm se tornando popular. Além da sua compatibilidade com o *shell Bourne*, o *bash* possui as melhores funcionalidades dos shells C e Korn.

A funcionalidade mais atrativa do *bash* é o modo de edição. Com esta funcionalidade é muito mais prático corrigir erros ou modificar comandos anteriores do que no **shell C**, e no **shell Bourne** isso nem é possível.

2. Conceitos Básicos

2.1. Comandos

Linhas de comando em shell consistem de uma ou mais palavras, que são separadas espaços ou TABs. A primeira palavra da linha é o comando e as palavras restantes (se existirem) são os argumentos do comando.

Por exemplo, a linha de comando **lp arquivo** consiste do comando *lp* e do argumento **arquivo**. No exemplo citado, *lp* irá imprimir o conteúdo do arquivo **arquivo**.

Além de argumentos, linhas de comando também podem possuir opções, que são argumentos especiais que dizem ao comando informações sobre o que ele deve fazer. estas opções geralmente são compostas por um traço seguido de uma letra, como **-h** ou **-a**. O comando **ls -l** contém a opção **-l**, que diz ao comando **ls** para mostrar informações adicionais sobre os arquivos do diretório corrente.

2.2. Arquivos

Argumentos passados pelos comandos não são sempre arquivos, porém arquivos são os objetos mais importantes em qualquer sistema UNIX. Estes arquivos são divididos em categorias diferentes, porém três delas são as mais importantes:

2.2.1. Arquivos normais

Este também são chamados de arquivos de texto. Eles contém caracteres que podem ser lidos, como por exemplo qualquer arquivo com a extensão **.txt**.

2.2.2. Arquivos executáveis

Também conhecidos como programas, estes arquivos são chamados como comandos no formato **./**<**nome-do-arquivo**>.

2.2.3. Diretórios

Diretórios também são um tipo de arquivo. Eles são um arquivo que contém outros arquivos (que podem ou não também serem diretórios).

2.3. Diretórios

O conceito mais importante e diretórios e que eles podem conter outros diretórios, criando uma estrutura hierárquica conhecida como árvore.

O topo desta árvore é conhecido como *raiz*, e não possui nome no sistema. Todos os arquivos podem ser encontrados a partir da *raiz*, e o modo que eles são encontrados é listando todos os diretórios que levam a este arquivo, separando o nome dos diretórios por barra (/). Este é conhecido como o caminho absoluto de um arquivo.

2.3.1. Diretório corrente

Ter que listar o caminho absoluto o tempo todo seria muito inconveniente, por isso também é possível encontrar um arquivo a partir do diretório em que o usuário está "dentro", isso é chamado de caminho relativo. Se o caminho de um arquivo for dado sem uma barra (/) inicial, o caminho de busca será o relativo.

2.3.2. Notação em til (~)

Diretórios *home* são recorrentes em caminhos de arquivos e por causa disso o *bash* tem uma maneira de abreviar estes diretórios: basta preceder o nome do usuário com um til (~). Por exemplo, se o usuário **paulo** pussuir um arquivo **readme.txt** em seu diretório **home**, é possível referenciar este arquivo pelo caminho *~/paulo/readme.txt*. Apesar do caminho ser sido encurtado por meio do til (~), este ainda é um caminho absoluto.

2.3.3. Mudando o diretório corrente

É possível mudar o diretório corrente através do comando **cd**. Este comando leva como argumento o nome do diretório que deseja-se que seja o diretório corrente, e o caminho deste diretório pode ser relativo, absoluto ou conter til (~). Caso não seja passado nenhum argumento, **cd** muda o diretório corrente para o diretório *home*.

Também é possível referenciar um diretório utilizando dois pontos (..), que significa "o diretório pai do diretório corrente". Por exemplo, se o diretório corrente for */home/paulo/livros*, o comando **cd ..** redireciona para o diretório */home/paulo*.

Outra funcionalidade do comando **cd** é **cd -**, que redireciona o usuário para qualquer diretório em que ele estava antes do diretório corrente.

2.4. Curingas

Às vezes é necessário executar um comando em vários arquivos ao mesmo tempo. O exemplo mais comum deste tipo de comando seria o *ls* que, ao ser chamado sem nenhum argumento ou opções, lista o nome de todos os arquivos que começam com ponto (.) no diretório corrente.

Normalmente o comando *ls* é chamado juntamente da opção **-l**, que lista várias informações como o dono do arquivo, tamanho do arquivo e o momento da última modificação, ou *-a*, que também lista arquivos ocultos. Mas às vezes é necessário verificar a existência de um certo grupo de arquivos, e são nestas situações em que os curingas são utilizados.

2.4.1. Interrogação (?)

Este curinga encontra qualquer único caractere. Por exemplo, se um diretório contém os arquivos *arquivo.c*, *arquivo.o* e *arquivo.log*, a expressão **arquivo.?** irá encontrar *arquivo.c* e *arquivo.o*, porém não irá encontrar *arquivo.log*.

2.4.2. Asterisco (*)

Este curinga encontra qualquer string de caracteres. Utilizando os mesmos arquivos do exemplo anterior, a expressão **arquivo.*** irá encontrar *arquivo.c*, *arquivo.o* e *arquivo.log*. Este curinga é amplamente utilizado e muito útil quando se deseja descobrir todos os arquivos de uma determinada extensão em um diretório. Por exemplo, se for desejado encontrar todos os arquivos com extensão **.txt** em um diretório, pode-se utilizar a expressão ***.txt**.

2.4.3. Conjunto ([*conj*])

Este curinga encontra qualquer carácter contido em *conj*. Por exemplo, a expressão **[abc]** irá encontrar as strings **a**, **b** ou **c**. No primeiro exemplo de curingas, **arquivo.[co]** encontraria *arquivo.c* e *arquivo.o*, porém não encontraria *arquivo.log*.

Conjuntos também porém ser utilizados para um vetor de caracteres. Por exemplo, a expressão **[a-z]** encontra todas as letras minúsculas de a até z.

2.4.4. Conjunto Negado ([*!conj*])

Este curinga funciona de maneira oposta ao listado anteriormente. Se a expressão **[abc]** encontra as strings **a**, **b** ou **c**, a expressão **[!abc]** encontra as strings que **não** são **a**, **b** nem **c**.

Analogamente ao exemplo anterior, **[a-z]** encontra todos os caracteres que não são letras minúsculas.

2.5. Expansão

Uma expansão funciona da seguinte forma: um inicio opcional, seguido por strings separadas por vírgula e dentro de chaves, seguido por um final opcional. No exemplo **b**{**ed,olt,ar**}**s**, o caractere **b** é o início, {**ed,olt,ar**} são as strings separadas por vírgula dentro de chaves, e **s** é o final.

Se a expressão acima fosse usada como parâmetro do comando **echo**, ele imprimiria as palavras *beds*, *bolts* e *bars*. Também é possivel utilizar chaves dentro de chaves, como em **b**{**ar**{**d,n,k**}**,ed**}**s**, onde o resultado seria *bards*, *barns*, *barks* e *beds*.

Também é possível utilizar expansões para representar um vetor. **echo** {**2..5**} imprime todos os números de 2 até 5, ou seja, *2 3 4 5*.

Finalmente, expansões podem ser utilizadas juntamente de curingas. O comando **ls *.**{**c,h,o**} mostra todos os arquivos de extensão **.c**, **.h** e **.o** no diretório corrente.

2.6. Entrada e Saída

O sistema de entrada e saída (I/O) do UNIX é baseado em duas ideias simples: primeiro, ele utiliza sequencias de caracteres (bytes); segundo, tudo dentro do sistema que produz ou aceita dados é tratado como um arquivo.

2.6.1. Entrada e Saída Padrão

Cada programa UNIX possui uma maneira de receber entrada, conhecida como *entrada padrão*, uma maneira de produzir saídas, conhecida como *saída padrão*, e uma maneira de produzir mensagens de erro, conhecida como *saída de erro padrão*; elas também são conhecidas como *stdin*, *stdout* e *stderr*, respectivamente.

estas entradas e saídas padrão são configuradas de tal maneira que a entrada padrão é o teclado, a saída padrão e saída de erro padrão são o monitor ou janela; porém, quando necessário, é possível redirecionar estas entradas e saídas para um arquivo.

2.6.2. Redirecionamento de Entrada e Saída

No shell é possível redirecionar entradas e saídas, fazendo com que a entrada de um comando venha de alguma fonte que não seja *stdin*, e que a sua saída vá para algum lugar que não seja *stdout* nem *stderr*. Este redirecionamento é feito por meio dos caracteres $<$ ou $>$.

Neste exemplo será usado o comando *cat*, que copia a entrada para a saída. Por padrão este comando aceita vários arquivos ou a entrada padrão como argumentos de entrada e copia eles para a saída padrão.

Se utilizarmos o comando **cat <readme.txt**, o arquivo *readme.txt* está sendo redirecionado para a entrada do comando *cat*. Neste caso *cat* irá imprimir o conteúdo de *readme.txt*

Também é possível fazer no sentido oposto, o comando **ls >saida.txt** irá listar todos os arquivos no diretório corrente e esta lista será o conteúdo do arquivo *saida.txt*.

Redirecionamentos de entrada e saída podem ser utilizados ao mesmo tempo. Por exemplo, a linha de comando **cat <readme.txt >saida.txt** faz com que o conteúdo de *readme.txt* seja copiado para *saida.txt*. Isso teria o mesmo efeito que **cp readme.txt saida.txt**.

2.7. Pipelines

Além de redirecionar a saída de um comando para a entrada de um arquivo, também é possível redirecionar a saída de um comando para a entrada de outro comando. Isso é feito por meio de um *pipeline*, notado como |.

Um exemplo de aplicação do pipeline seria na linha de comando **ls -l | more**. O que acontece nesta linha de comando é que a saída do comando **ls -l** gera em sua saída informações sobre os documentos do diretório corrente, e estas informações são utilizadas como entrada do comando **more**, que por sua vez cria uma lista mais detalhada.

3. Ambiente

Um ambiente (no contexto de computação) é um conjunto de conceitos que expressam ”coisas”diferentes em um sistema. Por exemplo, no contexto de um estúdio, a mesa de som é um ambiente. Conceitos que envolvem a mesa de som podem incluir iniciar ou reproduzir uma gravação. As ferramentas na mesa de som que são utilizadas para lidar com estes conceitos incluem a própria mesa de som, caixas de som, computador, equipamentos de raque (compressores, equalizadores, etc.). Cada uma destas ferramentas possuem uma característica que expressam como e quando utilizá-las.

É possível customizar este ambiente para que ele se torne mais agradável mudando as coisas de lugar, programando funções na mesa de som, etc. estas customizações são feitas baseadas nas necessidades pessoais de cada usuário.

Similarmente, shells do UNIX também possuem conceitos; estes conceitos podem ser arquivos, diretórios, entradas e saídas, enquanto o próprio UNIX proporciona ferramentas para trabalhar com estes conceitos, como comandos e editores de texto. O bash proporciona algumas ferramentas para customizar este ambiente, porém quatro delas são as mais importantes:

Arquivos Especiais Os arquivos *.bash_profile*, *.bash_logout* e *.bashrc* que são lidos pelo bash quando o usuário inicia ou encerra uma sessão ou inicia um novo shell.

Aliases São sinônimos para comandos ou linhas de comandos que podem ser definidos.

Opções Controla vários aspectos do ambiente que podem ser ligados ou desligados.

Variáveis Valores mutáveis que podem ser referidos pelo seu nome. O shell pode mudar o seu comportamento de acordo com o valor armazenado em certas variáveis.

3.1. Arquivos especiais

No diretório *home* existem três arquivos que possuem um significado especial para o *bash*. Estes arquivos permitem que o usuário configure o seu ambiente automaticamente ao iniciar uma sessão ou invocar um novo *bash* e também realizar comandos quando uma sessão se encerra.

3.1.1. .bash_profile

Este é o arquivo mais importante do *bash*, este arquivo é lido e os comandos contidos nele são executados pelo *bash* todas as vezes em que o usuário inicia uma sessão no sistema. As linhas deste arquivo definem o ambiente básico de quando uma sessão é iniciada.

Para editar este arquivo, basta adicionar novas linhas após as já existentes. Vale ressaltar que as alterações feitas neste arquivo só surtirão efeito ao reiniciar a sessão.

3.1.2. .bashrc

O arquivo *.bash_profile* é executado ao se iniciar uma sessão, porém ao iniciar um *bash* a partir de uma linha de comando, outro arquivo é executado; este arquivo é o *.bashrc*. Esta divisão de arquivos permite que o usuário tenha maior controle ao separar comandos que são necessários ao iniciar uma sessão dos comandos necessários ao iniciar um novo subshell.

3.1.3. .bash_logout

Este arquivo é lido e executado sempre que uma sessão se encerra. Ele é utilizado para concluir o ambiente. Se o usuário deseja executar comandos que removam arquivos temporários ou armazenem o tempo de duração de uma sessão, este é o arquivo que deve ser utilizado.

3.2. *Aliases*

Alguns comandos do *bash* podem ser extensos, e escreve-los sempre pode ser um processo demorado. Felizmente existe uma ferramenta que ajuda os usuários a definirem um comando muito grande em um menor; esta ferramenta se chama *alias*.

Um *alias* pode ser definido da seguinte forma nos arquivos *.bash_profile* ou *.bashrc*:

alias name=command

Esta sintaxe diz que sempre que o usuário digitar **name** como um comando, o comando **command** será executado. Vamos supor que um usuário não goste da língua inglesa e deseje utilizar comandos em português. Este usuário poderia definir o alias

alias gato=cat

E sempre que ele digitasse **gato** como um comando, o comando **cat** seria executado.

Além da substituição de comandos, aliases também são utilizados para simplificação de comandos extensos. Se um usuário tem o costume de sempre acessar o diretório *~/user/documents/ci320/2018-2* este usuário pode criar o alias:

alias cdci320='cd ~/user/documents/ci320/2018-2'

Para poder acessar este diretório digitando apenas o alias **cdci320**.

Também é possível fazer um alias "recursivo", onde o lado esquerdo e o lado direito da atribuição possuem o mesmo comando. De um usuário deseja sempre verificar as permissões, tamanhos, donos, etc. dos arquivos ao realizar o comando **ls**, ele pode criar o seguinte alias:

alias ls='ls -l'

Aparentemente isso criaria um loop infinito, porém o bash garante que o que acontece é exatamente o esperado: sempre que o comando **ls** for chamado, na verdade o que será executado será **ls -l**.

3.3. Opções

Aliases permitem que o usuário crie atalhos para seus comandos, porém não permite que o comportamento do shell seja alterado; para isso são usadas **Opções**. Uma opção é uma configuração do shell que pode estar ligada ou desligada.

Para ligar ou desligar uma opção são utilizados os comandos **set -o** <**opcao**> e **set +o** <**opcao**> que, respectivamente, ligam e desligam a opção. O motivo da ativação e desativação dos comandos serem contraintuitivos com os sinais de mais (+) e menos (-) é porque por padrão opções são utilizadas com um menos (-) na frente, e a implementação do mais (+) só aconteceu posteriormente. Uma lista com várias opções pode ser encontrada **aqui**.

3.4. Variáveis do Shell

Opções são características do ambiente que podem expressadas com opções de ligado/-desligado, porém existem características que não podem ser expressadas desta maneira. Estas características são chamadas de **variáveis**.

Assim como um alias, variáveis são um nome que possui um valor associado. Por convenção variáveis devem ser escritas em caixa alta e são definidas da seguinte forma:

VARNAME=value Note que não existe espaço nem antes nem depois do caractere de atribuição (=), e se o valor possuir mais de uma palavra ele deve estar isolado por

aspas simples (’). Para utilizar uma variável ela deve ser chamada com um dólar ($) antes do seu nome.

Se a variável **MENSAGEM=’Esta é uma mensagem’** for definida, pode-se checar o conteúdo desta variável chamando:

echo ”$MENSAGEM”

O comando acima imprimiria a string ”Esta é uma mensagem”.

3.4.1. Variáveis e Aspas

Note que no exemplo anterior foram utilizadas aspas duplas (”) para cercar a variável no exemplo do echo. Dentro de aspas duplas, alguns caracteres especiais ainda são interpretados, e um destes caracteres é o dólar, por isso o exemplo anterior imprime o conteúdo da variável e não a string ”$VARNAME”.

Supondo que uma variável **$ESPACO=’Quatro espaços’** foi definida. Ao chama-la da maneira **echo $ESPACO** o resultado seria:

Quatro espaços

Porém, ao chama-la da maneira **echo ”$ESPACO”** o resultado seria:

Quatro espaços

Isso acontece pois sem as aspas duplas o shell divide a string em palavras após substituir o valor das variáveis, assim como acontece no processamento de linhas de comando. As aspas duplas fazem com que o shell entenda que todo o conteúdo entre as aspas é uma única palavra.

Em linhas gerais, quando houver dúvida é melhor utilizar aspas simples. A não ser que a string contenha uma variável, neste caso é melhor utilizar aspas duplas.

3.4.2. Variáveis Reservadas

O shell possui algumas variáveis reservadas que possuem funções específicas. **Nesta lista** é possível encontrar elas e seus significados.

4. Manipulação Do Ambiente Gráfico Utilizando Teclado

Hoje, no século XXI, quase todas as casas estão equipadas com algum computador. Computadores estes que são de fácil acesso e entendimento para o usuário comum. Esta acessibilidade se dá por meio não só de dispositivos externos como teclado e mouse, mas também pela forma em que o ambiente de software é estruturado por meio de um ambiente gráfico com cursor, imagens, telas, textos, etc.

Porém houve tempos mais sombrios, tempos estes em que computadores não eram acessíveis para o usuário comum, tempos estes em que os únicos recursos disponíveis para manusear um computador eram uma tela com texto e um teclado, conhecidos como terminais (na verdade terminais são relativamente modernos, houveram tempos muito mais sombrios). Uma das primeiras máquinas criadas com o intuito de atingir um publico mais abrangente foi o terminal TRS-80 desenvolvido pela Tandy Corporation e lançado em 1977.

Apesar das grandes evoluções nas últimas décadas, ainda é possível utilizar um computador moderno de maneira similar à estes terminais antigos. Existem diferentes motivos para um usuário querer fazer isso, apesar de todas as facilidades dispostas hoje em dia: agilidade para o programador, limitação do sistema, limitação física do usuário ou simplesmente pelo sentimento nostálgico. Neste artigo iremos abordar como um usuário pode utilizar um sistema moderno, porém manuseando-o apenas com um teclado.

4.1. Primeiros Passos

Com o sistema devidamente inicializado pode-se dar início a utilização do mesmo. Pra isso, primeiramente deve-se executar o atalho **CTRL+ALT+T**; este atalho fará com que um terminal seja aberto e, por meio deste terminal, será possível executar as tarefas demonstradas neste artigo.

O terminal iniciado provavelmente estará rodando o **bash**, porém isso pode variar de sistema para sistema. Para compreender como utilizar o bash recomenda-se a leitura completa deste artigo, assim como do livro *"Learning The BASH Shell"*. Esta sessão irá tratar sobre a navegação do ambiente gráfico.

4.2. Navegação

4.2.1. Abas

Já se acabou o tempo em que para cada tarefa a ser realizada era necessário iniciar um processo novo e chavear entre os processos por meio de **ALT+TAB**. Agora é possível que hajam diversas abas contidas em um mesmo terminal, podendo designar cada uma para uma tarefa diferente.

Ao se iniciar o terminal não serão mostradas abas, pois haverá apenas uma (a que acabou de ser iniciada). Porém, ao executar o atalho **CTRL+SHIFT+T** uma nova aba será criada, e um atalho todas as abas existentes poderão ser vistas no topo do terminal. Como o objetivo deste artigo é utilizar o ambiente gráfico apenas por meio do teclado, não será possível mudar de aba clicando em seu atalho (porém isto é possível), para isto então será utilizado o atalho **ALT+#**, onde **#** representa um número de 1 a 9, ou pelo atalho **CTRL+PgUp** para mudar para aba anterior e **CTRL+PgDown** para mudar para a próxima aba.

Além de navegar entre as abas, também é possível reordená-las. Isto é feito por meio dos atalhos **CTRL+SHIFT+PgUp** para mover a aba corrente para a esquerda e **CTRL+SHIFT+PgDown** para mover a aba corrente para a direita.

Por fim, pode-se renomear uma aba para qualquer nome de preferência do usuário. Para isto deve-se executar o atalho **CTRL+SHIFT+I**, após isso uma caixa de texto aparecerá para que o novo nome da aba seja inserido. Após inserir o novo nome, deve-se apertar **ENTER** e a aba atual será renomeada.

4.2.2. Fonte

Outro ponto customizável do terminal é o tamanho da fonte. Mudar o tamanho da fonte é desejável em diversas situações, como em caso de haver dificuldade de leitura ou problemas devido a resolução do monitor.

Mudar o tamanho da fonte é tão fácil quanto apertar três botões (até porque realmente é necessário apertar apenas três botões). Para aumentar o tamanho da fonte é utilizado o atalho **CTRL+SHIFT+"+"**. Em contrapartida, para diminuir o tamanho da fonte, utiliza-se o atalho **CTRL+"−"**. Também é possível retornar a fonte para o tamanho padrão sem a necessidade de ficar aumentando ou diminuindo várias vezes, isso é feito pelo atalho **CTRL+SHIFT+")"**.

4.2.3. Copiar Texto

Similar aos atalhos **CTRL+C** e **CTRL+V**, no terminal linux é possível copiar um texto selecionado por meio do atalho **CTRL+SHIFT+C** e colar um texto por meio do atalho **CTRL+SHIFT+V**. Além disso, de maneira similar, é possível recortar um texto por meio do atalho **CTRL+SHIFT+X**.

4.2.4. Busca

Dentro do terminal é possível buscar por uma linha de comando que tenha sido executada anteriormente, podendo até ter sido executada em outra sessão. Esta busca é feita por meio do atalho **CTRL+R**. Ao executar este comando uma linha com o texto *"(reverse-i-search)"* aparecerá e ao lado dela será possível digitar o texto que se deseja ser buscado. Após encontrar o comando desejado, é possível executa-lo pressionando **ENTER**.

4.2.5. Encerrar

É possível encerrar todo o terminal (que, por sua vez, fará com que todas as abas contidas no terminal também sejam encerradas) ou também encerrar apenas uma aba específica. Para encerrar o terminal é utilizado, assim como para qualquer outro processo, o atalho **ALT+F4**, e para encerrar a aba atual pode-se digitar o comando **exit** ou utilizar o atalho **CTRL+SHIFT+W**.

5. Bash Na Prática

5.1. Manipulação De Arquivos .csv

Esta sessão apresenta uma base de dados no formato **.csv** e mostra como, através de comandos do bash, manipula-la. Todas as manipulações poderiam ser feitas "na mão"(ou seja, sem a necessidade de código), porém para arquivos muito grandes ou grande volume de arquivos isso se tornaria inviável.

5.1.1. Base de Dados

A base de dados a ser analisada se chama **patrimonio.csv**. Ela consiste de todos os itens dispostos no departamento de informática da UFPR.

Cada um dos itens consiste de cinco campos: o primeiro é um número de identificação, o segundo e o terceiro são campos de descrição, o quarto corresponde à marca do item e o último corresponde à sua localização.

5.1.2. Objetivo

Tendo em mão esta base de dados, deseja-se criar um diretório para cada localização. Dentro deste diretório deve haver um arquivo **.csv** contendo apenas os itens presentes em **patrimonio.csv** que estão nesta localização.

5.1.3. Passos

5.1.3.1 Remoção de Caracteres Indevidos

Neste arquivo **.csv** é utilizado o caractere **;** para separar os campos. É sabido também, que neste arquivo existem alguns campos que possuem **;** em seu conteúdo. Para que o algoritmo que será implementado funcione, é necessário remover estes **;** indevidos que estão no conteúdo de um campo.

Para fazer isso, utilizaremos o seguinte código:

```
1 #remove_semicolon.sh
2
3 sed 's/[^\"]\;[^\"]/\,/g' patrimonio > patrimonio_mod
```

O código acima utiliza o comando **sed** para remover os **;** indesejados. É sabido que os **;** corretos possuem um caractere **"** antes e/ou depois de sua ocorrência. Por causa disso, o comado **sed** está encontrando todos os **;** que **não** possuem **"** à esquerda e nem à direita e substituindo eles por **,** .

Após isso, um novo arquivo chamado **patrimonio_mod** é criado, contendo o arquivo modificado.

5.1.3.2 Criando Um Arquivo de Texto Com os Locais

O segundo passo é criar um arquivo **.txt** contendo todos os locais que devem ser criados. Para isso será utilizado o seguinte código:

```
1  #create_locais_txt.sh
2
3  OLDIFS=$IFS
4  IFS=";"
5
6  while read id desc1 desc2 marca loc
7  do
8    echo $loc
9  done < patrimonio_mod | awk '!a[$0]++' | sed '/^\s*$/d' |
10 sed 's/\"//g' > locais.txt
11
12 IFS=$OLDIFS
```

Primeiramente, a linha 4 está indicando qual é o caractere que separa os campos no arquivo **.csv** de entrada, que no nosso caso é **;**.

Na linha 6 se inicia um loop que lê uma linha do arquivo **patrimonio_mod** de cada vez e armazena cada campo em uma variável. O campo de interesse é apenas a localização do item, então está sendo feito um **echo** neste campo e o resultado está sendo armazenado em **locais.txt**, porém antes deste armazenamento ocorrer o texto está sendo tratado.

O comando **awk '!a[$0]++'** está removendo todas as linhas duplicadas, fazendo com que cada local seja único. A saída deste comando é utilizada como entrada do co-

mando **sed '/^ \s*$/d** através de um pipeline, fazendo com que linhas vazias sejam removidas do arquivo. Finalmente, a saída deste último comando é redirecionada para a entrada do comando **sed 's/\" / / g '** e os caracteres **"** são removidos para que, finalmente, tudo seja escrito no arquivo **locais.txt**.

5.1.3.3 Criação dos Diretórios

Até agora os caracteres indesejados foram removidos da base de dados e um arquivo de texto contendo os locais foi criado, resta utilizar estes dois arquivos em conjunto para completar a tarefa. O código que faz isso é o seguinte:

```
1 #create_locais_dir.sh
2
3 for i in $(cat locais.txt); do
4   lugar=$i;
5   mkdir $i;
6   cat patrimonio_mod | grep $lugar > "$i.csv"
7   mv "$i.csv" "$i"
8 done
```

O código tem inicio na linha 3 com um loop que executa uma sequencia de comandos para cada linha do arquivo **locais.txt**. Dentro do loop, *$i* representa uma linha do arquivo que, por sua vez, representa um local.

Para cada linha encontrada em **locais.txt**, um diretório de mesmo nome é criado. Após isso, na linha 6, o comando **grep** encontra na base de dados todas as linhas que possuem *$i*, e então copia todas estas linhas para um único arquivo de nome **$i.csv**.

Ao final da execução deste código, haverá um diretório para cada local e, dentro de cada diretório, um arquivo **.csv** correspondente ao local.

5.2. Manipulação De Arquivos E Diretórios I

Este sessão apresenta um conjunto de diretórios e arquivos e mostra como, através de comandos do bash, manipula-los. Todas as manipulações poderiam ser feitas "na mão"(ou seja, sem a necessidade de código), porém para arquivos muito grandes ou grande volume de arquivos isso se tornaria inviável.

5.2.1. Diretórios e Arquivos

Existe um diretório principal chamado **DadosMatricula**. Dentro deste diretório existem vários outros diretórios chamados **CIxxx**, sendo que cada um destes diretórios representa uma disciplina ministrada pelo departamento de informática entre os anos de 1988 até 2002. Dentro destes outros diretório existem vários arquivos que representam as matriculas para esta determinada disciplina em determinado ano/semestre.

Estes arquivos são chamados **xxxxx.dados**, onde os quatro primeiros dígitos são referentes ao ano referente ao arquivo e o quinto dígito é referente ao semestre. Cada linha destes arquivos possui dois campos separados por dois pontos (:). O primeiro campo é referente ao código do curso do aluno matriculado e o segundo campo é referente ao GRR do aluno.

5.2.2. Objetivo

O primeiro objetivo é contabilizar o número de matriculas no departamento de informática semestre a semestre. Para esta contagem deve-se considerar que os GRR's não se repetem (GRR's duplicados devem ser contabilizados apenas uma vez).

O segundo objetivo é acompanhar a evolução do número de matrículas de cada curso de fora do departamento de informática semestre a semestre ao longo do período analisado, por isso neste caso os GRR's podem se repetir pois um mesmo GRR cursa normalmente várias disciplinas.

5.2.3. Scripts

5.2.3.1 Primeiro Objetivo

Para realizar o primeiro objetivo será utilizado o seguinte script:

```
1  #script01.sh
2
3  # DMPATH='/home/bcc/mrc13/nobackup/Shell/Aula_11/DadosMatricula'
4  DMPATH=./DadosMatricula
5
6  find $DMPATH -type f -print0 | xargs -0 sed -i /curso:grr/d
7
8  rm -rf $DMPATH/*.tmp*
9  for file in $(find $DMPATH -name "*.dados")
10 do
11   year=$(echo $file | rev | cut -d/ -f1 | rev | cut -d. -f1)
12   cut -d: $file -f2 >> "$DMPATH/$year.tmp"
13 done
14
15 for file in $(find $DMPATH -name "*.tmp")
16 do
17   cat $file | sort -n | uniq > "$file 2"
18   students=$(wc -l < "$file 2")
19   year=$(echo "$file" | rev | cut -d/ -f1 | rev | cut -d. -f1)
20   echo "$year : $students"
21 done | sort -n > 'ano:matriculas.txt'
22 rm -rf $DMPATH/*.tmp*
```

Como este script foi utilizado em dois ambientes diferentes, o caminho dos arquivos é diferente em cada ambiente, então dependendo do ambiente as linhas 3 e 4 se alternam em qual fica comentada.

Nos arquivos **.dados** existe uma linha escrito "curso:grr"que atrapalharia a execução correta do script, por causa disso a linha 6 do script trata de remover esta linha de todos os arquivos **.dados**. Como este script foi testado várias vezes, talvez existissem arquivos remanescentes de uma iteração anterior que poderia comprometer a iteração atual, por causa disso a linha 8 trata de remover estes possíveis arquivos. Na linha 9 se inicia um lasso que, para cada iteração, um arquivo **.dados** é analisado.

O comando **find $DMPATH –name "*.dados"** encontra todos os arquivos **.dados**. Por ele ser o conjunto em que o comando **for** está sendo executado, para cada iteração a variável **file** será um dos arquivos encontrados pelo comando **find**.

Dentro do lasso a variável **year** recebe o nome do arquivo que está sendo analisado. Como o nome encontrado para o arquivo é referente a todo o caminho, a string referente ao caminho precisa ser tratada para que **year** seja uma string referente apenas ao ano/semestre do arquivo. Este tratamento é feito na própria linha 11.

Como cada linha do arquivo é composto de dois campos separados por dois pontos (:), como o campo de interesse é apenas o segundo (referente ao GRR do aluno), o comando **cut** é executado para que o primeiro campo seja removido. Tendo isso feito, se um arquivo temporário referente ao ano/semestre em análise não existe, ele é criado e o seu conteúdo são todos os GRR's do ano/semestre sendo analisado. Caso este arquivo já exista, é feito um *append* com todos os GRR's do ano/semstre sendo analisado. Tendo isso feito, haverá um arquivo temporário para cada ano/semestre, e o conteúdo de cada um destes arquivos serão todos os GRR's matriculados neste ano/semetre.

A segunda parte do script se da inicio na linha 15 e consiste em remover todos os GRR's duplicados nos arquivos temporários. Novamente o lasso para analisar cada um destes arquivos é feito utilizando o comando **for** em conjunto do comando **find**.

As linhas 17 e 18 removem GRR's duplicados, fazem uma contagem dos GRR's e armazenam este número na variável **students**. Vale ressaltar na linha 17 de o comando **uniq** (que elimina linhas duplicadas) só elimina linhas duplicadas que são consecutivas. Ou seja, antes de utilizar este comando é necessário ordenar o conteúdo do arquivo por meio do comando **sort -n** para que, mais importante que deixar os GRR's ordenados, os GRR's iguais estejam organizados consecutivamente.

O comando executado na linha 19 tem o mesmo objetivo da linha 11. Após isso, haverão duas variáveis: uma chamada **year** contendo uma string referente ao ano/semestre e outra chamada **students** contendo o número de alunos matriculados neste ano/semestre. Então é criado um arquivo chamado **ano semestre.txt** onde cada linha deste arquivo é referente a uma dupla de **year** e **students**.

5.2.3.2 Segundo Objetivo

Para realizar o segundo objetivo será utilizado o seguinte script:

```
1  #script02.sh
2
3  # DMPATH='/home/bcc/mrc13/nobackup/Shell/Aula_11/DadosMatricula'
4  DMPATH=./DadosMatricula
5
6  rm $DMPATH/*.tmp
7  for file in $(find $DMPATH -name "*.dados")
8  do
9    filename=$(echo $file | rev | cut -d/ -f1 | rev)
10   cat $file | cut -d: -f1 >> "$DMPATH/$filename.tmp"
11 done
12
13 for file in $(find $DMPATH -name "*.tmp")
14 do
15   sort -n $file | uniq -c | sed 's/\([0-9]\) \([0-9]\)/\1\:\2/g' > "
       $file 2"
16   cp "$file 2" "$file"
17   rm -rf "$file 2"
18 done
```

O segundo script é análogo ao primeiro script, porém na linha 10 são selecionados os cursos ao invés dos GRR's. Além desta diferença, nas linhas 15 e 16, ao invés de ser contabilizado o número de GRR's, são contabilizados o número de matriculas de cada curso por meio dos comandos **sort** e **uniq**. A opção **-c** do comando **uniq** faz com que sejam contabilizados o número de ocorrências de uma determinada linha e este número é retornado em conjunto da linha analisada. Por exemplo, se existe um arquivo com o conteúdo:

```
04
04
04
03
03
02
02
02
02
01
```

O comando **uniq -c**, tendo o arquivo acima como entrada, irá retornar:

```
3  04
2  03
4  02
1  01
```

Ao final do script haverá um arquivo **.tmp** para cada ano/semestre e os arquivos estarão no formato acima. O valor da direita representa o código do curso e o valor da esquerda representa o número de matrículas deste curso em determinado ano/semestre.

5.3. Manipulação De Arquivos E Diretórios II

Esta sessão apresenta um conjunto de arquivos e mostra como, através de comandos do bash, manipula-los para analisar os bloqueios em um log de um firewall. Todas as manipulações poderiam ser feitas ”na mão”(ou seja, sem a necessidade de código), porém para arquivos muito grandes ou grande volume de arquivos isso se tornaria inviável.

5.3.1. Arquivos

Existem dois arquivos que são utilizados em conjunto para a análise. O primeiro, e mais importante deles, é o arquivo **log-firewall.xz**, que é um arquivo compactado de um log de um firewall. Cada linha deste log corresponde a um bloqueio realizado pelo firewall. Cada bloqueio possui vários campos separados por um espaço em branco que informam coisas como a data e hora do bloqueio, o tipo do bloqueio, dentre outras. Para os fins desta atividade, o campo de interesse é o sexto campo, que corresponde ao tipo de bloqueio.

O segundo arquivo se chama **tipos-bloqueios** e contém uma lista com todos os tipos possíveis de bloqueios. Esta lista será utilizada em conjunto do arquivo de log para identificar o número de ocorrências de cada bloqueio no log.

5.3.2. Objetivo

A partir dos dois arquivos fornecidos (log do firewall e os tipos de bloqueios) deve-se gerar um arquivo onde cada linha consiste de um tipo de bloqueio, seguido da quantidade de vezes em que este bloqueio ocorreu no log. Deseja-se também que este arquivo gerado esteja com os bloqueios no protocolo IPv4 e IPv6 estejam separados. Além disso, sempre em que houverem mais de 20000 (vinte mil) ocorrências de um determinado tipo de bloqueio, deve-se enviar um e-mail para o administrador listando todos os bloqueios que ultrapassaram este número.

5.3.3. Scripts

Para realizar este objetivo foram criados dois scripts, sendo que o primeiro script chama o segundo durante a sua execução. O primeiro script trata de estruturar e enviar o e-mail e estruturar o arquivo a ser gerado, além de chamar o segundo script.

```
1  #dummy.sh
2
3  NOW="$(date +'%d-%m-%Y')"
4  TMP="output.tmp"
5
6  rm -rf *.tmp
7  echo 'Os seguintes tipos de bloqueios foram detectados mais de 20000 (
       vinte mil) vezes no log do dia ' $NOW $'\n' > email.tmp
8
9  cat tipos-bloqueios | xargs -P 0 -n 1 ./script.sh
10
11 cat email.tmp | mail -s "Log do dia $NOW" mrc13@inf.ufpr.br
12
13 echo '*** V4 ***' > log-$NOW.txt
14 grep -v 'V6*' $TMP >> log-$NOW.txt
15 echo $'\n''*** V6 ***' >> log-$NOW.txt
16 grep 'V6*' $TMP >> log-$NOW.txt
17 rm -rf *.tmp
```

As linhas 3 e 4 deste script são declarações e atribuições de variáveis que contém, respectivamente, a data do sistema (dia/mês/ano) e um arquivo de saída temporário. A linha 7 inicia a estruturação do e-mail.

A linha 9 é a parte principal deste script, onde o script principal é chamado. Para isso é feito um **cat** nos tipos de bloqueios e a saída deste comando é utilizada como entrada do comando **xargs**. As *flags* **-P 0** e **-n 1** são utilizadas para que, respectivamente, o script chamado seja executado em *multi thread* com o maior número de *threads* possíveis e que apenas uma linha da entrada (saída do comando **cat**) seja lida de cada vez.

Após a execução do script principal, resta somente enviar o e-mail e terminar de estruturar o arquivo de saída. Na linha 11 o e-mail é enviado para *mrc13@inf.ufpr.br* (que é o meu e-mail, mas pode ser enviado para qualquer outro). Nas linhas 13 até 16 é terminada a estruturação do arquivo de saída, separando os bloqueios IPv4 dos IPv6 por meio do comando **grep**.

Para a execução destas linhas é utilizado um arquivo temporário que contém os bloqueios encontrados e número de ocorrências deles que foi preenchido no script principal. Na linha 14, por meio da flag *-v*, são selecionadas todas as linhas que não começam com "V6", ou seja, todas as linhas que contém bloqueios do tipo IPv4, e as coloca no começo do arquivo de saída. Na linha 16 é feito a mesma coisa, porém desta vez são selecionadas as linhas que começam com V6 e elas são colocadas no final do arquivo de saída.

Ao final da execução deste script, na linha 17, são removidos todos os arquivos temporários que foram gerados durante a execução do script.

O segundo script trata de escrever os bloqueios e número de ocorrências no arquivo de saída e de adicionar um bloqueio ao e-mail a ser enviado caso ele ocorra mais

de 20000 (vinte mil) vezes.

```
1  #script.sh
2
3  TMP="output.tmp"
4  TYPE=$1
5
6  NUM=$(xzcat log-firewall.xz | grep -c $TYPE)
7  echo "$TYPE : $NUM" >> $TMP
8  if [ "$NUM" -gt "20000" ]
9  then
10   # adiciona $TYPE para uma lista
11   echo $TYPE >> email.tmp
12 fi
```

Este script está sendo chamado como argumento do **xargs** no script anterior, e é necessário utilizar a saída do **cat** utilizado no último script (que corresponde ao tipo de bloqueio) neste script. Para isto, basta utilizar a variável $1, que corresponde ao primeiro argumento do **xargs**. Isso está ocorrendo na linha 4.

Na linha 6 a variável NUM recebe o número de ocorrências de determinado bloqueio. Isso é feito por meio do comando **grep -c**, que retorna a quantidade de linhas que contém determinado padrão (neste caso, o padrão é o bloqueio desejado). Após isso, na linha 7, o bloqueio e o número de ocorrências deste bloqueio é escrito no arquivo de saída.

Nas linhas 8 até 12 é checado se determinado bloqueio ocorreu mais de 20000 (vinte mil) vezes e em caso positivo este bloqueio é adicionado ao e-mail a ser enviado.

5.4. Conclusão

Os exemplos que foram demonstrados são apenas um pequeno exemplo do poder do shell e são apenas uma maneira de resolver cada problema, cada um dos problemas poderia ter sido resolvido de diversas maneiras diferentes. Quando se deve fazer um trabalho repetitivo, principalmente em muitos arquivos ou arquivos muito extensos, o shell é uma ferramenta muito poderosa.

6. Referências

Learning The BASH Shell, Cameron Newhan and Bill Rosenblatt, 3rd edition, O'Reilly, 2009

http://www.unix.org/what_is_unix/history_timeline.html - The UNIX System, Hystory and Timeline

https://www.fossmint.com/difference-between-unix-and-linux/ - What is the Difference Between Unix and Linux?

https://www.bsdcan.org/2015/schedule/events/612.en.html - Stephen Bourne Keynote for BSDCan 2015

http://unix.harley.com/instructors/timeline.html - Harley Hahn, Harley Hahn's Guide to Unix and Linux

https://www.gnu.org/gnu/gnu-history.html - GNU Operating System

https://askubuntu.com/questions/506510/what-is-the-difference-between-terminal-console-shell-and-command-line - What is the difference between Terminal, Console, Shell, and Command Line?

https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html - GNU, 4.3.1 The Set Builtin

http://wiki.bash-hackers.org/syntax/shellvars - Bash Hackers Wiki, Special parameters and shell variables

https://en.wikipedia.org/wiki/TRS-80 - Wikipedia, TRS-80