Introdução a sistemas Unix e Shell

Rafael Sanfelice Castilho
rsc15@inf.ufpr.br

11 de Agosto de 2018

Abstract

Este material foi escrito no decorrer de uma disciplina universitária focada em programação em Shell e sistemas Unix, com o intuito de possivelmente ser usado por outros alunos que possivelmente cursem a disciplina no futuro

Resumo

Este material foi escrito por um aluno do curso de ciência da computação na universidade federal do paraná (UFPR), no decorrer de uma disciplina focada em programação shell em sistemas UNIX, escrito com o intuito de ajudar futuros alunos que cursem a mesma disciplina ou disciplinas similares.

O material aborda de forma abrangente a história do shell e do UNIX, o que são, como surgiram, qual sua importância na computação. Assim como diversas funcionalidades e particularidades dos sistemas UNIX como a estrutura de arquivos, atalhos, o que são processos e como manipulá-los, como customizar seu sistema. Terminando por fim em alguns exemplos de problemas reais que são resolvidos com a utilização de scripts em shell.

Sumário

Capítulo 1

Introdução

Tanto UNIX como Shell são ainda hoje em dia partes importantes da computação, e como tal consideramos que seja importante ter uma noção básica do que são e como surgiram, logo, escrevemos aqui a respeito disto, com a expectativa de que após ler este breve texto, o leitor tenha pelo menos um pouco de entendimento sobre o que são UNIX e Shell.

O sistema UNIX foi criado em 1969 por Ken Thompson, tendo em vista a criação de um novo sistema operacional (SO) após a falha na criação do sistema Multics e a saída de um dos colaboradores do projeto por divergências de objetivos [6].

Feito originalmente na linguagem assembler, em 1973 foi reescrito na linguagem C, fato parcialmente responsável por dar aos sistemas UNIX facilidade de ser portado para diferentes máquinas.

Seguindo o desenvolvimento de distribuições maiores, e a facilidade de portar o sistema para diferentes tipos de máquinas provida pela linguagem C, os fabricantes de computadores começaram a adotar o sistema UNIX.

Hoje em dia UNIX se refere a grande família de sistemas operacionais que foram feitas com base no sistema UNIX original (como por exemplo, Solaris, IRIX, AIX, HP-UX, Tru64).

Apesar das similaridades com sistemas UNIX, GNU/Linux não é considerado parte de família UNIX por não partilhar do código fonte de sistemas da família UNIX, e também por possuir um objetivo e uma filosofia de desenvolvimento diferente dos seguidos ainda hoje pelos sistemas UNIX.

Sistemas UNIX são classificados como sistemas baseados em arquivos, Multitarefa e Multiusuários, ou seja, para um sistema UNIX tudo são arquivos (isto sera explicado em maior detalhes no capítulo 2.2), capaz de suportar diversos usuários na mesma máquina, e capaz de rodar múltiplas tarefas simultaneamente (ou pelo menos alternar as tarefas que são realizadas de tal forma que os usuários tenham a impressão que múltiplas tarefas estão sendo executadas simultaneamente).

UNIX também foi um dos primeiros sistemas a tornar o sistema operacional independente da interface de usuário, no caso do UNIX esta interface é o Shell,

que sera discutido em mais detalhes na seção 2.

Capítulo 2

Shell

Como dito no capítulo 1, UNIX foi um dos primeiros sistemas a separar a interface do usuário independente do sistema operacional, o nome dado a família de interfaces usadas por sistemas UNIX hoje em dia é Shell.

É difícil falar de Shell sem mencionar a existência de comandos, porém estes serão tratados na seção 2.1, e esta seção tratará mais da história dos Shells em geral.

O primeiro Shell UNIX foi o thompson Shell, criado por Ken Thompson a partir do Shell Multics, que era baseado no programa RUNCOM([5]).

O primeiro shell de grande porte foi o Bourne Shell (sh) escrito por Stephen Bourne em 1977, tendo como objetivos primários permitir que scritps fossem usados como filtros, incluir ao shell uma capacidade de ser programável, que incluísse variáveis e funções, controle de entrada, saída e sinais mandados para comandos e scripts, e um mecanismo de ambiente, que permitia caracterização de contexto para os comandos[2].

Algumas funcionalidades interessantes introduzidas pelo sh foram:

- Invocação de scripts a partir de seus nomes;
- Execução síncrona e assíncrona de comandos;
- Redirecionamento de entradas e saídas de comandos, assim como a criação de pipelines (que serão vistos com mais detalhes na seção 2.5);
- Scripts não precisam ser compilados para rodar;
- Laços “for” e checagem de condições “case”;
- Variáveis de ambiente;
- comandos “test”, “echo”, “unset” e “type”

Em 1978 Bill Joy, um estudante da Universidade da Califórnia em Berkley, criou o Cshell (csh ou tcsh como passou a ser chamado após receber algumas melhorias), com o objetivo de criar um Shell mais “amigável” ao uso interativo

com o usuário, resultando em um Shell que se aproximava muito em aparência a um programa escrito em C (logo o nome)[3].

Entre as diversas funcionalidades adicionadas ao csh para torná-lo mais "amigável" são notáveis:

- Histórico de comandos e atalhos que permitem fácil reutilização de comandos que já foram usados;
- Criação de apelidos (aliases) para comandos;
- notação "~" para se referir ao diretório "home" de um usuário (mais detalhes em 2.2.1);
- Sugestão de possíveis formas de completar um caminho;
- Coringas (mais detalhes na seção 2.3);
- execução em segundo plano de comandos.

Apesar de ser reconhecido o fato que a interface em csh era melhor do que em sh, devido a diversos outros problemas com a própria estrutura do CShell, incluindo problemas de sintaxe, falta de algumas funcionalidades entre outros problemas na implementação, Bourne Shell ainda acabou sendo considerado superior em todos os demais aspectos para comparação.

Uma variante de Bourne Shell que contribuiu para o insucesso do CShell foi o Korn Shell (ksh), escrito por David Korn em 1983 para ser um meio termo entre sh e csh.

A implementação do Korn Shell partiu do sh, mas puxou inspiração de outros lugares, controle do fluxo de trabalhos, apelidos e históricos (que só foram adicionados ao sh em 1989[4]), foram baseados no Cshell por exemplo.

Um dos maiores e mais difundidos Shells de hoje em dia é definitivamente o Bourne Again Shell (BASH, nomeado em piada com o Bourne Shell), escrito em 1988 por Brian Fox, com o intuito de ser um Shell escrito em código aberto poderoso o suficiente para ser capaz de executar comandos de qualquer Shell já existente, para ajudar na criação de um sistema implementado completamente em GNU([1]).

Baseando-se no CShell e no Korn Shell, Bash se aproveitou de diversas características destes dois como, histórico de comandos, edição de comandos do histórico, variáveis "$RANDOM" e "$PPID", e adicionou muitas outras como, edição interativa na linha de comando , avaliação de expressões aritméticas sem o uso de outro programa, melhoramentos no redirecionamento de entrada e saídas de programas(seção 2.5.1), expansão de chave (seção 2.4).

Vieram ainda diversas outras versões de Shell após estas, como:

- Zshell, que tinha como objetivo ser uma versão melhor do sh, com algumas funcionalidades de Csh, Ksh e bash;
- PowerShell, feito para sistemas microsoft;

- BeanShell, feito para parecer código java;
- Friendly Interactive Shell (Fish), que foi projetado para ser mais amigável com os usuários.

Mais variantes de Shell podem ser encontradas e comaparadas em https://en.wikipedia.org/wiki/Comparison_of_command_shells

2.1 Comandos

Uma linha de comando é uma sequência de palavras separadas por espaços em branco, a primeira palavra sendo o comando propriamente dito, e o resto, quando existir, sendo os argumentos do comando.

Argumentos dão algumas informações importantes para os comandos, como onde executar o comando (comando ls por exemplo) ou origem e destino de arquivos (comandos cp e mv por exemplo).

Alguns argumentos especiais que dão informações mais específicas ao comando são chamados de opções, e geralmente são precedidas de um "-", como é o caso com a opção "-v", que muitos comandos possuem e geralmente ou indicam a versão do comando, ou mandam o comando executar em modo verboso.

Em alguns casos opções podem possuir argumentos próprios, como é o caso com a opção "n" nos comandos "head" e "tail" que necessitam como argumento um número inteiro dizendo aos comandos quantas linhas devem ser processadas.

2.1.1 Comandos Úteis

Alguns comandos que podem aparecer em alguns dos exemplos que serão dados nas seções seguintes, e que são extremamente práticos de se usar no dia a dia são:

- man [comando] - Mostra uma página de manual para [comando] contendo uma descrição detalhada de seu funcionamento, as opções que [comando] pode utilizar e uma descrição detalhada do que a opção faz;
- cat - Copia entrada para saída;
- grep [PADRÃO] [arquivo] - Procura por [PADRÃO] em [arquivo];
- sort [arquivo] - Ordena as linhas de [arquivo];
- cut [arquivo] - Extrai determinadas colunas de [arquivo];
- sed - Editor para filtrar e editar arquivos;
- tr [set1] [set2] - Traduz [set1] para [set2], ou deleta [set1];
- head [arquivo] - Exibe o começo de [arquivo];

- tail [arquivo] - Exibe o fim de [arquivo];
- echo [linha] - Exibe [linha] ;
- ls [arquivo] - Lista os conteúdos de um diretório;
- cd [caminho] - troca do diretório atual para [caminho];
- wc [arquivo] - Exibe o número de linhas, palavras e bytes de [arquivo];
- pwd - Exibe posição do usuário na árvore de arquivos;
- mkdir [nome] - cria um diretório chamado [nome];
- touch [nome] - cria um arquivo chamado [nome];
- rm [arquivo] - remove [arquivo] (este arquivo é removido do sistema, não existe "lixeira" para o rm);
- rmdir [diretório] - remove um diretório, desde que esteja vazio.

Frequentemente [arquivo] pode ser também uma lista de arquivos, ou arquivo nenhum.

2.2 Arquivos e Diretórios

Como dito na seção 1, sistemas UNIX são ditos sistemas baseados em arquivos, logo a importância de arquivos em um sistema UNIX é facilmente percebida.

Arquivos podem conter informações dos mais diversos tipos, e baseado nesta informação eles são separados em alguns tipos diferentes:

- Arquivos regulares - Arquivos de texto que possuem caracteres legíveis, como arquivos ".txt" ou arquivo de código fonte por exemplo;
- Arquivos executáveis - Ou programas são arquivos que podem ser invocados como comandos, não necessariamente legíveis por humanos, todos os comandos detalhados acima podem ser achados em arquivos guardados no sistema, scripts e outros arquivos como arquivos de office e pdfs entram nesta categoria também(scripts sendo, normalmente, os únicos humanamente legíveis quando abertos em um editor de texto);
- Diretórios - Arquivos especiais que podem conter outros arquivos, incluindo outros diretórios (que passam a ser chamados de subdiretórios)

Diretórios sendo capazes de conter outros diretórios é a base para a existência da chamada "Árvore de diretórios" nos sistemas UNIX.

Todos os sistemas UNIX contém um diretório no topo da hierarquia chamado de diretório raiz (ou root) denotado pelo nome "/"(barra), do qual partem todos os outros diretórios que existem no sistema.

Entre os subdiretórios do diretório raiz, alguns dos mais notáveis são os diretórios “bin” e “sbin” que contém os binários executáveis importantes do sistema, incluindo os comandos mencionados acima, o diretório “lib” que contém as bibliotecas do sistema, os diretórios “media” e “mnt” que são usados para montar mídias externas como pendrives, o diretório home (também chamado de users ou outro sinônimo em outros sistemas), que contém os diretórios pessoais dos usuários do sistema.

Diretório Pai é o nome dado ao diretório que contém outros diretórios, por exemplo, os diretórios “bin”, “sbin”, “lib”, “usr” e “home” têm como pai o diretório “/” (raiz), ou seja eles são filhos de “/”. “/” é o único diretório que não possui um diretório pai.

Todo e qualquer diretório em sistemas UNIX possuem dois subdiretórios especiais, que são criados junto com o diretório, são os diretórios “.” (ponto) e “..” (ponto ponto), “.” se refere ao próprio diretório, enquanto “..” se refere ao pai do diretório, no caso da raiz “..” se refere a própria raiz.

2.2.1 Caminhos

Como visto anteriormente o comando “cd” permite ao usuário navegar pelos diretórios usando [caminho], este [caminho] nada mais é do que a localização do diretório na árvore de diretórios, existem dois tipos de [caminho]:

- Caminho absoluto - Caminho partindo do diretório raiz (“/”) até o diretório alvo, o caminho “/home/bugu/Downloads” leva até a pasta “Downloads” do usuário “bugu” caso ele exista no sistema;
- Caminho relativo - É fácil de imaginar que é cansativo ficar constantemente escrevendo o caminho absoluto toda vez que se quer mudar de diretório, o caminho relativo existe para sanar este problema, basta não colocar a “/” no começo do caminho que o sistema irá interpretar o caminho a partir do diretório em que o usuário se encontra (“.”).

Como visto na seção 2, a partir do Cshell, foi introduzida uma notação muito útil ao shell usada para se referir a pasta“home” de um usuário, a notação “~”, ou seja, `~bugu` é outro jeito de se referir a pasta “home” do usuário bugu, enquanto que apenas “~” se refere a pasta “home” do usuário que está usando o sistema.

Outra notação incluída que facilita navegar pela árvore de diretórios é usar o ”-” para se referir ao último diretório em que o usuário estava antes de mudar para o diretório corrente, ou seja se deseja retornar para o último diretório visitado basta usar o comando “cd -”.

2.3 Coringas

Coringas (ou caracteres coringa) são caracteres especiais usados pelo sistema, que como sua contraparte nos jogos de baralho, podem ser usados para substituir outros caracteres.

O coringa mais básico de todos é o "?" (ponto de interrogação), ele pode ser usado para substituir qualquer caractere, mas apenas um único caractere por ponto de interrogação (Nenhum caractere ainda é uma expansão válida para um "?"). Suponha um sistema que possui os usuários ed, edd, eddy e edgy, o comando "ls e?" vai listar apenas o diretório de ed, pois a única expansão válida para "?" no momento é ed, por outro lado o comando "ls ed?y" pode substituir o "?" tanto por "g" quanto por "d", então lista ambos os diretórios eddy e edgy.

Um coringa muito mais poderoso, e que requer muito mais cuidado ao ser usado é o "*" (asterisco), que é expandido em qualquer número de caracteres, por exemplo, "*.txt" se refere a todos os arquivos que terminam em ".txt" independente do nome, o comando "cat as*" irá exibir os conteúdos de todos os arquivos que começam em "as", e o comando "cat t*rs" irá exibir os conteúdos de todos os arquivos cujos nomes comecem com t, tenham algum número arbitrário de caracteres (novamente, nenhum caractere ainda é uma substituição válida) e terminem em rs.

Existe ainda um outro coringa, o conjunto (também chamado de set ou expansão de chaves), porém como ele se enquadra melhor com o conceito de expansão este coringa será abordado na seção 2.4.

Deve ser notado que o sistema expande os coringas antes de executar comandos, logo durante a execução de um comando o sistema possui apenas uma lista de argumentos que foram expandidos pelos coringas, sem saber como ela foi originada, logo cuidado é extremamente necessário ao se usar coringas.

Para usar caracteres coringas como eles próprios e não coringas basta "escapá-los", ou seja colocar uma "\" (contra barra) na frente do caractere coringa desejado.

2.4 Expansões

Como dito na seção 2.3 existe em sistemas UNIX o conceito de expansão, comumente usado em conjunto com caracteres coringa, os três tipos de expansão são:

2.4.1 Expansão de chaves

Expansão de chaves (ou [set], ou conjunto) é uma lista de caracteres, uma coleção de caracteres ou uma combinação de ambos, [abc] é um conjunto que significa, a ou b ou c, ou seja bo[abc]a sera expandido para boaa, boba e boca. [a-z] é um uma coleção de caracteres que pega todos os caracteres entre a e z para a expansão. Uma "!" (exclamação) pode ser usada para negar uma expressão, ou seja [!abc] irá fazer todas as expansões que não contenham os caracteres a, b ou c.

Para usar o caractere "-" em um set basta que ele seja o primeiro ou último caractere do set, já para usar "!" basta que ele não seja o primeiro caractere do set ou seja escapado.

Usar a notação de coleção de caractere é útil, mas não devem ser assumidas muitas coisas sobre os caracteres que serão encaixados na coleção, coleções como "a-z", "A-Z" ou "0-9" ainda são seguras e podem ser usadas sem problemas, mas coleções como "a-Z", "0-t" ou "A-r" não tem seu funcionamento garantido e podem incluir mais ou menos caracteres do que o esperado devido a forma com que o sistema lida com a expansão.

2.4.2 Expansão de caminho

Expansão de caminho é o nome dado ao uso de coringas e outras expansões no nome de um caminho no sistema como por exemplo o comando "cd Do*" executado a partir do diretório "home" de um usuário pode se referir aos diretórios que comecem com as letras "D" e "o", como "Downloads" e "Documentos".

2.4.3 Expansão de colchetes

Expansão de colchetes é um conceito próximo da expansão de caminho, enquanto a expansão de caminho usa coringas para expandir em nomes que existem, a expansão de colchetes expande para uma palavra arbitrária formada por um preâmbulo (opcional), uma sequência de palavras separadas por vírgula, e um preâmbulo (opcional). Por exemplo ca{sa,lha,da} será expandido para as palavras "casa", "calha" e "cada" cada palavra dentro dos colchetes é substituída e adicionada aos preâmbulos e preâmbulos (quando existirem).

É possível também colocar expansões de colchetes dentro de outras expansões de colchetes.

Também é possível usar números separados por ".." para indicar uma sequência de números, e um outro ".." opcional para indicar o incremento desejado, 1..10 irá resultar em uma sequência de números de 1 até o 10, enquanto 1..100..10 irá resultar em uma sequência dos números de 1 a 100, pulando de 10 em 10 números. O mesmo se aplica à sequências de caracteres, salvo o opcional de alterar o incremento.

Em todos os casos de expansão, caso algum erro ocorra, ao invés de terminar o comando com erro, simplesmente é usado o literal dela, ou seja, se algum erro ocorrer durante a expansão de "{oi,ola}" o comando usará "{oi,ola}" ao invés de oi e ola separadamente.

2.5 Entrada e saída

O esquema de I/O (entrada e saída) em sistemas UNIX se baseia em duas ideias simples, Todas as operações de I/O são feitas com sequências arbitrariamente longas de caracteres, e tudo no sistema que processa I/O é considerado como um arquivo.

Todos os programas em UNIX tem uma forma de aceitar entradas, a chamada entrada padrão (standard input ou stdin), uma forma de produzir saídas,

a saída padrão (standard output ou stdout), e uma forma de exibir erros, a saída de erro (standard error output ou standard error ou sdterr).

Shells no geral usam por padrão três configurações para entrada e saída, a entrada padrão é o teclado, a saída padrão é a tela e e a saída de erro também é a tela, por exemplo os comandos são inseridos usando o teclado, e os comandos "ls" e "echo" mostram seu resultado na tela, incluído as mensagens de erro. Lembrando que tanto o teclado como o monitor são tratados como arquivos nos sistemas UNIX.

2.5.1 Redirecionamento

É possível redirecionar, conforme o necessário, a entrada e saída padrões de um programa para outros arquivos.

Caso se deseje usar um arquivo já existente como entrada para um programa usa-se a notação " [comando] < [arquivo]", faz com que a entrada de [comando] seja [arquivo] ao invés do teclado.

De forma similar as notações "[comando] > [arquivo]" e "[comando] >> [arquivo]" podem ser usadas para redirecionar a saída de [comando] para arquivo, a diferença entre os métodos sendo que ">" sobrescreve [arquivo], enquanto ">>" concatena a saída de [comando] ao fim de [arquivo].

O redirecionamento da saída de erro funciona da mesma forma, com a diferença que as notações são "2>" que redireciona apenas a saída de erro e "&>" que redireciona a saída padrão e a de erros para o mesmo lugar.

2.5.2 Pipelines

Também é possível redirecionar a saída de um comando para a entrada de outro comando ao invés de um arquivo. Para isso usa-se a chamada "|" (pipe), qualquer linha de comando contendo mais de um comando separados por "|" é chamada de pipeline.

Pipelines podem ser combinadas com redirecionamento de I/O e podem acabar ficando complexas por isto.

Comandos podem ser separados por ";" (ponto e vírgula), para serem executados em sequências, porém sem terem suas entradas e saídas interferindo umas com as outras.

Redirecionamento de I/O e pipelines suportam a arquitetura UNIX, são notações extremamente poderosas e eliminam a necessidade de arquivos temporários para guardar resultados intermediários entre comandos.

2.6 Conclusões

Existem ainda muitos conceitos de sistemas Unix a serem explorados, e que não foram abordados neste capítulo por não se enquadrarem no objetivo de discussão presentes neste capítulo, assim como muito mais versões de Shell que

foram deixadas de fora por ser desnecessário abordá-las para dar aos leitores um entendimento básico de Shell.

2.7 Exercícios de Shell

Dica: o comando man é seu melhor amigo para ver que comandos usar para resolver os problemas apresentados abaixo.

1. Acesse o link https://www.letras.mus.br/hinos-brasileiros/hino-nacional-brasileiro/, e sem usar editores de texto, copie os hino nacional para um arquivo chamado Hino.txt, e crie uma cópia do arquivo chamada Hino2.txt.

   R:

   cat Hino.txt > ENTER

   Colar hino; ENTER

   CTRL-D

   cp Hino.txt Hino2.txt

2. No arquivo Hino.txt acima troque todas as letras maiúsculas pelo numero 3, e todas as letras minúsculas por letras maiúsculas.

   R:

   cat Hino.txt | tr [A-Z] 3 | tr [a-z] [A-Z] > Hino.txt

3. Usando Hino2.txt substitua todas as ocorrências da palavra Brasil nas 20 primeiras linhas por lisarB, nas 20 linhas do meio por BR e nas 18 ultimas linhas por Pais.

   R:

   cat Hino2.txt | head -n 20 | sed -i s/Brasil/lisarB/g;

   cat h.txt | head -n 40 | tail -n -20 | sed -i s/Brasil/BR/g;

   cat h.txt | tail -n 18 | sed -i s/Brasil/Pais/g

Capítulo 3

Utilizando o computador sem mouse

O mouse, apesar de útil para o usuário comum interagir com o computador, não é necessário para se usar um computador, tudo pode ser feito através do teclado.

Partindo do básico pra um programador, para abrir um terminal Shell, existe um atalho padrão no teclado, $ctrl + alt + t$ que abre diretamente o terminal.

Uma outra forma que pode ser usada para abrir quase qualquer aplicação é utilizar $alt + f2$ para abrir uma linha de comando onde pode-se entrar o nome do programa que se deseja abrir, gnome-terminal para o terminal padrão do gnome, firefox para o navegador, e assim por diante.

Por vezes durante a utilização de um terminal pode ser desejável ter outro terminal, neste caso é possível abrir uma nova aba no terminal que possui um Shell independente com o atalho $ctrl + shift + t$. Para navegar nas abas do terminal o processo é similar a navegar em abas de navegadores web que muitos devem estar mais acostumados a usar, $alt + numerode1a0$ (com 0 estando no lugar do 10) permite pular para as abas de 1 a 10, caso existam mais de 10 abas basta utilizar $ctrl + pagedown$ para mudar para a aba a direita da aba corrente, e $ctrl + pageup$, para mudar para a aba a esquerda da aba corrente. Para fechar a aba quando ela não for mais necessária basta utilizar $ctrl + shift + w$.

Alguns atalhos no teclado se aplicam a quase todos os programas que seja lançados, como $ctrl+$ para aumentar a fonte, $ctrl-$ para diminuir a fonte, a tecla menu (ou $shift + f10$) possui o mesmo efeito que pressionar o botão direito do mouse, utilizando-se das setas para navegar nos menus, e da tecla enter para "clicar" na opção selecionada.

Navegar na internet sem mouse, apesar de difícil não é impossível, com uma combinação de tabs para passar pelos campos clicáveis da janela, a tecla de menu e a tecla enter para "clicar" e da forma de navegar abas mostrada acima, permitem ao usuário atingir qualquer pagina que conseguiria com o mouse com apenas um pouco mais de esforço.

As vezes quando se tem janelas demais sendo usadas pode ser complicado gerenciar as janelas apenas com $alt + tab$ para alternar entre diferentes janelas abertas, neste caso pode ser interessante mandar algumas janelas para outro espaço de trabalho (basicamente um ambiente separado em que não é possível acessar as janelas de outros espaços) para mandar uma janela para outro espaço de trabalho basta pressionar $ctrl + shift + seta$ para mandar a janela para o espaço adjacente (por padrão as janelas são criadas no espaço 1 portanto só podem ser mandadas para o espaço de trabalho a esquerda), dependendo do ambiente de trabalho ha um numero diferente de espaços de trabalhos que podem ser usados, no caso do cinnamon este numero é fixo em 4. Para acessar os outros espaços de trabalho basta apertar $ctrl + setas$.

Caso o setup usado seja de dois monitores pode ser preferível mandar a janela para outro monitor ao invés de outro espaço de trabalho, neste caso basta pressionar $windows + shift + seta$, com a seta na direção do monitor para o qual se deseja mandar a janela.

Por fim um truque simples para encerrar rapidamente uma sessão, pressionar $ctrl + backscape$ encerra a sessão fechando também todos os programas abertos executando na sessão.

Capítulo 4

Controle de Processos

Uma das características mais proeminentes em sistemas UNIX é quanto controle sobre os processos é dado aos usuários. UNIX foi o primeiro sistema a prover seus usuários a capacidade de controlar múltiplos processos simultaneamente. Este capítulo irá tratar deste aspecto de controle de processos.

4.1 Foreground e Background

Parte da razão pela qual isto é possível é a divisão do ambiente em dois: o Foreground (ou primeiro plano) em que os processos rodando bloqueiam o seu Shell só retornando-o quando o processo termina; E o Background (ou segundo plano) é onde a maior parte dos processos (todos os processos do kernel, o sistema operacional, todas as aplicações com janelas como browsers, entre outras) executam sem interferir com o controle que o usuário tem sobre seu Shell ou a máquina em si.

Por padrão todos os processos lançados pelo usuário em seu Shell executam no foreground, isto pode ser difícil de ver em comandos que executam rápido como um ls ou um cd, editores de texto são uma boa forma de observar isto, abra o editor gedit (ou algum outro editor de texto visual que esteja instalado em sua máquina) e abra-o pelo terminal, você poderá ver que não consegue enviar mais comando para seu shell até que feche o editor de texto.

Existem duas formas de um usuário mandar processos para o background, a primeira é iniciando o processo já no background, o que é feito colocando-se um & (e comercial ou Ampersand) após a invocação do comando ou programa. A segunda é parar o processo no meio de sua execução (sem matá-lo) e continuá-lo em background, isto é feito apertando **ctrl** + **z** (como isto funciona será explicado mais adiante em 4.3) parando o processo, e em seguida digitando **bg** para mandar todos os processo parados serem executados em background.

De forma similar caso seja necessário é possível trazer comandos que estão executando em background para o foreground com o comando **fg**, sem argumentos o processo que será trazido é o último que foi mandado para executar em

background naquele Shell, ou utilizando o número do job, precedido por um por cento(%)(mais a respeito disto no capítulo 4.2) ou o nome (também precedido por um por cento) do job que se deseja trazer para o foreground.

Mover processos entre foreground e background não é muito utilizado em ambientes gráficos, porém em Shell é uma abstração extremamente útil, por vezes será executado um código demorado e o usuário precisa trabalhar em outras coisas durante a execução do código demorado por exemplo trabalhar no código de uma interface enquanto é feita uma consulta a um banco de dados.

4.2 Identificadores de processos e Jobs

No momento de sua criação, todos os processos UNIX recebem um número que os identifica unicamente na máquina em que existem, os chamados identificadores de processo (ou ID), de forma similar, se o processo está sendo lançado em background em um Shell recebe um identificador único para aquele Shell o Número de job (ou ID de job) do processo. Todos os processos da máquina irão possuir um ID, mas nem todos possuem um ID de job.

Ambos são úteis para o controle dos processos sendo executados, como IDs são globais nem todos os processos podem ser manipulados através do ID por um usuário normal, pois são processos do sistema operacional ou processos de outros usuários. Jobs por outro lado, por estarem atrelados ao Shell que os invocou podem ser manipulados pelo usuário que os criou.

Formas de manipular processos que estão sendo executados serão exploradas no capítulo 4.3, porém há algumas formas de utilização interessantes dos IDs e IDs de job que cabem melhor a esta secção, como o fato de que é possível referir-se ao job que foi colocado em background mais recentemente chamando pelo processo %+, de forma similar é possível se referir ao segundo processo mais recentemente adicionado ao background utilizando-se de %-.

O comando **jobs** lista todos os jobs em execução no Shell em que foi executado.

4.3 Sinais

Já foi dito que a forma com que o usuário se comunica com os processos é utilizando-se da entrada e saída do processo, porém esta forma de comunicação é limitado pois muitos processos não aceitam coisas vindo da entrada e não escrevem nada em nenhuma saída, além disso, processos em background não conseguem interagir com a entrada padrão (neste contexto me refiro a entrada como a entrada do Shell, em aplicações que lançam uma janela obviamente ainda é possível interagir com a aplicação através da entrada padrão).

Para lidar com isso e dar ao usuário a habilidade de mandar certas mensagens para processos que estão em execução e/ou em background existe o conceito de sinais, que são mensagens especiais para os processos, e que, em geral, caso

não sejam tratados de alguma forma (vide capítulo 4.4) resultam na morte do processo.

Sinais podem ser referidos por nomes especiais que os diferenciem como SIGINT para **int**errupções, SIGSTOP para paradas, SIGTERM para ex**term**inar o processo, o SIG em comum a todos é de **sig**nal, nome em inglês para sinais. Também podem ser referidos por números, que número se relaciona a que sinal varia de sistema para sistema, para escrever códigos que se comportem de maneira consistente entre sistemas é recomendado o uso dos nomes, para uma lista de todos os sinais no seus sistema com seu nome e respectivo número utilize o comando **kill -l**.

O comando **kill** manda um sinal passado como argumento para um processo, o sinal pode ser passado tanto por nome quanto por número, o processo deve ser passado como um ID. Por padrão quando nenhum sinal for passado para o comando kill o sinal enviado e o SIGTERM. A maioria dos sinais em sistemas UNIX só podem ser enviadas com o uso do comando kill, porém, alguns sinais especiais possuem atalhos no teclado que podem ser utilizados.

O atalho **ctrl** + **z** mencionado no capítulo 4.1 manda um sinal que faz o processo parar sua execução (um dos sinais que não mata o processo que o recebe). Outros atalhos úteis para mandar sinais são **ctrl** + **c** que manda um SIGINT para o processo e resulta em sua morte. Uma versão um pouco mais forte deste comando é **ctrl** + \ que manda SIGQUIT para o processo, por ser mais forte que o SIGINT recomenda-se que só se use o SIGQUIT quando SIGINT não funcionar.

Um sinal ainda mais forte que QUIT e INT é o sinal SIGKILL, que imediatamente mata o processo alvo e deve ser usado apenas em último recurso quando SIGTERM e SIGQUIT não conseguirem resolver o problema. A razão para isso é que ambos QUIT e TERM deixam o processo alvo executar suas rotinas de limpeza antes de encerrar, SIGKILL mata o processo e pronto, o que pode resultar em problemas dependendo do processo que foi o alvo.

Uma forma fácil de checar os IDs dos processos para utilizar o comando **kill** é utilizando-se do comando **ps** que lista os processos rodando. Sem argumentos **ps** irá listar apenas os processos no Shell corrente do usuário (o próprio comando incluso), mas opções podem ser dadas ao comando para pegar mais processos como todos os processos de um usuário, ou todos os processos que estão sendo executados na máquina, para uma lista de todas as opções do comando **ps** e suas funcionalidades consulte a página do manual do comando.

O comando **ps** normalmente ordena as informações dadas de forma que a primeira coluna contenha o ID do processo, a segunda contenha o terminal (ou pseudo terminal) em que o processo está sendo executado, a terceira contém o tempo de processador que o processo utilizou, e a quarta contém o comando que iniciou o processo. O comando **ps** é porém incapaz de relacionar IDs e ID de job dos processos.

4.4 Trap

Como mencionado no capítulo4.3 existem formas de tratar sinais para que um processo não morra ao receber um sinal. O comando **trap** faz exatamente isso, o comando recebe como argumento um comando (ou script) e uma lista de sinais, e sempre que o ambiente em que **trap** foi definido (pode ser o próprio Shell do usuário ou um script) receber um dos sinais definidos, ao invés de utilizar a ação padrão quando receber o sinal (em geral se encerrar), executa o comando passado e continua sua execução normalmente. Os sinais passados para o comando **trap** podem ser representados tanto pelo número quanto pelo nome. Caso seja executado em argumentos o comando simplesmente retorna uma lista das traps sendo utilizadas.

Em geral o usuário comum não necessita muito do uso de traps, mas seu uso é extremamente importante durante o reforço de grandes projetos, para proteger o código contra eventos inesperados no sistema, como uma entrada inválida. Uma das formas mais comuns de se usar o comando **trap** é mandar o código executar suas funções de limpeza de ambiente e encerramento antes de parar o código.

Traps possuem um escopo persistente após sua chamada, o que quer dizer que traps definidas em uma função por exemplo, passam a valer a partir do momento em que a função é executada, mesmo depois que a função terminar de executar. Não tema porém que tenha sido feito um programa que não pode ser morto de forma alguma por acidentalmente mandar todos os sinais serem ignorados, existem alguns sinais que são "imunes" a traps, SIGKILL é um deles, então em caso de absoluta necessidade é sempre possível matar um processo.

Foi mencionado no parágrafo anterior que é possível mandar um sinal ser ignorado, para fazer isto basta passar uma string vazia '"' ou '' como o comando a ser executado. Um dos sinais mais comuns a ser ignorado é o SIGHUP (de hangup ou desligar) que encerra o processo. Este sinal é ignorado com tanta frequência que um comando foi criado especialmente para executar comandos ignorando este sinal, o comando se chama **nohup**, e talvez seja de seu interesse olhar seu manual.

4.5 Exercícios de processos

1. Abra através do terminal uma aplicação gráfica (editor de texto, browser, interface gráfica) no foreground, e mande a aplicação para background

   - R:
     $ gedit
     ctrl + z
     $ bg

2. crie um script chamado LoopOi.sh e coloque o seguinte codigo dentro:

```
#!/bin/bash
```

```
while true; do
    echo oi
    sleep 60
done
```

de permissão de execução para o script e rode-o. Abra outro terminal use o comando mande o sinal SIGTERM para LoopOi.sh, encerrando sua execução.

- R:
  ps ax | grep LoopOi.sh
  kill -15 numero_do_processo_de_LoopOi.sh

3. No script utilizado acrescente uma linha que faça o scritp ignorar o SIGTERM, e tente matar o script como no exercício anterior.

- R:
  no script entra a linha : trap TERM. antes do laço.
  para matar o script troque o sinal enviado de 15 para 9.

Capítulo 5

Customização de Ambiente

Um ambiente é um conjunto de conceitos que expressa as coisas que outro conjunto de ferramentas faz, para ser entendível, coerente e confortável para o usuário. Um escritório é um ambiente, da mesma forma que o quarto de uma pessoa, ou até mesmo a casa de alguém. Todos esses ambientes descritos possuem características próprias que expressam como são usados, como a organização dos móveis ou dos objetos nos móveis.

Quanto mais personalização é feita nestes ambientes, mais eles se adéquam as necessidades de quem os usa, logo são melhor utilizados.

De forma similar, sistemas UNIX possuem arquivos, diretórios, entradas e saídas, assim como ferramentas para utiliza-los, manipulá-los e personalizar seu uso. Parte deste ambiente UNIX é composta pelo teclado e monitor, mas outra parte é composta pela própria organização dos arquivos no sistema, assim como os nomes dados aos arquivos e diretórios.

Além destes métodos mais básicos de customização do ambiente UNIX, ha ainda alguns mais avançados que serão discutidos logo abaixo.

5.1 Customização do Bash

A forma mais facilmente visível de customizar seu ambiente de trabalho em sistemas UNIX, é customizar seu Shell para atender suas preferências.

Nos sistemas UNIX, é comum que arquivos de configuração usados para customizar o ambiente terminem em "rc", alguns grupos dizem que se refere a uma abreviação de "run command", enquanto outros acham que é um nome dedicado ao antigo sistema ROMCOM, brevemente mencionado no capítulo 2

Apesar de haver certa padronização na forma de customizar diferentes Shells, esta seção em particular trata da customização do Bash, para customização de outros Shells, existem diversos tutoriais na internet explicando como fazer as customizações e suas particularidades.

Bash se utiliza de quatro ferramentas, que serão discutidas em maiores detalhes nas subseções abaixo, para permitir a customização de ambiente, são elas,

arquivos especiais, Apelidos (ou aliases), opções e variáveis (Na verdade estas ferramentas são usadas por grande parte dos sistemas UNIX, diferindo apenas em suas implementações e algumas poucas particularidades em seu uso).

5.1.1 Arquivos especiais

Existem, na pasta home de todos os usuários, alguns arquivos especiais, que são usados para customizar o ambiente Shell do usuário. São estes .bash_profile, .bashrc e .bash_logout.

O mais importante deles sendo .bash_profile este arquivo é lido e os comandos nele contido são executados, toda vez que é chamado um bash de login (no caso do DINF, isto significa apenas abrir um terminal).

Apesar de suportar as três nomenclaturas de .bash_profile, .bash_login e .profile, apenas um destes arquivos é lido durante a inicialização da sessão do Bash, caso .bash_profile não exista, será lido .bash_login, e caso este também não exista será então lido o .profile.

Como bash_profile é executado apenas quando um novo Shell é iniciado, quando subshells são executados (por exemplo digitando bash na linha de comando) é lido o arquivo .bashrc. Isto permite separar comandos de inicialização dos que podem ser necessários durante a execução de subshells.

Nos casos em que os comandos a serem rodados serem os mesmos, basta executar .bashrc de dentro de .bash_profile.

Caso um .bashrc não exista são executados os comandos de inicialização em /etc/barh.bashrc.

No caso do DINF, o .bash_profile, simplesmente verifica se existe um .profile, caso exista ele o executa, por sua vez o .profile verifica se o Shell sendo usado é Bash, e se o usuário possui um .bashrc, caso sim, executa o .bashrc.

Caso o usuário abra o arquivo .bashrc, irá encontrar diversas linhas com esta cara:

```
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/usr/local/bin
SHELL=/bin/bash
MANPATH=/usr/man:/usr/X11/man
EDITOR=/usr/bin/vi
PS1=\h:\w$
PS2='> '
export EDITOR
```

Estas linhas definem algumas coisas básicas do ambiente Bash, e o que fazem será detalhado melhor na seção 5.1.4, por hora é melhor deixar estas linhas de lado até que seus funcionamentos sejam mais bem entendidos, caso deseje editar seu .bashrc apenas adicione linhas após as já existentes

O arquivo .bash_logout é executado sempre que um Bash é encerrado, e pode ser útil caso se deseje apagar alguns arquivos temporários sempre que se encerrar uma sessão, ou caso seja desejado marcar a duração do login no Shell.

O arquivo não precisa existir, e caso não exista o arquivo, nenhum comando extra é executado.

5.1.2 Aliases

Apelidos, ou aliases como vou me referir pois é o nome do comando utilizado, são uma pequena ferramenta provida pelo Bash para facilitar o uso de comandos com nomes mais complicados, ou com muitas opções que ninguém quer ter que digitar constantemente.

Aliases podem ser definidos em quatro lugares diferentes, na linha de comando, no seu .bash_profile, no seu .bashrc, ou ainda, caso ele seja chamado pelo seu .bashrc, em um arquivo chamado .bash_aliases.

Apesar disto a forma de declarar um alias é a mesma em todos os arquivos:
alias nome=comando
ou
alias nome="comando"

Alias é um comando, que coloca em "nome" o "comando". Algumas pessoas que tem problemas frequentes com erros de digitação costumam colocar aliases para erros cometidos frequentemente, como:

- alias sl=ls
- alias gerp=grep
- alias dc=cd

Às vezes também pode ser desejado um alias para um arquivo ou diretório acessado frequentemente mas que está em um lugar profundo na árvore de diretórios.

Caso o comando que se deseja usar em um alias possua mais de uma palavra separada por espaços, é obrigatório o uso de aspas para que o comando seja executado como pretendido.

Alias são recursivos, ou seja, é possível dar um alias para um alias.

Essa recursividade porém traz o risco de entrar em um loop de execução, isso é tradado de duas formas, quando um alias é analisado, a primeira palavra no comando é analisada para verificar se não é a mesma que o alias usado, e quando é detectado um alias redundante, o alias redundante é eliminado.

Aliases são extremamente úteis em customizar o ambiente Shell, porém, com o passar do tempo foram sendo passados por scripts e funções que serão discutidos em outro momento.

5.1.3 Opções

Opções são configurações que alteram o funcionamento de Shells e scripts, em geral são associados ao comando "set".

Para ativar uma opção basta usar "set -o opção", e para desativar uma opção basta usar "set +o opção", em ambos os casos múltiplas opções podem ser alteradas colocados mais "-o/+o opção".

O uso do + e do - para ativar e desativar as opções, parece e é, contraintuitivo. O - ativa uma opção pois em geral é o sinal usado para passar opções para comando, o + foi usado como um oposto sem parar para pensar muito.

Uma lista detalhada com mais opções e o que elas fazem pode ser encontrada em https://www.tldp.org/LDP/abs/html/options.html

5.1.4 Variaveis

Apenas a capacidade de ativar e desativar certas características do sistema não é o suficiente para customizar seu sistema, principalmente porque diversos fatores não podem ser expressos em termos de "ativado" e "desativado".

Variáveis são declaradas da forma "nome=valor" e de forma similar aos aliases, caso "valor" seja composto de diversas palavras separadas por espaço deve estar entre aspas. Para usar variáveis basta usar o símbolo do dólar ($) junto com o nome da variável.

É possível remover uma variável usando o comando unset, em geral isto é desnecessário pois todas as variáveis que não foram declaradas são, por padrão atribuídas o valor nulo. Porém dependendo das opções sendo utilizadas no sistema pode ser interessante remover variáveis.

A forma mais simples de checar qual o valor de uma variável é simplesmente imprimi-la com o comando echo da forma "echo $var". Esta técnica também é boa para ver como serão expandidos os coringas discutidos no capítulo 2.3.

Assim como com opções, algumas variáveis padrão do Shell são importantes para usuários de UNIX em geral, e serão discutidas logo abaixo.

Qualquer um que já passou algum tempo próximo a programadores de UNIX, facilmente percebeu que o prompt de comandos não é fixado pelo sistema. Muitos usuários de UNIX customizaram seu prompt para mostrar informações relevantes, como o diretório em que se encontra, a data, em qual branch de um repositório do git se encontra (caso o diretório seja um diretório do git).

Bash usa quatro variáveis para definir qual é o prompt a ser exibido, PS1, PS2, PS3, e PS4. PS1 é chamado de prompt primário, e em geral seu valor é: "`\s-\v\$`". PS2 é o prompt de quebra de linha, em geral é ">" e pode ser visto quando uma quebra de linha está presente em um comando. PS3 e PS4 geralmente são usadas para debug.

Para mais informações em como customizar seu prompt de comando acesse https://www.howtogeek.com/307701/how-to-customize-and-colorize-your-bash-prompt/ para mais detalhes.

Outra variável notável do sistema é a variável PATH, que serve para ajudar o sistema a achar os comandos chamados pelos usuários.

Como dito na seção 2.2 existem arquivos no sistema que são usados especialmente para guardar os binários do sistema, o /bin e o /usr/bin, programadores ainda costumam ter o próprio diretório para armazenar os binários de produção Própria, e em geral é indesejável saber o caminho completo até um comando no seu sistema para executá-lo.

PATH serve exatamente para isso, o valor de PATH é uma lista de diretórios que o sistema busca toda vez que um comando é invocado, os nomes dos diretórios são separados por dois pontos (:).

É importante ter conhecimento desta variável pois há diversas situações em que executáveis estarão em diretórios fora de PATH e nestes casos é necessário

saber colocá-los na PATH caso necessário.

Para adicionar diretórios novos a sua PATH e fazer com que eles automaticamente seja colocados em PATH toda vez que um novo Bash for executado, basta adicionar ao seu .bashrc a linha "PATH=$PATH:seu_diretório" isso faz com que PATH seja o que era antes, mais o se diretório.

Outra forma de fazer isto é "PATH=seu_diretório:$PATH" que coloca seu diretório como o primeiro a ser procurado para executar comandos, isto é perigoso, pois seus executáveis que possuírem nomes iguais a comandos do sistema serão executados antes dos do sistema, assumindo que a funcionalidade deveria ser a mesma, ainda a o risco de sua própria implementação estar incorreta.

Para evitar um excessivo consumo de tempo na busca pelos comandos nestes diretórios, Bash usa uma tabela que vai preenchendo conforme acha os comandos que foram chamados pelo usuário, sempre que um comando não estiver nesta tabela será feita a busca por PATH.

Por padrão algumas variáveis são conhecidas por todos os subprocessos criados, estas variáveis são uma classe especial de variáveis chamadas variáveis de ambiente. Algumas destas variáveis são HOME, MAIL, PATH, PWD entre outras. A razão pela qual os subprocessos necessitam saber dos valores destas variáveis deveria ser evidente, editores de texto precisam saber qual o terminal sendo utilizado, a variável TERM cuida disso, programas de e-mail tem que saber qual editor de texto deveria ser usado, a variável EDITOR cuida disso, há diversos outros casos em que subprocessos precisam saber de informações provenientes de variáveis de ambiente.

É ainda possível definir valores para variáveis de ambientes apenas para subprocessos em particular, isto é feito colocando a declaração da variável antes da chamada do comando

5.2 Customização de editores de texto

Assim como é possível configurar o comportamento de Bash com arquivos como .bashrc, editores de texto como gedit, nano e vim podem ser configurados para serem customizados, quase todos os editores de texto provavelmente possuem um arquivo de rc para serem configurados, por predileção minha, irei falar um pouco sobre configurações do vim, formas de customizar outros editores podem facilmente ser achadas na internet.

Existem duas formas de customizar as configurações do vim, a primeira, apenas para os que tiverem acesso de superusuário na máquina, é criar configurações globais que serão utilizadas por todos os usuários, para isto seriam colocadas as customizações nos arquivos de configurações globais do vim, que em geral ficam em /etc/vim/vimrc ou em /etc/vimrc dependendo da distribuição utilizada. A outra forma é usar um arquivo no diretório home chamado .vimrc que suplanta as configurações globais quando houver conflito.

De forma similar ao .bashrc o .vimrc é chamado e executado toda vez que uma nova instância do vim é aberta, e ainda de forma similar podem ser alterados durante a execução do vim (porém de forma similar ao Bash, alterações

feitas aos valores destas variáveis dentro de uma instância do vim são validos apenas para aquela instância).

Entre as configurações possíveis do vim estão:

- mostrar números das linhas
- automaticamente endentar linhas novas
- trocar tabs por espaços em brancos
- controlar o tamanho de um tab

e muitos outros, para ver todas as configurações possiveis para o vim, o que elas fazem, e como usa-las, acesse o link https://www.linode.com/docs/tools-reference/tools/introduction-to-vim-customization/

5.3 Exercicios de customização

1. Edite seu .bashrc para incluir o diretório corrente em PATH.

   R: acrescentar ao .bashrc a linha:

   export PATH=$PATH:.

2. acesse o link:

   https://canaltech.com.br/linux/conheca-e-aprenda-a-usar-o-editor-vim-no-linux/ e se familiarize um pouco com o funcionamento do Vim, caso ainda não conheça.

3. Abra seu .bashrc e edite o prompt de comando para que seu usuário e a máquina sejam mostrados em verde, a diretório corrente, a a partir da home seja mostrado em azul, e o branch do git seja mostrado em vermelho, apenas quando existir.

   R:PS1=`${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[01;31m\]$(parse_git_branch)\[\033[00m\]\$`

Capítulo 6

Exemplos práticos

Neste capítulo serão apresentadas três situações reais em que problemas que afetam pessoas de verdade foram resolvidos utilizando-se de scritps em Shell.

6.1 Organizando as informações de patrimônio

O primeiro problema é o de lidar com os arquivos de patrimônio do departamento de informática. Por questões de melhor organização e facilitamento de trabalho futuro eram necessárias as conclusões de três tarefas base, sanitizar o arquivo de entrada, obter um arquivo separado apenas com os locais onde patrimônios se encontram e separar o arquivo original em diferentes arquivos baseado nas diferentes localizações, cada arquivo contendo apenas as informações de um local.

Para obter o arquivo com as informações do patrimônio foi executado o comando:

```
cp ~marcos/tmp/patrimonio.csv .
```

Para melhor explicar o problema segue abaixo um fragmento d arquivo utilizado:

```
"460805";"FORNO DE MICROONDAS";"30L";"CONSUL";"2100.13.02.24";
"460806";"VENTILADOR DE TETO (5234)";"127V ";"";"2100.13.02.21";
"460807";"VENTILADOR DE TETO (5234)";"127V ";"";"2100.13.02.21";
"460808";"VENTILADOR DE TETO (5234)";"127V ";"";"2100.13.02.21";
"460809";"VENTILADOR DE TETO (5234)";"127V ";"";"2100.13.02.21";
```

Como é possível ver o arquivo possui 5 campos de texto separados por “;”(ponto e vírgula), o que ocorre de tempos em tempos é que por erro humano são inseridos dentro dos campos com aspas ponto e vírgulas, e para programas que dividem os campos automaticamente baseados nos ponto e vírgulas isto é um problema, logo a primeira tarefa necessária, sanitizar o arquivo e remover os ponto e vírgulas

dentro dos campos delimitados por aspas para evitar problemas.

Além disso, seria ideal para a análise se as informações estivessem divididas baseado no local em que os objetos se encontram. Para saber onde os objetos se encontram basta olhar o 5º campo do arquivo, então para poder separar os patrimônios em arquivos menores baseando-se no local onde os objetos se encontram é preciso primeiro obter um arquivo que diga quantos locais diferentes existem e quais são seus códigos.

Para prevenir potenciais perdas em caso de erro, foi feito um backup do artigo de patrimônios chamado patrimonio.csv.bkp com o comando

```
cp patrimonio.csv patrimonio.csv.bkp
```

6.1.1 Sanitizando o arquivo de entrada

O código utilizado para a resolução do problema de achar ";"(ponto e vírgulas) no meio de campos do arquivo csv foi:

```
awk -F'"' -v OFS='"' '{for(i=2;i<NF;i+=2) gsub(";", "", $i)}1'
    patrimonio.csv >tmp.csv;

mv tmp.csv patrimonio.csv
```

- awk é uma linguagem de programação, boa para ser usadas em manipulação de strings em conjunto com scritps shell. Em particular, quando se quer fazer manipulações com arquivos awk é considerado extremamente eficiente, já que foi criado para este propósito.
- a opção -F'"' faz com que o awk intérprete aspas como separador de campos do arquivo.
- a opção -v OFS serve para que a saída use aspas como separador também.
- 'for(i=2;i<NF;i+=2) gsub(";", , $i)1' é o script usado pelo awk, o uso de aspas simples sendo necessário para indicar que esse é o código que o awk deve rodar.
  - o comando "for"vai iterar sobre os campos de cada linha, pegando apenas os campos pares, porque os campos ímpares, devido a foram como os arquivos csv são gerados, contém apenas ponto e vírgula.
  - o comando gsub faz uma chamada para substituir o que está no primeiro campo (ponto e vírgula), pelo conteúdo do segundo campo (nada), no terceiro campo (o campo da linha).
  - o 1 no final do comando serve para imprimir a linha.

- o comando é executado no arquivo patrimonio.csv, e o resultado de todos os comandos é colocado no arquivo tmp.csv Concluída a remoção dos ponto e vírgula desnecessários o arquivo temporário criado sobrescreve o arquivo patrimonio.csv

6.1.2 Obtendo o arquivo com os locais

Para resolver o problema de separar o campo com os números referentes ao local onde se encontram os bens patrimoniados foi usado o comando:

```
cut -d\; -f 5 patrimonio.csv | cut -d\" -f 2 | sort -u > locais.txt
```

- cut é um comando que remove campos de arquivos.
- -d; é uma opção do cut para usar como delimitador de campos o caractere desejado ao invés do padrão que é o espaço em branco, neste caso foi usado o ponto e vírgula como separador, a contra barra é usada para escapar o ponto e vírgula pois é um caractere reservado no bash.
- -f é uma opção que me permite escolher quais os campos desejados, no caso, o campo 5 é o desejado pois sabe-se que este é o campo que contém os números referentes aos locais.
- no segundo comando da pipeline eu escolho o segundo campo delimitado por aspas para não pegar as aspas em volta do número.
- o ultimo comando da pipeline sort, é um comando usado para ordenar linhas de arquivos, a opção -u serve para eliminar duplicatas da ordenação.

6.1.3 Separando em arquivos diferentes

Para resolver o problema de criar um arquivo para cada local de patrimônio, contendo apenas as informações de patrimônio referentes a localização foi usado o comando abaixo:

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

- o for irá iterar para cada linha gerada pela saída do comando "cat locais.txt"
- para cada um dos locais no arquivo locais.txt, irá usar o comando grep no arquivo patrimonio.csv, para obter todas as linhas que possuem o número do local. O comando grep percorre um arquivo procurando linhas que encaixem com um padrão, e imprime elas na tela.
- todas as linhas retornadas pelo grep são então colocadas em um arquivo chamado, "número procurado".csv

- o laço termina quando todos os locais tiverem sido iterados.

6.2 Contabilização de matriculas

Com o propósito de fazer uma análise estatística da situação dos alunos matriculados nos cursos ofertados pelo departamento de informática são necessários o cumprimento de duas tarefas. Contabilizar o número total de matrículas semestre a semestre e montar uma tabela com o número de inscritos de cada departamento nas matérias do departamento de informática.

Isto é de interesse para o departamento pois permite saber a distribuição de cursos que têm aulas oferecidas por ele, porém a única informação que o departamento possui é uma lista com os GRRs dos alunos matriculados em cada disciplina juntamente com o curso que cursam, não há uma informação mais consolidada e organizada. O propósito destes scripts é separar e organizar esta informação para facilitar a análise dela.

Para preparar os arquivos para manipulação é rodado um pequeno script que remove a linha de cabeçalho de todos os arquivos dados.

```
for j in $(ls | grep -v sh); do
    cd $i
    for j in $(ls); do
        tail -n +2 $j > tmp.txt; cat tmp.txt > $j
    done
    rm tmp.txt
    cd ..
done
```

6.2.1 Obtendo o número total de matrículas

Para obter o total de matrículas, semestre a semestre o seguinte script foi executado o seguinte script:

```
#!/bin/bash
set -e
if [[ -d res ]] then;
    rm -rf res
fi
mkdir res
if [[ -e Tot_mat_per.txt ]]; then
    rm Tot_mat_per.txt
fi
for i in 19881 19882 19891 19892 19901 19902 19911 19912 19921 19922
     19931 19932 19941 19942 19951 19952 19961 19962 19971 19972 19981
     19982 19991 19992 20001 20002 20011 20012 20021 20022; do
```

```
    for j in $(ls | grep -v sh | grep -v res ); do
        cd $j
        cat $i.dados >> ../res/$i.dados
        cd ..
    done
done
cd res
for i in 19881 19882 19891 19892 19901 19902 19911 19912 19921 19922
     19931 19932 19941 19942 19951 19952 19961 19962 19971 19972 19981
     19982 19991 19992 20001 20002 20011 20012 20021 20022; do
    var=$(sort -u $i.dados | wc -l )
    echo $i : $var >> ../Tot_mat_per.txt
done
```

- set -e é uma opção que, quando ativada, faz com que o script encerre sua execução quando acontecer o primeiro erro
- Os dois "ifs"são para remover os resultados gerados por uma execução anterior do script, o teste é para que não se tente remover os diretórios e arquivos quando estes não existirem.
- O primeiro “for” passar por todos os diretórios e armazena as informações de todas as disciplinas de um semestre em um arquivo no diretório "res", isto ocorre para todos os semestres, e o nome do arquivo é "número_ do_ semestre.dados". São ignorados scripts e o conteúdo dos arquivos em "res".
- O segundo "for"de anos, serve para olhar todos os arquivos em "res"e :
  - sort -u -> Ordenar as linhas do arquivo removendo repetições;
  - wc -l -> Contar quantas linhas o arquivo possui, o que se traduz na quantidade de matrículas únicas no semestre, tudo armazenado na variável "var".
- Por fim é impresso para o arquivo de resultado uma sequência de pares "semestre : Número de matrículas".

6.2.2 Montando a tabela de matrículas

Para resolver o problema de contabilizar o número de matrículas por curso, por semestre foi usado o script abaixo.

```
#!/bin/bash
set -e
if [[ -d res ]]; then
    rm -rf res
fi
mkdir res
```

```
if [[ -e Tot_mat_per.txt ]];then
    rm Tot_mat_per.txt
fi
if [[ -e tabela.txt ]];then
    rm tabela.txt
fi
for i in 19881 19882 19891 19892 19901 19902 19911 19912 19921 19922
     19931 19932 19941 19942 19951 19952 19961 19962 19971 19972 19981
     19982 19991 19992 20001 20002 20011 20012 20021 20022; do
    for j in $(ls | grep -v sh | grep -v res ); do
        cd $j
        cat $i.dados | cut -d: -f 1 >> ../res/$i.dados
        cd ..
    done
done
cd res
for i in 19881 19882 19891 19892 19901 19902 19911 19912 19921 19922
     19931 19932 19941 19942 19951 19952 19961 19962 19971 19972 19981
     19982 19991 19992 20001 20002 20011 20012 20021 20022; do
    sort $i.dados | uniq -c | awk '{print $2" "$1} ' > $i.count
done

for i in 19881 19882 19891 19892 19901 19902 19911 19912 19921 19922
     19931 19932 19941 19942 19951 19952 19961 19962 19971 19972 19981
     319982 19991 19992 20001 20002 20011 20012 20021 20022; do
    cat $i.count >> tmp
done

cut -d' ' -f 1 tmp | sort -u | tr '\n' ';' |sed 's=;= | =g' > cursos
echo "" >> cursos
cat cursos > tmp
echo -n " | " > cursos
cat tmp >> cursos
rm tmp
echo -e "\t$(cat cursos)" > tabela.txt
sed -i 's= | =\n=g' cursos

for i in 19881 19882 19891 19892 19901 19902 19911 19912 19921 19922
     19931 19932 19941 19942 19951 19952 19961 19962 19971 19972 19981
     19982 19991 19992 20001 20002 20011 20012 20021 20022; do
    echo -n "$i| " >> tabela.txt
    for j in $(cat cursos); do
        awk -v a="$j" '{if ($1==a) printf $2} END {printf " | "}'
            $i.count >> tabela.txt
    done
    echo "" >> tabela.txt
done
cat tabela.txt | column -t -s"|" > ../tabela.txt
```

- Até o primeiro for funciona exatamente da mesma maneira como apresentado na solução do problema anterior.
- O segundo “for” passa por todos os arquivos de dados extraídos e usa o comando "uniq"para achar linhas repetidas (como o arquivo $i.dados contém apenas os cursos linhas repetidas significam múltiplas matrículas de um curso) , a opção -c serve para que seja reportado o número de ocorrências da linha repetida no arquivo. como o comando só usa linhas adjacentes na busca, é necessário ordenar o arquivo com o comando "sort"antes de contar as linhas. a comando em awk no final serve para que os dados sejam exibidos da forma "curso número_de_ matrículas". todos os semestres passam por este processo e os resultados são armazenados em arquivos chamados "número_ do_ semestre.count".
- O próximo "for"coloca todas as informações extraídas em um único arquivo.
- Este arquivo é ordenado de forma a remover repetições, resultando na informação de quais são os cursos existentes no período observado, o arquivo é então formatado de forma a melhor ser usado depois e salvo em um arquivo chamado "cursos". O arquivo temporário que armazena estas informações antes de serem formatadas é então deletado.
- Os cursos são colocados na primeira linha de um arquivo que irá armazenar a tabela final com os resultados e o arquivo contendo os cursos é novamente formatado para uma forma mais adequada aos comandos que virão.
- O par de "for"s na sequência serve para iterar por todos os semestres e cursos possíveis, para cada semestre, se o curso ocorre no semestre, o número de matriculas é impresso, caso contrário nada acontece, a impressão de espaços e "serve para separação e preparação de formatação mais pra frente. Foi usado printf ao invés de print para que não houvesse uma quebra de linha a cada valor inserido.
- O echo vazio serve para colocar uma quebra de linha entre os semestres na tabela.
- Pôr fim a tabela gerada pelo par de laços acima é formatada pelo comando "column"que serve para formatar a entrada em colunas, a opção -t serve para que as colunas sejam organizadas em forma de tabela e a opção -s"serve para que o comando saiba que o separador de campos são as ".

6.3 Manutenção do firewall do departamento de informática

Sendo um local com um grande número de acesso a internet, não é surpresa que os professores e técnicos do departamento queiram monitorar o firewall do dinf,

e caso aja algum tipo de mensagem sendo bloqueada em excesso e queiram que seja enviada uma notificação que isto está ocorrendo.

Diariamente são feitas diversas requisições web no dinf, algumas que passam pelo firewall outras que são bloqueadas por ele. As requisições que são bloqueadas ocorrem com frequências diferentes, e por razões óbvias pacotes bloqueados mais necessitam mais atenção do que os pouco frequentes, de forma similar aos problemas anteriores, a informação no log contém apenas as informações brutas, e para fazer uma análise decente dos bloqueios que estão ocorrendo e notificar as pessoas relevantes quando uma quantidade anormal de determinados tipos de bloqueios forem detectados é necessário um script que não só consolide a informação, mas que o faça de forma rápida, pois o log cresce constantemente.

Para resolver o problema de contabilizar o número de bloqueios no firewall do DINF foram usados 2 scripts, o primeiro percorre o log do firewall contabilizando os erros, e termina gerando dois arquivos, um contendo o numero de vezes que cada bloqueio ocorreu, e outro contendo os bloqueios que ocorreram mais que 20000 vezes. O segundo script chama o primeiro, contabilizando o tempo de execução dele, e manda um email reportando quais erros ocorreram mais do que o número limite setado como 20000 neste caso.

6.3.1 Script 1

```
#!/bin/bash

WORK_DIR="/home/bcc/rsc15/nobackup/nobackup/ci320/ex_firewall"
FIRE_FILE="log-firewall"
BLOCKS="tipos-bloqueios"
TMP="resultados.txt"
OUT="saida.txt"
ADDR="rsc15@inf.ufpr.br"
EXC="over.txt"



cd ${WORK_DIR}
# olha o arquivo de log e conta os erros
time ./script.sh
#manda um e-mail com os erros acima do limite
if [[ -s ${EXC} ]]; then
  mail -s "LIMITE DE BLOQUEIOS EXCEDIDO" -a "Os erros que excederam o
      limite foram:" ${ADDR} < ${EXC}
fi
```

Este é um script simples, ele seta todas as variáveis que serão usadas como, onde é o diretório em que devem ser executados os comandos, onde estão os arquivos que serão usados e onde devem ser armazenados os arquivos gerados.

o script em si não faz muito, ele muda o diretório para o diretório de trabalho, chama o script que contabiliza o número de ocorrências de cada bloqueio, mede quanto tempo ele levou para executar e manda um único e-mail quando houverem bloqueios que ocorreram mais do que o número limite, se não existe nenhum bloqueio que ocorreu mais do que o número limite de vezes não é mandado o e-mail.

6.3.2 Script 2

```
#!/bin/bash
export LC_ALL=C
export WORK_DIR="/home/bcc/rsc15/nobackup/nobackup/ci320/ex_firewall"
export FIRE_FILE="log-firewall"
export BLOCKS="tipos-bloqueios"
export TMP="resultados.txt"
export OUT="saida.txt"
export EXC="over.txt"

#mudar para diretorio de trabalho
cd ${WORK_DIR}

# cria out se nao existir
# sobrescreve caso ja exista
> ${OUT}

#FNR==NR executa os comandos apenas para o primeiro arquivo,
#serve para inicializar todos os erros possiveis com 0;
#arr[$6] = arr[$6] + 1 -> para cada incidencia do erro, aumenta a
    contagem
#for -> printa o par de chave : contagem, para todos os bloqueios
awk ' FNR==NR{arr[$1":"] = 0; next} {arr[$6] = arr[$6] + 1}
     END { for(x in arr)
            print x " "arr[x];
     }' ${BLOCKS} ${FIRE_FILE} > ${TMP}

#formata a saida
echo "*** V4 ***" > ${OUT}
grep -v "V6" ${TMP} >> ${OUT}
echo "" >> ${OUT}
echo "*** V6 ***" >> ${OUT}
grep  "V6" ${TMP} >> ${OUT}

# imprime os bloqueios que ocorreram mais do que 20000
awk -F: '$2 > 20000{print $1" : "$2}' ${TMP} > ${EXC}
```

Este script contabiliza o número de ocorrências dos bloqueios da seguinte forma:

```
awk ' FNR==NR{arr[$1":"] = 0; next} {arr[$6] = arr[$6] + 1}
      END { for(x in arr)
             print x " "arr[x];
      }' ${BLOCKS} ${FIRE_FILE} > ${TMP}
```

- FNR==NR é uma construção em awk para lidar com múltiplos arquivos de entrada, e garante que os comandos na cláusula só sejam executados para o primeiro arquivo, no caso inicializar um vetor com os tipos possíveis de bloqueio que podem ocorrer.
- arr[$6] = arr[$6] + 1 incrementa a contagem de bloqueios na posição correspondente no vetor de bloqueios, $6 é usado pois é garantidamente a coluna que contém os erros
- END executa os comandos apenas ao final da execução do awk, neste caso ele percorre o vetor de bloqueios e imprime todos os pares de chave/valor existentes

o resto do script serve para formatar a saída, com exceção da última linha

```
awk -F: '$2 > 20000{print $1" : "$2}' ${TMP} > ${EXC}
```

que serve para obter os bloqueios que ocorreram mais do que o número limite para que seja possível mandar o e-mail.

6.4 Conclusão

Estes foram alguns exemplos de situações reais em que problemas complicados foram resolvidos com a utilização de shell em sistemas UNIX. Existem dezenas de milhares de outros problemas simples e complexos que podem ser resolvidos com aplicações criativas de comandos e scripts em Shell. Esperamos que este capítulo tenha servido de inspiração para que o leitor pesquise mais a respeito de shell, descobrindo comandos e shells que não foram mencionados neste livro, assim como esperamos que o leitor se sinta mais compelido a tentar usar scripts shell para resolver problemas.

Referências Bibliográficas

[1] Wikipedia. Boune again shell.

[2] Wikipedia. Bourne shell.

[3] Wikipedia. Cshell.

[4] Wikipedia. Korn shell.

[5] Wikipedia. Shell.

[6] Wikipedia. Unix.