Johanna Elisabeth Rogalsky

SHELL E BASH

Trabalho redigido como requisito parcial à conclusão da disciplina Tópicos em Programação de Computadores (CI320), setor de Ciências Exatas, da Universidade Federal do Paraná.

Curitiba PR
2018

Resumo

O UNIX, sistema operacional multitarefa e multiusuário, vem evoluindo e sendo modificado ao longo dos anos desde 1965. Ele traz consigo uma componente que possui inúmeras funcionalidades e que foi criada na concepção do seu projeto: a shell. Ela possui comandos que podem ser agrupados em linhas de comandos e/ou scripts em sua própria linguagem. Sendo assim, é usada para manipulação de arquivos e diretórios, redirecionamentos de entrada/saída e pipelines, além de dispensar o uso de mouse e proporcionar o controle de processos no Linux de diversas formas.

Sumário

1 Introdução

UNIX é um conjunto de sistemas operacionais multitarefa e multiusuário compostos por um ambiente de desenvolvimento de software contendo compiladores de linguagens de programação, bibliotecas e um *kernel* possuindo arquivos e documentos para configuração do sistema dependentes de máquina, além de códigos fonte portáveis e modificáveis dessas componentes. Sendo assim, é um sistema auto contido e uma ferramenta largamente utilizada para o ensino. Além disso, também possui um *assembler*, manuais dos comandos utilizados na *shell* (interpretador de linha de comando) e documentos detalhando os subsistemas.

1.1 História do UNIX

Sua origem partiu da ideia, que surgiu na metade da década de 60, de construir um sistema que permitisse o acesso de múltiplos usuários de forma simultânea ao *mainframe*. Na época, era chamado de *Multiplexed Information and Computing Service* (Multics) por pesquisadores do *Massachusetts Institute of Technology* (MIT), dos laboratórios *AT&T Bell* e da *General Electric* que estavam envolvidos na criação [10]. Porém o tamanho e a complexidade do projeto levaram a diferenças entre os três grupos e como consequência o fracasso.

Mas Ken Thompsom, integrante da *AT&T Bell*, não ficou contente com o fim do projeto e, em 1969, se juntou com mais alguns pesquisadores para desenvolver o UNICS (*Uniplexed Information and Computing Service*), um sistema menos complexo. Ele era uma sistema de arquivos contendo utilidades de manipulação, como cópia, impressão, edição e deleção, que obteve a capacidade de se sustentar apenas após um *assembler* ter sido completado. Ademais, tudo era feito utilizando o interpretador de linha de comando [7], sendo um componente importante do sistema.

O nome UNIX apareceu apenas em 1970, quando o pesquisador teve pretensões de utilizar o sistema com conceitos de processos em uma máquina maior que a PDP-7 que vinha sendo utilizada até então. Dessa forma, passava a ser rodado em uma máquina PDP-11/20. Nesse estágio, foram adicionados um programa de formatação de texto chamado *roff* e um editor. Logo, o sistema inicial de processamento de texto consistia do UNIX, do *roff* e do editor, sendo escrito em linguagem *assembly* e usado principalmente para formatação e edição de patentes [7].

Em 1972, com a intenção de possibilitar o uso de fluxos de controle não-hierárquicos que caracterizam corrotinas, o interpretador de linha de comando recebia o *pipe* para construir

linhas de comando chamadas *pipelines*. Então um ano depois, o sistema UNIX era re-escrito em uma linguagem de mais alto nível, a linguagem C, para que obtivesse portabilidade para outras plataformas computacionais. Sendo assim, sua quinta versão era lançada e licenciada para fins educacionais e em 1975, era licenciada sua sexta versão para outras empresas e companhias [10], com uma nova versão da sua *shell*.

E além da portabilidade de plataforma, também era necessário que o sistema se tornasse independente de máquina. Para isso, os pesquisadores dos laboratórios da *AT&T* (*Bell Labs*) desenvolveram uma versão para uma máquina chamada *Interdata 8/32* em 1977. Sendo em 1979 lançada a Versão 7 UNIX, provendo aproximadamente 50 *system calls* e também a primeira versão popular da *shell*: a *Bourne Shell*, que concorria com a C-*Shell* na área acadêmica.

No começo dos anos 80, várias versões foram desenvolvidas pela empresa *Bell* e com isso o interesse de estudantes e profissionais cientistas da computação aumentou, fazendo com que o UNIX fosse visto como um potencial sistema operacional adequado para todos os computadores. Assim, em 1983, o UNIX *kernel* era composto de código majoritariamente escrito em C, com mais de 75% independente de máquina e chamado *UNIX System V Release 1* [7]. Nesse mesmo ano, o projeto de software livre GNU, foi fundado.

Devido ao fato de existirem muitas implementações ao mesmo tempo, a empresa *AT&T* criou e emitiu um padrão em 1985, o *System V Interface Definition* (SVID), exigindo que os sistemas operacionais fossem marcados como "*System V*"como conformidade. Ademais, em 1988, a empresa tomou mais um passo na padronização e fez parcerias com outras empresas do mercado. A principal, foi buscar colaboração com a *Sun Microsystems* e juntar esforços para o que viria a se tornar o *System V Release 4* [7].

Por volta de 1991, o UNIX vinha ganhando concorrência do Linux *kernel*, que utilizava partes do projeto GNU e era uma reimplementação do UNIX. Trazendo consigo uma nova versão para o interpretador de linha de comando, a *Bourne-Again Shell* (Bash), e sendo chamado de "GNU/Linux". De modo que, a partir disso foram sendo lançadas versões e distribuições do GNU/Linux [8], que por muitos é chamado apenas de Linux, contribuindo para que fosse o sistema operacional baseado em UNIX mais utilizado até hoje. Com base nisso, este trabalho pretende abordar múltiplas questões relacionadas a uma das componentes desse sistema, a *shell*.

1.2 História da Shell

A *shell*, como mencionado anteriormente, é o interpretador de comandos ou linhas de comando do sistema operacional UNIX que foi criada em 1971, quando era chamada de *shell V6*. Embora o seu projeto de implementação já existisse desde o sistema Multics, foi apenas quando surgiu o primeiro sistema UNIX que ela foi de fato acrescentada ao sistema. Dessa forma, foi utilizada como referência para as que viriam.

Anos mais tarde, em 1977, nascia a chamada *Bourne shell* com a função de executar comandos de forma interativa, que podiam ser aninhados em linhas de comando ou para criação

de *scripts*. Na mesma época, a *C-shell* era desenvolvida por um grupo de pesquisa da universidade da Califórnia, no Estados Unidos da América, com o objetivo de possuir *scripts* similares aos da linguagem C. Além disso, também possibilitava a visualizção do histórico dos comandos [3].

Ainda por volta da metade da década de 70 até 1980, surgia a *Korn shell*, que também podia ser usada como linguagem de *script*. Ela era compatível com versões anteriores do *Bourne shell* original, que passou a ser pouco utilizada quando a *Bourne-Again shell* foi criada em 1988. Popularmente chamada *Bash*, é um dos interpretadores mais utilizado em sistemas baseados em UNIX. Sendo assim, será utilizada como referência para explicação de comandos, funcionalidades e resultados de comandos neste trabalho.

1.3 BASH e terminal

O terminal, que é o interpretador de linha de comando, é a *shell* em sistemas baseados em UNIX, como o GNU/Linux, e uma de suas linguagens próprias mais utilizada é a *Bash*. Esse terminal pode ser inciado com o atalho de teclado **CTRL+SHIFT+T** e terminado pressionando **CTRL+D** ou digitando *exit*, sendo o tempo em que foi usado de forma interativa (entre início e término) chamado de sessão [11].

As linhas de comando normalmente são constituídas por uma ou mais palavras, em que a primeira é o comando a ser executado e as outras (se existirem) são argumentos (parâmetros) ou opções para esse comando. De modo que, os argumentos são nomes de elementos sobre os quais o comando deve agir e as opções (argumentos especiais) fornecem ao comando informações adicionais sobre o que deverá ser feito. Normalmente, são compostas por um traço (-) seguido de uma letra.

2 Arquivos, Coringas e Redirecionamento

Embora nem sempre os argumentos de comandos sejam arquivos, os arquivos são os elementos mais importantes em um sistema UNIX. Dessa forma, sua manipulação é um evento bastante frequente para programadores, seja para criação, remoção ou alteração dos próprios arquivos ou apenas de seu conteúdo. Existem dois principais tipos de arquivos:

- **Arquivos regulares**: contém caracteres legíveis para um humano e também podem ser chamados de arquivos-texto.
- **Arquivos executáveis**: também conhecidos como programas, são arquivos invocados como comandos (explicados a seguir) e a maioria não é legível aos humanos. Já outros, como os shell scripts, são apenas arquivos-texto especiais.

Esses arquivos são armazenados em diretórios que, além dos arquivos, podem conter sub-diretórios. Sendo assim, cria-se uma estrutura hierárquica conhecida como *árvore*, onde o diretório mais acima é a *raiz*. Dessa forma, todo arquivo pode ter nomeada sua localização no sistema em relação a *raiz* ao listar todos os diretórios (em ordem, a partir da *raiz*) separados por barras (/), seguido do nome do arquivo. Essa nomeação é chamada de *caminho absoluto*. Dessa forma, o acesso e manipulação desses arquivos e diretórios pode ocorrer via terminal.

O comando mais utilizado para listagem de nomes de arquivos e diretórios contidos no diretório corrente, é o **ls**, o qual possui opções que mostram mais informações. Por exemplo, **ls -l**, que devolve permissões, proprietário, grupo, tamanho e data/hora modificada, e **-ha**, que devolve o tamanho de forma legível para um humano e inclui entradas começadas com **.** (ponto). Além disso, listagens ordenadas por tamanho e por data/hora de modificação podem ser obtidas com a opção **-S**, na qual o maior aparece primeiro, e com **-t**, com o mais recente primeiro, respectivamente.

Arquivos e diretórios vazios podem ser criados com as linhas de comando `touch nome_do_arquivo.txt` ou `> nome_do_arquivo.txt` (o símbolo **>** será explicado mais para frente), e `mkdir nome_do_diretório`, respectivamente. Vale ressaltar que o **touch** cria um arquivo vazio apenas se ele não existir e caso contrário, apenas atualiza sua data de modificação. Indo além, a opção **-p** permite que seja verificada a existência de um diretório antes de criá-lo, podendo ser acessado com **cd**. A verificação do nome do diretório corrente é dada por **pwd**.

Já a cópia de arquivos é feita utilizando **cp**, que deve receber como argumento a origem e o destino. Para diretórios, esse comando deve ser utilizado com a opção **-R** uma vez que

pode conter arquivos e sub-diretórios [2]. De forma análoga, o comando **mv** é utilizado para mover arquivos e diretórios, mas também para renomeá-los. E da mesma maneira que arquivos e diretórios podem ser criados, eles podem ser removidos, sendo o comando **rm** utilizado para arquivos e a opção **-r** para a remoção recursiva de algum diretório [2].

A exibição de arquivos pode ocorrer com o comando **cat**, que mostra o tanto de caracteres que pode-se observar na tela. Mas caso o arquivo seja maior que essa quantidade máxima, basta ir pressionando *enter* até o final do documento com o comando **more**, *b* para voltar uma ou mais páginas e *q* para sair (quit). Além dos comandos **head** e **tail**, usados para visualizar apenas as dez primeiras e últimas linhas do arquivo, respectivamente. Ademais, também permitem estipular outras quantidades de linhas com a opção **-n** e a quantidade de bytes com a opção **-c** [9].

Além disso, arquivos e diretórios também podem ser procurados a partir do comando **find**. Ele possui várias opções que permitem a busca de arquivos, que normalmente são buscados com a opção **-iname**, sendo que a string após a opção deve casar com nome do arquivo e ainda, o *i* significa que pode ser tanto minúsculo quanto maiúsculo. Ou então para procurar arquivos que contém números, deve-se acrescentar o número desejado após o **-inum** [2].

2.1 Coringas e Expansões

Comandos executados apenas com suas opções disponíveis, muitas vezes, não são o suficiente para a ação que se deseja como resultado da execução. Para isso, existem *coringas*, como colchetes e chaves, que auxiliam para um resultado mais específico, além das expansões de caminho. Na Tabela 2.1, são apresentados os coringas e suas funções.

Coringa	**Substituição**
?	qualquer caracter único
*	qualquer sequência de caracteres
[*conjunto*]	qualquer caracter dentro do *conjunto*
[!*conjunto*]	qualquer caracter fora do *conjunto*
{*termo*}	parte do nome por *termo*

Tabela 2.1: Coringas básicos (adaptado de [11]).

Por exemplo, digamos que no diretório corrente existam os arquivos *bst.c*, *bst.log*, *bst.o* e *bst.h*, então o comando `ls bst.?` irá devolver todos os arquivos, excluindo *bst.log*. Já o comando `ls bst.*` devolve todos [11]. Assim, para melhor entendimento do coringa presente no último comando, a Tabela 2.2 demonstra alguns exemplos de seu funcionamento. Considere os arquivos *alice*, *clarice*, *daniel*, *joice* e *lucas*, presentes no diretório de trabalho.

Expressão	Substituição
*e	alice, clarice, joice
l	alice, clarice, daniel, lucas
l*	lucas
*	alice, clarice, daniel, joice, lucas

Tabela 2.2: Utilizando o coringa * (adaptado de [11]).

O penúltimo coringa remanescente é o construtor de *conjuntos*. Esse conjunto é uma lista de caracteres, um intervalo inclusivo, ou ainda a combinação dos dois. Um exemplo comum de uso das listas, é quando queremos casar o nome do arquivo com um certo número de letras, por exemplo **[abc]**, que casa com *a*, *b* ou *c*. Já os conjuntos mais utilizados são o intervalo de números, denotado por **[0-9]**, e o "intervalo"de letras minúsculas, **[a-z]**, e maiúsculas, **[A-Z]**. Vale ressaltar que cada *conjunto* é usado para substituir um único caracter, de forma que, no exemplo dado acima, seria necessário incluir **[a-z]** três vezes para encontrar o arquivo *bst.log*.

E por fim, as chaves, que contêm um ou mais termos separados por vírgulas. Cada termo deve ser exatamente o nome de algum arquivo ou outro coringa. Assim, no exemplo dado acima, a expressão **{*.c,*.o,*.log}** poderia ser acrescentada ao **ls** para procurar os três programas [1]. Esse processo de combinar expressões que contem coringas com nomes de arquivos, é chamado *expansão*. Além disso, as expansões fazem parte de um conceito mais geral chamado *expansão de caminho*, uma vez que é possível usá-las como parte de um caminho.

2.2 Entrada/Saída e Pipelines

Em sistemas UNIX, entrada e saída de dados é tudo que faz parte do sistema que produz ou aceita algum tipo de dado, tendo como principal objetivo atender as demandas dos usuários de terminais. Sendo assim, para cada shell existem três tipos de *I/O* (do inglês, input/output) padrão: *entrada padrão* (teclado), *saída padrão* (tela/janela) e *erro padrão* para as mensagens de erro (tela/janela). Eles podem ser redirecionados de ou para um arquivo, fazendo parte de todo e qualquer programa que pode ser invocado/executado (inclusive comandos) [11].

Então digamos que os comandos não aceitem nomes de arquivos como argumentos e só aceitem dados da entrada padrão. Dessa forma, a shell permite o redirecionamento da entrada padrão de modo que a entrada venha de um arquivo, onde a notação `cat < nome_do_arquivo` faz isso. De forma análoga, a notação `cat > nome_do_arquivo` redireciona a saída padrão do comando para o arquivo, e assim, o **cat** pode ser usado para concatenação de arquivos [11].

Além disso, também pode-se aninhar comandos em programas, conhecidos como scripts, ou em um *pipeline*. Nele, a saída de um programa serve como entrada para o próximo, possibilitando o uso de utilidades do UNIX como blocos de construção para programas maiores. Muitos desses utilitários do UNIX devem ser utilizados desta maneira, de forma que cada um

deles executa um tipo específico de filtragem no texto de entrada, sendo essenciais para o uso produtivo do shell. Os mais utilizados podem ser vistos na Tabela 2.3.

Utilitário	Fim
cat	imprime a entrada na saída padrão; concatena arquivos
grep	procura por strings na entrada
sort	ordena as linhas da entrada
cut	extrai colunas da entrada

Tabela 2.3: Utilitários mais utilizados (adaptado de [11]).

O construtor que permite a formação dos *pipelines* é o *pipe*, representado por **|** e fazendo a junção de dois ou mais comandos em uma linha. Por exemplo, a linha `ls -l` lista detalhadamente o conteúdo do diretório corrente e pode ser utilizada na forma `ls -l | more` para mostrar a lista uma janela por vez. Sendo assim, essas junções podem se tornar complicados e também combinadas com operadores de *I/O*, por exemplo, `cat nome_do_arquivo | more` (o < foi omitido pois **cat** recebe nome de arquivo como argumento).

Um exemplo mais complexo é o arquivo */etc/passwd* que contém informações sobre as contas em um sistema UNIX. Nele, cada linha é formada pelo nome de login, número identificador do usuário, senha criptografada, diretório da home, login da shell e outras informações, sendo as colunas separadas por dois pontos (:). Assim, para que seja possível obter uma lista ordenada dos usuários do sistema, a linha `cut -d: -f1 /etc/passwd | sort`. Dessa forma, pipelines permitem a execução de mais de um programa/comandos utilizando apenas uma linha.

Apesar disso, essa forma faz com que esse pipeline tome conta do terminal, fazendo com que outros comandos não possam ser executados até que a primeira linha de comando seja terminada. Porém, caso mais comandos devam ser executados, é possível adicionar um e comercial (&) após o comando, rodando o comando em *background* (segundo plano). Inclusive, se houver entrada ou saída padrão, ela pode ser redirecionada de ou para arquivos da mesma forma que para comandos executados em *foreground*.

2.3 Exercícios

Tendo isso em vista, a seguir são propostos três exercícios referentes aos assuntos discutidos neste capítulo. Cada um dos comando citados, referenciados ou exemplificados, possui uma página do *bash* com todas as opções utilizáveis, códigos de saída, expressões regulares, descrição, entre outros, que pode ser acessada digitando `man` *`comando`*. (Todos comandos do *bash* possuem essa *man page*.)

1. Crie o diretório *teste* e dentro dele crie três arquivos texto vazios: sinonimo, tradutor e funcao.

Resposta:

```
mkdir teste

cd teste

touch sinonimo.txt

touch tradutor.txt

touch funcao.txt
```

2. Liste todos os arquivos texto presentes no diretório corrente que comecem com letra minúscula e não contenham dígitos no segundo caracter do nome.

   Resposta:

   ```
   ls [a-z][!0-9]*.txt
   ```

3. Concatene os arquivos criados do exercícios 1 em um novo arquivo chamado saida.txt que contém um pequeno texto que não pode ser sobrescrito. Suponha que os arquivo criados possuem conteúdo de no mínimo cinco linhas.

   Resposta:

   ```
   cat sinonimo.txt tradutor.txt funcao.txt >> saida.txt
   ```

3 Customização do Ambiente

Ambiente é um conjunto de conceitos que expressam as ferramentas de um computador e também onde os programas são executados. Os conceitos são apresentados pela shell como arquivos, diretórios e entrada e saída, onde as ferramentas são os editores e comandos de manipulação. Sua aparência é definida pelo modo como são configurados os diretórios, onde cada arquivo é armazenado e que nomes eles recebem, mas também pelo teclado e tela.

Além disso, existem quatro recursos que são os mais importantes quanto a customização de ambiente bash. Eles são os arquivos especiais *.bash_profile*, *.bash_logout* e *.bashrc* executados no login e logout ou cada vez que uma nova shell é iniciada, os *aliases*, que são sinônimos para comandos definidos por conveniência, as opções, que são controles que podem ser ativados ou desativados, e por fim, as variáveis, que possuem determinados comportamentos de acordo com seus valores [11].

3.1 Arquivos especiais

Dos três arquivos, o *.bash_profile* é o mais importante, uma vez que é executado pelo bash toda vez que o login é realizado. Recomenda-se que as linhas existentes não sejam modificadas até que sua função seja completamente compreendida, embora novas linhas possam ser acrescentadas. Vale ressaltar que suas modificações terão efeito apenas no próximo login, embora a execução do comando `source .bash_profile` seja uma alternativa [11].

Além disso, quando uma nova shell (uma subshell) é iniciada ao digitar `bash` na linha de comando, os comandos lidos serão do arquivo *.bashrc*. O que permite flexibilidade, de forma que os comandos que devem ser executados no login e os que devem ser executados em uma subshell possam ser mantidos separados. Já o arquivo *.bash_logout* é executado toda vez que é feito logout, podendo ser utilizado, por exemplo, para remover arquivos temporários.

3.2 Aliases

Muitas vezes os comandos mais utilizados são os que são usados com uma string de opções e argumentos que precisam ser especificados. Para que seja mais fácil invocá-los, existe uma ferramenta bash que permite renomear esses comandos chamada **alias**. Aliases podem

ser definidos em uma linha de comando nos aquivos *.bash_profile* ou *.bashrc*, com a forma `alias nome=comando`. Assim, toda vez que **nome** é digitado como comando, a bash executará o **comando** definido após o igual (=).

Um dos usos mais básicos para um alias, é utilizar nomes mais mnemônicos para determinados comandos. Por exemplo, o comando **grep**: uma utilidade de busca cujo nome surgiu de *Generalized Regular Expression Parser* e que pode ser trivial para um cientista da computação, mas não para um administrador de escritório, que usaria o alias `alias search=grep` com mais facilidade. Outro uso básico de aliases, são para erros tipográficos na hora de invocar alguns comandos.

Além disso, comandos possuem opções que devem ser escritas maiúsculas e pode ser inconveniente digitar **-** seguido de uma letra maiúscula. Por exemplo o comando **ls**, que possui a opção **-F** para mostrar uma barra (/) após cada diretório e um asterisco (*) após executáveis. Dessa forma, `alias lf='ls -F'`, é um alias largamente usado [11].

Porém, alguns pontos sobre aliases devem ser lembrados. Primeiro, que a bash faz uma substituição textual, de forma que o comando é repassado a um processador de texto antes de ser interpretado e executado. Sendo assim, caracteres especias, como coringas, são interpretados de forma correta. Segundo, um alias é recursivo, de modo que é possível criar um alias para um alias, mas não um loop infinito. Por exemplo, `alias pa=printall` é um alias para `alias printall='pr * | lpr'`.

3.3 Opções da Shell

As opções da shell permitem mudar de fato o comportamento da shell, pois elas podem ser 'ligadas' e 'desligadas' através de `set -o nomeopção` e `set +o nomeopção`, respectivamente. O motivo do uso contraintuitivo dos sinais, é por que o traço (-) é o modo convencional de especificar alguma opção de comando. Além disso, o estado de todas pode ser obtido com o comando `set -o`. Sendo assim, na tabela 3.1 são vistas algumas das opções básicas (*nomeopção* citado anteriormente).

Opção	**Descrição**	**Estado padrão**
emacs	Aciona o modo *emacs* de edição	Ligado
ignoreeof	Logout ocorre apenas ao digitar **exit**	Desligado
noclober	Não permite que o redirecionamento (>) sobre-escreva um arquivo existente	Desligado
noglob	Não expande coringas como * e ?	Desligado

Tabela 3.1: Opções básicas de shell (adaptado de [11])

Outra ferramenta com uma finalidade parecida é o **shopt**, que substitui as configurações de opções através de set e variáveis de ambiente. A funcionalidade **shopt -o** é uma duplicação de

partes do comando set. Sendo usado da forma `shopt` *`opções nomeopção`*, onde a opção **-p** lista cada opção e seu valor corrente. Além disso, as opções **-s** e **-u** sem *nomeopção* retornam as opções habilitadas e desabilitadas, respectivamente [5]. Algumas de suas opções (como de comandos) são vistas na tabela 3.2.

Opção	**Descrição**
-s	Habilita (set) toda *nomeopção*
-u	Desabilita (unset) toda *nomeopção*
-q	Suprime a saída padrão, o retorno é se *nomeopção* esta habilitada ou não (zero se todas estão habilitadas, diferente de zero caso contrário)
-o	Permite que o valor de *nomeopção* seja igual ao do comando set -o

Tabela 3.2: Opções do shopt (adaptado de [11])

3.4 Variáveis da Shell

As variáveis são características do ambiente que não podem ser apenas ligadas/habilitadas e desligadas/desabilitadas, permitindo especificar tudo, desde seu prompt até em quão frequente é checada sua caixa de entrada de e-mails. Ademais, possuem um valor associado. Por convenção, seus nomes devem ser escritos em caixa alta, com duas exceções, e a sintaxe para definição é parecida com a dos aliases: `nomevar=valor`.

Além disso, a utilização do valor em um comando, deve ter um cifrão ($) precedido do nome. Sendo assim, para saber o valor de alguma variável, o comando **echo** é utilizado. Por exemplo, se a variável **wonderland** possui o valor **alice**, `echo "$wonderland"` fará a shell imprimir **alice**, e se não tiver valor definido, a shell imprimirá uma linha em branco [11].

3.5 Exercícios

Isto posto, a seguir são propostos três exercícios referentes aos assuntos discutidos neste capítulo. Cada um dos comando citados, referenciados ou exemplificados, possui uma página do bash com todas as opções utilizáveis, códigos de saída, expressões regulares, descrição, entre outros, que pode ser acessada digitando `man` *`comando`*. (Todos comandos do bash possuem essa *man page*.)

1. A partir do diretório 'Documentos', crie o diretório 'ci320/relatorios/trabalho4'. Após isso, crie um alias para esse último diretório e depois entre nele utilizando o comando

cd. Atenção: crie os aliases apenas na shell corrente, ou se criar em algum dos arquivos especiais, remova-os depois.

Resposta:

```
mkdir -p ci320/relatorios/trabalho4

alias dir='ci320/relatorios/trabalho4'

alias cd='cd '

cd dir
```

2. Faça com que pequenos erros de digitação no nome do diretório para o qual você deseja fazer um **cd** serão ignorados. Por exemplo, faça o comando 'cd Docments' entrar no diretório 'Documentos'. Dica: use alguma das opções da shell com o comando **shopt**.

   Resposta:

   ```
   shopt -s cdspell
   ```

3. Crie uma variável chamada 'chuva' e atribua o valor 'Dias de chuva necessitam de mais espaços entre as palavras.' (com quatro espaços entre duas quaisquer palavras da frase). Depois imprima a variável exatamente como atribuído.

   Resposta:

   ```
   chuva="Dias de chuva necessitam de mais espaços entre as"palavras.

   echo "$chuva"
   ```

4 Navegação sem o uso do Mouse

Vários comandos que foram explicados e exemplificados neste trabalho (até aqui) são úteis para manipular (remoção, edição, renomeação, cópia, etc) a hierarquia de diretórios e arquivos utilizando apenas o terminal e seus comandos via teclado. Como já mencionado, é aberto com **CTRL + ALT + T**. Além disso, uma nova aba é aberta na mesma janela pressionando **CTRL + SHIFT + T**. Sendo o acesso entre diferentes abas feito com **ALT + n**, onde **n** é o número da aba, que por sua vez, pode ser fechada utilizando **CTRL + SHIFT + W** ou **exit**.

Existe uma variação do terminal chamada Terminator [6], que quando instalado é aberto da mesma forma que seu terminal-pai. Nele, uma nova aba é aberta com o mesmo comando no teclado, porém a alternação é feita com **CTRL + pgdn** ou **CTRL + pgup**. No entanto, o terminator possui a funcionalidade de arranjar vários terminais (sub-shell) em grid, onde **CTRL + SHIFT + E** e **CTRL + SHIFT + O** dividem a aba corrente vertical e horizontalmente, podendo ser fechadas da mesma forma que no terminal. Vale ressaltar que as novas abas e divisões possuem o mesmo diretório corrente a partir de onde foram abertas.

Assim como é possível alternar entre abas/divisões, é possível alternar entre janelas de terminal e janelas de outros aplicativos utilizando a combinação **ALT + TAB**. Essas janelas também podem ser trocadas de área de trabalho, que podem ser chaveadas com **CRTL + ALT + setas**, combinando **CTRL + SHIFT + ALT**. Além disso, também é viável aumentar ou diminuir a fonte de uma janela/aba de terminator com **CTRL + SHIFT + +** e **CTRL + -**, respectivamente. Vale ressaltar que esses atalhos de teclado funcionam em sistemas GNU/LINUX, inclusive se comportam de forma muito semelhante quando lidando com dois monitores.

No entanto, ao utilizar comandos e linhas de comando em uma shell UNIX no terminal ou no terminator, erros de sintaxe podem ser críticos. Uma vez que a sintaxe da shell é poderosa, mas também concisa e cheia de caracteres especiais que podem ser utilizados de diferentes formas, ela não é muito mnemônica e permite construções de linhas de comandos que são tão obscuras quanto complexas [11]. Felizmente a linguagem *bash* possui um histórico de comandos que possibilita a recuperação de linhas de comando e evita a reescrita das mesmas.

Além desse histórico, *bash* permite editar comandos e linhas de comandos com comandos similares aos dos que pertencem aos dois editores mais populares do UNIX: *vi* e *emacs*. Dos quais, o primeiro possui comandos geralmente compostos por letras sozinhas ou em duplas, sendo elas maiúsculas e minúsculas, ou nas duas formas. Apenas em alguns casos são utilizados caracteres especiais. Já o segundo modo, tem comandos que geralmente combinam as teclas

CTRL e **ESC** com alguma letra ou símbolo, sendo um modo de edição mais facilmente entendido e usado.

4.1 Exercícios

Isto posto, a seguir são propostos três exercícios referentes aos assuntos discutidos neste capítulo. Cada um dos comando citados, referenciados ou exemplificados, possui uma página do bash com todas as opções utilizáveis, códigos de saída, expressões regulares, descrição, entre outros, que pode ser acessada digitando `man` *`comando`*. (Todos comandos do bash possuem essa *man page*.)

1. Leia o capítulo 2 do livro "Learning the bash shell: Unix shell programming", dos autores Cameron Newham e Bill Rosenblatt [11].

2. Verifique qual é o modo de edição padrão da sua shell. Então habilite o outro modo de edição e depois volte para o padrão. (Considere que a resposta tem como modo padrão o emacs.)

   Resposta:

   `set -o`

   `set -o vi`

   `tset -o emacs`

3. Quais são os comandos de mover o cursor um caracter para frente e para trás dos modos de edição *emacs* e *vi*, respectivamente?

   Resposta:

   emacs: `CTRL + F` e `CTRL + B`

   vi: `h` e `l`

5 Controle de Processos

Como mencionado, UNIX é sistema operacional multiusuário e multitarefa, permitindo que vários usuários acessem um mesmo recurso de uma vez e também que cada usuário tenha controle sobre mais de um processo. Sendo assim, no UNIX cada processo recebe um ID (identificador) quando criado, o que é facilmente verificado quando um comando é executado em background.

Nesse caso, sempre aparece um número entre colchetes ([e]) logo após o comando ou na linha logo abaixo, seguido de outro número, como em [1] 93. Onde o 1 entre colchetes significa o *job number*, que é o identificador dos processos rodando em background, e o 93 é o ID do processo, atribuído a partir de todos os processos de todos os usuários [11].

Além disso, quando um processo desse tipo é terminado, uma mensagem com o número de *job* é mostrada. Ela pode significar que o processo terminou de forma normal com *done*, ou de forma anormal com *exit*. De um jeito ou outro, além desse número, também aparece o nome do comando ou da linha de comando executada.

5.1 Controle de *Jobs* e *Signals*

O modo mais óbvio de controlar um job é soltar ele em background e uma vez que esteja executando, deixá-lo terminar, trazê-lo para o foreground ou mandar uma mensagem chamada *signal*. Onde o segundo pode ser feito a partir do comando **fg**, normalmente significando que o processo em questão irá tomar conta do terminal.

Sendo assim, quando só um processo está em segundo plano, ele é o único que pode ser trazido pata o primeiro plano. Porém se existirem mais, o comando **fg** irá invocar o processo colocado em segundo plano mais recentemente e caso seja necessário, o comando pode ser executado seguido de algum *job number* como argumento.

Para isso, os números de *job* devem ser conhecidos, mas caso não sejam, o comando **jobs** retorna a lista desses números juntamente com o comando sendo executado. Além disso, se mais de um processo possuir o mesmo comando, o que foi colocado mais recentemente em baskground pode ser trazido para o primeiro plano com um `fg % comando`.

De forma análoga, um *job* também pode ser colocado para continuar executando em background com **bg** após ter sido suspenso com **CTRL + Z**. Desse modo, o *job* executa até o final ou deve ser matado utilizando um *signal* (explicado mais adiante). Essa ferramenta é

bastante utilizada quando se quer utilizar apenas um terminal e há a necessidade de editar um programa/script e também executá-lo [11].

Ainda para tomar conhecimento do ID do processo, é possível invocar o comando **ps**, que possui inúmeras opções que podem se comportar de formas diferentes de uma versão do UNIX para outra. Esse comando é complexo, porém muito útil na hora da necessidade de terminar um processo a partir de seu identificador para o sistema e não apenas da shell corrente.

Atalhos como **CTRL + Z**, que suspende um *job*, e **CTRL + C**, que mata um *job*, são casos particulares de mandar um *signal* a um processo. Esse *signal* é uma mensagem mandada de um processo a outro quando há um comportamento considerado anormal ou quando um deles quer que o outro faça algo. Por exemplo, o **CTRL + C** nada mais é que um sinal **INT** e **CTRL + Z** é um sinal **TSTP**.

Também há o sinal **KILL**, que serve para situações onde nenhum outro jeito de parar/terminar processos funcionou e que pode ser enviado a qualquer processo criado, tendo como argumento o ID do processo. Normalmente, funciona como um sinal **TERM**, mas pode ter o sinal especificado pelo seu nome ou número precedido de um traço (-).

Além disso, para o controle de sinais e processos de dentro de scripts em bash o comando **trap** é utilizado. Ele é o comando mais importante na hora de proteger esses scripts e programas shell, pois permite que eles tenham comportamentos apropriados em relação a condições anormais, assim como programas em qualquer outra linguagem [11].

5.2 Co-rotinas

Controle de processos é realizado diretamente na shell (terminal), como já posto, mas também dentro de programas shell. E quando dois ou mais deles executam ao mesmo tempo com possibilidade de comunicação, são denominados co-rotinas. Essas co-rotinas nada mais são que o recurso utilizado por pipelines, que por sua vez encapsulam uma série de regras sobre como processos devem se comunicar.

Um exemplo seria o pipeline `ls | more`, no qual (1) são criados dois subprocessos, P1 e P2 (chamada ao sistema fork), (2) é configurada a entrada/saída entre os processos de modo que a saída de P1 sirva como entrada para a entrada de P2 (pipe), (3) dá início ao ls (exec), (4) dá início ao more (exec) e (5) aguarda o término dos processos (wait).

Mas caso não haja comunicação entre processos executados simultaneamente, o controle é um pouco mais fácil. Por exemplo, se dento de um script há um comando executado em background (utilizando &) e ele continuar rodando após o término do script em si, ele se torna órfão e é preferível que isso não aconteça. Assim, para que isso seja evitado, utiliza-se o comando **wait** sem argumentos, afim de esperar até que todos os processos em background estejam terminados [11].

5.3 Sub-shells

Outro tipo de relação inter-processo, é a de uma sub-shell e sua shell-pai. Das duas, a primeira herda da segunda (como já mencionado) o diretórios corrente, variáveis de ambiente, os descritores de arquivos e sinais ignorados, além de uma parte das variáveis da shell e como são tratados os demais sinais. Além disso, quando um script shell é executado ele cria uma sub-shell.

5.4 Substituição de Processos

Quando há a necessidade de comparar a saída de duas versões de um mesmo programa ou de dois comandos, uma alternativa é executar cada versão/comando, redirecionar as saídas para arquivos e então comparar esses arquivos de saída. No entanto, esse processo pode demorar muito tempo pois é requerido esperar os dois processos terminarem, buscar os arquivos e só então comparar a saída.

Para isso, existe uma ferramenta bash chamada substituição de processos, que espera os dois processos terminarem e já compara as saídas, sem a necessidade de arquivos intermediários. Essa substituição é a entrada para um processo ou a saída de um processo, feita com o uso dos símbolos de redirecionamento < e >, respectivamente. Por exemplo, a comparação da saída de dois **ls** de diretórios diferentes `diff <(ls primeiro_dir) <(ls segundo_dir)` [11, 4].

5.5 Exercícios

Isto posto, a seguir são propostos três exercícios referentes aos assuntos discutidos neste capítulo. Cada um dos comando citados, referenciados ou exemplificados, possui uma página do bash com todas as opções utilizáveis, códigos de saída, expressões regulares, descrição, entre outros, que pode ser acessada digitando `man` *`comando`*. (Todos comandos do bash possuem essa *man page*.)

1. Verifique quais processos estão rodando na shell corrente, com um comando que mostre o PID, a TTY e o comando sendo executado.

   Resposta:

   `ps`

2. Escreva uma utilidade que permite fazer várias cópias de um mesmo arquivo de forma paralela. Considere que o usuário não passe destinos de mesmo nome. (Leia a seção 8.5 do livro "Learning the bash shell: Unix shell programming", com autores Cameron Newham e Bill Rosenblatt [11].)

   Resposta:

```
#!/bin/bash

file=$1
shift
for dest in "$@"; do
    cp $file $dest &
done
wait
```

3. Pesquise sobre as ferramentas **top** e **htop**.

6 Exemplo Práticos

6.1 Problema 1

A descrição, código e local dos itens pertencentes ao patrimônio do DInf (Departamento de Informática da UFPR) estão armazenados em um único arquivo, sendo que algumas linhas possuem campos com ponto e vírgula entre as aspas. Dessa forma, foram propostos os seguintes problemas:

Parte 1 : eliminar automaticamente as linhas do arquivo de entrada (patrimonio.csv) que contenham o símbolo do ponto e vírgula (;) que estão entre aspas;

Parte 2 : obter em um arquivo separado a lista de locais (coluna 5 do arquivo) de forma única e ordenada;

Parte 3 : para cada código de local da lista de locais obtida na parte 2 (exemplo de local: 2100.13.01.02), criar um arquivo cujo nome seja o código com a extensão .csv (exemplo de arquivo: 2100.13.01.02.csv) que contenha todas, e apenas, as linhas do arquivo de entrada que contenham este padrão. (exemplo de conteúdo para o arquivo 2100.13.01.02.csv, ele contém todas as linhas que contém a string "2100.13.01.02").

6.2 Solução problema 6.1

Para solução do problema proposto na seção anterior, foi utilizado um script na linguagem bash (The Bourne Again Shell), que pode ser visto a seguir.

```
1  #!/bin/bash
2
3  declare -r source_file="patrimonio.csv"
4  declare -r locals_file="locais.txt"
5  declare -r tmp_file="patrimonio.txt"
6
7
8  egrep '^(\"[^\;]?\"\;){5}$' patrimonio.csv > ${tmp_file}
9
10 cut -d';' -f5 ${tmp_file} | sort -u | sed 's=\"==g' > ${locals_file}
```

```
11
12
13 while read -r line
14 do
15     arq=$(echo $line | cut -d';' -f5 | sed 's=\"==g')
16     echo ${line} >> $arq".csv"
17 done < ${tmp_file}
18
19 rm ${tmp_file}
```

No código bash acima, as linhas de 3 a 5 são para a criação de variáveis utilizadas ao longo do código. A opção **-r** ali colocada, serve para que elas não possam receber um novo valor após essa primeira atribuição, uma vez que nesse caso não é necessário. Na sequência, a linha 8 possui o comando **egrep**, que é o mesmo que **grep -E**, onde o **e** significa que o **grep** receberá uma expressão regular como entrada a ser procurada dentro do arquivo 'patrimonio.csv'.

A expressão regular, explicada a seguir, é posta entre aspas simples para que seja interpretada como uma string. Primeiramente, os símbolos ^ e $ indicam começo e fim de linha, isto é, as linhas do arquivo devem iniciar com aspas duplas (") e terminar com aspas duplas seguidas de ponto e vírgula (";). A parte {5} significa que o grupo, que está entre **(** e **)** é repetido cinco vezes em cada linha.

O grupo é sempre delimitado por parenteses. Sendo assim, o grupo procurado pelo **egrep** através da expressão regular é começado em aspas, seguido de qualquer caracter (*) que não ponto e vírgula (especificado pelo intervalo entre os símbolos **[** e **]**) o menor número de vezes possível (?) e terminado em aspas seguido de ponto e vírgula. Ademais, o símbolo \ serve para escapar o caracter que segue, pois muitos desses caracteres são palavras reservadas do bash.

Desse modo, o arquivo especificado em **tmp_file** contém todas as linhas que não contém o ponto e vírgula extra entre as aspas e é utilizado na linha 10 para gerar o arquivo especificado em **locals_file**. Na qual, o comando **cut** pega a quinta coluna de cada linha do arquivo, em que as colunas são separadas por ponto e vírgula. Através do pipe (|), o comando **sort** ordena os valores obtidos dessas quintas colunas e deixa apenas uma aparição de cada valor sem aspas envolvendo-os por causa do comando **sed**.

Mas o arquivo **locals_file**, não precisaria ser criado, uma vez que o arquivo **tmp_file** pode fornecer as mesmas informações. Assim, o bloco de linhas 13 a 17, percorre esse arquivo temporário (que é apagado na linha 19), pega a quinta coluna de cada linha, retira as aspas (com comandos **echo**, **cut** e **sed** juntados com o pipe) e joga o local de determinado item na variável **arq**. Então, a linha é escrita no arquivo cujo nome é o valor de **arq** com a extensão ".csv". Vale ressaltar que com o redirecionamento **> >**, o arquivo é criado em sua primeira aparição e nas seguintes, o conteúdo é adicionado ao final do arquivo.

6.3 Problema 2

O número de matrículas em cada disciplina do DInf (Departamento de Informática da UFPR) entre os anos de 1988 (primeiro semestre) até 2002 (segundo semestre) estão armazenados em uma pasta "DadosMatricula", onde há um subdiretório para cada disciplina que o DInf ministrava no período em análise. Dentro de cada um desses, existe um arquivo para cada ano/semestre contendo dados de matrícula na respectiva disciplina.

Cada arquivo contém duas colunas: o código do curso cujo aluno se matriculou na disciplina (exemplo: a engenharia civil é o código 02) e o GRR deste aluno. O cabeçalho dos arquivos deve ser ignorado (pode ser apagado). O carcter separador de colunas é o ":"(dois pontos). Sendo assim, foram propostos os seguintes problemas:

- Construir um script que dê como saída um arquivo que contém o total de matrículas semestre a semestre considerando que os GRR's não se repetem. O objetivo é ver a evolução do número de matrículas semestre a semestre no período em análise. A notação usa a concatenação do ano com o semestre, por exemplo, o primeiro semestre de 1988 é denotado 19881.

- acompanhar a evolução do número de matrículas em cada disciplina semestre a semestre ao longo do período analisado, por isso neste caso os GRR's podem se repetir. Um mesmo GRR cursa normalmente várias disciplinas. A saída pode ser formatada de maneira que cada linha inicie com o ano/semestre (ex. 19881) e que tem em suas colunas o total de matrículas para cada disciplina.

6.4 Solução problema 6.3

Para solução do problema proposto na seção anterior, foi utilizado um script na linguagem bash (The Bourne Again Shell), que pode ser visto a seguir.

```
1  #!/bin/bash
2
3  declare -r dir="DadosMatricula"
4  declare -r final_dir="anos"
5  declare -r file_ex1="lab2_ex1.txt"
6  declare -r file_ex2="lab2_ex2.txt"
7
8
9  mkdir -p anos
10
11 declare -r file_list=$(ls $dir/CI/*.dados)
12 for i in ${file_list}
13 do
14     file=${i##*/}
```

```
15      file=${file%%.}
16      name=${final_dir}/$file".txt"
17      sed '1d' $i >> ${name}
18  done
19
20  declare -r semester=$(ls anos/.txt)
21  for i in ${semester}
22  do
23      lines=$(cut -d':' -f2 ${i} | sort -u | wc -l)
24      sem=${i##/}
25      sem=${sem%%.}
26      echo ${sem}" : "${lines} >> ${file_ex1}
27  done
28
29  declare -r courses=$(cat anos/.txt | cut -d':' -f1 | sort -u)
30  for s in ${semester}
31  do
32      sem=${s##/}
33      sem=${sem%%.}
34
35      acum[0]=${sem}
36
37      cont=0
38      for c in ${courses}
39      do
40          cont=$(( $cont + 1 ))
41
42          lines=$(cut -d':' -f1 ${s} | grep ${c} | wc -l)
43          acum[$cont]=${lines}
44      done
45      echo ${acum[@]}
46  done | column -t >> ${file_ex2}
```

No código acima, as linhas que contém a palavra reservada `declare` do bash, são para declarar variáveis. Elas são criadas somente para leitura com a opção **-r**, pois não são alteradas ao longo do código. Vale ressaltar que a atribuição pode ser uma string dada diretamente, como no caso das linhas de 3 a 6. ou pode ser resultado da execução de um comando, como no caso das linhas de 11, 20 e 29. As quais, recebem a lista de todos os arquivos com extensão *.dados*, a lista de todos os arquivos com extensão *.txt*, e a lista da primeira coluna de todos arquivos de extensão *.txt* com uma ocorrência de cada valor, nos quais as colunas são separadas por dois pontos (:), respectivamente.

O bloco `for`, que vai da linha 12 a 18, itera sobre todos os arquivos com a extensão *.dados* que estão dentro das pastas das disciplinas, e joga seu conteúdo em um arquivo cujo nome é o respectivo semestre. Sendo assim, todo conteúdo, excluindo a primeira linha (feito na linha 17 do script), dos arquivos cujo nome é *19881.dados* será armazenado em um arquivo *19881.txt* dentro da pasta *anos*. Essa pasta é criada na linha 9 e seus arquivos são iterados no segundo `for` nas linhas 21 a 27, de forma a pegar a segunda coluna dos mesmos, deixar apenas uma

ocorrência de cada valor e então contabilizar as entradas. Logo, ao final desse processo, obtem-se um arquivo em que cada linha contém um semestre e seu respectivo número de matrículas.

No último `for` das linhas de 30 a 46, os arquivos da pasta *anos* também são utilizados. Primeiramente é apenas utilizado o nome para pegar o semestre, e em um segundo momento feito dentro do for das linhas 38 a 44, são percorridos para contabilizar quantas entradas correspondem a cada um dos valores da variável da linha 29. Sendo assim, cria-se um arquivo onde cada linha possui um semestre e o respectivo número de matrículas em cada curso encontrado na primeira coluna (os valores da variável da linha 29), sendo formatado na linha 46. Nela, um aspecto interessante de blocos `for` no bash pode ser observado: como são de certa forma comandos, sua saída pode ser redirecionada a outro comando através do pipe (|)

Alguns pontos importantes dentro dos laços `for` devem ser notados e explicados. Nas linhas 14 e 15, bem como as linhas 24 e 25, e 32 e 33, são utilizadas expansões de parâmetros para guardar apenas a parte que interessa de uma string. Desse modo, a forma **##*/** remove tudo que vem antes da última ocorrência da barra (/) no caminho do arquivo, incluindo a barra e partindo do começo da string. E a forma **%%*.** remove tudo o que vem antes da primeira ocorrência do ponto (.), incluindo o ponto, partindo do final da string.

6.5 Problema 3

Um firewall gera logs diários contendo informações dos pacotes que foram barrados por casarem com alguma regra de filtragem. Um administrador de sistemas está interessado em saber o total de bloqueios por tipo de regra de filtragem, por exemplo para enviar alertas de possíveis comportamentos anormais que tipicamente podem ocorrer em casos de ataques.

O log é diário e pode ocorrer de alguma regra não ter sido filtrada em um determinado dia, neste caso, queremos que seja impresso um zero, explicitando que não houve problema para aquela regra naquele dia ao invés de simplesmente não imprimir a totalização para aquela regra. Com isso se consegue observar um histórico periódico (semanal, mensal, anual) para que se consiga observar um comportamento médio.

Dessa forma, o objetivo do trabalho é, a partir do log do firewall e das regras apresentadas como exemplo, gerar um arquivo de saída similar ao exemplo dado acima. O total de ocorrências não deve ultrapassar 20.000 (vinte mil), pois é o limite aceito pelo administrador como de filtragens normais.

Portanto, além de gravar em disco a saída diária, você deve enviar email para caso alguma filtragem ocorra além do limite. Finalmente, supondo que existe um log para cada dia, você também deve se preocupar para que os arquivos de saída tenham nomes únicos, usando para isso a data do sistema. Um cuidado com a eficiência deve ser tomado, o script não pode demorar mais do que meio segundo para o caso do exemplo fornecido.

6.6 Solução problema 6.5

Para solução do problema proposto na seção anterior, foram utilizados um script na linguagem awk para realizar todo o processamento do arquivo de log e outro em bash para enviar o email. Eles podem ser vistos a seguir.

```
#!/usr/bin/awk -f


# done before the main loop
# inicialization
BEGIN {
    # clean the temporary file
    # wich will store the types that occur over 20000 times
    print > "tmp_file.txt"
}

# main loop
# done for every line in the archive
# arr: identifier; []: access operator; $6: sixth column; ++: increment
# get an array and use the sixth column of each line to index it,
# where the sixth column its the type of occurrence
# increment the indexed value
{
    arr[$6]++
}

# done after passed trough the entire file
# prints what there is to print
END {
    for (x in arr){
        # 'x' is the index and '[x]' is the value correspondent to the index 'x'
        # (because all arrays in awk are associative)
        # prints both in the standart output
        print x, arr[x]
        # tests if the number of times that 'x' ('[x]') appears is over 20000
        # prints type and number of times to a temporary file
        if (arr[x] >= 20000){
            print x, arr[x]  >> "tmp_file.txt"
        }
    }
    # calls bash script that sends email with the types of occurrences that appear
    over 20000 times
    # in background (symbol &)
    system("./send_email.sh &")
}
```

No código em linguagem awk acima, as linhas que começam com o símbolo # e que estão em verde são os chamados comentários da linguagem e não são interpretados como comandos a

serem executados, ou seja, são ignorados pelo compilador da linguagem. Já as demais linhas, são comandos a serem executados ou indicadores de que comandos virão a seguir.

Nessa linguagem, um script é composto por três blocos delimitados por **{** e **}**. Onde o primeiro serve para inicializações e comandos executados antes do processamento do texto de entrada (geralmente um arquivo). O segundo contém comandos que serão executados para cada linha do arquivo, e o terceiro são comandos executados após o processamento de arquivo, os quais normalmente consistem de impressões.

Na linha 9 do script é visto o único comando do bloco **BEGIN** (primeiro bloco), que limpa o arquivo temporário, fazendo o redirecionamento de nada para dentro do arquivo. Seguindo para o loop principal, que pega um vetor, utiliza sua sexta coluna como índice, sendo ela o tipo de ocorrência, e então incrementa o número de vezes em que aparece no vetor (arquivo de entrada).

Por fim, o último bloco (**END**), percorre esse vetor e imprime qual o tipo de ocorrência e o número de vezes em que aparece no arquivo de entrada. Como uma notificação deve ser enviada por email para os tipos que aparecem mais de 20.000 vezes no log, eles são escritos no arquivo temporário citado acima, juntamente com suas quantidades correspondentes na linha 33 do script.

Além disso, o comando da linha 38 faz uma chamada ao script em linguagem bash, mostrado abaixo, que envia o email contendo os tipos acima do limite. Essa chamada, ocorre de forma que o script bash executa em background. Nele, o comando *mail* presente na linha 7 do mesmo envia o email, onde `-s` significa que segue o assunto do email e < redireciona o conteúdo do arquivo como corpo da mensagem.

```
1 #!/bin/bash
2
3 # send email
4 # -s: precedes the subject of the email
5 # <: redirects the file to the command,
6 # so that its contents will be the message in the email
7 mail -s "Protocols over limit, please check." jer14@inf.ufpr.br < "tmp_file.txt"
```

A título de curiosidade: o script executa em 0.3 segundo no laboratório 4 do DInf.

7 Conclusão

Após tomar conhecimento da história do UNIX, de seus sistemas filhos e também da evolução da shell e suas linguagens, foi possível conhecer conceitos referentes a esses elementos tão largamente utilizados em ambientes de ensino e aprendizado. Além disso, os conceitos são amplamente relacionados a arquivos e diretórios, coringas e expansões e entrada/saída e pipelines, que são os principais componentes de comandos e linhas de comandos. Suas combinações na linguagem bash, criam uma ferramenta poderosa de manipulação e navegação.

Sendo assim, o ambiente em que ocorrem essa manipulação e navegação deve estar devidamente configurado para que se possa fazer o uso adequado da ferramenta. O qual, sofre modificações de aparência e comportamento a partir da modificação de arquivos especiais da bash, alteração dos efeitos de variáveis e opções exclusivas da shell e ainda, da criação de aliases que facilitam o uso de comandos. Dessa forma, possibilitando um uso do ambiente gráfico sem a utilização de mouse.

E ainda, o controle de processos realizados em alto e baixo nível, para que todos os tipos de programadores shell sejam abrangidos. Esse controle ocorre diretamente na shell (terminal) com comandos e linhas de comandos. Além disso, pode ocorrer através de scripts com os mesmos comandos e linhas de comandos, mas também com comandos desenvolvidos especialmente para esses programas, e também através da comunicação existente entre os mesmos.

Referências

[1] Coringas e expressões. http://tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm. Accessed: 2018-08-10.

[2] Gerenciamento de arquivos. https://canaltech.com.br/linux/Aprenda-a-gerenciar-arquivos-no-modo-console-do-Linux/. Accessed: 2018-08-10.

[3] Linux shells. https://www.ibm.com/developerworks/br/library/l-linux-shells/index.html. Accessed: 2018-08-05.

[4] Process substitution. http://tldp.org/LDP/abs/html/process-sub.html. Accessed: 2018-11-24.

[5] Shopt options. https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html. Accessed: 2018-09-09.

[6] Terminator. https://gnometerminator.blogspot.com/p/introduction.html. Accessed: 2018-10-15.

[7] Unix history. http://webarchive.loc.gov/all/20100506231949/http://cm.bell-labs.com/cm/cs/who/dmr/hist.html. Accessed: 2018-08-05.

[8] Unix history. https://www.computerhope.com/history/unix.htm. Accessed: 2018-08-05.

[9] Visualização de arquivos. https://www.vivaolinux.com.br/dica/Visualizando-arquivos-em-modo-texto. Accessed: 2018-08-10.

[10] Wikipédia unix history. https://en.wikipedia.org/wiki/History_of_Unix. Accessed: 2018-08-05.

[11] C. Newham and B. Rosenblatt. *Learning the bash shell: Unix shell programming*. "O'Reilly Media, Inc.", 2005.