CI320 Tópicos em Programação de Computadores

Miguel Angelo Neumann Salerno

23 de Novembro de 2018

Resumo

Neste trabalho será brevemente apresentada a história do *UNIX* e do *shell*, assim como uma comparação entre o bash e o zsh e uma introdução aos comandos e funcionamentos, bem como uma explicação de como usá-los para desenvolver os conhecimentos nessa área. Também serão mostrados os atalhos mais úteis para um desenvolvedor, o básico para o controle de processos, dicas para customização e alguns exemplos da programação em *shell*. A intenção deste é evidenciar conceitos básicos e proporcionar uma fundamentação teórica sobre o conteúdo abordado.

1 Introdução

Originalmente lançado em 1969, o *UNIX* é um sistema conhecido por ter características de portabilidade e ser multitarefas e multiusuários. Ele é mundialmente usado em computadores, servidores e dispositivos móveis [22].

O *UNIX* foi criado inicialmente em 1965 por Ken Thompson e Dennis Ritchie [15] e tinha o nome de *Unics*, dado por Ken, visto que na mesma época estava reescrevendo o sistema *Multics* (*Multiplexed Information and Computing System*). Esse projeto era realizado pelo *Massachusets Institute of Technology* (MIT), pela *General Eletric* (GE) e pelos laboratórios Bell (*Bell Labs*) e *American Telephone and Telegraph* (AT&T). Os relógios do sistema começaram a funcionar em 1º de janeiro de 1970, sendo considerado a data de nascimento do *UNIX*. Em 1973 recebeu o nome de *Unix Time-Sharing System* (TSS) e em 1976 iniciou-se o uso em plataformas de universidades para fins educativos.

O que popularizou o *Unix* foi o fato de que, em 1977, começou a ser disponibilizado para ser usado em muitas empresas. A partir de 1979 o sistema já estava muito mais atualizado e muitos recursos adicionados, devido à sua grande propagação. Em 1983 foi lançada a versão comercial do *Unix*, chamada de *System V*, que é considerado até hoje como o padrão internacional. Vários desenvolvedores adaptaram o sistema às suas necessidades, já que o mesmo era considerado simples, gerando diversas versões distintas do software, como *BSD, Solaris, AIX, IRIX, HP-UX*, entre outros.

Em 1987 foi criado o *Minix*, por Andrew Tanenbaum, com o objetivo de ensinar a programação de sistemas operacionais para seus alunos. O Minix era gratuito e de código aberto, tornando possível que qualquer um pudesse fazer alterações nele. No ano de 1991 Linus Torvalds desenvolveu o GNU/Linux e disponibilizou a versão 0.2 do *kernel*, capaz de funcionar em arquiteturas i386, mais tarde (1994) foi lançada a versão 1.0 [3].

Um recurso muito interessante do *UNIX*, senão o mais importante, é o *shell* [23], um interpretador de comandos que provê uma interface com um sistema *UNIX*, ou seja, usa como entrada (*input*) os comandos digitados pelo operador e usa essa entrada para executar algum programa. Quando o programa termina a sua execução é mostrada a sua saída (*output*). Essa interface é um ambiente que pode executar programas, comandos e *scripts*, possuindo diversas versões, assim como existem várias distribuições de *Linux*. Cada versão do *shell* tem seus próprios comandos e funções.

2 O *Shell*

O *shell* [21] existe há mais tempo do que a maioria dos usuários esteve vivo. Ele está presente há tanto tempo devido ao fato de que é uma ferramenta muito poderosa e permite realizar coisas complexas com comandos simples. Mais importante, ele permite combinar programas existentes e automatizar tarefas repetitivas para que as pessoas não precisem ficar digitando as mesmas coisas várias e várias vezes.

2.1 Histórico

O primeiro *shell* [24] foi o *Thompson Shell* (tsh) feito por Ken Thompson. Este foi distribuído nas versões 1 até 6 do *Unix*, de 1971 até 1975, já não vem nos sistemas *Unix/Linux* modernos. O *PWB shell* ou *Mashey shell* foi uma atualização (uma versão modificada) do *Thompson shell* e foi criado por John Mashey. Esse *shell* foi feito com o intuito de encorajar a programação em *shell* e foram adicionados diversos recursos que o *shell* anterior não possuía.

Entre 1977 e 1979 foi desenvolvido o *Bourne shell* (sh), escrito por Stephen Bourne e foi distribuído na versão 7 do *Unix*. O sh possui muitas características básicas que, posteriormente, foram herdadas por outros *shells*. O *C shell* (csh) surgiu em 1978, criado por Bill Joy e é amplamente distribuído em versões *BSD Unix*. O csh possui uma versão melhorada, o tcsh, que é basicamente o csh com vários recursos adicionados.

O *Korn shell* (ksh), criado por David Korn em 1983, é baseado no código fonte do *Bourne shell*. Também inclui muitos recursos do *C shell* e possui muitas variantes como dtksh, tksh, oksh, mksh, sksh etc. Em 1989 foi escrito o *Almquist shell*, também conhecido como *A shell* (ash), por Kenneth Almquist. O ash é um *shell* leve e é uma variante do *Bourne shell*, posteriormente substituindo o mesmo nas versões do *Unix* lançadas no começo dos anos 90. Algumas versões modernas de distribuições do *Linux*, como o *Debian*, vêm com uma versão do ash.

O *Bourne-Again shell* (bash) foi feito por Brian Fox em 1987 como substituto gratuito do *Bourne shell*. Esse é o mais popular e mais usado dentre todos os *shells*, todas as distribuições de *Linux* vêm com o bash. Por causa de sua popularidade foi adaptado ao *Windows* e ao *Android*. O *Z shell* (zsh) foi escrito por Paul Falstad em 1990 e é uma extensão do *Bourne shell* com muitas melhorias, que incluem recursos do bash, ksh e tcsh.

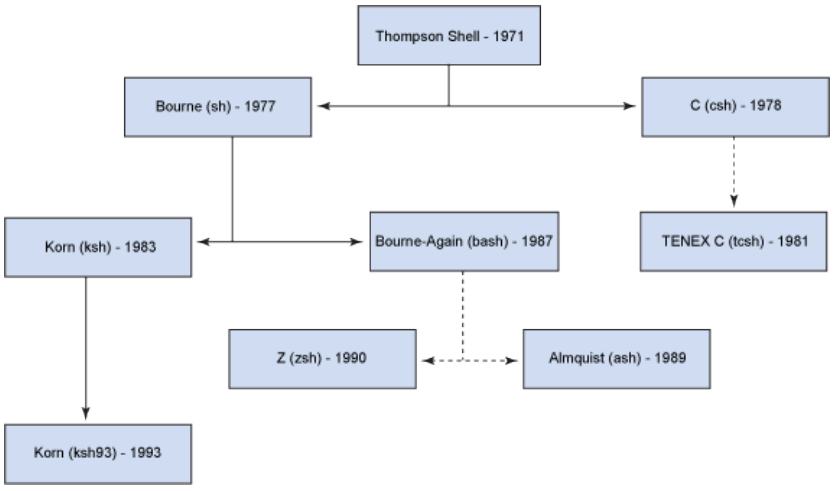

Figura 1: Evolução do *Shell* [16]

Como podemos ver na Figura 1, que apresenta a evolução do *shell*, é possível perceber quais *shells* serviram de base para a criação de novos interpretadores, tornando visível a relação entre os mesmos.

2.2 *Bash* vs *Zsh*

Tanto o bash como o zsh foram lançados na mesma época, porém, como dito anteriormente, o bash é o *shell* mais popular dentre todos, mas isso não descarta o zsh como uma alternativa poderosa e muito usada mundialmente [18].

O bash foi inicialmente feito com o objetivo de substituir o *Bourne shell* e por muitos anos tem sido o *shell* que vem na maioria das distribuições *Linux*. Alguns comandos interessantes que o bash tem como recurso:

- `Alt` + `.` : insere o último argumento do comando anterior. Se continuar pressionando, vai buscando os argumentos anteriores a esse e assim por diante por todo o histórico.
- `disown -h <pid>` : mantém um processo rodando mesmo depois de ter feito *logout*. Deve-se usar o ID do processo no lugar de `<pid>`.
- `!!` : para executar o último comando novamente.
- `Ctrl` + `r` : fazer uma busca reversa no histórico do bash.
- `Tab` : ao pressionar duas vezes é mostrada uma lista com possíveis palavras que completam a que você escreveu ou está escrevendo.
- Usar a opção `-x` ao rodar *scripts* para mostrar o conteúdo dos mesmos ao longo da execução.

Assim como o bash, o zsh também é visto como uma extensão do *Bourne shell* e tem muitas funções em comum com bash, além de lembrar um pouco do *Korn shell*. Alguns recursos do zsh que valem a pena serem mencionados:

- Englobamento: combinar partes de palavras com coringas para a formação de todas as possíveis combinações de nomes de arquivos.
- Corretor Ortográfico.
- Criar *aliases* de diretórios.
- *Scripts* para ligar e desligar por meio dos comandos zshenv, zprofile, zshrc, zlogin, and zlogout.
- *Autocomplete* de comandos do git.
- Expansão de caminhos: entre com `cd /u/lo/b` e irá completar para `cd /usr/local/bin` já que é a única opção que se encaixa.

Enquanto o zsh tem muitos usuários ao redor do mundo, ainda é mais seguro criar *scripts* para o bash, já que é amplamente mais utilizado e terá um maior número de pessoas que poderão usar esse *script*. Mais uma vez devido à popularidade do bash, ele contém mais conteúdo disponível para o seu aprendizado e materiais para ajudar a aprendê-lo.

Apesar da popularidade do bash, o zsh tem suas vantagens. Ele é muito customizável, tem o *autocomplete* mais rápido que o bash, pode apresentar um *prompt* de cada lado, como uma divisão de tela etc. Agumas das funções que o zsh tem além do bash são:

- O comando embuto `zmv` permite que você renomeie vários arquivos ao mesmo tempo. Exemplo: para adicionar ".txt"ao final de todos os arquivos seria necessário executar somente `zmv –C '(*)(q.)' '$1.txt'`.
- O utilitário `zcalc` é uma calculadora na linha de comando que é conveniente para um cálculo rápido sem sair do terminal. Para carregá-la é necessário usar `autoload -Uz zcalc` e para rodar é `zcalc`.
- O comando `autopushd` permite que você use o `popd` depois de usar um `cd` para voltar ao diretório anterior.
- Suporte para ponto flutuante.
- Suporte para estruturas de dados hash.

3 Conceitos básicos

Essa é uma introdução à programação em *shell*, recomenda-se buscar fontes mais amplas para uma fundamentação mais completa e também realizar exercícios, incluindo os que estão no final deste documento. A programação apresentada aqui é a presente no bash, ou seja, no Bourne-Again *shell*.

3.1 Comandos

Os comandos no *shell* são compostos de uma ou mais palavras, sendo a primeira palavra da linha o comando em si e as outras palavras sendo os argumentos ou parâmetros [14]. Exemplo: comando `ls`, usado para listar o conteúdo de diretórios.

- `ls`: aqui está sendo usado somente o comando, será listado o conteúdo do diretório, mostrando somente o nome dos arquivos (exceto arquivos que comecem com ".").
- `ls arquivo`: nesse exemplo será verificado se existe ou não o arquivo cujo nome foi colocado no argumento, caso exista, será listado.

- `ls -a`: aqui o comando apresenta um tipo especial de argumentos, a opção, que determina alguma função a ser executada pelo comando, como uma especificação. Nesse caso, a opção `-a` indica que todos os arquivos serão listados, inclusive arquivos e diretórios ocultos. Observação: as opções geralmente são precedidas por um traço (`-`), que é o que difere as mesmas de argumentos comuns.
- `ls -l arquivo`: esse comando listará as informações detalhadas do arquivo passado como argumento. Nesse caso o argumento é para a opção `-l`.

Esses conceitos se aplicam para quaisquer comandos no *shell*. No caso de opções, deve ser verificado se as mesmas existem para o determinado comando, já que diferentes comandos apresentam diferentes opções.

3.2 Arquivos

Arquivos são considerados o principal recurso do *shell* e podem conter diversos tipos de informações. Existem 3 tipos básicos de arquivos, são eles:

- **Arquivos comuns**: contempla os arquivos de texto e arquivos que contém caracteres legíveis.
- **Arquivos executáveis**: são programas e alguns arquivos de texto especiais que foram feitos de maneira a se comportarem como programas.
- **Diretórios**: são pastas que podem conter outras pastas e/ou outros arquivos.

3.3 Diretórios

Como dito anteriormente, diretórios são pastas que podem conter outras pastas ou arquivos. Sendo assim, podemos obter uma estrutura hierárquica de diretórios até outros diretórios ou até arquivos. Exemplo: `/home/user` ou `/home/user/arquivo.pdf`.

Para não ser necessário utilizar sempre os caminhos completos dos diretórios, o *shell* usa um recurso chamado de "diretório atual" ou "diretório de trabalho"(do inglês *working directory*) que já considera o caminho do diretório em que você está para executar os comandos, sem necessidade de acrescentá-lo, porém caso o faça, também irá funcionar.

Exemplo: O comando `ls` não tem nenhum parâmetro, então irá listar os conteúdos do diretório atual, enquanto `ls pasta/` irá listar o conteúdos do diretório "pasta".

Observação: para saber qual é o diretório atual é usado o comado `pwd` (*print working directory*) e para mudar de diretório é usado o comando `cd` *change directory*, como por exemplo `cd pasta/`.

A figura 2 mostra como são organizados os diretórios padrão no *linux*. A estrutura hierárquica em árvore é evidenciada pelo esquema da imagem e é possível verificar como funcionam os comandos de `cd` vistos anteriormente.

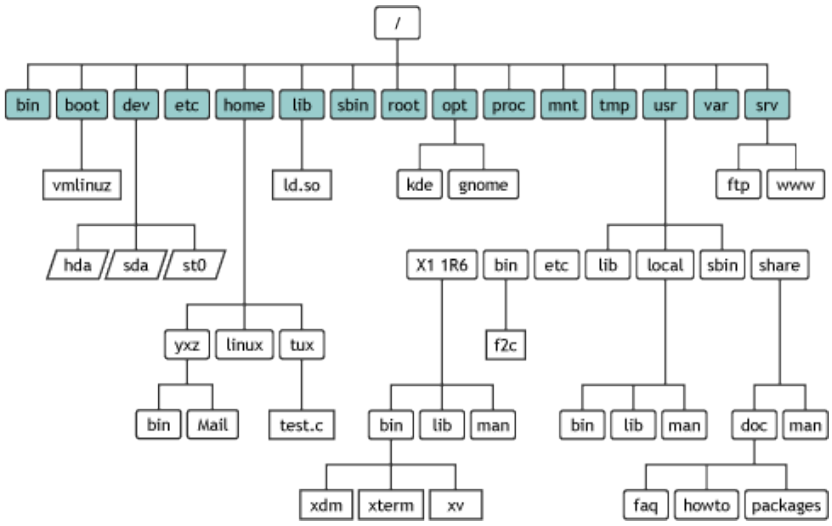

Figura 2: Diretórios do *Linux* [17]

3.4 Coringas

Os coringas são caracteres usados em comandos para representar padrões em nomes de arquivos. Dessa forma não é necessário saber os nomes dos aquivos em questão. Digamos que temos os arquivos "codigo.pdf", "codigo.txt"e "codigo.c", alguns exemplos de coringas usados para identificar os arquivos são:

- `?`: representa qualquer caractere. Exemplo: `ls codigo.?` irá retornar o arquivo "codigo.c", já que esse coringa representa somente um caractere.
- `*`: representa qualquer sequência de caracteres. Exemplo: `ls codigo.*` irá retornar os três arquivos, devido à esse coringa representar uma *string*.
- `[texto]`: representa qualquer caractere em "texto". Exemplo `ls codigo.[a-z]`, irá retornar todos os arquivos em que sua extensão é uma única letra (de a até z), incluindo "codigo.c".
- `[!texto]`: representa qualquer caractere que não está em "texto". Exemplo: `ls codigo.[!o]` irá retornar todos os arquivos que não tem sua extensão como o caracter "o".

3.5 Expansões

As expansões funcionam de maneira muito similar aos coringas, proporcionando um método para gerar todas as combinações possíveis de *strings* arbitrárias com a expansão, o prefixo (opcional) e o sufixo (opcional). Exemplo: Digamos que existam os arquivos "casa", "carroça"e "cadeira"no diretório. Se usarmos o

comando `ls ca{sa,rro,deira}`, será apresentado na saída os três arquivos em questão.

O uso de prefixo e do sufixo não é obrigatório, podendo gerar as seguintes combinações de comandos [1]:

- `{EXPANSÃO}` - Exemplo: `ls {casa,carroça,cadeira}`
- `<PREFIXO>{EXPANSÃO}` - Exemplo: `ls ca{sa,rroça,deira}`
- `{EXPANSÃO}<SUFIXO>` - Exemplo: `ls {cas,carroç,cadeir}a`
- `<PREFIXO>{EXPANSÃO}<SUFIXO>` - Exemplo: `ls ca{s,rroç,deir}a`

Todos estes exemplos resultam na listagem dos três arquivos, porém são feitos de maneira diferente, usando todas as possibilidades do comando. Também pode ser usado uma sequência de caracteres na expansão, como no exemplo `ls {1..7}.pdf` que listará os arquivos 1.pdf até 7.pdf, caso existam.

3.6 Entrada e Saída

Entradas e saídas no *UNIX* representam o que é "dado"ao programa para ele executar e o que ele "devolve", respectivamente. A entrada de dados padrão é o teclado e a saída padrão é a tela do computador. Quando dizemos que é entrada ou saída padrão, queremos dizer que é a opção que está predefinida a ser usada a não ser que outra opção seja forçada sobre esta, como no caso de redirecionamentos.

Os redirecionamentos [2] ocorrem quando é escolhida outra entrada ou saída para a execução que não sejam as padrões. Isso é feito por meio dos caracteres `>` e `<`. O operador `>` redireciona a saída do comando, criando um novo arquivo (caso não exista) e o preenche com a respectiva saída. Ainda é possível usar `>>` para concatenar a saída ao arquivo ao invés de substituí-lo. Já o operador `<` redireciona a entrada do comando, deixando de ler da entrada padrão para ler do redirecionamento especificado. Exemplos:

- `ls > arquivo.txt`: irá criar e colocar a listagem de conteúdos em "arquivo.txt".
- `ls >> arquivo.txt`: irá concatenar a listagem de conteúdos em "arquivo.txt".
- `script.sh < arquivo.txt`: irá executar o programa lendo a entrada a partir do conteúdo de "arquivo.txt".

Observação: no *shell*, CTRL-D representa "EOF"(*end of file* ou fim de arquivo) e pode ser usado para finalizar uma entrada de dados na entrada padrão.

3.7 *Pipelines*

O conceito de *pipelines* se baseia na ideia de colocar a saída de um comando na entrada de outro, direcionando os dados como num "tubo". O caractere `|` é usado para fazer o *pipe* entre comandos. Exemplo:

- `ls | grep nome`: irá retornar todas as ocorrências de "nome"na listagem de conteúdo. O comando `grep` irá buscar no resultado de `ls` pelo padrão "nome".

Pode ser realizado mais de um *pipe* ao mesmo tempo, sem restrição de quantidade, sempre usando a saída do comando anterior como entrada para o próximo comando.

3.8 Comandos associados

Alguns comandos associados [20] aos conteúdos vistos aqui serão listados para que possam ser utilizados para prática no *shell*:

- `cat`: mostra o arquivo na saída padrão. Exemplo: `cat arquivo`.
- `grep`: procura pelo padrão em arquivos. Exemplo: `grep "texto" arquivo`.
- `cp`: copia arquivos. Exemplo: `cp arquivo1 arquivo2`.
- `man`: mostra instruções para o comando procurado. Exemplo: `man ls`.
- `mv`: mover ou renomear um arquivo. Exemplo: `mv arquivo1 arquivo2`.
- `rm`: remover arquivos ou diretórios. Exemplo `rm arquivo`.
- `file`: mostra informações sobre o arquivo. Exemplo: `file arquivo`.
- `cd`: muda de diretório. Exemplo: `cd pasta/`.

4 Customização de ambiente

O *shell* permite que você personalize seu ambiente para que o mesmo esteja preparado para as suas necessidades e o seu jeito de trabalhar. Ele pode ser customizado de maneira a alterar cores, diretórios, onde você coloca tais diretórios ou arquivos ou até mesmo que nomes você dá a arquivos, diretórios e comandos. Configurar seu *shell* é muito importante para obter agilidade e facilidade ao trabalhar nesse ambiente.

4.1 Arquivos especiais do *shell*

Existem três arquivos que proporcionam um meio para configurar o seu *shell*, são eles: `.bash_profile`, `.bash_logout`, e `.bashrc`. Esses arquivos podem já existir em seu diretório *home* e caso não existam, seu sistema está usando os arquivos padrão. Esses arquivos podem ser criado e modificados em editores de texto para poder personalizar seu *shell*.

O arquivo `.bash_profile` contém configurações que serão executadas cada vez que você logar no sistema. Essas opções definem o ambiente básico para a sua conta caso queira adicionar novas opções, basta acrescentá-las no final do arquivo. As modificações terão efeito após reiniciar o *shell*.

Em `.bashrc` se definem as coisas que se aplicam somente ao bash, como aliases, definições de funções, opções do *shell* e configurações do *prompt*. [4]

O arquivo `.bash_logout` é executado cada vez que um *shell* é fechado. Basicamente é um conjunto de comandos que será executado quando sair do *shell*, como remover arquivos temporários, marcar o tempo de uso, etc.

4.2 Aliases

Alias é um recurso para poder transformar longas linhas de comando em código simples e mais fáceis de usar, usando de um sinônimo definido nas configurações do *shel*. Os alias podem ser definidos nos arquivos `.bash_profile` ou `.bashrc`, seguindo a seguinte estrutura:

```
alias nome=comando
```

Esse comando significa que "nome" é um alias para "comando", então, toda vez que você executar "nome", o *shell* vai substituir por "comando" e executar a linha. Como no exemplo a seguir, onde o comando `la` irá listar todos os arquivos e diretórios ocultos, com exceção dos diretórios `.` e `..` que também são ocultos:

```
alias la='ls -A'
```

Os aliases são muito utilizados para trocar nomes de comandos para que sejam mais intuitivos, adicionar comandos que foram digitados incorretamente para que esses erros não impeçam que o comando execute e substituir longas linhas de comando por comandos pequenos e simples.

4.3 Opções do *shell*

O *shell* permite que opções sejam definidas para o seu uso e mudem seu comportamento. Essas opções podem ser definidas como "ligadas" ou "desligadas" no ambiente. Para definir alguma opção basta executar:

`set -o opção` : para ativar
`set +o opção` : para desativar

Veremos um exemplo disso na seção 5, onde apresentamos as opções para mudar entre o modo de edição `emacs` e o modo de edição `vi`.

4.4 Variáveis

Algumas configurações não podem ser definidas como "ativadas" ou "desativadas", sendo assim necessário usar variáveis para customizar essas características. As variáveis podem ser definidas pelo comando:

```
variavel=valor
```

Para usar uma variável, coloca-se um $ antes de seu nome e para checar o valor de uma variável basta usar o comando `echo variavel` para ver seu conteúdo.

4.5 Variáveis embutidas

Algumas variáveis do *shell* são relativas ao ambiente atual e representam valores que podem ser relevantes aos usuários. Alguns exemplos de variáveis embutidas relativas ao histórico:

- `HISTCMD` : número do comando atual no histórico.
- `HISTFILE` : nome do arquivo onde é salvo o histórico de comandos.
- `HISTFILESIZE` : número máximo de linhas no arquivo de histórico.

4.6 Variáveis de ambiente

Para obter uma lista de todas as variáveis definidas no ambiente é necessário usar o comando `env`. Esse tipo de variável precisa ser conhecida por subprocessos para que possam funcionar corretamente. Uma variável de ambiente é criada por meio do comando `export`, com por exemplo:

```
export VARIAVEL="valor"
```

Alguns exemplos de variáveis de ambiente mais utilizadas:

- `HOME` : contém o caminho para o diretório `home`.
- `PATH` : contém os caminhos dos binários que podem ser executados sem ser digitado seus caminhos completos.
- `PWD` : contém o caminho do diretório atual.
- `TERM` : contém o tipo de terminal que está sendo utilizado.

Para deixar esta variável de ambiente permanentemente definida, deve-se adicioná-la ao arquivo `.bashrc`. [11]

5 Atalhos

Para um programador em *shell*, usar o *mouse* pode ser um atraso nas ações ou até mesmo uma perda de tempo. Praticamente tudo o que precisa ser feito no *shell* ou no Linux pode ser feito via teclado, sem a necessidade de um *mouse*.

O *mouse* está associado a interface gráfica do sistema, ou seja, com a aparência e atalhos mais intuitivos. Se voltarmos às primeiras versões do Linux, não existiam interfaces gráficas, sendo necessário utilizar qualquer coisa do sistema por meio do teclado. Com o surgimento dessas interfaces, o uso do teclado para as mais variadas tarefas diminuiu muito devido à facilidade que o *mouse* traz, mas pode ser que em determinado momento não esteja disponível uma interface e tudo precisará ser feito pelo teclado. Nesta seção vamos recuperar esta prática de usar os recursos do teclado para realizar tarefas e ações.

5.1 Linux

No Linux há diversos atalhos para navegar pelo sistema, sendo muitos deles feitos pelo clique do *mouse*. Os mesmos atalhos podem ser usados pelo teclado e até outros atalhos mais complexos que só estão disponíveis pelo uso do teclado. Dentre os mais úteis [12] estão:

1. Gerais
    - `WINDOWS` abre o menu de aplicações.
    - `SHIFT+F10` abre o menu de contexto do item selecionado.
    - `ALT+F2` abre a janela de execução rápida.
    - `CTRL+C` copia o item selecionado.
    - `CTRL+X` recorta o item selecionado.
    - `CTRL+V` cola o item selecionado.
    - `CTRL+Z` desfaz a última ação.
    - `CTRL+Y` refaz a última ação.
2. Janelas e áreas de trabalho
    - `CTRL+ALT+BAIXO` lista todas as aplicações abertas na área de trabalho atual. Quando estiver na lista é possível usar as setas do teclado para navegar entre as aplicações e pressionar `ENTER` para selecionar uma.
    - `CTRL+ALT+CIMA` lista todas as aplicações abertas em todos as áreas de trabalho. Quando estiver na lista é possível usar as setas do teclado para navegar entre áreas e pressionar `ENTER` para selecionar uma. Também há um botão com um `+` no canto da tela, ao clicar nele é criada uma nova área de trabalho.
    - `ALT+TAB` permite navegar entre as aplicações abertas na mesma área de trabalho. Para navegar no sentido contrário o atalho é `SHIFT+ALT+TAB`.

- `ALT+'` navega entre janelas da mesma aplicação em áreas de trabalho diferentes.
- `WINDOWS+D` mostra a área de trabalho. Para voltar à visualização anterior basta usar o atalho novamente.
- `CTRL+ALT+SETA` navega entre áreas de trabalho na direção da seta usada no atalho.
- `SHIFT+CTRL+ALT+SETA` move a janela atual entre as áreas de trabalho, na direção da seta usada no atalho. Para mover por mais áreas basta pressionar a seta mais de uma vez.
- `SHIFT+WINDOWS+SETA` move a janela atual para o monitor posicionado na direção da seta usada no atalho.

3. Janela atual

   - `ALT+F4` fecha a janela atual.
   - `ALT+F5` desmaximiza a janela atual.
   - `ALT+F10` maximiza a janela atual. Para voltar para o tamanho original é só usar o atalho novamente.
   - `WINDOWS+SETA` permite posicionar a janela na direção da seta usada no atalho.
   - `CTRL+WINDOWS+SETA` coloca a janela atual na posição da seta usada no atalho e preenche o espaço restante com outras janelas maximizadas.
   - `ALT+SPACE` abre o menu de janela quando está em uma janela.
   - `CTRL+(+ ou -)` aumenta ou diminui o *zoom* da janela.

4. Sistema

   - `WINDOWS+E` abre o gerenciador de arquivos na pasta *home*.
   - `CTRL+ALT+DELETE` encerra a sessão do sistema.
   - `CTRL+ALT+END` abre o menu de energia (suspender/reiniciar/desligar).
   - `CTRL+ALT+L` bloqueia a tela.

5. Captura e gravação de tela

   - `PRTSC` faz uma captura da tela. Para fazer uma captura e copiá-la para a área de transferência basta usar `CTRL+PRTSC`.
   - `ALT+PRTSC` faz uma captura da janela atual. Para fazer uma captura de uma janela e copiá-la para a área de transferência basta usar `CTRL+ALT+PRTSC`.
   - `SHIFT+CTRL+ALT+R` faz uma gravação da área de trabalho.

6. Ajuda

   - `F1` abre a ajuda online do Linux Mint.

- `CTRL+ALT+ESC` reinicia o Cinammon, mantendo todas as aplicações e janelas abertas.
- `WIN+P` re-detecta os dispositivos de *display*.
- `WIN+L` ativa a ferramenta de depuração.
- `CTRL+ALT+BACKSPACE` reinicia o servidor X, fechando todas as aplicações e janelas abertas.

7. Acessibilidade

    - `ALT+WINDOWS+(+ ou -)` aumenta ou diminui o zoom na área de trabalho.

Apesar de querermos nos livrar do uso do *mouse*, alguns atalhos são complementados por ele. Vejamos alguns exemplos:

- `ALT+F7` prende a janela ao *mouse*, assim, pode ser arrastada para qualquer local. Para fixá-la é só usar o botão esquerdo do *mouse*.
- `ALT+F8` permite que o tamanho da janela seja ajustado se a janela não estiver maximizada. Para ajustar basta mover o *mouse* em qualquer direção.
- `SHIFT+PRTSC` captura somente de uma parte da tela. Um indicador aparecerá, clique na tela e arraste para definir a área que deseja salvar. Se for usado `CTRL+SHIFT+PRTSC` a região selecionada será colocada na área de transferência.

Todos estes atalhos [19] estão estabelecidos nas configurações de sistema, mas podem ser alterados para outras teclas sem nenhum problema. Há também outros atalhos que não são ativados por padrão no sistema e que podem ser configurados para serem utilizados.

Observação: alguns atalhos apresentados aqui usam a tecla `WINDOWS` por ser a tecla presente na maioria dos teclados, porém essa tecla pode ser entendida como `SUPER KEY`.

5.2 *Shell*

No *shell* também há atalhos dedicados a tarefas específicas, muitas são essenciais para trabalhar no terminal sem precisar tocar uma única vez no *mouse*. Dentre os principais [7] estão:

1. Gerais

    - `CTRL+ALT+T` abre uma janela do terminal.
    - `CTRL+SHIFT+T` cria uma nova aba no terminal no mesmo local que o anterior.
    - `CTRL+SHIFT+W` fecha uma aba no terminal.

- `ALT+(0,1,2,...,9)` alterna entre as abas aberta em um mesmo terminal. Esse atalho também funciona em navegadores.
- `Tab` automaticamente completa o arquivo, diretório ou comando que está sendo digitado. Caso exista mais de uma possibilidade, todas as possibilidades são mostradas ao pressionar duas vezes o `Tab`.
- `SHIFT+(PAGE UP ou PAGE DOWN)` navega verticalmente na saída do terminal, para cima ou para baixo, respectivamente.
- `CTRL+SHIFT+(+)` aumenta a fonte do terminal.
- `CTRL+(-)` diminui a fonte do terminal.
- `CTRL+ALT+Fn` (n = 1..6) troca para o enésimo terminal de texto.
- `CTRL+ALT+Fn` (n = 7..12) troca para o enésimo terminal GUI.

2. Processos

   - `CTRL+C` interrompe o processo corrente em primeiro plano. Esse comando envia um sinal SIGINT para o processo, que é basicamente uma solicitação de interrupção.
   - `CTRL+Z` suspende o atual processo corrente em primeiro plano. Esse comando envia um sinal SIGTSTP para o processo. Para retornar o processo ao primeiro plano, deve-se usar `fg nome_processo`.
   - `CTRL+D` fecha o *shell*. Esse comando envia um sinal EOF (*End-of-File*) para o bash. É semelhante a usar o comando `exit` para sair do terminal.

3. Controle de tela

   - `CTRL+L` limpa a tela do terminal. É semelhante a usar o comando `clear`.
   - `CTRL+S` interrompe toda a saída na tela. É útil quando comandos têm uma saída extensa de dados, mas não se quer encerrar o processo.
   - `CTRL+Q` retoma a saída de dados para a tela que o `CTRL+S` interrompeu.

4. Cursor

   - `CTRL+A` ou `Home` vai para o início da linha.
   - `CTRL+E` ou `End` vai para o final da linha.
   - `ALT+B` volta (para a esquerda) uma palavra.
   - `CTRL+B` volta (para a esquerda) um caractere.
   - `ALT+F` avança (para a direita) uma palavra.
   - `CTRL+F` avança (para a direita) um caractere.
   - `CTRL+XX` alterna entre o início da linha e a posição atual do cursor. Usa-se uma vez para ir ao início da linha e outra vez para voltar a posição anterior, sucessivamente.

5. Remoções

    - `CTRL+D` ou `Delete` remove o caractere embaixo do cursor.
    - `ALT+D` remove todos os caracteres na linha corrente após o cursor.
    - `CTRL+H` ou `Backspace` remove o caractere anterior ao cursor.

6. Correções

    - `ALT+T` troca a palavra atual pela anterior.
    - `CTRL+T` troca os dois últimos caracteres anteriores ao cursor entre eles. É muito útil quando é digitado dois caracteres na ordem errada.
    - `CRTL+_` desfaz a última tecla pressionada. Pode ser repetido múltiplas vezes.

7. Recortar e colar

    - `CTRL+W` recorta a palavra anterior ao cursor, adicionando-a à área de transferência.
    - `CTRL+K` recorta a parte da linha posterior ao cursor, adicionando-a à área de transferência.
    - `CTRL+U` recorta a parte da linha anterior ao cursor, adicionando-a à área de transferência.
    - `CRTL+Y` cola a última coisa que foi copiada para a área de transferência.

8. Capitalização de caracteres

    - `ALT+U` capitaliza todos os caracteres a partir do cursor até o fim da palavra atual.
    - `ALT+L` descapitaliza todos os caracteres a partir do cursor até o fim da palavra atual.
    - `ALT+C` capitaliza o caractere embaixo do cursor, após isso o cursor se posiciona no final da palavra atual.

9. Histórico

    - `CTRL+P` ou Seta para cima vai para o comando anterior no histórico. Ao pressionar múltiplas vezes continua voltando no histórico.
    - `CTRL+N` ou Seta para baixo vai para o próximo comando no histórico. Ao pressionar múltiplas vezes continua avançando no histórico.
    - `ALT+R` reverte quaisquer mudanças feitas a um comando recuperado do histórico caso o mesmo tenha sido editado.
    - `CTRL+R` recupera o último comando contendo os caracteres que são digitados. É preciso usar este atalho e digitar o padrão a ser procurado no histórico.

- `CTRL+O` executa um comando que foi encontrado com o `CTRL+R`.
- `CTRL+G` sai do modo de busca no histórico sem executar nenhum comando.

10. Caracteres especiais

    - `Ctrl+I` simula um `Tab`.
    - `Ctrl+J` simula um `Newline`.
    - `Ctrl+M` simula um `Enter`.
    - `Ctrl+[` simula um `Escape`.

Os atalhos acima estão definidos assumindo que está sendo usado o padrão *emacs* de configuração. É possível mudar para uma configuração usando o padrão *vi*:

- `set -o vi` para mudar para o padrão *vi*.
- `set -o emacs` para mudar para o padrão *emacs*.

6 Controle de processos

Um processo é uma instância de um programa que está sendo executado, ou seja, é um programa que está na fase de execução. Quando usamos qualquer comando no *shell* é iniciado um novo processo. O *Kernel* do *Unix* faz a administração e o controle desses processos. [6]

6.1 Processos e *Jobs*

O *Unix* provê os IDs dos processos quando eles são criados. Ao executar um programa em segundo plano, adicionando o caractere `&` ao final do comando, a resposta do *shell* será uma linha como no exemplo:

```
$ programa &[1] 8451
```

No exemplo, 8451 é o ID do processo para o processo "programa". O `[1]` é o número do *job* designado pelo *shell*. A diferença entre os IDs de processos e os *jobs* é que os IDs são referentes a todos os processos executando em todos os usuários e o *job* é reativo aos processos executados pelo seu *shell*. A medida que forem executados novos *jobs*, seu número vai incrementando, como por exemplo `[2]` ou `[3]`.

Assim que o *shell* terminar a execução do *job* será apresentada uma mensagem como:

`[1]+ Done programa` ou `[1]+ Exit 1 programa`

No primeiro exemplo o *job* foi finalizado e no segundo exemplo foi finalizado com um *status* diferente de zero. Outras mensagens podem ser apresentadas, mas estas são as mais comuns.

6.2 Primeiro plano e segundo plano

O comando `fg` traz de volta um processo que foi colocado em segundo plano para o primeiro plano. Tornando-o visível em seu terminal e aceitando entradas do usuário. Esse comando funciona como se o comando tivesse sido executado sem o caractere `&`.

O `fg` traz de volta o processo em segundo plano caso só exista um, ou traz de volta o mais recentemente adicionado ao segundo plano caso existam vários. Para trazer para o primeiro plano um processo específico, basta utilizar o caractere % antes do nome do processo (também podem ser utilizados o número do *job* ou o ID do processo) e utilizar como argumento do `fg`. São possíveis as seguintes formas para referenciar *jobs*:

- `%N` : *job* de número N
- `%string` : *job* que começa com "string"
- `%?string` : *job* que contém "string"
- `%+` : *job* mais recentemente colocado em segundo plano
- `%%` : *job* igual ao de cima
- `%-` : segundo *job* mais recentemente colocado em segundo plano

6.3 Suspensão de *jobs*

Como visto na seção 5, o atalho `CTRL+Z` suspende o atual processo corrente em primeiro plano. Quando isso é feito é apresentada a seguinte mensagem:

```
[1]+ Stopped programa
```

Dessa maneira o processo que estava sendo executado em primeiro plano é suspendido. Para recuperar o programa basta utilizar o comando `fg` como visto anteriormente, resumindo o programa em primeiro plano. Para resumir o *job* em segundo plano basta usar o comando `bg` em vez de `fg`. Taambém é possível suspender o programa com `CTRL+Y`, esse método suspende o programa quando ele tenta ler uma entrada do terminal.

6.4 Sinais

Quando o comando `CTRL+C` é utilizado, o *shell* manda um sinal de `INT` (*interrupt*) para o *job* atual, assim como `CTRL+Z` envia um sinal de `TSTP` (*terminal stop*). O comando `kill` envia um sinal de `TERM` (*terminate*) que tem funcionamento semelhante ao `CTRL+C`. A lista de sinais disponíveis no *shell* pode ser encontrada por meio do comando `kill -l`.

6.5 Comandos associados

A seguir temos alguns comandos que podem ser utilizados para o controle de processos no *shell* [8] [25] :

- `top` : lista os processos em execução e mostra o uso de recursos do sistema, assim como quais processos estão utilizando a maior quantidade desses recursos.
- `htop`: faz o mesmo que o `top`, porém tem uma interface melhorada e possui algumas funções a mais.
- `ps` : lista os processos em execução.
- `pstree`: lista os processos em forma de árvore, para uma melhor vizualização de processos "pais" e processos "filhos".
- `kill` : mata o processo passado como parâmetro.
- `pgrep` : retorna os IDs dos processos que contém o padrão que foi passado como parâmetro para o comando.
- `pkill` ou `killall` : também são usados para matar processos.
- `pidof` : mostra o ID do processo cujo nome foi passado como parâmetro.
- `nohup` : permite que o comando executado junto com ele seja imune a sinais de suspensão ou terminação do processo, sendo assim, o programa continua executando mesmo após o usuário deslogar-se do sistema.

7 Exemplos

Nesta seção serão apresentados alguns exemplos para mostrar como a programação em *shell* é utilizada na prática. Foram aplicados diversos conceitos vistos nas seções anteriores e, eventualmente, algum conceito mais avançado que requer uma explicação maior.

Quando for necessário usar vários comandos para realizar uma tarefa deve ser criado um *script*, que nada mais é do que um arquivo executável contendo o conjunto dos comandos que você irá usar, para que todos os comandos possam ser executados simplesmente ao rodar o *script*. Esses arquivos têm em sua primeira linha o padrão `#!/bin/bash` para que o *shell* interprete que é de fato um *script* e não um arquivo de texto.

7.1 Patrimônio

Esse exemplo tem como base um arquivo (patrimonio.csv) disponibilizado pelo professor. O arquivo contém cinco colunas de dados, separados por identificadores, como no exemplo:

```
"460819";"VENTILADOR DE TETO (5234)";"127V ";" ";"2100.13.02.21";
```

A primeira etapa tem como objetivo buscar no arquivo as linhas que contém o caractere ";"dentro dos campos de dados, pois esse é o separador das colunas e pode causar problemas ao tentar executar comandos no arquivo ou abri-lo em editores de planilhas. Após encontrar essas linhas, o caractere ";"deve ser removido ou substituído por um espaço em branco.

O comando `awk -F'"' -v OFS='"' '{ for (i=2; i<=NF; i+=2) gsub(";"," ",$i) } 1' patrimonio.csv > processed_file.csv` [9] encontra essas linhas e substitui o padrão ";"pelo padrão " ". Explicando o comando:

- `awk`: comando utilizado. O awk é usado para processar arquivos, fluxos de dados ou *pipes* baseados em texto.
- `-F'"'`: a opção -F serve para indicar qual é o separador dos campos, no caso são aspas duplas.
- `-v`: a opção -v indica que uma variável vai ser usada e ela precisa ser definida em seguida (item abaixo).
- `OFS='"'`: essa variável representa o separador de campos na saída do código. Por padrão é um espaço em branco, então foi alterado para aspas duplas.
- `for (i=2; i<=NF; i+=2)`: esse for serve para percorrer os campos da planilha de dados. Começa na coluna 2 e vai até NF (número de colunas naquela linha), incrementando de dois em dois para não pegar o ";".
- `gsub(";"," ",$i)`: essa linha diz que cada ocorrência do caractere ";"será substituída por um espaço em branco.
- `1`: esse número significa que cada linha, modificada ou não, será impressa.
- `patrimonio.csv > processed_file.csv`: correspondem, respectivamente, aos arquivos de entrada e saída do programa.

A segunda parte tem como objetivo obter em um arquivo separado a lista de locais (coluna 5 do arquivo) de forma única e ordenada. O comando `awk -F';' '{print $5}' processed_file.csv | sort -u > locais.txt` faz exatamente isso. Explicando:

- `awk`: comando utilizado. O awk é usado para processar arquivos, fluxos de dados ou *pipes* baseados em texto.
- `-F';'`: a opção -F serve para indicar qual é o separador dos campos, no caso é o ";".
- `'{print $5}'`: esse comando do awk significa que a coluna 5 do aqruivo será impressa.
- `processed_file.csv`: representa o arquivo de entrada.

- `sort -u`: pega a saída do awk e ordena pelo nome, juntado as repetições e deixando cada local de forma única na lista.

- `locais.txt`: arquivo de saída com a lista ordeada.

A terceira parte tem como objetivo gerar um arquivo .csv para cada um dos itens na lista de locais gerada na parte anterior. Exemplo: o arquivo 2100.13.01.02.csv contém todas as linhas do arquivo de entrada que contém a string "2100.13.01.02".

O comando `awk -F';' '{print >> $5".csv"}' processed_file.csv` [10] separa os arquivos pelo local do patrimônio, cada um com todos os itens que pertencem àquele local. Explicando:

- `awk`: comando utilizado. O awk é usado para processar arquivos, fluxos de dados ou *pipes* baseados em texto.

- `-F';'`: a opção -F serve para indicar qual é o separador dos campos, no caso é o ";".

- `'{print >> $5".csv"}'`: esse comando do awk significa que a coluna 5 do aqruivo será impressa com a adição de um ".csv"no final. Está sendo usado o operador >> que cria e concatena as saídas no mesmo arquivo.

- `processed_file.csv`: representa o arquivo de entrada.

7.2 Dados de Matrícula

Esse exemplo tem como base um arquivo (dadosmatricula.tgz) disponibilizado pelo professor. O arquivo contém diversas pastas de disciplinas do departamento de informática em que cada pasta contém arquivos com dados de estudantes desde o primeiro semestre de 1988 até o segundo semestre de 2002.

A ideia era 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. A ideia é ver a evolução do número de matrículas semestre a semestre no período em análise. *Script* desenvolvido:

```
#!/bin/bash
# entra na pasta principal
cd DadosMatricula
# percorre todas as disciplinas
for d in */
do
  # entra nas pastas das disciplinas
  cd $d
  # percorre todos os arquivos
  for f in *.dados
  do
    # pega somente os GRRs e coloca num arquivo fora da
```

```
    # pasta com o nome do semestre
    awk -F':' '{print $2}' $f >> ../$f.temp
  done
  # sai da pasta atual
  cd ..
done
# percorre todos os arquivos criados nos loop anteriores
for f in *.temp
do
  # ordena o arquivo, conta as linhas e soma as mesmas,
  # jogando num arquivo o resultado
  sort -u $f | wc -l | awk '{soma += $1}
  END {print soma-1}' > $f.soma
  # remove o arquivo atual que nao sera mais usado
  rm $f
done
# percorre todos os arquivos com os valores somados
for f in *.soma
do
  # pega o nome do arquivo sem a extensao
  filename=$(basename ''$f'' | cut -d '.' -f 1)
  # pega o conteudo do arquivo
  data=$(cat $f)
  # imprime o semestre e a quantidade de matriculas num
  # arquivo separado
  echo $filename'' : ''$data >> ../matriculas.txt
  # remove o arquivo atual que nao sera mais usado
  rm $f
done
```

Esse *script* gera um arquivo de saída de nome "matriculas.txt" que contém o total de matrículas semestre a semestre. Também seria interessante 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 se repetirão. Um mesmo GRR cursa normalmente várias disciplinas. A saída poderia ser formatada de maneira que cada linha inicie com o ano/semestre (ex. 19881) e em suas colunas o total de matrículas para cada curso. Os cursos que não tem matrículas em uma disciplina não devem aparecer neste relatório. *Script* desenvolvido:

```
#!/bin/bash
# entra na pasta principal
cd DadosMatricula/
# percorre todas as disciplinas
for d in */
do
  # entra nas pastas das disciplinas
  cd $d
```

```
  # percorre todos os arquivos
  for f in *
  do
    # pega somente o curso, ordena e joga num arquivo
    # separado
    awk -F':' '{print $1}' $f | sort -u >> ../c.temp
    # pega o nome do arquivo sem a extensao
    filename=$(basename "$f" | cut -d '.' -f 1)
    # joga o nome do arquivo num arquivo separado
    # para ter a lista de semestres
    echo $filename >> ../s.temp
  done
  # sai da pasta atual
  cd ..
done
# ordena os semestres e cria um novo arquivo
sort -u "s.temp" > semestres.temp
# ordena os cursos e cria um novo arquivo
# retira a ultima linha (cabecalho)
sort -u "c.temp" | head -n -1 > cursos.temp
# imprime um espaco no arquivo da tabela
# (para fins de formatacao)
printf " " > tabela.temp
# percorre todos os cursos
for c in $(cat cursos.temp)
do
  # imprime um separador e o curso
  printf "|$c" >> tabela.temp
done
# imprime uma quebra de linha
printf "\n" >> tabela.temp
# percorre todos os semestres
for s in $(cat semestres.temp)
do
  # imprime o semestre
  printf "$s" >> tabela.temp
  # percorre todos os cursos
  for c in $(cat cursos.temp)
  do
    # a variavel recebe as matriculas no semestre e
    # as conta
    num=$(grep "$c:" */$s.dados | wc -l)
    # se a contagem for igual a zero
    if [ $num -eq 0 ]
    then
      # imprime um separador e um espaco em branco
```

```
      printf ''| '' >> tabela.temp
    else
      # imprime um separador e a contagem
      printf ''|$num'' >> tabela.temp
    fi
  done
  # imprime uma quebra de linha
  printf ''\n'' >> tabela.temp
done
# ajusta o arquivo de acordo com as colunas e o separador
column -s ''|'' -t tabela.temp > ../tabela.txt
# remove todos os arquivos temporarios
rm *.temp
```

Esse *script* gera um arquivo de saída que contém, em formato de tabela [13], o número de matriculas semestre a semestre por curso.

7.3 *Firewall*

Esse exemplo tem como base um arquivo (log-firewall.xz) disponibilizado pelo professor. O arquivo contém um *log* de *firewall* que mostra quais regras de filtragem foram utilizadas. É um arquivo muito extenso (737.313 linhas), então é necessário tomar cuidado com o tempo de execução do *script*.

A ideia seria construir um *script* que apresente como saída o número de bloqueios por filtragem e envie alertas dependendo da quantidade de bloqueios. A saída deve ser dividida em bloqueios que chegam para o protocolo ipv4 e para o protocolo ipv6 e o arquivo deve ter a data do sistema como nome. *Script* desenvolvido:

```
#!/bin/bash

# Diretorios e arquivos que serao usados
log=''/dev/shm/log.tmp''
saidav4=''/dev/shm/ipv4.tmp''
saidav6=''/dev/shm/ipv6.tmp''

# Pega somente a regra de firewall e direciona para um
# arquivo
xzcat log-firewall.xz | awk '{conta[$6]++} END {for
(tipo in conta) print substr(tipo, 1, length(tipo)-1),
conta[tipo]}' > $log

# Inicializa os arquivos de saida
printf ''*** V4 ***\n'' > $saidav4
printf ''\n*** V6 ***\n'' > $saidav6

# Percorre os tipos de bloqueio
```

```
for tipoBloqueio in $(cat tipos-bloqueios)
do
  # Faz a contagem dos tipos de bloqueio
  contagem=$(grep ``^$tipoBloqueio[^-]'' $log)

  # Verifica se a string da contagem esta vazia
  if [ -z ``$contagem'' ]
  then
    contagem=0
  else
    contagem=$(echo $contagem | cut -d`` '' -f2)

  # Se a contagem for maior que o limite (20000)
  if [ $contagem -ge 20000 ]
  then
    # Imprime os dados na saida do for
    printf ``A regra de filtragem
    $tipoBloqueio passou do limite!\n
    Total de ocorrencias: $contagem\n''
    fi
  fi

  # Verifica se e uma regra ipv6, caso nao seja deve
  # atribuir 0 a variavel
  if [ $(echo ``$tipoBloqueio'' | grep -c ``V6'') -eq 0 ]
  # Se for ipv4
  then
      # Imprime o tipo de bloqueio e a contagem no
      # arquivo
      printf ``$tipoBloqueio: $contagem\n'' >> $saidav4
    # Se for ipv6
    else
      printf ``$tipoBloqueio: $contagem\n'' >> $saidav6
    fi

# Envia um e-mail para o administrador com os dados que
# passaram do limite
done | mail -E -s ``Alerta de seguranca de firewall''
``mans17@inf.ufpr.br''

# Pega data e hora atuais
data=$(date +``%d-%m-%Y-%T'')

# Concatena os arquivos de saida
cat $saidav4 $saidav6 > $data.log
```

```
# Remove os arquivos temporarios
rm $saidav4 $saidav6 $log
```

Esse *script* gera um arquivo de saída com a a data e hora do sistema em seu nome e contém uma listagem dos bloqueios e suas devidas quantidades separados entre protocolos ipv4 e ipv6. Também é enviado um e-mail ao administrador cada vez que a quantidade de bloqueios ultrapassa o limite (20.000). Algumas observações:

- Foi escolhido o diretório `/dev/shm/` [5] para armazenar os arquivos temporários, já que este diretório usa memória compartilhada, armazenando os arquivos em uma memória virtual em vez de armazenar no disco (o que causa lentidão).
- A função `substr` retorna a *substring* do parâmetro passado a partir do valor passado até um tamanho x.
- A variável `contagem` recebe um `grep` que seleciona os tipos que começam com aquele padrão e não possuem um traço logo após.
- A opção `-z` dentro do `if` é usada para verificar se uma *string* está vazia.
- A opção `-E` no comando `mail` serve para que o e-mail não seja enviado se o corpo do *e-mail* estiver vazio. Só estaria vazio caso nenhum bloqueio passasse do limite.

8 Exercícios

1. Qual foi o primeiro nome do *UNIX* e porque ele recebeu esse nome?
Resposta: era *UNICS*, devido ao criador Ken Thompson estar reescrevendo o sistema *MULTICS* ao mesmo tempo em que criava o *UNIX*.

2. Algumas versões modernas do Debian vêm com qual *shell* embutido no sistema?
Resposta: Almquist *shell* ou A *shell*.

3. Qual é o *shell* que possui as seguintes variantes: dtksh, tksh, oksh, mksh e sksh?
Resposta: Korn *shell*.

4. Para que serve um coringa?
Resposta: Para representar padrões em nomes de arquivos. Dessa forma não é necessário saber os nomes dos aquivos em questão. Ao usar um coringa, é possível obter todas as combinações possíveis para o nome de um arquivo.

5. Escreva um comando que faça um *backup* de um arquivo para que seu conteúdo não seja perdido ao ser modificado.

Resposta: `cp arquivo arquivo.bkp`

6. Escreva um comando que salve um arquivo contendo a listagem detalhada dos conteúdos da pasta em que está e que sejam somente arquivos .txt.
Resposta: `ls -l | grep ".txt"> ls_arquivos.txt`

7. Qual variável de ambiente contém o tipo de terminal que está sendo utilizado?
Resposta: A variável `TERM`.

8. Qual arquivo deve ser alterado para que seja executado um comando a cada vez que o *shell* for fechado?
Resposta: O arquivo `.bash_logout`.

9. Qual seria uma boa opção para um alias para o comando `grep`?
Resposta: Uma boa opção seria `alias busca=grep`.

10. É possível alterar os padrões de atalhos no Linux?
Resposta: Sim, é possível configurar os atalhos nas configurações do sistema e, no caso do terminal, é possível optar pelo padrão *emacs* ou pelo padrão *vi*.

11. Qual é o atalho para se gravar a tela do computador?
Resposta: O atalho é `SHIFT+CTRL+ALT+R`.

12. Caso queira interromper a saída do terminal devido à um fluxo muito intenso de dados e depois retornar a saída para análise, qual comando deverá ser utilizado?
Resposta: Deverá ser utilizado o comando `CTRL+S` para interromper o fluxo de saída e `CTRL+Q` para retomar o fluxo.

13. Como pode ser colocado uma execução de programa em segundo plano? E como voltar um programa em segundo plano para o primeiro plano?
Resposta: Para colocar um programa em segundo plano basta executar o programa seguido do caractere `&`. Para voltar um programa que esteja em segundo plano para o primeiro plano basta usar o comando `fg`.

14. Qual a diferença entre `CTRL+C`, `CTRL+Z` E `CTRL+Y` em relação à processos?
Resposta: O atalho `CTRL+C` interrompe o processo, enquanto o atalho `CTRL+Z` suspende o processo, podendo ser retomado mais tarde. Já o atalho `CTRL+Y` suspende o processo quando ele tentar ler uma entrada do terminal.

15. É possível manter um *job* em segundo plano mesmo após deslogar-se do sistema?

Resposta: Sim, é possível por meio do comando `nohup`, que permite fazer exatamente isso. Essa ação pode ser muito útil em deixar processos rodando numa máquina mesmo tendo que sair do sistema.

9 Conclusão

Os mais de 40 anos de evolução dos sistemas *Unix* e dos *shells* nos mostra que a ideia que surgiu inicialmente foi acolhida pelos usuários e a partir do momento em que se tornou código aberto sua popularização e seu uso se tornaram muito vastos, permitindo que desenvolvedores de todo o mundo trabalhassem em cima desse sistema e trouxessem diversas versões e atualizações para que cada vez mais avanços fossem feitos nessa área. Hoje temos uma gama imensa de opções de sistemas *Unix* e de *shells* graças a estes desenvolvedores.

Dentre as muitas opções de *shells*, as que recebem mais destaque são o bash pela popularidade e o zsh pelas funcionalidades. Os conceitos básicos do *shell* são suficientes para realizar diversas coisas na linha de comando. A partir dessa fundamentação vêm outras definições que se baseiam nas anteriores e se aprofundam mais no conteúdo. Com o aprendizado será possível otimizar tarefas que antes eram consideradas trabalhosas, mas agora são feitas em uma linha de código, tudo isso em conjunto com a customização do ambiente para que seja obtido um melhor desempenho e conforto duranto e seu uso.

Existem muitos comandos que tornam o uso do Linux e do *shell* muito mais rápidos por meio de seus atalhos no teclado. Não ter que mover alguma das mãos até o *mouse* é um grande ganho de tempo ao mesmo tempo ajuda a entender melhor como funcionam essas ferramentas. O *shell* possui muitos métodos para fazer o controle de processos no *linux* e por meio deles é possível ter um domínio ainda maior do seu ambiente de trabalho.

Referências

[1] Bash Hackers Wiki. Brace expansion. `http://wiki.bash-hackers.org/syntax/expansion/brace`, 2018. Acesso em: 2018-08-10.

[2] G. Berke-Williams. Input/output redirection in the shell. `https://robots.thoughtbot.com/input-output-redirection-in-the-shell`, 2015. Acesso em: 2018-08-11.

[3] Brasil Escola. História do linux. `https://brasilescola.uol.com.br/informatica/historia-do-linux.htm`, 2016. Acesso em: 2018-08-04.

[4] Gilles. Difference between .bashrc and .bash_profile. `https://superuser.com/questions/183870/difference-between-bashrc-and-bash-profile/183980`, 2010. Acesso em: 2018-11-10.

[5] V. Gite. What is /dev/shm and its practical usage. https://www.cyberciti.biz/tips/what-is-devshm-and-its-practical-usage.html, 2006. Acesso em: 2018-09-21.

[6] V. Gupta. Manage processes in unix. http://www.unixinterview.com/process-handling.html, 2018. Acesso em: 2018-11-10.

[7] L. Heddings. The best keyboard shortcuts for bash. https://www.howtogeek.com/howto/ubuntu/keyboard-shortcuts-for-bash-command-shell-for-ubuntu-debian-suse-redhat-linux-etc/, 2017. Acesso em: 2018-10-13.

[8] C. Hoffman. How to manage processes from the linux terminal: 10 commands you need to know. https://www.howtogeek.com/107217/how-to-manage-processes-from-the-linux-terminal-10-commands-you-need-to-know/, 2012. Acesso em: 2018-11-10.

[9] ITSwapShop. How to remove commas inside of quotes from a csv or text file on linux with awk. http://www.itswapshop.com/tutorial/how-remove-commas-inside-quotes-csv-or-text-file-linux-awk, 2015. Acesso em: 2018-08-24.

[10] Linux Questions. Bash or php: Split csv file based on field value? https://www.linuxquestions.org/questions/programming-9/bash-or-php-split-csv-file-based-on-field-value-702639/, 2015. Acesso em: 2018-08-24.

[11] L. O. Miranda. Variáveis de ambiente no linux. https://www.todoespacoonline.com/w/2015/07/variaveis-de-ambiente-no-linux/, 2015. Acesso em: 2018-11-10.

[12] G. Newell. Complete list of linux mint 18 keyboard shortcuts for cinnamon. https://www.lifewire.com/complete-list-of-linux-mint-4064592, 2018. Acesso em: 2018-10-13.

[13] G. Newell. Display file contents in column format within linux. https://www.lifewire.com/file-contents-in-column-format-linux-4018107/, 2018. Acesso em: 2018-09-07.

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

[15] Ninja Do Linux. A história do unix (sistema operacional). http://ninjadolinux.com.br/a-historia-do-unix/, 2016. Acesso em: 2018-08-03.

[16] Rad. Quest to learn the unix shell (bash). http://mt4.radified.com/2009/06/learn-unix-shell-bash-linux-ubuntu-command-terminal.html, 2009. Acesso em: 2018-08-17.

[17] Research Hubs. Linux directory structure. http://researchhubs.com/post/computing/linux-cmd/linux-directory.html, 2015. Acesso em: 2018-08-18.

[18] S. Robinson. Zsh vs bash. https://stackabuse.com/zsh-vs-bash/, 2016. Acesso em: 2018-08-18.

[19] rsking84. 82 shortcuts for linux mint (linux). https://shortcutworld.com/Linux-Mint/linux/Linux-Mint_Shortcuts, 2015. Acesso em: 2018-10-13.

[20] J. Sanders. Basic shell commands. https://www-xray.ast.cam.ac.uk/~jss/lecture/computing/notes/out/commands_basic/, 2011. Acesso em: 2018-08-09.

[21] Software Carpentry Foundation. The unix shell. https://swcarpentry.github.io/shell-novice/, 2018. Acesso em: 2018-08-17.

[22] Techopedia. What is unix? - definition from techopedia. https://www.techopedia.com/definition/4637/unix, 2018. Acesso em: 2018-08-01.

[23] TutorialsPoint. Unix - what is shells? http://www.tutorialspoint.com/unix/unix-shell.htm, 2018. Acesso em: 2018-08-01.

[24] Unixmen. Evolution of unix / linux shells - unixmen. https://www.unixmen.com/evolution-unix-linux-shells/, 2015. Acesso em: 2018-08-02.

[25] WikiBooks. Guide to unix/commands/process management. https://en.wikibooks.org/wiki/Guide_to_Unix/Commands/Process_Management, 2018. Acesso em: 2018-11-10.