Relatórios CI320

Luan Varaschim Welter

Novembro 2018

Sumário

1 UNIX **4**
- 1.1 Resumo . . . 4
- 1.2 Introdução . . . 4
- 1.3 Desenvolvimento . . . 4
  - 1.3.1 Árvore de diretórios . . . 5
- 1.4 Exercícios . . . 5
- 1.5 Conclusão . . . 6

2 Shell **6**
- 2.1 Resumo . . . 6
- 2.2 Introdução . . . 6
- 2.3 Desenvolvimento . . . 6
  - 2.3.1 Comandos . . . 7
  - 2.3.2 Redirecionamentos de entrada e saída . . . 8
  - 2.3.3 Expansões e coringas . . . 9
- 2.4 Exercícios . . . 9
- 2.5 Conclusão . . . 10

3 Customizando o Ambiente Bash **10**
- 3.1 Resumo . . . 10
- 3.2 Introdução . . . 10
- 3.3 Desenvolvimento . . . 10
  - 3.3.1 Aliases . . . 10
  - 3.3.2 Variáveis . . . 11
- 3.4 Exercícios . . . 12
- 3.5 Conclusão . . . 12

4 Introdução ao i3 Window Manager **12**
- 4.1 Resumo . . . 12
- 4.2 Introdução . . . 12
  - 4.2.1 Instalação . . . 12
  - 4.2.2 Configuração Inicial . . . 13
- 4.3 Desenvolvimento . . . 13
  - 4.3.1 Primeiras Impressões . . . 13
  - 4.3.2 Arquivos de Configuração . . . 13
- 4.4 Conclusão . . . 14

5 Exercícios de aula **14**
- 5.1 Problema dos patrimônios . . . 14
  - 5.1.1 Descrição . . . 14
  - 5.1.2 Solução . . . 14
  - 5.1.3 Otimizações . . . 16
- 5.2 Organizar Matriculas . . . 17
  - 5.2.1 Solução . . . 17
- 5.3 Análise de log do Firewall . . . 21
  - 5.3.1 Solução . . . 22
  - 5.3.2 Otimizações . . . 23

1 UNIX

1.1 Resumo

O sistema UNIX desde seu começo facilitou o trabalho de programadores, e, ao longo dos anos, novas versões para seu sistema e ferramentas foram surgindo, oferecendo novas funcionalidades e facilidades.

1.2 Introdução

Antes do surgimento do sistema operacional Linux, UNIX foi criado a partir de um projeto posteriormente abandonado pelos seus desenvolvedores, o MULTICS [4]. No início, computadores eram tão grandes que ocupavam salas [5], e possuíam seus próprios sistemas operacionais. Além de caros, programadores, antes do UNIX, deveriam lidar com cartões representando comandos computacionais e com a lentidão nos testes deles [18].

Para resolver esses problemas e criar um sistema operacional simples e portável, UNIX começou a ser desenvolvido em 1970 e logo se popularizou, dando origem a várias versões, como SunOS, Xenix e BSD. Ele inspirou outros sistemas que se utilizaram das mesmas ideias, como Linux e Minix, que visavam modularidade e reusabilidade, possibilitando a criação de novos programas a partir de antigos.

1.3 Desenvolvimento

Em 1965, as empresas *Bell Laboratories*, *General Eletric* e o Centro de Pesquisa de Ciência da Computação da AT&T se juntaram a um projeto do Instituto Tecnológico de Massachusetts (MIT) chamado de **MULTICS**. Seu nome se traduzia como Sistema Computacional e Informacional Multiplexado, e tinha como objetivo principal ser um único e confiável sistema computacional, além de garantir níveis de segurança para seus usuários [4].

MULTICS também foi desenvolvido para ser usado por vários usuários ao mesmo tempo, ajudá-las a se comunicar e armazenar e recuperar grandes quantidades de dados. Conseguiram avançar em todos esses objetivos, porém enfrentaram dificuldades em alguns aspectos do projeto que tornavam-o complexo desnecessariamente, causando a desistência da empresa *Bell Laboratories* depois de quatro anos [18].

Com a retirada da *Bell Laboratories*, três de seus desenvolvedores decidiram continuar com algumas das ideias do projeto *MULTICS*, e remover grande parte do que causou a sua interrupção, a complexidade. **Ken Thompson**, **Dennis Ritchie** e **Joseph Ossanna** criaram, então, o **UNIX**, um sistema operacional novo simples, elegante, escrito em uma linguagem de alto nível e de código reciclável [19].

UNIX inicialmente era uma mistura de código aberto e fechado, ou seja, possuía algumas partes de seu código que podiam ser alteradas por seus usuários, caso quisessem. Por causa disso, várias versões e mutações do UNIX surgiram desde que foi desenvolvido. Algumas substancialmente diferentes de sua versão original, como **BSD** (*Berkeley Software Distribution*) [2]. **BSD**, em específico, inicialmente tinha uma versão misturada entre código aberto e fechado [23]. Apesar disso, a maioria de seus descendentes tiveram seus códigos abertos.

BSD foi desenvolvido pela Universidade de Berkeley ao adicionarem novos códigos ao UNIX, incluindo uma nova versão da ferramenta Shell, o **sh**. Dele surgiu o sistema SunOS que, por fim, evoluiu para o sistema Solaris [26]. Esses dois novos sistemas tinham seus códigos fechados, ou seja, impossibilitava a alteração por parte da comunidade. A Solaris era uma versão do SunOS e foi comprado pela multinacional americana Oracle, procurando aumentar a escalabilidade do sistema.

Por fim, uma outra versão do UNIX, chamada Xenix, foi desenvolvida inicialmente pela Microsoft e teve seu código inteiro fechado para contribuições externas [27]. Era vendido para fabricantes como Intel e IBM, ao contrário de ser comercializado para usuários comuns, e, atualmente, está descontinuado.

O sistema UNIX cresceu em popularidade com o tempo desde seu desenvolvimento, em parte devido à larga distribuição de amostras gratuitas do sistema para universidades nos anos 70. Com a vantagem de ser maleável e barato, o sistema foi bem aceito e serviu de inspiração para novas versões e clones. Dele, surgiu um sistema com ideias técnicas bem similares ao UNIX, mas que respeita a liberdade dos usuários por ser composto totalmente de software livre, o **GNU** (ou, em tradução à mão livre, "GNU não é UNIX") [25].

Inspirando-se também no UNIX, surgiu o Linux, criado por um estudante universitário chamado **Linus Torvalds** ao frustrar-se com falta de funcionalidades em um dos clones do sistema, o **Minix**. Linux foi desenvolvido com o objetivo de ser utilizado em computadores domésticos munidos da arquitetura Intel. Com o tempo, ele foi portado para outras plataformas, e também é utilizado em servidores por ser minimalista.

1.3.1 Árvore de diretórios

O sistema UNIX introduziu uma ideologia que determina que tudo em seu sistema será tratado como um arquivo. Dessa forma, simples arquivos de texto, músicas, configurações de sistema, usuários logados e informações de hardware são reconhecidos como arquivos pelo UNIX [1] [18].

Para organizar todos os arquivos do sistema, o UNIX se utiliza de diretórios padrões, onde determinados arquivos normalmente são encontrados. A árvore começa da raiz (/) que contém os diretórios [3]:

- **bin**, para binários de sistema e funções carregadas na Shell.
- **boot**, possui informações necessárias na inicialização do sistema.
- **dev**, onde se encontra arquivos referentes a hardware.
- **etc**, com configurações que afetam todos os usuários.
- **home**, onde é guardado o diretório *"home"* de cada usuário.
- **lib**, possui bibliotecas compartilhadas e módulos de *kernel*.
- **proc**, possui informações sobre cada processo sendo executados na máquina.
- **root**, o diretório *home* do usuário root.
- **sbin**, contendo binários com permissão de execução apenas ao usuário root.
- **tmp**, diretório para arquivos temporários.

Outros diretórios também são encontrados porém estes são os mais importantes.

1.4 Exercícios

1. Leia as paginas de manual para Bash e UNIX com os comandos:

```
$ man bash
$ man unix
```

1.5 Conclusão

O sistema operacional UNIX tem contribuído - e muito - com a computação, desde do seu desenvolvimento. Diversos benefícios podem ser extraídos de suas funcionalidades e conceitos explorados com seu desenvolvimento [13]. Com o código fonte disponível para ser compartilhado e alterado gratuitamente, o UNIX se tornou uma escolha popular.

Apesar do aparente sucesso do sistema, é reconhecido que UNIX possui erros e pode causar frustração em seus usuários. Desta forma, surgiu o sistema Linux; o que deixa a dúvida de quantos outros mais sistemas - clones ou versões modificadas do UNIX - foram desenvolvidos com o mesmo propósito. Tal resistência também é facilmente vista em materiais como o livro [4], que descreve o UNIX como um vírus de uma maneira bastante bem-humorada (ou não).

2 Shell

2.1 Resumo

A ferramenta Shell sempre foi ligada ao UNIX, por de maneira simples, ser uma interface com o sistema. Através de comandos ela é capaz de utilizar a estrutura UNIX de arquivos e atingir resultados complexos por meios compreensíveis.

2.2 Introdução

Os desenvolvedores do sistema UNIX precisavam de uma maneira simples de se comunicar com a máquina, de uma forma de interação com o sistema. Pensando nisso, criaram uma das características mais importantes do UNIX, a ferramenta Shell, uma interface de comando que auxilia os usuários a acessar funções fundamentais do sistema operacional [20]. Sua primeira versão desenvolvida foi a **sh** (Bourne Shell)[22], nomeada assim em homenagem a Stephen R. Bourne, seu criador.

No mais, a ferramenta Shell pode interpretar comandos submetidos pelo usuários, desde comandos simples para saber o diretório atual como **pwd**, até comandos inteligentes e complexos para cópia e sincronização de vários arquivos como **rsync**. Para facilitar a comunicação entre os programas com eles mesmos ou com o programador, também existem funcionalidades como caracteres "coringa", expansão de colchetes/chaves e redirecionamento de entrada/saída.

2.3 Desenvolvimento

O sistema UNIX possui uma interface do usuário, chamada de **Shell**. Essa ferramenta interpreta as linhas de comando do usuário - ou de um arquivo - e as transforma em instruções do sistema operacional. Ao ler um arquivo com comandos, também chamados de *shell script*, a ferramenta **Shell** lê linha por linha, assimilando comandos no sistema, similar a um compilador convertendo um programa para uma forma que a máquina possa ler, ou seja, em um arquivo executável [6].

A ferramenta **Shell**, assim como o UNIX, inspirou diversas outras versões e clones, sendo **Bash** o tipo mais comum dessas interfaces, também conhecidas como "baseadas em caracteres" [16]. A imagem abaixo, na **Figura 1**, mostra a evolução dessas versões da interface Shell com o passar dos anos.

Como visto na **Figura 1**, o primeiro da ferramenta **Shell** se chamava **sh**, criado em 1977. Ele foi o shell original usado no UNIX, e era um programa pequeno e básico. Do **sh** surgiram vários descendentes, como a **csh** (C Shell), a **ksh** (KornShell) e a **Bash** (Bourne Again Shell).

Resumidamente, a **csh** foi desenvolvida em 1979 para o sistema BSD, deixando a **sh** original do sistema mais interativa e sua programação mais parecida com código C [24]. A **ksh** foi desenvolvida em 1983 para

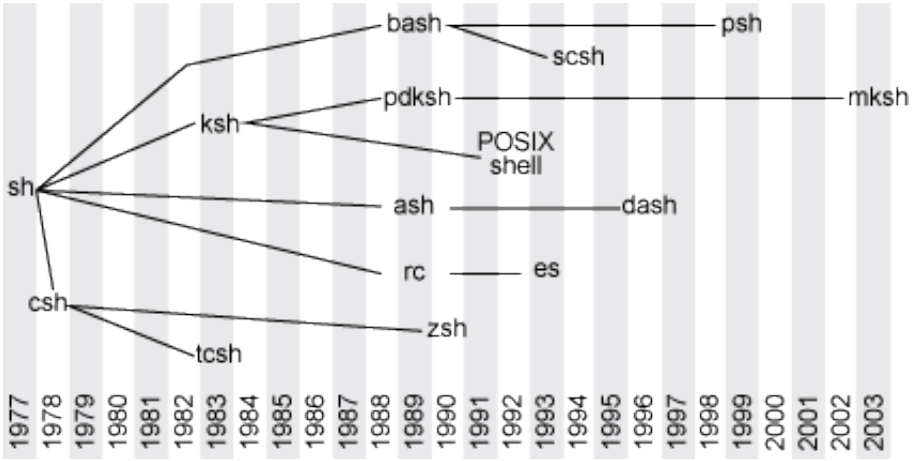


Figura 1: evolução da Shell
[12]

a interface POSIX, que adicionou controle de *jobs*, *aliases* e o histórico de comandos utilizados. Por fim, a **Bash** foi uma ferramenta desenvolvida em 1989 para substituir o **sh** no sistema Linux e se tornou a versão mais utilizada da **Shell**. Ela se inspirou pelas duas anteriores, trazendo funcionalidades como implementação de histórico, *aliases*, controle de *jobs*, edição de comando, autocompletar, entre outros [21].

A partir da **csh** (C Shell), surgiu a **tcsh** (TENEX C Shell), em 1983, adicionando a opção de autocompletar comandos, nomes de arquivos e afins emprestada do sistema Tenex [28]. Outro descendente da **csh** foi desenvolvido em 1990, a **zsh** (Z Shell), que, além das vantagens implementadas na **Bash**, possui suporte a contas em ponto flutuante - ao contrário da manipulação apenas de ponteiros sem ajuda de programas exteriores [28]. Ainda sobre a **zsh**, ela possui um autocompletar mais leve e rápido que a **Bash**, suporte a estruturas de dados indexadas por *hash*, funções para *parse* de parâmetros complexos, e outros.

2.3.1 Comandos

Para interagir com o sistema, muitos comandos podem ser usados e interpretados pela ferramenta **Shell**. Um comando é basicamente uma sequência de palavras, onde a primeira é o nome do programa a ser executado e é seguida de parâmetros para sua execução, alguns exemplos sendo [16]:

- **man**: o comando de manual, útil para entender funcionamento e função de diversos recursos UNIX, referência para todos os comandos listados nesta lista.
- **cat**: comando para imprimir na tela o conteúdo de arquivos, caso não encontre o nome de arquivo como parâmetro, ele utiliza a entrada padrão.
- **ls**: lista os conteúdos de diretórios.
- **echo**: exibe na tela texto passado como parâmetro.
- **cd**: altera o diretório atual, navegando sobre a árvore de diretórios.
- **cp**: copia arquivos e diretórios.
- **mv**: move arquivos e diretórios.

- **awk**: comando para processamento de texto.
- **tr**: comando para "tradução" de caracteres.
- **mkdir**: cria novos diretórios.
- **touch**: altera último acesso a um arquivo, também pode ser usado para criar um novo.
- **rm**: remove arquivos e diretórios.
- **find**: procura por arquivos a partir de um diretório.
- **ssh**: cliente *ssh* para conexão com servidores remotos.
- **finger**: utilizado para verificar informações sobre usuários.
- **grep**: imprime na tela linhas que possuem padrão especificado.
- **vim/nano**: editores de texto sem necessidade de terminal gráfico.

Para verificar os todos os diretórios presentes na raiz do sistema, por exemplo, é possível executar o comando:

```
$ ls /
```

O comando *ls* lista os arquivos dentro do diretório "/" por este ser seu parâmetro.

2.3.2 Redirecionamentos de entrada e saída

Na **Bash**, a saída de um programa pode ser a entrada do próximo, e isso pode ser atingido de alguns jeitos. O primeiro através de pipelines "|", por exemplo:

```
$ ls / | wc
```

Assim a saída do comando *ls* é transferida para o programa *wc* que conta palavras, linhas e caracteres recebidos.

Um outro jeito é pela substituição em linha, como:

```
$ echo ''$(ls)''
```

Onde o comando *ls* é executado e sua saída é passada como parâmetro para outro programa, o *echo*.

Os símbolos "textless" e "textgreater" também são usados para direcionar entradas e saídas:

```
$ echo ''alo mamae'' > arquivo.txt
$ cat < arquivo.txt
```

O primeiro comando irá colocar a frase "alo mamae" para dentro de *arquivo.txt*, criando ele caso não exista o arquivo ou reescrevendo caso contrário. Para apenas anexar, sem destruir o conteúdo já presente, pode-se usar "textgreater>". Por fim, o segundo comando **cat** lê o conteúdo do arquivo que vem pela entrada padrão e imprime na tela.

2.3.3 Expansões e coringas

Para facilitar a programação, a **Shell** entende caracteres coringa como o "?" e o "*", onde "?" combina com um caractere qualquer e "*" combina com qualquer número de caracteres quaisquer. Ela também expande os caracteres presentes dentro de colchetes[9] tentando encontrar algum arquivo que corresponda a expressão regular passada, expande caminhos[7] relativos como o .. para o diretório pai e ∼ para a pasta *home* e chaves[8] gerando todas as combinações possíveis referentes as palavras passadas. Exemplo:

```
$ echo a*
$ echo a?
$ echo a[a-z]
$ echo ~
```

A expansão tenta combinar com todos arquivos presentes no diretório atual. Assim, o programa **echo** irá imprimir na tela todos os arquivos começados em "a" no primeiro exemplo, todos com duas letras começados em "a" no segundo e todos com duas letras começados em "a" onde a segunda letra é parte do alfabeto e minúscula. Porém, caso não seja encontrado nenhum arquivo que respeite a expansão, ela própria é retornada. Dessa forma, a saída do primeiro exemplo, caso não exista nenhum arquivo começado em "a", é "a*". Para o último exemplo ele é expandido para a *home* do usuário, e assim sua saída é "/home/$USR" ou "/root" caso o usuário esteja logado com o mesmo.

Existe outra expansão, que utiliza chaves, porém ela não tenta encontrar entre os arquivos mas expande para todas as possibilidades, assim:

```
$ echo a{a,b,c{d,e}}
aa ab acd ace
```

2.4 Exercícios

1. Leia as páginas de manual para os comandos listados na seção 2.3.1 começando por *man man*:

2. Crie um diretório *dir*, altere o diretório corrente para o diretório *dir* e dentro dele um arquivo *arq*.

   Resposta:

   ```
   $ mkdir dir
   $ cd dir
   $ touch arq
   ```

3. Coloque o nome de todos os arquivos em sua home em um arquivo chamado *home_files.txt*, renomeie o arquivo para *files.txt*,imprima na tela o conteúdo do arquivo e também imprima traduzindo todos os caracteres minúsculos para maiúsculos. Termine removendo o arquivo.

   Resposta:

   ```
   $ ls ~ > home_files.txt
   $ mv home_files.txt files.txt
   $ cat files.txt
   $ tr [a-z] [A-Z] < files.txt
   $ rm files.txt
   ```

2.5 Conclusão

Uma das maiores contribuições do sistema UNIX foi a ferramenta Shell, cuja evolução de versões está ligada às versões do sistema. A cada alteração, busca-se oferecer mais facilidade de programação a cada iteração, como interagir com o sistema e geração de scripts que automatizem uma série de comandos. Foram também desenvolvidas várias versões da ferramenta, como **Bash**, **csh** e **ksh**, e nenhuma está necessariamente acima da outra, principalmente por apresentarem funcionalidades diferentes.

Apesar das facilidades desenvolvidas para a ferramenta Shell, ela é comprovadamente difícil de dominar. Desta forma, grande parte de seus usuários conhecem apenas um fragmento de todas as possibilidades da Shell.

Apesar dos problemas e resistências, tanto o UNIX quanto o Shell foram de suma importância para o desenvolvimento da tecnologia, oferecendo conceitos hoje utilizados em sistemas conhecidos, como **Mac OS X** e **Microsoft Windows**.

3 Customizando o Ambiente Bash

3.1 Resumo

A Bash possui várias ferramentas para acelerar o trabalho do programador, e neste capítulo serão apresentadas algumas maneiras de personalizar e customizar o ambiente de trabalho. Com tais opções é possível simplificar e facilitar a interação com a ferramenta, e também o sistema.

3.2 Introdução

Quando um terminal Bash é iniciado, mesmo antes de entregar a entrada para o usuário, uma série de comandos podem ser executados. Basta existir um arquivo de configuração padrão no diretório *home* do usuário. Este pode ser .bash_profile, .bash_login e .profile que serão procurados nesta ordem e o primeiro a ser encontrado é executado. Existe também o arquivo .bashrc que é executado sempre que uma nova instância Bash é criada a partir de uma antiga. Muitas vezes o arquivo .bashrc é executado através dos outros, assim todas as instâncias possuem .bashrc carregado. Este arquivo pode conter a inicialização de várias variáveis para a Bash, alias, funções e também pode executar comandos, facilitando o trabalho do programador.

3.3 Desenvolvimento

Assim como o arquivo .bashrc configura a Bash, outros programas possuem arquivos para configurações. Para o editor de texto **vim** existe o .vimrc, para o programa **git** o arquivo .gitconfig, e assim para vários programas é possível modificar seu comportamento padrão alterando arquivos de configuração. Dentro do arquivo de configuração para a Bash, por exemplo, é possível definir aliases, funções e variáveis a fim de diminuir o quanto deve ser escrito ou definir nomes mais óbvios para comandos com pouco significado.

3.3.1 Aliases

Alias é a maneira mais simples de renomear comandos. Quando uma linha de comando é lida no *prompt* da Bash, é verificado se a primeira palavra corresponde a um alias definido. Caso isso seja verdade, é feita a substituição do texto para a execução do comando.

Um alias pode ser definido através do comando:

```
alias nome=texto
```

Se o texto a ser substituído possui espaços, deve ser utilizado aspas para proteger o texto, porém o nome da substituição não pode conter espaços, além de não poder existir nenhum espaço entre o nome, o "=" e a substituição. Para desfazer um alias é possível utilizar o comando **unalias** e, como parâmetro, o nome a ser removido.

Para listar os aliases atualmente iniciados basta digitar o comando **alias**, ou **alias -p**, e ele retornará a tela os nomes e valores correspondentes.

Caso ocorra uma substituição e o texto termine em espaço é verificado se a segunda palavra do comando também faz parte do dicionário de aliases. O alias também acontece recursivamente, então o mesmo comando pode ser modificado por múltiplos alias, entretando não pelo mesmo duas vezes. Dessa forma, construções como **alias ls='ls -a'** não se tornam recursões infinitas.

3.3.2 Variáveis

Outro modo de substituição de texto na Bash é se utilizando de variáveis. Existem diversas delas que afetam diretamente o comportamento da Bash[14], mas também é possível criar variáveis que fazem sentido apenas ao usuário. Similarmente á criação de aliases, o comando para a criação de variáveis é:

```
nome=texto
```

Novamente, para utilizar texto com espaços é necessário proteger com aspas e não devem existir espaços entre as palavras e o espaço. Para expandir o texto sendo guardado na variável é utilizado o símbolo "$" seguido do nome da variável. Diferente de aliases que são válidas apenas no começo de um comando, as variáveis podem ser utilizadas em qualquer parte dele.

Algumas variáveis que podem modificar o comportamento da Bash por exemplo são:

- **HISTFILE**: nome do arquivo de histórico de comandos utilizados.
- **IGNOREEOF**: se definido, indica o número de vezes seguidas que deve ser lido o carácter EOF antes de sair da Bash.
- **LC_ALL**: variável para identificar a linguagem sendo utilizada pelo sistema.
- **PATH**: caminho para diretórios com programas a serem executados pela Bash.

Outras variáveis apenas guardam informações que podem ser relevantes, como:

- **BASH_ARGC**: lista com número de parâmetros existentes em cada nodo da pilha de chamadas.
- **BASH_ARGV**: lista de parâmetros existentes na chamada atual.
- **BASH_VERSION**: número da versão atual da Bash sendo rodada.
- **RANDOM**: retorna um valor aleatório entre 0 e 32767 a cada chamada.

Na Bash é possível customizar a linha de *prompt* modificando a variável **PS1** para o mais comum. Quando o comando é escrito em mais de uma linha, **PS2** é utilizado. Estas variáveis podem ser ainda mais customizadas com caracteres escapados, alguns exemplos:

- **\u**: expande para o nome do usuário atual.
- **\w**: expande para o diretório corrente, equivalente a **pwd**.
- **\n**: inclui quebra de linha no *prompt*.

3.4 Exercícios

1. Leia sobre os arquivos de configuração do seu editor de texto favorito.
2. Altere o arquivo de configuração ".bashrc" com um alias para ignorar quando o comando **sudo** for digitado

   Resposta:

   alias sudo=""
3. Atualize a variável de *prompt* para antes do já existente aparecer tempo no formato HH:MM:SS e o número do comando atual

   Resposta:

   $PS1=''t \# *$PS1''*

3.5 Conclusão

É sabido que a Bash oferece diversas ferramentas para seus usuários, de simples a complexas funções. Há quem as procure apenas por curiosidade, assim como programadores iniciantes e experientes. Com o sucesso de sistemas como Linux, observou-se uma preocupação com a acessibilidade de seus usuários com o sistema. Bash, ao permitir customização por parte de quem utiliza, oferece uma oportunidade te tornar o ambiente shell personalizado e facilitado para o uso do usuário.

Seja criando aliases ou modificando o comportamento padrão em arquivos de configuração, dá a usuários de diversos níveis de conhecimento a liberdade de uso, estudo, cópia, modificação e redistribuição de software. Dessa forma, através de estimular a criatividade, abre-se a possibilidade de surgimento de ideias e alternativas que enriquecerão a comunidade.

4 Introdução ao i3 Window Manager

4.1 Resumo

Existem diversos ambientes gráficos disponíveis para o sistema Linux como GNOME, KDE, Cinnamon, entre outras.

O foco deste trabalho será no gerenciador de janelas *i3*, um gerenciador direcionado a usuários experientes. Sua instalação já vem com alguns atalhos de teclado configurados e é bem simples adicionar novos, assim diminuindo muito a necessidade de utilizar o mouse.

4.2 Introdução

4.2.1 Instalação

Inicialmente devemos instalar o gerenciador de janelas **i3**:

```
sudo apt update
sudo apt install i3
```

4.2.2 Configuração Inicial

Com o programa instalado reinicie a sessão e na tela de login altere o manager para o i3.

Em seu primeiro login o arquivo de configuração ainda não foi criado, assim o i3 ira criar um prompt na tela pedindo se deseja criar um automaticamente e qual sera a tecla "Modifier" sendo suas opções *Mod1* (tecla alt) e *Mod4* (tecla "Windows") para evitar conflitos de atalhos é recomendado o uso do *Mod4*.

4.3 Desenvolvimento

4.3.1 Primeiras Impressões

Com a escolha de modificador feita na tela é visível uma tela preta e uma barra em baixo com alguns detalhes da máquina. Algo que irá notar rapidamente é a falta de um "menu iniciar" como em outros gerenciadores.

A barra em baixo tem o nome de *i3status*. Ao lado esquerdo esquerdo existe um numero indicando o desktop sendo exibido e seguindo para a direita é visível o status de conexão IPv6, espaço livre em disco, status de conexão wireless, status de conexão ethernet, status da bateria, uso de CPU e hora local.

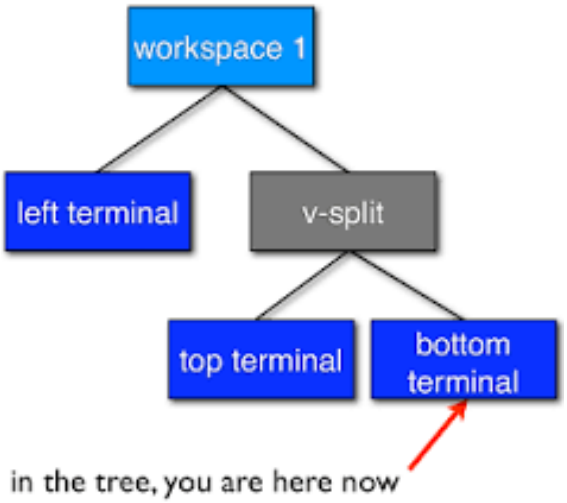
Um dos atalhos mais importantes no momento é como abrir um shell, para isso use as teclas $mod+enter sendo $mod o modificador escolhido na configuração inicial.

4.3.2 Arquivos de Configuração

Os arquivos de configuração do i3 suportam uma grande quantidade de opções, para simplicidade irei explicar apenas atalhos no arquivo padrão e para mais informações é possível usar o site do i3 com o **guia de usuário**.

Dois arquivos são importantes no momento, o primeiro sendo para a barra *i3status* encontrado provavelmente em "/etc/i3status.conf", outros locais válidos são encontrados na pagina de manual do i3status. Neste arquivo existe uma sequência de funções que definem como a barra deve ser escrita.

O outro arquivo pode ser encontrado em "~/.config/i3/config" e este se refere a atalhos de teclado e alguns detalhes dos contêineres das janelas.



O i3 trabalha com as janelas em sistema de *tiling*, assim ele ocupa toda a tela com quadrados. Os quadrados são estruturados em forma de árvore como visto nas imagens acima. para navegar entre as janelas são usadas as setas juntamente com a tecla $mod, também é possível mover uma janela pressionando juntamente a tecla shift.

Para gerar os nodos de split vertical e horizontal é usado $mod+[v,h]. Na figura o layout mostra todas as janelas.

- $mod+e é equivalente a figura
- $mod+f a tela em foco toma a tela cheia
- $mod+w as janelas filhas do split mais recente tem sua área unida e seus nomes separados em abas
- $mod+s equivalente ao "w" porém com os nomes separados em pilha

Além de separar janelas por splits existem desktops virtuais acessíveis pelas teclas $mod+número do desktop e pressionando juntamente shift causa a janela em foco ser movida para o desktop selecionado.

Outra funcionalidade é redimensionamento, que no arquivo de configuração aparece como um *modo*, quando é pressionado $mod+r o i3 passa a ignorar o modo "default" anterior e assume este outro onde pressionar as setas altera o tamanho da janela em foco e pressionando esc ou enter retorna ao modo normal.

4.4 Conclusão

O gerenciador de janelas possui algumas funcionalidades para o mouse porém seu foco é evitar o uso dele, transformando a navegação entre janelas bem simples. Apesar de o descrito no texto ser o suficiente para evitar o uso do mouse é apenas uma parte do oferecido pelo *i3* e para um melhor aproveitamento do sistema é recomendado estudar o seu guia [11].

5 Exercícios de aula

5.1 Problema dos patrimônios

5.1.1 Descrição

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").

5.1.2 Solução

Para resolver o problema eu utilizei o código abaixo.

```
#!/bin/bash
LIST=$(awk -F'"' -v OFS='"' '{\
  for (i=1; i<=NF; i++){\
    if( $i != ";"){\
      gsub( ";", "", $i)\
    }\
  }\
}1' patrimonio.csv | cut -d\; -f5 | tr -d \" | sort -u)
mkdir csv
for N in $LIST; do
  fgrep "\"$N\"" patrimonio.csv > csv/$N.csv
done
```

Para resolver a parte 1 [5.1.1] do exercício é necessário corrigir onde existem ";" perdidos dentro de aspas duplas então:

- **awk** para manipular o texto do arquivo
- **-F'"'** identifica o separador de campos como aspas duplas
- **-v** para valorar variáveis internas ao programa **awk**
- **OFS='"'** (output field separator) variável do awk com texto a ser inserido entre os campos na saída
- **for (i=1; i<=NF; i++)** NF (Number of Fields) é o numero de campos encontrados pelo **awk** e faz este laço verificar todos os campos encontrados
- **if($i != ";")** pula campos com apenas ";" assim selecionando os campos onde é necessário eliminar o ";" caso exista algum
- **gsub(";", "", $i)** comando do awk para substituição de padrões, aqui utilizado para trocar um ";" por nada
- **}1'** uma segunda clausula para o comado **awk**, que é sempre verdadeira e resulta no programa executando a operação padrão sobre todas as linhas que não passarem pela primeira clausula, que é ser impressa na saída
- **patrimonio.csv** e o ultimo parâmetro para indicar a entrada do comando

Neste ponto a parte 1 [5.1.1] foi resolvida, a saída do comando awk será o texto do arquivo sem ";" perdidos pelo meio.

Para a parte 2 [5.1.1] é necessário extrair a lista de números na quinta coluna, isso pode ser resolvido utilizando o comando:

- **cut** comando para cortar o texto em colunas
- **-d\;** parâmetro para identificar ";" como delimitador de campo
- **-f5** parâmetro selecionando a quinta coluna para ser impressa

A saída neste ponto possui a lista de todos os números no formato "numero", porém as aspas ainda devem ser removidas

- **tr** comando de tradução/troca de caracteres
- **-d** parâmetro para a deleção de caracteres
- \" passa as aspas duplas para serem deletadas

Neste ponto todos os números fazem parte da saída, para finalizar a parte 2 [5.1.1] é necessário ordenar os números e retirar repetições

- **sort** comando para ordenar a saída
- **-u** e parâmetro para retirar repetições e deixar a saída única

Este código é executado através da construção $() e inserido dentro da variável **LIST**.
Com a lista pronta falta resolver a parte 3 5.1.1, separar o arquivo **patrimonio.csv** em vários menores
Para manter a organização foi criado um diretório novo com **mkdir csv**.
A iteração sobre a lista de números foi feita usando o laço **for N in $LIST; do** e a separação com o comando

```
fgrep "\"$N\"" patrimonio.csv > csv/$N.csv
```

- **fgrep** comando para combinar padrões, equivalente a **grep -F**, onde o padrão procurado é fixo e não possui expressões regulares
- **"\"$N\""** o padrão a ser encontrado é o numero cercado por aspas duplas
- **patrimonio.csv** utilizando o arquivo original como entrada
- **> csv/$N.csv** a saída do fgrep será todas as linhas com o padrão então basta redirecionar para o arquivo de nome "[número].csv"

5.1.3 Otimizações

Após possuir uma versão funcional do código tentei otimizá-lo, algumas tentativas foram:

1. Como no código abaixo, não ler o arquivo a cada iteração do laço "for", porém como variáveis em Bash funcionam, ocorre a expansão do texto e o processamento para passar como parâmetro toda vez que a variável é referenciada. assim diminuindo a eficiência do código [10].

```
VAR=$(< patrimonio.csv)
...
echo "$VAR" | fgrep "\"$N\"" > csv/$N.csv
```

2. Outra possibilidade seria utilizar o parâmetro **–mmap** para a função **grep**, que de acordo com o manual pode ter um desempenho melhor dependendo do caso, porém ela foi descontinuada e não é mais reconhecida [15].
3. Inicialmente era utilizada a função **grep**, porém por ser um padrão fixo a ser procurado é possível utilizar a função **fgrep** já que possui um desempenho melhor.

4. Naturalmente o *Bash* assume arquivos em formatação *utf-8* [17] e não como *ascii*, caso o arquivo patrimonio.csv pudesse ser codificado em *ascii* isso aceleraria o processo pois caracteres em *ascii* ocupam um byte comparado com *utf-8* que ocupa 4 bytes por caractere, para indicar o *Bash* a usar *ascii* poderia ser adicionado **LC_ALL=C** antes do comando **fgrep**, como indicado abaixo

```
LC_ALL=C fgrep "\"$N\"" patrimonio.csv > csv/$N.csv
```

5.2 Organizar Matriculas

O objetivo deste trabalho é contabilizar o número de matrículas em cada disciplina do Departamento de Informática entre os anos 1988 (primeiro semestre) até 2002 (segundo semestre).

Os dados de entrada estão em um arquivo TAR-GZ em marcos/tmp/dadosmatricula.tgz e devem ser copiados para sua área $SEUHOME/nobackup, pois não queremos lotar o sistema de backups. Os scripts podem (devem) ficar na sua área com backup.

Quando expandir o arquivo TAR-GZ você terá uma árvore de diretórios com nome: "DadosMatricula". Dentro deste diretório, existe 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 carácter separador de colunas é o ":"(dois pontos).

Parte 1 Você deve 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. Um exemplo pode ser visto aqui. 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.

Parte 2 Como segundo exercício, o objetivo é 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 curso. Os cursos que não tem matrículas em uma disciplina não devem aparecer neste relatório.

5.2.1 Solução

Para resolver o problema foi utilizado o código:

```
#!/bin/bash

if [ ! -d files ]; then
  mkdir files
elif [ ! -z "$(ls files)" ]; then
  rm files/*
fi

FILES="/home/bcc/lvw15/nobackup/DadosMatricula"
for MAT in $(find $FILES -mindepth 2 ); do
  #todos os grrs para ex1
  tail -n +2 $MAT | cut -d: -f2 >> files/${MAT##/*/}.grr
  #todos os cursos para ex2
  tail -n +2 $MAT | cut -d: -f1 >> files/${MAT##/*/}.curso
done

for SEMESTRE in $(find files/ -mindepth 1 -iname "*.grr"| sort); do
  echo "${SEMESTRE:6:5} : $(sort -u $SEMESTRE | wc -l)" >> saida1
done

TABELA="ano,$(cat files/*.curso | sort -u | tr '\n' ',')\n"
HEADER="$TABELA"
TF=$(echo $TABELA | awk -F ',' '{print NF}')
for SEMESTRE in $(find files/ -mindepth 1 -iname "*.curso"| sort); do
  TABELA="$TABELA${SEMESTRE:6:5},"
  FIELD="2"
  for CURSO in $(sort -u $SEMESTRE) ; do
    NFIELD=$(echo $HEADER | awk -F ',' -v curso="$CURSO" '{
              for(i=1;i<=NF;i++)if($i==curso){print i; exit}}')
    for i in $(seq $FIELD $(($NFIELD - 1)));do
      TABELA="${TABELA}0,"
    done
    TABELA="$TABELA$(grep $CURSO $SEMESTRE |wc -l),"
    FIELD=$(($NFIELD + 1))
  done
  for i in $(seq $FIELD $(($TF - 1)));do
    TABELA="${TABELA}0,"
  done

  TABELA="$TABELA\n"
done
echo -e "$TABELA" > saida2.csv
echo -e "$TABELA" | sed 's/,$//' | column -t -s,
```

O código acima pode ser dividido em etapas. Na primeira etapa é feita uma preparação do ambiente, onde é verificado todos os arquivos de dados a partir do diretório extraído. Cada arquivo é separado entre o GRR e o curso do aluno e redirecionado para um arquivo contendo todas as matérias de um certo ano.

```
if [ ! -d files ]; then
  mkdir files
elif [ ! -z "$(ls files)" ]; then
  rm files/*
fi

FILES="/home/bcc/lvw15/nobackup/DadosMatricula"
for MAT in $(find $FILES -mindepth 2 ); do
  #todos os grrs para ex1
  tail -n +2 $MAT | cut -d: -f2 >> files/${MAT##/*/}.grr
  #todos os cursos para ex2
  tail -n +2 $MAT | cut -d: -f1 >> files/${MAT##/*/}.curso
done
```

Nesta etapa é criado um novo diretório para manter a organização, deletado seu conteúdo caso ele já exista, e é definida uma variável contendo o caminho raiz dos dados extraídos. Em seguida, através de um loop for, é removida a primeira linha contendo o cabeçalho, selecionada a coluna dos dados necessários e impressa em um arquivo temporário

- **find** : comando para identificar arquivos presentes na arvore de diretórios.
- **-mindepth 2** : Parâmetro para o comando **find** ignorar respostas de arquivos ou diretórios com menos de 2 níveis na arvore. Com **-mindepth 1** apenas o diretório inicial "/home/bcc/lvw15/nobackup/DadosMatricula" seria ignorado, porém dentro dele ainda há outro nível de diretórios com os nomes das matérias ofertadas. Com **-mindepth 2**: estes também não são retornados, deixando apenas os arquivos com estilo [ano][semestre].dados.
- **MAT** : A variável ira iterar sobre a resposta do comando **find**, exemplo: "/home/bcc/lvw15/nobackup/DadosMatricula/CI061/19921.dados"
- **tail** : Comando para imprimir as ultimas linhas de um arquivo.
- **-n** : Indica número de linhas a ser impresso de baixo para cima no arquivo, caso seja usado o sinal "+" é interpretado como de cima para baixo. O **+2** é usado para pular a primeira linha contendo o cabeçalho
- **cut -d: -f[1/2]** : Corta o arquivo em colunas separadas por ":" e imprime o primeiro/segundo campo encontrado
- **${MAT##/*/}** : Utilizando a construção ${var} ao invés de $var, além de proteger o nome da variável permite editá-la. Neste caso a construção ## é usada para deletar a maior correspondência ao padrão *regex*, sendo "/*/". Como no exemplo da variável MAT anterior "/home/bcc/lvw15/nobackup/DadosMatricula/CI061/19921.dados" vira "19921.dados"
- **»files/${MAT##/*/}.[grr/curso]** : Como cada ano/semestre está espalhado por vários arquivos, referentes a matéria, eles são condensados para apenas 2, um contendo os GRRs encontrados e outro com os códigos dos cursos. O nome deste arquivo será como no item acima adicionado ".grr" ou ".curso" ao final.

Ao final desta etapa o diretório "files" possui os arquivos para cada ano com suas informações condensadas. Agora podemos facilmente responder a primeira parte do exercício5.2.

```
for SEMESTRE in $(find files/ -mindepth 1 -iname "*.grr"| sort); do
  echo "${SEMESTRE:6:5} : $(sort -u $SEMESTRE | wc -l)" >> saida1
done
```

Para cada semestre é contado o número de GRRs únicos e gerada a saída do exercício 1

- **find** : Neste caso é usado **-mindepth 1** pois apenas o diretório "files" precisa ser ignorado. O parâmetro **-iname "*.grr"** retorna apenas os arquivos de GRR já que ambos de GRR e curso se encontram neste diretório.
- **SEMESTRE** : A variável ira conter arquivos com o nome "files/[YYYY][S].dados.grr"
- **echo** : Cada arquivo ira gerar uma das linhas da saída, na forma de "semestre : alunos únicos".
- **${SEMESTRE:6:5}** : Para recuperar o semestre é possível usando a construção ${var} cortar uma parte da variável. Aqui usado para pegar cinco caracteres a partir do sexto, estes sendo os quatro números do ano e o número do semestre

Com todas as linhas impressas o arquivo "saida1" possui a resposta do exercício 15.2. Para o segundo exercício é necessário gerar uma tabela ano/curso, para isso foi utilizado o código a seguir:

```
TABELA="ano,$(cat files/*.curso | sort -u | tr '\n' ',')\n"
HEADER="$TABELA"
TF=$(echo $TABELA | awk -F ',' '{print NF}')
for SEMESTRE in $(find files/ -mindepth 1 -iname "*.curso"| sort); do
  TABELA="$TABELA${SEMESTRE:6:5},"
  FIELD="2"
  for CURSO in $(sort -u $SEMESTRE) ; do
    NFIELD=$(echo $HEADER | awk -F ',' -v curso="$CURSO" '{
              for(i=1;i<=NF;i++)if($i==curso){print i; exit}}')
    for i in $(seq $FIELD $(($NFIELD - 1)));do
      TABELA="${TABELA}0,"
    done
    TABELA="$TABELA$(grep $CURSO $SEMESTRE |wc -l),"
    FIELD=$(($NFIELD + 1))
  done
  for i in $(seq $FIELD $(($TF - 1)));do
    TABELA="${TABELA}0,"
  done

  TABELA="$TABELA\n"
done
echo -e "$TABELA" > saida2.csv
echo -e "$TABELA" | sed 's/,$//' | column -t -s,
```

Inicialmente é gerado o cabeçalho da matriz com todos os cursos possíveis. A seguir Para cada arquivo de semestre é verificado o número de ocorrências de cada curso. Para gerar a matriz ainda é necessário acertar em qual coluna cada contagem deve ir, para isso há um loop interno que para cada curso presente no ano acerta a posição de impressão. Com a tabela feita e formatada como um *csv* ela é impressa no arquivo saida2 e também na tela formatada por colunas.

- **TABELA** : Variável responsável por guardar todos os dados da tabela. Ela é iniciada com todos os cursos possíveis (**cat files/*.curso**), ordenados (**sort -u**) e separados por virgulas (**tr '\n' ','**). Uma copia do cabeçalho é salva em **HEADER** para facilitar uma procura mais a frente.
- **TF** : O número total de colunas da tabela é salvo caso seja necessário preenchimento. **awk -F ',' 'print NF'** separa o texto por virgulas e imprime o total de campos.
- **find** : Semelhante a segunda etapa do código onde foi resolvido o exercício 1 porém retornando os arquivos de curso para a variável **SEMESTRE**. Para cada iteração será gerada uma nova linha na tabela final.
- **$TABELA${SEMESTRE:6:5},** : A tabela recebe o semestre sendo processado
- **FIELD** : Variável para manter o índice da coluna sendo atualizada na tabela
- **sort -u $SEMESTRE** : É verificado cada curso em um semestre.
- **NFIELD** : A posição de cada curso é procurada (**awk**) através do cabeçalho (**HEADER**), onde cada campo é verificado até a posição correta e esta é salva na variável
- **${TABELA}0,** : Para cada campo de entre o atual (**FIELD**) e o encontrado (**NFIELD**) a tabela é preenchida com zeros.
- **grep $CURSO $SEMESTRE |wc -l** : A tabela recebe o total de matriculas do curso no semestre sendo processados.
- **$(($NFIELD + 1))** : O campo atual é atualizado para o primeiro vazio e volta para o laço dos cursos
- **seq $FIELD $(($TF - 1)))** : Antes de seguir para a próxima linha da tabela, a atual é preenchida com zeros e recebe um '\n'
- **saida2.csv** : A tabela completa é impressa em um arquivo csv e tambem impressa na tela formatada como tabela

5.3 Análise de log do Firewall

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 arquivo marcos/tmp/tipos-bloqueios contém um exemplo do tipo de regras que podem ser de interesse do administrador. Observe que no campo 6 do log do firewall encontramos o nome da regra que foi usada para a filtragem. O arquivo exemplo deve ser gerado automaticamente a partir dos logs, pois o administrador inclui novas regras quando acha necessário. Como só temos um arquivo de exemplo, a lista de regras deve ser retirada deste único arquivo.

O administrador está também interessado em separar os bloqueios que chegam no protocolo IPv4 do IPv6, para cada tipo de regra. Um exemplo de saída pode ser encontrado em marcos/tmp/exemplo-saida.

Notem que como o log é diário, 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.

Objetivo do trabalho: a partir do log do firewall e das regras apresentadas como exemplo gerar um arquivo de saída similar ao exemplo dado acima. Considere, para fins de exercício, que o número mágico

20.000 (vinte mil) é o limite aceito pelo administrador como de filtragens normais. Além de gravar em disco a saída diária, você deve enviar email para você mesmo 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.

5.3.1 Solução

Para resolver o problema foram utilizados dois scripts, um em awk e um em bash

O código em bash apenas resolve a questão de mandar um email avisando que existe uma filtragem fora do normal

```
#!/bin/bash
export LC_ALL=C

./script.awk log-firewall ||
    mail -s "Suspect activity found" lvw15@inf.ufpr.br <\
        logs/$(date +%Y-%m-%d)
```

- **./script.awk log-firewall** : executa o script em awk sobre o arquivo log-firewall
- **||** : verifica se ouve uma saída de erro (aqui indicando uma filtragem fora do normal)
- **mail** : comando para enviar emails por linha de comando
- **-s** : parâmetro para o subject, aqui sendo "Suspect activity found"
- **lvw15@inf.ufpr.br** : destino do email
- **< logs/$(date +%Y-%m-%d)** : corpo do email é o arquivo gerado pelo script awk

Para contar os bloqueios foi utilizado o código a seguir

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

{
  bloqueios[$6]++
}

END {
  ret=0
  date=strftime("%Y-%m-%d")
  delete v6[0] ; delete v4[0] ;

  while ((getline line < "tipos-bloqueios") > 0 ){
    line = line":"
    if (bloqueios[line] == "") bloqueios[line] = 0
    else if (bloqueios[line] > 20000) ret = 1
    if (line ~ /V6*/)
        v6[length(v6)+1] = line " " bloqueios[line]
    else v4[length(v4)+1] = line " " bloqueios[line]
```

```
  }

  print "\n*** V4 ***" > "logs/"date
  for(i in v4) print v4[i] >> "logs/"date
  print "\n*** V6 ***" >> "logs/"date
  for(i in v6) print v6[i]>> "logs/"date
  exit(ret)
}
```

- **#!/usr/bin/gawk -f** : Por algumas funções não existentes em awk foi utilizado gawk (GNU awk)
- **bloqueios[$6]++** : a cada linha era contado o bloqueio encontrado, porém existe um ":" no fim da string que deve ser lembrado mais para frente
- **ret=0** : variável para retorno do código
- **date=strftime("%Y-%m-%d")** : variável para salvar a data no formato ano-mês-dia, uma das funções presentes apenas em gawk
- **(getline line < "tipos-bloqueios")** : lê uma linha do arquivo "tipos-bloqueios"
- **while** : Para cada linha do arquivo lido
- **line = line":"** : Conserta o dois pontos ignorado anteriormente
- **if (bloqueios[line] == "") bloqueios[line] = 0** : garante a impressão do 0 caso não tenha ocorrido nenhum bloqueio por um certo motivo
- **else if (bloqueios[line] > 20000) ret = 1** : Atualiza o retorno caso seja encontrado algum valor acima do número magico
- **if (line /V6*/)** : Separa linhas de ipv4 e ipv6
- **length(v6)** : Outra função não existente em awk, retorna o tamanho do vetor
- **v6[length(v6)+1] = line " " bloqueios[line]** : Adiciona a linha a ser impressa no vetor, analogamente para o vetor **v4**
- **print** : finalmente imprime tudo formatado como o arquivo de exemplo de saída

5.3.2 Otimizações

1. Qualquer ação junto da atualização do vetor de bloqueios seria rodada várias vezes, assim tanto a filtragem de quais bloqueios deveriam ser impressos e a correção do ":" foi adiada para a parte END do código.
2. Como o arquivo log-firewall pode ser ser codificado em ASCII foi utilizado LC_ALL=C

Referências

[1] David Both. everything-is-a-file. https://opensource.com/life/15/9/everything-is-a-file.

[2] Francisco Burzi. The linux knowledge base and tutorial. https://sourceforge.net/projects/linkbat/files/Linkbat%20PDF/Tutorial%200.3/Tutorial_0.3.pdf/.

[3] Ubuntu Docomentation. Linuxfilesystemtreeoverview. https://help.ubuntu.com/community/LinuxFilesystemTreeOverview.

[4] Simson Garfinkel, Daniel Weise, and Steven Strassmann. *The UNIX-HATERS Handbook*, pages 3–15. IDG Books Worldwide, Inc., San Mateo, 1994.

[5] Machtelt Garrels. *A Hands on Guide*, pages 7–8. CreateSpace Independent Publishing Platform, United States, 2008.

[6] Machtelt Garrels. *Bash Guide for Beginners*, pages 6–20. Fultus Corporation, 2010.

[7] GNU. caminhos. https://www.gnu.org/software/bash/manual/html_node/Tilde-Expansion.html.

[8] GNU. chaves. https://www.gnu.org/software/bash/manual/html_node/Brace-Expansion.html.

[9] GNU. colchetes. https://www.gnu.org/software/bash/manual/html_node/Filename-Expansion.html.

[10] Heath. Why is reading a file faster than reading a variable? https://superuser.com/a/279628.

[11] i3. guia de usuário i3. https://i3wm.org/docs/userguide.html.

[12] M. Jones. historia sh. https://www.ibm.com/developerworks/br/library/l-linux-shells/index.html.

[13] Ladislav Lhotka. History of unix. *XXX. konference EurOpen.CZ*, 1(1):23–35, 2007.

[14] GNU Manual. Bash variables. https://www.gnu.org/software/bash/manual/html_node/Bash-Variables.html.

[15] Linux Manual. grep(1) - linux man page. https://linux.die.net/man/1/grep.

[16] Cameron Newham. *UNIX Shell Programming*, pages 14–50. O'Reilly Media, 2009.

[17] Jacob Nicholson. Speed up grep searches with lc_all=c. https://www.inmotionhosting.com/support/website/ssh/speed-up-grep-searches-with-lc-all.

[18] Steve Parker. *Expert Recipes for Linux, Bash, and more*, pages 3–13. John Wiley & Sons, Inc., Indianapolis, 2011.

[19] Charles Severance. *A Brief History of Unix*. Equipe abnTeX2, 2008.

[20] Shantanu Tushar and Sarath Lakshman. *Linux Shell Scripting Cookbook*, pages 3–8. Packt Publishing, Indianapolis, 2013.

[21] wikipedia. Bourne again shell. https://en.wikipedia.org/wiki/Bash_(Unix_shell).

[22] wikipedia. Bourne shell. https://en.wikipedia.org/wiki/Bourne_shell.

[23] wikipedia. Bsd. https://en.wikipedia.org/wiki/Berkeley_Software_Distribution.

[24] wikipedia. C shell. https://en.wikipedia.org/wiki/C_shell.

[25] wikipedia. Linux. https://en.wikipedia.org/wiki/Linux.

[26] wikipedia. solaris. https://en.wikipedia.org/wiki/Solaris_(operating_system).

[27] wikipedia. Xenix. https://en.wikipedia.org/wiki/Xenix.

[28] wikipedia. Z shell. https://en.wikipedia.org/wiki/Z_shell.