Shells Unix

Relatório 3

Dalmon Ian

2018

Esse é um relatório feito como atividade da matéria Tópicos em Programação de Computadores (CI320) ministrada na Universidade Federal do Paraná (UFPR), com foco em programação em shell. Esse relatório cobre também aspectos relacionados como a história dos terminais, do Unix e das diversas implementações das shells Unix

1 Introdução

A *shell* é um programa interpretador de comandos, a interface entre o sistema operacional e o mundo exterior[18]. As principais funções da *shell* são: executar comandos, e.g um programa externo, e; facilitar a interação entre usuário e máquina, para isso conta com características como uso interativo, e a possibilidade de escrever programas utilizando a linguagem da *shell*, ambos conceitos de vital importância para o sistema operacional inspirados pelo Unix[19].

Esse documento é divido oito sessões, que tratam sobre:

1. Esta introdução;
2. História dos terminais;
3. História do Unix e de suas *shells*;
4. Demonstra características básicas do uso interativo da *shell*;
5. Scripts na linguagem da *shell*.
6. Diretórios;
7. Arquivos, *pipes*;
8. Processos;
9. Casos de uso avançados, e;

10. Apresentação das práticas feitas em laboratório.

Esse relatório assume que o leitor possui alguma familiaridade com sistemas baseados no Unix e conhecimentos básicos com programação estruturada em linguagens como C ou Pascal.

2 Terminais no Unix

2.1 Pré-história: Teletipo

O teletipo é uma máquina de escrever eletromecânica capaz de receber e enviar mensagens por meio de canais de comunicação em diversas configurações (ponto-a-ponto ou multi-ponto). A sigla TTY utilizada para descrever os terminais no Unix vem da palavra em inglês *TeleTYpewriter*.

2.2 Terminais de video: VT100

Os terminais de video são máquinas dedicadas que se conectam ao computador por meio de uma interface serial (geralmente UART[27]), compostas apenas por um monitor e teclado. Essas máquinas possuem pouca capacidade computacional e não têm armazenamento local.

Lançado pela *Digital Equipment Corporation* (DEC) em 1978, o terminal VT100 foi o primeiro terminal que seguia os conformes do padrão ANSI, possuindo características como formato de tela com 80 colunas.

2.3 Terminais gráficos

O desenvolvimento das tecnologias permitiu que terminais pudessem exibir texto e imagens. São exemplos de terminais gráficos no Unix o blit [16] e o AT&T 5620 [21].

Com os terminais gráficos surge a necessidade de um sistema de janelas, que permite que vários programas sejam executados simultaneamente, o que explora as capacidades multitarefa do Unix mais facilmente do que nos terminais de texto.

Surgem aqui também os emuladores de terminal, um *software* que replica a interface dos terminais de texto.

3 Histórico do Unix e suas shells

3.1 Pré-história: Multics

Multics (*Multiplexed Information and Computing Service*)[7] é um sistema operacional dos anos 60, um projeto colaborativo liderado pelo MIT juntamente com a *General Electric* e a *Bell Labs*, sendo um projeto de grande importância dentro da ciência da computação, onde muitos conceitos da área de sistemas operacionais sendo criados e implementados pela primeira vez aqui, inclusive a *shell* [17].

Aqui a *shell* se mostra de forma bem primitiva: Os programas chamados não geram novos processos, ao invés disso, os programas são carregados para dentro da *shell* e executados como chamadas de procedimento em tempo de execução (ligação dinâmica)[18].

3.2 Bell Labs: Unix, Thompson shell, Bourne shell e Korn shell

Após a *Bell Labs* deixar o projeto Multics em 1969 [26], o legado e experiência obtidos por aqueles envolvidos nesse projeto levaram a criação do Unix[19], e de sua primeira versão até a sexta, a *shell* utilizada foi criada por Kenneth "Ken"Thompson [14], que introduziu conceitos como estruturas de entrada e saída (*piping*, redirecionamento), estruturas de controle (if, goto), etc.

Apartir da sétima versão do Unix, a versão da *shell* que o sistema apresenta é a *Bourne shell* [4], e é daqui onde as *shells* modernas nascem. Criada por Stephen Bourne, difere-se da implementação anterior por ser uma linguagem de programação de alto nível que lembra outras como FORTRAN, Pascal ou ALGOL, além de proporcionar um melhor uso interativo do sistema.

Seguindo a linhagem da *Bourne shell*, David Korn cria a *Korn Shell* sendo um superconjunto das características das *shells* Unix do seu tempo, além de outras características avançadas, como por exemplo co-processos[11], uma forma de executar comandos em *subshells* de forma assíncrona, possibilitando uma melhor exploração do paralelismo nos scripts, além de extensões como a *dtksh*, que possui primitivas para criar elementos gráficos no nível da *shell*[15].

Porém o Unix e todas as *shells* mencionadas nessa sessão eram de propriedade da AT&T, mas outros fabricantes do Unix também possuíam suas implementações da *shell*.

3.3 BSD: C shell

Em 1977 a Universidade da Califórnia em Berkeley desenvolveu o BSD (*Berkeley Software Distribution*), um sistema baseado no Unix e que durante um tempo chegou a compartilhar código com a versão da AT&T, o que gerou um processo jurídico[13].

Criada por Bill Joy, a *C shell* difere-se das *shells* implementadas pelo time da *Bell Labs* por sua sintaxe ser inspirada pela linguagem C, além de possuir características mais amigáveis para o uso interativo, como a navegação pelo histórico de comandos executados, possibilidade de atribuir atalhos para comandos e arquivos, incluindo o uso do til (~) como um atalho para a *home* do usuário, auto-completar nomes de comandos e arquivos etc.

Porém, sua adoção fora do universo do BSD não foi muito grande, e houve uma grande crítica em relação ao seu uso como linguagem de programação [6]. Atualmente, O FreeBSD é um sistema popular que usa uma implementação da *C shell* como a *shell* nativa, mas outros sistemas BSD como o OpenBSD utilizam uma implementação da *Korn Shell*.

3.4 Linux: Bash e Zsh

Em 1991 surge um sistema operacional chamado Linux, criado por Linus Torvalds, que tornou-se uma das implementações mais populares de sistemas baseados no Unix atualmente[8]. Juntamente com o kernel Linux, as distribuições do sistema possuem as ferramentas criadas pela *Free Software Foundation* (FSF) e o projeto GNU (*Gnu is Not Unix*) para o espaço de usuário, incluindo a Bash (*Bourne again shell*) como *shell* nativa.

Criada por Brian Fox em 1989, Bash [9] aparece como uma alternativa de software livre à *Korn Shell* (que na época ainda era de propriedade da AT&T), além de possuir características da *C shell*, porém mantendo a familiaridade da sintaxe já ubíqua da *Bourne shell*, acrescentando uma grande variedade de características tanto para uso interativo e para programação, o que contribuiu para sua popularidade. Atualmente, essa é a implementação *de facto* quando refere-se à *shell* Unix, porém há outras implementações que vêm ganhado popularidade junto a comunidade.

Uma dessas implementações é a Zsh (*Z shell*) criada por Paul Falstad [23], cuja uma das principais diferenças à Bash é o fato de que a Zsh tenta emular a ksh, enquanto a Bash não[22]. Porém há outras características da Zsh, principalmente aquelas relevantes ao uso interativo, que tornam essa *shell* atrativa, como por exemplo o autocompletar[23] e as expansões da *shell*, além de sua grande capacidade de personalização e toda a comunidade que gira em torno disso, como por exemplo *"Oh My ZSH!"*[2].

3.5 Execícios

3.5.1 Ler o manual de alguma shell citada no texto. RECOMENDAÇÃO: sh

3.5.2 Imprimir o texto "hello, world"

Solução:

```
# Versao portatil
echo "hello␣word"
printf "hello␣word\n"

# ksh, Zsh
print "hello␣word"
```

3.6 Implementar uma shell utilizando a liguagem da shell. DICA: é possível realizar a tarefa utilizando apenas três comandos das shells compatíveis com a Bourne shell

Solução:

```
while read cmd; do eval "$cmd"; done
```

4 Uso interativo

4.1 Executando comandos

Ao iniciar uma sessão de uso da *shell* o usuário é apresentado com o interpretador de linha de comando, onde são realizadas as operações de entrada e saída da *shell*.

A entrada da *shell* são os comandos, linhas de texto que são que interpretados como requisições para executar outros programas[19]. A linha de comando é lida pelo interpretador e é divida em *tokens* [10], por exemplo, a linha de comando `echo "hello␣word"` é dividida nos *tokens* **echo** e **hello world**, onde o token **echo** é o nome do comando a ser executado e **hello world** é o argumento (também chamado de parâmetro) passado para esse comando. Argumentos são os elementos que o comando utiliza como operandos ou operadores.

Existem outras formas de comandos, como comandos compostos, definição de funções, etc[10], porém a explicação desses comandos saem do escopo dessa sessão, e serão devidamente explicados em um futuro próximo (seção Z). Sendo assim, define-se comando como um conjunto de palavras de separadas por espaço e terminado com o caractere `\n`.

4.2 Navegação pelos comandos

4.2.1 Histórico de comandos

Cada comando que é passado para a *shell* é escrito no histórico, geralmente o arquivo indicado pela variável **HISTFILE**, possibilitando a listagem e manipulação de comandos que a shell já executou. O código a seguir demonstra essas características:

```
$ history
120    man mksh
121    man sh
122    htop
123    man sh
124    sudo emerge -p man-pages-posix
125    apropos sh
126    apropos -s 1 sh
127    cls
128    apropos -s 1 sh
129    apropos -s 1p sh
130    man sh
131    cls
132    man sh
133    man gc
$ fc -l  # Alternativamente
    ...
$ !!     # Reexecuta o ultimo comando (Bash)
$ !n     # Reexecuta o n-ésimo comando (Bash)
$ !-n    # Reexecuta o n-ésimo ultimo comando (Bash)
$ !htop  # Reexecuta o comando que começa com "htop"(Bash)
$ # ou apenas "r"na korn shell e Bash (idem ao !!)
$ fc -e -
$ fc -s n    # r n (idem ao !n)
$ fc -s -n   # r -n (idem ao !-n)
$ fc -s htop # r htop (idem ao !htop)
$
$ # substitui no comando 133 gc por gcc e o Reexecuta
$ fc -s gc=gcc 133
$ r gc=gcc 133        # idem
$ # Substitui old por new no ultimo comando o reexecuta (Bash)
$ ^old^new
$
$ # edita o comando 124 em um editor de texto o reexecuta
$ fc 124 # ou
$ r 124
```

4.2.2 Edição da linha de comando

Uma lista de comandos para a edição de comandos[5] pelo terminal[1] está na tabela 1.

[1] A tabela refere-se ao modo de edição convencional da *shell*

Tecla	Comando
Backspace ou ^- H	Apaga letra a esquerda
Delete	Apaga letra a direita
^- W	Apaga última palavra
^- U	Apaga caracteres do cursor até o início da linha
Home ou ^- A	Apaga última palavra
End ou ^- E	Apaga última palavra
←	Move cursor uma posição à esquerda
→	Move cursor uma posição à direita
↑	Move para o comando anterior
↓	Move para o próximo comando

Tabela 1: Comandos de edição

Outro comando importante ao utilizar o modo interativo da shell é a tecla Tab, que auto-completa nome de comandos e arquivos:

```
$ mk # Ao teclar Tab duas vezes a shell tenta auto-completar o comando
/usr/lib/plan9/bin/mk            /bin/mknod
/usr/bin/                        /bin/mktemp
/usr/bin/mkfifo                  /usr/bin/mk_cmds
/usr/bin/mkfontscale             /usr/bin/mkfifo
/bin/mksh                        /usr/bin/mkfontdir
/usr/lib/plan9/bin/mk9660        /bin/mknod
/bin/mkdir                       /usr/bin/mktemp
/bin/mkfifo                      /usr/bin/mkdep
/usr/lib/plan9/bin/mklatinkbd    /bin/mkfifo
/usr/bin/mktemp                  /usr/bin/mkfontscale
/usr/bin/mk_cmds                 /bin/mksh
/bin/mkdir                       /bin/mktemp
/usr/bin/mkfontdir
$ mk
```

5 Scripts

A *shell* possui funcionalidades que permitem o seu uso como linguagem de programação.

5.1 Bourne shell

```
# Para a criação de variáveis
str="string"
var=321

# Para acessar o valor da variável, basta inserir o carácter "$"antes
# do nome da variável
echo $str
```

```
# São utilizadas estruturas de programação das linguagens
# de programação estruturadas
if cmd; then
# statements
elif cmd; then
# statements
else
# statements
fi

while cmd; do
# statements
done

# Para que os comandos (if, elif, while) sejam executados, o comando sendo executado
# deve executar sem erros, i.e, deve retornar 0

# O comando "for"se assemelha com o comando "for...in"do pascal
for var in elem0 elem1 ... ; do
# statements
done

# O comando test ([, ou [[), testa arquivos e compara valores, e geralmente é
# utilizado juntamente com o if, ou while
test -n "a␣string"  # testa se a string "a string"tem tamanho maior que 0
[ $str = "string" ] # testa se variável str é igual à string "string"
[[ $var -eq 123 ]]  # testa se variável var tem valor igual 123

# Vetores (bash e ksh)
arr=() # Cria um vetor vazio de nome arr
arr=(1 2 3) # Cria um array com valores 1, 2 e 3
echo ${arr[1]} # acessa posição 1 do vetor
echo ${arr[@]} # exibe todos os elementos do vetor
echo ${#arr[@]} # exibe o número de elementos do vetor
```

6 Arquivos

Uma das principais características do Unix é o sistema de arquivos [19], e os tipos de arquivo podem ser subdivididos em três subcategorias: Arquivos comuns, diretórios e arquivos especiais.

6.1 Arquivos comuns

Arquivos comuns são aqueles que contem qualquer informação que o usuário desejar, por exemplo um arquivo de texto ou um programa executável. Esse tipo de arquivo é normalmente encontrado em disco, e o sistema operacional (e a *shell*) não controla a estrutura desses arquivos, esse é o trabalho das aplicações do sistema. São comandos para a criação e manipulação de arquivos:

```
$ touch a     # cria arquivo "a"
$ rm a        # apaga arquivo "a'
$ >b          # cria arquivo "b", alternativa ao touch
$ mv b a      # renomeia arquivo "b"para "a"
$
$ mv a documents/     # move "a"para diretório "documents"
$ mv documents/a ~/c  # move "a"em "documents"para a home com o nome "c"
$ cp c c1             # copia "c"em "c1", funcionamento próximo ao comando mv
$ # entrada
$ cat <c      # passa o conteúdo de "c"como entrada padrão do comando cat
$ # comando lê entrada até encontrar um marcador, nesse caso, a string "EOF"
$ cat <<EOF
> teste
> mais teste
> EOF
teste
mais teste
$  # string "meu texto secreto"como entrada do comando
$ Lb64encode <<<"meu␣texto␣secreto"
$ # saída
$ echo "teste" >a # a saída padrão do comando sobrescreve o arquivo "a"
$ cat a c >>c1     # saída do comando é concatenada com o conteúdo de "c1"
$ # pipes (|) são uma forma de comunicação entre comandos
$ echo "teste" |tr a-z A-Z # saída de echo é a entrada de tr
TESTE
```

6.2 Diretórios

São os diretórios (também chamados de pastas) do sistema, arquivos que armazenam referencias a outros arquivos.

O principal diretório do sistema é o diretório raiz do sistema (*root*), que contém todos os diretórios do sistema. Uma lista de alguns diretórios importantes do sistema são indicadas na tabela 2 [2].

[2]Existem tentativas para a padronização da estrutura de diretórios como o *System V* [12] ou o *Filesystem Hierarchy Standard* [20], e esse documento tenta mostrar as características de ambos

Diretório	**Descrição**
/	Diretório raiz
/boot	Armazena arquivos para o *boot* do sistema
/dev	Armazena diversos arquivos de dispositivo
/dev	Armazena diversos arquivos de dispositivo
/home	Armazena arquivos dos usuários do sistema
/lib	Armazena bibliotecas compartilhadas
/media	Ponto para a montagem de dispositivos removíveis
/mnt	Ponto para a montagem temporária de um sistema de arquivos
/opt	Extensão para empacotadores de software
/proc	Armazena o sistema de arquivos procfs, que contém informações sobre os processos do sistema
/root	Diretório do super-usuário
/run	Armazena arquivos dos processos em execução
/bin	Comandos essenciais do sistema
/sbin	Executáveis essenciais do sistema
/srv	Informações dos serviços prestados pelo sistema
/tmp	Arquivos temporários
/usr	Hierarquia secundária
/usr/bin	Arquivos executáveis do usuário
/usr/lib	Bibliotecas
/usr/sbin	Executáveis não essenciais do sistema
/usr/share	Arquivos que são dependentes da arquitetura do computador
/var	Arquivos diversos

Tabela 2: Estrutura dos diretórios

São comandos para a criação e manipulação de diretórios:

```
$ pwd # indica diretório atual
/home/user
$ mkdir teste # cria diretório teste
$ cd teste # muda diretório atual para teste
$ pwd
/home/user/teste
$ cd /usr/local
$ cd # muda do diretório atual para a home do usuário
$ pwd
/home/user
$ mkdir -p teste/a # cria diretório "a"no diretório teste
$ cd teste/a # quando o caminho para o diretório não começa em /, cd usa o diretório
    atual como raiz
```

6.3 Arquivos especiais

Esse tipo arquivo é uma forma de usar dispositivos de entrada e saída (disco, terminal, etc) utilizando a interface dos arquivos, por exemplo comando `echo 'uma mensagem de erro ' >/dev/stderr` envia uma mensagem para o fluxo de erro padrão ao invés da saída padrão, onde `/dev/stderr` é o arquivo especial.

6.4 Permissões de arquivo

É possível alterar as configurações de acesso de um arquivo, por exemplo:

```
$ cd teste/
$ >a
$ ll # ls -l
total 0
-rw------- 1 user user 0 Aug 13 13:35 a
```

As permissões de um arquivo estão descritas na tabela 4, além disso, um arquivo têm três classes de níveis de acesso, indicadas pela tabela 3.

Modo	Descrição
usuário (*user*)	Permissões dadas para o usuário que criou o arquivo
grupo (*group*)	Permissões dadas para os usuário que pertencem ao grupo ao qual o arquivo foi atribuído.
outros (*others*)	Permissões dadas aos demais usuários do sistema

Tabela 3: Classes de acesso

Permissão	Notação numérica	Notação simbólica
Execução	1	x
Escrita	2	w
Leitura	4	r
Escrita e leitura	6	rw
Execução e escrita	3	wx
Execução e leitura	5	r-x
Execução, escrita e leitura	7	rwx

Tabela 4: Permissões de arquivo

São comandos para a manipulação de permissões de um arquivo:

```
$ chmod 755 a  # muda as permissões de "a"para 7 (usuário) 5 (grupo) 5 (outros)
$ umask 033    # Novos arquivos terão permissão 0744
$ >b
$ ll b
-rw-r--r-- 1 user user 0 Aug 13 14:24 b
$ groups           # mostra a quais grupos o usuário pertence
user video
$ chgrp video b    # muda o grupo de "b"para video
$ chown user b     # Muda o usuário do arquivo para user
chown: invalid user: 'user'
$ chown user:user b # Muda o usuário e grupo de "b"para o usuário user e grupo user
```

6.5 Entrada e saída

6.5.1 Redirecionamento de E/S

Ao ser executado, um comando herda da *shell* a entrada e saída padrão, assim como a saída de erros (descritores de arquivo 0, 1 e 2 respectivamente). Com o redirecionamento de entrada e saída é possível alterar a origem da entrada e saída dos comandos, criar e alterar arquivos.

```
$ # exemplos funcionam na ksh, bash
$ >a # cria arquivo a
$ echo 'some text' >b # cria arquivo b com conteúdo 'some text'
$ echo 'text' >a # substitui arquivo conteúdo de a por 'text'
$ cat <a # usa arquivo a como entrada do comando
text
$ echo 'more text' >>b # concatena 'more text' no arquivo b
$ echo 'this is an error' >&2 # redireciona saída para saída de erros padrão
this is an error
```

6.5.2 Pipes

Pipes são canais para comunicação inter-processo, onde os fluxos padrões dos diferentes processos são conectados entre si, por exemplo: `ls -l |grep "string"|less`, o funciona-

mento é o seguinte:

O comando `ls -l` é executado, e sua saída padrão torna-se a entrada para o comando `grep "string"`, que por sua vez passa a sua saída para a entrada do comando `less`.

7 Processos

É possível visualizar e controlar os processos da máquina em uma seção interativa da *shell*, e por extensão nos scripts. A tabela 5 descreve alguns comandos para o controle de processos na *shell*

8 Casos de uso avançados

8.1 Multiplexadores de terminal

Para facilitar o uso do terminal, diversos emuladores de terminal possuem características como criar diversas sessões *shell* em abas, permitindo a separação de diversas sessões.

Com utilitarios como GNU screen[1], tmux[3] ou dvtm[25], é possível criar e visualizar diversas sessões *shell* simultaneamente.

Alguns dos comandos do dvtm são mostrados na tabela 6, e um exemplo de uma sessão de uso pode ser vista na figura 1.

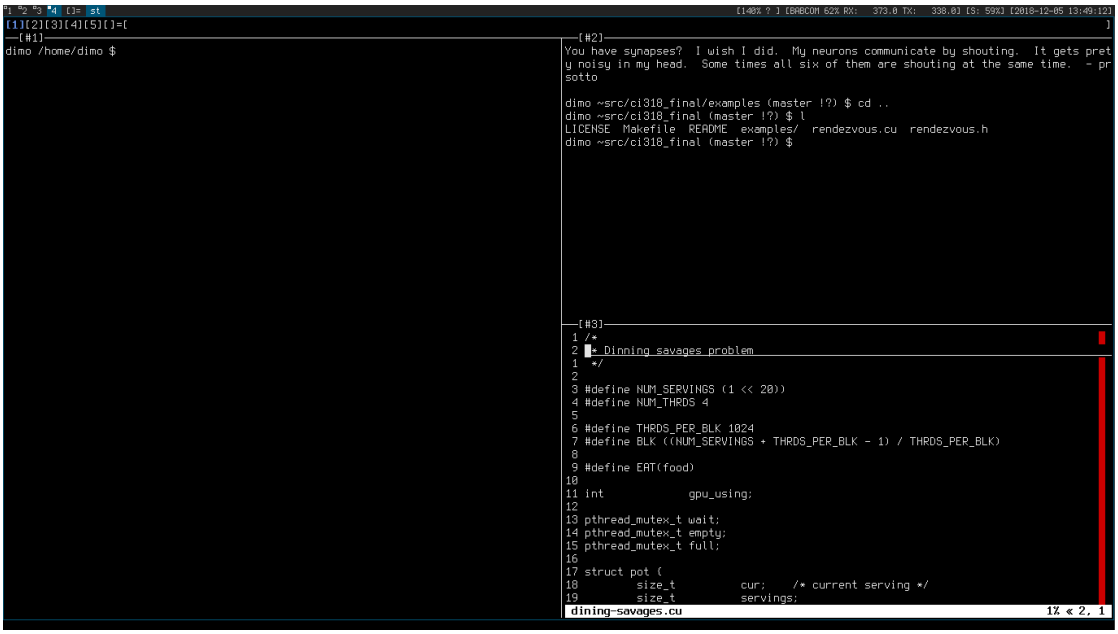

Figura 1: Exemplo sessão dvtm

Comando	**Descrição**
comando &	O e comercial após um comanda indica para a *shell* não esperar o término do comando em questão, e continuar sua execução normalmente. O *process id* (pid) do último comando executado se encontra na variável especial $!
ˆ- Z	Envia para o sinal SIGSTOP para o comando que está sendo executado, o que bloqueia o processo até o recebimento do sinal SIGCONT
jobs	Lista os processos criados pela *shell* e em qual estado se encontram.
fg [%n]	Executa o comando indicado pelo id *n* (ou o último processo) executado pela *shell* e executa-o em *foreground*, i.e, a *shell* irá bloquear e esperar o termino do comando em questão
bg [%n]	Executa o comando indicado pelo id *n* (ou o último processo) executado pela *shell* e executa-o em *background*, i.e, a *shell* não irá bloquear para esperar o término do comando em questão
wait pid ...	A a *shell* espera o término dos processos indicados pelos *pids* passados para o comando, caso não sejam passados nenhum argumento, o comando faz com que a *shell* espere pelo término de todos os comandos criados por ela
ps	Utilitário que exibe informações sobre processos ativos
kill [-9 |-s n] pid ...	Envia um sinal (padrão SIGTERM) para o processo indicado pelo *pid*
top	Utilitário que exibe informações sobre processos ativos, além de ter opções para enviar signais para esses processos, e uma interface "gráfica"para facilitar seu uso.

Tabela 5: Controle de processos

Comando	Descrição
MOD - c	Cria nova janela
MOD - C	Cria nova janela utilizando o diretório atual da janela que está focada
MOD - j	Foca na próxima janela
MOD - k	Foca na janela anterior
MOD - [0..9]	Foca na janela N
MOD - v - [0..9]	Foca na *tag* N

Tabela 6: Comandos dvtm, onde MOD é, por padrão, ^- G

8.2 Persistência de sessão

Além de realizar a multiplexação do terminal, o GNU Screen e o tmux, permitem com que a sessão do usuário tenha persistência, i.e, permitem que o usuário possa suspender sua sessão de uso e recupera-lá no futuro. O utilitário standalone `abduco`[24], possui essa mesma finalidade, porém permite com que a função seja utilizada com outros comandos que não sejam necessariamente um multiplexador de terminal.

Outro comando que realiza uma função parecida é o `nohup`, util para a execução de comandos de longa duração em uma sessão remota, já que redireciona a entrada e saída dos comandos para arquivos especiais e faz com que o sinal SIGUP seja ignorado pelo comando invocado.

9 Práticas em laboratório

9.1 Primeira prática

9.1.1 Parte 1

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

Solução:

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

BEGIN {
      FS = ";"
}

{
    for (i = 1; i < NF; ++i) {
        print $i
    }
}
```

9.1.2 Parte 2

Obter em um arquivo separado a lista de locais (coluna 5 do arquivo) de forma única e ordenada.

9.1.3 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") Solução:

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

{
        gsub("([^\"];|;[^\"])", "\t")
        print >>"new"
}

{
        nr = split($0, ln, ";")
        for (i = 1; i < nr; ++i)  {
                local = ln[5];
                locais[local] = locais[local] "\n" $0
        }
}



END {
        for (l in locais)
                print i >>"locais.txt"

        for (l in locais)
                print locais[l] >>l".csv"
}
```

9.1.4 Discussão

9.2 Segunda prática

Solução:

```
#!/bin/mksh

for f in ${DIR:=$PWD}/CI*/*.dados; {
        w=($(<$f))
        unset w[0]

        sem="${f%.dados*}"
        sem="${sem##*/}"
        ENROLLNO[$sem]="${w[@]} ${ENROLLNO[$sem]}"
}

for i in ${!ENROLLNO[@]}; {
        (( n = $(tr ' ' '\n' <<<"${ENROLLNO[$i]}" |sort -u |wc -l) - 1 ))
        printf '%5d:%d\n' $i $n
}
```

Solução:

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

BEGIN {
        FS = ":"

        for (i = 1988; i < 2003; ++i) {
                ln0 = ln0 sprintf("\t%d1\t%d2", i, i)
                SEMESTER[i"1"] = SEMESTER[i"2"] = ""
        }
}

/^curso/ {
        split(FILENAME, arr, "/")
        subj = arr[1]
        split(arr[2], arr, ".")
        sem = arr[1]
}

/^[0-9]/ {
        COURSE[$1] = ""
        all[$1, sem]++
}

END {
        print ln0

        for (c in COURSE) {
                ln = sprintf("%3d\t", c)
                for (s in SEMESTER)
                        ln = ln sprintf("%5d\t", all[c,s])
                print ln
        }
}
```

9.3 Terceira prática

Solução:

```
#!/bin/mksh
for log ; {
        out=${log/%xz/log}
        xzcat "$log" |
            awk '
                BEGIN {
                        while (getline <"tipos-bloqueios")
                                other[$0":"] = 0
                }

                $6 ~ /V6/ {
                        v6[$6]++
                        delete other[$6]
                        next
                }

                {
                        v4[$6]++
                        delete other[$6]
                }
                END {
                        print "***␣V4␣***"
                        for (filter in v4)
                                print filter v4[filter]

                        print "***␣V6␣***"
                        for (filter in v6)
                                print filter v6[filter]

                        for (filter in other)
                                print filter "0"
                }' >$out
}
```

Solução:

```
#!/bin/mksh
att=()

for log in ${DIR:=~/nobackup}*.log {
        awk -F':' '$2 >= 20000 { exit 0 }' && att=(${att[@]} "${log/%log/
            ↪ xz}")
}

(( ${#att[@]} > 0 ))  mail -a att[@] -s'WARNINGFIREWALL'my@email.com
```

Referências

[1] Gnu screen. https://www.gnu.org/software/screen/. [Online, acessado em 15 de Novembro de 2018].

[2] Oh my zsh! https://github.com/robbyrussell/oh-my-zsh/. [Online, repositório, acessado em 17 de Agosto de 2018].

[3] tmux. https://github.com/tmux/tmux/wiki. [Online, acessado em 15 de Novembro de 2018].

[4] Stephen R Bourne. *An introduction to the UNIX shell*. Bell Laboratories. Computing Science, 1978.

[5] cat v.org. Unix keyboard shortcuts. http://unix-kb.cat-v.org/. [Online, acessado em 10 de Outubro de 2018].

[6] Tom Christiansen. Csh programming considered harmful. http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/, 1995. [Online, acessado em 01 de Agosto de 2018].

[7] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. http://www.multicians.org/fjcc1.html, 2000. [Online, acessado em 30 de Julho de 2018].

[8] DistroWatch. Distrowatch page hit ranking. https://www.distrowatch.com/dwres.php?resource=popularity. [Online, acessado em 15 de Agosto de 2018].

[9] Brian Fox et al. Bash. https://www.gnu.org/software/bash/, 2017. [Online, acessado em 30 de Julho de 2018].

[10] IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. Standard, Institute of Electrical and Electronics Engineers and The Open Group, 2017. [Chapter 2: Shell Command Language, Online em http://pubs.opengroup.org/onlinepubs/9699919799/, acessado em 11 de Agosto de 2018].

[11] David G Korn. ksh: An extensible high level language. 1994.

[12] UNIX System Laboratories. *System V interface definition*. Unix System V, Vol. 1. UNIX Press, 1991.

[13] Marshall Kirk McKusick. Twenty years of berkeley unix. https://www.oreilly.com/openbook/opensources/book/kirkmck.html, 2000. [Online, acessado em 15 de Agosto de 2018].

[14] Jeffrey Allen Neitzel. History - etsh project (v6shell). https://etsh.io/history/. [Online, acessado em 31 de Julho de 2018].

[15] J.S. Pendergrast. *Desktop KornShell graphical programming*. Addison-Wesley professional computing series. Addison-Wesley Pub. Co., 1995.

[16] Rob Pike. The unix system: The blit: A multiplexed graphics terminal. *AT&T Bell Laboratories Technical Journal*, 63(8):1607–1631, 1984.

[17] Louis Pouzin. The origin of the shell. http://www.multicians.org/shell.html, 2000. [Online, acessado em 30 de Julho de 2018].

[18] Eric S. Raymond. The jargon file. http://www.catb.org/jargon/html/S/shell.html. [Online, acessado em 15 de Agosto de 2018].

[19] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[20] Rusty Russell, Daniel Quinlan, and Christopher Yeoh. Filesystem hierarchy standard. *V2*, 3:29, 2004.

[21] Eric Smith. Att 5620 (and related terminals) frequently asked questions. http://www.brouhaha.com/ eric/retrocomputing/att/5620/5620$_f aq.html$, 2005.[$Online, acessadoem1$

[22] Peter Stephenson. Z-shell frequently-asked questions. http://zsh.sourceforge.net/FAQ/. [Chapter 2: How does zsh differ from...?, Online, acessado em 17 de Agosto de 2018].

[23] Peter Stephenson et al. The z shell manual. http://zsh.sourceforge.net/Doc/Release/index.html. [Online, repositório, acessado em 17 de Agosto de 2018].

[24] Marc André Tanner. abduco. https://github.com/martanne/abduco. [Online, acessado em 15 de Novembro de 2018].

[25] Marc André Tanner. dvtm. https://github.com/martanne/dvtm. [Online, acessado em 15 de Novembro de 2018].

[26] Tom Van Vleck, John Gintell, Joachim Pense, Monte Davidoff, Andi Kleen, Doug Quebbeman, and Jerry Saltzer. Myths about multics, 2015. [Arquivado em https://web.archive.org/web/20180114035700/http://www.multicians.org/myths.htmlfail69, acessado em 31 de Julho de 2018].

[27] Linus Åkesson. The tty demystified. https://www.linusakesson.net/programming/tty/index.php. [Online, acessado em 10 de Outubro de 2018].