

Sumário

Capítulo 1

Funcionalidade do Sistema Operacional

1.1 crontab

O crontab é utilizado para programar tarefas constantes em determinadas datas. As tarefas estão descritas no arquivo passado como argumento. Esse arquivo deve possuir linhas com 6 campos separados por <tab> ou espaço. Cada linha especifica uma tarefa, seguindo o padrão abaixo:

```
<minuto> <hora> <dia> <mês> <dia da semana> <tarefa>
```

Utilizamos o caracter “*” caso o campo não seja especificado. Isso mostra ao crontab que a ação deve acontecer sempre.

Exemplo:

```
MAILTO=joao@mailqualquer.com.br
0 4 * * * cal
1,21 * * * * prog1
30 4 1 * * prog2
```

Neste exemplo o comando *cal* vai ser executado às 4 horas todos os dias, o programa *prog1* será executado no minuto 1 e no minuto 21 de cada hora todos os dias e o programa *prog2* será executado dia 1 às 4:30 de cada mês.

Sintaxe: `crontab [opções] <arquivo>`

Opções:

Opção	Descrição
-r	Remove a atual programação do usuário.
-l	Mostra atual programação do usuário.
-e	Edita o crontab usando o editor especificado na variável ambiente VISUAL.

Exemplos:

```
$ crontab arquivo1
```

Neste exemplo programa-se todas as tarefas que estão listadas no arquivo *arquivo1* que está no padrão do crontab. Feito isso é gerado um arquivo no diretório */var/spool/cron* com o nome do usuário, o qual o servidor cron procura e carrega na memória. Quando os comandos são executados, qualquer saída é enviada via mail para o dono do arquivo crontab (ou para outro usuário, que pode ser definido na variável de ambiente MAILTO, em arquivo1 do exemplo).

```
$ crontab -r
```

Remove a programação do usuário que executou o comando.

```
$ crontab -e
```

Edita a programação do usuário.

1.2 fstab

O arquivo */etc/fstab* contém a descrição dos vários sistemas de arquivos. Cada sistema de arquivos é descrito em uma linha, campos em cada linha são separados por tabulações ou espaços. A ordem dos registros no *fstab* é importante porque o *fsck*, *mount*, e *umount* interagem seqüencialmente através do *fstab* para executar suas tarefas. Os campos de cada registro são dispostos da maneira abaixo:

```
<fs_spec> <fs_file> <fs_vfstype> <fs_mntopts> <fs_freq> <fs_passno>
```

O primeiro campo (*fs_spec*) descreve um dispositivo especial de blocos ou um sistema de arquivos remoto a ser montado.

O segundo campo (*fs_file*) descreve o ponto de montagem para o sistema de arquivos. Para partições da área de *swap*, este campo deve ser especificado como “none”.

O terceiro campo (*fs_vfstype*) descreve o tipo do sistema de arquivos.

Exemplos:

<i>Opção</i>	<i>Descrição</i>
vfat	Sistema de arquivos FAT do windows, com suporte a nome de arquivos longos.
ext2	Um sistema de arquivos local, com nomes de arquivos longos, inodes maiores e muitas outras funcionalidades.
msdos	Um sistema de arquivos local para partições ms-dos.
iso9660	Um sistema de arquivos local usado para dispositivos cd-rom.
nfs	Um sistema de arquivos para montagem de partições a partir de sistemas remotos.
swap	Uma partição de disco usada como memória auxiliar.
auto*	Tenta descobrir automaticamente qual é o tipo do sistema de arquivos a ser montado.

O quarto campo (*fs_mntops*) descreve as opções de montagem associadas com o sistema de arquivos, cada opção vem separada por vírgula, as principais opções são descritas a seguir:

<i>Opção</i>	<i>Descrição</i>
auto	O sistema de arquivos será montado na inicialização do sistema operacional.
exec	Permite a execução de arquivos binários.
noauto	O sistema de arquivos não será montado automaticamente
noexec	Não permite a execução de arquivos binários.
nouser	Proíbe que um usuário normal monte o sistema de arquivos.
ro	Monta somente com permissão de leitura.
rw	Monta com permissão de leitura e gravação.
user	Permite que um usuário normal monte o sistema de arquivos.
defaults	Usa as opções <i>rw</i> , <i>exec</i> , <i>auto</i> e <i>nouser</i> para montar o sistema de arquivos.

O quinto campo (*fs_freq*) é usado nos sistemas de arquivos pelo comando *dump* para determinar quais sistemas de arquivos necessitam ser copiados. Caso o quinto campo não esteja presente, um valor zero é retornado e *dump* assumirá que o sistema de arquivos não necessita ser copiado.

O sexto campo (*fs_passno*) é usado pelo programa *fsck* para determinar a ordem em que os sistemas de arquivos são checados durante a reinicialização do sistema. O sistema de arquivos raiz deve ser especificado com um valor 1 em *fs_passno*, e outros sistemas de arquivos devem ter *fs_passno* com o valor 2. Caso tenha valor 0, o *fsck* não checa o sistema de arquivos.

*Cuidado, pois o *auto* utiliza um método heurístico e pode reconhecer de forma equivocada o sistema de arquivos.

Rotinas[†] para ler e alterar registros do arquivo `fstab` são declaradas em `<getmntent.h>`. `setmntent()` abre um arquivo para ser usado como `fstab`, `getmntent()` lê uma linha do arquivo aberto com a função `setmntent()`, `addmntent()` adiciona uma linha no `fstab`, `endmntent()` fecha e grava as informações do arquivo `fstab`.

1.3 mount

Monta sistema de arquivos. Para entender o que isso quer dizer é necessário saber que todos os arquivos acessíveis em um sistema Unix estão organizados em uma grande árvore, e a hierarquia desses arquivos é iniciada pelo raiz, simbolizado como `/`. Estes arquivos podem estar distribuídos por diversos dispositivos. O comando `mount` destina-se a incluir o sistema de arquivos encontrado em algum dispositivo à grande árvore de arquivos, enquanto que o comando `umount` executará a ação inversa.

Executando-se o `mount` sem nenhum argumento, temos como resultado todos os sistemas de arquivos montados.

Com o comando `mount` é possível montar sistemas de arquivos remotamente; para isso basta colocar o nome da máquina, `”:`” e nome do diretório no campo `<partição>`.

Sintaxe: `mount [opções] <partição> <diretório>`

Opção	Descrição
-t	Indica o tipo de partição a ser montada.
-a	Monta automaticamente todos os sistemas de arquivos indicados em <code>/etc/fstab</code> .
-r	Monta somente com permissão de leitura.
-w	Monta com permissão de escrita e leitura.
-o	São passadas as opções de montagem vistas na seção ??.

As rotinas para montar e desmontar sistema de arquivos são definidas em `<sys/mount.h>`. `mount()`, monta um sistema de arquivos de acordo com seus parâmetros. `umount()`, desmonta um sistema de arquivos previamente montado.

Exemplos:

```
$ mount /dev/hda1 /mnt
```

Monta a partição `/dev/hda1` no diretório `/mnt`.

[†]Funções para linguagem C.

```
$ mount -a
```

Monta automaticamente todos os sistemas de arquivos.

```
$ mount
```

Mostra todos os sistemas de arquivos montados.

```
$ mount maquina1:/home /mnt
```

Monta o sistema de arquivos da *maquina1* (remota) no diretório `mnt` local.

1.4 umount

Usado para desmontar sistemas de arquivos.

Sintaxe: `umount [opções] <partição/diretório>`

Opções:

Opção	Descrição
-t	Indica o tipo de partição a ser desmontada.
-a	Desmonta automaticamente todos os sistemas de arquivos indicados em <code>/etc/fstab</code> .

Exemplos:

```
$ umount -a
```

Desmonta todos os sistemas de arquivos montados.

```
$ umount /dev/hda1
```

Desmonta o sistema de arquivos `/dev/hda1`.

```
$ umount /mnt
```

Desmonta o sistema de arquivos que está montado no `/mnt`.

1.5 at

Agenda uma tarefa para ser executada numa determinada data. Existe dois comandos relacionados com o *at*: *atq* e o *atrm*. Estes comandos são semelhantes ao comando *at* com a opção *-l* ou *-r* respectivamente.

Sintaxe: *at* [hora] [mês] [dia] [opções] <comando>

Hora:

HHMM hora e minutos. Ex: 1500

HH(am/pm) hora pela manhã. Ex: 8am

Mês:

Mês nome do mês. Ex: jan

Dia:

DD dia do mês. Ex: 23

<i>Opção</i>	<i>Descrição</i>
-f	Especifica o nome do arquivo a se executado.
-l	Mostra os processos agendados
-d <job>	Remove o processo <job> agendado.

Exemplos:

```
$ at 8am jan 24 < teste
```

Executa o arquivo *teste* no dia 24 de janeiro às 8 horas da manhã.

1.6 processo

Um processo é um programa em execução. Vários processos são executados concorrentemente, a medida que são escalonados pelo kernel, e vários processos podem ser instâncias de um mesmo programa. Todos os processos são criados por algum outro processo, chamado processo pai. A cada processo do sistema são associados:

- PID (Process Identification): número que identifica de forma única um processo;
- PPID (Parent Process Identification): PID do processo pai;
- UID (User Identification): identificador do usuário que criou o processo (dono do processo);
- GID (Group Identification): número do grupo inicial do dono do processo;

O processo 0 é o único processo que não tem pai. Ele é criado pelo próprio kernel, durante o boot do sistema. O processo 1, conhecido como *init*, é criado pelo processo 0 e é o ancestral de todos os outros processos do sistema.

1.7 daemons

Daemons são processos que realizam operações para o kernel do sistema, que não estão associados a nenhum terminal. Existem daemons de sistema e de usuários. Os de sistema realizam tarefas exclusivas do sistema operacional, como é o caso do *init*. Os daemons de usuários realizam tarefas que são necessárias como suporte para atividades de usuários ou de aplicações usadas por estes, como por exemplo: *crond*, *nfsd*, *inetd*, etc.

O *nfsd* é o daemon responsável pela operação do NFS (Network File System) que é o responsável pela montagem de sistema de arquivos remotos. O *inetd* supervisiona os protocolos de transmissões de dados na internet como o TCP-IP, ele inicializa programas quando eles são requisitados. O *crond* é a agenda do sistema, ele agenda os processos que são executados periodicamente.

Os daemons geralmente mantêm-se ativos durante todo o tempo e a maioria deles são inicializados em tempo de boot nos scripts */etc/rc*.

Em `<unistd.h>` é declarada a rotina *daemon()* que serve para transformar um programa em um deamon, ou seja, desvincula do terminal que foi executado e deixa em background.

Capítulo 2

Servidores do Linux

2.1 SAMBA

Samba é um conjunto de programas para Linux que usam o protocolo SMB (Server Message Block). Vários sistemas operacionais, incluindo Windows e OS/2, usam SMB, por isso o samba pode comunicar-se com Windows. Proporciona compartilhamento de arquivos e impressoras.

O servidor de arquivos e impressão é inicializado com o comando *smbd* e o servidor de nomes para o protocolo NETBIOS é inicializado com o comando *nmbd*. O arquivo padrão de configuração do samba é o */etc/smb.conf* (arquivo ??);

2.1.1 Utilitários usados com o samba

findsmb

Procura por servidores para arquivos e impressão estilo NT. Exemplo:

```
$ findsmb
```

IP ADDR	NETBIOS NAME	WORKGROUP/OS/VERSION
192.168.1.1	alpha	
192.168.1.2	juli	

nmblookup

Procura por nomes no protocolo NETBIOS. Exemplo:

```
$ nmblookup alpha
Sending queries to 192.168.1.255
192.168.1.1 alpha<00>
```

Arquivo 2.1 smb.conf

```
[global]
# Nome do grupo
  workgroup = linux

# Descrição do grupo
  server string = Samba Server

# Definição básica de impressão
  printcap name = /etc/printcap
  load printers = yes

# Definição da criptografia para o servidor
  encrypt passwords = yes
  smb passwd file = /etc/smbpasswd

#===== COMPARTILHAMENTO =====
[homes]
  comment = Directorio Home
  browseable = yes
  writable = yes

[arquivos]
  comment = Arquivos
  browseable = yes
  writable = no
  path = /arquivos
  guest ok = yes

# Configuracao da Impressora
[printers]
  comment = All Printers
  browseable = yes
  path = /tmp
  printable = yes
  public = yes
  writable = yes
```

smbadduser

Adiciona um novo usuário ao servidor samba e o guarda dentro do arquivo `/etc/smbusers` (arquivo ??).

Arquivo 2.2 smbusers

```
# Unix_name = SMB_name1 SMB_name2 ...
root = administrator admin
nobody = guest pcguest smbguest
nene = teste
mp3 = zakharoffm
```

Sintaxe: `smbaduser <nome UNIX>:<nome samba>`

Exemplo:

```
$ smbadduser mp3:outro
Adding: mp3 to /etc/smbpasswd
Adding: {mp3 = outro} to /etc/smbusers
```

```
-----
ENTER password for mp3
New SMB password:
Retype new SMB password:
Password changed for user mp3.
```

Faz com que o usuário `mp3` do sistema linux seja também usuário do servidor samba mas com o nome de `outro`.

smbpasswd

Altera a senha de um usuario do servidor samba, todas as senhas dos usuarios do servidor samba ficam localizadas no arquivo `/etc/smbpasswd`.

Exemplo:

```
$ smbpasswd root
New SMB password:
Retype new SMB password:
Password changed for user root.
```

Altera as senha do usuário `root` no servidor samba.

smbmount

Monta um sistema de arquivo do servidor samba em um ponto de montagem local qualquer.

Sintaxe:

```
smbmount //sevidor/diretório <ponto de montagem local> [OPÇÕES]
```

Opção	Descrição
-U <user>	Nome do usuário que está montando o sistema de arquivos.
-I <IP>	IP da máquina da qual o sistema de arquivos será montado.

Exemplos:

```
$ smbmount //alpha/arquivos /mnt
```

Monta o diretório arquivos do servidor alpha no ponto de montagem local /mnt.

```
$ smbmount //alpha/dir /mnt -U aba -I 192.168.1.1
```

O usuário *aba* monta o diretório dir do servidor *alpha* no ponto de montagem local /mnt, somente se o servidor *alpha* tiver o IP 192.168.1.1.

smbumount

Desmonta um sistema de arquivo samba previamente montado em algum diretório.

Sintaxe: `smbumount <ponto de montagem local>`

Exemplo:

```
$ smbumount /mnt
```

Desmonta um sistema de arquivo montado no diretório /mnt.

smbstatus

Mostra a situação atual do servidor. Exemplo:

```
$ smbstatus
Samba version 2.0.5a
Service    uid      gid      pid      machine
-----
arquivos   juli     juli     2673     alpha    (192.168.1.1) Sun Aug 27 20:04:58 2000

No locked files
```

Share mode memory usage (bytes):

```
% 1048464(99%) free + 56(0%) used + 56(0%) overhead = 1048576(100%) total
```

smbclient

Acessa um servidor de maneira parecida a um cliente ftp.

Sintaxe: `smbclient //servidor/diretório`

Exemplo:

```
$ smbclient //alpha/arquivos -U juli
Added interface ip=192.168.1.1 bcast=192.168.1.255 nmask=255.255.255.0
Password:
Domain=[LINUX] OS=[Unix] Server=[Samba 2.0.5a]
smb: \> dir
  redhat                D            0 Tue May  2 14:35:39 2000
  kernel                D            0 Wed Aug  9 23:08:20 2000
  download.tar.gz       111125320   Wed Aug  9 15:35:51 2000
  arq_casa              D            0 Fri Feb 25 12:50:04 2000
  disco_d              D            0 Thu Dec  2 19:52:21 1999
  disco_e              D            0 Wed Aug 23 10:57:16 2000
  drivers              D            0 Tue May  2 13:45:58 2000

                                61613 blocks of size 65536. 3881 blocks available
smb: \> quit
```

2.2 LPD

Servidor de impressão para o linux (line printer spooler daemon). Usa chamadas de sistema para receber solicitações de impressão e consultas na situação atual da fila de impressão. Inicializado com o programa `lpd`, consulta o arquivo `/etc/printcap` (arquivo ??) para ver quais impressoras estão instaladas no sistema e procura pelas permissões de acesso em `/etc/hosts.lpd` ou `/etc/hosts.equiv`.

2.2.1 Utilitários usados com o servidor de impressão

lpc

Abre um prompt, para controlar as tarefas na linha de impressão. Exemplo:

```
$ lpc
lpc> ?
Commands may be abbreviated.  Commands are:

abort    enable  disable help    restart status topq   ?
clean   exit   down   quit   start  stop   up
lpc> topq canon root
```

Arquivo 2.3 printcap

```
##PRINTTOOL3## LOCAL bjc600 360x360 a4 {} BJC600 8 {}
canon:\
    :sd=/var/spool/lpd/canon:\
    :mx#0:\
    :sh:\
    :lp=/dev/lp0:\
    :if=/var/spool/lpd/canon/filter:
##PRINTTOOL3## SMB iwhi 120x144 letter {} ImageWriterHI Default {}
lexmark:\
    :sd=/var/spool/lpd/lexmark:\
    :mx#0:\
    :sh:\
    :if=/var/spool/lpd/lexmark/filter:\
    :af=/var/spool/lpd/lexmark/acct:\
    :lp=/dev/null:
```

```
canon:
    moved cfA209AsW4bax
lpc> down all
canon:
    printer and queuing disabled
lexmark:
    printer and queuing disabled
lpc> quit
```

O comando `?` mostra os comandos aceitáveis pelo `lpc`, o comando `topq canon root` mostra as tarefas do usuário `root` na impressora `canon`, `down all` faz com que todas as impressoras para de aceitar requisições, e o comando `quit` sai do programa.

lpr

Manda um arquivo para fila de impressão de uma determinada impressora.

Opção	Descrição
-P <impressora>	Manda imprimir alguma coisa na impressora especificada, se essa opção não for usada a tarefa será enviada para impressora que tenha o nome na variável ambiente <i>PRINTER</i> .
-# <cópia>	Escolhe o número de cópias a serem impressas. montado.

Exemplo:

```
$ lpr -Pimpr -#2 arquivo.ps
```

Imprime duas cópias do arquivo.ps na impressora *impr*.

lpq

Examina a fila de impressão.

<i>Opção</i>	<i>Descrição</i>
-l	mostra mais informações sobre as tarefas que estão na fila de impressão.
-P <impressora>	Manda imprimir alguma coisa na impressora especificada.

lprm

Apaga uma tarefa da fila de impressão.

Sintaxe: lprm [opções] [tarefa #] [usuário]

<i>Opção</i>	<i>Descrição</i>
-	remove todas as tarefas de um usuário. Caso seja root apaga todas as tarefas da fila de impressão.
-P <impressora>	Manda imprimir alguma coisa na impressora especificada.

Exemplos:

```
$ lprm -Pprin 343
```

Apaga o trabalho 343 da fila de impressão.

```
$ lprm -
```

Se for root apagará todas as tarefas da fila de impressão

2.3 RANDOM

Servidor que salva e restaura a geração de números aleatórios reais, não somente em relação ao relógio e data do sistema. Quando o computador é desligado ele salva uma semente de números aleatórios que será usada quando o computador for ligado novamente.

Para gerar números aleatórios reais o servidor busca maneiras de pegar tudo o que o computador faz, como por exemplo o tempo de digitação do

usuário ou a frequência de acesso a periféricos, e manipular tudo isso com objetivo de gerar esses números.

As seqüências de números aleatórios podem ser encontradas na variável ambiente `RANDOM` ou em `/dev/random` que provê grandes seqüências aleatórias.

Uma das vantagens de utilizar seqüências aleatórias reais é no envio de pacotes TCP/IP que são controlados por uma seqüência de números para sincronia da transmissão de dados. Quando não são usadas seqüências aleatórias reais um intruso pode prever qual será a próxima seqüência e enviar um pacote qualquer para o destinatário ou ainda interceptar mensagens privadas. (Joncheray, Laurent. *A Simple Active Attack Against TCP*)

2.4 SYSLOGD

Servidor encarregado de armazenar os logs do sistema. Há um módulo que guarda as mensagens da kernel. Junto com as mensagens guarda a hora em que ocorreu. Sua configuração está no arquivo `/etc/syslog.conf` (arquivo ??).

Arquivo 2.4 syslog.conf

```
# The authpriv file has restricted access.
authpriv.*                /var/log/secure

# Log all the mail messages in one place.
mail.*                    /var/log/maillog

# Todas as mensagens no terminal tty11
*. * /dev/tty11
```

Capítulo 3

Emacs

3.1 Introdução

Emacs é um dos editores de texto mais usados no mundo Unix hoje em dia. Muitos usuários preferem usar GNU Emacs a usar vi (editor padrão do Unix) ou um editor que é construído dentro de muitos sistemas de janelas. O Emacs foi reimplementado várias vezes, porém a primeira versão foi escrita por Richard Stallman com a idéia de ser um editor customizado, com display em tempo real, auto-documentado e extensível. Emacs é um completo ambiente de trabalho. Você pode usar o Emacs para: editar, renomear, deletar e copiar arquivos; para compilar programas; para trabalhar interativamente com uma Shell Unix; para ler e organizar e-mails; para acessar a internet, entre outras coisas. Emacs é também muito flexível pois você pode escrever seus próprios comandos, mudando as teclas associadas com os mesmos. Comandos em Emacs consistem de um modificador, como as teclas CTRL (CONTROL) or ESC (ESCAPE), seguidas por um ou dois caracteres. Para simplificar, vamos abreviar CTRL para C-. Por exemplo quando você encontrar nesse capítulo C-g, significa que você deve segurar CTRL e pressionar depois a tecla g.

3.2 O básico sobre o Emacs

3.2.1 Compreendendo Arquivos e Buffers

Editores não editam um arquivo em tempo real. Ao invés disso eles colocam o conteúdo de um arquivo em um buffer e o editam. O arquivo atual em disco não muda até que você diga ao editor para salvar o buffer. Lembre-se: um buffer se parece com um arquivo mas não o é, pois é somente um espaço temporário que contém uma cópia de um arquivo. Como arquivos,

os buffers no Emacs tem nomes. O nome de um buffer é usualmente o mesmo nome do arquivo que você está editando. Então, quando você abre um arquivo no Emacs, ele copia o conteúdo do arquivo para dentro de um buffer. Quando você edita o arquivo, você está na realidade modificando o buffer, e não o arquivo. Quando você estiver satisfeito com suas mudanças, você pode salvá-las. Seu arquivo somente é modificado quando você escolhe salvar suas mudanças. Se você decidir não salvar suas mudanças, você pode sair do Emacs sem salvar o arquivo, e nada será modificado no arquivo original.

3.2.2 Modos do Emacs

Um dos grande trunfos do Emacs é que ele tem vários modos de edição. Para cada modo de edição o Emacs se comporta de maneira diferente. Em um modo de programação por exemplo, o Emacs pode formatar corretamente o código, dependendo da linguagem. Para escrita, existe o modo texto; para programação existem vários tipos de modos; existem modos para diferentes linguagens, como o modo para linguagem C. Modos, então permitem ao Emacs ser um tipo de editor diferente para diferentes tarefas que você possa realizar. Alguns modos que o Emacs suporta:

- Fundamental mode → O modo default, nenhum comportamento especial
- Text mode → Para escrever textos
- Mail mode → Para escrever e-mails
- RMAIL mode → Para ler e organizar e-mail
- View mode → Para visualizar arquivos, mas não para editá-los
- Shell mode → Para rodar um shell Unix de dentro do Emacs
- Angle-ftp mode → Para fazer transferências ou visualizar arquivos em sistemas remotos
- Telnet mode → Para logar em um sistema remoto
- \LaTeX mode → Para formatar arquivos para \LaTeX
- C mode → Para escrever programas em C
- C++ mode → Para escrever programas em C++

3.2.3 Inicializando o Emacs

Para inicializar o Emacs, simplesmente digite *emacs* seguido do nome do arquivo que você deseja editar. Se você usar um nome de arquivo que não exista o Emacs criará um novo arquivo. Ex: *emacsmeuarquivo*

3.2.4 Abrindo um arquivo

Você pode abrir um arquivo especificando o nome do arquivo quando você inicializa o Emacs ou apenas digitando C-x C-f. C-x C-f cria um novo buffer que tem o mesmo nome do arquivo. Quando você digita o comando C-x C-f, aparecerá na última linha do Emacs uma linha em branco onde você poderá digitar o nome do arquivo que deseja abrir. O que você acha que acontece se você tentar abrir o arquivo duas vezes? O Emacs apenas move você para dentro do buffer que já existe. Ops!!, abri o arquivo errado, como faço para abrir o arquivo certo? Apenas digite o comando C-x C-v. O emacs irá substituir o buffer do arquivo errado pelo novo arquivo que você irá digitar. Quando você for digitar o nome do arquivo a ser aberto na última linha do emacs, tente digitar somente as iniciais do arquivo e pressionar a tecla TAB para ver o que acontece. O Emacs preenche o restante do nome do arquivo para você. Este é um recurso muito utilizado do Emacs, o recurso do preenchimento, que é utilizado também pelos *shell* atuais.

3.2.5 Inserindo um arquivo dentro de outro

Para inserir um arquivo, posicione o cursor no local onde você deseja iniciar a inserção. Digite o comando C-x i. O emacs colocará um prompt no minibuffer (última linha do emacs) para você digitar o nome do arquivo a ser inserido. É só digitar o nome e este arquivo será inserido.

3.2.6 Salvando Arquivos

Para salvar o arquivo que você está editando apenas digite o comando C-x C-s ou selecione Save Buffer do menu Files.

3.2.7 Deixando o Emacs

Para encerrar uma sessão no Emacs digite o comando C-x C-c ou selecione Exit Emacs do menu Files.

3.2.8 Quando o Emacs trava

Se o Emacs ficar em uma computação infinita (ou simplesmente muito longa) na qual você não quer continuar, você pode parar isso seguramente digitando C-g. Você pode também usar C-g para descartar um argumento numérico ou o começo de um comando que você não quer terminar. Se você digitou um <ESC> sem querer, você pode cancelar isso com um C-g.

3.3 Editando Arquivos

3.3.1 Movendo o cursor

A maneira mais fácil de mover o cursor é clicando o botão esquerdo de seu mouse. (se você tiver um) ou pressionando as teclas de setas no teclado. Existe alguns comandos Emacs para movimentar o cursor:

- C-p - move uma linha para cima
- C-n - move uma linha para baixo
- C-a - move para o início da linha corrente
- C-e - move para o final da linha corrente

3.3.2 Movendo a Tela

Você pode usar a barra de rolagem que fica em algum dos lados do seu Emacs. Se você clicar com o botão direito do mouse, você rola uma tela para cima, se você clicar com o botão esquerdo do mouse na tela de rolagem, você rola uma tela para baixo.

As teclas:

- PageUp - rola uma tela para cima
- PageDown - rola uma tela para baixa
- Home - vai para o inicio do buffer
- Down - vai para o final do buffer
- C-(seta para baixo) - move a tela uma linha para baixo
- C-(seta para cima) - move a tela um linha para cima

3.3.3 Repetindo Comandos

Agora vamos aprender alguns truques eficientes. O Emacs permite que você repita os comandos quantas vezes quiser. Você pode repetir um comando n vezes digitando o comando ESC n comando, onde n significa o número de vezes. Ex: o comando ESC 30 C- n , irá colocar o cursor 30 linhas para baixo.

3.3.4 Deletando Texto

A Tecla DEL apaga o caracter anterior ao cursor. C-d apaga o caracter sob o cursor. ESC-d apaga a próxima palavra. C-k apaga do cursor até o final da linha corrente.

3.3.5 Recuperando o que você deletou

No Emacs aquilo que você apaga vai para uma região chamada *kill region*. Você pode recuperar novamente aquilo que apagou por último digitando o comando C-y. Para recuperar aquilo que foi apagado há muito tempo atrás, digite ESC-y. A cada comando ESC-y, o Emacs recupera os textos que foram apagados e jogados na *kill region*. Um comando ESC-y somente poderá ser dado após um comando C-y. Então C-y recupera a deleção mais recente. Um ESC-y recuperará a segunda deleção mais recente. Outro ESC-y recuperará a terceira deleção mais recente, e assim por diante.

3.3.6 Marcando texto para deletar, mover ou copiar

No Emacs, uma área marcada é chamada de região. Para definir uma região, você usa um ponteiro secundário chamado mark.

Marcando uma região:

1. Mova o cursor para o início da área onde se deseja selecionar.
2. Digite o comando C-SPACE,
3. Mova o cursor para o final da área onde deseja marcar.
4. Digite o comando C-x C-x para alternar entre o início da área marcada e o final da área marcada.

Deletando uma região:

1. Marque a região
2. Digite o comando C-w ou selecione Cut do menu Edit.
3. Se você digitar C-x u, você recuperará aquilo que você apagou

Movendo uma região:

1. Marque a região
2. Digite o comando C-w para apagá-la
3. Mova o cursor para a nova posição onde se deseja mover o texto
4. Digite o comando C-y (yank).

Se você moveu para o local errado, apenas digite o comando C-x u (*undo*) para desfazer a operação e coloque o cursor no novo local a ser inserido e digite o comando C-y.

Copiando uma região:

1. Marque a região
2. Digite o comando ESC-w.
3. Mova o cursor para onde se desejar copiar o texto
4. Digite o comando C-subsection.

3.3.7 Formatando parágrafos

Para formatar parágrafos, apenas escolha o modo de edição que você desejar com ESC-x *nome-do-modo-de-edição*, posicione o cursor no final do parágrafo e digite o comando ESC-q.

3.4 Operações de procura e substituições

Para começar uma procura incremental, digite o comando C-s. Uma pergunta será colocada na área do minibuffer (última linha do emacs), *I-search:*. Digite a palavra que deseja procurar. A cada letra que você digitar o Emacs já vai automaticamente tentando encontrar a palavra no texto. Se

o Emacs não encontrar a palavra que você procura, uma mensagem *Search failed* é mostrada no minibuffer. Para se fazer uma substituição automática logo após uma procura, você deve digitar o comando ESC-%. Aparecerá no minibuffer uma pergunta (*Query replace:*). Você deve digitar a string que deseja procurar (substituir). Quando você pressionar ENTER, aparecerá uma nova pergunta (*with:*) que será a string que você deseja que seja a substituta da antiga. O Emacs perguntará por todas as ocorrências que ele encontrar da string a ser substituída. Se você digitar y (yes), o Emacs substituirá a string para você. Caso contrário digite n (não). Se você não quiser que o Emacs fique perguntando se você deseja substituir ou não a string, apenas digite ! que o Emacs substituirá todas as strings que ele encontrar automaticamente sem perguntar por confirmações. Por default o Emacs não faz distinções entre maiúsculas e minúsculas. Se você mandar procurar random, ele também achará Random, RANdom, RaNdOn, e qualquer variação de letras maiúsculas e minúsculas da palavra random.

3.5 Usando buffers e janelas

Uma das características mais interessantes do Emacs é a capacidade que ele tem de editar múltiplos buffers de uma só vez, e mostrar mais que um buffer utilizando janelas.

Trabalhando com múltiplos buffers Se você desejar criar um buffer que contém um arquivo, simplesmente digite o comando C-x C-f para encontrar o arquivo. A cada comando C-x C-f que você digita um novo buffer é criado. Mas como fazer para mudar entre os buffers existentes? Apenas digite o comando C-x b e o nome do buffer que você deseja visitar. Você também pode escolher o buffer a ser visitado diretamente do menu Buffer.

Deletando Buffers Para apagar um buffer (não salvará as mudanças em disco) digite o comando C-x k ou selecione Kill Current Buffer do menu File.

Buffers somente de leitura Para abrir um buffer no modo somente-leitura digite o comando C-x C-q. Se o comando C-x C-q for digitado novamente o buffer volta a ser um buffer de escrita.

3.5.1 Trabalhando com Janelas

Criando janelas horizontais Para dividir a janela corrente em duas janelas horizontais (uma acima da outra) digite o comando C-x 2.

Criando janelas verticais ou janelas lado a lado Para dividir a janela corrente em duas janelas verticais (uma do lado da outra) digite o comando C-x 3.

Movendo-se entre Janelas Digite o comando C-x o (outra) para mover-se entre as janelas. Se você digitar o comando C-x 1, a Janela corrente ocupará toda a área do Emacs.

3.6 Emacs como um ambiente de trabalho

Muitas das coisas que você faz em uma *shell* Unix, você pode fazer de dentro do Emacs. Você pode executar comandos Unix, trabalhar com diretórios e imprimir arquivos sem deixar o Emacs.

3.6.1 Executando comandos Unix em um Buffer Shell

Uma das características mais importantes do Emacs é a capacidade que ele tem de rodar uma *shell* Unix em um buffer. Depois que você inicializar um *buffer shell*, você pode fazer tudo que uma *shell* Unix faz só que dentro do Emacs. Para rodar um comando enquanto você está em uma sessão Emacs digite o comando ESC-!.

3.6.2 Trabalhando com arquivos e diretórios

Dired, o modo de edição de diretórios, é uma característica interessante do Emacs. Ele provê um meio de edição de diretórios. Existem duas formas de iniciá-lo. Ou você digita emacs <nome_de_um_diretório> ou dentro do emacs digite ESC-x dired.

A saída gerada pelo modo dired é igual ao que você vê quando digita ls -l em um prompt Unix. Para visualizar um arquivo basta colocar o cursor sobre o mesmo e pressionar a tecla ENTER.

Deletando arquivos

Para deletar arquivos, posicione o cursor sobre o arquivo que se deseja apagar e pressione a tecla `d`. Ele será marcado para deleção, mas não será apagado. O arquivo somente será apagado quando você confirmar a deleção pressionando a tecla `x`.

Movendo-se pela árvore de diretórios

Basta posicionar o cursor sobre os nomes dos diretórios ou sobre os `..` e pressionar a tecla `ENTER`.

Editando arquivos

Idêntico ao anterior.

Copiando arquivos

Posicione o cursor sobre a linha onde está o arquivo e digite `C` (copy). O Emacs perguntará então qual o nome do arquivo de destino.

Renomeando arquivos

Posicione o cursor sobre a linha onde está o arquivo e digite `R` (rename). O Emacs perguntará então qual o nome do arquivo de destino.

3.6.3 Imprimindo do Emacs

Imprimindo com cabeçalho e número de páginas

Para imprimir um buffer com páginas numeradas e cabeçalhos para o nome do arquivo, digite o comando `ESC-x print-buffer` ou selecione `Print Buffer` do menu `Tools`.

Imprimindo sem cabeçalhos e número de páginas

Para imprimir um buffer sem páginas numeradas e cabeçalhos, digite o comando `ESC-x lpr-buffer`.

Imprimindo somente uma região do buffer

Primeiro selecione a região (`??`). Depois digite o comando `ESC-x print-region` ou `ESC-x lpr-region` ou selecione `Print Region` do menu `Tools`.

3.6.4 Lendo *Manpages* no Emacs

Você pode ler a documentação online do Unix (ou manpages) de dentro do Emacs apenas digitando o comando ESC-x man ou selecionando Man do menu Help. A vantagem de você ler manpages de dentro do Emacs é que você pode navegar pelas manpages mais facilmente do que na *shell* do Unix.

3.6.5 Horas e Calendário

Para que você possa visualizar horas no minibuffer apenas digite o comando ESC-x display-time que um relógio aparecerá no minibuffer. Se você deseja que toda vez que você entrar no Emacs o relógio apareça, apenas acrescente a linha (display-time) no seu arquivo `.emacs`. Para mostrar calendário digite o comando ESC-x calendar.

3.7 Lendo e-mails e news de dentro do emacs

3.7.1 Trabalhando com e-mails

Vamos falar um pouco de um leitor de e-mails para Emacs chamado RMAIL. Existem outros leitores de e-mails tais como MH e vm, que não detalharemos aqui. O leitor de mail MH tem uma interface para Emacs chamada mh-e. Para mais informações, leia o livro *MH & xmh: Email for Users and Programmers* by Jerry Peek, da O'Reilly & Associates. Dentre outras características, mh-e inclui suporte para MIME, permitindo gravuras, programas, sons, e outros objetos de serem anexados em uma mensagem de mail. O RMAIL não suporta essas características ainda.

3.7.2 Mandando e-mails de dentro do Emacs

Para mandar mail de dentro do Emacs é muito fácil. Basta apenas digitar o comando C-x m que um buffer chamado mail se abrirá para você. Após o campo to: coloque o e-mail do destinatário. Após o campo Subject: coloque o título da mensagem. Escreva sua mensagem abaixo da linha pontilhada. Para enviar a mensagem apenas digite o comando C-c C-c e sua mensagem será enviada. Como você pode verificar, não existe como default o campo CC: (Carbon Copy), para mandar cópias para outros endereços. Para inserir um campo CC você deve digitar o comando C-c C-f C-c para que o Emacs crie um campo CC para você ou escolha Cc no

menu Headers ou ainda acrescente uma linha e digitar CC após o campo To:.

3.7.3 Lendo e-mails de dentro do Emacs

Para ler e-mails de dentro do Emacs digite o comando ESC-x rmail ou selecione Read Mail do menu Tools. Se você digitar h, vão aparecer todos os headers dos mails para que você possa se organizar melhor. Para verificar se existem novos mails digite g (get).

3.7.4 Lendo news com o gnus

Para inicializar o leitor de news de dentro do emacs digite o comando ESC-x gnus.

3.8 Emacs como uma ferramenta da Internet

Para usar telnet de dentro do Emacs, basta você digitar o comando ESC-x telnet.

Capítulo 4

VI/VIM

4.1 Introdução

O *vi* é um editor de textos extremamente simples, que tem como objetivo suprir as necessidades de um usuário quando este precisa editar arquivos, e nada mais. A principal preocupação do *vi* é ser um editor prático.

A maioria dos usuários (principalmente os novatos) não gostam nem um pouco da interface e das características do *vi*, mas após o aprendizado notam que essas mesmas características fazem do *vi* um editor extremamente prático e fácil de usar.

Apresentando uma interface extremamente simples e um modo especial de comandos e edição, o *vi* pode ser utilizado em praticamente qualquer terminal, e isso inclui, por exemplo, uma conexão aos antigos terminais VT100, que tinham largura de banda baixíssima, as atuais conexões por telnet em uma linha discada e conexões em máquinas com arquiteturas e teclados diferentes. (Você não precisa de nenhuma tecla especial para editar um texto no *vi*, bastam as teclas "ESC" e as letras para você fazer praticamente tudo o que precisa).

Com editores cada vez mais complexos e cheios de funções e funcionalidades (como o emacs), o VI se mantém como um dos editores mais utilizados no mundo UNIX devido a um único motivo: Simplicidade.

O *vi* ganhou popularidade e inspirou criadores de outros editores a seguir seu exemplo de simplicidade e, ao mesmo tempo, flexibilidade. Com isso surgiram os "clones" do *vi*, que são editores que mantêm as mesmas características, funcionalidades e simplicidades do *vi*, mas com funções a mais para tarefas específicas.

Este é o caso do *VIM* (VI Improved), criado por Bram Moolenaar.

Embora existam outros clones, e o próprio *vi* (o original), a maioria hoje utiliza o *vim* sem nem ao menos estar ciente disso. Isso por que a maioria

das distribuições atuais distribui o *vim* no lugar no *vi*, e, através de aliases ou links, você acaba acessando o *vim* simplesmente como *vi*.

A característica do *vim* é manter plena compatibilidade com o original, mas apresentar uma série de novas funções e facilidades para o usuário.

4.2 Vantagens do *vim* em relação ao *vi*

Uma série de funções foram adicionadas ao *vim*, incluindo suporte a teclas de locomoção e, principalmente, opções de programação (como sintaxe colorida e indentação). Abaixo temos uma lista (abreviada) com os “melhoramentos” implementados no *vim* em relação ao *vi*:

- Suporte ao teclado do PC;
- Sintaxe colorida para centenas de tipos de arquivo;
- Indentação ”inteligente”;
- Suporte a janelas (vários arquivos abertos ao mesmo tempo);
- Implementação de macros;
- VimInfo (gravação de opções em disco, como por exemplo posição do cursor ao re-editar um arquivo)
- Configurações diversas (principalmente através do uso do `.vimrc`)

Obviamente, todas essas características são opcionais, e, dependendo das opções compiladas e do arquivo de configuração, seu *vim* pode ser idêntico ao *vi* em suas funções.

Não vamos nos concentrar em explicar diferenças de um para outro. Vamos simplesmente explicar os comandos como se você estivesse utilizando o *vim*.

A explicação detalhada sobre a utilização das funções do *vi/vim* como editor de textos simples você encontra na apostila do curso de linux básico.

4.3 Programando com o VIM

As principais funcionalidades que fazem do VIM um excelente editor para programação são:

Suporte a inúmeras linguagens de programação Você pode escolher desde `asmh8300` e `basic` até `php` e `xml`. Enfim, praticamente qualquer linguagem que você puder imaginar.

Cores para sintaxe, Auto-identação e múltiplas janelas O `vim` detecta automaticamente o tipo de arquivo que você está abrindo (através do nome ou cabeçalho deste) e o `colore`, mesmo trabalhando na console texto.

Algumas opções extras você encontra no arquivo de configuração do `vim`, como será visto um pouco abaixo neste mesmo documento.

4.4 Configurando o `vim` (.vimrc)

A principal funcionalidade do `vim` consiste na possibilidade de configurá-lo através do arquivo “`.vimrc`” que deve estar no diretório `home` do usuário. Você encontra um arquivo de exemplo com inúmeras opções no site <http://www.inf.ufpr.br/~ademar/linux>. Descreveremos aqui apenas algumas das opções que você pode utilizar.

O formato de um arquivo `.vimrc` é bastante simples:

Comentários são iniciados por aspas duplas:

```
" Isto é um comentário
```

Comandos e macros são inseridos normalmente no arquivo, sendo que todos podem ser usados normalmente diretamente a partir do `vim`, pressionando-se `<ESC>:comando`.

Abaixo está um exemplo de arquivo `.vimrc` comentado. Esse exemplo contém apenas alguns comandos e macros de exemplo. A utilização de caracteres de escape (como comentado acima) é indispensável para a criação de macros mais eficientes.

```
" #####
```

```
" Exemplo de .vimrc
```

```
"
```

```
" Você pode encontrar um exemplo mais completo (com caracteres de
```

```
" escape) em http://www.inf.ufpr.br/~ademar/linux
```

```
" #####
```

```
"A T A L H O S & A B R E V I A Ç Õ E S
```

```
"=====
```

```
" ignorar erros de digitacao de comandos para sair:
```

```
cab Wq wq
```

```
cab WQ wq
```

```

cab Q q
cab W w

" gera arquivo html a partir do arquivo atual
cab gerahtml so /usr/share/vim/syntax/2html.vim

"L I G A / D E S L I G A
"-----
"sequencia deve ser feita sem pausa no modo comando
"
"      , começo de mapeamento
"      s comando :set
" [a-z] sigla da opção do set
"
"autoIndent
map ,ai :set ai!<cr>:echo "autoIndent="&ai<cr>
"ignoreCase
map ,sc :set ic!<cr>:echo "ignoreCase="&ic<cr>
"Highlightedsearch
map ,sh :set hls!<cr>:echo "Highlightedsearch="&hls<cr>
"Smartindent
map ,si :set si!<cr>:echo "SmartIndent="&si<cr>
"ExpandTAB
map ,et :set et!<cr>:echo "ExpandTAB="&et<cr>
"cindent
map ,ci :set cin!<cr> :echo "CIndent="&cin<cr>
"limpa os highlights (chuncho) :)
map ,hl /lixo - highlights limpos<cr> :echo "Highligh clean"<cr>

" C O M E N T Á R I O S
"-----
"para "ocultar" (F2) e "voltar" (F3) os comentários do arquivo atual
noremap <F2> :hi Comment ctermfg=black guifg=black<cr>
noremap <F3> :hi Comment term=bold ctermfg=cyan guifg=cyan<cr>

"C O N F I G U R A C O E S   D I V E R S A S   ( S E T )
"=====
set bs=0           " backspace normal (estilo VI)
set autoindent     " Autoidentação sempre ligada
set smartindent    " SmartIdentação sempre ligada
set ts=4           " numero de caracteres de avanco do TAB
set textwidth=76   " numero maximo de colunas de texto (quebra de linha)
set report=0       " reporta ações com linhas no rodapé
set shm=filmnrwxt  " encurta as mensagem do rodapé (SHortMessages)
set showcmd        " Mostra (parcialmente) os comandos no rodapé
set showmatch      " Avisa quando parênteses não foram fechados
set ruler          " Mostra número de linha e coluna no rodapé

```

```

set autowrite          " Salva documentos antes de comandos como next e make
set laststatus=0      " Rodapé completo (Barra horizontal)

" opcoes negadas (retire o prefixo "no" para ativar)
set nonumber          " Não exibe coluna com os números de linha
set noet              " Não converte TAB's em espaços (:retab-converte tudo)
set novisualbell     " Emite um beep em vez de piscar a tela
set nocompatible     " Usa um padrão VIM para alguns detalhes

" opcoes de busca
set is                " Busca incrementada
set hls              " Colore o fundo da palavra recém encontrada
set ignorecase       " Ignora maiúsculas/minúsculas em buscas

" Extensoes que tem prioridade menor quando usando TAB pra completar nomes
" de arquivos no VIM.
" Devem ser extensoes de arquivos que não se deseja editar.
set suffixes=.bak,~, .swp, .o, .info, .aux, .log, .dvi, .bbl, .blg, .brf, .cb, .ind
, .idx, .ilg, .inx, .out, .toc

" V I M I N F O
"=====
" guarda posicao do cursor e historico da linha de comando:
set viminfo='50,\"300,:20,%,n~/ .viminfo

" P R O G R A M A C A O   E M   C
"=====
" caracteres para endentação (tamanho do tab)
set shiftwidth=4
" opções para programação em C (consulte :help cinoptions)
set cinoptions=>s,e0,n0,f0,{0,}0,^0,:s,=s,ps,ts,+s,(2s,)25,*30

" E D I C A O   D E   G Z I P
"=====
" A sequência de macros abaixo permite a edição de arquivos compactados
em formato gzip.

augroup gzip
  " Remove todos os autocomandos gzip
  au!

  " Liga a edição de gzip
  "   read: Coloca em modo binário antes de ler arquivo
  "   Descompacta o arquivo antes de exibir
  "   write: Grava o arquivo após a edição
  "   append: Descompacta o arquivo, altera e comprime novamente

```

```

autocmd BufReadPre,FileReadPre *.gz set bin
autocmd BufReadPost,FileReadPost *.gz let ch_save = &ch|set ch=2
autocmd BufReadPost,FileReadPost *.gz '[,']!gunzip
autocmd BufReadPost,FileReadPost *.gz set nobin
autocmd BufReadPost,FileReadPost *.gz let &ch = ch_save|unlet ch_save
autocmd BufReadPost,FileReadPost *.gz execute ":doautocmd BufReadPost "
. expand("%:r")

autocmd BufWritePost,FileWritePost *.gz !mv <afide> <afide>:r
autocmd BufWritePost,FileWritePost *.gz !gzip <afide>:r

autocmd FileAppendPre *.gz !gunzip <afide>
autocmd FileAppendPre *.gz !mv <afide>:r <afide>
autocmd FileAppendPost *.gz !mv <afide> <afide>:r
autocmd FileAppendPost *.gz !gzip <afide>:r
augroup END

```

4.5 Configurando o *vim* como seu editor padrão

A maneira mais fácil de transformar o *vim* em seu editor de textos padrão é setando as duas variáveis de ambiente conforme mostrado abaixo em seu arquivo `.bashrc`.

Geralmente se cria um alias para o *vim* para que ele possa ser chamado simplesmente como “*vi*”, o que é mais cômodo.

Abaixo estão os comandos para os procedimentos citados acima, que podem ser colocados em seu arquivo `.bash_profile`:

```

export EDITOR=vim; # Variável EDITOR - editor de arquivos padrão
export VISUAL=vim; # Variável VISUAL - visualizador de arquivos padrão

alias vi=vim; # Habilita a chamada do vim através do comando "vi"

```

Capítulo 5

Programando em ambiente Linux

5.1 Introdução

Para a leitura desse capítulo, você deve ter um pequeno conhecimento do sistema de arquivos do linux, sua estrutura de diretórios e usuários, além de alguma familiaridade com a sintaxe de comandos. Você também deve saber consultar uma página manual e uma documentação info. Assuma-se também que você sabe utilizar um editor de textos simples, como o vi ou o emacs e, obviamente, tem conhecimentos de programação (os exemplos estão em C).

Se você ainda não conhece bem o sistema linux, é recomendado que, antes de continuar a leitura desse texto, você consulte outras fontes de documentação introdutória.

Primeiramente vamos lembrar que o ato de programar consiste (do ponto de vista prático) basicamente em:

1. Escrever o código
2. Compilar o código
3. Depurar o código

sendo que a segunda etapa pode ser um pouco estendida ao se programar com vários módulos e/ou bibliotecas.

Para quem está acostumado a programar em outros ambientes, como em MS-DOS , essas etapas geralmente estão agrupadas em uma IDE (**I**nterface de **D**esenvolvimento), como o Borland C.

Mas quando programamos em linux, geralmente utilizamos um utilitário para cada etapa. Isso apresenta algumas vantagens como flexibilidade, pois podemos utilizar nossos programas favoritos ou disponíveis para cumprir cada etapa.

5.2 O que é preciso para criar um programa

Embora existam IDEs disponíveis para o ambiente do linux, a maioria dos programadores optam por trabalhar independentemente, escolhendo suas ferramentas favoritas.

Seguindo o raciocínio visto para as etapas da programação, precisamos dominar os seguintes tipos de utilitários:

5.2.1 Editor de textos

Existem vários editores disponíveis no ambiente linux. Alguns apresentam funcionalidades extras para programadores (como endentação automática do código, apresentação do código com diferenciação de cores mediante sintaxe, verificação de erros na sintaxe, etc).

Os dois editores mais utilizados são o vim e o emacs, mas o importante é que você utilize o editor de sua preferência.

5.2.2 Compilador

A tarefa do compilador é a mais importante. Existem compiladores para as mais diversas linguagens no linux. Obviamente, você deve escolher aquele que melhor suprir suas necessidades.

Os dois compiladores mais utilizados são o gcc e o g++ (respectivamente, GNU C Compiler e GNU C++ Compiler).

É importante você conhecer as *flags* de compilação de cada um e as opções mais usadas.

As flags mais importantes são:

Warnings Elas ajudam a encontrar falhas na sintaxe e lógica do programa, mas que não impedem a compilação do mesmo.

Indispensável principalmente se você programa em C, pois essa linguagem é bastante flexível no que diz respeito à compilação, e permite que um código muito mal escrito seja compilado sem apresentar erros.

Otimização Permitem que o compilador utilize instruções otimizadas ou específicas de um determinado processador.

Para descobrir quais as flags de seu compilador, consulte sua documentação.

5.2.3 Depurador

Esse é o utilitário que você utilizará quando as coisas não funcionarem da primeira vez, ou seja, sempre...

O depurador mais utilizado no ambiente linux é o gdb (GNU DeBugger). Ele é um poderoso e excelente depurador para as linguagens C e C++.

5.2.4 Outros

Existem diversos outros utilitários que auxiliam muito o programador no ambiente linux.

Um dos mais indispensáveis e úteis é o *make*. Ele permite otimizar e automatizar várias etapas da compilação, facilitando muito essa tarefa.

Além deste, existem também bibliotecas para depuração, utilitários gráficos, utilitários para layout, patches, etc.

Abaixo uma lista do que você pode achar interessante: (procure documentação a respeito nas páginas manuais)

- make;
- EletricFence;
- checker;
- indent;
- diff;
- patch;

Resumindo, você precisa dominar pelo menos quatro utilitários no total: editor de textos, make, compilador e depurador. Mas não pense que isso faz do linux um ambiente difícil para a programação. Como foi citado a pouco, existem IDEs muito práticas no linux. Mas como quase toda a interface, elas são apenas *frontends* para os utilitários citados.

5.3 Os utilitários

Agora, veremos de forma mais detalhada os utilitários mais importantes acima citados.

Como já foi dito, você deve utilizar o editor de textos que você mais gosta e que você se sinta confortável para digitar seu o código. Portanto não entraremos em detalhes quanto a este aspecto.

5.3.1 gcc - O Compilador C/C++

Sem dúvida o compilador mais utilizado e importante do sistema linux. Embora existam outros compiladores para a linguagem C disponíveis, o gcc é o mais famoso. Ele foi criado e é mantido pela comunidade GNU, e está em um estágio que permite considerá-lo um dos melhores compiladores existentes.

Após criar um arquivo com o código corretamente escrito, você deve utilizar o gcc da seguinte forma:

```
$ gcc fonte.c
```

O resultado é a criação de um arquivo executável de nome *a.out* que é resultado da compilação do código contido no arquivo *fonte.c*

Se você quer que o gcc crie um arquivo executável com um nome específico (e geralmente você quer), você deve utilizar a *flag* `-o` que significa *output* (saída). Ela é utilizada antes do nome do arquivo que será criado.

Abaixo refaremos a compilação do exemplo anterior, mas criando um arquivo de nome “executável”:

```
$ gcc fonte.c -o executavel
```

Para compilar programas que estejam em vários arquivos, você deve primeiramente compilar os módulos como objetos e então compilar o programa principal.

Vamos ver como exemplo o caso da compilação de um programa que esteja separado em três partes (arquivos):

```
interface.c  
processamento.c  
principal.c
```

A sequência para a criação de nosso programa, que chamaremos de “teste” seria:

```
$ gcc -c interface.c -o interface.o  
$ gcc -c processamento.c -o processamento.o  
$ gcc interface.o processamento.o principal.c -o teste
```

A *flag* `-o` * já é por nós conhecida. A novidade fica por conta da *flag* `-c` (compilar mas não linkar), que é utilizada para a criação do arquivo-objeto.

*Neste caso a *flag* poderia ter sido suprimida e obter-se-ia o mesmo resultado, pois o *default* da *flag* `-o` é criar um arquivo com o mesmo nome do arquivo-fonte mas com extensão *.o* ao invés de *.c*

Esse processo se torna cansativo e demais trabalhoso para ser feito sem uma automação. Para essa tarefa veremos o utilitário *make* mais adiante.

Outras *flags* bastante importantes são as de *warnings* (alertas). São elas que vão detectar possíveis falhas no código que vão desde falta de estilo até problemas com passagem de parâmetros e, principalmente, ponteiros.

As mais utilizadas e recomendadas são:

```
-pedantic -Wall -W -Wtraditional -Wshadow -Wpointer-arith
-Wbad-function-cast -Wcast-qual -Wcast-align -Wwrite-strings
-Wconversion -Waggregate-return -Wmissing-prototypes
-Wmissing-declarations -Wnested-externs -Winline
-Wwrite-strings
```

A explicação de cada uma dessas opções é muito extensa para ser descrita aqui. Por isso é altamente recomendado que você faça uma consulta à documentação do gcc (páginas de manual e info).

Obviamente, algumas dessas opções podem se tornar incômodas na criação de programas que, por um motivo ou outro, precisam quebrar certas regras para chegar a algum resultado. Aliás, essa é uma das maiores características da linguagem C: flexibilidade, e você deve ter muito cuidado ao combinar flexibilidade com funcionalidade.

5.3.2 gdb - Depurador C/C++

Existem vários depuradores disponíveis para o ambiente do linux, mas o mais utilizado e suportado com certeza é o gdb. Ele apresenta um ambiente simples e recursos bastante avançados.

Nosso objetivo é apresentar a utilização básica do gdb. Uma documentação mais completa você encontra nas páginas de manual e info.

Executar o gdb é simples. Primeiramente, você deve compilar seu programa (e módulos objeto) com a *flag* `-g` ou uma das equivalentes, que incluirá no programa executável informações necessárias à depuração.

Ao executar o gdb, é bom que você tenha acesso ao código fonte de seu programa para ir conferindo a execução.

Você deve executar o gdb como abaixo:

```
$ gdb arquivo_executável
```

A interface do gdb é simples:

```

GNU gdb 4.17.0.12 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-Debian/GNU-linux"...
(gdb)

```

Agora você só precisa utilizar os comandos necessários para acompanhar a execução do código, depurando-o.

A técnica básica para depuração é:

- Inserir um *breakpoint* no ponto onde você deseja iniciar a depuração (linha N do código);
- Executar o programa (dentro do gdb);
- Avançar na execução, conferindo os parâmetros passados às funções e o tipo e conteúdo das variáveis.

A sintaxe dos comandos é simples: você tem o comando por extenso e o comando simplificado (geralmente a primeira letra do mesmo). Os parâmetros (como números de linha e nomes de variáveis) são passados imediatamente após o comando.

Abaixo estão os comandos mais importantes e úteis:

<i>Comando</i>	<i>Descrição</i>	<i>Simplificado</i>
break N	Inserir breakpoint na linha N	b N
delete	Remove todos os breakpoints	d
run <par>	Executa o programa até o primeiro breakpoint ou o fim do mesmo	r <par>
next N	Executa as próximas N linhas	n N
step N	Executa os próximos N passos	s N
kill	Mata (encerra) o programa	k
what <var>	Exibe informações sobre o tipo de uma variável	w <var>
print <var>	Exibe o conteúdo de uma variável	p <var>
backtrace	Visualiza a pilha (stack)	bt
help <cmd>	Obtem ajuda em relação a algum comando	h <cmd>
quit	Sai do gdb	q

A sintaxe em relação às variáveis é a utilizada na linguagem C. O último comando pode ser repetido pressionando-se <ENTER>.

Vamos agora conferir um exemplo bem simples em linguagem C (se você tiver dificuldades em entender o código, então você precisará estudar um pouco mais de C).

O programa a seguir tem como único objetivo o aprendizado do gdb. Ele não tem nenhuma utilidade ou objetivo lógico.

```
#include <stdlib.h>
#include <stdio.h>

#define MAGICO 16

void funcao1(char **c);
int funcao2(int x);

int main (void)
{
    int a = 100;
    char b;
    char *c;

    c = (char *) malloc(sizeof(char) * (MAGICO));
    funcao1(&c);
    a = funcao2(a);
    printf("%s\n", c);
    exit(0);
}

void funcao1(char **c)
{
    int i;
    for (i = 0; i < MAGICO; i++)
        *c[i] = 'a';
}

int funcao2(int x)
{
    return x * MAGICO + 50;
}
```

Agora, compilando e depurando:

```
$ gcc exemplo.c -o exemplo -g
$ ./exemplo
Segmentation fault (core dumped)
$ gdb exemplo
GNU gdb 4.17.0.12 with Linux support
```

Copyright 1998 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i386-Debian GNU/Linux"...

Core was generated by './exemplo'.

Program terminated with signal 11, Segmentation fault.

#0 0x80484b0 in funcao1 (c=0xbffff3bc) at exemplo.c:30

30 *c[i] = 'a';

(gdb) print i

\$1 = 4

(gdb)

Como você pôde ver, o gdb, a partir do arquivo de core gerado juntamente com o *segmentation fault*, já detecta em que linha ocorreu o erro, e você pode executar comandos como "print i". Isso muitas vezes já é o suficiente. Mas vamos executar o programa passo a passo assim mesmo:

(gdb) b 1

Breakpoint 1 at 0x8048430: file exemplo.c, line 1.

(gdb) r

Starting program: /home/fulano/prog/exemplo

Breakpoint 1, main () at exemplo.c:10

10 {

(gdb) next

main () at exemplo.c:11

11 int a = 100;

(gdb) next

15 c = (char *) malloc(sizeof(char) * (MAGICO + 1));

(gdb) next

17 funcao1(&c);

(gdb) step

funcao1 (c=0xbffff39c) at exemplo.c:29

29 for (i = 0; i < MAGICO; i++)

(gdb) print *c

\$2 = 0x8049640 ""

(gdb) next

30 *c[i] = 'a';

(gdb) print i

\$4 = 0

(gdb) next 8

30 *c[i] = 'a';

(gdb) print i

\$15 = 4

(gdb) print *c

\$18 = 0x8049640 "a"

Tem algo errado aqui. O comando *print* seguido de uma variável que seja um vetor de caracteres (string), imprime todo o seu conteúdo, desde o primeiro caracter até o `'\0'`. O que vemos aqui é que o vetor tem apenas um caracter.

Pensando um pouco, conseguimos notar que estamos acessando o vetor de maneira errada. O correto seria `(*c)[i]`, e não `*c[i]`.

Agora vamos executar o programa (corrigido) até seu término dentro do gdb, utilizando os comandos simplificados (apenas a primeira letra):

```
$ gcc exemplo.c -o exemplo -g
$ ./exemplo
aaaaaaaaaaaaaaaaa
$ gdb exemplo
[...]
(gdb) b 1
Breakpoint 1 at 0x8048430: file exemplo.c, line 1.
(gdb) r
Starting program: /home/ademar/prog/exemplo

Breakpoint 1, main () at exemplo.c:10
10      {
(gdb) n
main () at exemplo.c:11
11          int a = 100;
(gdb) n
15          c = (char *) malloc(sizeof(char) * (MAGICO + 1));
(gdb) n
17          funcao1(&c);
(gdb) s
funcao1 (c=0xbffff39c) at exemplo.c:29
29          for (i = 0; i < MAGICO; i++)
(gdb) n
30              (*c)[i] = 'a';
(gdb) n 30
30              (*c)[i] = 'a';
(gdb) p i
$1 = 15
(gdb) p *c
$2 = 0x8049640 'a' <repeats 15 times>
```

Agora sim. O vetor “c” contém 15 vezes a letra ‘a’.

```
(gdb) n
29          for (i = 0; i < MAGICO; i++)
(gdb) n
31      }
(gdb) n
main () at exemplo.c:18
```

```

18             a = funcao2(a);
(gdb) p a
$3 = 100
(gdb) n
20             printf("%s\n", c);
(gdb) p a
$4 = 1650
(gdb) n
aaaaaaaaaaaaaaaaaa
22             exit(0);
(gdb) n

Program exited normally.
(gdb)

```

Esse é o fim da execução.

Este exemplo é bastante simples, mas se você teve dificuldades em acompanhar, crie um programa mais simples ainda e faça os testes. Você verá grande utilidade no gdb principalmente quando estiver trabalhando com estruturas e ponteiros, pois com variáveis desse tipo é praticamente impossível depurar usando-se *printf*.

5.3.3 make

A utilização do make é bastante simples e prática. O make é extremamente prático quando estamos trabalhando em um projeto grande (com vários módulos a compilar) ou quando usamos várias opções de compilação.

Basicamente, o make é um automatizador de comandos. Ele pode ser utilizado para agrupar um conjunto de comandos qualquer em um rótulo que chamamos logo após o comando make.

Para utilizar o make, você deve criar um arquivo com os rótulos e os comandos que o make deve executar. Esse arquivo deve chamar-se Makefile ou makefile (o formato e utilização dos arquivos é a mesma, e quando precisarmos citá-los, vamos citar apenas como Makefile).

O formato de um arquivo Makefile simples é o seguinte:

```

# comentário
rótulo: dependências
<TABULAÇÃO>comando1; \
            comando2; \
            ...; \
            comandon;

```

Onde:

comentário É um texto ignorado, serve para documentar o arquivo;

rótulo Em compilações, deve ser o nome do arquivo resultante da utilização dos comandos (geralmente resultado da compilação);

dependências Nome dos arquivos e/ou rótulos que devem ser verificados antes da execução dos comandos (veja explicação abaixo);

<**TABULAÇÃO**> É uma tabulação comum. Não deve conter espaços, apenas uma tabulação. Esse é o principal erro ao criar-se um arquivo Makefile.

comandos A lista de comandos que devem ser executados pelo make para o respectivo rótulo. Ao final de cada linha, deve-se inserir uma barra inversa “\”, para continuar a lista (ou até mesmo um comando único) na linha logo abaixo.

É comum especificarmos algumas variáveis locais ao arquivo Makefile para torná-lo mais legível. A utilização de variáveis é simples, como mostrado abaixo:

Atribuição:

```
VARIAVEL=valor
```

e para referenciar a variável:

```
$VARIABLE
```

Uma das grandes vantagens em se utilizar o make é a verificação das dependências. O funcionamento é simples: antes de executar os comandos de um determinado rótulo, o make verifica se os rótulos listados na lista de dependências já foram executados. Para isso, ele verifica se o arquivo rótulo (que, como citado acima, deve ser o nome do arquivo resultante da compilação) é mais recente que suas dependências.

Essa é uma forma de evitar a recompilação de módulos que não precisem ser compilados, e sempre deve ser usada, pois otimiza em muito esse processo.

Abaixo temos um exemplo de arquivo Makefile comentado, que tem como objetivo criar um programa de nome teste, utilizando uma série de flags e opções para o gcc (através da utilização de variáveis).

```
#####
# Exemplo de arquivo Makefile
# PET Informática - UFPR
#####
# Compilador usado (C Compiler)
CC=gcc
```

```
# Opcoes diversas para o gcc
# Para uma explicação detalhada sobre a utilidade de cada opção,
# veja a documentação do gcc (man / info)

OVERALL_OPTIONS=-pipe

DIALECT_OPTIONS=-ansi

# Destaque para as opções de warning abaixo. Elas são extremamente úteis na
# detecção de erros.
WARN_OPTIONS=-pedantic -Wall -W -Wtraditional -Wshadow -Wpointer-arith\
             -Wbad-function-cast -Wcast-qual -Wcast-align -Wwrite-strings\
             -Wconversion -Waggregate-return\
             -Wmissing-prototypes -Wmissing-declarations\
             -Wnested-externs -Winline -Wwrite-strings

DEBUG_OPTIONS=-ggdb3

LD_FLAGS=-lm

# junta todas as opcoes de compilacao em uma variavel só (CFLAGS)
CFLAGS=$(OVERALL_OPTIONS) $(DIALECT_OPTIONS) $(WARN_OPTIONS) $(DEBUG_OPTIONS)\
       $(CODE_GENERATION_OPTIONS)

# all - usado pra simplificar, permitindo o uso do comando make sem nenhum
# parâmetro (rótulo)
all: teste

# programa principal
teste: modulo.o teste.c
       $CC teste.c modulo.o -o teste $(CFLAGS)

# compilacao de um modulo
modulo.o: modulo.c
       $CC -c modulo.c -o modulo.o $(CFLAGS)

# excluir tudo o que foi compilado
clean:
       rm *.o teste -f
```

Capítulo 6

AWK

6.1 Introdução

AWK é um interpretador de comandos que implementa uma linguagem de programação específica para processamento de textos, a linguagem AWK. Foi criado nos laboratórios Bell na década de 70 por Aho, Kernighan e Weinberger. Esse interpretador de comandos está disponível em praticamente todos os sistemas UNIX existentes, inclusive no Linux.

A linguagem AWK é comumente utilizada para:

- gerenciar pequenos bancos de dados pessoais;
- gerar relatórios;
- validar dados;
- produzir índices em documentos;
- testar algoritmos para serem implementados em outra linguagem de programação.

6.1.1 Utilização

Há duas formas de se utilizar o AWK:

Formato 1:

```
awk 'PROGRAMA' <ARQUIVO DE ENTRADA>
```

onde:

PROGRAMA : são os comandos que serão executados em cada linha do arquivo de entrada. Este formato é utilizado quando a tarefa a ser executada é pequena e simples.

ARQUIVO DE ENTRADA : arquivo contendo os dados de entrada.

Nessa forma de uso, o arquivo de entrada geralmente é o redirecionamento da saída de algum programa ou algum arquivo com dados bem organizados, como por exemplo o `/etc/hosts`.

```
awk '/host/' /etc/hosts
```

O exemplo acima imprime as linhas em que ocorre a palavra “host” no arquivo `/etc/hosts`

Formato 2:

```
awk -f <ARQUIVO-FONTE> <ARQUIVO DE ENTRADA>
```

onde:

ARQUIVO-FONTE : arquivo contendo as regras.

Este formato é utilizado quando o programa é grande demais para ser colocado em apenas uma linha. Este é também o uso mais comum do awk.

```
$ cat > hello.awk
BEGIN { print "Oi Mundo!" }
^D
$ awk -f hello.awk
Oi Mundo!
$
```

No exemplo acima, o AWK interpreta o comando contido no arquivo `hello.awk` e imprime a mensagem “Oi Mundo!” na tela.

Obs.: BEGIN é uma palavra reservada do AWK e significa o início do arquivo.

AWK faz a leitura dos arquivos de entrada linha por linha, e em cada linha ele procura identificar *campos*. Cada linha é chamada de *registro*. A definição de registro aqui é: uma seqüência de caracteres seccionada em campos e terminada por um separador de registros.

Para seccionar o registro em campos, o AWK usa uma variável de ambiente, FS (**F**ield **S**eparator), o *separador de campos*. Por padrão, o valor dessa variável é branco (espaço em branco ou `\t`), mas pode ser redefinido pelo usuário.

O *separador de registros* é definido na variável de ambiente RS (**R**ecord **S**eparator). O valor dessa variável é `\n`, por padrão, mas pode ser redefinido pelo usuário.

6.2 Exemplos de scripts AWK

Nesta seção serão mostrados alguns exemplos de scripts úteis em AWK, com algumas explicações sobre seu funcionamento.

6.2.1 Alguns exemplos simples

O script abaixo — que faz a mesma coisa que o exemplo da seção anterior — tem uma diretiva especial na primeira linha que chama o programa `/usr/bin/awk` para executar o próprio arquivo. Esta diretiva é interpretada pelo *shell* (praticamente todos os *shells* do Unix a reconhecem).

```
$ cat > hello.awk
#!/usr/bin/awk -f
BEGIN { print "Oi Mundo!" }
^D
$ chmod a+x hello.awk
$ ./hello.awk
Oi Mundo!
$
```

Exemplos comentados

Exemplo com uma regra:

```
$ awk '/MASK/ { print $0 }' /etc/init.d/network
NETMASK=255.255.255.0
ifconfig eth0 ${IPADDR} netmask ${NETMASK} broadcast ${BROADCAST}
ifconfig eth1 ${IPADDR} netmask ${NETMASK} broadcast ${BROADCAST}
$
```

Imprime todas as linhas em que ocorre o padrão “MASK” no arquivo `network`. Este padrão pode ser uma expressão regular.

Note que um dos exemplos anteriores fazia a mesma coisa. A questão é que uma instrução do AWK pode omitir o padrão sendo procurado ou o comando a ser executado. O comando padrão é imprimir todas as linhas em que ocorre o padrão. Quando nenhum padrão é fornecido, a ação é executada sobre todas as linhas. Esses padrões serão melhor explicados adiante.

Exemplo com duas regras:

```
$ cat > duasregras.awk
/route/ { print $0 }
```

```

/GATEWAY/ { print $0 }
$ awk -f duasregras.awk /etc/init.d/network
route add -net 127.0.0.0 netmask 255.0.0.0 dev lo
GATEWAY=200.201.6.1
#route add -net ${NETWORK}
["${GATEWAY}"] && route add default gw ${GATEWAY} metric 1
["${GATEWAY}"] && route add default gw ${GATEWAY} metric 1
route add 200.201.6.32 dev eth1
$

```

Quando um programa possui mais de uma regra, cada linha do arquivo de entrada é submetida a cada uma das regras.

Note que toda linha em que ocorrem os dois padrões, *route* e *GATEWAY*, é impressa duas vezes, já que todas as regras são aplicadas a cada linha.

O exemplo abaixo é um pouco mais complexo e dá uma idéia um pouco melhor da utilidade do AWK. A grande parte dos scripts AWK se destina a tarefas como a feita abaixo.

```

$ ls -lg /etc | awk '$6 == "May" { sum += $5 } \
> END { print sum }'
72100
$

```

Este comando imprime a soma dos números de bytes dos arquivos que foram modificados em maio (May) de qualquer ano. Note aqui o uso de uma variável, *sum*. Aqui está ilustrada uma importante característica do AWK: para utilizar uma variável, não é necessário declará-la previamente; esta existe a partir do momento em que é usada.

Neste exemplo vemos pela primeira vez o conceito de campo. O \$6 indica o sexto campo dentro do registro, ou seja, a sexta coluna, delimitada por um branco (espaço ou marca de tabulação), que na saída do comando “*ls -lg*” corresponde à coluna do mês. O \$5 indica a coluna correspondente ao tamanho do arquivo em bytes.

Veja algumas linhas da saída do comando “*ls -lg /etc*”:

```

drwxr-xr-x 10 root root 1024 Feb 10 19:03 GNUstep
-rw-r--r-- 1 root root 6254 Jan 5 15:58 Muttrc
drwxr-xr-x 22 root root 1024 May 13 14:13 X11

```

O programa imprime a soma dos bytes apenas no final do arquivo por causa da palavra reservada END, que representa a última linha do arquivo, ou seja, depois que a última linha for processada, o programa imprime a soma do número de bytes, calculada anteriormente.

6.2.2 Programas úteis em uma linha

Alguns dos comandos utilizados nos exemplos abaixo serão explicados somente depois. Por enquanto estes exemplos servem apenas como ilustração. Os dados utilizados nos exemplos estão gravados em um arquivo de nome "data".

Imprime o comprimento da linha mais longa.

```
$ awk '{ if (length($0) > max) max = length($0) }\
> END { print max }' data
```

Imprime todas as linhas com mais de 80 caracteres (contando espaços).

```
$ awk 'length($0) > 80' data
```

Imprime o comprimento da linha mais longa do arquivo data. O programa expand faz uma conversão das marcas de tabulação em espaços para que sejam contados como caracteres.

```
$ expand data | awk '{ if (x < length()) x = length() }\
> END { print "maximum line length is " x }'
```

Imprime toda linha que possua no mínimo um token, ou seja, remove todas as linhas em branco de um arquivo.

```
$ awk 'NF > 0' data
```

Este programa imprime 7 números aleatórios entre 0 e 100.

```
$ awk 'BEGIN { for (i = 1; i <= 7; i++)\
> print(101 * rand()) }'
```

Este programa imprime o total de bytes utilizados pelos arquivos listados pelo programa "ls -lg".

```
$ ls -lg FILES|awk '{x+=$5}; END{print "total bytes:" x}'
```

Este programa imprime o número de Kbytes utilizados pelos arquivos listados.

```
$ ls -lg FILES | awk '{ x += $5 } \
> END { print "total K-bytes: " (x + 1023)/1024 }'
```

Este programa imprime uma lista alfabética com os identificadores de login de todos os usuários. A opção “-F:” indica que o separador é um : (dois-pontos).

```
$ awk -F: '{ print $1 }' /etc/passwd | sort
```

Este programa imprime o número de linhas do arquivo.

```
$ awk 'END { print NR }' data
```

Este programa imprime as linhas de número par. Para imprimir as linhas de número ímpar seria necessário utilizar o teste ‘NR%2 == 1’.

```
$ awk 'NR % 2 == 0' data
```

6.3 A sintaxe do AWK

Desta seção em diante exploraremos os recônditos da sintaxe do AWK, que é extremamente compacta e poderosa. A princípio pode parecer complicada, mas com o tempo e prática de uso as dificuldades desaparecem. Quem já está acostumado com a sintaxe compacta da linguagem C não terá dificuldades em se adaptar à do AWK.

6.3.1 Introdução

Um programa AWK é uma seqüência de pares *padrão* {*ação*} e definições de funções de usuário.

6.3.2 Padrões

Um *padrão* pode ser:

- BEGIN
- END
- *expressão*
- *expressão1* , *expressão2*

Um dos dois, ou *padrão* ou *{ação}* pode ser omitido, mas nunca ambos. Se *padrão* for omitido, a ação será executada em todas as linhas da entrada. Se *{ação}* for omitida, o comando `{ print $0 }` será executado implicitamente.

padrão pode ser representado como uma expressão regular ou como uma expressão qualquer. Expressões regulares serão explicadas mais adiante, e, como o nome já diz, definem uma regra que permite identificar uma classe de palavras.

Os padrões BEGIN e END

O padrão BEGIN significa o início do arquivo. Os comandos associados a ele serão executados antes que a primeira linha da entrada seja lida. Já o padrão END representa o fim do arquivo, ou seja, os comandos associados a ele serão executados somente depois que a última linha da entrada for processada. Ambos os padrões exigem, obrigatoriamente, uma ação associada, ou seja, não é possível que a eles não seja associada nenhuma ação.

Estes padrões podem ser utilizados mais de uma vez, sendo que serão executados na ordem em que foram colocados no programa. Aconselha-se, por questões de legibilidade, que estes padrões sejam colocados no início e no final do arquivo.

O padrão EMPTY

EMPTY é o padrão vazio, que é encontrado em quaisquer linhas da entrada.

Intervalos de padrões

É possível se especificar intervalos delimitados por dois padrões, no seguinte formato:

```
'BEGPAT, ENDPAT'
```

onde BEGPAT indica o padrão inicial e ENDPAT o padrão final. A ação associada será executada em todas as linhas da entrada que se encontrarem entre os dois padrões.

Exemplo:

```
$ awk '$1 == "192.168.1.1", $1 == "192.168.1.3"' /etc/hosts
192.168.1.1 besouro.dcc.unioeste-foz.br besouro
192.168.1.2 ladybug.dcc.unioeste-foz.br ladybug
192.168.1.3 joaninha.dcc.unioeste-foz.br joaninha
besouro:/mnt/floppy$
```

O exemplo acima imprime todos os registros que estiverem entre a ocorrência do primeiro padrão (192.168.1.1) até a ocorrência do segundo padrão (192.168.1.3).

6.3.3 Expressões regulares

O AWK trabalha com expressões regulares com muita facilidade e pode-se fazer combinações de expressões regulares com comandos do AWK. Por exemplo, a expressão do awk:

```
'$0 ~ /PATTERN/'
```

compara a linha corrente inteira com o padrão especificado em PATTERN.

Construindo expressões regulares

Normalmente, em programas AWK registros, campos e strings são comparados com uma expressão regular. Expressões regulares são contidas entre barras, e a instrução

```
expr ~ /r/
```

tem resultado 1 se *expr* “casa” com *r*. Se trocarmos o `~` por seu negado, `!`, invertemos a comparação, ou seja, queremos saber se *expr* não casa com *r*. Como com os pares padrão-ação, temos que

```
/r/ { ação }
```

e

```
$0 ~ /r/ { ação }
```

são equivalentes.

Os caracteres-curinga das expressões regulares do AWK são as seguintes:

`^ $. [] | () * + ?`

Regras para construir expressões regulares

Para construirmos expressões regulares, usamos caracteres, e devemos seguir algumas regras, listadas abaixo.

`c` : casa com um caracter qualquer que não é curinga;

`\c` : casa com um caracter que é seqüência de escape de strings ou literalmente o caracter-curinga `c`;

`.` : casa com qualquer caracter. incluindo `\n`;

`^` : casa com o início da string;

`$` : casa com o final da string;

`[c1c2c3...]` : casa com qualquer caracter da classe `c1c2c3...`;

`[c1-c2]` : casa com qualquer caracter da faixa que vai de `c1` até `c2`;

`[^c1c2c3...]` : casa com qualquer caracter que não esteja na classe `c1c2c3...`;

`r1r2` : casa com `r1` seguida de `r2` (concatenação), em que `r1` e `r2` são expressões regulares;

`r1—r2` : casa com ou `r1` ou `r2` (alternância);

`r*` : casa com `r` repetida zero ou mais vezes;

`r+` : casa com `r` repetida uma ou mais vezes;

`r?` : casa com `r` zero ou uma vez;

`(r)` : casa com `r`, provendo agrupamento.

Vamos esclarecer com exemplos.

O exemplo abaixo sintetiza na primeira linha três comparações, pois qualquer das três palavras: *awk*, *sed* ou *grep* faz com que o contador *filtros* seja incrementado.

```
/awk|sed|grep/ { filtros++ }
END { print filtros, "filtros encontrados" }
```

Já o exemplo abaixo mostra a comparação de um padrão exato dentro do arquivo `/etc/hosts`. Se o primeiro campo (`$1`) for igual a `200.201.6.25`, o programa imprime o segundo campo, `$2`.

```
$ awk '$1 == "200.201.6.25" { print $2 }' /etc/hosts
peroba.dcc.unioeste-foz.br
```

Este exemplo lista separadamente as máquinas cujo endereço IP iniciem com `200`.

```
$ awk '$1 ~ /200.*.*./ { print $2 }' /etc/hosts
peroba.dcc.unioeste-foz.br
.dcc.unioeste-foz.br
palmeira.dcc.unioeste-foz.br
palmito.dcc.unioeste-foz.br
```

Veja abaixo o conteúdo do arquivo original:

```
127.0.0.1localhost
192.168.1.1 besouro.dcc.unioeste-foz.br besouro
192.168.1.2 ladybug.dcc.unioeste-foz.br ladybug
192.168.1.3 joaninha.dcc.unioest-foz.br joaninha
200.201.6.25 peroba.dcc.unioeste-foz.br peroba
200.201.6.26 pinky.dcc.unioeste-foz.br pinky
200.201.6.21 palmeira.dcc.unioeste-foz.br palmeira
200.201.6.33 palmito.dcc.unioeste-foz.br palmito
```

Mais um exemplo, que imprime toda linha em que ocorrem os padrões `200` e `33`:

```
$ awk '/200/ && /33/' /etc/hosts
200.201.6.33 palmito.dcc.unioeste-foz.br palmito
```

O exemplo abaixo imprime as linhas que ocorrem os padrões `200` ou `127`.

```
$ awk '/200/ || /127/' /etc/hosts
127.0.0.1localhost
200.201.6.25 peroba.dcc.unioeste-foz.br peroba
200.201.6.26 pinky.dcc.unioeste-foz.br pinky
200.201.6.21 palmeira.dcc.unioeste-foz.br palmeira
200.201.6.33 palmito.dcc.unioeste-foz.br palmito
```

Este imprime as linhas em que não ocorre o padrão `200`.

```
$ awk '! /200/' /etc/hosts
127.0.0.1localhost
192.168.1.1 besouro.dcc.unioeste-foz.br besouro
192.168.1.2 ladybug.dcc.unioeste-foz.br ladybug
192.168.1.3 joaninha.dcc.unioeste-foz.br joaninha
```

6.3.4 Expressões de controle

Expressões de controle, podem ser utilizadas em AWK e têm uma sintaxe semelhante à da linguagem C.

As expressões de controle são:

if-else : executa se a expressão for verdadeira;

while : repete enquanto a expressão for verdadeira;

do-while : faz enquanto a expressão for verdadeira;

for : executa várias vezes;

break :termina a expressão de repetição (do, while, for);

continue : volta ao início da expressão de repetição;

next : para o processamento do token corrente;

nextfile : para o processamento do arquivo corrente;

exit : para a execução do AWK;

if-else

Sintaxe:

```
if (CONDIÇÃO) CORPO-THEN [else CORPO-ELSE]
```

Se **CONDIÇÃO** for verdadeira, então execute o(s) comando(s) contidos em **CORPO-THEN**; senão, execute o(s) comando(s) contidos em **CORPO-ELSE**.

Uma condição é considerada falsa quando o seu valor é zero ou uma seqüência nula de caracteres. Em todos os outros casos ela é considerada verdadeira, *à la* C.

Exemplo:

```
if (x % 2 == 0)
    print "x é par";
else
    print "x é ímpar";
```

while

while é um comando que diz que o programa deve executar um trecho do código várias vezes, enquanto uma determinada condição for verdadeira.

Sintaxe:

```
while (CONDIÇÃO)
COMANDOS
```

Enquanto CONDIÇÃO for verdadeira, execute COMANDOS. A cada vez que todos os comandos do escopo do em *while* são executados, o teste de CONDIÇÃO é reavaliado.

Exemplo:

```
awk '{
    i = 1
    while (i <= 3) {
        print $i;
    }
}' /etc/hosts
```

Imprime os três primeiros tokens em cada linha (um por linha).

do-while

Esta expressão é uma variação do *while*. A diferença é que o teste é feito ao final do laço, o que garante que o código será executado pelo menos uma vez, independentemente do valor da expressão.

Sintaxe:

```
do
COMANDOS
while (CONDIÇÃO)
```

onde CONDIÇÃO é o teste a ser feito para encerrar o laço. O código em COMANDOS será executado pelo menos uma vez, e continuará sendo enquanto essa condição for verdadeira.

Exemplo:

```
awk 'BEGIN { i = 1 }
{
    do {
        print $0;
```

```

        i++;
    } while (i <= 10);
}'

```

O programa acima imprime as dez primeiras linhas e todas as linhas seguintes, pois o teste é feito apenas no final do comando. Mesmo que *i* seja maior ou igual a 10, o comando será executado.

for

Este comando é utilizado para se executar um número determinado de iterações de comandos.

Sintaxe:

```

for (INICIALIZAÇÃO; CONDIÇÃO; INCREMENTO)
COMANDOS

```

O campo **INICIALIZAÇÃO** é executado antes de qualquer comando em **COMANDOS** e serve para colocar o valor inicial na variável de controle do *for*. O campo **CONDIÇÃO** é utilizado para representar os testes que devem ser executados antes de cada execução dos comandos. É importante salientar que essa condição não precisa envolver a variável de controle do laço. Assim como em C, a condição pode ser qualquer expressão que o programador queira testar. O campo **INCREMENTO** é executado ao final de cada iteração e serve para incrementar — ou decrementar — o valor da variável de controle.

Para quem está acostumado com a sintaxe do Pascal, esta sintaxe é bem diferente e não possui o mesmo significado. Para quem está acostumado com a sintaxe do C, este comando não é exatamente igual, pois é possível inicializar apenas uma variável e executar apenas um incremento.

Exemplos:

```

awk '{
    for (i = 1; i <= 3; i++)
        print $i
}' /etc/hosts

```

Imprime os três primeiros campos de cada linha do arquivo */etc/hosts*, um campo em cada linha.

```
$ awk 'BEGIN {for (i=1; i<100; i*=2) print i}'

```

Imprime as potências de 2 no intervalo de 1 a 10

A instrução *break*

O *break* termina de maneira incondicional laços do tipo *while*, *for* e *do-while*. Após a interpretação deste comando, o próximo comando a ser executado será aquele o que está logo após o término do escopo do laço.

Eis um exemplo:

```
awk '# encontra o mínimo divisor de num
{
    num = $1;
    for (div = 2; div*div <= num; div++)
        if (num % div == 0)
            break;
    if (num % div == 0)
        printf "Menor divisor de %d é %d\n", num, div;
    else
        printf "%d é primo.\n", num;
}'
```

Quando o número (*num*) é divisível por *div*, o comando *break* é executado, e o próximo comando a ser executado será o *if*.

A instrução *continue*

Este comando, como o *break*, pode ser utilizado somente em laços do tipo *while*, *for* e *do-while*. Quando é executado, todos os comandos subseqüentes a ele são ignorados e uma nova iteração do laço é iniciada, sendo feito novamente o teste de condição de parada.

Exemplo:

```
awk 'BEGIN {
    for (x = 0; x <= 20; x++) {
        if (x == 5)
            continue;
        printf "%d ", x
    }
    print
}'
```

Imprime os números de 0 a 20, com exceção do número 5. O comando *continue* faz a execução voltar ao início do *for*, executando a operação de incremento “*x++*”.

Outro exemplo:

```
awk 'BEGIN {
    x = 0;
    while (x <= 20) {
        if (x == 5)
            continue;
        printf "%d", x;
        x++
    }
    print
}'
```

Este programa nunca termina, pois o comando *continue* faz com que a execução volte ao início do laço *while* sem executar a operação de incremento “*x++*”.

A instrução *next*

O comando *next* faz com que as regras para o registro sendo processado sejam ignoradas daquele ponto em diante, passando o processamento imediatamente para o próximo registro.

A instrução *nextfile*

Este comando possui implementação apenas no *gawk* (Gnu awk), que é a versão utilizada no Linux. Ele causa o final de processamento do arquivo corrente, passando imediatamente para o próximo arquivo. Se houver um padrão END, então este será executado.

A instrução *exit*

Este comando faz com que o processamento do arquivo seja imediatamente abandonado. Se houver um padrão END, então este será executado.

Sintaxe:

```
exit [CÓDIGO DE RETORNO]
```

O CÓDIGO DE RETORNO é um inteiro que será retornado para o sistema operacional. Pode ser ignorado. Os programas no Unix normalmente retornam zero (0) quando o processo termina corretamente e outro número caso contrário.

Vejamos um exemplo. No abaixo, vemos que, se não for possível obter a data do sistema, o programa retornará com *exit 1*, indicando que houve um erro. Senão, imprimirá a data normalmente.

```

BEGIN {
    if (("date" | getline data_agora) < 0) {
        print "Não pude obter data do sistema">"/dev/stderr";
        exit 1
    }
    print "A data de hoje é é ", data_agora;
    close("date")
}

```

Obs.: no exemplo acima, temos na condição do *if* o comando *date* do Unix entre aspas. Essa é uma maneira de executar comandos do *shell* dentro de scripts AWK. Há uma outra forma, que é utilizando a função *system()*, que será explicada mais adiante.

6.3.5 Variáveis do AWK

Variáveis pré-definidas

A linguagem AWK tem algumas variáveis especiais pré-definidas dentro do programa que são úteis para o programador. Listaremos abaixo as variáveis mais importantes.

CONVFMT : Define o formato de conversão de um número para uma string. O formato default é “%.6g”, ou seja, um ponto flutuante com seis casas de precisão (apenas visualmente, pois para efetuar as operações aritméticas ele utiliza variáveis de ponto flutuante de precisão simples, com 32 bits).

FIELDWIDTHS : Esta variável deve sobrepor a opção FS (Field Splitting) para indicar ao AWK como os campos de um registro são separados.

FS : Esta variável indica o separador dos campos dentro de um registro. O seu valor default é um espaço em branco. Isto significa que sinais de tabulação, espaços e quebras de linha são interpretados como separadores. Pode ser redefinido pelo programador, podendo ser uma expressão regular qualquer.

O exemplo abaixo mostra como podemos redefinir o separador de campos. Abaixo, o separador de campos é redefinido para o caracter vírgula (,).

```
awk -F, 'PROGRAM' INPUT-FILES
```

- IGNORECASE** : Se o valor de **IGNORECASE** for diferente de zero ou uma string não nula então todas as comparações e busca por expressões regulares não distinguirão letras maiúsculas de letras minúsculas.
- OFMT** : Esta variável tem a mesma função do **CONVFMT** e é mantida por razões de compatibilidade com versões anteriores. O seu valor padrão é “%.6g”.
- OFS** : Esta variável define o caracter que será usado para separar um campo do outro no comando *print*. O valor padrão é o espaço em branco.
- ORS** : Define o separador de registros na saída. O seu valor padrão é a quebra de linha ($\backslash n$).
- SUBSEP** : Define o separador de índices em arrays. O seu valor padrão é o caracter “\034”. Normalmente, essa variável não tem seu valor alterado.
- RS** : Separador de registros de entrada. O valor padrão é a quebra de linha ($\backslash n$).
- \$n** : Obtém o valor atual do campo de número n. Exemplo: \$2 acessa o segundo campo do registro atual.

Variáveis do AWK que contém informações

Abaixo descreveremos variáveis que contém informações úteis que podem ser acessadas pelo programador.

ARGC : Contém o número de parâmetros passados para o programa.

ARGV : é um vetor de strings contendo o valor dos parâmetros passados para o programa. O significado é similar ao de um programa em C. Veja o exemplo abaixo.

ENVIRON : é uma variável do tipo vetor que contém os valores de todas as variáveis de ambiente configuradas durante a execução do programa. O índice para este vetor são os nomes das variáveis de ambiente. Por exemplo:

```
ENVIRON[ 'HOME' ]
```

corresponde ao diretório home do usuário que executou o programa (por exemplo, /home/pedro). A alteração do valor deste vetor não altera o valor das variáveis de ambiente.

FILENAME : Contém o nome do arquivo que o AWK está processando.

FNR : Contém o número do registro sendo processado atualmente.

NF : Contém o número de campos do registro atual.

NR : Contém o número de registros processados desde o início do arquivo.

RLENGTH : Contém o tamanho da palavra encontrada.

RSTART : Contém o índice para o primeiro caracter de uma palavra encontrada.

Ambos RLENGTH e RSTART são comumente utilizados com a função *match()*, que será explicada adiante.

Exemplo de utilização de ARGV e ARGV:

```
BEGIN {
  for (i = 1; i < ARGV; i++) {
    if (ARGV[i] == "-v")
      verbose = 1;
    else if (ARGV[i] == "-d")
      debug = 1;
    else if (ARGV[i] ~ "-?" ) {
      e = sprintf("%s: opção inválida -- %c",
                  ARGV[0], substr(ARGV[i], 1, 1));
      print > "/dev/stderr"
    } else
      break;
    delete ARGV[i]
  }
}
```

Na passagem de parâmetros para um programa AWK é necessário utilizar uma marca especial (-) para indicar que os parâmetros seguintes serão interpretados internamente. No GAWK isto não é necesssário, desde que a opção não seja reconhecida pelo próprio AWK.

Exemplo:

```
$ awk -f myprog -- -v -d file1 file2 ...
```

6.3.6 Funções do AWK

A seguir descrevemos algumas funções pré-definidas pelo awk.

Funções de manipulação de strings

gsub(r,s) : Faz a substituição global da expressão regular *r* pela string *s* em *t*. Quando o parâmetro *t* não é passado, a substituição é feita em todo o registro. *t* pode ser uma variável, um campo, ou uma expressão.

index(s,t) : Se *t* é uma substring de *s*, então a posição onde *t* inicia é retornada. O primeiro caracter em *s* inicia na posição 1 (diferente da linguagem C). O retorno é zero quando *t* não é encontrada em *s*.

length(s) : Retorna o número de caracteres em *s*.

match(s,r) : Retorna o índice da substring mais longa que pode ser encontrada em *s* através da expressão regular *r*. Como efeitos secundários, RSTART conterà o valor de retorno da função e RLENGTH conterà o tamanho da string encontrada através da expressão regular ou -1 quando não for encontrada.

split(s,A,r)* ou *split(s,A) : A string *s* é dividida em campos tendo como separador a expressão regular *r*. Quando esta expressão não é fornecida utiliza-se o valor de FS. Os campos identificados são colocados no vetor A (A[1], A[2], etc). O número de campos lidos é retornado pela função.

sprintf(format,expr-list) : Retorna uma string construída a partir de uma lista de expressões (*expr-list*) de acordo com o formato utilizado (veja o formato no *printf*).

sub(r,s,t)* ou *sub(r,s) : Faz exatamente *uma* substituição de *r* por *s* em *t*. Quando o parâmetro *t* não é utilizado, a substituição é efetuada no registro todo.

substr(s,i,n)* ou *substr(s,i) : Retorna uma substring de *s*, a partir da posição *i*, com *n* caracteres. Se o parâmetro *n* for omitido, é retornado o sufixo de *s* a partir da posição *i*.

tolower(s) : Retorna uma cópia de *s* com todos os caracteres maiúsculos convertidos para minúsculos.

toupper(s) : Retorna uma cópia de *s* com todos os caracteres minúsculos convertidos para maiúsculos.

Funções Aritméticas

<i>atan2</i> (<i>y</i> , <i>x</i>)	Arco tangente de <i>y/x</i> entre $-n$ e n .
<i>cos</i> (<i>x</i>)	Cosseno em radianos.
<i>exp</i> (<i>x</i>)	Exponencial.
<i>int</i> (<i>x</i>)	Valor inteiro de <i>x</i> .
<i>log</i> (<i>x</i>)	Logaritmo natural.
<i>rand</i> ()	Número aleatório entre zero e um.
<i>sin</i> (<i>x</i>)	Seno em radianos.
<i>sqrt</i> (<i>x</i>)	Raiz quadrada de <i>x</i> .

Funções de entrada e saída.

Existem duas funções utilizadas para mostrar a saída de um programa em AWK:

print e *printf*.

print : Imprime o registro atual na saída padrão, seguido do valor de **ORS** (separador de registros de saída), cujo valor normal é a quebra de linha ($\backslash n$).

printf* *expr1*, *expr2*, ..., *exprn : Imprime o valor das expressões na saída padrão. O separador para as expressões é definido em **OFS** (cujo valor normal é um espaço em branco).

printf* *format*, *expr-list : Utiliza a mesma sintaxe do *printf* da linguagem C.

Existe uma função para tratar a entrada de dados. Esta função aceita variações:

<i>getline</i>	Lê um registro em \$0.
<i>getline</i> < <i>file</i>	Lê em \$0 a partir de um arquivo
<i>getline</i> <i>var</i>	Lê da entrada padrão e coloca o resultado em <i>var</i>
<i>getline</i> <i>var</i> < <i>file</i>	Lê do arquivo <i>file</i> e coloca o resultado em <i>var</i> .
<i>command</i> <i>getline</i>	Coloca o resultado do comando em \$0
<i>command</i> <i>getline</i> <i>var</i>	Coloca em <i>var</i> o resultado do comando.

Operação de fechamento de arquivos

***close*(*expr*)** : fecha o arquivo indicado em *expr*. Retorna 0 quando o arquivo estava realmente aberto e -1 caso contrário. Este comando é útil quando se quer ler um arquivo novamente do início.

flush(expr) : faz com que o conteúdo do buffer de arquivo seja liberado.

flush(“”) : faz com que todos os buffers de saída sejam liberados.

6.3.7 Chamada de comando do shell

A função *system(expr)* executa um comando do shell “/bin/sh” e retorna o código de sua execução. Não retorna a sua saída para um *pipe* como seria o esperado. A sua utilidade se limita a criar arquivos no Sistema de Arquivos e utilizá-los como interface com um programa awk.

6.3.8 Definição de funções

Podemos definir nossas próprias funções num programa AWK. A sintaxe é a seguinte:

```
function nome( argumentos ) comandos .
```

Podem ser usados quaisquer comandos dos descritos acima. Um ponto que é importante destacar é que as variáveis que são usadas dentro de funções são globais e de acesso livre em qualquer parte do programa.

Assim como em C, as funções podem retornar algum valor. Para tanto, usa-se a instrução *return*, cuja sintaxe é:

```
return [expressão].
```

expressão é opcional.

Capítulo 7

Interpretador de comandos (shell)

Quando o usuário loga no sistema, logo aparece um prompt, que nada mais é do que um identificador no seu terminal, indicando que a shell está pronta para receber comandos. Muitas vezes este prompt pode ser uma seqüência de caracteres, mas o padrão é o caracter \$.

As funções do interpretador de comandos são: avaliar os comandos digitados no prompt e executar as funções desse comando; fazer o redirecionamento de entrada e saída; encadeamento de comandos (pipelines); interpretar os metacaracters e servir como uma linguagem de programação.

7.1 Comandos simples

Comando simples é uma seqüência de palavras separadas por espaços em branco ou tabulações. A primeira palavra é normalmente o comando do Linux ou um comando interno da shell. Adicionalmente, podem vir outras palavras, como nomes de arquivos a serem processados ou argumentos que modificam o comportamento do comando.

No ambiente Linux, um comando sempre retorna um valor numérico, chamado exit status (estado de saída ou resultado), para a shell. Um resultado zero, normalmente significa que o comando foi executado sem erros, e um resultado não zero ocorrerá quando for encontrado uma condição de erro ou o programa terminar de forma anormal. Quando um comando if é avaliado, a shell trata o resultado zero como uma condição verdadeira e um resultado não zero como uma condição falsa.

7.2 Redirecionamento de entrada e saída

Todos os arquivos abertos estão associados a um número de descritor de arquivo. A shell executa os comandos com 3 arquivos já abertos:

stdin - entrada padrão (teclado) - descritor 0;

stdout - saída padrão (terminal) - descritor 1;

stderr - saída de erros (terminal) - descritor 2.

Todo comando que escreve no vídeo (stdout), pode ter a saída redirecionada e todo comando que lê o teclado (stdin), pode ter a sua entrada redirecionada.

Os sinais utilizados pela shell para fazer o redirecionamento são:

1> *ou* > - redireciona a saída padrão.

2> - redireciona a saída padrão de erros.

< - redireciona a entrada padrão.

>> - redireciona e adiciona ao arquivo a saída padrão

2>> - redireciona e adiciona ao arquivo a saída padrão de erros.

Exemplos:

Redirecionar a saída de um comando `ls` para um arquivo chamado teste. Se o arquivo já existe seu conteúdo é apagado. Dependendo da variável da shell usada, um aviso é mostrado ao usuário de que o arquivo já existe.

```
$ ls > teste
```

Adicionar a saída de um comando `testesf` no final do arquivo teste já existente sem apagar o seu conteúdo.

```
$ testesf >> teste
```

Envia para alguém uma mensagem que está em um arquivo chamado teste.

```
$ mail alguem < teste
```

Procurar um arquivo chamado onde-estou e direcionar a saída de erros para o arquivo teste.

```
$ find . -name "onde-estou" 2> teste
```

7.3 Pipelines

O sistema Linux também permite que a saída padrão (stdout) de um comando seja usada como a entrada padrão (stdin) de um outro comando. Esta facilidade é chamada piping e uma série de comandos conectados através de pipes formam um pipeline.

O símbolo (|) é chamado de pipe. A saída do comando à esquerda do pipe é usada como entrada para o comando à direita dele.

Um pipeline é um tipo particular de multiprocessamento sincronizado pelo kernel, onde a saída de um comando é a entrada para o comando seguinte. Dessa forma, tarefas complexas podem ser realizadas pela conexão de comandos simples através de pipes.

A vantagem do uso de pipes é que não precisamos de arquivos temporários para guardar a saída de um comando que posteriormente vamos usar.

O comando `who` lista todos os usuários que estão no sistema e o comando `wc -l` conta o número de linhas de um arquivo. Usando estes dois comandos e um pipe, podemos descobrir quantos usuários estão logados no sistema.

Exemplo:

```
$ who | wc -l
19
```

No exemplo existem 19 usuarios logados no sistema.

7.4 Seqüência de comandos

Uma seqüência de comandos consiste em um ou mais comandos simples separados por ponto-e-vírgula, por e-comercial &, por duplo e-comercial && ou duplo sinal de pipe ||. Em uma lista de comandos, estes separadores são avaliados da esquerda para a direita com a seguinte precedência: &&, ||, ; e &.

Um ponto-e-vírgula permite a execução de uma lista de comandos. A shell espera que cada comando na lista tenha terminado antes de executar o próximo comando da lista. A saída de cada comando é enviada para a saída padrão.

Exemplo:

Listar o conteúdo do diretório /tmp e em seguida do diretório home

```
$ ls /tmp ; ls /home
```

O operador `and (&&)` faz com que o comando seguinte seja executado somente se o comando precedente resultar num valor zero.

Exemplo:

Se o comando `cc` depois de aplicado ao arquivo `teste` resultar em zero, o comando `teste` sera executado

```
cc teste.c && teste.
```

Para executar um comando e transformar a sua saída em uma lista de parâmetros, basta colocar os argumentos do comando entre crases duplas.

Exemplo:

Mostrar o conteúdo de todos os arquivos que terminam em `.txt`

```
$ cat `ls *.txt`  
teste1.txt teste15.txt teste2.txt
```

No `bash` também pode-se usar:

```
$ cat $(ls *.txt)
```

7.5 Grupo de comandos

Você pode agrupar uma lista de comandos seqüenciais colocando-os dentro de um par de parênteses `()`. A entrada padrão pode ser redirecionada para o grupo, assim como a saída padrão do grupo inteiro pode ser redirecionada também.

Exemplo:

Os comandos que estão separados por ponto e virgula serão executados normalmente e a saída do comando `date` sera redirecionada como entrada do arquivo `status`.

```
$ ls ; who ; date > status
```

Neste exemplo , como os comandos estão agrupados pelos parênteses, a saída de todos irá para o arquivo `status`

```
$ (ls ; who ; date) > status
```

Agora a saída de todos os três comandos foram enviadas para o arquivo `status`.

Estando a lista de comandos entre parênteses, os comandos serão executados em uma nova instância da shell. Todas as atribuições de variáveis serão desfeitas após o fecha-parênteses.

7.6 Histórico de comandos

A shell mantém um histórico de comandos que poderão ser reutilizados pelo usuário. Estes comandos são gravados em um arquivo, a cada vez que o usuário encerra uma seção da shell, ou seja, desloga do sistema. O comando `history` lista os comandos que foram executados.

Para recuperar os comandos você poderá executar:

```
$ !!      Recupera e executa o último comando.  
$ !texto  Recupera e executa o último comando que começa com texto.  
$ ?texto? Recupera e executa o último comando que contendo o texto.
```

7.7 Bash

Bash é uma shell, significa “Bourne-Again SHell” em homenagem a Steve Bourne, o autor da shell `sh`, um ancestral direto do `bash`. O `sh` apareceu na sétima edição do Unix da Bell Labs.

Algumas características do `bash`:

- Incorpora funções do `ksh` e `csh` (korn e C shell respectivamente) e tenta ser fiel à especificação IEEE POSIX para shell e ferramentas (IEEE Working Group 10003.2).
- É muito portátil e está disponível em praticamente todos os sistemas Unix. Existe atualmente versões para OS/2 e Windows NT.
- Provê uma interface em forma interativa (comando a comando) e programada (diversos comandos organizados em forma de um programa – veja “criando um programa shell”).

O objetivo desta seção é fornecer informações básicas e exemplos de utilização do `bash` para facilitar a vida do usuário que utiliza ou administra um sistema UNIX.

7.7.1 Iniciando o bash

O bash é geralmente iniciado quando o usuário entra no sistema, caso esta seja sua shell padrão. Um usuário pode utilizar outros interpretadores de comando se desejar, mas isto é normalmente definido no momento de cadastro do mesmo e fica anotado no arquivo passwd. Um usuário, ou super-usuário, pode trocar a sua shell com o comando chsh (change shell). Os tipos de shell disponíveis podem ser vistos no arquivo /etc/shells.

7.7.2 Tratamento de expansões

Expansões são expressões regulares. Listamos abaixo algumas expansões que o bash faz quando utilizado em forma interativa (em um terminal Linux, por exemplo).

`\` : É tratado como um caracter especial, quando fora das aspas, significa que o próximo caracter deve ser tratado como um caracter simples. Quando a barra é o último caracter ao final de uma linha, significa que o comando continuará na linha seguinte, fazendo com que o interpretador ignore a quebra de linha.

`'` : A aspa simples serve para preservar o valor literal dos caracteres - principalmente dos caracteres especiais, como a `\` .

```
$ echo '\e'
\e
```

```
$ echo \e
e
```

`“` : As aspas duplas também preservam o valor literal dos caracteres especiais, com exceção do `$`, `'` e `\` (cifrão ou dólar, crase e barra invertida). A barra invertida continua mantendo seu significado especial quando utilizado antes de `$`, `'`, `\` e `<enter>` (cifrão, crase, barra invertida e quebra-de-linha). As aspas duplas podem ser representadas utilizando-se o `\`”

`\a` : Bip

Exemplo: o `-e` é utilizado para habilitar as expansões

```
$ echo -e "\a"
```

\b : Backspace. Exemplo:

```
$ echo -e  
"erro\b\b\bro "  
erro
```

\f : Quebra de página (form feed)

\n : Quebra de linha (newline)

\r : Retorno de linha

\v : Tabulação vertical

**** : Barra invertida

\NNN : Representação octal do caracter ASCII

: Quando utilizado em programas, significa início de comentário (até uma quebra de linha)

~ : Abrevia o diretório pessoal do usuário.

***** : Qualquer seqüência de caracteres (nenhum inclusive).

```
$ ls teste*  
teste1 teste2 teste3 teste10
```

? : Exatamente um caracter qualquer.

```
$ ls teste1?.txt  
teste10.txt teste12.txt
```

[abc] : Qualquer caracter do conjunto.

```
$ ls teste[123]
teste1 teste2 teste3
```

[!abc] : Qualquer caracter fora do conjunto.

```
$ ls teste[!23].txt
teste1.txt teste4.txt
```

Exemplo :

```
echo -e "L\vI\vN\vU\vX\v\x020\x20\x20\x20\x20\x20\x033[2A\x20I\033[1A\x20I\x20\x20\x20\x20\x20\x033[7mP\x020E\x020T\033[0m\x020-\x20\033[32;1mU\x020F\x020P\x020R\033[0m\033[2B"
```

7.7.3 Variáveis de ambiente

A shell utiliza algumas variáveis com significado especial, cujos valores influenciam na execução de comandos e funções da shell. Seguem abaixo as principais utilizadas:

IFS : Contém os caracteres que servem como separadores de palavras na linha de comando. O valor padrão é o <enter> (“\n”).

PATH : Contém uma lista dos diretórios onde os programas executáveis devem ser procurados. Os nomes dos diretórios devem ser separados por dois-pontos.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

HOME : Indica qual é o diretório pessoal do usuário. Este diretório, normalmente chamado de casa ou home, é definido no momento de cadastro do mesmo.

PS1 : Contém a primeira parte do prompt (os símbolos que aparecem no início de cada linha de comando da shell).

```
$ echo PS1
PS1='\$ '
$ export PS1='\s-\v $PWD\$ '
bash-2.01 /home/sysadm$
```

(O exemplo mostra a alteração do `prompt` para que ele contenha o nome da shell que esta sendo utilizada, sua versão e o nome do diretório corrente)

PS2 : Corresponde à segunda parte do prompt. Valores especiais que podem ser utilizados no prompt (PS1 e PS2):

‘\a’ bip

‘\d’ data (“Tue May 26”).

‘\e’ esc (ASCII 27)

‘\h’ nome da máquina

‘\n’ quebra de linha

‘\s’ nome da shell (bash, csh, etc.)

‘\t’ hora em HH:MM:SS (24 horas)

‘\T’ hora em HH:MM:SS (12 horas)

‘\h’ hora em HH:MM:SS (12 horas, com am/pm)

‘\v’ versão do bash

‘\V’ número de distribuição (release) do bash

‘\w’ o diretório de trabalho atual (caminho completo)

‘\W’ o diretório de trabalho atual (apenas o nome do diretório, sem o caminho)

‘\u’ nome do usuário

‘\!’ número do comando (de acordo com o history)

‘\#’ número deste comando

‘\\$’ # para o root e \$ para usuários normais

‘\nnn’ caracter ASCII indicado pelo número octal nnn

‘\’ a barra invertida

‘\[’ inicia uma seqüência de caracteres não visíveis (abaixo de 32)

‘\]’ termina uma seqüência de caracteres não visíveis.

7.7.4 Expansões especiais

A shell executa algumas expansões de forma especial:

{ } : Os padrões colocados entre chaves serão utilizados para substituir parte do parâmetro final.

```
$ echo arq{teste,deteste,reteste}
arqteste arqdeteste arqreteste
```

(Este comando teve na realidade 3 parâmetros que foram criados na expansão das chaves)

```
$ echo arq{{{,de,re}teste}.txt
arqteste.txt arqdeteste.txt arqreteste.txt
```

(Pode-se utilizar chaves dentro de chaves, causando uma recombinação a cada expansão feita)

~user : Corresponde ao valor do diretório pessoal do usuário especificado.

```
$ echo ~sysadm
/home/sysadm
```

~+ : Corresponde ao diretório atual.

~- : Corresponde ao último diretório visitado.

```
$ cd
$ cd /usr/lib
$ echo ~+
/usr/lib
$ echo ~-
/home/sysadm
```

7.7.5 Arquivos de configuração

O bash procura diversos arquivos de configuração. Estes arquivos contêm normalmente definições de variáveis de ambiente e comandos a serem executados no momento de login. O usuário tem liberdade de modificar alguns deles, mas o administrador de sistema também podem interferir de maneira global na configuração do bash. Veja abaixo, na seqüência de leitura do bash, os arquivos de configuração:

/etc/profile : É o primeiro arquivo lido pelo bash quando o login é interativo (a partir de uma tela de login, telnet, etc.). Se o arquivo não existir o bash procura o `/.bash_profile`.

```
$ cat /etc/profile
```

```
# /etc/profile: system-wide .profile file for bash(1).
```

```
PATH="/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games"
```

```
PS1="\$ "
```

```
Export PATH PS1
```

/.bash_profile, /.bash_login, /.profile : No caso de login interativo, estes arquivos são lidos nesta seqüência e ficam na área do usuário (diretório pessoal). Não é necessário que todos estes arquivos existam. Somente um deles é executado.

```
# /.bash_profile: executed by bash(1) for login shells.
```

/.bash_logout : Quando este arquivo existe, antes do bash terminar a sua execução normal, o script é executado.

/.bashrc : Este arquivo é executado no lugar do `.bash_profile` quando uma shell interativo é inicializado, mas não a partir de um login. (Uma shell pode ser inicializada simplesmente chamando um "bash" ou como um comando a ser executado dentro de um xterm).

7.7.6 Edição das linhas de comando

O bash utiliza vários recursos de edição na linha de comando. Este recursos são similares aos utilizados pelo emacs* (editor de textos) porque utilizam como base a mesma biblioteca readline.

*Para utilizar recursos similares aos do vi, executar `set -o vi`.

Os comandos digitados são armazenados em um arquivo chamado `.bash_history` e podem ser reutilizados mediante alguns comandos.

Listaremos abaixo diversas teclas de atalho disponíveis durante a edição de um comando no prompt da shell. A seguinte convenção será utilizada:

C - significa Control <Ctrl>.

M - significa a tecla Meta, muitas vezes ela esta associada à tecla <Alt> e outras vezes a tecla <esc> .

	move-se na lista de comandos digitados.
! N	N-ésimo comando.
! -N	N-ésimo comando anterior.
!!	último comando
! STRING	último comando iniciando pela STRING
! ?STRING[?]	último comando contendo a STRING
^ STRING1 ^ STRING2 ^	repete o último comando substituindo a STRING1 pela STRING2
<C-b>	volta um caracter
<C-f>	avança um caracter
	apaga um caracter à esquerda do cursor
<C-d>	remove o caracter abaixo do cursor
<C-_)>	desfaz o último comando de edição (pode desfazer vários comandos)
<C-a>	vai para o início da linha
<C-e>	vai para o final da linha
<M-f>	vai a próxima palavra
<M-b>	volta uma palavra
<C-k>	apaga até o final da linha
<M-d>	remove a linha atual
<M-DEL>	remove a palavra atual
<C-w>	remove até o último espaço em branco
<C-y>	cola o último texto recortado (ou removido)
<M-y>	faz a rotação dos últimos trechos recortados
<C-l>	limpa a tela

7.7.7 Estruturas de controle (loops)

O bash fornece algumas estruturas de controle similares às estruturas de linguagens de programação. Estas estruturas de controle são utilizadas normalmente em programas (scripts), mas podem ser utilizadas normalmente na linha de comando do terminal. Os ponto-e-vírgula utilizados nos modelos podem ser substituídos por quebras de linha em um programa. O que aparece entre colchetes [] são parâmetros opcionais.

until

```
until TESTE; do COMANDOS; done
```

Enquanto **TESTE** for falso executa **COMANDOS** (o falso é um retorno diferente de zero e verdadeiro o valor zero – exatamente o contrário da linguagem C).

Exemplo 7.7.7.1 Enquanto o arquivo */tmp/teste.lock* não existir resulta falso e o comando *du* é executado.

```
$ until ls /tmp/teste.lock ; do du /tmp/teste.txt; done
ls: /tmp/teste.lock: No such file or directory
1 /tmp/teste.txt}
ls: /tmp/teste.lock: No such file or directory
1 /tmp/teste.txt
ls: /tmp/teste.lock: No such file or directory
1 /tmp/teste.txt
```

while

```
while TESTE; do COMANDOS; done
```

Enquanto o resultado do **TESTE** for verdadeiro executa a lista de **COMANDOS**.

Exemplo 7.7.7.2 Enquanto o arquivo */tmp/teste.lock* existir executa o comando *du* */tmp/teste.txt*.

```
$ while ls /tmp/teste.lock ; do du /tmp/teste.txt; done
ls: /tmp/teste.lock: No such file or directory
```

for

```
for VARIÁVEL [in PALAVRAS ...] ; do COMANDOS; done
```

Executa **COMANDOS** tantas vezes quanto o número de **PALAVRAS**, a cada ciclo **VARIÁVEL** é setado com a **PALAVRA** atual. Os valores que **VARIÁVEL** irá assumir são listados um após o outro, separados por espaço. Quando nenhum valor é listado, o programa assume os parâmetros passados durante a sua chamada.

Exemplo 7.7.7.3 Repete o comando `echo teste$i > teste$i.txt` cinco vezes. A cada vez que a repetição é feita o valor de `$i` é trocado por 1, 2, 3, etc., conforme listado no próprio comando.

```
$ for i in 1 2 3 4 5; do echo teste$i $>$ teste$i.txt; done
$ cat teste1.txt
teste1
```

break

Termina a execução de um loop (`until`, `while`, `for`), ignorando todos os comandos subsequentes.

continue

Ignora todos os comandos subsequentes e volta ao início do loop.

7.7.8 Comandos condicionais (if, case, select)

São comandos similares aos disponíveis em linguagens de programação. O que aparece em colchetes [] são parâmetros opcionais.

if

```
if TESTE1; then
    COMANDOS1;
[elif TESTE2; then
    COMANDOS2;]
[else OUTROS-COMANDOS;]
fi
```

Se o **TESTE1** for verdadeiro (retorno de comando igual a zero) então execute **COMANDOS1**; Se o **TESTE1** for falso e o **TESTE2** for verdadeiro, então execute os **COMANDOS2**; senão, se todos os testes anteriores forem falsos, execute **OUTROS-COMANDOS**.

Exemplo 7.7.8.1 Executa o primeiro *ls teste16.txt* mas não encontra o arquivo, então executa o segundo *ls teste15.txt* e encontra o arquivo, então executa o *cat teste15.txt*.

```
$ if ls teste16.txt; then \
> cat teste16.txt; \
> elif ls teste15.txt; then \
> cat teste15.txt;
> else cat teste1.txt;
> fi
ls: teste16.txt: No such file or directory
teste15.txt
teste15
```

case

```
[case PALAVRA in [(PADRAO[|PADRAO]...)] COMANDOS
;;]... [(PADRAO[|PADRAO]...)] COMANDOS ;;]... esac]
```

Caso a PALAVRA combine com o PADRAO então execute os COMANDOS. Faça o teste para todos os PADRÃO utilizados.

Exemplo 7.7.8.2 Esta sequência de comandos lê o nome de um animal e faz a comparação para ver se ele pertence à lista de palavras conhecidas. Quando encontra uma palavra que combine executa o comando. No exemplo acima, o *echo -n* mostra uma mensagem sem colocar a quebra de linha no final da linha. O padrão * significa que qualquer palavra combina, mas esta regra só será executada se nenhum padrão anterior combinar.

```
echo -n "Digite o nome de um animal: "
read ANIMAL
echo -n "O $ANIMAL tem "
case $ANIMAL in
(horse | dog | cat) echo -n "quatro";;
(man | kangaroo ) echo -n "duas";;
(*) echo -n " um número desconhecido de";;
esac
echo " patas."
```

select

```
select VARIABEL [in PALAVRAS ...]; do COMANDOS; done
```

Constrói um menu com as PALAVRAS fornecidas na lista e as associa a um número. O valor selecionado é armazenado em VARIABEL. Executa uma seqüência de comandos logo após a leitura de uma opção. É mais útil se utilizado em conjunto com o comando case. Exemplo:

Exemplo 7.7.8.3 Utilização do comando *select* em conjunto com o comando *case*.

```
$ cat >select.sh
select i in listdir mostreuso mostredir; do
case $i in
(listdir) ls * ;;
(mostreuso) du -s * ;;
(mostredir) find . -type d ;;
esac;
done
^D
$ chmod 755 ./select.sh
$ ./select.sh
1. )listdir

2. ) mostreuso

3. ) mostredir

#?
```

7.7.9 Definindo funções

O bash permite que um programa declare funções de maneira similar às linguagens de programação estruturada.

function NOME () { COMANDOS; } :

A declaração de uma função pode utilizar a palavra chave “function” para dar maior clareza à definição, mas é opcional. O NOME da função é livre e deve iniciar com uma letra. Os COMANDOS são executados na mesma shell na qual a função foi chamada. A passagem de parâmetros funciona de maneira análoga à passagem de parâmetros para um programa shell (veja “Passagem de Parâmetros”).

```
$ function liste () \{
> ls -l | more --5;
> \}
$liste
-rw-rw-r-- 1 sysadm sysadm 7 Jan 1 15:14 teste1.txt
-rw-rw-r-- 1 sysadm sysadm 8 Jan 1 15:14 teste10.txt
-rw-rw-r-- 1 sysadm sysadm 8 Jan 1 15:14 teste11.txt
-rw-rw-r-- 1 sysadm sysadm 8 Jan 1 15:14 teste12.txt
-rw-rw-r-- 1 sysadm sysadm 8 Jan 1 15:14 teste13.txt
--More--
```

Foi definida uma função chamada “liste” na shell atual, contendo um comando “ls -l | more -5”. Esta função é executada chamando-se “liste”

7.7.10 Passagem de Parâmetros

A passagem de parâmetros para um programa shell pode ser feita através de uma lista passada no momento da invocação do programa ou através de variáveis da própria shell.

O tratamento dos parâmetros dentro do programa shell são acessados da seguinte forma:

\$* : Representa todos os parâmetros concatenados um após o outro.

\$: Representa todos os parâmetros em palavras separadas, uma após a outra.

\$# : Indica o número de parâmetros passados para o programa.

\$i : Representa o i-ésimo parâmetro passado para o programa.

Exemplo 7.7.10.1 Utilização de parâmetros.

```
$ cat > exemplo.sh
echo "Numero de parametros passados: $#"
```

```
echo "Parametros passados \${@}: ${@}"
echo "Parametro 1: $1"
echo "Parametro 2: $2"
```

```
^D
```

```
$ ./exemplo.sh x y z
Numero de parametros passados: 3
Parametros passaos $*: x y z
Parametros passados ${\@}: x y z
Parametro 1: x
Parametro 2: y
```

\$? : Contém o valor de retorno do último comando executado (status).

\$- : Corresponde a opções passadas para o próprio bash no momento de sua inicialização

\$\$: Representa o número do processo sob o qual a shell está executando. Quando outra shell é executada mantém o identificador original da shell pai.

- \$!** : Representa o número do processo do último comando executado em background.
- \$0** : Corresponde ao nome do próprio arquivo que contém o programa shell.
- \$_** : Tem múltiplos valores: no início contém o nome do arquivo de programa sendo executado; posteriormente contém o último argumento do comando anterior e também o nome do arquivo de comandos sendo executados no mesma shell.

Exemplo 7.7.10.2 Utilização de \$_

```
$ cat exemplo.sh
echo "Primeira vez: $_"
ls
echo "Segunda vez:$_"
ls teste1.txt
echo "Terceira vez: $_"
ls --l
echo "Quarta vez: $_"
$ ./exemplo.sh
Primeira vez: bash
exemplo.sh teste11.txt teste14.txt teste3.txt
teste13.txt teste2.txt teste5.txt teste8.txt
Segunda vez: ls
teste1.txt
Terceira vez: teste1.txt
total 16
-rwxr-xr-x 1 sysadm sysadm 117 Jan 1 17:33 exemplo.sh
-rw-rw-r-- 1 sysadm sysadm 7 Jan 1 15:14 teste1.txt
...
Quarta vez: -l
```

\${VARIÁVEL}\$VARIÁVEL : Substitui pelo valor da variável. A primeira forma, entre chaves { }, é a mais formal. Se a variável não existe ou não tem valor, a substituição retorna uma string vazia.

```
$ echo "Meu diret\ '{o}'rio \ '{e}' ${HOME}"
Meu diretorio e /home/sysadm
```

\${VARIÁVEL1:-VARIÁVEL2} : Se a VARIÁVEL1 existir, ou estiver definida dentro do programa ou na shell que o invocou, seu valor será utilizado. Caso contrário e valor da VARIÁVEL2 será utilizado.

```
MYHOME=${HOME:-' /home/pedro' }
```

`${VARIÁVEL:+OUTRADEFINICAO}` : Se a VARIÁVEL existir, então expande o valor de OUTRADEFINICAO. †

```
$ echo "Meu diretório e ${HOME:+já definido}"
Meu diretório e já definido
$
```

7.7.11 Redirecionamento de entrada e saída

A shell permite diversos tipos de redirecionamento de arquivos. Um arquivo pode ser referenciado através de seu nome no sistema de arquivo ou através dos seus números associados. Três arquivos são tratados de maneira especial pela shell:

entrada padrão (descriptor 0) : Quando a shell é iniciado este arquivo está associado diretamente ao teclado. É possível redirecionar outro arquivo para substituir o teclado. (Na linguagem C corresponde ao `stdin`)

saída padrão(descriptor 1) : Este arquivo é associado inicialmente ao terminal de vídeo. (Na linguagem C corresponde ao `stdout`)

saída padrão de erros (descriptor 2) : Os programas utilizam este arquivo para mostrar os erros ocorridos durante a sua execução. Está inicialmente associada ao terminal de vídeo. (Na linguagem C corresponde ao `stderr`).

Descreveremos os tipos de redirecionamento através de exemplos:

ls > ARQUIVO : A saída do comando `ls` é redirecionada para o ARQUIVO (exceto as mensagens de erro enviadas para a saída padrão de erros).

†Existem outras modalidades de expansão de variáveis que podem ser vistas na documentação on-line.

Exemplo 7.7.11.1 Redirecionamento da saída.

```
$ ls teste16.txt teste1?.txt > texto.txt
ls: teste16.txt: No such file or directory
$ more --5 texto.txt
teste10.txt
teste12.txt
teste13.txt
teste14.txt
--More--(83{\%})
```

`ls > ARQUIVO 2>&1` : A saída do comando `ls` é redirecionada para o ARQUIVO, inclusive a saída padrão de erros (o descritor 2 é redirecionado para o descritor 1).

Exemplo 7.7.11.2 Redirecionamento da saída padrão e de erros.

```
$ ls teste16.txt teste1?.txt > texto.txt 2>&1
$ more --5 texto.txt
ls: teste16.txt: No such file or directory
teste10.txt
teste12.txt
teste13.txt
--More--(79{\%})
```

`ls >> ARQUIVO` : Causa o redirecionamento para o final do ARQUIVO, isto é, não destrói o conteúdo de arquivo, apenas adiciona.

7.7.12 Busca e execução de comandos

O bash segue alguns passos para encontrar um comando na estrutura de diretórios. Todos os comandos no sistema Unix correspondem a algum arquivo.

- Se o comando não contém barras de diretórios, a shell tenta localizá-lo. Se existe um alias do próprio bash com aquele nome, este comando será imediatamente interpretado. Se o comando contém barras de diretórios o bash vai para o passo 4.
- Se o comando não é nenhuma função do bash, mas faz parte de alguma outra função pré-definida, então esta função é executada (builtins).

- Se o comando não corresponde a nenhuma função do bash, então é feita uma busca nos diretórios especificados na variável de ambiente PATH. Os arquivos destes diretórios são analisados no momento de execução do bash para diminuir o tempo de busca de um comando. Quando um comando não é encontrado na lista do PATH é feita uma busca completa em todos os arquivos de todos os diretórios indicados pelo PATH.
- Quando o comando é encontrado no Sistema de Arquivos ele é imediatamente executado.
- Se o programa não está em formato executável (binário) e não é um diretório, o bash assume que ele é um programa shell (shell script).
- Se o diretório atual não estiver incluído no PATH, então o arquivo executável que estiver no diretório atual não será incluído na busca e provavelmente não será executado. Pode-se então indicar o diretório atual explicitamente utilizando-se o `.` (diretório atual).

```
./teste
```

Executa o arquivo teste no diretório atual sem fazer a busca no PATH.

7.7.13 Valores de retorno de uma função

Geralmente os programas retornam um código numérico quando terminam sua execução. A utilização dos códigos segue a seguinte convenção:

0 : Quando o comando termina com sucesso, o seu valor de retorno é zero.

127 : Quando o comando não foi encontrado

128+N : Quando o comando foi encerrado com um sinal do sistema de valor N (veja o comando kill).

126 : Quando o comando foi encontrado mas não é executável

7.7.14 Comandos internos do bash - builtin ou função pré-definida

: [-ARGUMENTOS] : Apenas expande os argumentos e executa os redirecionamentos.

. **ARQUIVO** : Executa os comandos do arquivo sem abrir uma nova shell para interpretá-los (quando se executa um script, uma nova shell é invocada). É útil quando se deseja alterar o valor das variáveis de ambiente na shell atual.

```
. .bash{\_}profile
```

(lê novas definições feitas no .bash_profile sem ser necessário reiniciar a shell).

break : Descontinua um for, while, until ou select.

cd : Muda de diretório (sem argumentos vai para o diretório indicado pela variável de ambiente HOME).

continue : Reinicia o loop em um comando do tipo for, while, until e select.

eval [-ARGUMENTOS] : Os ARGUMENTOS são concatenados em um único comando e são então executados.

exec [-c [-a NOME] [COMANDO] [ARGUMENTOS]] : O COMANDO substitui a execução da shell corrente. A opção -c causa a execução sem nenhuma das variáveis de ambiente existentes. A opção -a faz com que a string NOME seja passada como o nome do comando (\$0). Os ARGUMENTOS são passados diretamente para o COMANDO.

exit [-N] : Termina a execução da shell e retorna o valor de N como status. Ao contrário de return volta todos os níveis superiores.

export [-f [-p][NAME[=VALE]]] : Configura o valor de uma variável de ambiente. A opção -f indica que NAME é uma função da shell. A opção -p mostra todos os nomes de variáveis sendo utilizados.

```
export PATH=$PATH:.
```

(Inclui o diretório corrente no caminho de programas executáveis).

getopts **OPTSTRING** **NAME** [**ARGS**] : Extrai parâmetros da shell.

Exemplo 7.7.14.1 Parâmetros passados na linha de comando.

```
$ cat ./teste.sh
getopts 'h:' HELP
echo "opcao $HELP: $OPTARG"
getopts 'n:' NOME
echo "opcao $NOME: $OPTARG"
$ ./teste.sh -h help -n leonardo
opcao h: help
opcao n: leonardo
$
```

O argumento **OPTSTRING** deve conter o nome do argumento sendo analisado, seguido de dois-pontos caso este argumento espere uma string adicional. Os valores obtidos são colocados em **OPTIND** (o índice relativo ao argumento encontrado) e **OPTARG** (o valor lido como opção para o argumento). Quando o argumento não é encontrado, o valor “?” é colocado em **OPTIND**.

hash [**-r** [**-p** **FILENAME**] [**NAME**]] : Mostra os comandos executados e a sua localização ou indica a localização de um novo comando. Isto evita que o shell procure o comando novamente quando for invocado uma próxima vez. A opção **-r** elimina as localizações anteriores. A opção **-p** faz a indicação da localização do comando indicado por **NAME** sem a necessidade de se buscar no **PATH**.

Exemplo 7.7.14.2 Número de vezes que cada comando foi executado.

```
$ hash
hits command
1 /usr/bin/touch
3 /usr/bin/whereis
6 /bin/cat
1 /bin/chmod
2 /usr/bin/clear
4 /bin/date
4 /usr/bin/man
4 /bin/more
16 /bin/ls
4 /bin/su
19 /usr/bin/vi
$
```

pwd [-LP] : Contém o nome do atual diretório de trabalho. A opção **-L** faz com que os links simbólicos sejam mostrados e a opção **-P** faz com que os links simbólicos não sejam mostrados.

readonly [-apf [NAME] ...] : Faz com que uma variável de ambiente torne-se uma constante, desabilitando qualquer alteração posterior de seu valor. A opção **-f** indica que o **NAME** é uma função; a opção **-a** indica que **NAME** refere-se a um vetor de variáveis e a opção **-p** faz com que sejam mostradas as variáveis de ambiente que são “readonly”

return [N] : Utilizado apenas dentro de funções definidas na shell para retornar um valor para o nível superior.

shift [-N] : Causa um deslocamento na indexação dos parâmetros passados para um programa shell. O argumento **N** indica em quantas posições devem ser deslocados os parâmetros. O argumento **\$1** para a ser o **\$(1-N)** e assim sucessivamente.

test : Utilizados para avaliar expressões. (Veja avaliação de expressões)

times : Mostra o tempo utilizado pela shell e pelos seus processos filhos.

[trap [-lp [ARG] [SIGSPEC ...]] : O comando especificado em **ARG** deve ser executado se o programa shell receber um dos sinais especificados na lista. O tipos de sinais pode ser vistos com o comando **kill -l** (Veja o comando **kill**).

```
$ cat ./sinais
function fim() {
echo Recebi um SIGINT ^C
}
trap fim SIGINT
while (( 1 )) ; do sleep 1; done
$ ./sinais
Recebi um SIGINT ^C
Recebi um SIGINT ^C
```

Mesmo após o recebimento do sinal o programa continua executando. A rotina **fim** é executada cada vez que se pressiona o **^C** sobre o programa ou um **kill** é executado sobre ele. A forma de matar este processo seria utilizando um **kill -9**.

umask [-S [MODE]] : Mostra ou modifica o modo padrão de criação de arquivos. Toda vez que um novo arquivo for criado nessa shell ele terá as permissões especificadas pelo umask.

```
$ umask -S
u=rwx,g=rwx,o=rx
$ umask
002
$
```

O primeiro formato indica que o usuário pode ler (r), alterar (w) e executar (x) qualquer arquivo; o grupo idem e outros usuário podem ler e executar arquivos mas não podem alterá-los ou removê-los.

unset [-fv [NAME]] : Desfaz a definição de uma variável ou função. A opção **-f** indica que está removendo-se uma função e a opção **-v** indica que está se removendo uma variável.

pushd DIR : Empilha um diretório. Caso não seja passado parâmetro faz a troca entre os dois últimos diretórios empilhados.

popd : Desempilha o último diretório empilhado. A ordem de manipulação da pilha é LIFO, ou seja, o último que entra será o primeiro que sai.

dirs : Mostra uma lista com os últimos diretórios empilhados.

history : Mostra os últimos comandos executados.

logout : Termina a shell.

source : Faz o mesmo que o comando **.** (ponto), ou seja, executa comandos de um arquivo de comandos sem chamar outra shell. Serve para alterar as configurações de variáveis da shell atual.

7.7.15 Expressões lógicas

O bash provê diversos testes sobre expressões aritméticas e sobre arquivos que retornam valores verdadeiros ou falsos e podem ser utilizados em comandos **if**, **while** e **until**. Uma expressão deve vir depois de um comando **test** ou entre colchetes [].

-b FILE : Verdadeiro quando FILE é um arquivo especial de blocos (discos rígidos, disquetes, etc.).

Exemplo:

```
$ if [ -b /dev/fd0 ]; then echo Existe disquete; else echo Não existe ; fi
Existe disquete
```

-c FILE : Verdadeiro quando FILE é um arquivo de caracteres (impresora, terminal texto, etc.).

-d FILE : Verdadeiro quando FILE é um diretório

-e FILE : Verdadeiro quando FILE existe.

-f FILE : Verdadeiro quando o FILE existe e é um arquivo comum (não termina, disco, memória ou outros dispositivo).

-L FILE : Verdadeiro quando FILE é um link simbólico.

-r FILE : Verdadeiro quando FILE existe e possui permissão para leitura.

-s FILE : Verdadeiro quando FILE existe e possui tamanho diferente de zero.

-w FILE : Verdadeiro quando FILE existe e possui permissão para escrita.

-x FILE : Verdadeiro quando FILE existe e é executável.

FILE1 -nt FILE2 : Verdadeiro quando FILE1 é mais recente (modificado) que FILE2.

FILE1 -ot FILE2 : Verdadeiro quando FILE1 é mais antigo que FILE2.

-z STRING : Verdadeiro quando o tamanho de STRING é zero.

-n STRING : Verdadeiro quando o tamanho de STRING é diferente de zero.

STRING1 == STRING2 : Verdadeiro se o conteúdo das duas STRINGS são iguais.

STRING1 != STRING2 : Verdadeiro se as STRINGS são diferentes.

STRING1 < STRING2 : Verdadeiro se a STRING1 é menor que a STRING2. (Ordem lexicográfica).

STRING1 > STRING : Verdadeiro se a STRING1 é maior que a STRING2.
(Ordem lexicográfica).

! EXPR : Verdadeiro se o resultado da EXPR é falso (negação).

EXPR -a EXPR2 : Verdadeiro se ambas as expressões são verdadeiras
(and).

EXPR1 -o EXPR2 : Verdadeiro se uma das expressões é verdadeira
(or).

ARG1 OP ARG2 : Quando ARG1 e ARG2 são números os seguintes
OP (operadores) são válidos:

-eq igual

-ne diferente

-lt menor que

-gt maior que

-ge maior ou igual

7.7.16 Expressões Aritméticas

O bash executa várias operações aritméticas diretamente na linha de comando.

```
$ let a=1 + 2; echo $a
3
```

É uma operação feita pelo própria shell. O comando let indica que a expressão seguinte é uma expressão aritmética.

Uma expressão aritmética é substituída por seu valor utilizando-se a forma:

```
$(( EXPR ))
```

Exemplo:

```
$ echo $(( 465 * ( 23 / -2 + 5 % 3 ) ))
-4185
```

A tabela abaixo mostra os operadores em ordem decrescente de precedência:

- + : operadores unários de sinal (negativo e positivo).

Exemplo:

```
$ let "a = 4 * -3";echo $a
```

```
-12
```

```
$
```

! ~ : negação lógica e negação

* / % : multiplicação, divisão e resto da divisão.

- + : subtração e soma.

<< >> : deslocamento de bits à esquerda e à direita. Exemplo:

```
$ let "a=1"; let "a=a<< 2"; echo $a
```

```
4
```

<= >= < > : menor ou igual, maior ou igual, menor e maior.

== != : igual e diferente.

& : “e” binário.

^ : “ou exclusivo” binário

| : “ou” binário

&& : “e” lógico

|| : “ou” lógico

teste ? expr1 : expr2 : Se o teste é verdadeiro, retorna a expressão expr1, caso contrário retorna a expressão expr2.

Diversos tipos de atribuições aritméticas:

=	a = b
*=	a *= b ou a = a * b
/=	a /= b ou a = a / b
%=	a %= b ou a = a % b
+=	a += b ou a = a + b
-=	a -= b ou a = a - b
<<=	a <<= b ou a = a << b
>>=	a >>= b ou a = a >> b
&=	a &= b ou a = a & b
^=	a ^= b ou a = a ^ b

Obs.: Pode-se utilizar diferentes bases numéricas: números começando com zero são considerados em base octal, 0x e 0X como base hexadecimal e sem nenhum desses prefixos são considerados números em base decimal. Pode-se estipular uma base arbitrária da forma [BASE#]N, que significa N na base BASE. Exemplo:

$$123 = 0x7b = 0173 = 2\#1111011$$

$$10 = 16 = 8 = 2$$

Todos estes números são iguais mas em diferentes bases numéricas.

7.8 Exercícios

1. Altere o prompt para que tenha as seguintes formas:
 - (a) (usuário-hora)
 - (b) [usuario@nome da maquina]
 - (c) [versao do bash - nome da maquina - usuario]>
 - (d) c:\>
2. Crie um script que execute o seguinte algoritmo:
 1. Tenta abrir um arquivo de nome cont;
 2. Se arquivo cont existir vai para passo 4;
 3. Cria o arquivo cont;
 4. Se existir algum número no arquivo vai para o passo 6;
 5. Joga 0 no arquivo cont;
 6. Incrementa o número contido em cont;
 7. Grava o novo número em cont.
3. Utilizando os conceitos vistos nesse capítulo crie uma calculadora. A entrada de dados terá sempre <valor1> <operação> <valor2>. As operações são:

SOMA soma os dois termos;

SUB subtrai o segundo termo do primeiro;

MULT multiplica o primeiro pelo segundo;

DIV retorna a parte inteira da divisão do primeiro pelo segundo;

AND and binário entre os dois;

OR or binário entre os dois;

Na saída deve ser mostrado o valor1 a operação, valor2 um sinal de igual e o resultado, como no exemplo:

```
33 SUB 1 = 32
```

```
15 AND 16 = 0
```

Capítulo 8

Configuração do ambiente

A configuração do ambiente do sistema se baseia em vários arquivos que geralmente residem no diretório home do usuário e tem como nome algo do tipo “.nomedoprograma” ou “.nomeespecificodoarquivo”. Abaixo iremos analisar alguns dos arquivos mais importantes.

8.1 Configuração da shell BASH

A configuração da shell bash se dá a partir de dois arquivos: `.bash_profile` e `.bashrc`.

O primeiro arquivo é executado imediatamente após o usuário logar-se no sistema, e contém configurações que sejam úteis para aquela seção.

Já o segundo é executado todas as vezes que uma nova shell for iniciada. Ou seja, se, após o logon, você executar o programa “bash”, o arquivo `.bashrc` será lido e seus comandos serão executados.

As duas próximas seções se dedicam a explicar mais detalhadamente um exemplo desses arquivos de configuração.

8.1.1 `.bashprofile`

```
#####  
# Exemplo de arquivo .bash_profile  
# Executado uma vez a cada login  
# PET Informática - UFPR  
#####  
  
# lê a configuração padrão do sistema  
if [ -f /etc/profile ]; then  
source /etc/profile;  
fi  
  
# Faz com que a linha de comando do bash se comporte no estilo vi, e nao
```

```

# emacs (um pouco mais difícil de usar, mas mais poderoso)
set -o vi

# Prompt: [user@host diretorio]$
PS1="[\\u@\\h \\W]\\\$ "

# variáveis diversas (veja man bash)
PATH=$HOME/bin:$PATH
USER='id -un'
LOGNAME=$USER
MAILDIR=$HOME/mail
MAIL="$HOME/Mailbox"
TMOUT=3600
HISTSIZE=10000
HISTFILESIZE=10000
VISUAL=/usr/bin/vim
EDITOR=/usr/bin/vim
HOSTNAME='/bin/hostname'
export PATH USER LOGNAME MAILDIR MAIL TMOUT HISTSIZE HISTFILESIZE
export VISUAL EDITOR HOSTNAME

# não permite a criação de arquivos de core
ulimit -c 0

# lê a listagem de cores para o ls
if [ -f $HOME/etc/DIR_COLORS ]; then
eval 'dircolors -b $HOME/etc/DIR_COLORS';
fi

```

8.1.2 .bashrc

```

#####
# Exemplo de arquivo .bashrc
# Executado a cada nova shell aberta
# PET Informática - UFPR
#####

# lê a configuração padrão do sistema
if [ -f /etc/bashrc ]; then
source /etc/bashrc;
fi

# Aliases
# # # #
alias l="/bin/ls -l -F --color=tty --sort=ext"
alias ll="/bin/ls -lF --color=tty --sort=ext"
alias la="ls -laF --color=tty --sort=ext"
alias rm="rm -i"
alias cp="cp -i"

```

```
alias mv="mv -i"
alias vi=vim

# Funções
# # # # #
# Monta o floppy, mostra seu conteúdo e desmonta
function lsfloppy() {
mount /floppy;
ls /floppy -lF --color;
umount /floppy;
}; export -f lsfloppy

# Executa um find de modo rápido.
# uso: ffind <expressao>
function ffind() {
    local a="$1"; shift; find . -name $a -print "$@"
}; export -f ffind
```

8.2 Recebimento de e-mails

8.2.1 .qmail

Às vezes deseja-se redirecionar os mails que chegam para outros endereços, para isso usa-se o arquivo *.qmail*, que deve ficar no home do usuário.

Seguem abaixo dois exemplos comentados:

```
# .qmail
# arquivo de redirecionamento de mails

&joao@email.com.br # redireciona para o endereço citado
./Mailbox           # deixa uma cópia na mailbox do usuário
```

ou então:

```
# .qmail
# arquivo de redirecionamento de mails

| preline procmail # redireciona para o filtro de mail procmail
```

8.2.2 .procmailrc

Alguns usuários costumam receber muitos mails diariamente, e para fazer-se a filtragem destes mails de maneira que eles sejam armazenados em

arquivos separados (isso é particularmente útil para quem assina muitas listas de discussão), faz-se uso do utilitário *procmail*, que lê o arquivo *.procmailrc*, que tem o seguinte formato:

```
# primeira ação
:0:
* condição 1
[* condição 2]
  ...
[* condição n]
<exatamente uma ação>
```

...

```
# ação n
:0:
* condição 1
[* condição 2]
  ...
[* condição n]
<exatamente uma ação>
```

A ação pode enviar o mail para alguém, colocar o mail na entrada padrão de um programa, ou enviar o conteúdo do mail para um arquivo, com outros mails que apresentam as mesmas características garantidas pelas condições impostas no *.procmailrc*.

Exemplo:

```
!alguem@mail.dominio # envia o mail para o endereço alguem@mail.dominio
| sort                # ordena as linhas do mail
/tmp/arq              # coloca o conteúdo do mail no arquivo /tmp/arq
```

Deve-se deixar este arquivo no *HOME* do usuário em questão. Segue abaixo um exemplo de arquivo *.procmailrc* para um usuário chamado João. Na sequência algumas considerações após este exemplo.

```
# Configuração do procmail - .procmailrc
# João da Silva. <joao@inf.ufpr.br>
# 22/02/2000
```

```
# DEFINIÇÃO DE ALGUMAS VARIÁVEIS PARA O PROCMAIL
```

```

PATH=/bin:/usr/bin:/usr/local/bin
MAILDIR=$HOME/mail
LOGFILE=$HOME/var/procmail.log

# INICIO DO REDIRECIONAMENTO DE MENSAGENS

# MENSAGENS DO SISTEMA DESCARTÁVEIS
# mensagens que contêm no subject a palavra Excess, ou foram enviadas
# pelo root
# =====
:0:
* ^Subject.*Excess*
* ^From.root
/dev/null

# LISTAS DE DISCUSSÃO
# mensagens que foram enviadas para o endereço lista-de-discussao@egroups.com
# =====
:0:
* ^T0lista-de-discussao@egroups.com
listas/lista-de-discussao

# PESSOAL
# mensagens enviadas para o endereço joao@qualquercoisa
# =====

:0:
* ^T0joao@qualquercoisa
ufpr/pessoal

# MENSAGENS IMPORTANTES DO SISTEMA
# =====
:0:
* ^Subject:.Cron <joao@optimus>
sistema/cron_daemon

:0:
* ^From.maria@inf.ufpr.br
pessoal/maria

```

8.3 Configuração do Ambiente Gráfico

A configuração do ambiente gráfico é diretamente dependente do gerenciador de janelas utilizado. Atualmente existem inúmeros gerenciadores disponíveis para a plataforma linux, e sua configuração se mostra bastante fácil.

Além da configuração do gerenciador de janelas, é possível configurar

alguns detalhes do ambiente através de alguns arquivos de configuração residentes no diretório `home` do usuário. Esse arquivos permitem a configuração de parâmetros como cor de fundo e programas a serem inicializados automaticamente.

8.3.1 `.xsession`

O arquivo `.xsession` serve para configurar a área de trabalho no uso do modo gráfico. Nele ficam as configurado as opções do usuário. Se não houver um arquivo `xsession` o ambiente gráfico será o default, que é o configurado no sistema.

Neste arquivo pode-se escolher o gerenciador de janelas entre os disponíveis (`fvwm2`, `fvwm`, `wmaker`, `kde`, `gnome`...) que dará aparência à tela. Os programas que são inicializados toda vez que é feito o login também podem ser incluídos no arquivo.

Esses programas podem ser : editores de texto, `netscape`, `xclock`, `xterm`, `xman`. Em alguns programas e na imagem de fundo podemos definir as cores .

```
# cor verde no fundo
xsetroot -solid LimeGreen

# xterm com um top
xterm -geometry 29x30+831+191 -e top &

#emacs iconizado
emacs -iconic -fg white -bg SeaGreen&

# xterm verde
xterm -geometry 29x30x831x191 -fg white -bg SeaGreen&

#xman iconizado
xman -iconic&

#programas

netscape &
emacs &

#relogio digital
xclock -digital -geometry 181x25+831+0 -fg white -bg LimeGreen&

exec fvwm2
```

O texto acima é um exemplo de arquivo *.xsession*. Para configurar a cor de fundo da área de trabalho usa-se o comando *xsetroot*. As opções de cores podem ser visualizadas usando *xcolors*. Para escolher as cores de programas como *emacs* e *xclock* foi feito: *-fg* para setar a cor das bordas e *-bg* para setar a cor do fundo da janela do programa. A opção *-e* faz a *xterm* executar um programa. A opção *-iconic* faz a aplicação aparecer iconizada.

O gerenciador de janelas escolhido foi o *fvwm2*. Para os programas serem inicializados a partir do *.xsession* basta fazer a chamada em background. A área de trabalho utilizando o arquivo acima ficará com a cor de fundo verde, com um programa *top* rodando numa *xterm*, um *emacs* iconizado, um *xterm* verde na geometria indicada por 29x30x831x191, um *xman* iconizado, um *netscape* e um *emacs* que serão inicializados.

Haverá também um relógio digital *xclock -digital*.

8.4 Exercícios

1. Crie um *.bashrc* com as seguintes características:
 - coloque o diretório */usr/local/bin* na variável *\$PATH*;
 - o comando *m_nfs* monta uma partição do tipo NFS que está no ponto de montagem */servidor* do servidor passado como argumento, no diretório local */mnt/nfs*;
 - o comando *rec* abre uma nova shell;
 - coloca o *vi* como editor padrão;
 - se o terminal ficar 20 segundo inativo, será fechado;
 - no prompt aparece um ponto seguido por um espaço.
2. Crie um *.qmail* que envie as mensagens para *teste@mail.dominio*, *maisum@mail.dominio*, *lista@listas.mail* e copie as mensagem para os arquivos */tmp/arq1*, */tmp/arq2*, */tmp/lixo*.
3. Crie um *.procmailrc* que jogue todas as mensagens que tenha no subject a mensagem *subscribe* na entrada padrão do programa */tmp/lista_insc*, as mensagens que tenha no subject a mensagem *unsubscribe* na entrada padrão do programa */tmp/lista_uninsc* e as mensagens com outros subject devem ser enviadas para o mail *aki@mail.dominio*.
4. Baseado nos dois exercícios anteriores crie uma "lista de discussão". Quando alguém enviar um mail com *subscribe* no subject, se existir

um endereço de mail* no corpo da mensagem, esse será acrescentado no arquivo .qmail. Para que alguém se desinscreva da lista deve enviar um mail com unsubscribe no subject e com seu endereço no corpo da mensagem.

5. Configure seu .xsession de modo que seu gerenciador de janelas seja o fvwm2, e a cada vez que logar execute um xterm e um netscape.

*cuidado!