APOSTILA DE SISTEMAS OPERACIONAIS - II
LABORATÓRIO

UNIX

PROGRAMAÇÃO SHELL

Prof. Renato Jensen

ÍNDICE

I – EDITOR DE TEXTO

1 - DEFINIÇÃO

Os editores de texto permitem a criação, edição e visualização de arquivos no formato texto.

2 - ALGUNS EDITORES

Existem vários editores de texto para o UNIX.

O **vi** é o editor de textos padrão do UNIX. Não existe um UNIX que não tenha o **vi**.

O **joe** é um editor de texto que foi lançado no Linux, para fazer companhia ao **vi** e oferecer um pouco das interfaces amigáveis dos editores do DOS. O **joe** é semelhante ao WordStar. Todos os comandos para manipulação de textos são idênticos aos deste editor para DOS.

Outro editor de textos é o ***pico*** é um editor de texto simples, fácil de usar, com o *layout* bem parecido ao editor de *mail* ***pine***. Ambos foram desenvolvidos na Universidade de Washington.

2.1 - VI

O editor **vi** inicializa em Modo de Comando.

A inicialização do **vi** é da forma:

$ **vi** [-R -r [*arquivo*]] [*arquivo*]

-R	entra no **vi** em modo de leitura, não permitindo modificar o arquivo.
-r [*arquivo*]	recupera a última edição salva de um arquivo antes de um *crash*.

- ***Comandos Gerais***

Esc	termina o modo de inserção
n	muitos comandos permitem que se digite um inteiro *n* antes dele, de modo que o comando seja executado *n* vezes
:set	permite definir: nu numeração ts= tabs prefixando um atributo com "no" irá desativá-lo

- ***Controle de Páginas***

^E	rola a tela para baixo linha a linha
^D	rola a tela para baixo meia tela
^F	rola a tela para baixo uma tela inteira
^Y	rola a tela para cima por uma linha
^U	rola a tela para cima meia tela
^B	rola a tela para cima uma tela inteira

obs.: ^ = Ctrl

- ***Movimento do Cursor***

h	move o cursor uma posição a esquerda
l	move o cursor uma posição a direita
k	move o cursor uma linha acima
j	move o cursor uma linha abaixo
H	move o cursor para o topo da tela
L	move o cursor para o final da tela
G	move o cursor para o final do arquivo

- ***Modificação de Texto***

r*x*	substitui o caracter sobre o cursor por *x*
dd	apaga a linha corrente
dw	apaga a palavra sob o cursor
x	apaga o caracter sobre o cursor

- ***Gravando e Saindo***

:w [*arquivo*]	grava o arquivo editado (grava com *arquivo*, se especificado)
:wq	grava o arquivo e sai
:q	sai (se o arquivo foi modificado mostra mensagem de advertência)
:q!	sai sem salvar as modificações

- ***Modo de Inserção***

a	entra no modo de inserção e posiciona o cursor uma posição na frente da atual
A	entra no modo de inserção e posiciona o cursor no final da linha
i	entra no modo de inserção e não altera a posição do cursor
I	entra no modo de inserção e posiciona o cursor no início da linha
o	entra no modo de inserção e adiciona uma linha em branco abaixo da linha atual
O	entra no modo de inserção e adiciona uma linha em branco acima da linha atual

obs.: a tecla **Esc** sai do modo de inserção e entra no modo de comando

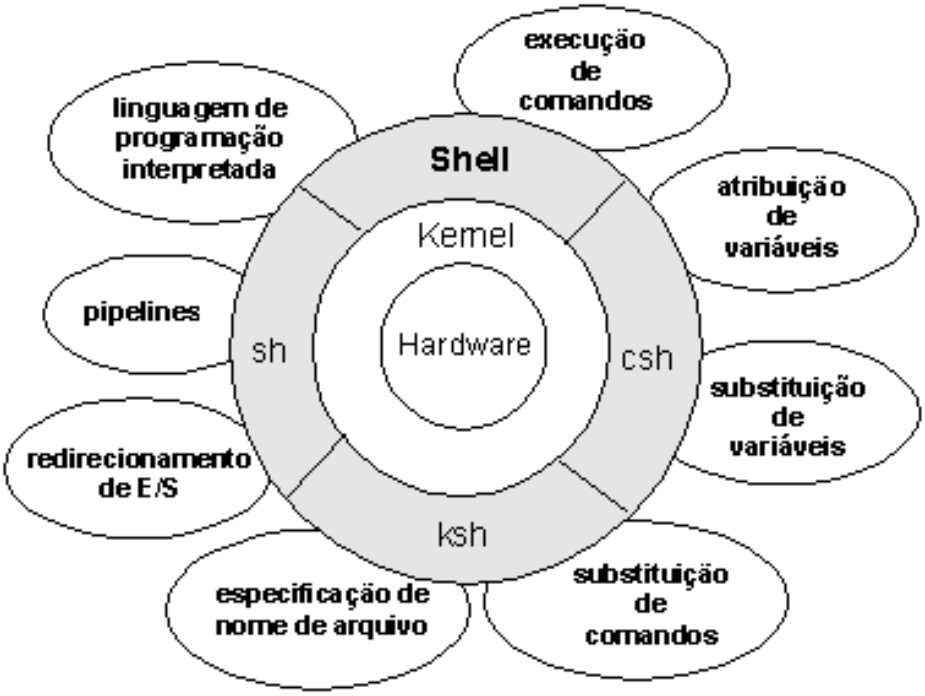
II - INTRODUÇÃO AO SHELL

1 - DEFINIÇÃO

O shell é um programa que serve como um interpretador de comandos. É separado do sistema operacional, que fornece ao usuário a facilidade de selecionar a interface mais apropriada às suas necessidades, ou seja, escolher qual shell ele gostaria de trabalhar. A função do shell é permitir que você digite seu comando para realizar várias funções e passar o comando interpretado para o sistema operacional.

O shell oferece várias funcionalidades, das quais podemos destacar:

- suporte a interface programável interpretativa (testes de condição, desvios, loops)
- substituição de valores em variáveis shell especificadas
- substituição de comandos
- suporte a redirecionamentos e pipelines
- pesquisa a comandos com execução do programa associado



Ao inicializar uma sessão UNIX, o shell define as características para o terminal e envia um prompt. Dentre os Shells mais conhecidos pode-se citar:

- **sh** ou **bash** Bourne Shell - o mais tradicional. (prompt: $)
- **ksh** Korn Shell - o mais usado atualmente. (prompt: $)
- **csh** C Shell - considerado o mais poderoso. (prompt: %)
- **rsh** Remote Shell - shell remoto.
- **Rsh** Restricted Shell - versão restrita do *sh*.
- **Pdksh** Public domain Korn Shell - versão de domínio público do *ksh*.
- **Zsh** Z Shell - compatível com o *sh*.
- **Tcsh** versão padronizada do *csh*.

O C shell foi escrito depois do Bourne shell e oferece um ambiente mais poderoso. Tem histórico, uso do alias, complementação do nome de arquivo e outras características úteis. Oferece ainda um conjunto mais poderoso de comandos, com mais opções e capacidades do que o Bourne shell.

Um dos problemas do C shell é a incompatibilidade com programas do Bourne shell. Além disso, existe a dificuldade de realizar algumas tarefas, como redirecionamento de erro padrão.

O Korn shell tem todas as características melhores do C shell, além do que pode fazer o que o Bourne shell faz. De fato, o K shell é um superconjunto do Bourne shell.

Shell **Atributo**	**csh**	**ksh**	**sh**
tamanho	grande	médio	pequeno
velocidade	lenta	rápida	média
características extras	algumas	todas	nenhuma
disponibilidade	muitos sistemas	muitos sistemas e expandido	todos os sistemas
compatibilidade	alguns sh todos csh	todos sh todos ksh	somente sh

Comparação do csh, ksh e sh

2 - VARIÁVEIS

Uma variável shell é semelhante a uma variável em álgebra. É seu nome que representa um valor. Todas as variáveis shell, por *default*, são inicializadas com NULL (nada). O valor da variável pode ser modificado a qualquer momento que se desejar. O valor da variável pode ser acessado fazendo referência ao nome da variável.

2.1 - Armazenamento

O Shell possui 2 áreas em memória para as suas variáveis: **área de dados local** e de **ambiente**. A memória é alocada na área de dados local quando uma nova variável é definida. As variáveis nesta área são restritas ao shell corrente. Isto é, nem todo subprocesso terá acesso a estas variáveis. Entretanto, as variáveis que são movidas através do ambiente podem ser acessadas por qualquer subprocessos.

Existem variáveis especiais do shell que são definidas através do processo de *login*. Estas variáveis são armazenadas no ambiente, e seus valores podem mudar para caracterizar a sessão. As principais variáveis são:

HOME	define o diretório inicial do usuário;
LOGNAME ou USER	define a identificação do usuário no login
TERM	define o tipo do terminal;
PATH	define o(s) diretório(s) onde procurar os comandos para execução;

PWD	define o diretório corrente;
SHELL	define o shell *default* para a sessão no terminal
PS1	define o prompt do shell primário
PS2	define o prompt do shell secundário

Os nomes das variáveis locais são geralmente definidos usando-se caracteres minúsculos e das variáveis ambientais são definidos com caracteres maiúsculos. Esta convenção é só para facilitar a identificação entre os 2 tipos de variáveis no código de um programa.

A manipulação dessas variáveis depende do Shell que está se utilizando. O C-Shell realiza uma divisão clara entre variáveis do Shell e de ambiente. Para criar ou trocar o valor de uma variável **local** utiliza-se o comando **set**. Para as variáveis de **ambiente**, o comando **setenv**.

set [variável = valor]

setenv [variável valor]

A remoção de variáveis do Shell e do ambiente é realizada através dos comandos **unset** e **unsetenv**, respectivamente.

unset variável

unsetenv variável

2.2 - Atribuição de Valores

A atribuição de valores às variáveis permite associar um valor ao nome da variável. O seu valor pode ser acessado através do nome da variável. Por exemplo, um contador que conta o número de interações através de um loop. A variável pode ser incrementada de um cada vez que você completa o loop. Ao atribuir um valor a uma nova variável, este será armazenado na área de dados local.

A atribuição pode ser feita dentro de um programa shell ou digitando-se diretamente no prompt do shell:

```
$ color=blue                    (variável local)
$ count=5                       (variável local)
$ dir_name=/home/ricardo        (variável local)
$ PATH=.:/bin:/usr/bin          (variável ambiental)
```

2.3 - Referenciando Variáveis

Cada variável que é definida será associada a um valor. Quando o nome da variável for imediatamente precedido por um sinal **$**, o shell troca o parâmetro pelo valor da variável. Estes procedimento é conhecido como substituição de variável e sempre ocorre antes do comando ser executado.

Depois do shell fazer todas as substituições na linha do comando, ele executa o comando. Assim, variáveis podem também representar um comando, argumentos de comandos ou uma linha de comando completa. Isto fornece um mecanismo conveniente para usar um pseudônimo freqüentemente utilizado para longos caminhos e longas cadeias de comandos.

```
$ echo$color
blue

$ echo o valor de color é $color
o valor de color é blue

$ echo $PATH
.:bin:/usr/bin

$ file_name=$PATH/arquivo.txt
$ more $file_name
```

O comando **echo** $ *name* exibe o valor corrente de uma variável.

III - PROGRAMAÇÃO DO SHELL

O Shell é um interpretador de comandos. Quando você quer executar muitas vezes uma série de comandos, é conveniente salvar estes comandos em um programa shell e executar os comandos invocando o script shell.

Este capítulo apresenta métodos alternativos de passar informação para seu programa shell, de modo que o programa do usuário a ser executado não precise saber como definir e exportar variáveis shell.

1 - ARGUMENTOS NA PROGRAMAÇÃO SHELL

Da mesma forma que os comandos, programas também se utilizam de argumentos na linha de comando, através de variáveis especiais que são definidas relativamente à sua posição do argumento na linha de comando.

Você pode desenvolver seu programa para aceitar argumentos na linha de comando e assim, passar nomes de arquivos ou diretórios para o seu utilitário manipular com os comandos do sistema UNIX. Você também pode definir opções na linha de comando para estender as capacidades do seu programa shell.

A definição das variáveis do programa está associada à posição do argumento na linha de comando. O nome das variáveis são numeradas de 0 a 9. O valor destas variáveis é acessado da mesma forma que qualquer valor de outra variável é acessado, prefixando o nome como símbolo $. Portanto, para acessar os argumentos da linha de comando no seu programa shell, você deve se referir aos valores **$0** a **$9**. O argumento $0 sempre terá o nome do programa ou do comando. Você pode acessar somente 9 argumentos ($1 a $9) num dado instante, mas a linha do comando pode ter mais que 9 argumentos.

$ contabilizar arg1, arg2, arg3
$0 $1 $2 $3

2 - VARIÁVEIS ESPECIAIS

Em algumas situações é necessário criar um programa shell para aceitar argumentos variáveis na linha de comando. As variáveis especiais do shell **#** e ***** oferecem esta flexibilidade:

$# - contém o número de argumentos

$* - contém a lista inteira de argumentos. O comando ($0) nunca é incluído nesta lista.

Sendo **Prog1**

$ Prog1 azul amarelo vermelho preto

echo Existem $# argumentos na linha de comando	Existem 4 argumentos na linha de comando
echo São eles: $*	São eles: azul amarelo vermelho preto
echo O primeiro argumento é $1	O primeiro argumento é azul

3 - COMANDOS BÁSICOS

3.1 - READ

Sintaxe **read** variável

Descrição: o comando **read** especifica uma variável , cujo valor será designado às palavras que serão fornecidas pelo usuário no prompt durante a execução do programa. Uma vez designada, você pode acessar esta variável como qualquer outra variável do shell.

Normalmente você vai querer fornecer um prompt para o usuário com o comando **echo**, para que ele saiba que o programa está esperando por dados, e informar ao usuário que tipo de dados está esperando. Assim, cada comando **read** deveria estar precedido por um comando **echo**.

Opções: variável - nome da variável que vai receber o dado a ser digitado.

Exemplo: Sendo **Prog3**:

```
echo "Forneca o nome do diretorio"
read    nomedodiretorio
ls –l  nomedodiretorio
```

3.2 - ECHO

Sintaxe **echo** [texto]

Exemplo:
```
$ echo "Bom dia"
Bom dia

$ echo $SHELL
/bin/bash
```

3.3 - TEST

Sintaxe **test** expressão ou **[** expressão **]**

Descrição: o comando **test** é usado para avaliar expressões e gerar um código de retorno. O código de retorno não é enviado para a saída padrão, ou seja, deve ser atribuído para alguma variável. O código de retorno será dependerá do resultado da expressão, podendo assumir **0** (verdadeiro) ou **1** (falsa).

o comando **test** pode ser usado sozinho, mas é mais comum usá-lo com as intruções **if** e **while**.

nota: o **[** expressão **]** deve ter espaço em branco em volta deles.

a variável **$?** informa o código de retorno do último comando executado.

Opções: expressão - qualquer expressão que envolva os operadores descritos abaixo:

Exemplo:
```
$ x=xyz
$ test "$x"  !=  "xyz"
$ echo $?
1

$ idade=30
$ test idade  -le  10  ||  test idade  -ge  25
$ echo $?
0
```

OPERADORES DE TESTE COMUNS

Condição	Verdadeira se ...
-f arquivo	existir arquivo
-d diretório	existir o diretório
cad1 = cad2	cadeia 1 igual a cadeia 2
cad1 ! = cad2	cadeia 1 diferente da cadeia 2

COMPARAÇÕES NUMÉRICAS

-eq	*(equal)*	***igual***
-ne	*(not equal)*	***diferente***
-gt	*(greater than)*	***maior que***
-ge	*(greater or equal)*	***maior ou igual a***
-lt	*(less than)*	***menor que***
-le	*(less or equal)*	***menor ou igual a***
&&	*(AND)*	***E***
||	*(OU)*	***OU***

3.4 - EXPR

Sintaxe **expr** expressão

Descrição: o comando **expr** executa uma expressão aritmética e retorna o resultado da mesma.

Opções: expressão - qualquer expressão aritmética

Operadores aritméticos

+	Adição
-	Subtração
*****	Multiplicação
/	Divisão
%	Resto de divisão

Exemplos:

```
$ expr 3 + 1
4
```

```
$ x=3
$ x= `expr $x + 1`          (comando expr está entre crases)
$ echo $x
4
```

3.5 - EXIT

Sintaxe **exit** [argumento]

Descrição: o comando **exit** termina a execução de um programa shell e define o código de retorno. Se não tiver argumento, o código de retorno é definido para o código de retorno do último comando executado antes do comando **exit**.

Opções: argumento - código de retorno

Exemplo: Sendo **prog4**:

```
echo  Saindo do programa prog4
exit 77
```

Executanto **prog4** temos:

```
$ prog4
saindo do programa prog4
$ echo $?
77
```

4 - COMANDOS DE DESVIOS

4.1 - IF-ELSE

Sintaxe

```
if  expressão
then
        lista A de comandos
else
        lista B de comandos
fi
```

Descrição: a construção **if-else** fornece o controle de fluxo para um programa baseado no código de retonro (?) de um comando. Se o código de retorno da expressão for 0 (verdadeiro), uma lista A de comandos especificados será executada, se for diferente de 0 (falso) então a lista B de comandos será executada.

Opções: expressão - qualquer expressão. Deve-se utilizar o comando **test** na expressão.

Exemplo: Sendo **prog6**:

```
if  test  $Cont  -lt  2
then
        echo "Foi fornecido valor menor que 2:    $Cont"
else
        echo "Foi fornecido valor maior ou igual a 2:    $Cont"
fi
```

4.2 - CASE

Sintaxe

```
case variável in
padrão 1)        lista A de comandos
                        ;;
padrão 2)        lista B de comandos
                        ;;
padrão n)        lista N de comandos
                        ;;
esac
```

Descrição: a construção **case** fornece uma sintaxe conveniente para desvios múltiplos. O desvio selecionado é baseado na comparação sequencial de uma palavra com o padrão fornecido. Estas comparações são baseadas em cadeias. Quando uma coincidência é encontrada, a lista de comandos correspondente é executada. Cada lista de comandos termina com dois ponto-e-vírgulas (;;). Depois de terminar de executar a lista de comandos, o controle do programa continua até **esac**.

Opções: variável - refere-se ao valor da variável shell.

padrão - valor a ser testado na variável.

Exemplo:

```
case $RESPOSTA in
    "sim")   echo  OK  ;;
    "Nao")   echo  NAO PODE IR ;;
    "talvez")   echo  'PODE-SE NEGOCIAR ;;
```

5 - COMANDOS DE LAÇOS

5.1 - WHILE

Sintaxe

```
while condições
do
    ações
done
```

Descrição: a construção **while** permite que seja executada uma série de ações enquanto as condições forem verdadeiras.

Exemplo:

```
while test $cont  –le  10
do
   echo "Valor: $cont"
   cont=`expr $cont  +  1'
done
```

5.2 - UNTIL

Sintaxe

```
until condições
do
    ações
done
```

Descrição: a construção **until** é semelhante a **while**, diferenciando-se pelo fato de executar uma vez, antes de realizar o teste das condições. Executa as ações enquanto as condições forem falsas, isto é, até que as condições sejam verdadeiras.

5.3 - BREAK

Sintaxe **Break**

Descrição: o comando **break** permite que o usuário interrompa um laço.

Exemplo:

```
while test $x = "SIM"
do
  if test $valor –eq 999
  then
        break
  else
        echo Continua o processamento
  fi
done
```

5.4 - CONTINUE

Sintaxe **Continue**

Descrição: o comando **continue** permite que a lógica volte ao início do laço sem que para isso seja processado algo adicional.

Exemplo:

```
while test $x = "SIM"
do
  echo "Entre com o nome do arquivo"
  read nomearq
        if test -f $nomearq
        then
              processa $nomearq
        else
              continue
        fi
  echo "Processando arquivo $nomearq"
done
```