EA872 Laboratório de Programação de Software Básico
Atividade 1

1. Tema

Interpretador de comandos de um sistema operacional.

2. Objetivos

- Familiarização com o ambiente UNIX no contexto de programação de comandos.
- Introdução aos *shells* do sistema UNIX.
- Introdução à linguagem de programação do *Bourne shell* do UNIX.
- Implementação de novos comandos.

3. Conceitos de sistema operacional

O sistema operacional deve fornecer, entre outros, os seguintes recursos para possibilitar a operação normal de um computador:

- gerência de memória;
- controle e gerência de unidades de disco;
- carregamento e execução de programas;
- atendimento de requisições de programas em execução;
- comunicação com o usuário.

Para tais funções, o sistema operacional conta com dois componentes básicos: o núcleo e o interpretador de comandos.

3.1. Núcleo do sistema operacional

O núcleo (*kernel*) contém as rotinas básicas do sistema operacional, responsáveis pela operação do sistema no nível de máquina e pelas conexões com os dispositivos de hardware.

As funções do núcleo são de dois tipos: autônomas e não-autônomas. Alocação de memória e de CPU são exemplos de funções autônomas, pois são executadas pelo núcleo sem serem requisitadas explicitamente pelos processos do usuário. Por outro lado, alocação de recursos e criação de processos são requisitados pelos processos do usuário através de chamadas ao sistema (*system calls*). Exemplos de chamadas ao sistema incluem: *fork*, *exec*, *kill*, *open*, *read*, *write*, *close* e *exit*.

3.2. Interpretador de comandos

O interpretador de comandos é o programa que implementa a interface do sistema operacional com o usuário. Este programa é projetado para facilitar o acesso do usuário ao potencial do sistema operacional, sem a necessidade de comunicação direta com o núcleo.

No UNIX, o interpretador de comandos é fornecido por um programa denominado *shell*. O núcleo e o *shell* do sistema operacional UNIX se relacionam com os utilitários, o hardware e o usuário de acordo com o esquema apresentado na Figura 1.

4. *Shells* do Sistema UNIX

O processo *shell* pode ser chamado automaticamente durante o *login* ou manualmente através da entrada do nome do processo pelo teclado. Independente da forma como é chamado, ele trabalha na seguinte sequência:

Copyright © 2015 DCA/FEEC/UNICAMP

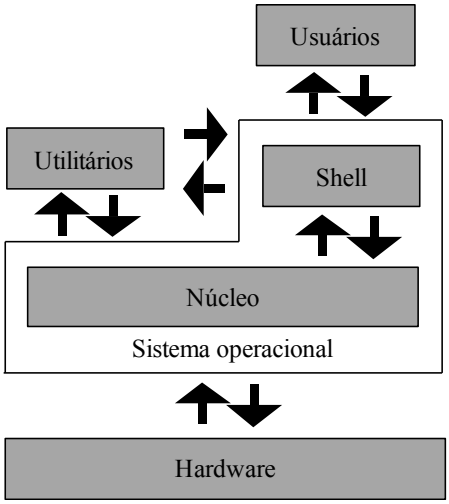

Figura 1 - Anatomia do UNIX

1. lê as informações de iniciação do ambiente *shell* (tipo de terminal, tipo de *prompt*, os caminhos de busca em diretórios, etc.) a partir dos seguintes arquivos: *.profile*, *.login* e *.cshrc*, que se encontram no diretório-raiz do usuário e no diretório */etc* do sistema;
2. o sinal de *prompt* é apresentado e o processo aguarda um comando do usuário (lembre-se que o UNIX diferencia letras maiúsculas de minúsculas);
3. se o usuário entrar com *ctrl-d*, o processo é terminado; caso contrário, executa o comando do usuário e retorna ao passo (2).

Normalmente encontram-se instalados no sistema UNIX pelo menos um dos seguintes tipos de *shells*:

- *Bourne Shell* (*sh*): escrito por Steve Bourne (Bell Labs) e disponível em praticamente todos os sistemas UNIX e Linux. Foi o primeiro *shell* popular do UNIX e é considerado o *shell* padrão. No entanto, ele não apresenta muitos dos recursos interativos disponíveis no *C shell* e *Korn shell*, mas fornece uma linguagem muito fácil de usar na escrita de *shell scripts*. Uma nova versão do mesmo chamada *Bash* (*Bourne-again shell*) contempla os melhores recursos do *C shell* e do *Korn shell*, além de procurar ser compatível com o *sh* e com a norma de padronização de UNIX (POSIX).
- *C Shell* (*csh*): escrito na *University of California at Berkeley*, segue o mesmo conceito do *Bourne shell*, mas os seus comandos obedecem a uma sintaxe similar à da linguagem C (daí o seu nome).
- *Korn Shell (ksh)*: escrito por David Korn (Bell Labs). Ele fornece todas as características do *C shell*, juntamente com uma linguagem de programação para *shell scripts* similar ao *Bourne shell* original. Portanto, ele é uma extensão do *Bourne shell*, com aperfeiçoamentos no controle de tarefas, na edição de linhas de comando e na linguagem de programação. É o mais poderoso dos três, e é fornecido como o *shell* padrão em alguns sistemas UNIX.

Uma vez disponíveis, é possível utilizar qualquer um destes *shells,* de acordo com a preferência do usuário.

Destacam-se as seguintes facilidades oferecidas pelos *shells* do sistema UNIX:

- interpretar uma sequência de comandos concatenados por palavras reservadas;
- iniciar a execução de comandos ou códigos executáveis;
- redirecionar a entrada e a saída dos processos;
- concatenar uma sequência de comandos num arquivo, denominado *shell script* (procedimento de *shell*), de modo que, para executar a tal sequência, é só entrar o nome do arquivo (o qual deve estar com permissão para execução);
- suportar a execução de vários processos concorrentes em *background*;
- criar novos *shells*-filho ou *subshells* para executar, por exemplo, os processos em *background*;
- suportar variáveis locais e globais para, por exemplo, definir e acessar de forma flexível as características do ambiente de um *shell*;
- admitir o uso de metacaracteres nas linhas de comando para mapear um conjunto de arquivos;
- concatenar o fluxo de dados entre os processos através do *pipe* (representado pela barra vertical);
- substituir os comandos pelo seu resultado.

Copyright © 2015 DCA/FEEC/UNICAMP

5. Linguagem de Programação do *Bourne Shell*

Vários fatores justificam a adoção do *Bourne shell* (*sh*) neste curso:

- o *Bourne shell* foi o primeiro interpretador de comandos a ganhar popularidade, sendo que muitos *scripts* foram escritos na sua linguagem;
- o *Bourne shell* é um subconjunto do *Korn shell*, um interpretador que tem bastante popularidade. O conhecimento da sua sintaxe vai certamente ajudar a compreender o *Korn shell;*
- uma variante mais completa do *Bourne shell* conhecida como *bash* também é muito popular e é adotada como padrão e distribuições Linux importantes, como o Ubuntu.

Conforme já mencionado, um *shell* lê tanto os comandos do terminal, arquivo-padrão de entrada, como os comandos de um *script* (arquivo com comandos definido pelo usuário).

Obs:
- adotaremos a convenção de sublinhar aquilo que é digitado pelo usuário;
- podemos usar *bash* no lugar de *sh* nos exemplos mostrados a seguir;
- os arquivos de script descritos neste roteiro estão disponíveis na página da disciplina.

Exemplo 1 *Seguem alguns exemplos de comandos que podem ser acionados a partir do shell; experimente-os e veja quais são seus resultados.*

```
$ ps

$ ps u

$ ps aux

$ ps aux | grep seu_nome_de_usuário
```

Tarefa 1: consulte as páginas de manual (man ps) e explique o que o comando "ps aux" faz.

Exemplo 2 *Os seguintes blocos de comandos:*

```
$ ps aux | grep swapper
```

e

```
$ sh teste1 swapper (ou bash teste1 swapper)
```

onde o arquivo de script <teste1> contém a linha de comando:

```
ps aux | grep $1
```

são equivalentes. Os argumentos $1, $2, ... em <teste1> são os parâmetros posicionais a serem fornecidos ao processo sh*, como veremos mais adiante.*

O fluxo de execução dos comandos/programas num *script* não é necessariamente sequencial. O shell *sh* provê algumas primitivas de controle rudimentares que podem causar desvios condicionais ou incondicionais nesse fluxo. Outra flexibilidade oferecida pela linguagem de programação do *sh* é um conjunto de operações sobre variáveis.

5.1. Variáveis de Shell

Os nomes de variáveis válidos são concatenações de letras, dígitos e *underscore* (_), precedidas de uma letra. O operador que fornece acesso ao valor de uma variável é $. Com a operação de atribuição = pode-se atribuir um valor a uma variável. Caso a variável ainda não exista, ela é "criada" automaticamente.

Exemplo 3 *No seguinte bloco de comandos é criada uma variável <x> e atribuído a ela um valor:*

```
prompt> sh            (iniciar novo shell sh)
$ echo $x             (mostrar o conteúdo de x)
$ x='Hello world'     (alterar conteúdo de x)
$ echo $x             (mostrar o conteúdo de x)
Hello world           (conteúdo retornado)
$ ctrl-d              (terminar sessão de shell atual e voltar para shell
                       anterior)
prompt>
```

Copyright © 2015 DCA/FEEC/UNICAMP

Antes do comando de atribuição de <x>, o seu valor é indefinido, como mostra o primeiro comando echo*. O "prompt"* $ *indica que* sh *aguarda a entrada de um novo comando.*

O interpretador *sh* oferece a possibilidade de ler o valor de uma variável do arquivo de entrada padrão através do comando `read`.

Exemplo 4 *O seguinte arquivo* script <***teste2***> *demonstra o uso do comando* `read`.

```
#! /bin/sh          # ==> garante que o script será executado por /bin/sh
echo "Entre com o seu nome:"
read nome sobrenome
echo "Entre com o seu RA:"
read RA
echo "Seu nome e'" $nome
echo "Seu sobrenome e'" $sobrenome
echo "Seu RA e'" $RA
```

Ao entrarmos com o comando

```
$ sh teste2
```

teremos o seguinte resultado na tela:

```
Entre com o seu nome:
Jose Silva
Entre com o seu RA:
900000
Seu nome e' Jose
Seu sobrenome e' Silva
Seu RA e' 900000
```

Tarefa 2: experimente executar o script teste2 de duas formas: com o comando `sh teste2` **e depois entrando linha a linha em um prompt do shell. Que diferenças você nota no comportamento do script e nos resultados comparando estas duas execuções. dos comandos linha a linha no prompt de comando com a execução via arquivo de script?**

Para garantir que o *shell script* sempre vai ser executado pelo *Bourne shell* padrão, a primeira linha do arquivo deve ser

```
#! /bin/sh
```

O interpretador *sh* mantém uma tabela de **variáveis de ambiente** para armazenar um conjunto de informações sobre o contexto em que ele é executado. Normalmente os **valores globais** dessas variáveis são atribuídos no momento de *login* e modificáveis **localmente** pelo comando de atribuição dentro de cada processo *sh*. Como ocorre com todas as variáveis definidas num *shell*, as modificações locais só afetarão os processos-filho se os valores forem **exportados** explicitamente pelo comando `export`.

Para listar todas as variáveis de ambiente de um *shell* podemos usar o comando `env`. A Tabela 1 mostra algumas dessas variáveis.

Tabela 1

Variável	Significado	Verificação no seu ambiente corrente
DISPLAY	identificação do terminal	echo $DISPLAY
EDITOR	caminho para seu editor *default*	echo $EDITOR
HOME	diretório no qual a árvore de arquivos privado de cada usuário está armazenada	echo $HOME
LOGNAME	nome de login do usuário	echo $LOGNAME
MAIL	arquivo de correio eletrônico. Quando este arquivo é modificado, recebe-se a mensagem "*you have mail*"	echo $MAIL
PATH	lista de diretórios que devem ser varridos para localizar os comandos	echo $PATH
PS1	"prompt" para a entrada de um novo comando	*(valor default é* $)
PS2	"prompt" para a continuação de um comando	*(valor default é* >)
SHELL	tipo de interpretador de comandos corrente	echo $SHELL
TERM	tipo do terminal	echo $TERM
USER	nome do usuário	echo $USER

Copyright © 2015 DCA/FEEC/UNICAMP

Tarefa 3: descubra e documente o valor atual de cada variável da tabela 1.

Quando *sh* lê os comandos de um arquivo criado pelo usuário, ele aceita, além do nome do arquivo (parâmetro posicional $0), nove argumentos referenciáveis no arquivo, que são os parâmetros posicionais $1, $2, … , $9. O *sh* reserva duas variáveis $* e $@ para designar todos os parâmetros posicionais, de $1 até $9, e a variável $#, para designar o número de parâmetros posicionais diferentes de $0, ou seja, o número de argumentos do comando.

Outras variáveis locais pré-definidas no *sh* são:

Tabela 2

Variável	Significado
$-	as opções entradas ao iniciar o processo *sh*, tais como <-x> e <-v>
$$	identificação do processo *sh* corrente (pid)
$?	valor retornado pelo último comando executado
$!	identificação do último processo (pid) executado em *background*

Tarefa 4: descubra e documente o valor atual das variáveis $$, $? e $! no seu shell.

Finalmente, *sh* oferece a possibilidade de proteger uma variável de manipulações indevidas através do uso do comando `readonly`.

Exemplo 5 *Este exemplo mostra o uso do comando* `readonly`.

```
$ x='Alo!'
$ y='Curso EA-872!'
$ echo $x $y
Alo! Curso EA-872!
$ readonly x y
$ readonly
readonly x
readonly y
$ x='Hello!'
x: is read only
```

5.2. Blocos de Controle

O *sh* suporta os seguintes blocos de controle:

- `if...then...else...fi` : equivalente ao comando `if...then...else...` da linguagem C.
- `if...then...elif...fi` : equivalente ao comando `if...then...else if...fi fi`
- `while...do...done` : equivalente ao comando `while` da linguagem C.
- `until...do...done` : é equivalente ao comando `do...until` da linguagem C.
- `for...do...done` : é equivalente ao comando `for` da linguagem C.
- `case...in...)...;;...)...;;esac` : equivalente ao comando `switch...case` da linguagem C.

5.3. Expressões em *sh*

A linguagem de *sh* suporta um conjunto de operadores para definir comandos mais complexos:

Atribuição: =

Redireção de Entrada e Saída: no *sh* existem diferentes formas para redirecionar os arquivos de entrada e saída padrão (Tab. 3).

Tarefa 5: crie 2 exemplos de redirecionamento de entrada e saída usando a tabela 3.

Copyright © 2015 DCA/FEEC/UNICAMP

Tabela 3

Instrução	**Significado**
> *arquivo*	redirecionar a saída-padrão para <*arquivo*>
>> *arquivo*	anexar dados da saída-padrão a <*arquivo*>
< *arquivo*	usar <*arquivo*> como entrada-padrão
p1 | p2	conectar a saída-padrão do processo p1 com a entrada-padrão do processo p2
n > *arquivo*	redirecionar a saída do arquivo de programa cujo descritor é <n> para <*arquivo*>
n >> *arquivo*	anexar a saída do arquivo de programa cujo descritor é <n> a <*arquivo*>
n > &m	concatenar a saída do programa cujo descritor é <n>, com o arquivo de descritor <m>
n < &m	concatenar a entrada do arquivo cujo descritor é <n>, com o arquivo de descritor <m>
<< c	aceitar caracteres da entrada-padrão até a sequência <c>, onde <c> é formada por caracteres quaisquer, sendo que caracteres especiais são interpretados
<< \c	equivalente a “<< c”, mas sem a interpretação de caracteres especiais
<< 'c'	equivalente a “<< \c”
<< "c"	equivalente a “<< 'c' ”, mas com a interpretação dos operadores $, '...' e \

Substituição de caracteres: *sh* reserva alguns caracteres para denotar um conjunto de caracteres ou um conjunto de sequências de caracteres (Tab. 4).

Tabela 4

Caracter	Interpretação	Exemplo
*	Qualquer sequência de caracteres incluindo a sequência nula	ls *
?	qualquer caracter	
[ ]	Qualquer caracter definido no domínio especificado	ls [a-j]*
a | b	opção alternativa entre as sequências <a> e <b>. Só é usado nos blocos de controle `case`	

Tarefa 6: crie, execute e documente 3 exemplos de uso de substituição de caracteres com *, ? e [ ] (um para cada).

Interpretação de caracteres: Para distinguir os caracteres com interpretações especiais dos caracteres comuns, *sh* dispõe dos seguintes mecanismos (Tab. 5):

Tabela 5

Notação	Interpretação	Exemplo
...	os caracteres são interpretados	echo caminhos = $PATH
'...'	os caracteres são tomados literalmente	echo 'caminhos = $PATH'
”...”	os caracteres são tomados literalmente, depois que os operadores $, ’...’ e \ forem interpretados	echo ”caminhos = $PATH”
`...`	os caracteres são tomados como um comando	echo `pwd`

Tarefa 7: execute os exemplos da tabela 5 e discuta as diferenças nos resultados.

Substituição de Variáveis: quando o valor de uma variável não é setado, então ele assume a sequência nula. *sh* dispõe, entretanto, de operadores adicionais para substituir o valor das variáveis (Tab. 6).

Tabela 6

Notação	Interpretação	Exemplo
$var $var=	se o valor de <var> não é setado, ele assume a sequência nula	
${var:-op}	se o valor de <var> não é setado, o valor *default* assumido é a sequência <op>	echo ${x:-`pwd`}
${var:=op}	se o valor de <var> não é setado, o valor *default* assumido é a sequência <op> e <var> é setado como <op>	echo ${x:=`pwd`}
${var:?msg}	se o valor de <var> não é setado, <msg> é impressa; <msg> pode ser uma sequência vazia; neste caso a mensagem **var: parameter null or not set** é impressa se <var> for nulo	echo ${x:?variavel x vazia}
${var:+op}	se o valor de <var> é setado, executar a sequência <op>; caso contrário, não fazer nada	echo ${p:+$PATH}

Copyright © 2015 DCA/FEEC/UNICAMP

Combinação de Comandos: O *sh* provê vários operadores, tais como "||" (or), "&&" (and), "|" (pipe) e ";" , para combinar os comandos. Por exemplo, a combinação `comando_1 && comando_2` executará o comando_2 apenas se o código de saída do comando_1 for zero (sucesso). Já a forma `comando_1 || comando_2` executará o comando_2 apenas se o código de saída do comando_1 for não nulo (fracasso). O código de saída final (global) destas combinações é o código de saída do último comando executado na lista de comandos. A combinação `comando_1 | comando_2` criará uma conexão direta entre a saída de `comando_1` e a entrada de `comando_2`, fazendo com que os dados de saída do primeiro sejam utilizados como dados de entrada do segundo. Por fim, a combinação `comando_1 ; comando_2` executará os comandos em sequência, pois o shell esperará o término de um antes de começar o outro.

Vamos exemplificar mais com o comando `test` , que é um programa disponível no UNIX para verificar a validade de uma expressão. Ele retorna 0 (*sucesso*), se a expressão é verdadeira; caso contrário, ele retorna um valor diferente de 0 (*fracasso*).

Exemplo 6 A expressão

```
test -e <nome_do_arquivo> && echo arquivo <nome_do_arquivo> existe
```

é equivalente ao bloco

```
if test -e <nome_do_arquivo> then echo arquivo <nome_do_arquivo> existe
fi
```

enquanto a expressão

```
test -e <nome_do_arquivo> || echo arquivo <nome_do_arquivo> não existe
```

é equivalente ao bloco

```
if test !-e <nome_do_arquivo>           # Obs: o símbolo "!" nega a condição
       then echo arquivo <nome_do_arquivo> nao existe
fi
```

Agrupamento: Existem duas formas para agrupar um bloco composto de mais de um comando:

- por chaves, como em `{comando_1 ; comando_2;}` ou
- por parênteses, como em `(comando_1 ; comando_2)` .

No primeiro caso, os comandos são simplesmente executados no shell corrente; no segundo, um novo processo-filho *sh* é criado para executar os comandos. Após executá-los o processo-filho é terminado.

Exemplo 7 A expressão

```
$ (cd /usr; ls -l)
```

é equivalente ao seguinte bloco de comandos executados em um processo-filho iniciado pelo *sh*:

```
prompt> sh
$ cd /usr; ls -l
$ ctrl-d
```

Execução em *background*: basta colocar o símbolo & após o comando para que ele seja executado em segundo plano (background).

Expressões aritméticas : *sh* não suporta expressões aritméticas diretamente, mas pode-se usar o programa `expr` para avaliar os valores de expressões que podem ser construídas com os seguintes operadores binários: * (multiplicação), / (divisão), % (resto), + (adição), - (subtração), = (igual), \> (maior), \>= (maior ou igual), \< (menor), \<= (menor ou igual), != (diferente), \& (E lógico) e \| (OU lógico).

Exemplo 8 Os operadores e operandos devem ser separados pelo espaço em branco, como mostram os seguintes casos:

```
$ y=`expr 40 / 2`
$ x=`expr $y \* 5`
$ z=`expr \( $x + $y \) - 6`
```

Copyright © 2015 DCA/FEEC/UNICAMP

5.4. Linhas de Comentários

As linhas de comentários não são processadas pelo *sh* e são demarcadas pelos caracteres # e *linefeed*.

5.5. Tratamento de sinais de erros

O sistema UNIX pode gerar diferentes sinais durante a execução de um processo e o processo pode incluir um "tratador de sinais" (*handler*) para processá-los convenientemente.

No processo *sh* pode-se usar o comando

```
trap '<comandos>' <sinais>
```

para tratar um conjunto de sinais <sinais> captado pelo *sh*. Os comandos na lista <comandos> devem ser separados por **;** .

A seguinte tabela apresenta alguns sinais mais comuns:

Tabela 7

Sinal	Representação	Significado
SIGHUP	1	*hangup*
SIGINT	2	interrupção
SIGQUIT	3	*quit*
SIGILL	4	instrução de máquina ilegal
SIGFPE	8	erro no processamento de pontos flutuantes
SIGKILL	9	matar um processo (não pode ser capturado ou ignorado)
SIGBUS	10	erro no barramento
SIGSEGV	11	violação na segmentação (referência a end. de memória inválido)
SIGSYS	12	argumentos incorretos para a chamada ao sistema
SIGALRM	14	alarme
SIGTERM	15	sinal (em software) para terminar um processo
SIGCONT	19	continua um processo que está parado (suspenso)
SIGCHLD	20	sinal de parada enviado do processo-pai para o processo-filho

6. Dicas sobre implementação de scripts

O *Bourne shell* (*sh*) usa o arquivo *.profile* (ou *.bashrc* no caso de *bash*) existente no diretório de cada usuário (use `echo $HOME` para ver o seu) como arquivo de iniciação no processo de *login*. Se também existir o arquivo do sistema */etc/profile*, este será usado primeiro. É necessário utilizar o comando `export` para que as variáveis definidas no *login sh* sejam reconhecidas por outros *shells*.

Quando o usuário entra com um comando, o *shell* verifica se ele está definido internamente (*built-in command*). Se não estiver, o *shell* faz uma busca ao comando (arquivo executável cujo nome é o comando) em cada diretório que está definido na variável de ambiente $PATH. Sendo assim, para que o arquivo correspondente a cada *shell script* que você vai criar (utilizando um editor de texto) seja executável, adicione ao modo do arquivo recém-criado a opção 'executável pelo proprietário' através do comando mostrado no próximo exemplo.

Exemplo 9 Comando de alteração de permissões usado para permitir a execução de um script:

```
$ chmod u+x meuscript
      (chmod =  comando change mode; u = usuário; + = adicionar permissão;
           x = permissão de execução)
$ meuscript
      (agora, basta digitar meuscript para ter a execução do script)
```

Uma forma alternativa de execução é passar o seu *shell script* (arquivo pode ser não-executável) como argumento para o *shell,* que irá interpretá-lo.

```
$ sh meuscript (executa meuscript mesmo sem usar o chmod u+x antes)
```

Nesta atividade vamos criar um diretório "bin" em nosso diretório $HOME e adicionar ao $HOME/bin alguns comandos úteis. Portanto, para que estes comandos sejam reconhecíveis sob qualquer diretório, é preciso verificar se o caminho $HOME/bin faz parte da lista na variável $PATH. Caso não faça, pode-se introduzir o caminho através da atribuição mostrada a seguir.

Copyright © 2015 DCA/FEEC/UNICAMP

Tarefa 8: execute os comandos do Exemplo 10 abaixo e explique o que se pede em cada um.

Exemplo 10

```
$ echo $PATH (qual foi o resultado?)

$ PATH=$HOME/bin:$PATH (o que faz esta linha?)

$ export PATH (o que faz esta linha? Quando ela é necessária?)

$ echo $PATH (mudou algo no resultado? O que? Por que?)
```

Para verificar onde um *script* produz um erro (se aplicável) use o comando:

```
$ sh -x meuscript
```

A opção -x avisa ao *shell* que exiba os comandos que estão sendo executados, permitindo assim descobrir que comando é responsável pelo erro.

7. Outros exemplos

Alguns exemplos abaixo estão disponíveis na página web da disciplina, mas podem também ser copiados deste roteiro.

Exemplo 11

Para ler a entrada padrão no *shell script* utilize o comando `read`.

```
echo "Entre com o seu nome:"
read name
echo "Prazer em conhecê-lo(a), $name"
```

Se há mais de uma palavra na entrada, cada palavra pode ser atribuída a diferentes variáveis. Todas as palavras excedentes são atribuídas à última variável.

Exemplo 12

O comando `eval` toma o argumento na linha de comando e o executa.

```
echo "Entre com um comando:"
read comando
eval $comando
```

Exemplo 13

Os arquivos *shell scripts* podem agir como se fossem comandos padrões do UNIX, tomando argumentos a partir da linha de comando, os quais são atribuídos aos parâmetros posicionais $1 até $9. O parâmetro posicional $0 se refere ao nome do comando ou nome do arquivo executável que contém o *shell script*. O caracter especial $* referencia todos os parâmetros posicionais.

```
$ cat prog1
# Este script ecoa os primeiros 5 argumentos
# fornecidos ao script
echo Os primeiros 5 argumentos na linha
echo de comando sao: $1 $2 $3 $4 $5
$ prog1 Estou fazendo a disciplina EA872
Os primeiros 5 argumentos na linha
de comando sao: Estou fazendo a disciplina EA872
```

Exemplo 14

Para executar uma ação condicional, utilize o comando `if`.

```
$ cat prog2
if who | grep -s Maria > /dev/null
then
echo "Maria esta' logada"
else
```

Copyright © 2015 DCA/FEEC/UNICAMP

```
echo "Maria nao esta' disponivel"
fi
```

Neste exemplo, a opção `-s` faz com que o comando `grep` opere silenciosamente, sendo que qualquer mensagem de erro é direcionada para o arquivo /dev/null em lugar da saída padrão.

Exemplo 15

O comando `case` permite operar o fluxo de controle para múltiplas condições definidas a partir de uma única variável. O conteúdo da variável é comparado com padrões até que um casamento ocorra, quando os comandos associados são executados. Em seguida, o controle é passado ao primeiro comando após `esac`. Cada linha de comando deve terminar com um ponto-e-vírgula duplo. Um comando pode estar associado a mais de um padrão, desde que os padrões estejam separados por |. O caracter * pode ser utilizado para especificar um padrão *default*.

```
$ cat diario
hoje=`date +%m/%d`     # apresenta a data no formato mes/dia
case $hoje in
       03/02) echo   "aula de EA872";;
       03/09) echo   "atividades praticas de EA872";;
*)            echo   "estudar EA872";;
esac
$ date +%m/%d
03/02
$ diario
aula de EA872
```

Exemplo 16

O *script* prog2 visto antes pode ser estendido para operar múltiplos usuários, agora introduzidos como argumentos. Para tanto, utiliza-se o comando `for`.

```
$ cat prog3
for i in $*
do
if who | grep -s $i > /dev/null
then
echo "$i esta' logado(a)"
else
echo "$i nao esta' disponivel"
fi
done
```

Exemplo 17

O comando `while` executa um comando enquanto a condição for verdadeira.

```
$ cat prog4
while who | grep -s $1 >/dev/null
do
sleep 60
done
echo "$1 nao esta' mais logado(a)"
```

Dentro de loops, é possível utilizar os comandos `break` e `continue`.

```
$ cat prog5
while echo "Entre com um comando"
read response
do
case "$response" in
'done')       break;;             # nao tem mais comandos
"")           continue;;          # comando nulo
*)            eval $response;;    # executa o comando
esac
done
```

Exemplo 18

Para incluir texto em um *shell script*, é possível utilizar um tipo especial de redirecionamento.

```
$ cat prog6
```

Copyright © 2015 DCA/FEEC/UNICAMP

```
cat << EOF
No momento, este shell script esta' em fase de desenvolvimento.
Favor relatar qualquer problema ao seu autor (nome@dominio)
EOF
exec /usr/local/teste/versao_de_teste
```

8. Atividades Práticas

Estas atividades deverão ser realizadas e documentadas em seu relatório. A pontuação referente a cada unidade está indicada no início de seu enunciado. Os scripts aqui utilizados estão disponíveis na página da disciplina, mas podem também ser copiados deste roteiro.

1) (0,5) Liste quais são os shells *disponíveis em seu sistema e explique como os encontrou.*

2) (0,5) Mostre por meio de um exemplo concreto como se faz para incluir novos caminhos na variável ***PATH*** *caso o shell utilizado seja o* ***C shell (csh)****. Execute seu exemplo para confirmar (em um ambiente* ***C shell****, obviamente) e documente no relatório.*

3) (0,5) Seja um shell script denominado prog. Imediatamente após o início de sua execução através da seguinte linha de comando:

```
$ prog 1e-27 feec unicamp campinas são paulo brasil américa do sul
```

quais são os valores assumidos pelas seguintes variáveis: $0, $2, $4, $8, $$, $#, $* *e* $@ *? Explique o porquê de cada um destes valores.*

5) Documente cada linha dos scripts abaixo por meio de comentários colocados à frente de cada uma delas. Cada comentário deve ser iniciado com o sinal # e deve explicar clararmente o que está sendo feito na respectiva linha.

(a) (0,5) **menu** (veja na Tabela 3 o uso do comando de redireção <<)

```
#! /bin/sh
echo menu
stop=0
while test $stop -eq 0
do
      echo
      cat <<FIMDOMENU
      1    : imprime a data
      2,3  : imprime o diretorio corrente
      4    : fim
FIMDOMENU
echo
echo 'opcao? '
read op
echo
case $op in
       1)   date;;
       2|3) pwd;;
       4)   stop=1;;
       *) echo 'opcao invalida!';;
esac
done
```

(b) (0,5) **folheto**

```
#! /bin/sh
case $# in
     0) set `date`; m=$2; y=$6;
        case $m in
             Feb) m=Fev;;
             Apr) m=Abr;;
             May) m=Mai;;
             Aug) m=Ago;;
             Sep) m=Set;;
```

Copyright © 2015 DCA/FEEC/UNICAMP

```
             Oct) m=Out;;
             Dec) m=Dez;;
        esac;;
     1) m=$1; set `date`; y=$6;;
     *) m=$1; y=$2 ;;
esac
case $m in
     jan*|Jan*)      m=1;;
     fev*|Fev*)      m=2;;
     mar*|Mar*)      m=3;;
     abr*|Abr*)      m=4;;
     mai*|Mai*)      m=5;;
     jun*|Jun*)      m=6;;
     jul*|Jul*)      m=7;;
     ago*|Ago*)      m=8;;
     set*|Set*)      m=9;;
     out*|Out*)      m=10;;
     nov*|Nov*)      m=11;;
     dez*|Dez*)      m=12;;
     [1-9]|10|11|12) ;;
     *)              y=$m; m="";;
esac
/usr/bin/cal $m $y
```

(c) (0,5) **path**

```
#! /bin/sh

for DIRPATH in `echo $PATH | sed 's/:/ /g'`
                                  # Consulte o manual do sed!
do
  if [ ! -d $DIRPATH ]
       then
       if [ -f $DIRPATH ]
           then
           echo "$DIRPATH nao e diretorio, e um arquivo"
           else
           echo "$DIRPATH nao existe"
       fi
  fi
done
```

(d) (1,0) **classifica**

```
#! /bin/sh
case $# in
 0|1|[3-9]) echo 'Uso: classifica arquivo1 arquivo2' 1>&2; exit 2 ;;
esac
total=0; perdida=0;
while read novalinha
do
     total=`expr $total + 1`
     case "$novalinha" in
          *[A-Za-z]*)     echo "$novalinha" >> $1 ;;
          *[0-9]*)        echo "$novalinha" >> $2 ;;
          '<>')           break;;
          *)              perdida=`expr $perdida + 1`;;
     esac
done
echo "`expr $total - 1` linha(s) lida(s), $perdida linha(s) nao aproveitada(s)"
```

6) (1,0) **traps** *Explique o funcionamento do script* **traps***. Para entender o funcionamento deste script, execute o mesmo em* background *(usando o operador &), liste o diretório para encontrar um arquivo criado pelo script, o qual informa seu* PID*, execute um* kill *conforme o especificado abaixo e explique o que acontece.*

```
$ traps &

$ ls
```

Copyright © 2015 DCA/FEEC/UNICAMP

```
$ kill <PID>

$ ls
```

Repita o procedimento com kill –2 <PID> *e* kill –15 <PID> *e explique o que ocorreu.*

(conteúdo do script traps)

```
#! /bin/sh

ARQUIVO=arq.$$
touch $ARQUIVO

trap "echo 'Algum processo enviou um TERM' 1>&2; rm -f $ARQUIVO; exit;" 15
trap "echo 'Algum processo enviou um INT' 1>&2; rm -f $ARQUIVO; exit;" 2

while true
  do
  # Espera 5 segundos
  sleep 5
done
```

7) (1,0) **subspar** *Consultando (i) a tabela 6, (ii) a seção sobre combinação de comandos (logo abaixo da tabela 6) e (iii) o manual para o comando test, explique as saídas produzidas pelo programa* ***subspar*** *quando as seguintes linhas de comando são executadas.*

```
$ subspar

$ subspar sp rj mg es df pr mt ms
```

(conteúdo do arquivo subspar)

```
#! /bin/sh

test -n "$1" && param1=$1
test -n "$2" && param2=$2
test -n "$3" && param3=$3
test -n "$4" && param4=$4

echo "1º resultado do teste:${param1-rs} com param1 = $param1"
echo "2º resultado do teste:${param2=pa} com param2 = $param2"
echo "3º resultado do teste:${param3+to} com param3 = $param3
echo "4º resultado do teste:${param4?Quarto parâmetro não iniciado}
com param4 =  $param4"
```

8) (1,0) Explique como funciona o script abaixo, mostrando qual é sua utilidade prática e detalhando cada uma das operações.

```
#! /bin/sh
test -d $HOME/lixo || mkdir $HOME/lixo
test 0 -eq "$#" && exit 1;
case $1 in
        -l) ls $HOME/lixo;;
        -r) case $# in
                  1) aux=$PWD; cd $HOME/lixo; rm -rf *; cd $aux;;
                  *) echo pro_lixo: Uso incorreto;;
            esac;;
        *) for i in $*
           do
                  if test -f $i
                       then  mv $i $HOME/lixo
                       else echo pro_lixo: Arquivo $i nao encontrado.
                  fi
           done;;
esac
```

9) (1,0) Explique como funciona o script abaixo, mostrando qual é sua utilidade prática e detalhando cada um dos comandos.

```
#! /bin/sh

if [ $# -eq 0 ]
```

Copyright © 2015 DCA/FEEC/UNICAMP

```
    then
    set $PWD
fi

for ARG in $*
  do
  case $ARG in
      --prof=*)
          PROFUNDIDADE=`echo $ARG | cut -f 2 -d '='`
          ;;
      *)
          if [ -d $ARG ]
              then
              CONT=${PROFUNDIDADE=0}
              while [ $CONT -gt 0 ]
                do
                echo -n "  "
                CONT=`expr $CONT - 1`
              done
              echo "+$ARG"
              cd $ARG
              for NAME in *
                do
                tree --prof=`expr $PROFUNDIDADE + 1` $NAME
              done
          else
              if [ -f $ARG ]
                  then
                  CONT=${PROFUNDIDADE=0}
                  while [ $CONT -gt 0 ]
                    do
                    echo -n "  "
                    CONT=`expr $CONT - 1`
                  done
                  echo "-$ARG"
              fi
          fi
          ;;
  esac
done
```

10)(2,0) Implemente em Bourne Shell ou Bash um script que executa um comando passado como argumento. Este script deve suportar as seguintes opções: - -repeticoes=N*, que indica que o comando passado como argumento deve ser executado* N *vezes, e* - -atraso=M*, que indica que um atraso de* M *segundos deve ser efetuado antes da primeira execução e entre as demais execuções. Seu script deve tentar casar as opções acima no início dos argumentos. Assim que algo não for casado como as duas opções acima, deve-se interpretar todo o restante dos argumentos como o comando a ser executado. Sugestão: utilize case e shift. Caso alguma das duas opções não seja passada, estabelecer os valores default 1 para repetições e 0 para atraso usando recursos similares aos usados no script* ***subspar*** *mais acima.*

Atividade extra opcional: *ganhará um ponto extra quem fizer e documentar a atividade opcional abaixo.*

11) (1,0) Implemente em Bourne Shell ou Bash um script recursivo (o script chama a si mesmo) que deve ler um valor do terminal e calcular o fatorial deste valor. O script deve indicar um erro se o valor lido for negativo. Explique o funcionamento de seu script e dê alguns exemplos de execução que explorem todas as possibilidades, incluindo situações de erro de entrada e de cálculo acima das possibilidades do computador. O script deve usar recursos da linguagem de shell e não recorrer a comandos ou programas em outras linguagens.

Copyright © 2015 DCA/FEEC/UNICAMP