

VII - 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

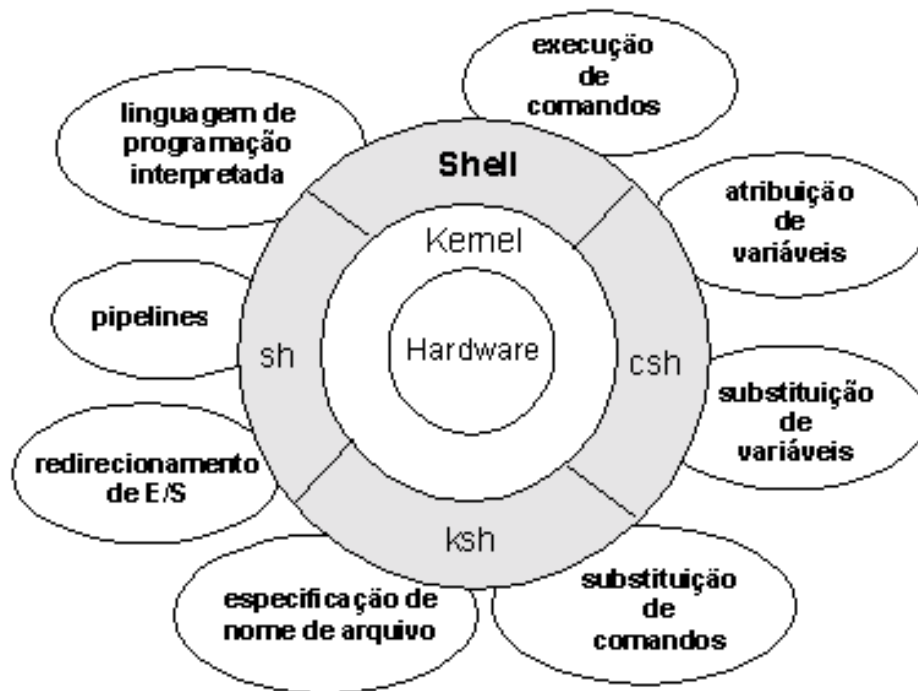


Figura VII.1 – Shell

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 <i>sh</i> . |
| • Pdksh | Public domain Korn Shell - versão de domínio público do <i>ksh</i> . |
| • Zsh | Z Shell - compatível com o <i>sh</i> . |
| • Tcsh | versão padronizada do <i>csh</i> . |

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.

Tabela VII.1 - Comparação do csh, ksh e sh

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

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 <i>default</i> 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 - Substituição de 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. Este 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 frequentemente 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.

2.4 - Transferência de Variáveis Locais para o Ambiente

Para tornar uma variável disponível para os processos-filhos, ela deve existir no ambiente. No Bourne Shell, para o usuário criar uma variável de ambiente, ele deve criar uma variável como local e exportá-la com o comando `export`. A criação de uma variável local é realizada utilizando um comando de atribuição "=", conforme visto anteriormente.

O comando `export name` transfere as variáveis específicas da área local para a área de ambiente. Sem argumento, o comando `export` exibe as variáveis que foram exportadas para este shell.

```
$ variável = valor          (cria uma variável de Shell)
$ export variável           (cria uma variável de ambiente)
```

Para remover o mecanismo é semelhante:

```
$ unset variável
$ export variável           (remove a variável de ambiente)
```

O comando `env` exibe todas as variáveis e seus valores que estão armazenados no ambiente. Quando um processo-filho é iniciado, ele recebe uma cópia das variáveis de ambiente de seu pai, por isso, se um processo-filho modifica o valor de uma variável de ambiente, o valor modificado será propagado para qualquer um dos processos-filho, mas o valor do pai permanece inalterado. Isto significa dizer que um filho não pode alterar o ambiente de seu pai.

3 - HISTÓRICO (History)

Algumas atividades de um usuário em uma sessão fazem com que comandos sejam repetidos muitas vezes. O C shell permite que o usuário mantenha um buffer de histórico de comandos, capaz de manter seus comandos anteriores. Para usar o mecanismo de histórico do C shell de forma automática (toda a vez que abrir a sessão), você deve colocar as 3 linhas seguintes no arquivo `.cshrc` localizado no seu diretório HOME:

```
set history=20
set savehist=20
set prompt="[!]%"
```

Estas declarações irão:

- Permitir que seu buffer de histórico de comandos armazene os 20 comandos anteriores.
- Salvar os últimos 20 eventos de seu buffer do histórico de comandos, quando você terminar a sessão e restaurá-los a próxima vez que se logar.
- Definir o prompt do seu C shell, para mostrar o número de cada evento (comando) que você digitar no terminal

Para referenciar comandos armazenados na lista, pode-se usar alguns parâmetros:

!!	executa o último comando;
!n	executa o n-ésimo comando;
!string	executa o comando mais recente que comece com o "string";
! -n	executa o n-ésimo comando a contar do último.

VIII - PROCESSOS

1 - DEFINIÇÃO

Um processo é um programa de execução independente que tem seu próprio conjunto de recursos. Para um mesmo programa (entidade estática), podem existir vários processos (entidades dinâmicas).

O UNIX, como sistema operacional multitarefa, permite a existência de vários processos ao mesmo tempo. Em máquinas monoprocessadas, o kernel do UNIX se encarrega de escalonar os recursos de execução do único processador, para os vários processos do sistema. Já em máquinas multiprocessadas, pode-se ter processos executando em paralelo, e não concorrentemente como nas máquinas monoprocessadas.

Quando for digitado algum comando no ambiente do UNIX, este comando é executado como um processo subordinado (chamado processo filho) do processo corrente (chamado processo pai). Todos os processos criados pelo usuário são filhos do processo de login.

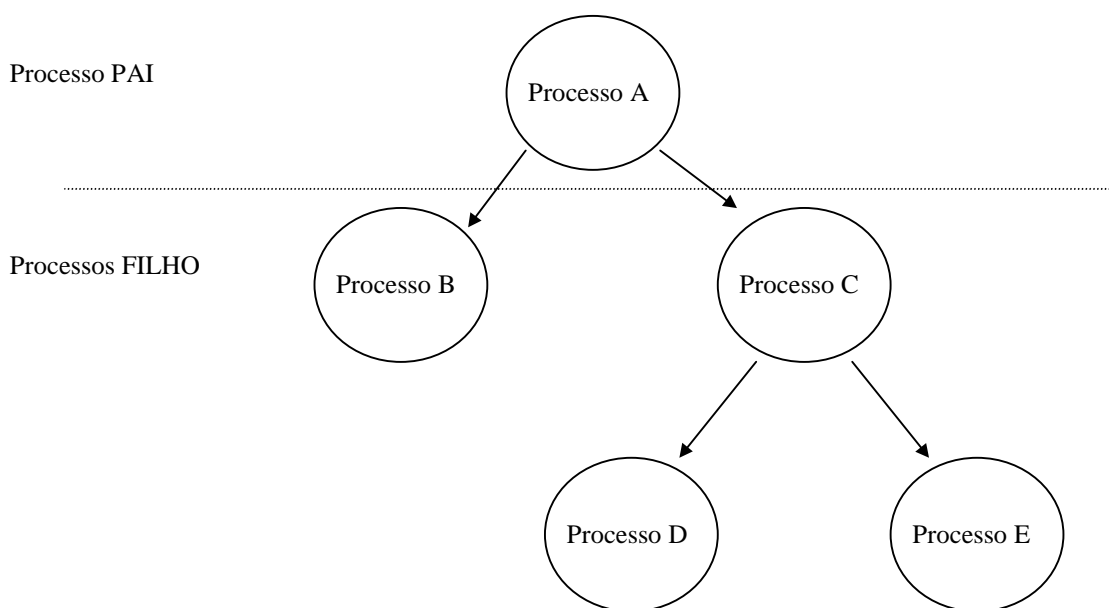


Figura VIII.1 - Processo e Subprocessos

O sistema operacional tem muitos recursos que devem ser gerenciados, incluindo recursos de hardware e software. Um dos recursos de software que deve ser gerenciado é um programa em execução. As características do processo são:

- É um programa que pode ser executado concorrentemente.
- Pode ser criado e destruído
- Possui recursos alocados para ele
- Possui um ambiente associado a ele que: é herdado do processo pai, consiste de todas as informações relativas ao processo e pode ser alterado através de comandos no ambiente de shell
- Pode criar outros processos
- Pode se comunicar com outros processos

No UNIX cada processo possui o seu próprio ambiente onde existem todas as informações relativas ao processo e que afetam a sua execução:

- Dados
- Arquivos abertos
- Diretório corrente
- User ID
- Process ID
- Parente Process ID
- Conjunto de variáveis

2 - CRIANDO UM PROCESSO

Quando o sistema operacional é inicializado, o processo `init` é inicializado. Este processo é responsável pelo processo de login que aguarda pela entrada de comandos através dos terminais dos usuários. O processo `init` pertence ao superusuário e é controlado pelo console. O console é o terminal para onde o kernel escreve as mensagens de erro de sistema.

No momento que o usuário se conecta ao sistema, o processo `init` inicia um processo shell de usuário com um ambiente padrão. A partir deste ponto o usuário cria outros processos executando comandos, rodando programas ou shell scripts.

Supondo que ao inicializar a sessão no UNIX o seguinte ambiente foi criado:

```
LINUX
© Copyrights by
Login: grupo5
grupo5's Password: *****

$

User ID = grupo5
Diretório=/home/grupo5
Programa=ksh
ProcessID=9054
Parent=init
ParentPID=1
Arquivos abertos=/dev/tty5 (stdin stdout stderr)
```

Após inicializada a sessão o usuário executa alguns comandos no prompt do UNIX.

```
$ pwd
/home/grupo5

$ cat arquivo.txt
Exibe o conteúdo de arquivo
```

Durante a execução dos comandos o ambiente dos subprocessos foi alterado para:

```
User ID = grupo5
Diretório=/home/grupo5
Programa=cat
ProcessID=10202
Parent=ksh
ParentPID=9054
Arquivos abertos=/dev/tty5 (stdin stdout stderr)
/home/grupo5/arquivo.txt (aberto pelo comando cat)
```

3 - EXECUTANDO PROCESSOS EM BACKGROUND E FOREGROUND

Quando você executa um comando shell ou um script, por default é executado em foreground (primeiro plano). Quando termina a execução do comando só então é possível executar outro pelo prompt do UNIX.. Isto ocorre porque quando um comando está sendo executado em foreground o shell não pode aceitar outra entrada até que a execução seja concluída.

Para permitir mais do que um comando sendo executado ao mesmo tempo, os comandos precisam ser executados em background que não possui controle direto da entrada e saída de dados do terminal. Este método é aconselhável quando o comando a ser executado consome tempo de CPU e não requer entrada de dados interativa. Como exemplo citamos programas de classificação de dados, compilação, cálculos matemáticos complexos.

Para que um programa seja executado em background acrescente o caracter `&` após o comando no prompt do UNIX:

```
$ ls -la &
```

Quando um processo é colocado em background, o shell informa o número do job e a identificação do processo (process-ID se a opção `monitor` estiver ligada: `set -o monitor`). O número do job identifica o número do pedido do job na sua sessão do terminal, e process-ID identifica o número que o sistema UNIX associa a cada processo executado. A opção `monitor` também envia uma mensagem quando é exibida quando o processo em background é concluído.

Outra característica é que os comandos executados em background não podem ser interrompidos com a tecla Ctrl-c que suspende o job corrente em foreground, e um prompt do Shell é fornecido ao usuário para a execução de novas tarefas.

Ao pressionar ^Z, um sinal de suspensão (SIGTSTP) é enviado para o processo ou grupo de processos que estão executando, fazendo com que estes parem sua execução e fiquem suspensos. Esse sinal pode ser tratado pelo programa, de modo que ele não seja aceito, não suspendendo os processos disparados por aquele programa.

O login Shell é o líder da sessão, e todos os processos criados na sessão são “filhos” deles. Quando processos em background de uma sessão tem seu processo “pai” finalizado, eles passam automaticamente a serem filhos diretos do processo init e continuam sua execução até o seu término. O processo init é criado no momento da inicialização do sistema (boot) com PID igual a 1, e somente termina quando o sistema é desligado (shutdown).

IX - REDIRECIONAMENTO DE E/S, FILTROS E PIPELINES

1 - DEFINIÇÃO

Um comando do UNIX é um programa (conjunto de instruções) que geralmente lê dados de um ou vários arquivos de entrada, efetua algum tipo de processamento sobre eles (computação) e finalmente produz os resultados em arquivo(s) de saída. Na arquitetura do UNIX, note-se, dispositivos como terminal, discos e impressoras também são tratados como arquivos.

Ao criar um processo, o UNIX predefine e automaticamente lhe associa três arquivos padrões:

- stdin (0) - entrada padrão;
- stdout (1) - saída padrão;
- stderr (2) - saída padrão para erros.

A entrada padrão (stdin) é um arquivo de onde os programas obtêm dados de entrada para o seu processamento. O sistema associa a esta entrada padrão o teclado do terminal, caso o usuário não especifique uma alternativa. Essa redefinição é um recurso oferecido pelo Shell conhecido como redirecionamento.

A saída padrão (stdout) é o arquivo para onde usualmente são enviados os resultados do programa. O monitor do terminal é o arquivo de saída padrão associado ao processo pelo sistema. Assim como a entrada padrão, a saída padrão pode ser redirecionada para outros arquivos (impressora, disco, etc).

Além da saída e entrada padrão, o UNIX reserva uma conexão de I/O para mensagens de erros do programa. Se um usuário redireciona a saída padrão de um programa, mensagens de erros emitidas por este programa serão ainda enviadas para o terminal, pois a saída padrão de erros também é o terminal. O Shell permite redirecionar a saída padrão de erros para outros arquivos no sistema.

Como exemplo, pode-se utilizar o comando cal.

```
$ cal 10 1999
```

Nessa execução do comando cal, os dados de entrada para seu processamento são os parâmetros 10 e 1999. Nesse caso, o dispositivo de entrada foi o teclado. A saída do comando será realizada na tela do terminal do usuário, por default o dispositivo de saída padrão. O usuário poderia redirecionar essa saída para um arquivo ou algum outro dispositivo de saída, como uma impressora.

Ou seja, se um comando espera uma entrada e nenhum arquivo de entrada é especificado, então esse comando vai buscar seus dados de uma entrada padrão, que é o terminal do usuário. Se um comando vai jogar a saída de seu processamento para um arquivo, e nenhum arquivo de saída foi especificado, o comando utilizará a saída padrão que também é o terminal do usuário.

Quase todos os comandos do UNIX seguem essa filosofia de trabalho. Alguns comandos foram previamente implementados com a especificação de qual arquivo seria utilizado para saída e/ou entrada. Por exemplo, o comando ls sempre utiliza como entrada padrão um diretório. Já o comando lpr, utiliza como saída padrão a impressora.

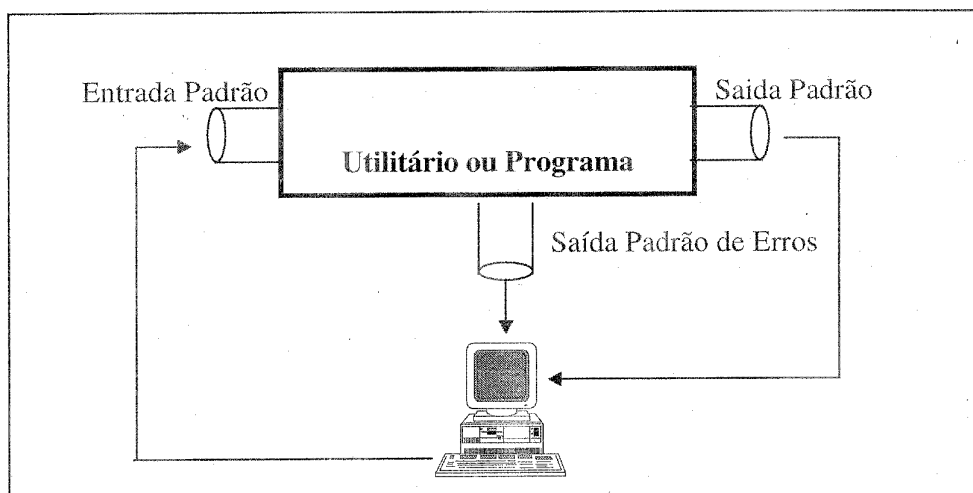


Figura IX.1 - Entrada e Saídas Padrões

2 - COMANDOS DE REDIRECIONAMENTO DE E/S

Redirecionamento é uma facilidade que o UNIX oferece, permitindo que o usuário altere as saídas e a entrada padrão do programa, especificando novos arquivos.

2.1 - REDIRECIONANDO A SAÍDA PADRÃO (> e >>)

No exemplo do comando `cal`, a saída padrão do comando foi o terminal. O usuário pode definir que a saída do programa (um determinado calendário) seja realizada para um arquivo em disco, para posterior utilização. Esse redirecionamento é feito utilizando-se o símbolo “>” (maior que) e o nome do arquivo que armazenará a saída do programa.

```
$ cal 10 1999 > calendario99.dat
```

Nesse exemplo, a saída do comando `cal` será redirecionada para o arquivo `calendario99.dat`. O arquivo será criado, caso não exista, ou sobrescrito se já existir um arquivo com mesmo nome no diretório.

Uma outra forma de redirecionar a saída padrão é feita utilizando o símbolo “>>”. Nesse caso, a saída é colocada no final (append) de um arquivo existente, não destruindo os dados do arquivo. Caso o arquivo não exista, ele será criado para armazenar a saída do programa.

```
$ cal 11 1999 >> calendario99.dat
```

2.2 - REDIRECIONANDO A ENTRADA PADRÃO (<)

O redirecionamento de entrada pode ser feito em todos os comandos que lêem seus dados de uma entrada padrão. O redirecionamento de entrada é realizado utilizando o símbolo “<” (menor que), indicando ao Shell para trocar a entrada padrão (terminal) para um novo arquivo especificado.

Como exemplo, pode-se ter o comando `write` com a entrada padrão redirecionada para outro arquivo. No caso, a entrada estará contida no arquivo em disco, chamado `calendario99.dat`.

```
$ write antonio < calendario99.dat
```

Desse modo, o conteúdo do arquivo `calendario99.dat` será transmitido para a tela do terminal do usuário `antonio`.

2.3 - REDIRECIONANDO A ENTRADA E A SAÍDA SIMULTANEAMENTE

O UNIX permite a utilização em conjunto dos símbolos > / >> e <, oferecendo ao usuário a capacidade de redirecionar a entrada e a saída padrão do programa de uma única vez.

```
$ who > users.dat
```

```
$ cat users.dat
```

maria	ttyp1	Jun	10	13:03	1999
ana	ttyp3	Jun	10	13:13	1999
antonio	ttyp5	Jun	10	14:07	1999
marco	ttyp6	Jun	10	14:23	1999

```
$ sort < users.dat
```

ana	ttyp3	Jun	10	13:13	1999
antonio	ttyp5	Jun	10	14:07	1999
maria	ttyp1	Jun	10	13:03	1999
marco	ttyp6	Jun	10	14:23	1999

```
$ sort < users.dat > users2.dat
```

```
$ cat users2.dat
```

ana	ttyp3	Jun	10	13:13	1999
antonio	ttyp5	Jun	10	14:07	1999
maria	ttyp1	Jun	10	13:03	1999
marco	ttyp6	Jun	10	14:23	1999

```
$ sort -r +1 < users.dat >> users2.dat
$ cat users2
ana          tty3          Jun    10    13:13    1999
antonio      tty5          Jun    10    14:07    1999
maria        tty1          Jun    10    13:03    1999
marco        tty6          Jun    10    14:23    1999
ana          tty3          Jun    10    13:13    1999
antonio      tty5          Jun    10    14:07    1999
maria        tty1          Jun    10    13:03    1999
marco        tty6          Jun    10    14:23    1999
```

2.4 - REDIRECIONANDO A SAÍDA DE ERROS

A saída padrão de erros é muito utilizada para a depuração de programas e emissão de relatórios de erros aos usuários. O sistema operacional pode utilizar a saída padrão de erros, associada ao processo, para relatar a seu usuário algum erro que ocorreu na execução do processo. A saída de erros é redirecionada utilizando a notação: `>&`

```
$ ls arquivo_nao_existente > saida_erro.dat
ls: arquivo_nao_existente: No such file or directory
$ cat saida_erro.dat
$ ls arquivo_nao_existente >& saida_erro.dat
$ cat saida_erro.dat
ls: arquivo_nao_existente: No such file or directory
```

Da mesma forma que a saída padrão, a saída padrão de erros pode ser redirecionada para o final (append) de um arquivo existente, não destruindo os dados do arquivo. Caso o arquivo não exista, ele será criado para armazenar a saída do programa.

```
$ ls arquivo_nao_existente >& saida_erro.dat
$ cat saida_erro.dat
ls: arquivo_nao_existente: No such file or directory
$ ls arquivo_nao_existente2 >>& saida_erro.dat
$ cat saida_erro.dat
ls: arquivo_nao_existente: No such file or directory
ls: arquivo_nao_existente2: No such file or directory
```

3 - FILTROS

Filtro é um programa que lê de sua entrada padrão, executa um processamento e escreve para sua saída padrão. Filtros são normalmente conectados linearmente em Shell pipelines.

São exemplos de filtro: `wc`; `sort`; `grep`

Programas que não são filtros: `who`; `ls`; `cd`; `mv`; `pwd`; `whoami`; `clear`

4 - PIPELINES

Pipes são uma velha forma de comunicação entre processos no UNIX, e são fornecidos por todos os sistemas UNIX. Um pipe (duto) é uma conexão que conduz a saída padrão de um programa para a entrada padrão de outro programa. Esse conceito facilita aos programas UNIX trabalharem juntos, sem a necessidade de arquivos intermediários.

4.1 - PIPELINE (|)

O símbolo para a criação de pipelines é a barra vertical (|). Quando em uma linha de comando, dois programas são separados por uma barra vertical (|), a saída do primeiro é transferida para a entrada do segundo.

```
$ who | wc -l
6
```

A resposta da linha de comando acima é o número de usuário conectados no sistema naquele instante.

Os pipes são half-duplex, isto é, os dados fluem somente em uma direção. Eles somente podem ser usados entre processos que possuam o mesmo ancestral comum (processo pai). No caso anterior, o processo “pai” comum aos dois processos era o Shell do usuário.

4.2 - CAPTURANDO A SAÍDA INTERMEDIÁRIA DE UM *PIPELINE*

A possibilidade de se ter pipeline traz a necessidade de se capturar sua saída intermediária. Para isso, tem-se o comando `tee`.

```
$ tee nome_arquivo
```

Esse comando copia a saída de um dos programas pertencente ao pipeline, o qual está imediatamente anterior ao comando `tee`, para o arquivo determinado.

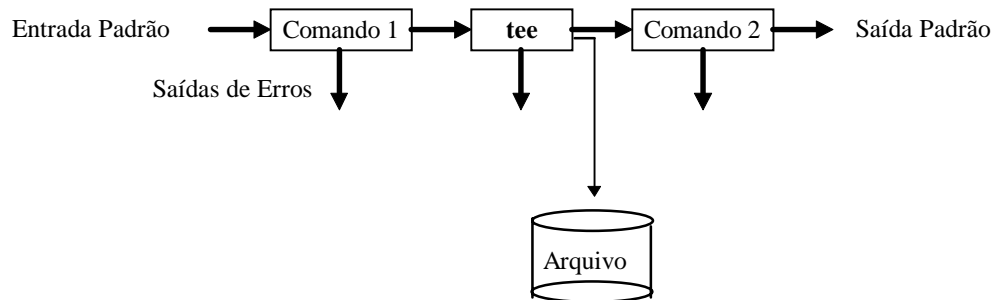


Figura IX.2 - Comando `tee` e *Pipelines*

```
$ who | tee usuarios | sort
```