Fundamentos de Utilização

Valter Alves
Paulo Almeida

Sistemas Operativos

Engenharia de Sistemas e Informática

Escola Superior de Tecnologia de Viseu

Instituto Politécnico de Viseu

2003/2004, v1.0

Índice

Prefácio

O presente documento visa apoiar a leccionação de uma das componentes do programa de estudos da disciplina de Sistemas Operativos do curso de Engenharia de Sistemas e Informática, da Escola Superior de Tecnologia de Viseu.
Este *não* é o texto principal da disciplina. A matéria que aborda apenas se refere a um tópico de carácter muito introdutório. Mesmo nesse sentido, deve ser entendido como um guia para a leccionação e/ou como uma ferramenta de apoio ao acompanhamento das respectivas aulas; não como um compêndio de referência.

Convenções adoptadas no texto

Significado das etiquetas usadas

[EX:] Exemplo.
[EXC:] Exercício a realizar no computador.
[TA:] Tópico avançado (pelo menos, considerando o momento em que é referido). Questão de algum pormenor.
[***] Assunto que posteriormente se refere com maior detalhe (neste texto ou em outro momento da disciplina).

Outras convenções

- Texto envolto em parêntesis quebrados ("<" e ">") alude à inserção de um valor, caracter ou comando (conforme o contexto), em sua substituição.
- As combinações de teclas de controlo seguem a notação "`Ctrl+...`". Não é usada a notação "`^...`".
- O termo "caracter", embora não fazendo parte do léxico português, é usado em substituição de "carácter" dada ser essa a adopção típica na linguagem do domínio.
- A expressão "por omissão" é usada para exprimir o conceito associado ao termo anglo-saxónico "*default*".

Unix/Linux — Fundamentos de utilização

Este documento contém indicações muito sumárias sobre alguns aspectos relevantes do sistema operativo, à luz dos propósitos da sua utilização na disciplina.
Aos alunos é remetido o esforço de exploração e aprendizagem que um conhecimento mais aprofundado deste sistema exige.
Embora as indicações contidas neste texto se refiram, na sua generalidade, ao sistema operativo Unix, é importante considerar que a plataforma utilizada como base de teste e estudo foi o Red Hat Linux (RHL) 9.0. Assim, é possível que determinados aspectos aqui referidos representem alguma especificidade desse sistema, não verificável nas distribuições de Unix nem, eventualmente, em outras versões ou distribuições de Linux.
A ordem pela qual os vários temas se encontram posicionados neste texto está relacionada com a sequência em que são abordados nas aulas da disciplina.

1. Interface

O Linux disponibiliza uma interface de texto tipo terminal (característica do Unix) e uma interface gráfica (com um princípio de utilização similar às de outros sistemas, como o MacOS e o Windows).
A interface relevante para o estudo que se segue, salvo menção em contrário, é a interface de texto.

A utilização do sistema é feita por introdução de comandos numa linha de comando disponibilizada por uma *shell* (um interpretador de comandos) a correr num terminal. O terminal pode ser local ou pode ser mantido através de ligação *telnet* a um servidor remoto.

As instalações de Linux dão, localmente, acesso a vários terminais. Os terminais são designados por *ttyk* em que *k* é o seu índice (*tty1* até *tty6*). Existe ainda a interface gráfica (quando instalada) na qual podem ser abertas janelas de terminal (designadas de *pts/k* (*k* é o índice)).
Para permutar entre terminais usar `Ctrl+Alt+F1` até `Crtl+Alt+F6`. A interface gráfica acede-se com `Ctrl+Alt+F7`.
[TA:] Para permutar de um terminal (texto) para outro terminal ou para a interface gráfica, pode utilizar-se, alternativamente, `Alt+F1` até `Alt+F7`. No entanto, para permutar da interface gráfica para um terminal de texto é mesmo necessário recorrer a `Ctrl+Alt+F1` até `Crtl+Alt+F6`.

Existem vários tipos de *shell* que diferem em alguns aspectos da sua utilização, mas que são similares para a maior parte dos efeitos.

A introdução de um comando faz-se no espaço que segue o *prompt* apresentado pela *shell* e toma, tipicamente, a forma

```
nomedocomando –switches parâmentro1 parâmentrok
```

Os *switches* (inscritos após '`-`', por convenção) permitem alterar o comportamento do comando em causa, de acordo com a respectiva programação.
Todos os aspectos relacionados com a utilização do comando, dos seus *switches* e dos parâmetros esperados, estão invariavelmente bem documentados.
Ao conjunto dos *switches* e dos parâmetros inscritos em frente a um comando costuma chamar-se de argumentos.

É importante ter em atenção que, à semelhança da linguagem C (que esteve sempre na base do desenvolvimento do Unix), o Unix é *case-sensitive*, ou seja é relevante se cada letra é maiúscula ou minúscula.

Todos os comandos acabam por interagir de forma directa ou indirecta com ficheiros. Os ficheiros são as unidades básicas de gestão e manutenção dos dados em Unix.

2. Sistema de ficheiros

Um dos aspectos mais característicos do Unix é o seu sistema de ficheiros.

O sistema de ficheiros do Unix tem um formato diferente dos sistemas de ficheiros usados noutros sistemas operativos (e incompatível, para a maior parte dos efeitos). O seu nome é simplesmente *Unix File System* (*UFS*). O Linux utiliza as extensões *ext2* e *ext3*.
Por exemplo, o DOS/Windows usa *FAT* (e derivados) e *NTFS*.

O sistema de ficheiros baseia-se numa estrutura hierárquica (árvore) de directorias que contém os ficheiros, com uma única raiz para todo o sistema, mesmo na presença de diversos *drives*. A estrutura de ficheiros de cada um dos *drives* (CD-ROM, disquete, outros discos rígidos ou partições, etc.) é "pendurada" (montada) algures na árvore principal, passando a ficar perfeitamente integrada no conjunto de directorias.
Em termos de interface com o utilizador/programador, qualquer objecto é um ficheiro. As próprias directorias são ficheiros. Os dispositivos (teclado, rato, monitor, leitor de CD-ROM, etc.) são também tratados como ficheiros.

Existe um mecanismo de definição de permissões que possibilita indicar, para cada ficheiro, as acções que podem ser levadas a cabo pelos utilizadores.
Normalmente, cada utilizador tem uma directoria própria, a que é comum chamar de directoria pessoal (ou *homedir*, ou simplesmente *home*), na qual pode livremente criar e remover ficheiros e definir as respectivas permissões para si e para os restantes utilizadores.

2.1. Caminhos

Os objectos (ficheiros) são referidos por um caminho (path) que indica o percurso desde a directoria raiz até à sua localização. As directorias mencionadas no caminho são separadas pelo caracter '`/`' (ao contrário de '`\`' que é o usado em DOS e Windows).
[EX:]

```
/usr/bin/cc
```

(Admitir, neste e nos exercícios imediatos, que os nomes indicados no caminho correspondem a directorias e ficheiros que existem).

Em cada instante está definida uma directoria corrente (não confundir com a directoria pessoal do utilizador).
[TA:] A directoria corrente é muitas vezes chamada de "working directory", o que pode gerar alguma confusão já que, por vezes, a directoria pessoal do utilizador é designada de "directoria de trabalho do utilizador". "Working directory" deve ser entendido como "a directoria sobre a qual se está a trabalhar, no momento".

Existem duas formas de referir o caminho para um determinado ficheiro:

- Caminho absoluto: refere explicitamente o caminho desde a raiz. Começa necessariamente por '`/`' (que representa precisamente a raiz do sistema de ficheiros).
  [EX:]
  `/usr/bin/cc`

- Caminho relativo: indica o caminho desde a directoria corrente. Fica subentendido o caminho desde a raiz até à directoria corrente.
  [EX:]
  `exercicios/hello`
  Se a directoria corrente for
  `/home/aluno`
  então a localização exacta (absoluta) do ficheiro seria
  `/home/aluno/exercicios/hello`

Na especificação dos caminhos, o nome '`..`' refere-se à directoria acima e '`.`' refere-se à directoria em causa.
[EX:]

```
../xpto
```

refere-se ao ficheiro (ou directoria) `xpto` que se encontra na directoria mãe (a directoria acima da directoria corrente).
[EX:]

```
./xpto
```

ou simplesmente

```
xpto
```

refere-se a `xpto` na directoria corrente.

[EX:]

```
/dev/../usr/bin/./.././.././etc/./profile.d/../profile.d/./.
```

é equivalente a

```
/etc/profile.d
```

(isto não é interessante, mas é correcto).

Um aspecto típico e interessante dos sistemas de ficheiros do Unix é que, normalmente, é seguida uma estrutura comum para nomeação e localização das respectivas directorias (e até de ficheiros), o que facilita a manutenção/aprendizagem/desenvolvimento.
O RHL segue o standard FSH (File System Hierarchy Standard).

2.2. Estrutura do sistema de ficheiros

Algumas das directorias mais relevantes e respectivo conteúdo típico, de acordo com o standard FSH, são:

`/bin` : comandos essenciais (muitas vezes designados de “binários”) usáveis por todos os utilizadores.

`/sbin` : comandos destinados ao super-utilizador e comandos de sistema essenciais (por exemplo, para o arranque do sistema).

`/dev` : *device files*. Ficheiros (especiais) que representam os dispositivos que estão ligados ao sistema.
Este ficheiros não são os *device drivers*. São ficheiros que permitem interagir com eles. Deste modo, implementam uma interface que simplifica a interacção com os dispositivos.

`/etc` : ficheiros com as configurações locais.
[TA:] Esta directoria não contém ficheiros binários (em sistemas mais antigos isso acontecia; agora eles encontram-se noutras directorias (nomeadamente `/bin` e `/sbin`)).

`/lib` : bibliotecas partilhadas (*shared libraries*) essenciais (usadas pelos comandos contidos em `/bin` e `/sbin`) e módulos do *kernel*.

`/mnt` : ponto de montagem para sistemas de ficheiros temporariamente montados (como CD-ROM e disquete).

`/tmp` : ficheiros temporários.
[TA:] Útil durante o desenvolvimento de aplicações. As suas permissões normalmente dão acesso a todos os utilizadores. O seu conteúdo é tipicamente removido de forma automática.

`/home` : directoria que contém as directorias pessoais dos utilizadores (normais).
Normalmente cada directoria pessoal tem o nome do respectivo utilizador.

`/root` : directoria pessoal do super-utilizador.

`/usr` : ficheiros que possam ser partilhados por todo um site.
Algumas das subdirectorias de `/usr` são homónimas de outras contidas na raiz, mas os seus conteúdos enquadram-se no critério acima mencionado.
[TA:] Normalmente, fica numa partição separada que é montada em modo *read-only* (donde, até pode estar em CD-ROM).
[TA:] Não são contidos nesta directoria ficheiros que sejam essenciais para o arranque e funcionamento básico da máquina, para prever a situação em que a (eventual) montagem não seja bem sucedida.

`/usr/bin` : a maior parte dos comandos destinado aos utilizadores.

`/usr/sbin` : comandos de sistema não essenciais.

`/usr/etc` : ficheiros de configuração (não específicos da máquina).

`/usr/lib` : bibliotecas (*libraries*) relacionadas com a programação e os *packages*.

`/usr/include` : ficheiros de *include* (explicitamente referidos na programação em C).

`/usr/include/sys`: ficheiros de *include* (explicitamente referidos na programação recorrendo a chamadas ao sistema, em C).

`/usr/man` : documentação do sistema (nomeadamente *manpages*).

[TA:] `/usr/local`: no RHL a directoria `/usr/local` é assumidamente usada para um efeito algo diferente do indicado no standard FHS. O FHS especifica que é onde deve ser instalado o software que se pretende que seja salvaguardado de eventuais *upgrades* do sistema. O RHL utiliza esta directoria para instalação de software que seja local à máquina.

2.3. Nomes dos ficheiros

Em Unix os ficheiros podem ter nomes longos (eventualmente com espaços).
O conceito de extensão (típico do DOS/Windows) não faz sentido em Unix. O caracter '`.`' (que em DOS/Windows funciona como separador entre o nome e a extensão) pode ser usado livremente.
[EX:]

```
xpto
..x.p...t.o..
....
```

são nomes de ficheiro válidos.

Existe contudo a convenção (apenas isso) que ficheiros iniciados por '`.`' (*dot*) são ficheiros de configuração (do sistema ou de aplicações). Esses ficheiros são chamados *dot files*.

Recordar que os nomes

```
.
```

e

```
..
```

estão reservados para representar a própria directoria e a directoria sua mãe.
[TA:] Na realidade '`.`' e '`..`' são de facto dois ficheiros que são automaticamente criados e colocados dentro de cada nova directoria e que contém informação relativa às directorias em causa. Por esse motivo, na realidade, em Unix, uma directoria nunca está vazia (!)... Tem pelo menos esses dois ficheiros.
[TA:] Determinados editores de texto (por exemplo, o `vi`) permitem visualizar o conteúdo destes dois ficheiros.

[TA:] Determinados programas podem assumir algumas indicações em função da construção do nome dos ficheiros. Por exemplo, o compilador de C (`cc`) assume que os ficheiros a compilar (com código fonte C) terminam em '`.c`' e dá erro se isso não se verificar. Trata-se duma opção do programa em causa e não tem relação com nenhuma restrição do sistema operativo.

[TA:] A visualização de *dot files* é muitas vezes ocultada (dada a natureza do seu conteúdo), podendo ser necessário solicitar explicitamente a sua exibição **[***]**.

2.4. Navegação no sistema de ficheiros

[EXC:]
O comando

```
pwd
```

permite saber qual a directoria corrente ("*print current/working directory*").

[EXC:]
O comando

```
ls
```

permite listar (*ls* vem de "*list*") o conteúdo das directorias e características dos ficheiros. Sem parâmetros refere-se à directoria actual (corrente).
Pode indicar-se uma directoria como parâmetro.

```
ls /etc
```

O seu comportamento pode também ser modificado com a utilização de *switches*.

```
ls -a .
ls -a
```

mostra todos os ficheiros da directoria corrente (incluindo *dot files*).

[EXC:]
O comando `cd` faz com que a directoria corrente passe a ser a indicada como parâmetro (*cd* vem de "*change dir*").

```
cd /etc
pwd
```

Quando invocado sem argumentos faz com que a directoria corrente passe a ser a directoria pessoal do utilizador

```
cd
pwd
```

[EXC:]
O comando `cat` permite observar o conteúdo do ficheiro indicado em argumento.

```
cat /etc/passwd
```

Ter em atenção que, em princípio, só faz sentido utilizar este comando com ficheiros de texto. No entanto, como não existe nenhuma forma automática de distinguir o tipo de um ficheiro, a operação é possível mesmo com ficheiros binários. Se for indicado um ficheiro binário, o seu

conteúdo é exibido no ecrã representando em ASCII cada um dos respectivos bytes (o mesmo tratamento dado aos ficheiros de texto).
[TA:] Os "apitos" que se ouvem quando se visualiza deste modo um ficheiro binário tem a ver com a interpretação do caracter cujo código é 7 ("*bell*").

[EXC:]
Fazer alguma navegação livre no sistema de ficheiros, recorrendo a estes comandos, de modo a constatar aspectos que tenham entretanto sido referidos (tais como: estrutura, caminhos e nomes).

[TA:] Muitas vezes a janela do terminal usado é pequena para visualizar o resultado dos comandos efectuados, como acontece com algumas das listagens dos exercícios anteriores. Existem várias formas de contornar essa limitação.
A técnica mais simples consiste em recorrer às combinações de teclas `Shift+PageUp` e `Shift+PageDown` para navegar no texto já apresentado no terminal.
Outra solução passa pela utilização de um programa paginador (como o `more` ou o `less`) através de um *pipe* [***].

3. *Standard input*, *standard output* e *standard error*

3.1. *Standard input* e *standard output*

O *standard input* é a origem (entrada), por omissão, dos dados lidos pelos programas. Normalmente está associado ao teclado.
O *standard output* é o destino (saída), por omissão, dos dados escritos pelos programas. Normalmente está associado ao monitor.
É habitual chamar, abreviadamente, ao *standard input* e ao *standard output* respectivamente *stdin* e *stdout*.

A forma mais abrangente de um comando em Unix pode ser expressa como

```
xpto
```

em que o comando espera a entrada dos dados vindos do *stdin* (teclado), processa-os e envia-os para o *stdout* (monitor).
[TA:] Na realidade, muitos dos comandos do Unix (eventualmente a maioria) acabam por não seguir exactamente este comportamento mas, mesmo esses, podem ser encarados como casos particulares da forma indicada.

[EXC:]
O comando

```
cat
```

comporta-se como um filtro neutro entre *stdin* e *stdout*.
A introdução do EOF faz-se usando `Ctrl+D` (e não `Ctrl+Z`, como no DOS! Em Unix `Ctrl+Z` desencadeia uma acção completamente diferente **[***]**).
[TA:] A terminação (forçada) do processo consegue-se com `Ctrl+C` (e não `Ctrl+Z` (parece equivalente mas é muito diferente **[***]**)).

[EXC:]
O comando

```
wc
```

indica o número de linhas, palavras e bytes (inclui espaços e *newlines*) do texto introduzido (*wc* vem de "*word count*").

Também é típico que quando se indica um ficheiro na linha de comando ele seja usado em vez do *stdin* (situação mais comum) ou do *stdout*:

```
xpto ficheiro
```

[EXC:]
Executar

```
cat /etc/passwd
```

(o ficheiro indicado tem de existir).
[TA:][EXC:] No `cat`, se forem indicados vários ficheiros, eles são todos mostrados (pela ordem indicada), ou seja, eles são concatenados na saída (daí o nome `cat`, que vem de "*concatenate*").

```
cat /etc/passwd /etc/group /etc/issue
```

[EXC:]
Executar

```
wc /etc/passwd
```

(o ficheiro indicado tem de existir).

Exemplos de comandos que não seguem estritamente o comportamento de receber do *stdin* e enviar para *stdout* são: `ls`, `cd`, `pwd`, etc.

3.1.1. Redireccionamento do *stdout*

É possível solicitar que, em vez de a saída ser enviada para o *stdout* (monitor), ela seja redireccionada para um ficheiro indicado.

[EXC:]
Começar por recordar que o comando `cat` (sem argumentos) lê do *stdin* e escreve para o *stdout*.
Executar

```
cat > exp0
```

(`exp0` é criado automaticamente, não precisa de existir; se existir, o conteúdo anterior perde-se).
[EXC:]
Observar o ficheiro produzido no exercício anterior

```
cat exp0
```

Repetir ambos os exercícios (este e o anterior), para confirmar que o conteúdo prévio se perde.

Se não se desejar que o conteúdo anterior se perca, usar '`>>`' para juntar (no fim) (*append*):
[EXC:]

```
cat >> exp0
```

[TA:] Existe um ficheiro especial para o qual podem ser redireccionados as saídas que não se pretende de facto guardar (quando a ideia é simplesmente "calar" um processo). Funciona como um buraco negro.
[EXC:]

```
cat > /dev/null
cat exp0 > /dev/null
```

3.1.2. Redireccionamento do *stdin*

É possível solicitar que, em vez de a entrada de dados ser efectuada a partir do *stdin* (teclado), ela se realize a partir de um ficheiro indicado.

[EXC:]

```
cat < exp0
```

(`exp0` tem de existir).
Reparar que, de acordo com o que se anunciou como típico, é equivalente a:

```
cat exp0
```

3.2. *Standard error*

[EXC:]
O comando

```
cat nomedeficheiroquenaoexiste
```

gera uma mensagem de erro relativa à inexistência do ficheiro indicado.
Redireccionado a sua saída para um ficheiro

```
cat nomedeficheiroquenaoexiste > ta_calado
```

seria de esperar, à primeira vista, que não fosse escrita a mensagem de erro no monitor.
A questão é que existe uma outra saída definida por omissão, para onde os programas costumam enviar as suas mensagens de erro: *stderr*, que normalmente também está associada ao monitor.

Assim, os redireccionamentos do *stdout* e do *stderr* constituem aspectos distintos.

3.2.1. Redireccionamento do *stderr*

De forma análoga ao que acontece com o *stdout*, é possível redireccionar o *stderr*.
A sintaxe desse redireccionamento pode depender da *shell* usada. As considerações apresentadas neste tópico referem-se à *shell* `bash` **[***]** (que é a usada por omissão no RHL).
[EXC:]

```
cat nomedeficheiroquenaoexiste 2> ta_calado
```

(seria inválido na *shell* `csh`).

```
cat ta_calado
```

[TA:][EXC:]

```
cat exp0 nomedeficheiroquenaoexiste
```

O conteúdo do ficheiro que existe é mostrado no *stdout*. Também é mostrada uma mensagem de erro relativa ao ficheiro que não existe.

```
cat exp0 nomedeficheiroquenaoexiste 2> erros
```

O conteúdo do ficheiro que existe continua a ser exibido no *stdout*. A mensagem de erro é redireccionada para o ficheiro indicado.

[TA:] A utilização de '`2>>`' em vez de '`2>`' permite que, no caso do ficheiro indicado como destino já existir, os novos dados sejam anexados aos anteriores (ou seja, o conteúdo anterior do ficheiro não é destruído).

3.3. Combinação de redireccionamentos

Caso geral:

```
xpto < ficheirodeentrada > ficheiroparasaida 2> ficheiroparaerros
```

[EXC:]

```
cat < exp0 > exp1
```

(`exp0` é usado como entrada e é criado um ficheiro `exp1` que fica com a saída).
Esta linha de comando, na prática, cria um ficheiro exp1 que é uma cópia de `exp0`. Existe um comando específico, `cp` (de "*copy*"), para esse fim:

```
cp exp0 exp1
```

[EXC:]

```
cat < exp0 > exp1 2> erros
```

Mesmo que não haja erros a exibir, o ficheiro indicado no redireccionamento de *stderr* é criado. Se não houver erros, fica vazio.

[TA:][EXC:]

```
cat exp0 exp1 nomedeficheiroquenaoexiste > osqueexistem 2> erros
```

os ficheiros que existem são apresentados em sequência. O ficheiro indicado no redireccionamento passa conter a colagem dos respectivos conteúdos. Eventuais mensagens de erro (por exemplo relativas a ficheiros que não foram encontrados) são todas despejadas no ficheiro indicado para redireccionamento do *stderr*.

Existe ainda uma outra combinação que permite redireccionar o *stdout* e o *stderr* (em conjunto) para um ficheiro. O caso geral é:

```
xpto < ficheirodeentrada >& ficheiroparasaidaeerros
```

[EXC:]

```
cat exp0 exp1 nomedeficheiroquenaoexiste >& osqueexistemmaiserros
```

Tanto as saídas destinadas ao *stdout* quanto a destinada ao *stderr* são concatenadas no ficheiro indicado.

[TA:][EXC:]
Se na mesma linha houver a necessidade de fazer também o redireccionamento apenas do *stdout*, usar a seguinte sintaxe:

```
(cat exp0 exp1 nomedeficheiroquenaoexiste > osqueexistem) >& osqueexistemmaiserros
```

3.4. *Pipes*

Quando existe um comando que produz uma saída (dirigida ao *stdout*) que nos interessa usar como entrada de outro (que a procura no *stdin*), é possível escrever uma única linha de comando e evitar a utilização de um ficheiro temporário.
A especificação de um *pipe* faz-se colocando à esquerda o comando que produz a informação e à direita o comando que a consome. A meio introduz-se o carácter '`|`' (que também se chama "*pipe*").

```
xpto1 | xpto2
```

[EXC:]
Suponha-se que, para saber quantos ficheiros existem numa directoria, se quer contar o número de entradas que são apresentadas na respectiva listagem.
Uma possibilidade, que recorre aos comandos `ls` e `wc`, é criar um ficheiro temporário de modo a aproveitar a saída do `ls` e usá-lo como entrada para o `wc`.

```
ls –1 > aux
wc –l < aux
```

(o *switch* '`-1`' do comando `ls` garante que é listado um ficheiro por linha e o *switch* '`-l`' do comando `wc` indica que se pretende apenas contabilizar o número de linhas).
O raciocínio é que se o comando `ls` produzir uma saída com um ficheiro por linha, então a contagem de linhas dessa saída é equivalente ao número de ficheiros contidos na directoria.
O ficheiro auxiliar criado, deixa de ter interesse após a execução dos comandos anteriores, pelo que pode/deve ser removido:

```
rm aux
```

Recorrendo a um *pipe*, a resolução deste exercício poder-se-ia resumir a:

```
ls -1 | wc –l
```

Todo o comando foi feito numa única linha e não houve necessidade de criar um ficheiro temporário.
[TA:] Neste exercício, usar também o switch '`-a`' do comando `ls` no caso de se pretender que realmente todos os ficheiros sejam incluídos na contagem.

[TA:][EXC:] A saída apresentada no ecrã por

```
ls
```

é diferente do que fica no ficheiro quando se faz

```
ls > aux
```

Este tipo de situação é comum. É necessário ter este aspecto em consideração, porque em determinados casos isso pode ser relevante.

[EXC:] Filtrar a saída de um comando com um programa paginador

```
cat /etc/passwd | more
cat /etc/passwd | less
```

Podem ser colocados vários *pipes* na mesma linha de comando:

```
xpto1 | xpto2 | xpto3 | ... | xptok
```

[EXC:]

```
ls | cat | wc
ls | sort | head | tail | cat | wc | more
```

(não é muito relevante, nesta fase, saber exactamente o propósito de cada um dos comandos aqui referidos; a ideia é perceber que este tipo de encadeamento é possível).
[TA:] Ao primeiro comando basta que (em condições normais) envie a saída para o *stdout*. Ao último basta que leia do *stdin*. Os indicados no meio têm de ler do *stdin e* escrever para o *stdout*.

4. Comandos

Há três tipos de comandos que importa distinguir:

- *aliases* **[***]**: definíveis pelos utilizadores e que acabam por se traduzir em outros comandos;
- comandos internos (*builtin commands*) **[***]** da *shell*: funcionalidades da *shell*;
- programas (binários e *scripts* **[***]**): ficheiros executáveis que se encontram em alguma directoria.

Podem existir definições homónimas para comandos de diferentes tipos. No caso dos programas, até podem existir vários com o mesmo nome desde que se encontrem em diferentes directorias.
As ambiguidades são desfeitas de acordo com os seguintes critérios:

- se é indicado um caminho para o nome, então ele refere-se necessariamente ao programa contido na directoria indicada.
- não sendo indicado um caminho, a *shell* verifica a existência de um comando, com o nome indicado, pela seguinte ordem (até ter sucesso):
  - primeiro, na lista de *aliases* definidos;
  - depois, entre os seus comandos internos;
  - finalmente, numa lista ordenada, pré-definida (mas configurável), de directorias a que se chama *path*.

Naturalmente, a não ocorrência do nome em nenhum daqueles passos resulta numa situação de erro.

Existe um conjunto de comandos que pode ser útil, para o utilizador, no sentido de se reconhecer o tipo de um determinado comando ou de saber qual o comando invocado para um determinado nome:

- O comando `which` permite saber que ocorrências existem de um determinado nome entre os *aliases* e os programas contidos nas directorias indicadas no *path*

  ```
  which –a nome
  ```

  e designadamente, qual deles é utilizado (se não for explicitado um caminho)

  ```
  which nome
  ```

  Porém, este comando ignora os comandos internos da *shell*.

- O comando `whereis` indica as diversas localizações de programas (entre outros, como ficheiros de configuração) com um determinado nome, num conjunto de directorias padrão.

  ```
  whereis nome
  ```

  Não considera *aliases* nem comandos internos da *shell*.
- O comando `whatis` **[***]**, para além de fazer uma pequena descrição dos comandos, indica explicitamente quando se trata de um comando da *shell*.

  ```
  whatis nome
  ```

  Contudo, se existirem *aliases* definidos com o nome indicado não faz qualquer menção à sua existência.

Dadas as limitações/especificidades de cada um destes comandos, há quer ter cuidado com as conclusões a tirar em cada situação. Habitualmente torna-se necessário combinar a sua utilização.

Como foi dito, a definição do *path* permite que, em condições normais, para invocar os programas baste escrever o seu nome na linha de comando, não havendo necessidade de explicitar o local onde eles se encontram (desde que eles se encontrem em alguma das directorias incluídas).
Não é raro que nos sistemas Unix a directoria corrente não esteja contida no *path* e ao contrário do que acontece em DOS/Windows, a *shell* não verifica a existência do comando na directoria corrente antes de procurar nas directorias indicadas no *path*.
Assim, em geral, para executar um comando que se encontre na directoria corrente é necessário indicar explicitamente o caminho onde ele se encontra. Isso pode ser conseguido de forma simples do seguinte modo:

```
./xpto
```

(em que `xpto` é um programa contido na directoria corrente).
Uma alternativa, eventualmente mais interessante, é inserir a directoria corrente na lista de directorias que constituem o *path* **[***]**.

Segue-se uma relação mínima de comandos (programas do Unix e comandos internos das *shells*) de cujas funcionalidades os alunos devem, por sua iniciativa, tomar conhecimento. Alguns desses comandos ainda serão objecto de estudo no decurso deste texto:

- para lidar com ficheiros e directorias: `ls`, `cp`, `mv`, `rm`, `ln` **[***]**, `pwd`, `cd`, `mkdir`, `rmdir`, `chmod`, `chown`, `chgrp`, `cat`, `wc`, `grep`, `find`, `whereis`, `which`, `sort`, `more`, `less`, `head`, `tail`, `pushd`, `popd`, `umask`.
- para lidar com processos **[***]**: `ps`, `kill`, `killall`, `bg`, `fg`, `jobs`, `sleep`.
- relacionados com utilizadores e sessões: `su`, `who`, `w`, `users`, `groups`, `tty`, `finger`, `login`, `logout`, `source`, `exit`, `history`, `alias`, `unalias`, `set`, `unset`, `export`, `setenv`, `unsetenv`, `env`, `printenv`, `stty`.
- para acesso a documentação **[***]**: `man`, `whatis`, `apropos`, `info`.
- para desenvolvimento: `cc`, `vi`, `vim`, `gedit`.
- *shells* **[***]**: `sh`, `bash`, `csh`, `tcsh`.
- diversos: `echo`, `clear`, `date`, `cal`, `du`.

Relativamente aos comandos internos das *shells* ter em atenção que:

- um comando pode estar definido numa *shell* mas não estar noutra;
- mesmo que um comando esteja definido em várias *shells*, ele na realidade não é o "mesmo", mas sim a implementação da funcionalidade respectiva em cada uma delas.

[TA:] Note-se que é possível indicar mais do que um comando numa mesma linha de comando. Para tal basta separá-los por '`;`'.
[EXC:]

```
ls -1 > aux ; wc –l < aux ; rm aux
sleep 30 ; echo –e 'Passou meio minuto\a'
```

Em determinadas situações (simples) esta técnica pode ser uma alternativa à criação de um *script* **[***]**.

5. Documentação

A abundância e qualidade de informação disponível, relativa aos vários aspectos da utilização e configuração do sistema, é um dos aspectos característicos do Unix, que permite e suscita estratégias de auto-aprendizagem/autodidactismo.

A utilização da documentação deve constituir um aspecto rotineiro. Isto é válido para os comandos mas também para outras vertentes do sistema, como ficheiros de configuração, funções de programação ou software adicionado.

5.1. Acesso à documentação

Existem várias formas típicas disponíveis, que se complementam, para acesso à informação procurada:

- `man`: *manpage* do termo procurado. Forma mais característica de ajuda do Unix; disponível (também) na linha de comando. Contém descrição, *switches*, eventualmente exemplos e comandos relacionados.

- `whatis`: apenas a linha de cabeçalho da *manpage*; equivalente a '`man -f`'.

- `apropos`: indica todos os comandos que nas suas descrições, nas respectivas *manpages*, incluem o termo indicado; equivalente a '`man -k`'.

- *switch* `--help`: *switch* que tipicamente dá acesso à "*usage*" do comando. Normalmente consiste numa versão simplificada da informação disponível na *manpage*. Nem todos os comandos têm implementado.

- `info`: *infopage* do termo procurado (similar ao `man`; formato mais recente, mais conveniente).

- *how-to*'s: mais comuns quando estão implicadas instalações ou configurações (não propriamente utilização de comandos).

- Outras fontes e meios: páginas offline (**[EX:]** *Help* na interface gráfica) e online (**[EX:]** *http://www.europe.redhat.com/documentation/man-pages/*)

[EXC:]

```
man ls
whatis ls
apropos directory
cat --help
info cat
```

(normalmente usa-se '`q`' (de "*quit*") para terminar a exibição das páginas de ajuda; `Ctrl+C` pode ser útil se a saída não for paginada).

É importante ter em conta que existem vários volumes de *manpages*, que estão organizados segundo uma certa lógica. Essa distribuição pode ser visualizada, por exemplo, em:
http://www.europe.redhat.com/documentation/man-pages/

[TA:] Um mesmo termo pode encontrar-se em vários volumes o que significa que pode ser necessário indicar o volume procurado.

[EXC:]

```
whatis kill
```

indica que existe uma página no volume 1 (que é dedicado a comandos) e outra no volume 2 (que é dedicado a funções). Indica ainda que também existe um comando da *shell* com esse nome.
Ao fazer-se

```
man kill
```

seria indicada a primeira página disponível. Ou seja, seria equivalente a

```
man 1 kill
```

que diz respeito à ajuda relativa ao programa `kill`. Se se pretendesse ajuda sobre a função `kill`, ter-se-ia de fazer

```
man 2 kill
```

[EXC:]

```
whatis signal
man signal
man 2 signal
man 7 signal
```

[TA:] O `man` recorre ao programa paginador `less`, para apresentar as *manpages*. O `less` é similar ao `more`, mas é mais versátil.
Para procurar um termo dentro de uma *manpage*, escrever

```
/textoprocurado
```

(em frente a '`:`'). Para repetir a procura, usar '`n`' (de "*next*").
Para uma lista completa das funcionalidades disponíveis, usar '`h`' (de "*help*").

Tipicamente a ajuda relativa aos comandos internos de uma *shell* encontra-se inserida na ajuda da própria *shell*.
[EXC:] Chamar a manpage da *shell*

```
man bash
```

e procurar a ajuda sobre o seu comando interno `kill`.

5.2. *Switches*

Um dos aspectos mais característicos dos comandos do Unix é a abundância de *switches* que muitas vezes os tornam autênticos "canivetes suíços".

[EXC:]

```
ls
```

(listagem por omissão),

```
ls -l
```

("*long listing*"),

```
ls -a
```

("*list all*" (inclui *dot files* (ou seja os ficheiros que começam por '`.`'))).

Os *switches* podem ser combinados.
[EXC:]

```
ls -l -a
ls -la
```

[EXC:]
Verificar a semântica dos *switches* usados nos exercícios anteriores e constatar a diversidade de outras variações permitidas (usando outros *switches*)

```
man ls
```

Ter em atenção que é relevante se um determinado *switch* deve ser escrito com letra maiúscula ou minúscula (como sempre acontece no ambiente Unix).
[EXC:]

```
ls –r
```

exibe os ficheiros na directoria em ordem inversa ("*reverse*");

```
ls –R
```

lista recursivamente o conteúdo da directoria.

6. *Shell*

Uma *shell* é um programa onde o utilizador indica o que pretende fazer. Disponibiliza uma linha de comando (anunciada por um *prompt*) e faz algum pré-processamento do texto introduzido, transformando-o então nos comandos a executar. É um interpretador de linhas de comando.
De certo modo é a interface com o utilizador (principalmente se esquecermos a eventual existência de uma interface gráfica).

Em cada terminal aberto (em que se tenha feito *login*) (e em cada janela de terminal do ambiente gráfico) corre uma *shell*.

6.1. *Shells* mais conhecidas

Segue-se uma breve descrição de algumas das *shells* mais conhecidas e interessantes:

Bourne Shell (`sh`): a *shell* original do Unix, desenvolvida pela AT&T.

C-Shell (`csh`): inicialmente desenvolvida em BSD Unix (Berkeley). A sua utilização tem algumas semelhanças com a linguagem C (por exemplo na sintaxe de *scripts* **[***]** avançados). Tem muitos adeptos, principalmente entre utilizadores experimentados.

Tab C-Shell (`tcsh`): *C-Shell* melhorada (por exemplo com a capacidade de completar nomes) . Compatível com *C-Shell*. No Linux substitui a *C-Shell*.

Korn Shell (`ksh`): (desenvolvida nos Laboratórios Bell). É considerada da família da *Bourne Shell* (a sintaxe dos *scripts* é similar), mas também contempla todas as funcionalidades da *C-Shell* e da *Tab C-Shell*. Há quem a considere a mais eficiente e fácil de usar. É usada por omissão em alguns Unix. Não é instalada no RHL 9.0. (Disponível em *http://www.research.att.com/sw/download*).

Bourne-Again Shell (`bash`): *Bourne Shell* melhorada, com algumas características da *Korn Shell* e da *C-Shell*. É a versão *free* da *Bourne Shell* (projecto GNU), distribuída e usada por omissão no Linux.

O ficheiro `/etc/shells` indica as *shells* disponíveis no sistema.
[EXC:]

```
cat /etc/shells
```

[TA:][EXC:]

```
ls -la /bin/*sh*
```

Constatar, por exemplo, que não está instalada uma `csh` e uma `tcsh`. Só está a `tcsh`. Quando se chama a `csh` é de facto a `tcsh` que é chamada. `csh` é um *link* **[***]** para `tcsh`.

[TA:] Para "mudar" de *shell*, ocasionalmente, basta correr o programa associado à *shell* pretendida (`sh`, `csh`, `tcsh`, etc.), "por cima" da actual. Para terminar (e voltar à de "baixo"), executar simplesmente o comando `exit` (que é um comando da própria *shell*).
O `exit` executado na *shell* mais abaixo provoca o *logout*.
[EXC:]

```
csh
exit
exit
```

[TA:] Para alterar a *shell* usada por omissão (quando se faz *login*), existem várias possibilidades, entre as quais:

- correr o comando `chsh` (de "*change shell*") (que altera o ficheiro `/etc/passwd`);
- editar directamente o ficheiro `/etc/passwd` (operação permitida apenas ao super-utilizador).

[TA:] A alteração da *shell* por omissão, só se reflecte aquando do próximo *login* do utilizador.

[TA:][EXC:] O comando

```
finger nomedoutilizador
```

(quando disponível) mostra (entre outros) a *shell* definida por omissão para o utilizador em causa.

6.2. Funcionalidades das *shells*

É importante, no âmbito deste texto, perceber a fronteira entre os comportamentos que são advindos do sistema (nomeadamente dos seus comandos) e aqueles que são reflexo de funcionalidades da *shell* utilizada.
As *shells*, sendo interpretadores de comandos, fazem algum processamento do texto introduzido na respectiva linha de comando antes de solicitar ao sistema a execução dos comandos indicados.
Inclusivamente, as *shells* têm tipicamente um conjunto de instruções (*builtin commands*) cuja utilização, a bem da transparência, se confunde com a dos comandos do sistema.
As considerações e exercícios apresentados nos subtópicos seguintes referem-se, salvo menção em contrário, à *Bash Shell* (`bash`) (que é a usada por omissão no RHL).

6.2.1. Completamento de nomes

Esta funcionalidade encontra-se disponível em quase todas as *shells* mais recentes.
Consiste em escrever-se a parte inicial de uma palavra (comando ou ficheiro, conforme a situação) e carregar na tecla "*tab*" (`<TAB>`) para solicitar o completamento automático do texto restante. Naturalmente, esse completamento pode ser parcial ou nem poder ser efectuado devido à existência de eventuais "empates" (ou seja, quando há mais que um nome que comece pelas mesmas letras).
As *shells* permitem que em situações de "empate" se possa solicitar que seja mostrada a lista de nomes candidatos (útil quando não se sabe o resto do nome). Na `bash` premir a tecla `<TAB>` uma segunda vez. Na `tcsh`, pode usar-se `Ctrl+D` (mesmo que não seja de facto uma questão de "empate" e desde que já haja algo escrito na linha de comando).

[EXC:]
Supondo que se quer escrever '`cat /etc/passwd`'

```
cat /e<TAB>/pas<TAB>
c<TAB><TAB>a<TAB><TAB>t /e<TAB>/pas<TAB>
```

[EXC:]
Numa linha de comando vazia carregar em

```
<TAB>
```

e dizer que se quer ver todas as possibilidades.

6.2.2. Metacaracteres

Interessa conhecer os seguintes metacaracteres (*wildcards*), que as *shells* disponibilizam:

```
*
?
```

Os metacaracteres podem ser usados em qualquer posição do nome dos ficheiros, o número de vezes que se desejar.
'`*`' vale por zero ou mais caracteres. '`?`' vale por um e só um caracter.
A um termo construído com base em metacaracteres chama-se uma "máscara".
A *shell*, antes de invocar os respectivos comandos faz a expansão da máscara para o conjunto de nomes que de facto a verificam.

[EXC:]

```
ls exp*
```

A *shell* encarrega-se de substituir a máscara '`exp*`' por '`exp0 exp1`' e na realidade o que é executado é '`ls exp0 exp1`'. O `ls` vai de facto receber tantos argumentos quantos os ficheiros existentes (que se adeqúem à máscara).

[EXC:]

```
ls *p*
ls ?p?
ls ??p?
ls ??p*
ls *****p***
ls *?*exp*
```

6.2.3. História

Quase todas as novas *shells* contemplam esta funcionalidade, embora a forma de utilização não seja sempre exactamente igual.
Permite a visualização e repetição de linhas de comando já executadas. Assim, trata-se não só de um modo de rever acções passadas mas também de as repetir de forma rápida (por exemplo, por serem de sintaxe complexa e/ou longa).
É uma funcionalidade que aumenta muito a produtividade, por exemplo quando se está a programar e iterativamente se repetem os mesmos comandos (**[EX:]** chamar o editor de texto, compilar, correr o executável).

[EXC:]

```
history
!351
```

(em que `351` é o número da linha de comando que se deseja repetir);

```
!ca
```

repete o último comando executado que comece pelos caracteres '`ca`';

```
!!
```

repete o último comando;

```
!-2
```

repete o penúltimo comando.

É necessário ter em atenção que a utilização desta técnica apresenta alguns riscos, na medida em que é relativamente fácil "falhar o alvo", situação que pode ter consequências graves.
[EX:] Suponha-se que em determinado instante se usa '`!r`' com o intuito de repetir a execução de um programa chamado '`resolveproblema`' mas que, na verdade, o último comando executado começado por '`r`' tenha sido '`rm *`' (eventualmente corrido numa outra directoria).
[EX:] Aquilo que se julga ter sido o último comando, pode de facto não o ser (pode-se estar a esquecer um outro entretanto efectuado), principalmente quando se trabalha simultaneamente em vários terminais/janelas.

[EXC:] Uma situação em que é típico recorrer-se a '`!!`' é quando, depois de lançado um comando, se conclui que é necessário repeti-lo de modo a filtrar a sua saída com um paginador

```
ls /usr/bin
!! | more
```

[EXC:]
Saliente-se que `history` não é um comando do Unix mas sim um comando (uma instrução) (um *builtin command*) da *shell* que se está a usar.
Por esse motivo a pesquisa

```
whereis history
```

não exibirá a localização de nenhum ficheiro binário correspondente.

Algumas *shells* (**[EX:]** `bash` sim; `tsch` não) vão gravando a história num ficheiro, pelo que são exibidos também os comandos executados em sessões anteriores (do mesmo utilizador, naturalmente).

Se tivermos vários terminais abertos (ou várias janelas, no ambiente gráfico) cada um está a correr a sua *shell*, pelo que cada uma tem a sua *history* própria (embora possam ter em comum a história já gravada antes do seu início).

As utilização das teclas direccionais (para cima e para baixo) permite também rebuscar linhas de comando anteriores, com a possibilidade de edição antes da validação.

6.2.4. *Aliases*

Quase todas as novas *shells* contemplam esta funcionalidade, embora a sintaxe e os pormenores de utilização não sejam sempre exactamente iguais.
Um *alias* é um nome alternativo para um comando (ou mesmo para toda uma linha de comando) ("*alias*" significa "nome falso" ou "pseudónimo").
A definição de um *alias* não se traduz na criação de nenhum ficheiro (por exemplo, não consiste em nenhuma cópia ou geração de ficheiro de atalho para os comandos envolvidos).
Um *alias* torna-se disponível apenas ao nível da *shell* em que tiver sido definido.
A *shell*, ao executar uma linha de comando que contém *aliases*, procede à respectiva substituição antes da execução dos comandos correspondentes.

[EXC:] Suponha-se que dada a frequente necessidade de executar o comando

```
ls -la
```

se opta por criar um *alias* denominado `lsla`. O comando `alias` permitir definir

```
alias lsla='ls -la'
```

(sintaxe usada na bash; notar que não podem ser deixados espaços em torno do '=');

```
alias lsla 'ls -la'
```

(sintaxe usada na csh/tcsh; não usa o '=').
Em ambos os casos poderiam ser utilizadas aspas em vez de plicas.
Definido o *alias*, passa a ser possível a execução de linhas de comando como

```
lsla
lsla /etc
```

[TA:][EXC:] Seria inclusivamente possível definir

```
alias ls='ls -la'
```

Como a *shell* faz a substituição do texto do *alias* antes de proceder a qualquer invocação dos comandos envolvidos na linha de comando, nunca existiria confusão relativamente à interpretação de `ls`.

[EXC:] O comando `alias` sem argumentos permite visualizar todos os *aliases* definidos, no momento, na sessão.

```
alias
```

[TA:] [EXC:] O comando `unalias` permite "anular" um *alias* previamente definido.

```
unalias ls
unalias lsla
```

Os *aliases* são válidos apenas para a sessão em que são definidos. É contudo possível colocar a respectiva definição num ficheiro de configuração **[***],** de modo a que passem a ser definidos de forma automática, em futuras sessões.

6.2.5. Outras funcionalidades

As *shells* disponibilizam muitas outras funcionalidades que se podem revelar interessantes em determinadas situações. Coerentemente com os propósitos deste texto, remete-se para os alunos a tarefa da respectiva exploração, o que pode ser conseguido, nomeadamente, pela consulta das *manpages* das próprias *shells*.

[EX:]

```
jobs
```

auxilia significativamente a manipulação de processos **[***]**.

[EX:] Nas *shells* mais recentes (e na csh, pelo menos) o caracter '~' pode ser usado para representar a directoria pessoal do utilizador.
[EXC:]

```
ls ~/exp*
```

7. Multi-utilização

Neste tópico são abordados alguns aspectos relacionados com o relacionamento entre os vários utilizadores de um sistema e os objectos (ficheiros) nele existentes.

Uma das formas possíveis de constatar que utilizadores estão definidos num sistema é observar o conteúdo do carismático ficheiro `/etc/passwd`.
[EXC:]

```
cat /etc/passwd
```

Existem também formas de saber (com maior ou menor detalhe) que utilizadores se encontram realmente a usar o sistema, no momento em causa.
[EXC:]

```
users
who
w
finger
```

Os utilizadores encontram-se contidos em grupos.
Os grupos encontram-se definidos em `/etc/group`.
[EXC:]

```
cat /etc/group
```

A informação relativa ao grupo inicial de cada utilizador pode ser visualizada no próprio `/etc/passwd`.
[TA:] Na realidade, um utilizador pode pertencer a vários grupos.
[EXC:] O comando

```
groups
```

indica o(s) grupo(s) a que utilizador pertence.

[TA:] Quando no processo de criação de um utilizador não é explicitado o grupo (inicial) a que ele deve pertencer, normalmente é criado automaticamente um novo grupo cujo nome (e número (ID)) é igual ao nome do utilizador. É por esse motivo que muitas vezes se vê o nome do utilizador no nome do grupo.

7.1. Permissões e Pertenças

Cada objecto tem definido um conjunto de permissões que dita as acções que podem ser levadas a cabo, sobre ele, pelos utilizadores.

Estão definidas três classes de utilizadores:

- o *owner* (dono) do ficheiro: *user* (u);
- os utilizadores que pertencem ao mesmo grupo do *user*: *group* (g);
- todos os outros: *other* (o).

Estão definidas três tipos de acções:

- *read* (r);
- *write* (w);
- *execute* (x).

O significado dessas acções é diferente consoante se apliquem a um ficheiro ou a uma directoria:

- Ficheiro:
  r: visualizar o conteúdo do ficheiro;
  w: alterar o conteúdo do ficheiro ou remover o ficheiro;
  x: executar o ficheiro (relevante para programas binários e *scripts*);
- Directoria:
  r: listar ficheiros dentro da directoria;
  w: criar ou remover ficheiros na directoria;
  x: mudar para a directoria (com `cd`, por exemplo).

7.2. Expressão das permissões

As permissões de um objecto são expressas usando nove bits representando os três tipos de acção para cada uma das três classes de utilizadores.
A apresentação desses nove bits pode tomar várias formas, designadamente:

- três sequências de caracteres '`rwx`' (respectivamente para `u`, `g`, `o`). Quando o bit se encontra a '1' (quando a permissão existe) é inscrito o caracter; quando não, o seu espaço é preenchido com caracter '`-`'.
  [EX:]

```
rwxrwxrwx
rwxrw-r--
r--r-----
--------x
```

- três dígitos octais, cada um representando um dos conjuntos de três bits (`rwx`).
  [EX:]

```
777
764
440
001
```

(Nos exemplos acima há combinações que semanticamente não farão grande sentido, mas saliente-se que são possíveis).

[EXC:] Observar as permissões associadas aos ficheiros da directoria pessoal.
Observar também outros aspectos como o nome do dono e o grupo de cada ficheiro.

```
cd
ls -la
```

[TA:] Reparar que, na listagem assim apresentada, existe um caracter imediatamente antes (junto) da expressão das permissões de cada objecto. Esse caracter não está relacionado com as permissões (nem sequer é um bit). Trata-se da representação do tipo de ficheiro ('-'

significa ficheiro regular (normal); ‘`d`’ significa directoria; ‘`l`’ significa *link* **[***]**; ‘`p`’ significa *pipe* nomeado **[***]**; etc.).

As permissões nunca referem utilizadores (ou grupos) concretos (ao contrário da abordagem do Windows). Referem-se apenas a “quem for o dono” e a “quem pertencer ao grupo dono”.

Dois objectos (ficheiros ou directorias) podem ambos ter as permissões `700` e um utilizador ter todos os direitos sobre esse ficheiro e nenhuns sobre o outro... Basta que ele seja dono do primeiro e não o seja desse outro.
[EXC:] Observar as permissões das directorias contidas na directoria acima da directoria pessoal do utilizador (tipicamente as directorias dos outros utilizadores da máquina).

```
ls -l ~/..
```

Observar também os donos e grupos de cada uma dessas directorias. Em princípio cada directoria pessoal será pertença do respectivo utilizador.
Ter em atenção que, apesar de as expressões das permissões dessas várias directorias serem idênticas (tipicamente ‘`rwx------`’), as permissões de um determinado utilizador sobre elas são completamente diferentes. A questão é que as permissões são de facto similares mas para os respectivos donos. Como cada directoria pessoal tem um dono diferente (o respectivo utilizador), resulta que as permissões são diferentes para todos os outros utilizadores.

[EXC:] Observar as pertenças e permissões em outras directorias do sistema.
[EX:]

```
ls -l /
```

[TA:] Um ficheiro (ou directoria) só pode ter um dono e um grupo dono.

[TA:] O “grupo dono” normalmente é o “grupo do dono”, mas isso não é forçoso. Pode ser um grupo ao qual o dono nem sequer pertence.

7.3. Alteração das pertenças

As alterações de pertenças podem ser operadas por recurso aos comandos `chown` (de “*change owner*”) e `chgrp` (“*change group*”):

```
chown utilizador nomedoficheiro
chown utilizador:grupo nomedoficheiro
chown :grupo nomedoficheiro
```

Esta última linha é equivalente a:

```
chgrp grupo nomedoficheiro
```

Reparar que a alteração do dono de um ficheiro *não* provoca nenhuma alteração automática do grupo (por exemplo para o (um) grupo a que o dono pertença).

A capacidade de usar `chmown` e `chgrp` para “oferecer” pertenças a outros utilizadores pode variar com a versão de Unix utilizada. Enquanto há versões que o permitem a qualquer utilizador, outras (como é o caso do Linux), por questões de segurança (por exemplo), apenas o permitem ao super-utilizador.

Assim, em Linux um utilizador normal apenas pode alterar as pertenças para si próprio (e para o seu grupo) em ficheiros que já lhe pertençam (o que, na prática, não tem qualquer interesse).
[EXC:] Testar a veracidade, no que diz respeito ao Linux, do parágrafos anteriores.

7.4. Alteração das permissões

A alteração das permissões processa-se utilizando o comando `chmod` (de "*change mode*"):

```
chmod alteraçãodaspermissões ficheiro
```

Existem várias formas de especificar as alterações das permissões.

No caso de se pretender ignorar as permissões actuais, basta indicar em octal as novas permissões.
[EXC:]

```
ls -l exp0
chmod 600 exp0
ls -l exp0
```

No caso de se pretender alterar determinadas permissões, mantendo as restantes, pode usar-se o caracter '`+`' para acrescentar e o caracter '`-`' para retirar as permissões indicadas pela respectiva letra ('`r`', '`w`' ou '`x`') às classes indicadas também pela letra correspondente ('`u`', '`g`', ou '`o`').
[EXC:]

```
ls -l exp0
chmod u+rwx exp0
ls -l exp0
chmod ug+r exp0
ls -l exp0
```

A letra '`a`' (de "*all*") pode ser usada em vez da combinação '`ugo`'.

```
chmod a-rw exp0
ls -l exp0
```

Existe uma outra forma de especificar a alteração de permissões, em que não são indicadas as classes de utilizadores, sendo subentendido que todas elas devem ser alteradas.
[EXC:]

```
ls -l exp0
chmod +x exp0
ls -l exp0
chmod -x+r exp0
ls -l exp0
```

[TA:][EXC:] No entanto, a alteração de permissões assim processada não é exactamente equivalente às referidas anteriormente.
[EX:] Ao contrário do que acontece com

```
chmod 777 exp0
```

ou

```
chmod a+rwx exp0
```

ou

```
chmod ugo+rwx exp0
```

ao fazer-se

```
chmod +rwx exp0
```

pode de facto não se ficar exactamente com '`rwxrwxrwx`'. Pode ficar por exemplo '`rwxrwxr-x`'. Isto está relacionado com a *umask* definida (`u=rwx,g=rwx,o=rx`)**[***]**. Ou seja, quando não se utiliza a forma octal ou quando não se indica explicitamente o destino das permissões, o resultado é condicionado pela *umask* definida.

8. Ficheiros executáveis

Em DOS/Windows o sistema operativo decide se um ficheiro é executável em função da sua extensão (`.COM`, `.EXE`, `.BAT`).
Em Unix o nome do ficheiro não tem qualquer relevância para esse efeito (e nem sequer faz sentido falar de extensões). Um ficheiro é identificado como executável se tiver permissões de execução.

[EXC:]

```
ls -l /bin
```

Reparar que os ficheiros contidos nesta directoria têm todos permissões de execução (tipicamente).

Existem dois tipos de ficheiros executáveis (programas): aqueles a que normalmente se chama de binários e os *scripts*.

Exemplos de binários são os programas produzidos como resultado da compilação do código fonte C ou os comandos do Unix.
[TA:] Notar que a aplicação do termo "binários" para designar este tipo de programas é algo abusiva, na medida em que (obviamente) também existem ficheiros de dados em formato binário (todos aqueles que não são de texto (por exemplo, uma imagem em formato `.gif`)). Assim, seria mais correcto falar de "executáveis binários".

Os *scripts* (que são similares aos ficheiros de *batch* do DOS/Windows) são ficheiros de texto nos quais, tipicamente, é inscrito um comando em cada linha. As diferentes *shells* possibilitam que sejam incluídas também construções algo similares às encontradas nas linguagens de programação de alto nível (passagem de parâmetros, ciclos, saltos, condições, etc.).

8.1. Criação de *scripts*

O primeiro passo na elaboração de um *script* é a criação do ficheiro de texto que contém as instruções a executar.
[EXC:] Criar um ficheiro com as seguintes linhas:

```
echo Hello, World!
echo Segue-se um ls da directoria corrente
ls
```

Para criar o ficheiro pode recorrer-se a um editor de texto ou usar outra qualquer solução, como

```
cat > umscript
```

Seguidamente, existem várias possibilidades (não necessariamente equivalentes) para correr o *script*.

Uma forma simples de correr um *script* é executá-lo como se de um binário se tratasse.
[EXC:]Tentar executar o ficheiro já construído

```
./umscript
```

Surgirá o erro '`permission denied`', já que se tentou executar um ficheiro para o qual não foi definida essa permissão.
Verificar as permissões actuais

```
ls -l umscript
```

Acrescentar, então, permissão de execução para o seu dono

```
chmod u+x umscript
```

Passará a ser possível executar o *script*

```
./umscript
```

Uma forma alternativa de correr um *script*, sem haver a necessidade de lhe dar permissões de execução, é lançar uma *shell* à qual se dá como parâmetro esse *script*.
[EXC:]

```
chmod a-x umscript
```

(só para evidenciar que não há necessidade de permissões de execução)

```
sh umscript
csh umscript
```

[TA:] Quando um *script* é lançado (de qualquer uma das formas indicadas), ele corre sobre uma nova *shell*.
No caso de ser conveniente que o *script* corra usando a *shell* em que é chamado, pode-se recorrer ao comando `source`. Esta técnica também não exige que sejam definidas permissões de execução.
[EXC:]

```
source umscript
```

O comando `source` é uma instrução das *shells* (não é um comando do Unix, propriamente).
[EXC:]
Considerando um *script* `outroscript` com o conteúdo

```
alias z='ls –l'
```

se ele for corrido como

```
./outroscript
```

(depois de configuradas as permissões de execução), ou como

```
bash outroscript
```

o *alias* `z` não fica disponível após a terminação do *script*, já que ele foi definido no âmbito de uma *shell* que deixou de existir quando o próprio *script* terminou.

No entanto, se o mesmo *script* for corrido como

```
source outroscript
```

o *alias* continua, de facto, disponível, porque os comandos do *script* foram executados sobre a *shell* original.

[TA:] Quando um *script* contém comandos, instruções ou sintaxe específicos de uma determinada *shell*, é possível forçar que essa *shell* seja usada durante a sua execução. Para tal, indicar a *shell* a seguir a "`#!`", logo no início do *script*.

[EXC:]

```
#!/bin/csh
alias z ls
z
```

[TA:] Quando a *shell* usada para correr um *script* não é explicitada, aquela que de facto é usada não é necessariamente do mesmo tipo daquela em que o *script* foi invocado. No RHL 9.0, nestas situações, é usada a `bash`.

8.1.1. *Scripts* avançados

Não existe margem na disciplina para se estudar, nas aulas, a elaboração de *scripts* que envolvam as funcionalidades avançadas disponibilizadas pelas *shells*. Os alunos devem tomar consciência da sua utilidade e procurarem, por sua iniciativa, desenvolver esta vertente.
A título meramente ilustrativo do "aspecto" de um *script* avançado apresentam-se alguns exemplos. Notar peculiaridades das sintaxes consoante a *shell* utilizada.

[EX:] Este *script* inicia um processo sem fim (ciclo infinito).

```
#!/bin/csh
ciclo:
goto ciclo
```

Para forçar a terminação do processo usar `Ctrl+C`.

[EX:] *Script* que permite calcular o valor do factorial do número indicado (usando a `tcsh`).

```
#!/bin/tcsh
if ($# != 1) then
   echo "Script para calcular factoriais"
   echo "Usage: $0 n"
   echo "com n entre 0 e 12 (para o resultado caber em 32 bits)"
   exit 1
endif

@ indice = 1
@ fim = $1
@ factorial = 1
while ($indice <= $fim)
   @ factorial *= $indice
   @ indice++
end
echo "$1! = $factorial"
```

[EX:] *Script* que permite calcular o valor do factorial do número indicado (usando a bash).

```
#!/bin/bash
if test $# != 1
then
   echo "Script para calcular factoriais"
   echo "Usage: $0 n"
   echo "com n entre 0 e 12 (para o resultado caber em 32 bits)"
   exit 1
fi

indice=1
fim=$1
factorial=1
while [ $indice -le $fim ]
do
   factorial=$(($factorial*$indice))
   indice=$(($indice+1))
done
echo "$1! = $factorial"
```

9. Links

Existem dois tipos de links em Unix: *hard links* e *links* simbólicos. Na realidade, trata-se de duas funcionalidades bastante distintas, que importa estudar.

9.1. *Hard Links*

Um aspecto de extraordinária relevância em Unix é o facto de a identificação unívoca dos ficheiros ser feita com base numa estrutura de dados, conhecida por *i-node*, e não com base no seu nome.
Os nomes, pelos quais os ficheiros são conhecidos, são apenas definidos no âmbito das directorias em que foram criados, nomeadamente dentro do ficheiro '.'; ou seja, essa informação é local a cada directoria. No ficheiro '.' é estabelecida a relação entre o nome utilizado na directoria e o *i-node* do ficheiro.
Os *i-nodes* são guardados numa área do disco denominada de "área de *i-nodes*".

Na realidade, o ficheiro (*i-node*) não "sabe" por que nomes é conhecido nas diversas directorias. A informação que é guardada, a esse respeito, a nível do *i-node*, é apenas a *quantidade* de nomes que o referem no conjunto de todas as directorias.
Isso significa, entre outros aspectos, que um mesmo ficheiro pode ter diferentes entradas, eventualmente em diferentes directorias, eventualmente também com diferentes nomes.

Quando um ficheiro é criado, é-lhe associado um *i-node* e um nome.
A partir desse momento, torna-se possível definir outros nomes, eventualmente noutras directorias.
A existência de um ficheiro justifica-se enquanto existir algum nome que o refira.

Os comandos de remoção, como o rm, visam, em geral, a eliminação do *nome* indicado na operação.

A eliminação do *i-node*, ou seja, do ficheiro propriamente dito, só ocorre quando é removido o último (ou único) nome a ele associado (em todo o sistema de ficheiros).
Como em geral a cada ficheiro (*i-node*) está associado apenas um nome, o processo de remoção do nome é simplesmente referido como remoção do ficheiro.

Um dos campos da estrutura de dados *i-node*, é um número único que permite a sua identificação. Muitas vez, por comodidade, chama-se simplesmente *i-node* ao número do *i-node*.

Chama-se "*hard link*" (ou simplesmente "*link*" se for implícito o seu tipo) a um nome alternativo para um ficheiro. Em rigor, qualquer nome de um ficheiro (ou seja também o nome com que foi criado) é um *hard link*, na medida em que não existe qualquer diferença, para quaisquer efeitos, entre o primeiro nome e eventuais nomes seguintes.

Podem ser criados *hard links* adicionais para um ficheiro recorrendo ao comando `ln`.

```
ln nomejaexistente novonome
```

É importante ter em atenção que características dos ficheiros, como as suas permissões e pertenças, são guardadas no *i-node* e não nas directorias junto com os nomes. Desse modo, alterações que se façam sobre o ficheiro (mesmo que recorrendo ao seu nome), tem efeito no próprio *i-node*, pelo que se tornam efectivas e observáveis a partir de qualquer dos nomes existentes nas várias directorias.

O comando `ls` com os *switches* '`-l`' ("*long listing*") e '`-i`' ("*i-node*") é útil para a visualização e compreensão dos aspectos referidos.

[EXC:] A operação

```
cat > exp2
```

gera um novo ficheiro (um novo *i-node*) e cria uma entrada na directoria corrente usando o nome `exp2`. No *i-node* fica registado que existe, no momento, um único *hard link* para ele.
A saída de

```
ls -li exp2
```

exibe algo como

```
1908 -rw-rw-r--  1   aluno     alunos     9   Oct 17 10:54 exp2
```

O número da esquerda é o *i-node* (número do *i-node*). O número indicado a seguir às permissões é a quantidade de *hard links* que no momento estão definidos para o ficheiro em causa.

[EXC:] Criar outro nome (outro *hard link*), chamado `exp3`, para o ficheiro criado anteriormente

```
ln exp2 exp3
```

Listar os ficheiros

```
ls -li exp?
```

e observar as informações apresentadas:

```
1908 -rw-rw-r--  2   aluno     alunos     9   Oct 17 10:54 exp2
1908 -rw-rw-r--  2   aluno     alunos     9   Oct 17 10:54 exp3
```

Reparar que em ambos é mostrado o mesmo número de *i-node* (porque se trata do mesmo ficheiro). O número de nomes associado ao ficheiro, agora exibido, é dois. As informações restantes (tamanho do ficheiro, data e hora) são as mesmas já que estão associadas ao *i-node* e não ao nome.

[EXC:] Observar o conteúdo (dados) do ficheiro a partir de qualquer um dos nomes

```
cat exp2
cat exp3
```

Com um editor de texto, alterar o conteúdo do ficheiro (acedendo por um dos nomes) e repetir as instruções anteriores.

[EXC:] Fazer alterações no ficheiro a partir de um dos seus nomes

```
chmod 700 exp3
chown tempuser:users exp3
```

(em Linux, esta última linha só pode ser realizada com direitos de super-utilizador).
Constatar que as alterações ocorrem a nível do ficheiro pelo que são visíveis a partir de ambos os nomes:

```
1908 -rwx------  2   tempuser  users      9 Oct   17 10:54 exp2
1908 -rwx------  2   tempuser  users      9 Oct   17 10:54 exp3
```

[EXC:] Eliminar sucessivamente os vários nomes

```
rm exp3
```

Listar de novo

```
ls -li exp?
```

e observar as informações

```
1908 -rwx------  1   tempuser  users      9 Oct   17 10:54 exp2
```

Remover o último nome restante

```
rm exp2
```

A partir desta acção, o ficheiro deixa de ser acessível (deixou de haver algum nome para o referir) e o respectivo *i-node* foi de facto eliminado do sistema de ficheiros.

[EXC:] Realizar experiências análogas envolvendo simultaneamente várias directorias, de modo a fazer as mesmas constatações.

9.2. *Links* simbólicos

Existe um tipo de *links* bastante distinto daquele que foi referido no subtópico anterior. A este tipo chama-se "*link* simbólico" *(*ou simplesmente "*link*" se for implícito o seu tipo).
Para utilizadores de MacOS ou de Windows a sua compreensão é em geral fácil na medida em que é similar aos respectivos "ficheiros de atalho".

Trata-se da criação de um (novo) pequeno ficheiro que tem como propósito permitir o acesso a um outro ficheiro (já existente). É comum dizer-se que o novo ficheiro "aponta" para um outro ficheiro.
Normalmente, a ideia é evitar fazer uma cópia do ficheiro apontado ou fazer o mapeamento entre ficheiros.

A criação de *links* simbólicos também se consegue por recurso ao comando `ln`, mas utilizando o *switch* '`-s`':

```
ln -s nomedoficheiroexistente nomedoatalho
```

[EXC:] Criar um novo ficheiro e um *link* simbólico que aponte para ele.

```
cat > exp4
ln -s exp4 exp5
```

(assumindo que `exp4` existe na directoria corrente).
Listar os ficheiros em causa

```
ls -li exp?
```

e reparar atentamente na informação apresentada

```
1857 -rw-rw-r--  1  aluno  alunos  39  Oct 17 12:37 exp4
2045 lrwxrwxrwx  1  aluno  alunos   4  Oct 17 12:40 exp5 -> exp4
```

da qual devem salientar-se os seguintes aspectos:

- números de *i-node* diferentes: são de facto dois ficheiros distintos;
- contador de *hard links* não foi incrementado: a criação de *links* simbólicos não gera nenhum *hard link* no ficheiro apontado (são técnicas diferentes);
- tipo de ficheiro do *link* simbólico: os *links* simbólicos são marcados com '`l`';
- permissões do *link* simbólico são '`rwxrwxrwx`': as permissões de um *link* simbólico têm sempre esta forma e não são alteráveis; as permissões controláveis a partir do *link* simbólico são as do ficheiro apontado.
- tamanhos diferentes: um *link* simbólico é sempre um pequeno ficheiro (a informação que guarda é basicamente a localização do seu apontado); o tamanho do apontado está apenas relacionado com o seu próprio conteúdo;
- forma gráfica que mostra o ficheiro apontado: na linha do *link* simbólico, após '`->`', é exibido o ficheiro apontado

[EXC:] Quando se tenta utilizar o *link* simbólico é de facto o ficheiro apontado que é usado.

```
cat exp4
cat exp5
```

[EX:][EXC:] Reparar que embora em Linux seja possível chamar a *C-Shell* e a *Bourne Shell* como

```
csh
sh
```

na realidade esses programas não existem no sistema. No entanto, estão definidos *links* simbólicos com esses nomes que apontam para os respectivos substitutos.
Observar os ficheiros em causa:

```
ls -l /bin/*sh*
```

Neste caso, a utilização dos *links* simbólicos permite tornar transparente para o utilizador (e para outros programas) o facto de em substituição de cada uma daquelas *shells* clássicas serem usadas outras (compatíveis e melhoradas).

[EXC:] Tentar alterar as permissões de um *link* simbólico.

```
chmod 400 exp5
```

Listando os ficheiros conclui-se que as permissões representadas no *link* simbólico (777) não são alteradas. A alteração é feita no ficheiro apontado.
[TA:] No entanto, é possível alterar as pertenças do *link* simbólico. Essa operação não têm reflexo nas pertenças do ficheiro apontado.

```
chown tempuser:users exp5
```

(em Linux, esta última linha só pode ser realizada com direitos de super-utilizador).

Um *link* simbólico pode ser removido sem que haja quaisquer implicações no ficheiro apontado.
Por outro lado, a remoção do ficheiro apontado também não implica a remoção do *link* simbólico. No entanto, esse *link* fica quebrado ("*broken link*").
Operações que impliquem o acesso ao ficheiro apontado por um *link* quebrado, resultam naturalmente em erro.
[EXC:]

```
rm exp4
cat exp5
```

Em algumas situações (dependendo das configurações da *shell* e do terminal) os *links* quebrados, quando listados, são exibidos com um visual que permite identificar esse estado.

10. Controlo de processos

Um processo é o decurso da execução de um programa.
Um programa é um ficheiro que contém as instruções a ser executadas. Um programa pode ser invocado várias vezes; em cada uma dessas vezes é gerado um novo processo.

A cada processo está associado um número único (que não está atribuído a nenhum outro processo), a que se chama PID (de "*Process IDentity*"). É comum que os PIDs sejam atribuídos sequencialmente, por ordem crescente, à medida que os processos vão sendo gerados, mas isso não deve ser tido como regra.

Cada processo é gerado por um outro processo. Quando um processo gera outro, é comum chamar-se "pai" ao processo gerador e "filho" ao processo assim gerado.
[TA:] A excepção é o processo "pai de todos" que se chama `init` e cujo PID é 1.
Todos os processos conhecem o PID do seu pai. Esse valor é conhecido por PPID ("*Parent PID*").

10.1. Monitorização de processos

Em cada instante existem inúmeros processos em execução, num sistema. Alguns desses processos estão relacionados com tarefas do próprio sistema e dos serviços nele instalados, outros estão relacionados com os vários utilizadores, etc. Dada essa diversidade, é comum que as listagens de processos exibidas sejam filtradas, segundo algum critério, de modo a que apenas sejam visíveis os processos relevantes para determinado efeito.

O comando `ps` (de "*process status*") permite visualizar os processos em execução.

É importante conhecer alguns dos *switches* do comando ps. Tenha-se em consideração que os *switches* deste comando têm algumas variações entre sistemas/versões de Unix. Os exemplos aqui referidos dizem respeito ao RHL 9.0:

- para filtragem dos processos:
  - '-A': todos os processos;
  - '-t': os processos no *tty* indicado; 't': (ver nota);
  - '-u', '-U': os processos do utilizador indicado; 'U': (ver nota);
  - '-C': os processos com o nome indicado;
  - '-p': o processo cujo PID é indicado; 'p': (ver nota);
  - '-m': todas as *threads*; 'm': (ver nota);
- para formatação da saída:
  - '-f': formato completo;
  - '-l': formato longo; 'l': (ver nota);
  - '-H': ilustração da hierarquia dos processos (em forma de árvore);
  - 's': informação sobre os sinais associados a cada processo **[***]**;
  - 'e': variáveis de ambiente definidas para cada processo;
- ajuda:
  - '--help': *switches* num formato bastante mais interessante para uma consulta rápida do que na *manpage*.

Nota: **[TA:]** Neste comando, os *switches* indicados que não usam o prefixo '-' seguem o estilo do Unix BSD e apresentam, em geral, uma saída mais completa.

[EXC:] Num terminal, suponha-se *tty1*, executar

```
man ps
```

Mudar para outro terminal e experimentar as seguintes combinações

```
ps
ps –A
ps -u aluno
ps U aluno
ps -f -u aluno
ps -t tty1
ps -fHt tty1ps -C bash -C mingetty
```

A observação dos PID e PPID dos vários processos permite perceber as relações de hierarquia entre eles.

[EXC:] A utilização do comando ps é uma forma comum de se averiguar que *shell* está a ser utilizada em determinado momento. No caso de haver vários níveis de *shells* a avaliação dos PID (e PPID) é suficiente para esclarecer as dúvidas.

```
ps
csh
bash
ps
```

10.2. Terminação de processos

Na gíria do domínio, chama-se "matar" um processo ao acto de o terminar de forma não programada (ou seja, de forma abrupta (embora, em rigor, este tipo de terminação também poder ser programada)). Os termos "matar" e "terminar" são usados de forma indiferenciada no texto que se segue.

A forma mais simples de terminar um processo consiste em recorrer a `Ctrl+C`.
[EXC:]

```
cat > naovaichegarasercriado
<Ctrl+C>
```

No entanto, para que esta técnica seja aplicável é necessário que o processo em causa se encontre a correr em primeiro plano.
Também existem situações em que o processo programa um tratamento diferente para este controlo, ou até situações de falha, que inviabilizam esta abordagem.

Uma forma alternativa de matar um processo consiste em recorrer aos comandos `kill` ou `killall`.

Recorrendo a `killall` tudo o que é necessário é conhecer o nome do processo a matar e dispor de uma linha de comando onde correr a instrução.
[EXC:] Num terminal/sessão (por exemplo *tty1*) fazer

```
cat > naovaichegarasercriado
```

(sem concluir o processo).
A partir de outro terminal/sessão (por exemplo o *tty2*) executar

```
killall cat
```

Voltar a *tty1* e constatar a terminação do processo
[EXC:]
Criar o *script* `cicloinfinito` com o seguinte conteúdo:

```
#!/bin/csh
inicio:
goto inicio
```

Lançar o *script* num terminal/sessão (seja em *tty1*)

```
./cicloinfinito
```

A partir de outro terminal/sessão (seja *tty2*) executar

```
killall cicloinfinito
```

Voltar a *tty1* e constatar a terminação do processo

Como o nome sugere, o comando `killall` termina ("mata") *todos* os processos que tiverem o nome indicado. Por esse motivo, tendo em conta todos os processos do utilizador (no conjunto de todas as suas sessões), esta abordagem é conveniente quando estivermos na presença de uma das seguintes situações:

- existe apenas um processo com o nome indicado;
- existem vários processos com o nome indicado, mas pretende-se que todos eles sejam terminados.

Uma alternativa, que permite matar um determinado processo (específico), é a utilização do comando `kill`. Este comando implica que o processo seja identificado pelo seu PID.
[EXC:] Num terminal/sessão (por exemplo *tty1*) fazer

```
cat > naovaichegarasercriado
```

(sem concluir o processo).
A partir de outro terminal/sessão (por exemplo o *tty2*) executar `ps`, com *switches* que permitam listar o processo, por exemplo

```
ps –t tty1
```

e localizar o processo e o seu PID.
Seja `<pid>`, neste e nos exercícios seguintes, a representação do PID (um número) do processo em causa.
Executar

```
kill <pid>
```

Voltar a *tty1* e constatar a terminação do processo.

[***]A terminação de processos, usando os procedimentos descritos, são casos particulares de uma funcionalidade designada de envio de sinais.
Basicamente, quando se envia um sinal a um processo, é desencadeada uma acção, definida por omissão para o sinal em causa, que em geral se traduz na terminação desse processo (embora também existam sinais aos quais correspondem outras acções, como colocar o processo em estado "*stopped*"). Porém, é possível programar um processo para fazer a intercepção desses sinais de modo a que sejam desencadeadas outras acções (há, contudo, sinais não são passíveis de ser interceptados).
Assim, se a recepção do sinal estiver prevista no código do processo alvo, é executada uma função para isso especificada e a recepção desse sinal já não dita, por si, que o processo morra (ou outro comportamento definido por omissão para o sinal).
Os comandos `kill` e `killall` enviam, por omissão, o sinal designado por `SIGTERM`, que normalmente não é previsto pelos programas e que resulta na respectiva morte.

[EXC:]Listar todos os sinais enviáveis aos processos

```
kill -l
```

[EXC:] Colocar um processo a correr (repetir o cenário dos exercícios anteriores) e enviar-lhe outros sinais, por exemplo:

```
kill -s USR1 <pid>
kill -USR2 <pid>
kill -s 14 <pid>
kill -19 <pid>
```

Reparar que quando é usado o nome do sinal, deve ser omitido o prefixo '`SIG`'.

De acordo com o que foi explicado, por vezes um `kill` "normal" não é suficiente para matar um processo, nomeadamente se ele interceptar o sinal `SIGTERM`. Nesses casos, usar explicitamente o sinal `SIGKILL` (que não é interceptável e que leva à terminação do processo):

```
kill -9 <pid>
kill -KILL <pid>
```

Esta situação é típica quando o processo em causa é uma *shell*.

[TA:] Embora exista(m) o(s) programa(s) `kill` (em `/bin` e/ou em `/usr/bin`), `kill` é também um comando interno da *shell*, pelo que, se não for indicado explicitamente um caminho, é este último que é invocado.

10.3. Planos de execução

Em condições normais, um processo é lançado em primeiro plano ("*foreground*"). Em termos práticos isso significa que, enquanto o processo não terminar, a *shell* não volta a apresentar o *prompt* e portanto não é possível lançar um novo processo.

[EXC:]

```
sleep 1d
```

[EXC:]

```
ls / -R
```

Muitas vezes, pela natureza do processo corrido, é desejado que ele seja lançado de modo a que permita que a *shell* continue a dar a possibilidade de executar, em concorrência, outros processos. Quando um processo se encontra a correr nessas circunstâncias diz-se que está em segundo plano ("*background*").

[EXC:]

```
sleep 1d &
ps
```

Quando um processo se encontra em segundo plano não tem possibilidade de receber dados do *stdin*. Os dados introduzidos no *stdin* são sempre dirigidos ao processo que se encontra em primeiro plano, que eventualmente até pode ser a *shell* (linha de comando).

[EXC:]

```
cat > exp6 &
```

Um aspecto que com frequência gera confusão entre os utilizadores menos experimentados é a relação entre o facto de um processo se encontrar em segundo plano e a utilização que faz do *stdout* (e *stderr*).

Na verdade, ao se enviar um processo para segundo plano não se está a "calá-lo". Um processo que se encontre nessa situação continua a poder enviar dados para o *stdout* (e *stderr*).

[EXC:]

```
ls / -R &
```

Reparar também que nesta situação `Ctrl+C` não resulta para matar o processo, porque ele não está em primeiro plano. Na realidade, a *shell* está a exibir a linha de comando só que, com a quantidade de informação "despejada" pelo comando, não é fácil usá-la. Para terminar o processo, tentar escrever (mesmo que não se consiga ler o que se escreve)

```
killall ls
```

É um facto que, em determinadas situações, o envio de texto para o monitor, por parte de um processo em segundo plano, pode ser inconveniente e até tornar inviável a utilização da linha de comando. Nessas situações o redireccionamento prévio das saídas pode constituir uma solução.

[EXC:]

```
((ls / -R | grep "sh") > resultado 2> /dev/null; echo "Já está." )&
```

10.3.1. Mudança de plano

Por vezes é conveniente passar um processo, que já se encontra a correr, de primeiro para segundo plano e vice-versa. Um exemplo é quando, por lapso, o processo é lançado da forma não pretendida.

[EXC:] Passagem de primeiro para segundo plano.
Lançar um processo em primeiro plano

```
cat > exp7
```

(introduzir algum texto mas não concluir).
Suspender o processo com `Ctrl+Z`.
Executar `bg` (de "*background*")

```
bg
```

(reparar que o *prompt* passa a estar disponível e que texto que agora seja escrito já não é destinado ao `cat`).

[EXC:] Passagem de segundo para primeiro plano.
Lançar um processo para segundo plano

```
cat > exp8 &
```

(reparar que o *prompt* passa a estar disponível e que texto que agora seja escrito já não é destinado ao `cat`).
Executar `fg` (de "*foreground*")

```
fg
```

(reparar que já não existe *prompt* disponível e que o texto é destinado ao `cat`).

Os comandos `bg` e `fg` referem-se sempre ao processo que mais recentemente se tornou candidato à respectiva acção. Contudo, é possível solicitar que essas acções sejam feitas sobre outros possíveis processos.
O comando `jobs` permite, apenas para o terminal em que é corrido, observar os processos em segundo plano e parados ("*stopped*", por exemplo como consequência de um `Ctrl+Z`).
[EXC:]
Lançar processos para segundo plano

```
cicloinfinito &
sleep 1d &
sleep 10m &
cicloinfinito
<Ctrl+Z>
sleep 6m
<Ctrl+Z>
```

Correr

```
jobs -l
```

Observar que atrás de cada processo é indicado um índice. É também indicado o estado respectivo.
Fazer

```
bg <indice>
```

para um processo que se queira passar de parado para segundo plano.
Fazer

```
fg <indice>
```

para um processo que se queira passar de parado ou segundo plano para primeiro plano.

11. Configurações

Neste texto abordam-se apenas algumas formas de personalização exequíveis por qualquer utilizador, sem necessidade de direitos de super-utilização. Excluem-se assim, neste âmbito, as configurações ao nível do próprio sistema.

A personalização do ambiente de trabalho é um dos aspectos mais vastos do Unix, não só pela imensidão de funcionalidades disponíveis, mas também porque o factor criatividade facilmente alarga ainda mais essas possibilidades.
Muitas das configurações são definidas ao nível da *shell*. Como as *shells* apresentam diferenças entre si, tanto no que se refere às possibilidades quanto nos aspectos sintácticos, a exploração e explanação deste tema torna-se ainda mais vasta.

A ajuda sobre aspectos das configurações dependentes das *shells* está disponível na *manpage* (e/ou *infopage*) da própria *shell*.

Quando se pretende efectuar uma determinada configuração há que ter em conta a longevidade para ela desejada. Tipicamente, alterações feitas directamente na linha de comando são válidas até à terminação da respectiva sessão e para que uma configuração perdure deve-se colocá-la num ficheiro pré-definido com esse propósito.

11.1. Variáveis

Muitas configurações passam pela definição ou alteração do valor de variáveis que estão relacionadas com as respectivas funcionalidades.

Existem dois tipos de variáveis que importa distinguir:

- variáveis “internas” ou “da *shell*”: definidas ao nível da *shell* e que não são herdadas por processos por ela lançados (por exemplo, outra *shell* que seja gerada para correr um determinado *script*);
- variáveis “externas” ou “de ambiente”: variáveis que a *shell* exporta de modo a que fiquem visíveis para qualquer processo nela lançado.

O modo como a `csh`/`tcsh` e a `sh`/`bash` lidam com estes tipos de variáveis apresenta algumas disparidades entre si.

É típico que os nomes das variáveis da `sh`/`bash` utilizem caracteres maiúsculos, enquanto que na `csh`/`tcsh` o habitual são as minúsculas.
Para as variáveis de ambiente é comum usar maiúsculas.

Na `tcsh` existe um conjunto de variáveis internas que a *shell* mantém sincronizadas com variáveis de ambiente “homónimas” (mesmo nome mas em maiúsculas): `afsuser`, `group`, `home`, `path`, `shlvl`, `term` e `user`. Isso significa que quando a variável de ambiente é alterada a variável interna é actualizada e vice-versa.

11.1.1. Visualização e invocação

Para visualizar o conjunto das variáveis da *shell* pode usar-se o comando `set`.

Para visualizar o conjunto das variáveis de ambiente pode usar-se o comando `env` ou `printenv`. Na `csh`/`tcsh` pode ainda usar-se `sentenv`. Na `bash`, para visualizar o conjunto das variáveis exportadas para o ambiente pode usar-se `export`.

Para utilizar o valor de uma variável preceder o seu nome de '$'.
A invocação de uma variável não definida não se traduz visivelmente num erro e resulta em "". Por esse motivo é importante prestar redobrada atenção à sintaxe dos nomes das variáveis para evitar erros de difícil detecção.

[EXC:] Listar as variáveis definidas.

```
set
env
printenv
```

Na `tcsh`

```
setenv
```

Na `bash`

```
export
```

Imprimir o valor de variáveis referidas pelo respectivo nome (em ambas as *shells*)

```
echo $NAOEXISTE
echo $USER
```

11.1.2. Definição

Existem várias formas de definir uma variável e/ou lhe atribuir um valor:

- variáveis da *shell*, na `bash`:

  ```
  variavel=valor
  read variavel
  ```

  esta última forma é útil, por exemplo, na criação de *scripts* interactivos;

- variáveis da *shell*, na `tcsh`:

  ```
  set variavel = valor
  set variavel = $<
  ```

  esta última forma é útil, por exemplo, na criação de *scripts* interactivos;

- variáveis de ambiente, na `bash`:

  ```
  export variavel=valor
  ```

- variáveis de ambiente, na `tcsh`:

  ```
  setenv variavel valor
  ```

Ter em especial atenção que, em geral, na `bash` não pode usar-se espaços em torno do caracter '='.

[EXC:]

```
#!/bin/bash
read -p "primeiro nome: " PRIMEIRO
read -p "apelido: " APELIDO
echo Nome completo: $PRIMEIRO $APELIDO
```

[EXC:]

```
#!/bin/tcsh
echo -n "primeiro: "
set primeiro = $<
echo -n "apelido: "
set apelido = $<
echo Nome completo: $primeiro $apelido
```

11.1.3. Exportação

A definição de uma variável da *shell* limita o seu âmbito precisamente a essa *shell*. Se for pretendido que os programas corridos nessa *shell* também tenham acesso à definição, é necessário que ela passe para o ambiente.
Na bash isso pode ser feito exportando a variável (para o ambiente).

[EXC:] Na bash

```
EXPERIENCIA=1234
echo $EXPERIENCIA
bash
echo $EXPERIENCIA
exit
export EXPERIENCIA
export OUTRAEXPERIENCIA=5678
bash
echo $EXPERIENCIA
echo $OUTRAEXPERIENCIA
exit
```

Constatar que enquanto a variável não for exportada a sua definição não se torna visível para outros programas (neste caso outra *shell*).

11.1.4. Eliminação

O comando unset pode ser usado para eliminar variáveis definidas. Na tcsh as variáveis de ambiente são eliminadas com unsetenv.

[EXC:] Na bash

```
echo $EXPERIENCIA
unset EXPERIENCIA
echo $EXPERIENCIA
```

11.2. Ficheiros de inicialização

A definição/alteração de uma variável na linha de comando apenas tem validade durante a sessão em curso. Isso significa, por exemplo, que um eventual erro durante a execução da redefinição pode ser resolvido simplesmente abandonando a *shell*; mas também significa que, se for desejado fazer perdurar a definição, existe a necessidade de a colocar num ficheiro de definições.

Esses ficheiros são tipicamente lidos em momentos muito específicos, nomeadamente na inicialização ou finalização das aplicações com que estão relacionados.

É comum que os ficheiros desta natureza possuam nomes iniciados por '`.`' (são *dot files*).

De seguida, referem-se alguns ficheiros que, no que diz respeito às *shells*, interessa conhecer (RHL 9.0). O utilizador pode/deve criar/alterar os ficheiros que é suposto estarem na sua directoria pessoal. Os ficheiros definidos a nível do sistema não são editáveis pelo utilizador comum.

- Para a bash:
  - `/etc/profile`: ficheiro do sistema, corrido no início de uma sessão (quando se faz *login*). Configura ambiente e programas de arranque. Para *shells* de *login* é o primeiro a ser carregado; seguem-se `~/.bash_profile`, `~/.bash_login` e `~/.profile` (por essa ordem e até que um deles exista (os outros são ignorados));
  - `~/.bash_profile`: corrido, se existir, após `/etc/profile`, quando é iniciada uma *shell* de *login*;
  - `~/.bash_login`: corrido, se existir, após `/etc/profile`, se não existir `~/.bash_profile`, quando é iniciada uma *shell* de *login*;
  - `~/.profile`: corrido, se existir, após `/etc/profile`, se não existir `~/.bash_profile` nem `~/.bash_login`, quando é iniciada uma *shell* de *login*;
  - `~/.bashrc`: corrido de cada vez que é lançada uma *shell* interactiva que não seja de *login*. Na realidade, pode ser lido também por solicitação de `~/.bash_profile` (por exemplo) pelo que, assim, acaba também por ser lido na fase de *login*. Não é usado em *shells* não interactivas (por exemplo, as que são usadas para executar *scripts*);
  - `/etc/bashrc`: ficheiro do sistema, contém configurações de funções e *aliases* para todas as *shells*, incluindo as que não sejam de *login* ou que não sejam interactivas. Pode ser chamado por `~/.bashrc`;

  - `~/.bash_logout`: corrido, se existir, quando se faz *logout* (se a *shell* de *login* for a `bash`);

  - `~/.bash_history`: (não é um *script*) mantém a história dos últimos comandos (1000, por omissão, no RHL 9.0) executados na *shell* corrente e nas que já haviam terminado antes do seu início;

- Para a `tcsh`:

  - `/etc/csh.cshrc`: ficheiro do sistema, corrido no início de uma sessão (quando se faz *login*) mas também é usado para todas as outras instâncias, incluindo as que não sejam de *login* ou que não sejam interactivas. No RHL 9.0, durante um *login*, é chamado antes de `/etc/csh.login` mas o contrário também pode acontecer;

  - `/etc/csh.login`: ficheiro do sistema, corrido no início de uma sessão (quando se faz *login*); configura ambiente e programas de arranque;

  - `~/.tcshrc`: primeiro ficheiro do utilizador corrido durante um *login*. Também lido sempre que é lançada uma *shell* (para qualquer instância; mesmo que não seja de *login* ou que não seja interactiva). Se não existir, é procurado `~/.cshrc`;

  - `~/.cshrc`: usado em substituição de ~/`.tcshrc` se esse ficheiro não existir (é ignorado se existir `~/.tcshrc`);

  - `~/.login`: corrido quando a *shell* é de *login*. No RHL 9.0 só é corrido após `~/.tcshrc` (ou `~/.cshrc`) e `~/.history`, mas o contrário também pode acontecer;

  - `/etc/csh.logout`: ficheiro do sistema, corrido, se existir, quando se faz *logout* (se a *shell* de *login* for a `csh`/`tcsh`);

  - `~/.logout`: corre após `/etc/csh.logout` (RHL 9.0).

É importante conhecer os momentos em que os ficheiros de configuração são corridos, de modo a tomar a melhor opção de cada vez que for necessário introduzir alterações.

[EXC:] De modo a ter uma melhor percepção dos momentos em que os vários *scripts* de inicialização são corridos, incluir em cada um deles uma acção que denuncie a sua execução. Sugere-se, na primeira linha,

```
echo "A correr xxxxx..."
```

(em que `xxxxxx` deve ser substituído pelo nome do *script* em causa.)
A modificação dos ficheiros de sistema implica a existência de permissões.

As alterações nos ficheiros de configuração só se tornam efectivas no momento em que eles voltarem a ser corridos, pelo pode ser necessário reiniciar a componente configurada (ou iniciar uma nova instância dela). Por exemplo, tratando-se de uma alteração num ficheiro de configuração de uma *shell*, se for iniciada uma nova *shell* ela já reflectirá as modificações em causa.

[EXC:]

Editar o ficheiro `~/.bashrc` e fazer alguma alteração na mensagem de arranque sugerida no exercício anterior.

Iniciar uma nova *shell* e constatar as alterações na dita mensagem

```
bash
exit
```

A necessidade de iniciar uma nova sessão ou um novo processo de cada vez que se quer testar uma alteração a um ficheiro de configuração pode ser incómoda ou mesmo inconveniente.

A utilização do comando `source` permite tornar efectivas as alterações nesses ficheiros sem forçar a dita reinicialização.

[EXC:]

Voltar a fazer alguma alteração da mensagem em `~/.bashrc`.

Na própria sessão, correr `~/.bashrc` por recurso a `source`

```
source ~/.bashrc
```

[EXC:]

Editar o ficheiro `~/.bashrc` e anexar a linha

```
VAMOSVER="A variável está definida"
```

Na mesma sessão constatar o insucesso de

```
echo $VAMOSVER
```

Lançar uma nova *shell* e constatar que a variável já se encontra disponível (nessa *shell*)

```
bash
echo $VAMOSVER
```

Reparar também que na *shell* anterior a alteração continua sem efeito

```
exit
echo $VAMOSVER
```

Provocar a actualização e testar

```
source ~/.bashrc
echo $VAMOSVER
```

11.3. Personalização

Apresentam-se de seguida, a título de exemplo, algumas configurações relevantes que utilizam as técnicas entretanto referidas.

Quando não indicado explicitamente, assuma-se a utilização da *shell* definida por omissão no Linux (a `bash`).

É importante ter em consideração que as definições propostas estão orientadas para os utilizadores comuns. Deliberadamente, não são aqui mencionadas intervenções ao nível do sistema.

11.3.1. Inclusão da directoria corrente no *path*

Quando se solicita a execução de um comando sem explicitar a sua localização, a *shell* tenta encontrá-lo numa lista de directorias (caminhos) a que se chama "*path*".

O *path* encontra-se definido na variável de ambiente `$PATH`.
[TA:] Na `csh` existe também a variável interna `$path`, que a *shell* mantém sincronizada com a variável de ambiente.
[TA:] O formato usado para exprimir a lista é diferente em cada uma dessas variáveis.
[EXC:] Em cada uma das *shells*, listar as variáveis da *shell* e as de ambiente e localizar a variável do *path*. Reparar na diferença de formatos referida.

Ao contrário do que acontece em DOS/Windows, o comando não é automaticamente procurado na directoria corrente (antes de consultar a dita lista).
Em Unix, para se conseguir um comportamento análogo ao desses sistemas, é necessário que a directoria corrente seja explicitamente referida na lista. É comum que essa não seja a situação pré-definida.
Isso significa que, nesse cenário, para executar um comando localizado na directoria corrente é preciso indicar o caminho para ele. Uma forma simples e comum de o fazer é

```
./xpto
```

(em que `xpto` é um ficheiro executável contido na directoria corrente).

[EXC:] Na linha de comando, acrescentar a directoria corrente ao *path*, como primeira opção de procura.
Na `bash`

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

ou

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

Na `csh`

```
setenv PATH .:$PATH
```

ou

```
set path = (. $path)
```

Observar de novo a definição das variáveis e testar executando um comando contido na directoria corrente

```
xpto
```

[TA:] Na `bash` a exportação da variável indicada, sendo feita na linha de comando, não é de facto necessária, porque nessa altura ela já foi tornada variável de ambiente.
[TA:][EXC:] Na `tcsh` confirmar que é mantida sincronização entre a variável interna e a de ambiente.

[TA:] `$PATH` é uma variável de ambiente. Por esse motivo, se fosse iniciada uma nova *shell* na mesma sessão ("por cima"), a alteração é propagada.
Contudo, a alteração efectuada apenas tem validade na sessão em curso, pelo que não é válida para *shells* em qualquer outra sessão.

[EXC:]
Iniciar uma nova sessão e/ou recorrer a uma outra sessão já existente. Observar a definição da variável nessa sessão.

Para que a modificação se torne efectiva em futuras sessões, pode ser incluída num ficheiro de inicialização da *shell* em causa. Tratando-se de uma variável de ambiente, neste caso será suficiente que a definição seja feita uma vez, no início da sessão (*login*).
[EXC:]
Inserir as alterações ao *path* no ficheiro de inicialização da *shell* referida: para a `bash` sugere-se a utilização de `~/.bash_profile`; para a `tcsh`, sugere-se `~/.login`.
Iniciar uma nova sessão, de modo que o ficheiro de inicialização seja corrido.
Constatar a redefinição da variável.

[EXC:] Tornar efectivas as alterações aos ficheiro de inicialização, na própria sessão corrente, recorrendo ao comando `source`.
Para a `bash`

```
source ~/.bash_profile
```

Para a `tcsh`

```
source ~/.login
```

[TA:] Notar que dada a forma (recursiva) como o *path* é construído com base na sua definição anterior, se estes comandos forem repetidos, vão acumulando referências à directoria corrente.
Neste caso, a anomalia é, em termos práticos, irrelevante. Naturalmente, existiriam formas de contornar esta questão, se isso fosse relevante. A mais simples seria definir o valor da variável de forma estática.

[TA:] Reparar que estando a trabalhar na interface gráfica do Linux, como o *login* é realizado no momento da entrada, se a alteração ao *path* for feita como sugerido acima, mesmo que se abram novas janelas com *shells*, a alteração não se torna visível. Assim, ou se usa o comando `source` em cada uma dessas janelas, ou se volta a entrar na interface para que seja feito um novo *login*.

11.3.2. Alteração do *prompt*

É possível alterar o *prompt* de uma *shell* de modo a que inclua elementos úteis durante a sua utilização.

A especificação do *prompt* é guardada, na forma de uma *string*, numa variável da *shell*.

Na `sh/bash` essa variável é `$PS1`. Na realidade, existem também `$PS2`, `$PS3` e `$PS4` que estão relacionadas com *prompts* utilizados em situações menos comuns.
Na `csh/tcsh` é usada a variável `$prompt` (e analogamente `$prompt2` e `$prompt3`).

A *string* de definição do *prompt* utiliza sequencias especiais de caracteres para permitir a inclusão de diversos elementos que são expandidos no momento da exibição do *prompt*.

Sugestão de alguns elementos com pertinência para inclusão no *prompt*

- identificação do utilizador;
- identificação do terminal;
- identificação da máquina;
- identificação da *shell*;
- índice da linha na história;
- directoria corrente.

Ter em conta que um *prompt* muito longo, embora útil, pode ser visualmente pouco interessante.

[EXC:] Na bash

```
PS1='Faça o favor: '
PS1='\u@\h \s$'
PS1='\! \u@\h:\l \w \s>'
```

[EXC:] Na tcsh

```
set prompt = 'Faça o favor: '
set prompt = '%n@%m tcsh %$shell%#'
set prompt = '%! %n@%m:%l %c tcsh%#'
```

[EXC:] Colocar nos ficheiros de inicialização das respectivas *shells* as configurações desejadas para o *prompt*. Tratando-se da definição de variáveis internas, sugere-se a utilização de ~/.bashrc, para a bash, e ~/.tcshrc (ou ~/.cshrc, consoante o que se tiver optado por usar), para a tcsh.

11.3.3. Definição de *aliases*

A personalização de um conjunto de *aliases* que reflictam os gostos e objectivos de um utilizador podem ser um factor importante em termos do seu desempenho.
Trata-se de um aspecto bastante pessoal e que depende muito do tipo de utilização. Os seguintes exemplos são meramente ilustrativos:

```
h para 'history'
lsla para 'ls -la'
onde para 'find / -name'
```

A sintaxe da definição dos *aliases* já foi objecto de estudo neste documento.

[EXC:] Colocar nos ficheiros de inicialização das respectivas *shells* as definições dos *aliases* desejados. É adequada a utilização de ~/.bashrc, para a bash, e ~/.tcshrc (ou ~/.cshrc, consoante o que se tiver optado por usar), para a tcsh.

[TA:][EXC:] Melhorar o *alias* onde acima sugerido de modo a que não sejam visualizadas as mensagens dirigidas ao *stderr*.
Na bash

```
alias onde='find / -name $* 2> /dev/null'
```

Na tcsh

```
alias onde '( find / -name \!* > /dev/$tty ) >& /dev/null'
```

11.3.4. Configuração do terminal

Embora muitas das definições de terminal estejam vocacionadas para configuração de dispositivos de comunicação desse tipo, existem mesmo assim alguns aspectos que se revestem de utilidade para o tipo de utilização em que se enquadra este documento.

O comando `stty` é usado para este efeito.

É possível conhecer as definições de terminal num determinado instante.
[EXC:]

```
stty -a
stty -a < /dev/tty2
stty size < /dev/tty2
```

(a obtenção de informação relativa a um terminal só é possível se nesse terminal estiver a correr uma sessão sobre a qual o utilizador tenha direitos).

11.3.4.1. Teclas de controlo

É possível redefinir as teclas de controlo. Os valores pré-definidos dos principais controlos são:

- `Ctrl+C` ("^C"): Interrupção (terminação forçada) do processo que se encontra em primeiro plano (envio do sinal `SIGINT` **[***]**);
- `Ctrl+Z` ("^Z"): Paragem (ou "suspensão") do processo que se encontra em primeiro plano (envio do sinal `SIGTSTP` **[***]**);
- `Ctrl+D` ("^D"): Introdução do caracter EOF;
- `Ctrl+S` ("^S"): Pára a saída no terminal ("*stop*");
- `Ctrl+Q` ("^Q"): Retoma a saída no terminal ("*start*");

[EXC:]
Colocar a indicação

```
stty kill 27
```

no ficheiro de inicialização da *shell*, de modo que a tecla <ESC> passe a ser usada para limpar a linha de entrada corrente.

11.3.4.2. Funcionalidades diversas

É possível proceder a uma série de configurações que se podem revelar importantes em determinados tipos de aplicação. Seguem-se alguns exemplos.

[EX:] Fazer com que para retomar a saída no terminal, depois de ter sido parada (com `Ctrl+S`, por omissão), possa ser usada qualquer tecla e não apenas o controlo para isso definido (`Ctrl+Q`, por omissão).

```
stty ixany
```

Desactivar com

```
stty -ixany
```

[EX:] Controlar o eco das teclas premidas.
[EXC:]

```
stty -echo
stty echo
```

[EXC:] Criar e correr o seguinte *script* que exemplifica a introdução de informação sensível que não deve ser visualizada no momento da sua introdução.

```
#!/bin/bash
read -p "username: " USERNAME
stty -echo
read -p "passwd: " PASSWD
stty echo
echo
echo Identificação: $USERNAME $PASSWD
```

[TA:] Neste último exercício, uma alternativa às instruções em estudo seria a utilização do *switch* '`-s`' do comando `read`, que permite precisamente que o texto entrado não seja visualizado.

[EX:] Fazer com que qualquer processo que esteja a correr em segundo plano seja parado (*stopped*) no caso de tentar escrever para o *stdout*.
[EXC:]

```
stty tostop
```

Desactivar com

```
stty -tostop
```